Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій та робототехніки
(повна назва факультету)
Кафедра комп'ютерних та інформаційних технологій і систем
(повна назва кафедри)

Пояснювальна записка
до дипломного проекту (роботи)
магістра
(освітньо-кваліфікаційний рівень)
на тему

Застосування KAN в автоматизованій системі керування роботизованою платформою

Виконав: студент 6 курсу, групи 601-ТН
спеціальності
122 Комп'ютерні науки
(шифр і назва напряму)
Калинович М.С.
(прізвище та ініціали)
Керівник Скакаліна О.В.
(прізвище та ініціали)

Полтава – 2025 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

спеціальність 122 «Комп'ютерні науки»
на тему

«Застосування KAN в автоматизованій системі керування роботизованою платформою»

Студента групи 601-ТН Калиновича Максима Сергійовича

Керівник роботи
кандидат технічних наук,
доцент Скакаліна О.В.

Завідувач кафедри
кандидат фізико-математичних наук,
Двірна О.А.

Полтава – 2025

РЕФЕРАТ

Кваліфікаційна робота магістра: 113 с., 55 рисунків, 9 таблиць, 22 формули, 3 додатки, 20 джерел.

Об’єкт дослідження: застосування KAN для підтримки прийняття рішень в динамічних нелінійних системах керування в реальному часі.

Мета роботи: розробка адаптивної системи керування на основі KAN та порівняння з іншими методами керування.

Методи: проектування та розробка системи керування роботизованою платформою.

Ключові слова: KAN, PID, MLP, теорія керування, роботизована платформа, підтримка прийняття рішень, автоматизовані системи.

ABSTRACT

Master's qualification work: 113 p., 55 figures, 9 tables, 22 formulas, 3 appendices, 20 sources.

Object of research: application of KAN to support decision-making in dynamic nonlinear control systems in real time.

Purpose of work: development of an adaptive control system based on KAN and comparison with other control methods.

Methods: design and development of a control system for a robotic platform.

Keywords: KAN, PID, MLP, control theory, robotic platform, decision-making support, automated systems.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

PID – Proportional Integral Derivative.

MLP – Multilayer Perceptron.

KAN – Kolmogorov-Arnold Network.

ШНМ – Штучні Нейронні Мережі.

GPU – Graphics Processing Unit.

TPU – Tensor Processing Unit.

ООП – Об’єктно-Орієнтоване Програмування.

UAT – Universal Approximation Theorem.

KAT – Kolmogorov-Arnold Theorem.

RMSE – Root Mean Square Error.

MOSFET – Metal-Oxide-Semiconductor Field-Effect Transistor.

ШІМ – Широтно-Імпульсна Модуляція.

АЦП – Аналого-Цифровий Перетворювач.

I2C – Inter-Integrated Circuit.

CMOS – Complementary Metal-Oxide-Semiconductor.

CSI – Camera Serial Interface.

GPIO – General-Purpose Input/Output.

SPI – Serial Peripheral Interface.

UART – Universal Asynchronous Receiver-Transmitter.

DIP – Dual In-line Package.

UML – Unified Modeling Language.

ВСТУП

У сучасному світі автоматизовані системи керування набувають все більшої популярності в усіх сферах життя. Однією з ключових складових таких систем є розв'язання задачі оптимального керування складними об'єктами. Роботизовані платформи використовуються у різноманітних галузях — від автономних транспортних засобів до промислових маніпуляторів та дронів. Однак управління такими платформами вимагає складних обчислювальних рішень, що враховують невизначеність, непередбачувані впливи та взаємодію з оточенням у реальному часі.

В автоматизованих системах керування широко застосовуються класичні методи, такі як пропорційно-інтегрально-диференціальне (PID) регулювання. PID-регулятор є ефективним для управління простими динамічними системами, забезпечуючи точне керування на основі зворотного зв'язку. Однак його ефективність може суттєво знижуватися при роботі з системами, що мають високий рівень невизначеності або складну нелінійну динаміку. Для більш складних нелінійних систем, доречно застосовувати нейронні мережі, зокрема багатошарові перцептрони (MLP). Проте недоліком MLP є їхня залежність від великої кількості параметрів і значних обчислювальних ресурсів для навчання, що може бути проблемою для систем реального часу.

Одним із перспективних методів, що можуть бути застосовані для розв'язання задач керування, є використання мереж Колмогорова-Арнольда (KAN). На відміну від багатошарових перцептронів, KAN здатна ефективно працювати зі складними нелінійними залежностями, зменшуючи вимоги до обчислювальних ресурсів, що робить її важливим напрямком дослідження для використання в роботизованих платформах, де швидкодія та енергетична ефективність є критичними факторами.

Розробка системи на основі KAN дозволить оцінити її переваги у порівнянні з традиційними методами та сприятиме подальшому розвитку теорії та практики інтелектуального керування.

В роботі проводиться детальний аналіз можливостей застосування KAN для автоматизованих систем керування роботизованими платформами. Розроблена система спрямована на вирішення реальних задач, таких як стійке керування рухом, адаптація до зовнішніх умов та ефективне використання апаратних ресурсів. Вивчення KAN у контексті робототехніки дозволяє глибше дослідити її потенціал, порівняти з іншими підходами, такими як PID-регулятори та MLP, і визначити оптимальні сценарії її використання. Розроблено прототип роботизованої платформи, на основі якого проведено перевірки та порівняння різних підходів до автоматизованого керування. В рамках проекту було детально розглянуто кожен етап розробки платформи, починаючи від вибору апаратного забезпечення до інтеграції алгоритмів керування. На кожному етапі проектування було враховано специфічні обмеження, пов’язані з апаратними можливостями платформи, такими як обчислювальна потужність, енергоспоживання, вага та габарити конструкції. Відзначено виклики, що виникли під час інтеграції KAN у систему реального часу, зокрема необхідність оптимізації алгоритмів для забезпечення швидкодії без втрати точності. Проведено серію тестувань, які дозволили визначити критичні моменти в роботі платформи.

Актуальність теми: дослідження перспективного напрямку розробки інтелектуальних систем для підвищення ефективності автоматизованих систем керування, що працюють в складних умовах.

Практична цінність: створення ефективного інструменту для реалізації високоточного керування, який може бути впроваджений у різноманітних робототехнічних системах.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД АВТОМАТИЗОВАНИХ СИСТЕМ КЕРУВАННЯ РОБОТИЗОВАНИМИ ПЛАТФОРМАМИ

1.1 Загальна характеристика проблеми

Задача керування роботизованою платформою є складною та багатогранною, вона передбачає обробку багатьох сенсорних даних для прийняття рішень щодо пересування, маніпуляцій з об'єктами та взаємодії з навколишнім середовищем.

Основними проблемами автоматизованих систем є:

1. **Нелінійність та складна динаміка.** Роботизовані платформи, особливо ті, що функціонують у реальному середовищі (наземні роботи, дрони, маніпулятори), мають складну нелінійну динаміку. Їхнє керування вимагає врахування багатьох факторів, таких як інерція, тертя, гравітація та зовнішні впливи, що ускладнює створення ефективних моделей керування.

2. **Обмеження в реальному часі**. Такі системи вимагають керування в реальному часі. Алгоритми повинні працювати швидко та ефективно, обробляючи великі обсяги даних від сенсорів та приймаючи рішення без затримок. Це створює додаткове навантаження на обчислювальні ресурси і алгоритми управління.

3. **Інтеграція сенсорної інформації.** Автоматизовані системи повинні обробляти дані з різних джерел (камери, лідари, інерційні датчики, ультразвукові сенсори тощо) для точного орієнтування в просторі та побудови моделей оточення. Це вимагає розробки ефективних методів обробки сенсорних даних і їх інтеграції в процес прийняття рішень.

4. **Масштабованість та адаптивність.** Система керування повинна бути гнучкою та адаптуватись до змін, що виникають як всередині платформи, так і в її оточенні. Крім того, важливо забезпечити масштабованість рішень, щоб вони могли бути застосовані для різних типів платформ і задач.

5. **Енергетична ефективність.** Для мобільних платформ, що працюють від акумуляторів, важливо оптимізувати алгоритми керування для мінімізації енергоспоживання, оскільки це безпосередньо впливає на тривалість автономної роботи.

Таким чином, побудова автоматизованої системи керування роботизованими платформами потребує вирішення складних технічних і математичних завдань, які включають розробку адаптивних, швидкодіючих і надійних алгоритмів, здатних ефективно працювати в умовах динамічної невизначеності та реального часу. Традиційні методи керування, такі як класичні пропорційно-інтегрально-диференціальні (PID) регулятори або адаптивні алгоритми на основі спостереження, часто мають обмеження у випадках складних динамічних систем із нелінійностями та затримками.

1.2 Сучасні підходи до автоматизованого керування

Теорія керування є фундаментальною галуззю інженерії та математики для розуміння та розробки складних динамічних систем. Вона забезпечує інструменти для аналізу, моделювання та оптимізації поведінки систем, що мають динаміку і змінні вхідні параметри. Від простих систем керування побутовими пристроями до складних автономних роботів, теорія керування є ключовою для досягнення ефективного і надійного функціонування в реальних умовах.

Основною метою теорії керування є розробка стратегій, що дозволяють впливати на поведінку системи таким чином, щоб вона виконувала бажані завдання або досягала заданих цілей.

Система керування – це набір компонентів або процесів, які взаємодіють для досягнення певної мети. У контексті теорії керування динамічна система описується набором рівнянь, які визначають, як змінюється її стан у часі під впливом зовнішніх або внутрішніх факторів [1].

Вхід і вихід – керовані системи мають вхідні змінні (керуючі впливи) і вихідні змінні (результати або спостережувані параметри). Метою системи керування є підтримання або зміна виходу у відповідь на вхідні сигнали.

Керуючий вплив – це сигнал або дія, що подається на систему для впливу на її стан або вихід. Наприклад, у роботизованій системі керуючим впливом може бути сигнал, який змінює швидкість двигуна або положення маніпулятора.

Зворотний зв’язок – один із ключових принципів теорії керування. Він полягає в тому, що інформація про поточний стан системи (вихідні сигнали) повертається до контролера, що дозволяє коригувати керуючий вплив для досягнення бажаного результату.

Системи керування поділяють на відкриті та замкнені. У відкритих системах керування, вплив здійснюється без урахування поточного стану системи. Тобто керуючий сигнал подається на систему заздалегідь визначеним чином, незалежно від того, який результат був отриманий. Такі системи не мають механізму зворотного зв’язку. Замкнені системи керування використовують зворотний зв’язок для контролю результату і коригування керуючих сигналів. Контролер постійно отримує інформацію про вихід системи і коригує вхід на основі цих даних, щоб зменшити відхилення від бажаного стану.

Також виділяють лінійні та нелінійні системи. Лінійні системи керування описуються лінійними диференціальними рівняннями. Вони передбачають, що керуючий вплив і вихідні сигнали лінійно пов'язані, що спрощує аналіз і проектування систем. Для лінійних систем часто використовуються класичні методи керування, такі як пропорційно-інтегрально-диференціальне (PID) керування. Нелінійні системи керування описуються нелінійними рівняннями, що робить їх поведінку більш складною для аналізу. Для таких систем застосовують методи нелінійного керування, наприклад, лінеаризацію або адаптивні методи, що дозволяють керувати системами з нелінійними характеристиками.

У класичному підході до автоматизованих систем керування використовуються різні типи моделей:

1. **Математичні моделі.** Вони базуються на точних фізичних моделях процесів і об'єктів, для яких розробляються алгоритми керування. Проте ці моделі часто спрощують дійсність, що робить їх менш ефективними для високо динамічних систем.

2. **Нейронні мережі.** Використання штучних нейронних мереж (ШНМ) значно підвищує точність моделювання та прогнозування поведінки складних систем. Проте ШНМ потребують великих обчислювальних ресурсів і великої кількості даних для навчання.

3. **Еволюційні алгоритми.** Цей підхід базується на пошуку оптимальних рішень через еволюційні моделі, проте такі алгоритми можуть бути повільними при роботі в реальному часі.

4. **Методи машинного навчання.** Вони дозволяють системам самостійно навчатися і покращувати свою ефективність з часом, проте їхнє впровадження вимагає великих обсягів даних та точних моделей.

1.2.1. Класичні методи керування. PID-регулятор є одним із найбільш поширених методів керування лінійними системами в промисловості та інженерії. Його основна функція полягає у мінімізації помилки між бажаним станом системи (заданою величиною) та її фактичним станом шляхом корекції керуючого сигналу [2].

Він комбінує три елементи:

- пропорційний компонент (P), який реагує на поточну помилку (різницю між бажаним і фактичним станом);
- інтегральний компонент (I), який враховує сумарну помилку з часом і компенсує невеликі систематичні відхилення;
- диференціальний компонент (D), який оцінює швидкість зміни помилки, допомагаючи уникнути надмірних коливань. Цей компонент дає можливість реагувати на можливі майбутні зміни в системі.

В загальному вигляді, керуючий сигнал визначається наступним чином:

$$f(t) = k_P e(t) + k_I \int_0^t e(t)dt + k_D \frac{d}{dt} e(t) \quad (1.1)$$

де $f(t)$ – значення керуючого сигналу, $e(t)$ – різниця між бажаним та теперішнім станом системи в момент t, $k_P,\ k_I,\ k_D$ – відповідно пропорційний, інтегральний та диференціальний коефіцієнт.

PID-регулятор є ефективним для стабілізації систем і зменшення помилок, проте потребує налаштування параметрів для кожної конкретної системи. В нелінійних динамічних системах ефективність даного алгоритму різко падає.

Переваги PID-регулятора:

1. **Простота реалізації.** PID-регулятори легко зрозуміти та впровадити в систему, що робить їх популярними у промислових процесах. Більшість систем контролю в реальному часі можуть бути налаштовані через PID.

2. **Гнучкість у налаштуванні.** PID-регулятори можна налаштувати відповідно до вимог системи, змінюючи значення коефіцієнтів $k_P,\ k_I,\ k_D$, щоб досягти бажаної поведінки системи.

3. **Реалізація для широкого кола задач.** PID підходить для керування різними типами систем, від простих до складних динамічних систем, таких як температура, швидкість, позиціонування.

4. **Робота в реальному часі.** PID-регулятори швидко реагують на зміни системи, що дозволяє використовувати їх у системах з високими вимогами до точності та швидкості. Наприклад, для стабілізації дрону в повітрі.

Недоліки PID-регулятора:

1. **Проблеми з нелінійними системами.** PID-регулятори можуть бути неефективними для керування складними або нелінійними системами, де динаміка системи змінюється непередбачувано.

2. **Труднощі з налаштуванням.** У складних системах правильний вибір коефіцієнтів може вимагати багато часу і зусиль. Неправильне налаштування призводить до осциляцій, або повільного реагування.

3. **Відсутність адаптивності.** PID-регулятори не можуть адаптуватися до змін умов системи без ручного переналаштування. Це означає, що система керування може втратити ефективність, якщо динаміка системи змінюється з часом.

Оптимальне керування спрямоване на розробку стратегії, що мінімізує або максимізує певну цільову функцію. Це може бути, наприклад, мінімізація енергоспоживання, часу досягнення мети або помилки. До методів оптимального керування належать лінійно-квадратичний регулятор та методи програмування, як-от динамічне програмування.

Робастне керування полягає в побудові системи керування, стійкої до невизначеностей у моделі або зовнішніх впливів. Цей підхід забезпечує надійність системи в реальних умовах, коли модель може бути наближеною або діяти в умовах змінних параметрів середовища.

Адаптивні системи керування здатні змінювати свої параметри в процесі роботи для забезпечення оптимальної реакції на змінні умови. Це важливо для систем, що працюють в умовах невизначеності або мінливих середовищах, таких як динамічні економічні системи.

1.2.2. Нейромережеві методи керування. В складних динамічних системах керування, доречним може бути використання нейромережевих алгоритмів. Наприклад, багатошарові перцептрони (MLP) можуть виконувати важливу роль у таких завданнях, як стабілізація, навігація та розпізнавання об'єктів. MLP представляє собою нейронну мережу, яка складається з вхідного шару, кількох прихованих шарів і вихідного шару. Ця модель належить до класу штучних нейронних мереж і добре підходить для апроксимації складних нелінійних функцій, що робить її ефективною для вирішення багатьох завдань керування. У контексті стабілізації та управління рухом, MLP може використовуватися для прогнозування майбутніх станів системи на основі поточних даних від сенсорів, таких як акселерометри та гіроскопи. Це дозволяє системі виконувати корекції сигналів для підтримання рівноваги роботизованої

платформи. Крім того, MLP можна інтегрувати з традиційними методами управління, наприклад, з PID-контролерами, для кращого управління системами з нелінійною динамікою. У такій комбінації MLP виступає як нелінійний контролер, який оптимізує роботу системи або коригує роботу PID-контролера [3].

Одна з переваг MLP полягає в його здатності до навчання на основі існуючих даних. Мережа може вивчати поведінку системи в різних умовах, коригуючи свої ваги за допомогою алгоритму зворотного поширення помилки, щоб мінімізувати помилки в керуванні. Це дозволяє платформі навчатися самостійно на основі досвіду і підвищувати точність. MLP також може бути використаний для навігації автоматизованих систем. Отримуючи дані від сенсорів, таких як камери або лідари, мережа допомагає системі приймати оптимальні рішення щодо руху та обходу перешкод. Крім цього, MLP дозволяє системам адаптуватися до змінних умов навколишнього середовища або характеристик самої системи, що важливо для забезпечення надійного і стабільного управління в різних ситуаціях. Наприклад, у випадку мобільного робота, MLP може використовувати дані про швидкість, положення коліс та показники від сенсорів, щоб спрогнозувати необхідні корекції для моторів, забезпечуючи стабільність руху і ефективне виконання завдань навігації. Таким чином, MLP додає системам керування гнучкість, адаптивність і можливість ефективно реагувати на зовнішні зміни.

Іншим, відносно новим та перспективним напрямком є мережі Колмогорова-Арнольда (KAN) – це особливим класом мереж, що побудовані на основі теореми Колмогорова про представлення функцій багатьох змінних. Теорема Колмогорова (1957) стверджує, що будь-яку неперервну функцію $f(x_1, x_2, \dots, x_n)$, де x_i є компонентами вектора з простору R^n, можна представити у вигляді суми одновимірних функцій [4]. Це значить, що функція, яка залежить від кількох змінних, може бути перетворена на функції, що залежать від однієї змінної:

$$f(x_1, x_2, \dots, x_n) = \sum_{k=1}^{2n+1} \psi_k \left(\sum_{i=1}^{n} \phi_{ik}(x_i) \right) \quad (1.2)$$

де ψ_k і ϕ_{ik} — це неперервні одновимірні функції. Таким чином, складна багатовимірна функція зводиться до набору більш простих одновимірних функцій, що значно знижує обчислювальну складність задачі. KAN мережі дозволяють ефективно вирішувати задачі високої розмірності за рахунок використання вузлів, кожен з яких відповідає за одну змінну, що робить систему керування більш простою і стійкою до шуму. У контексті керування автоматизованими системами це може означати суттєве спрощення складних керуючих алгоритмів, зокрема для завдань навігації, контролю руху, стабілізації та адаптивного керування. Наприклад, вони можуть допомогти у побудові моделей, які автоматично адаптуються до змінних умов оточення або до нових поведінкових сценаріїв системи. Це особливо важливо для автономних роботів, де швидка адаптація до невідомих середовищ є ключовим завданням. KAN мережі також можуть бути ефективними у завданнях обробки сенсорних даних, що дозволяє автоматизованим системам краще розуміти оточення та приймати рішення на основі багатовимірних вхідних даних.

KAN мережі використовують цю математичну теорему як основу для своєї архітектури. Вони мають багатошарову структуру, де кожен вузол відповідає за обчислення одновимірної функції. Архітектура мережі може бути поділена на два основні блоки:

- шар передобробки (encoding layer). Цей шар обробляє кожен вхідний параметр і застосовує до нього функції $\phi_{ik}(x_i)$. Це відповідає за "попереднє" перетворення вхідних даних, зводячи складність функцій до одновимірних задач;
- шар сумування (summation layer). На цьому етапі всі перетворені одновимірні функції підсумовуються за допомогою функцій ψ_k, що дозволяє відновити складні багатовимірні залежності, але у спрощеній формі.

Така архітектура дозволяє скоротити обчислювальну складність, порівняно з класичними нейронними мережами, оскільки функції від кожної змінної розглядаються окремо.

До переваг KAN можна віднести:

- **зменшення складності.** Завдяки теоремі Колмогорова, багатовимірні задачі можна розкласти на набір одновимірних функцій, що дозволяє знизити кількість параметрів і спростити моделі;
- **гнучкість у моделюванні.** KAN мережі добре підходять для задач, де необхідно працювати з неперервними функціями великої розмірності, оскільки дозволяють зберігати високий рівень точності з меншою обчислювальною складністю;
- **апроксимація нелінійних функцій.** Вони можуть ефективно апроксимувати нелінійні функції, що робить їх корисними в задачах з нелінійною динамікою, таких як керування роботизованими системами або складними фізичними процесами;
- **стійкість до шуму.** Оскільки система зводиться до одновимірних функцій, вона може бути стійкішою до шуму у вхідних даних, порівняно з класичними нейронними мережами, що спрощує задачу контролю та керування в реальному часі.

Однак, на даний момент є певні виклики та обмеження в реалізації KAN:

- **вибір функцій.** Однією з основних проблем у застосуванні KAN мереж є вибір конкретних функцій ψ_k і ϕ_{ik}, які використовуються для апроксимації. Це вимагає ретельного навчання, що може бути складним у деяких прикладних задачах;
- **масштабованість.** Хоча KAN мережі значно скорочують кількість параметрів порівняно з класичними нейронними мережами, їх застосування для дуже великих систем може потребувати модифікацій або оптимізації, особливо якщо кількість змінних значно зростає;

- **час навчання.** Оскільки в KAN апроксимуються не ваги між вузлами, а самі функції на вузлах, це вимагає значно більше обчислювальних ресурсів та часу на етапі навчання, в порівнянні з MLP.

1.2.3. Порівняння KAN з іншими методами керування. Мережі Колмогорова-Арнольда (KAN) представляють інший підхід до керування, використовуючи нейронні моделі для апроксимації складних багатовимірних функцій і адаптації до динаміки системи. Нижче наведено порівняння між PID-регуляторами та KAN.

Переваги KAN перед PID:

1. Адаптивність. KAN мережі можуть самостійно адаптуватися до змін динаміки системи, в той час як PID-регулятори потребують переналаштування для адаптації до нових умов. Це робить KAN більш гнучким підходом для систем із непередбачуваною динамікою або у випадках, коли параметри змінюються в процесі роботи.

2. Моделювання складних функцій. KAN дозволяють ефективно працювати з нелінійними системами та багатовимірними вхідними даними, в той час як PID-регулятори оптимальні для лінійних або простих систем. KAN може набагато точніше моделювати складні системи, що мають нелінійну поведінку.

3. Автоматизація процесу навчання. KAN використовують принципи навчання на основі вхідних та вихідних даних, що дозволяє системі автоматично вдосконалювати своє керування без необхідності в ручному втручанні.

Переваги PID перед KAN:

1. **Простота і зрозумілість.** PID-регулятори легко налаштовуються і зрозумілі для широкого кола інженерів. Їхня математична основа проста, що дозволяє швидко застосовувати їх на практиці.

2. **Мала обчислювальна складність.** PID-регулятори мають низькі вимоги до обчислювальних ресурсів. KAN, особливо при роботі з великими системами або складними функціями, можуть вимагати значно більше ресурсів для навчання та виконання.

3. **Широка підтримка в індустрії.** PID-регулятори широко використовуються в промислових системах і автоматизації, оскільки для них доступні готові рішення та інструменти для налаштування і впровадження. KAN є менш поширеними і вимагають більше спеціалізованих знань для розробки і налаштування.

4. **Відсутність потреби в навчанні.** PID-регулятори не потребують навчання або великих обсягів даних для налаштування, на відміну від KAN, які вимагають тренування для точного моделювання системи.

Мережі Колмогорова-Арнольда (KAN) та багатошарові перцептрони (MLP) обидва є нейромережевими методами, тому їх порівняння є більш цікавим. Розглянемо основні переваги KAN над MLP:

1. Неоптимальність у представленні функцій:

- **KAN.** KAN можуть представити будь-яку функцію з відносно невеликими обчислювальними витратами та кількістю параметрів;
- **MLP.** Для представлення складних функцій MLP часто вимагають великої кількості шарів та нейронів, що ускладнює модель. Чим складніша функція, тим більше прихованих шарів і нейронів потрібні, що може призвести до значного збільшення обчислювальних ресурсів та часу навчання.

2. Проблема з вибором архітектури:

- **KAN.** Мережі Колмогорова-Арнольда мають теоретично визначену структуру, яка дозволяє точно апроксимувати будь-яку функцію за допомогою фіксованої кількості функцій однієї змінної. Це дає змогу легко визначити кількість елементів у мережі;
- **MLP.** Вибір оптимальної архітектури для MLP (кількість шарів, кількість нейронів у кожному шарі) є значно більшою проблемою. Це зазвичай вимагає експериментального підходу або методу пошуку оптимальних гіперпараметрів, що може значно ускладнити процес розробки моделі.

3. Надмірна складність навчання:

- **KAN.** Завдяки одновимірній апроксимації, KAN мають менші вимоги до навчання та швидше адаптуються до нових умов. Мережі KAN можуть

ефективніше використовувати ресурси для представлення функцій без надмірного навантаження;

- **MLP.** Навчання MLP може бути значно складнішим через великий обсяг параметрів та необхідність їх оптимізації за допомогою методів градієнтного спуску. Це може призвести до проблем збіжності, особливо при навчанні глибоких мереж.

4. Проблема з локальними мінімумами:

- **KAN.** Мережі KAN базуються на математичній теорії, яка дає більш точне представлення функцій без необхідності вирішення задач глобальної оптимізації у великому просторі параметрів. Це дозволяє уникнути багатьох проблем із потраплянням у локальні мінімуми під час навчання;
- **MLP.** MLP часто стикаються з проблемами локальних мінімумів або плато під час навчання, особливо у випадках, коли функція втрат має складний ландшафт. Це може зробити процес навчання неефективним і тривалим.

5. Погана інтерпретованість:

- **KAN.** Мережі KAN використовують явні функції для кожного параметра, що дозволяє чітко розуміти, як зміна вхідних даних впливає на вихід мережі. Це робить їх інтерпретацію простішою у порівнянні з нейронними мережами загального призначення;
- **MLP.** MLP є "чорним ящиком", що означає, що навіть при ефективному навчанні зрозуміти, як саме нейронна мережа ухвалює рішення, може бути складно. Це є одним із ключових недоліків у задачах, де інтерпретованість є важливою.

6. Чутливість до параметрів навчання:

- **KAN.** KAN використовують більш детерміновану структуру для апроксимації функцій, що робить їх менш чутливими до параметрів навчання. Параметри мереж KAN не вимагають складного підбору і можуть бути досить стійкими до змін;
- **MLP.** MLP дуже чутливі до параметрів навчання, таких як швидкість навчання, вибір ініціалізації ваг, кількість епох тощо. Це може спричинити

нестабільність під час тренування і вимагати точного налаштування для досягнення бажаного результату.

7. Збільшені вимоги до даних:

- **KAN.** Оскільки KAN можуть апроксимувати функції однієї змінної і комбінувати їх для створення складніших моделей, вони ефективніше працюють із меншими наборами даних;
- **MLP.** MLP зазвичай потребують великих обсягів даних для ефективного навчання, особливо коли йдеться про складні або глибокі мережі. Недостатність даних може призвести до перенавчання або недостатньої узагальнюваності моделі.

Здійснено огляд сучасних підходів до автоматизованих систем керування роботизованими платформами, а також висвітлено основні проблеми, з якими стикаються розробники таких систем. Мережі Колмогорова-Арнольда були представлені як перспективний підхід для вирішення задач адаптивного керування, який дозволяє знизити обчислювальну складність і забезпечити високу гнучкість у змінних умовах.

1.3. Огляд існуючих рішень

Побудова автоматизованої системи керування роботизованою платформою є комплексною апаратно-програмною задачею. Як правило, такі системи є вузькоспеціалізованими і розробляються під конкретні вимоги бізнесу, чи промисловості.

Побудова автоматизованої системи керування роботизованою платформою є комплексною апаратно-програмною задачею, що поєднує в собі інженерні, математичні та комп'ютерні науки. Як правило, такі системи є вузькоспеціалізованими і зазвичай розробляються для вирішення конкретних завдань, що виникають у бізнесі або промисловості, таких як автоматизація виробничих процесів, логістика, автономне управління транспортними засобами

чи виконання специфічних завдань у важкодоступних середовищах, або небезпечних для людини умовах.

Основними викликами при побудові таких систем є:

1. **Визначення вимог і постановка задачі.** Системи керування можуть відрізнятися залежно від завдань. Наприклад, автономні роботи для логістики мають зовсім інші вимоги до навігації, ніж платформи для обслуговування небезпечних середовищ. Важливо правильно визначити мету, якої система має досягти, і умови, у яких вона функціонуватиме.

2. **Апаратне забезпечення.** Розробка роботизованої платформи вимагає вибору відповідних сенсорів (лідарів, камер, датчиків руху), актуаторів (електромоторів, гідравліки тощо), а також контролерів, які забезпечать ефективне управління. Важливим аспектом є інтеграція всіх апаратних компонентів з мінімальними затримками і високою надійністю.

3. **Програмне забезпечення та алгоритми.** Задачі автоматизованого керування включають розробку алгоритмів керування рухом і поведінкою робота. Тут можуть застосовуватися класичні методи, такі як PID-регулювання, для стабілізації руху і корекції траєкторій, а також сучасні підходи з використанням штучних нейронних мереж (наприклад, MLP або KAN-мережі), що дозволяють здійснювати адаптивне управління в умовах невизначеності або зміни середовища.

4. **Реакція на зміни середовища.** Автономні роботизовані платформи повинні мати здатність до самостійного аналізу оточення і прийняття рішень у реальному часі. Це може вимагати побудови складних моделей навколишнього середовища та алгоритмів планування траєкторій, що дозволяють обробляти дані з численних сенсорів, використовуючи методи обробки зображень, машинного навчання або системи підтримки прийняття рішень.

5. **Тестування та оптимізація.** Побудова автоматизованої системи керування потребує постійного тестування в реальних умовах або симуляційних середовищах. Це дозволяє оптимізувати роботу платформи, виявляти недоліки та покращувати точність та ефективність алгоритмів керування.

Існує ряд компаній, що займаються виготовленням роботизованих систем на замовлення, або для потреб конкретної компанії.

Robotic Automation Systems розпочала свою діяльність як Geiger Handling USA у 1994 році як інтегратор автоматизації промисловості пластмас і північноамериканський дистриб'ютор якісних швейцарських роботів Geiger. Вони почали надавати прості рішення для машин лиття пластмас під тиском із використанням 3-осьових пневматичних систем і незабаром перейшли до сервороботів та іншої роботизованої автоматизації. Компанія стала лідером повного спектру послуг автоматизації для промисловості пластмас, включаючи горизонтальне та вертикальне лиття пластмаси під тиском, лиття вставок, накладне формування, декорування в формі і маркування в формі та видувне лиття [5].

Robotic Automation Systems об'єднує широкий спектр роботів від провідних виробників роботів, що дозволяє вибирати та інтегрувати найкращих роботів для конкретних застосувань – 3-осьові роботи з верхнім входом, 6-осьові роботи, колаборативні роботи, 4-осьові роботи SCARA від таких виробників, як WEMO Robots, ABB Robotics, Epson Robots і FANUC Robotics.

Іншим прикладом є Amazon Robotics LLC, раніше Kiva Systems, це компанія зі штату Массачусетс, яка виробляє мобільні роботизовані системи. Це дочірня компанія Amazon.com. Його автоматизовані системи зберігання та пошуку використовувалися в минулому такими компаніями, як The Gap, Walgreens, Staples, Gilt Groupe, Office Depot, Crate & Barrel, Amazon і Saks 5th Avenue. Співробітники Erstwhile Kiva тепер працюють лише на складах Amazon. Після роботи в групі бізнес-процесів у Webvan Мік Маунц дійшов висновку, що падіння компанії сталося через негнучкість існуючих систем обробки матеріалів і високу вартість виконання замовлень. Ці виклики надихнули Маунца на розробку методу комплектування, пакування та доставки замовлень за допомогою системи, яка могла б доставити будь-який товар будь-якому оператору в будь-який час [6].

Традиційно товари переміщуються навколо розподільчого центру за допомогою конвеєрної системи або машин, якими керує людина. У підході Kiva, предмети зберігаються в переносних блоках зберігання. Коли замовлення вводиться в систему бази даних Kiva, програмне забезпечення знаходить найближчий до товару автоматичний транспортний засіб і спрямовує його забрати. Мобільні роботи пересуваються по складу, слідкуючи за рядом комп'ютеризованих наклейок зі штрих-кодом на підлозі. Кожен приводний блок має датчик, який запобігає його зіткненню з іншими. Коли привід досягає цільового місця, він заїжає під контейнер і піднімає його за допомогою штопора. Потім робот несе капсулу до вказаної людини-оператора, щоб той забрав предмети. Kiva продавала системи на основі двох різних моделей роботів. Менша модель мала розміри приблизно 0,61 на 0,76 м, висоту 460 мм і здатна піднімати 450 кг. Більша модель була здатна перевозити піддон із вантажем вагою до 1400 кг. Обидва були характерного оранжевого кольору. Максимальна швидкість роботів становила 1,3 метра в секунду. Мобільні роботи працювали від акумулятора і потребували заряджання щогодини протягом п'яти хвилин.

Система вважається набагато ефективнішою та точнішою, ніж традиційний метод, коли працівники переміщуються складом, знаходячи та збираючи товари. Робот також дозволяє оптимізувати переміщення вантажів Amazon і допомагає в забезпеченні безпеки як персоналу, так і самих посилок.

Виклики сучасної війни породжують новий напрямок розробки роботизованих платформ для використання як у військових, так і гуманітарних цілях, наприклад для евакуації поранених.

TEMERLAND – унікальний проект української компанії з розробки безпілотних роботизованих комплексів та платформ, модернізації транспортних засобів в безпілотні наземні комплекси. Інноваційні безпілотні роботизовані комплекси TEMERLAND покликані рятувати і захищати людські життя, а також бути ефективним інструментом військових дій – розвідки, патрулювання, наступу та оборони [7].

Відносно молода компанія вже пропонує широку номенклатуру різноманітних роботизованих платформ для виконання різних задач, як наприклад:

Безпілотна роботизована платформа-роботоносець Скорпіон 2

Скорпіон 2 являє собою багатофункціональну роботизовану платформу з наступним функціоналом:

- роботоносець для роботів – сателітів;
- дрономет – точка базування для літальних апаратів;
- платформа для установки автоматичної турелі;
- трактор для обробки сільськогосподарських угідь.

Скорпіон 2 оснащений ліфтами для чотирьох роботів – сателітів. Сателіти здійснюють функції моніторингу та репітеру для збільшення дальності зв'язку між оператором та материнською безпілотною роботизованою платформою. Ліфти для сателітів кріпляться на задній частині платформи і виконують функцію опускання/підйому сателітів з платформи на поверхню землі. Ліфти управляються дистанційно з переносного командного пульта управління.

Безпілотна платформа для логістики ТУРАН

Безпілотна розробка ТУРАН є багатофункціональною роботизованою платформою для логістики з наступним функціоналом:

- перевезення логістичних вантажів;
- доставка їжі та боєприпасів;
- транспортування поранених.

Логістика

Роботизована платформа TURAN для транспортування логістичних піддонів стандарту НАТО розміром 1200 х 1000 мм та масою до 500 кг.

Доставка їжі та боєприпасів

Доставка продовольства та боєприпасів у бою з використанням безпілотних платформ TURAN значно знижує ризик втрати військовослужбовців. Невеликі розміри та низький рівень шуму під час руху платформи спрощують виконання прихованих бойових завдань.

Транспортування поранених

Евакуація поранених із поля бою за допомогою платформ дозволяє знизити втрати серед військово-медичного персоналу.

Безпілотна роботизована платформа ГНОМ

Безпілотна розробка ГНОМ представляє собою багатофункціональну роботизовану платформу для вирішення наступних задач:

- робот для ведення спостереження й розвідки;
- доставка боєприпасів, продовольства й евакуація поранених;
- репітер для збільшення дальності радіозв’язку й управління;
- робот – сателіт для більших роботизованих платформ.

Спостереження й розвідка

Завдяки невеликим габаритам й майже беззвучному переміщенню, ГНОМ може вести приховане спостереження за допомогою кругової оглядової камери на телескопічній щоглі. Система зв’язку і запас енергії дозволяють вести розвідку й спостереження віддалено до 5 км.

Доставка й транспортування

Оптимальні габарити й вантажність дозволяють виконувати майже непомітну доставку боєприпасів й продовольства в умовах ведення бою, без ризику для особового складу. Роботизована платформа ГНОМ може транспортувати поранених за допомогою спеціального транспортного візка.

Збільшення дальності радіозв'язку й управління безпілотних систем

ГНОМ може приймати радіосигнал, посилювати й передавати далі, виконуючи функцію репітера. Застосовуючи декількох ГНОМів ця функція є надзвичайно корисною, бо дозволяє в умовах ведення бою розгорнути власну мережу для координації дій підрозділів й роботи безпілотних систем.

Наземна роботизована платформа SLMNDR

НРП SLMNDR, є легким, малопомітним універсальним комплексом, який швидко трансформується з логістичного засобу у протимінний трал або в елемент штурмового порядку піхоти, здатний працювати під впливом засобів РЕБ, при цьому дальність керування складає 150-200 м.

Унікальним концептуальним рішенням є застосування колісної формули 4X4 з незалежним керуванням кожним з 4-х моторних коліс та шарнірне зчленування двох незалежних осей, що значно підвищує прохідність і керованість у важких умовах та надає можливість завершити місію маючи лише одну працюючу вісь.

Використання платформи дозволить:

- підвищити ступінь безконтактності ведення бойових дій і, як наслідок, знизити рівень втрат через можливість дистанційного керування;
- зменшити ступінь участі традиційної бойової техніки у виконанні бойових завдань.

НРП SLMNDR притаманні малопомітність, висока прохідність та відносно невелика вага, простота у використанні, обслуговуванні та відповідному навчанні, придатність до швидкого масштабування виробництва та внесення покращень без значних змін у технологічному процесі.

В цьому розділі було розглянуто теорію керування в контексті автоматизованих систем, основні задачі та проблеми. Описано класичні методи керування, такі як PID, та нейромережеві, на прикладі MLP та KAN. Хоча класичні підходи, такі як PID-регулятори, залишаються актуальними для простих динамічних систем, їх можливості обмежуються в ситуаціях з високою складністю та нелінійністю. Нейронні мережі, такі як багатошарові перцептрони, можуть покращити управління такими системами, проте вимагають значних ресурсів для навчання. У цьому контексті використання мереж Колмогорова-Арнольда відкриває нові перспективи для оптимального керування складними системами з меншою кількістю параметрів і зниженими вимогами до обчислювальних ресурсів. Це робить KAN ефективним інструментом для забезпечення адаптивності, стійкості та швидкості прийняття рішень в автоматизованих системах керування, особливо у випадках роботи в реальному часі.

РОЗДІЛ 2
ПОСТАНОВКА ЗАВДАННЯ

Розробка автоматизованої системи керування роботизованою платформою на основі KAN потребує ретельного планування як апаратної частини, так і програмного забезпечення. Ось основні вимоги до такої системи, враховуючи специфіку застосування KAN для керування роботизованою платформою:

1. Функціональні вимоги:

- **адаптивне керування.** Роботизована платформа повинна мати здатність адаптуватися до нових умов та навчатися на основі даних, отриманих в реальному часі. Завдяки KAN, система має навчатися в режимі онлайн, покращуючи свої дії відповідно до змін в середовищі або умовах роботи;
- **сенсорна інтеграція та обробка даних.** Система повинна бути здатна інтегрувати декілька сенсорів та на основі їх даних приймати рішення в режимі реального часу. KAN забезпечить високий рівень обробки вхідних даних, що дозволить створити комплексну модель навколишнього середовища.

2. Продуктивність та швидкодія:

- **низькі затримки.** Оскільки система керування працюватиме в реальному часі, необхідно забезпечити мінімальні затримки між обробкою сенсорних даних, навчанням та діями роботизованої платформи. Це може вимагати оптимізації KAN для роботи на апаратному рівні з використанням GPU для пришвидшення обчислень;
- **масштабованість.** KAN система повинна бути масштабованою, щоб підтримувати більш складні завдання чи більшу кількість сенсорів і каналів даних без втрати швидкодії.

3. Надійність та стабільність:

- **шум та навчання в реальному часі.** Система має вміти працювати в умовах, коли деякі сенсори можуть давати зашумлені або некоректні дані. Завдяки KAN, платформа повинна адаптуватися та компенсувати можливі похибки через самонавчання;

- **відмовостійкість.** Важливо, щоб система мала механізми автоматичного відновлення у випадку помилок або апаратних відмов.

4. Апаратні вимоги:

- **обчислювальна потужність.** Роботизована платформа повинна мати достатню кількість обчислювальних ресурсів для використання KAN мережі в реальному часі. Це може включати багатоядерні процесори або спеціалізовані прискорювачі для нейронних мереж, такі як графічні процесори (GPU) або тензорні процесори (TPU). При цьому, рішення має бути компактним, що вимагає використання мікроконтролерів, або одноплатних комп'ютерів;
- **потужні сенсори.** Для навчання KAN потрібна велика кількість якісних даних, тому платформа повинна мати набір точних сенсорів для збору даних з довкілля;
- **автономне живлення.** Система має працювати в автономному режимі, тому важливо передбачити ефективні енергозберігаючі режими для обчислювальних і сенсорних компонентів, а також надійне джерело живлення, наприклад, батарею з великою ємністю.

5. Програмні вимоги:

- **реалізація KAN.** Необхідно розробити або адаптувати бібліотеки для ефективної реалізації KAN. Програмне забезпечення має включати компоненти для навчання і корекції на основі вхідних даних. Оптимізація алгоритмів під конкретну платформу, на якій працюватиме робот, є ключовим фактором;
- **алгоритми навігації та уникнення перешкод.** KAN повинна вміти швидко адаптувати маршрути на основі отриманих даних про перешкоди. Для цього важливо інтегрувати алгоритми планування шляху та уникнення перешкод, що коригуватимуться в режимі реального часу.

6. Безпека. Функція аварійної зупинки. В разі відмови програмного забезпечення або небезпечної ситуації система повинна мати механізми для швидкої і безпечної зупинки або переходу в аварійний режим.

Вхідна інформація: дані з сенсорів платформи.

Вихідна інформація: чіткі команди на елементи керування платформою.

Необхідно розробити простий інтерфейс для взаємодії з системою, зміни її налаштувань, а також моніторингу стану. Такий інтерфейс має підтримувати будь-який (або декілька) протокол бездротової передачі даних, наприклад Wi-Fi або Bluetooth. Розроблювана система повинна бути оптимізована для роботи у вбудованих системах з обмеженими обчислювальними ресурсами та підтримувати роботу з такими операційними системами, як: Windows, Linux.

Апаратна частина проекту має бути побудована на основі доступних компонентів, що дозволить значно зменшити вартість розробки і забезпечити простоту повторення проекту. Роботизована платформа повинна бути малогабаритною, автономною та енергоефективною для тривалого використання в приміщенні. Апаратно, система повинна бути побудована за модульним принципом, що дозволить легко замінювати чи оновлювати окремі компоненти без необхідності переробки всієї системи, та включати ряд основних сенсорів для орієнтації в просторі та уникнення перешкод.

В розділі було викладено вимоги до програмної та апаратної частини розроблюваної системи. Особливу увагу варто приділити вибору недорогих та малогабаритних комплектуючих, що дозволяє використання платформи у внутрішніх приміщеннях. Було визначено, що апаратна частина має включати мікроконтролери або одноплатні комп’ютери з обмеженими ресурсами та низьким енергоспоживанням, автономні модулі живлення, базовий набір сенсорів для навігації, та двигуни з редукторами для керування рухом. Система має бути модульною, що дозволяє легко оновлювати або замінювати її елементи. Також, важливим елементом є простий інтерфейс для дистанційного моніторингу стану системи та аварійної зупинки.

РОЗДІЛ 3
ПРОЕКТНІ РІШЕННЯ

3.1 Програмні рішення

Відповідно до вимог, викладених у попередньому розділі, для розробки програмного забезпечення автоматизованої системи керування, було обрано мову програмування Python. Це високорівнева мова програмування загального призначення, яка поєднує в собі простоту синтаксису та потужні можливості. Вона була створена Гвідо ван Россумом і вперше представлена в 1991 році. Python орієнтований на підвищення продуктивності програміста, підтримує декілька парадигм програмування та широко використовується у багатьох галузях [8].

Основні характеристики Python:

1. **Простий і зрозумілий синтаксис.** Python легко вивчати навіть новачкам завдяки читабельному коду, що нагадує природну мову.

2. **Інтерпретована мова.** Програми Python виконуються без попередньої компіляції, що дозволяє швидко розробляти, тестувати та налагоджувати код.

3. **Типізація.** Python – мова з динамічною, сильною та неявною типізацією. Це значить, що інтерпретатор автоматично визначає тип даних змінної, без необхідності явного його вказання, що зменшує кількість коду, але вимагає уважного підходу до написання програм. Також, внаслідок сильної типізація, не допускаються операції з різними типами даних в одному виразі, без явного їх приведення до одного типу.

4. **Підтримка різних парадигм програмування:**

- об’єктно-орієнтоване програмування (ООП);
- функціональне програмування;
- імперативне програмування;
- процедурне програмування.

5. **Кросплатформеність.** Python працює на більшості сучасних операційних системах: Windows, macOS, Linux, Android.

6. **Широкий вибір бібліотек та фреймворків.** Python має багатий стандартний набір модулів і величезну кількість сторонніх бібліотек для різних сфер:

- наука і обчислення: NumPy, SciPy, pandas;
- штучний інтелект: TensorFlow, PyTorch, scikit-learn;
- веб-розробка: Django, Flask;
- автоматизація: Selenium, PyAutoGUI;
- графіка: Matplotlib, Plotly.

7. **Відкрите програмне забезпечення.** Python є проектом з відкритим кодом, що сприяє розвитку великої спільноти розробників.

8. **Інтерактивність.** Python підтримує інтерактивний режим роботи (REPL), що дозволяє виконувати код по рядках і миттєво отримувати результат.

9. **Вбудовуваність та розширюваність.** Python можна вбудовувати в інші програми, а також використовувати C/C++ для створення високопродуктивних модулів.

10. **Популярність та підтримка спільноти.** Python – одна з найпопулярніших мов у світі, і її спільнота активно підтримує новачків та розробляє нові інструменти.

Зазначені вище переваги дозволяють використовувати Python в різних сферах науки та повсякденного життя, а саме: аналіз даних, машинне навчання, статистика, створення серверів і веб-застосунків, автоматизація задач, розробка ігор, робототехніка, моделювання, прогнозування даних тощо.

Для розробки тестового середовища з елементами візуалізації було застосовано бібліотеку Pygame, яка є високорівневим програмним засобом для побудови мультимедійних застосунків та 2D-ігор. Вона забезпечує інтегровані інструменти для обробки графічних зображень, анімації, аудіо та користувацьких подій [9]. Завдяки своїй модульній архітектурі, Pygame дозволяє

ефективно реалізовувати інтерактивні середовища, орієнтовані на обробку подій і виведення графічних елементів у реальному часі.

Робота з конфігураційним файлом симуляційного середовища здійснена з використанням бібліотеки PyYAML, яка забезпечує ефективний інструментарій для парсингу, серіалізації та десеріалізації даних у форматі YAML. Цей формат є зручним для структурованого подання конфігурацій завдяки його читабельності та здатності зберігати вкладені структури, такі як словники та списки. Використання PyYAML дозволяє автоматизувати завантаження параметрів симуляції, забезпечуючи гнучкість і зручність їх налаштування без необхідності зміни основного коду програми [10].

3.2 Вибір алгоритму керування

Багатошарові перцептрони (MLP) засновані на універсальній теоремі про наближення (UAT). Коротко, ця теорема стверджує, що нейронна мережа лише з одним прихованим шаром, що містить кінцеву кількість нейронів, може апроксимувати будь-яку неперервну функцію з достатньою точністю на компактній підмножині R^n, з використанням відповідної функції активації.

Математично, для будь-якої неперервної функції f та $\in > 0$, завжди існує нейронна мережа $\hat{f}$ така, що:

$$\left|f(x) - \hat{f}(x)\right| < \in \tag{3.1}$$

В даній роботі розглядається теорема Колмогорова-Арнольда про представлення, яка може бути реалізована за допомогою нового типу нейронних мереж. Володимир Арнольд і Андрій Колмогоров встановили, що якщо f є багатовимірною неперервною функцією на обмеженій області, то f можна записати як скінченну композицію неперервних функцій однієї змінної та двійкової операції додавання. Точніше, для гладкого $f: [0,1]^n \to R$,

$$f(x) = f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right), \qquad (3.2)$$

де $\phi_{q,p}: [0,1] \to R$ and $\Phi_q: R \to R$. У певному сенсі вони показали, що справжня багатовимірна функція є додаванням, оскільки будь-яка інша функція може бути записана за допомогою одновимірних функцій і суми.

Може здатися, що це чудова новина для машинного навчання: вивчення функції великої розмірності зводиться до вивчення поліноміального числа одновимірних функцій. Однак ці функції можуть бути негладкими і навіть фрактальними, тому їх не можна вивчити на практиці. Через це, теорема представлення Колмогорова-Арнольда була знецінена в машинному навчанні як така, що теоретично обґрунтована, але практично незастосовна [11].

Однак, розглянемо підхід, при якому теорема Колмогорова-Арнольда може бути корисною для машинного навчання. По-перше, не потрібно дотримуватися оригінального рівняння (3.2), яке має тільки двошарові нелінійності і невелику кількість термів (2n+1) у прихованому шарі: мережу буде узагальнено на довільну ширину і глибину. По-друге, більшість функцій у науці та повсякденному житті часто є гладкими і мають розріджені композиційні структури, що потенційно сприяє гладким представленням Колмогорова-Арнольда.

Нехай, маємо задачу навчання з вчителем, яка складається з пар вхід-вихід {xi, yi}, де необхідно знайти f таке, що $y_i \approx f(x_i)$ для всіх точок даних. Відповідно до рівняння (3.2), задача вважатиметься виконаною, якщо будуть відомі відповідні одновимірні функції $\phi_{q,p}$ та Φ_q. Оскільки всі функції, які потрібно вивчити, є одновимірними, можна параметризувати кожну одновимірну функцію як криву B-сплайну з коефіцієнтами локального базису B-сплайна, які можна вивчати. Отже, є прототип KAN мережі, чий обчислювальний графік точно задано рівнянням (3.2) і проілюстровано на рисунку 3.1 (з вхідною розмірністю n = 2), що виглядає як двошарова нейронна мережа з функціями

активації, розміщеними на ребрах замість вузлів (проста сума виконується на вузлах), і з шириною 2n + 1 у середньому шарі.

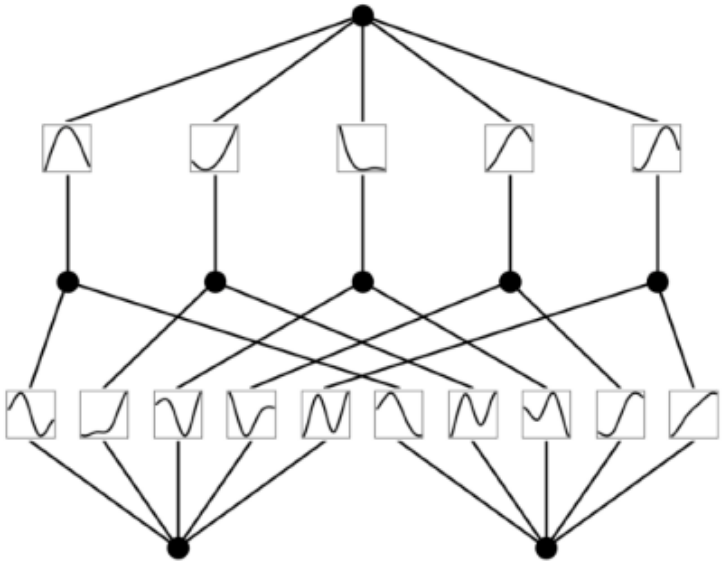

Рисунок 3.1 – Архітектура KAN мережі

Як уже згадувалося вище, така мережа надто проста, щоб апроксимувати довільну функцію на достатньому рівні на практиці з гладкими сплайнами. Тому, варто узагальнити KAN мережу, щоб вона була ширше та глибше. Для цього варто провести аналогію між MLP і KAN. В MLP, після визначення шару (який складається з лінійного перетворення та нелінійності), можна додавати більше шарів, щоб зробити мережу глибшою. Для побудови глибоких KAN мереж, необхідно дати визначення шару KAN. Виявилося, що шар KAN з n_{in}-розмірним входом та n_{out}-розмірним виходом можна визначити як матрицю одновимірних функцій

$$\Phi = \{\phi_{q,p}\}, \qquad p = 1,2,\dots,n_{in}, \qquad q = 1,2,\dots,n_{out} \tag{3.3}$$

де функції $\phi_{q,p}$ мають параметри, які можна натренувати. У теоремі Колмогова-Арнольда внутрішні функції утворюють шар KAN з $n_{in} = n$ та $n_{out} = 2n + 1$, а зовнішні функції утворюють шар KAN з $n_{in} = 2n + 1$ та $n_{out} = 1$. Отже, представлення Колмогорова-Арнольда в рівнянні (3.2) є просто

композиціями двох шарів KAN. Звідси, узагальнена форма KAN мереж виглядає наступним чином:

$$f(x) = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1,i_L,i_{L-1}} \left(\sum_{i_{L-2}=1}^{n_{L-2}} \dots \left(\sum_{i_2=1}^{n_2} \phi_{2,i_3,i_2} \left(\sum_{i_1=1}^{n_1} \phi_{1,i_2,i_1} \left(\sum_{i_0=1}^{n_0} \phi_{0,i_1,i_0}(x_{i_0}) \right) \right) \right) \dots \right) \quad (3.4)$$

«Для мережі глибиною L шарів, з однаковою шириною шару $n_0 = n_1 = \dots = n_L = N$ та сплайнами порядку k (зазвичай $k = 3$) на G інтервалах розраховано часову складність алгоритму. Описана мережа містить $O(N^2L(G + k)) \sim O(N^2LG)$ параметрів. Для порівняння, MLP з глибиною L і шириною N складається лише з $O(N^2L)$ параметрів, що на перший погляд ефективніше, ніж KAN. Але, на практиці, KAN зазвичай потрібно набагато менше N, ніж для MLP, що не тільки зберігає параметри, але й досягає кращого узагальнення і полегшує інтерпретацію.»[4]

Потужність повнозв'язних нейронних мереж пояснюється універсальною теоремою про наближення, яка стверджує, що для заданої функції та похибки $\epsilon > 0$, двошарова мережа з $k > N(\epsilon)$ нейронів може апроксимувати функцію в межах похибки ϵ. Однак UAT не гарантує, як $N(\epsilon)$ масштабується з ϵ, тому мережі засновані на даному принципі, страждають від прокляття розмірності. Різниця між KAT і UAT зумовлена тим, що KAN використовують переваги внутрішнього низькорозмірного представлення функцій, тоді як MLP цього не забезпечують. У контексті KAT, увагу зосереджено на кількісному визначенні помилки апроксимації у композиційному просторі. З іншого боку, для загальних функціональних просторів, таких як простори Соболєва або Бесова, результати нелінійної теорії n-ширини свідчать, що уникнути прокляття розмірності неможливо. У той же час, MLP з функціями активації ReLU демонструють можливість досягнення високих швидкостей збіжності. Цей факт знову підкреслює важливість роботи з функціями композиційної структури — класом функцій, що є значно «зручнішими» з точки зору практичних і наукових застосувань, дозволяючи ефективно долати прокляття розмірності.

На відміну від MLP, KAN дозволяють використовувати компактніші архітектури завдяки здатності працювати із загальними нелінійними функціями активації. Наприклад, для досягнення необхідної швидкості збіжності MLP з функціями активації ReLU вимагають глибини щонайменше $log\ n$, де n – розмір вибірки. Також, що важливо для більшої інтерпретованості, KAN природно узгоджуються із символьними функціями, тоді як MLP такої властивості не мають.

3.2.1 Навчання KAN мережі. Як було зазначено раніше, значення виходів шару мережі можна отримати за допомогою наступної матриці переходу:

$$\phi^l = \begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{12} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \cdots & \phi_{mn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{3.5}$$

де ϕ^l – вектор виходів шару l, x_n – вектор входів шару l, n – кількість входів шару, m – кількість виходів шару.

Отже, загальний вихід мережі можна записати наступним чином:

$$KAN(x) = \phi^L(\phi^{L-1}(\dots(\phi^2(\phi^1(x))))) \tag{3.6}$$

де L – кількість шарів, x – вектор вхідних значень.

Для порівняння, виходи MLP мережі розраховуються наступним чином:

$$MLP(x) = \theta^L(\sigma(\theta^{L-1}(\dots(\sigma(\theta^2(\sigma(\theta^1(x)))))))) \tag{3.7}$$

Ключова різниця в тому, що параметри θ^i є лінійними перетвореннями, а σ позначає функцію активації, що використовується для нелінійності, яка є спільною для всіх шарів. У випадку KAN, матриці ϕ^l самі по собі є нелінійними матрицями перетворення, і кожна одновимірна функція може відрізнятися.

Для оцінки параметрів цих одновимірних функцій існує декілька підходів: криві Безьє та B-сплайни. Крива Безьє – параметрично задана крива, яку використовують у комп'ютерній графіці та суміжних галузях. Нехай, необхідно знайти криву, що проходить через 4 точки (Рис. 3.2):

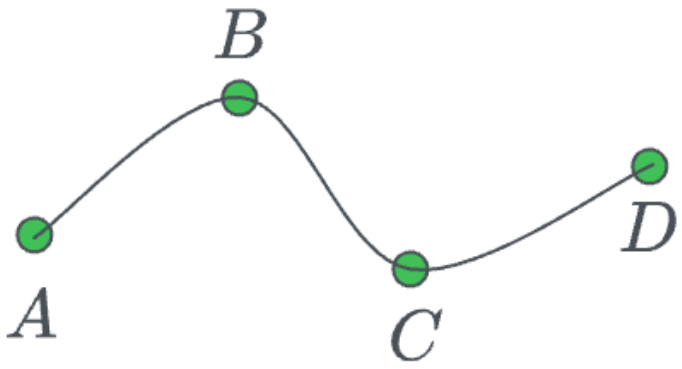

Рисунок 3.2 – Плавна траєкторія

Для пошуку такої кривої, необхідно оцінити коефіцієнти полінома вищого ступеня шляхом розв’язання системи лінійних рівнянь:

$$f(x) = ax^3 + bx^2 + cx + d \tag{3.8}$$

В загальному вигляді, якщо є N точок даних, необхідно вирішити N рівнянь для визначення параметрів. Розв’язування такої системи лінійних рівнянь буде обчислювально складною задачею, що є неприйнятним в системах реального часу. Криві Безьє ефективно вирішують цю проблему. Вони забезпечують спосіб представлення плавної кривої, яка проходить поблизу набору контрольних точок, без необхідності розв’язувати велику систему рівнянь.

Для двох точок, крива визначається наступним чином:

$$P = (1 - t)P_1 + tP_2 \tag{3.9}$$

де P_1, P_2 – контрольні точки, $0 \leq t \leq 1$. Звідси випливає, що при $t = 0$, позиція точки P буде співпадати з точкою P_1, і навпаки, при $t = 1$, позиція точки P буде співпадати з P_2.

Для трьох точок, алгоритм пошуку кривої аналогічний. Спочатку знаходяться точки Q_1 та Q_2, після чого розраховується положення точки P (Рис. 3.3).

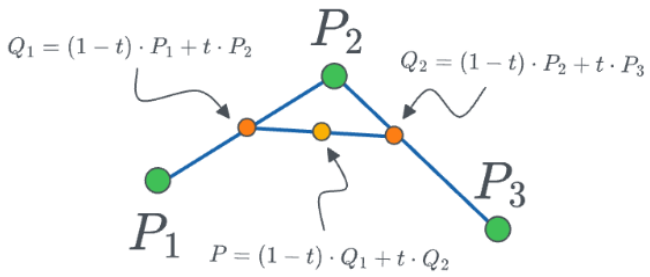

Рисунок 3.3 – Пошук кривої для 3 контрольних точок

Отже, рівняння кривої Безьє для трьох точок має наступний вигляд:

$$B(t) = (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3, t \in [0,1] \quad (3.10)$$

Узагальнюючи ідею, можна отримати рівняння кривої Безьє для n точок:

$$B(t) = \sum_{i=0}^{n} \frac{n!}{i!\,(n-i)!} (1-t)^{n-i} t^i P_i \quad (3.11)$$

Хоча даний алгоритм ефективніше, ніж розв'язання системи лінійних рівнянь, при великих n пошук кривої може бути обчислювально дорогим процесом. Для великої кількості контрольних точок, набагато ефективнішим алгоритмом відображення кривих є B-сплайни. На відміну від поліномів високого порядку, B-сплайни використовують набір сегментів полінома нижчого порядку, які гладко з'єднані. Іншими словами, замість розширення кривих Безьє на десятки сотень точок даних, що призводить до отримання поліному такого ж

високого порядку, використовується декілька поліномів нижчого порядку, які з'єднані між собою, щоб утворити гладку криву. Для n контрольних точок та поліноміальних кривих Безьє порядку k буде створено $(n-k)$ кривих в загальному B-сплайні. Основна ідея полягає в тому, щоб забезпечити певні умови неперервності в точках, де з'єднуються криві:

Неперервність позиції. Кінцева точка сегмента повинна збігатися з початковою точкою наступного сегмента.

Неперервність дотичної. Досягається шляхом забезпечення неперервності першої похідної в точках з'єднання. Оскільки окремі криві Безьє є поліномами, вони завжди неперервні.

Неперервність кривизни. Включає в себе похідні вищого порядку і є більш складною умовою, але забезпечує більш плавні переходи.

Подібно до кривих Безьє, остаточна крива B-сплайну представлена як лінійна комбінація точок P_i:

$$S(t) = \sum_{i=0}^{n} P_i N_{i,k} \tag{3.12}$$

де P_i – контрольні точки, що визначають форму кривої, $N_{i,k}(t)$ – базисні функції B-сплайну порядку k.

Отже, B-сплайни можна використовувати для тренування KAN мереж. Змінюючи положення контрольних точок під час навчання, модель KAN може динамічно формувати функції активації, які найкраще відповідають даним. Алгоритм тренування моделі звучить наступним чином:

1. Задаються початкові положення контрольних точок, аналогічно вагам в класичних нейронних мережах.

2. Положення контрольних точок оновлюються в процесі навчання за допомогою алгоритму зворотного поширення помилки, подібно до того, як оновлюються ваги в MLP.

3. Значення функції втрат мінімізується за допомогою алгоритму оптимізації (наприклад, градієнтний спуск).

В KAN кожна функція активації $\phi(x)$ визначається наступним чином:

$$\phi(x) = w(b(x) + spline(x)) \tag{3.13}$$

$$b(x) = \frac{x}{1 + e^{-x}} \tag{3.14}$$

$$spline(x) = \sum_i c_i B_i(x) \tag{3.15}$$

де c_i – змінюваний параметр, що позначає положення контрольних точок, B_i – базисна функція, w – змінюваний параметр, який є опціональним і використовується для контроля загальної величини функції активації.

Кожна функція активації ініціалізується з $spline(x) \approx 0$. Це досягається за допомогою коефіцієнта $c_i \sim N(0, \sigma^2)$ з малим значенням σ близько 0.1. Коефіцієнт w ініціалізовано за допомогою методу Xavier, основна ідея якого полягає в ініціалізації параметру випадковими значеннями за нормальним розподілом, середнє значення якого рівне нулю, а стандартне відхилення знаходиться в межах одиниці:

$$w \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right) \tag{3.16}$$

де n_{in} – кількість входів шару, n_{out} – кількість виходів шару.

3.2.2 Переваги KAN над MLP. KAN більш ефективні для представлення функцій, ніж MLP, у різних завданнях. Порівнюючи два сімейства моделей, справедливо порівнювати як їх точність (втрати), так і складність (кількість

параметрів). Як було зазначено раніше, при однаковій кількості нейронів, KAN має більшу кількість параметрів, але зазвичай потребує меншої кількості нейронів для досягнення аналогічної точності. Було проведено ряд експериментів на різних функція, що доводить це твердження (Рис. 3.4-3.5).

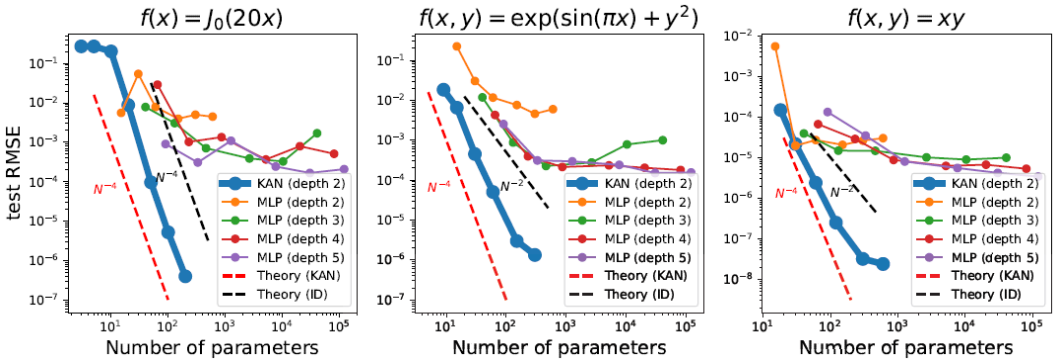

Рисунок 3.4 – Порівняння MLP з KAN

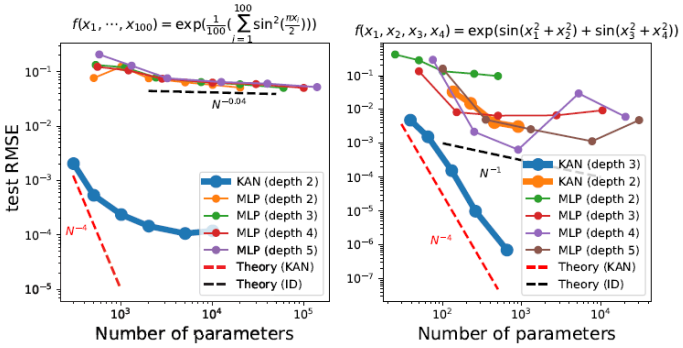

Рисунок 3.5 – Порівняння MLP з KAN на багатовимірних функціях

На графіках (Рис. 3.4-3.5) відображено залежність середньоквадратичного відхилення (RMSE) від кількості параметрів. Фактичні результати здебільшого відповідають теоретичним значенням (пунктирні лінії). KAN набагато краще масштабується, особливо для випадку багатовимірних функцій. MLP масштабується повільніше та швидко виходить на плато.

Іншою важливою перевагою KAN є підтримка постійного навчання. Катастрофічне забування є серйозною проблемою в сучасному машинному навчанні. Коли людина опановує одне завдання і перемикається на інше, вона не забуває, як виконувати перше завдання. На жаль, це не стосується нейронних

мереж. Коли нейронна мережа навчається на завданні 1, а потім перейшовши до навчання на завданні 2, мережа скоро «забуде» як виконувати завдання 1. Ключова різниця між штучними нейронними мережами та людським мозком полягає в тому, що людський мозок має функціонально відмінні модулі, розміщені локально в просторі. Коли вивчається нове завдання, реорганізація структури відбувається лише в локальних регіонах, відповідальних за відповідні навички, залишаючи інші регіони недоторканими. Більшість штучних нейронних мереж, включаючи MLP, не мають цієї локальності, що, ймовірно, є причиною катастрофічного забування. KAN мають локальну пластичність і можуть уникнути катастрофічного забування, використовуючи локальність сплайнів. Оскільки основи сплайнів локальні, вибірка впливатиме лише на декілька найближчих коефіцієнтів сплайну, залишаючи віддалені коефіцієнти недоторканими, що бажано, оскільки далекі регіони, можливо, вже зберегли інформацію, яку необхідно залишити. MLP зазвичай використовують глобальні функції активації, наприклад, ReLU/Tanh/SiLU тощо, тому будь-які локальні зміни можуть неконтрольовано поширюватися на віддалені регіони, знищуючи інформацію, яка там зберігається. Для підтвердження цієї гіпотези, KAN та MLP натреновано для розв’язання задачі одновимірної регресії, що складається з 5 Гаусових піків. Дані навколо кожного піку подаються послідовно у KAN та MLP, як показано на рисунку 3.6 у верхньому рядку. Прогнози KAN і MLP після кожної фази навчання показані в середньому та нижньому рядках відповідно.

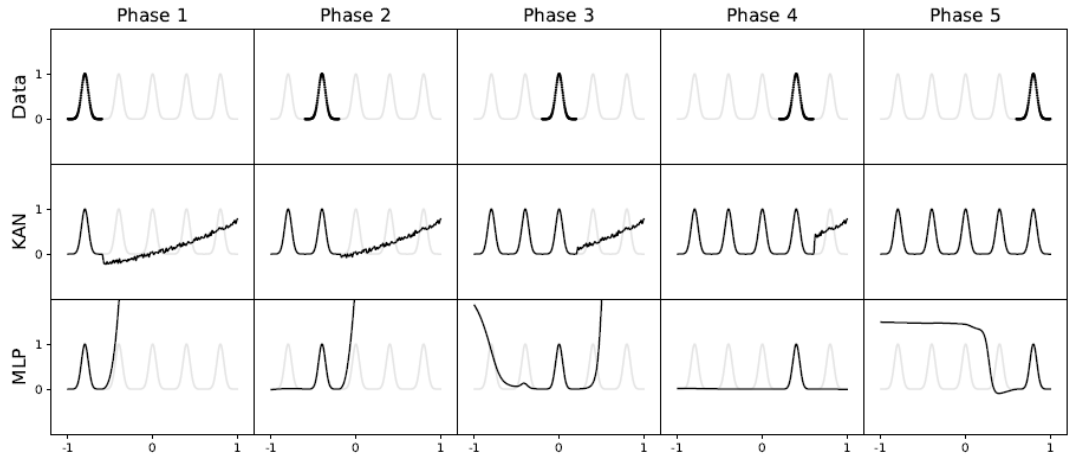

Рисунок 3.6 – Поетапне навчання KAN та MLP на демонстраційній вибірці

Як і очікувалося, KAN оновлює лише ті регіони, у яких присутні дані на поточному етапі, залишаючи попередні регіони без змін. На противагу, MLP повністю перебудовує всі регіони після отримання нових зразків даних, що призводить до катастрофічного забування.

Також варто відзначити інтерпретованість KAN мереж. На відміну від MLP мереж, які є «чорним ящиком», KAN мережі навчають одновимірні функції на всіх рівнях, що робить дослідження структури моделі досить простим. Щоб це продемонструвати, було побудовано мережу з архітектурою KAN(2, 1, 1), тобто 2 нейрони у вхідному шарі, 1 в прихованому, і 1 нейрон у вихідному шарі, $k = 3, G = 3$. Навчальну вибірку для такої мережі згенеровано наступною функцією:

$$f(x_1, x_2) = e^{\sin(\pi x_1)+x_2^2} \tag{3.17}$$

Отримано мережу з середньоквадратичним відхиленням на рівні $RMSE = 1.37e-02$ на навчальній вибірці. Структуру побудованої мережі відображено на рисунку 3.7.

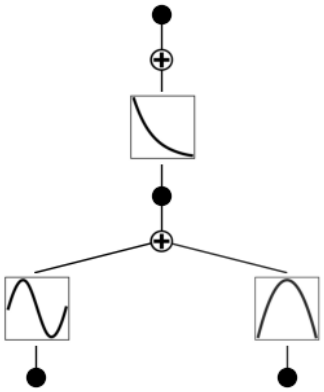

Рисунок 3.7 – Структура мережі

Помітно, що мережа змогла виділити синусоїдальну та параболічну функції з тренувальної вибірки. Якщо більш точно, мережа на основі вибірки, змогла вивести функцію регресії, яка має наступний вигляд:

$$f(x) = 0.1287 - 2.5527 * log(-0.3227 * (-x_2 - 0.0001)^2 - 0.3243 * sin(3.1404 * x_1 - 0.0022) + 0.6842) \quad (3.18)$$

З формули видно, що мережа досить точно визначила символьні функції та майже повністю відновила функцію, якою було згенеровано вибірку. Ця особливість відкриває нові можливості для дослідження складних процесів та виявлення нових знань в існуючих даних.

3.2.3 Застосування KAN в контексті поставленої задачі. Відповідно до вимог, на вхід побудованої KAN мережі мають подаватися значення з сенсорів платформи. Мережа оброблює ці дані, та подає команду на керування моторами. Отже, необхідно розв'язати задачу регресії.

Побудовано діаграму структури автоматизованої системи (Рис. 3.8).

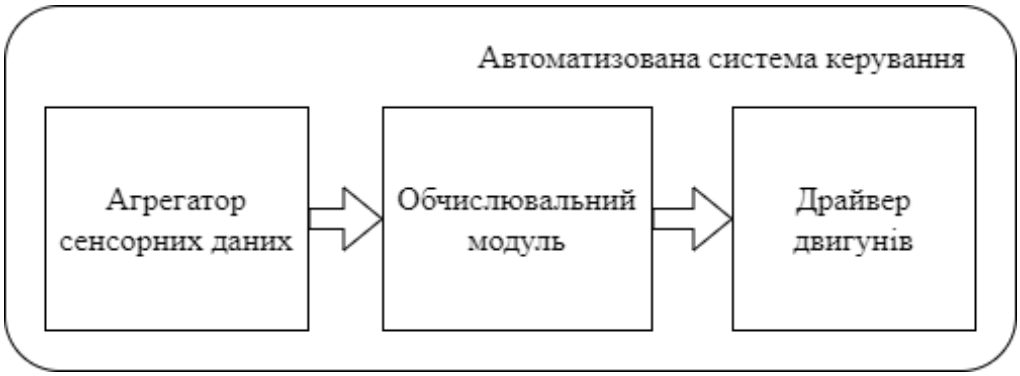

Рисунок 3.8 – Діаграма структури автоматизованої системи керування

Вихідний шар такої мережі повинен містити 2 нейрони, які відповідають за ліві та праві приводи платформи. Якщо значення одного з виходів перевищує значення іншого, відбувається поворот платформи в певну сторону, і навпаки, якщо значення виходів однакові, платформа має рухатися вперед. Значення виходів позначають відсоток від максимальної швидкості приводів. Обране рішення дозволяє перейти на вищий рівень абстракції, де система підтримки прийняття рішень нічого не знає про апаратну частину роботизованої платформи і лише віддає бажаний напрямок та відсоток «газу» на окремий модуль драйверу

двигунів, який самостійно визначає необхідну швидкість обертання кожного мотору. Це робить дану систему універсальною та дозволяє використовувати її на платформах з різними наборами приводів з мінімальними змінами. Розмірність вхідного вектору ознак залежить від кількості сенсорів та величини вектору даних кожного сенсору. Оскільки кожне значення – це окремий нейрон, значимість даних кожного сенсору рівна, але в ході навчання деяким сенсорам може бути присвоєний вищий пріоритет за рахунок їх більшого впливу на значення цільової функції. Альтернативно, поставлену задачу можна описати в термінах задачі класифікації, тоді виходи мережі позначали б дискретні команди, на зразок «Їхати вперед», «Повернути вліво» тощо. Але такий підхід ускладнює архітектуру мережі, збільшуючи розмірність вихідного вектору. Крім того, досягти плавного руху платформи у випадку дискретних команд неможливо.

3.3 Апаратні рішення

Апаратну частину розроблюваної роботизованої платформи можна умовно поділити на 3 модулі, аналогічно діаграмі на рисунку 3.8. Згідно з вимогами до платформи, модуль сенсорних даних повинен містити набір сенсорів, що дозволяє орієнтуватися в просторі та уникати перешкоди. Для спрощення задачі керування, буде розроблено платформу, що може слідувати за лінією. Отже мінімальний набір сенсорів повинен включати в себе: датчик лінії та датчик відстані. Також можна додати датчик кольору, що розширює можливості по керуванню платформою. Так, наприклад, можна контролювати максимальну швидкість руху платформи, змінюючи колір лінії. Модуль драйверу двигунів складається з двох окремих модулів, кожен з яких керує парою моторів (передньою та задньою). Контроль та керування модулями здійснюється обчислювальним модулем, який представляє собою одноплатний комп’ютер з встановленим програмним забезпеченням.

Модульність розроблюваної платформи дозволяє легко змінювати будь-який компонент системи, додавати, або прибирати сенсори. Було обрано колісну базу майбутньої платформи (Рис. 3.9).

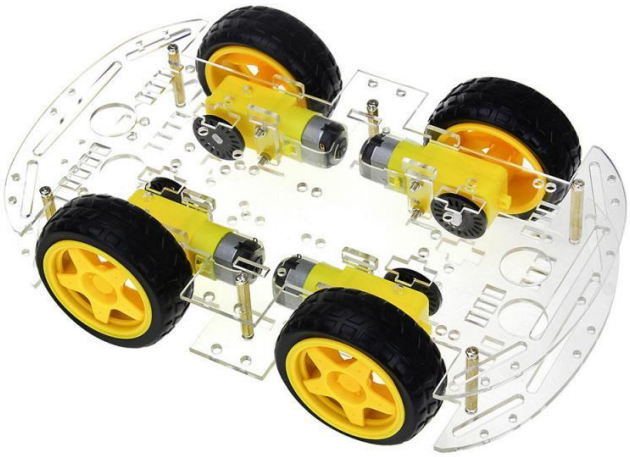

Рисунок 3.9 – Двопалубна колісна база

Обрана колісна база є повнопривідною та містить достатню кількість отворів для кріплення, що дозволяє легко розміщувати всю необхідну електроніку. Хоча платформа має дві палуби, чого достатньо для розміщення великої кількості обладнання, габарити цього обладнання обмежуються габаритами колісної бази, які становлять 235 х 155 мм.

Варто більш детально розглянути з яких апаратних блоків складається кожен модуль розроблюваної роботизованої платформи.

3.3.1 Модуль сенсорів. Як було зазначено вище, до набору сенсорів мають входити датчик лінії, відстані до перешкод та датчик кольору. На ринку наявна широка номенклатура подібних сенсорів, які базуються на дещо різних принципах, мають відмінний клас точності, діапазони вимірювань та габарити.

Враховуючи вимоги та обмеження розроблюваної системи, було обрано ультразвуковий датчик відстані HC-SR04, інфрачервоний датчик лінії QTR-8A та датчик кольору TCS230. Детально розглянуто кожен сенсор.

Датчик лінії QTR-8A

Pololu QTR-8A використовується як датчик лінії, але модуль може використовуватися як датчик наближення або відбиття. Кожен модуль складається з восьми пар ІЧ-випромінювачів і приймачів (фототранзисторів), рівномірно розташованих на відстані 9,525 мм один від одного. Усі виходи незалежні, але світлодіоди розташовані парами, щоб зменшити струм споживання вдвічі. Світлодіодами керує MOSFET-транзистор із затвором, який зазвичай підтягується до напруги живлення, що дозволяє світлодіодам вимикатися при низькій напрузі на затворі транзистору. Вимикання світлодіодів може бути корисним для обмеження споживання електроенергії, коли датчики не використовуються, або для зміни ефективної яскравості світлодіодів за допомогою ШІМ. Модуль підтримує роботу в двох режимах: 5 В та 3.3 В. Струм споживання світлодіода становить приблизно 20-25 мА, що робить загальне споживання плати трохи менше 100 мА. Принципова схема модуля показана нижче (Рис. 3.10):

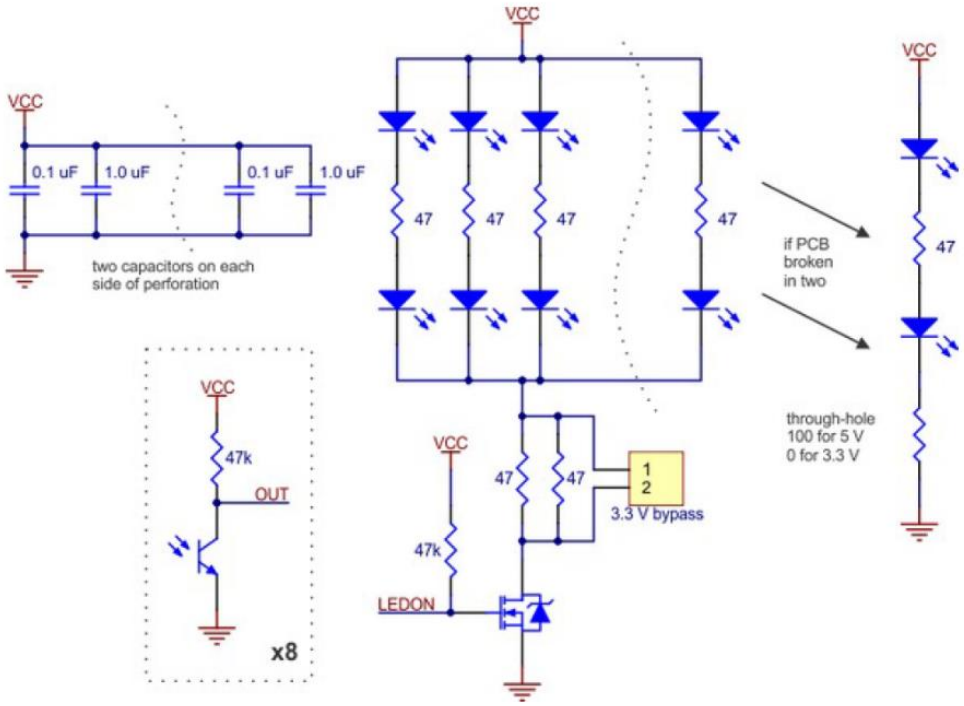

Рисунок 3.10 – Принципова схема датчика лінії QTR-8A

Специфікація модуля:

- розміри: 75x13 мм;
- робоча напруга: 3,3-5,0 В;
- струм живлення: 100 мА;
- формат виведення для QTR-8A: 8 аналогових виходів в діапазоні від 0 В до напруги живлення;
- оптимальна відстань детекції: 3 мм;
- максимальна рекомендована відстань детекції для QTR-8A: 6 мм.

Плата датчику має наступну розпіновку (Рис. 3.11):

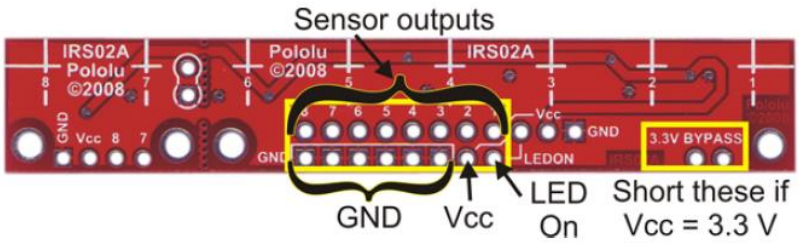

Рисунок 3.11 – Розпіновка датчику лінії QTR-8A

Масив датчиків відбиття QTR-8A має вісім окремих виходів датчиків, по одному від кожної пари світлодіод/фототранзистор. Напруга на виходах змінюється в діапазоні від 0 В до Vcc (від 3,3 до 5 В). При сильному відбитті, наприклад, коли датчик знаходиться над білою поверхнею, його вихідна напруга буде наближатися до 0 В і навпаки, з дуже слабким коефіцієнтом відбиття, наприклад, коли датчик знаходиться над чорною поверхнею, його вихідна напруга буде близькою до Vcc. Для отримання чіткого діапазону показань між білою та чорною поверхнею, рекомендовано встановлювати датчик на відстані не більше ніж 6 мм від поверхні [12].

Обробка аналогових даних цифровими системами потребує попередньої конвертації за допомогою аналого-цифрового перетворювача (АЦП). Процес конвертації аналогового сигналу включає три етапи: спочатку АЦП вимірює значення аналогового сигналу в певний момент часу, що називається дискретизацією; потім отримане значення проходить процес квантування, під

час якого сигнал розбивається на фіксовані рівні залежно від розрядності пристрою; на завершальному етапі, сигнал кодується в цифровий формат, що відповідає найближчому рівню квантування.

Хоча мікроконтролери зазвичай мають набір аналогових пінів, їх кількість та розмірність обмежені. Недостатній діапазон вимірювань призводить до втрати частини інформації аналогового сигналу, що може бути критичним в системах з високою чутливістю. В поставленій задачі, доречним буде використання зовнішніх АЦП. Таким чином, простіше підібрати необхідну частоту дискретизації та розрядність пристрою, незалежно від обраного обчислювального модулю.

Для даної системи було обрано модуль АЦП ADS1115. Серед характеристик модуля, варто відзначити наступне: розрядність – 16 біт, максимальна частота дискретизації – 860 вимірювань/с, кількість каналів – 4, інтерфейс – I2C. Важливою особливістю даного модуля є інтерфейс комунікації I2C – це послідовний інтерфейс для зв'язку між мікроконтролерами та периферійними пристроями, розроблений компанією Philips. Він дозволяє підключати декілька пристроїв до однієї шини, використовуючи лише дві лінії: SCL для синхронізації та SDA для передачі даних. Основна ідея I2C полягає в організації зв'язку між ведучим і підлеглими пристроями. Ведучий генерує тактовий сигнал та ініціює передачу даних, в той час як підлеглі пристрої відповідають на його запити. Кожен пристрій на шині має унікальну адресу, яка дозволяє ідентифікувати пристрій, з яким відбувається обмін. I2C підтримує двобічну передачу даних, використовуючи механізми підтвердження для перевірки успішності обміну. Передача даних виконується пакетами, що складаються зі стартового біта, адреси пристрою, даних та біта зупинки. Протокол широко використовується завдяки його простоті, гнучкості та можливості підключення до 127 пристроїв на одній шині.

Оскільки модуль має лише 4 аналогові канали, для повноцінного використання датчику лінії QTR-8A необхідно два таких модуля. Використання інтерфейсу I2C дозволяє вивільнити додаткові піни мікрокомп’ютеру

(мікроконтролеру) для інших потреб, оскільки обидва модулі ADS1115 потребують лише двох пінів та можуть бути підключені до однієї шини. Представлено спрощену блочну діаграму модуля (Рис. 3.12).

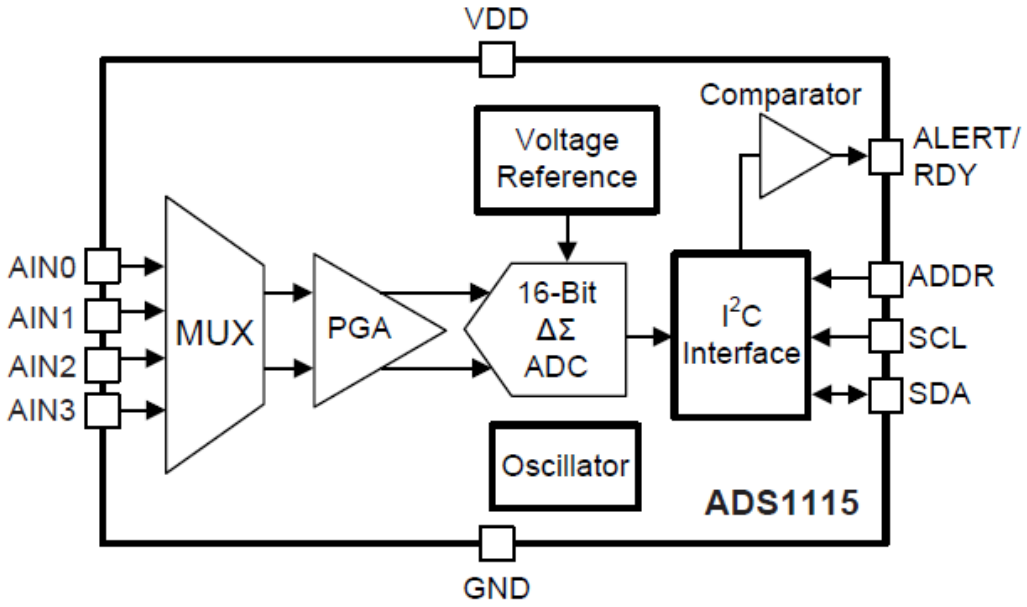

Рисунок 3.12 – Блочна діаграма модуля ADS1115

Модуль постачається на друкованій платі з габаритами 30x20 мм та має наступний вигляд (Рис. 3.13):

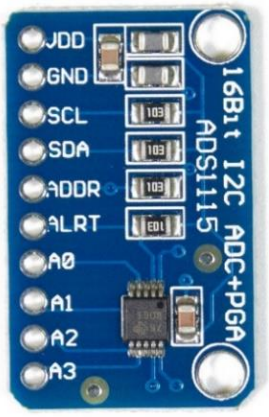

Рисунок 3.13 – Зовнішній вигляд модуля ADS1115 з розпіновкою

Робочі діапазони напруг на входах мікросхеми представлено в таблиці 3.1.

Таблиця 3.1 – Робочі діапазони напруги на входах мікросхеми

	Піни	Мінімальна	Максимальна	Величина
Напруга живлення	VDD, GND	-0,3	7	В
Напруга аналогових входів	A0, A1, A2, A3	GND – 0,3	VDD + 0,3	В
Напруга цифрових входів	SDA, SCL, ADDR, ALRT	GND – 0,3	5,5	В
Вхідний струм	Всі піни крім живлення	-10	10	мА

Модуль ADS1115 працює в одному з двох режимів: безперервна конвертація та разова конвертація. Біт MODE в регістрі Config вибирає відповідний режим роботи. Коли біт MODE в регістрі конфігурації встановлений на 1, модуль переходить у стан вимкнення живлення та працює в режимі разової конвертації. Цей стан живлення є станом за замовчуванням для ADS111x під час першого подання живлення [13]. Незважаючи на вимкнене живлення, пристрої реагують на команди. ADS111x залишається в цьому стані, доки біт зі значенням 1 не буде записаний в біт робочого стану (OS) у регістрі Config. Коли встановлено біт OS, пристрій увімкнеться приблизно через 25 мкс, скине біт OS на 0 і розпочне одиничну конвертацію. Коли дані зконвертовані та готові до отримання, пристрій знову вимикається. Запис 1 у біт OS під час конвертації не має жодного ефекту. Для переходу в режим безперервної конвертації, необхідно записати 0 в біт MODE в регістрі Config. Після завершення конвертації ADS111x розміщує результат в регістрі перетворень і починає наступну конвертацію.

ADS1115 має один пін ADDR, який відповідає за адресу I2C пристрою. Даний пін може бути підключений до GND, VDD, SDA або SCL, що дозволяє обрати чотири різні адреси одним контактом, як показано в таблиці 3.2. Стан контакту ADDR безперервно зчитується. Якщо SDA використовується для задання адреси пристрою, необхідно утримувати лінію SDA на низькому рівні

принаймні 100 нс після переведення лінії SCL в низький стан, щоб переконатися, що модуль правильно зчитує адресу під час зв’язку по I2C.

Таблиця 3.2 – Підключення ADDR піну та відповідний адрес

Пін ADDR	**Адрес пристрою**
GND	1001000
VDD	1001001
SDA	1001010
SCL	1001011

Оскільки в даній роботі буде використано два модулі ADS1115 на одній шині I2C, пін ADDR одного з них необхідно під’єднати до GND, пін іншого – до VDD. Таким чином, датчик лінії QTR8A можна підключити до будь-якого мікроконтролеру чи мікрокомп’ютеру, що підтримує I2C, лише за допомогою двох контактів.

Ультразвуковий датчик відстані HC-SR04

Ультразвуковий модуль вимірювання відстані HC-SR04 забезпечує функцію безконтактного вимірювання дистанції від 2 см до 400 см, точність вимірювання може досягати 3 мм. Модуль включає ультразвуковий передавач, приймач, та схему керування. Модуль має наступну розпіновку (Рис. 3.14).

Рисунок 3.14 – Розпіновка модуля HC-SR04

Специфікацію модуля приведено в таблиці 3.3.

Таблиця 3.3 – Специфікація модуля HC-SR04

Робоча напруга	5 В
Робочий струм	15 мА
Робоча частота	40 кГц
Максимальна відстань	400 см
Мінімальна відстань	2 см
Кут вимірювання	15°
Вхідний сигнал Trigger	Імпульс довжиною 10 мкс
Вихідний сигнал Echo	Значення, пропорційне ширині імпульсу та дистанції
Габарити	45*20*15 мм

Щоб розпочати вимірювання відстані, необхідно лише подати короткий імпульс в 10 мкс на вхід тригера. Після цього модуль надсилає вісім сплесків ультразвуку на частоті 40 кГц. Exo – це величина, яка пропорційна ширині імпульсів та відстані. Відстань можна розрахувати за формулою (3.19), вимірюючи інтервал часу між надсиланням тригерного сигналу та отриманням ехо-сигналу. Рекомендовано використовувати цикл вимірювання більше 60 мс для запобігання переходу тригерного сигналу в ехо-сигнал [14].

$$d = t * v/2 \tag{3.19}$$

де d – відстань до перешкоди, t – тривалість високого сигналу, $v \approx 340\frac{\text{м}}{\text{с}}$ – швидкість звуку в повітрі.

Датчик кольору TCS230

Програмований конвертер світла в частоту TCS230 поєднує кремнієві фотодіоди з можливістю налаштування та перетворювач струму в частоту на одній монолітній інтегральній схемі CMOS. Вихід конвертера представляє собою прямокутну хвиля (50% робочого циклу) з частотою, прямо пропорційною інтенсивності світла. Вихідна частота може бути масштабована за допомогою одного з трьох заданих значень за допомогою двох вхідних контактів керування. Цифрові входи та виходи забезпечують прямий доступ до мікроконтролера або іншої логічної схеми. Увімкнення виходу ($\overline{OE}$) переводить вихід у стан високого опору для спільного використання вхідної лінії мікроконтролера декількома блоками. Модуль зчитує матрицю з 8x8 фотодіодів. 16 фотодіодів мають сині фільтри, 16 фотодіодів мають зелені фільтри, 16 фотодіодів мають червоні фільтри та 16 фотодіодів прозорі без фільтрів. Чотири типи (кольори) фотодіодів зміщені між собою, щоб мінімізувати ефект нерівномірності падаючого випромінювання. Усі 16 фотодіодів одного кольору з'єднані паралельно, і тип фотодіода, який пристрій використовує під час роботи, вибирається окремими контактами. Фотодіоди мають розмір 120x120 мкм і розташовані на центрах 144 мкм. Функціональна діаграма модуля виглядає наступним чином (Рис. 3.15).

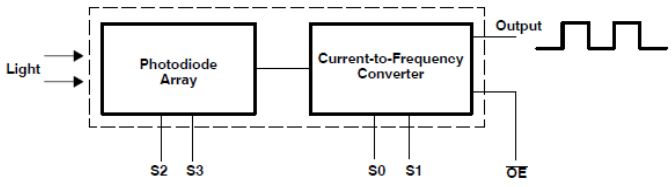

Рисунок 3.15 – Функціональна діаграма модуля TCS230

Схематично, модуль має наступний вигляд з розпіновкою (Рис. 3.16).

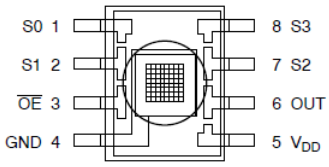

Рисунок 3.16 – Розпіновка модуля TCS230

Функціональне призначення контактів модуля описано в таблиці 3.4.

Таблиця 3.4 – Функціональне призначення контактів TCS230

Пін	**№**	**I/O**	**Опис**
GND	4		Загальна земля
$\overline{OE}$	3	I	Увімкнення виходу (активний при низькому сигналі)
OUT	6	O	Вихідна частота
S0, S1	1, 2	I	Масштабування вихідної частоти
S2, S3	7, 8	I	Вибір типу вхідного фотодіода
V_{DD}	5		Напруга живлення

Режими роботи модуля задаються пінами S0-S3. Доступні режими представлені в таблиці 3.5 [15].

Таблиця 3.5 – Режими роботи модуля TCS230

S0	**S1**	**Масштабування вихідної частоти**
L	L	Відключення живлення
L	H	2%
H	L	20%
H	H	100%
S2	**S3**	**Тип фотодіоду**
L	L	Червоний
L	H	Синій
H	L	Без фільтра
H	H	Зелений

Всі описані модулі здатні працювати від 5 В, отже їх можна підключати до однієї спільної шини живлення. Для зручності монтажу на платформу, було розроблено 3Д-моделі кріплень під кожний сенсор та надруковано на 3Д-

принтері (Додаток Б). Для об’єднання всіх сенсорів в один модуль з можливістю швидкої заміни, або відключення певного сенсору, спроектовано та розведено друковану плату (Додаток В). Проектування принципової схеми модуля виконано в EasyEDA. Він представляє собою веб-набір інструментів EDA, який дозволяє інженерам апаратного забезпечення проектувати, моделювати, ділитися та обговорювати схеми, моделі та друковані плати [16]. Інші функції включають створення списку матеріалів, файлів Gerber, а також файлів вихідних документів у форматах PDF, PNG і SVG. Безкоштовний акаунт пропонується для публічних проектів, а також обмеженої кількості приватних проектів. Кількість приватних проектів можна збільшити, надаючи високоякісні публічні проекти, схематичні символи та схеми друкованих плат та/або сплачуючи місячну підписку.

Плати виготовлено за допомогою сервісу JLCPCB. JLCPCB – це провідний сервіс виготовлення друкованих плат. Компанія пропонує швидке та доступне виробництво плат різної складності, включаючи послуги поверхневого монтажу компонентів. JLCPCB відома низькими цінами, високою якістю продукції, а також підтримкою прототипування і серійного виробництва. Сервіс дозволяє замовляти плати через зручний онлайн-інтерфейс, де можна завантажувати файли Gerber і налаштовувати параметри виготовлення, такі як: кількість шарів, товщина текстоліту, колір паяльної маски тощо [17].

3.3.2 Обчислювальний модуль. Розроблювана платформа складається з модулів, які потребують значних обчислювальних можливостей для коректної роботи в реальному часі. Враховуючи, що необхідно розробити інтелектуальну систему керування, малопотужні мікроконтролери в даній задачі є незастосовними, тому було прийнято рішення використовувати мікрокомп’ютер Raspberry Pi Zero 2W в якості обчислювального модулю.

Raspberry Pi Zero 2W – це наступне покоління Raspberry Pi Zero W в тому ж форм-факторі, але відмінність в тому, що на платі встановлений чотириядерний 64-бітний процесор Arm Cortex-A53 з тактовою частотою 1 ГГц.

В його основі лежить система в корпусі (SiP) Raspberry Pi RP3A0, в якій інтегрована матриця Broadcom BCM2710A1 з 512 МБ ОЗУ. Модернізований процесор забезпечує Raspberry Pi Zero 2 W на 40% більше однопоточної продуктивності та в п'ять разів більше багатопоточної продуктивності, ніж у оригінального одноядерного Raspberry Pi Zero. Мікрокомп’ютер пропонує бездротову локальну мережу 802.11 b/g/n на частоті 2,4 ГГц та Bluetooth 4.2, а також підтримку Bluetooth з низьким енергоспоживанням та модульну сертифікацію відповідності [18]. На платі є слот для карт пам'яті microSD, роз'єм для камери CSI-2, порт USB OTG і нерозпаяний 40-контактний роз'єм GPIO. Мікрокомп'ютер живиться від роз'єму micro USB. Виведення відео здійснюється через порт mini HDMI. За потреби композитний відеовихід можна легко зробити доступним через контрольні точки. Плата живиться від напруги в 5 В та споживає 2.5 А. Приведено загальний вигляд та розпіновку GPIO роз’єму Raspberry Pi Zero 2W (Рис. 3.17).

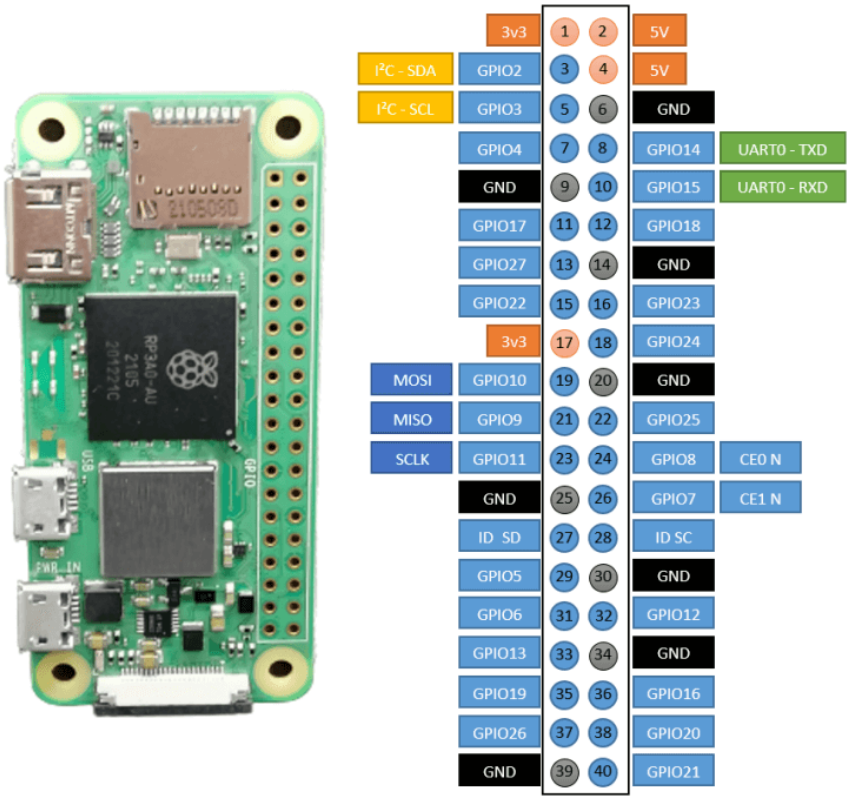

Рисунок 3.17 – Розпіновка Raspberry Pi Zero 2W

Завдяки GPIO пінам, мікрокомп’ютер підтримує ряд послідовних інтерфейсів, таких як: I2C, SPI та UART. Підтримка WiFi на частоті 2.4 ГГц дозволяє легко взаємодіяти з Raspberry за допомогою бездротових протоколів, моніторити стан системи та налаштовувати прошивку. Дані особливості роблять Raspberry Pi Zero 2W ідеальним вибором для розроблюваної автоматизованої системи керування. Розведено друковану плату для обчислювального модуля (Додаток В).

3.3.3 Драйвер двигунів. Встановлені на платформі колекторні двигуни з редуктором живляться від напруги в 6 В та номінально споживають струм до 0.5 А. Отже, для керування парою таких моторів достатньо одного малопотужного Н-мостового драйверу моторів. Для даної задачі було обрано драйвер на базі мікросхеми L293D.

L293D складається з чотирьох високострумових наполовину Н-мостових драйверів. L293D розроблено для забезпечення двонаправлених струмів приводу до 600 мА при напрузі від 4,5 В до 36 В [19]. Мікросхема призначена для керування індуктивними навантаженнями, такими як реле, соленоїди, крокові двигуни постійного струму та біполярні двигуни, а також інші навантаження з високим струмом/високою напругою із позитивним джерелом живлення. Приведено схематичний вигляд драйверу та його розпіновку в корпусі DIP-16 (Рис. 3.18).

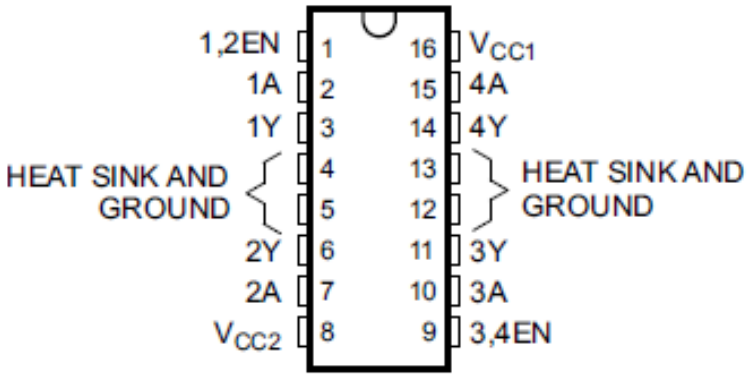

Рисунок 3.18 – Розпіновка драйверу L293D

Описано призначення контактів драйверу в таблиці 3.6.

Таблиця 3.6 – Опис пінів драйверу L293D

Пін	**№**	**I/O**	**Опис**
1,2EN	1	I	Вмикає канали 1 та 2
<1:4>A	2, 7, 10, 15	I	Неінвертовані входи драйверу
<1:4>Y	3, 6, 11, 14	O	Виходи драйверу
3,4EN	9	I	Вмикає канали 3 та 4
GND	4, 5, 12, 13	–	Заземлення схеми та тепловідвід
$V_{CC}1$	16	–	5 В живлення внутрішньої логіки
$V_{CC}2$	8	–	Напруга силової частини від 4.5 В до 36 В

В таблиці 3.7 представлено граничні значення напруги, струму та температури.

Таблиця 3.7 – Мінімальні та максимальні величини драйвера L293D

	Мінімальне	**Максимальне**	**Величина**
Напруга живлення, V_{CC1}	–	36	В
Вихідна напруга, V_{CC2}	–	36	В
Вхідна напруга, V_I	–	7	В
Вихідна напруга, V_O	-3	V_{CC2} + 3	В
Піковий вихідний струм, I_O	-1.2	1.2	А
Постійний вихідний струм, I_O	-600	600	мА
Максимальна робоча температура	–	150	°C
Температура зберігання	-65	150	°C

Представлено функціональну діаграму драйверу (Рис. 3.19).

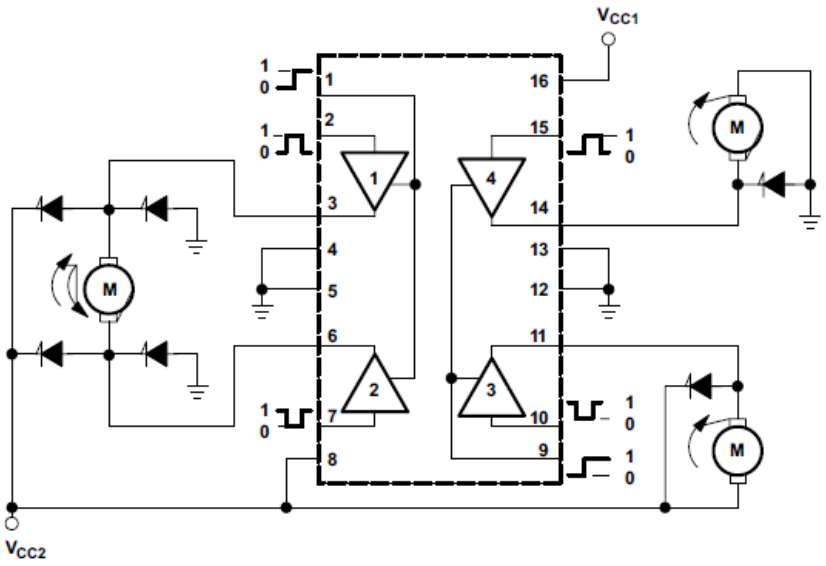

Рисунок 3.19 – Функціональна діаграма драйверу L293D

В залежності від схеми підключення, драйвер може керувати одним крововим двигуном, двома двигунами постійного струму, або чотирма двигунами постійного струму в одному напрямку. Приведено функціональну таблицю драйверу (Таблиця 3.8).

Таблиця 3.8 – Функціональна таблиця

Входи		**Виходи (Y)**
A	**EN**	
H	H	H
L	H	L
X	L	Z

де L – низький рівень сигналу, H – високий рівень, X – рівень сигналу не важливий. В режимі температурного відключення, вихід переходить в стан Z – високо імпедансу не залежно від входів.

Драйвер працює з досить суттєвим навантаженням, відносно його лінійних розмірів, тому використання в схемах з двома моторами постійного струму потребує заходів з відведення тепла. Рекомендується з’єднувати землю драйвера з площиною загальної землі друкованої плати за допомогою перехідних отворів,

або залишати мідні площадки біля контактів землі драйверу. Також можна використовувати радіатор для більш ефективного відведення тепла.

Для керування напрямком оберту двигуна, достатньо підключити один з входів драйверу через інвертор. Це дозволяє спростити інтерфейс підключення драйверу. Інвертор виконано за допомогою біполярного npn-транзистора SS8050 в корпусі SOT-323. Приведено розпіновку даного транзистору (Рис. 3.20).

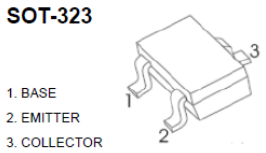

Рисунок 3.20 – Розпіновка транзистору SS8050

Отже, загальна принципова схема модуля драйверу двигунів виглядає наступним чином (Рис. 3.21):

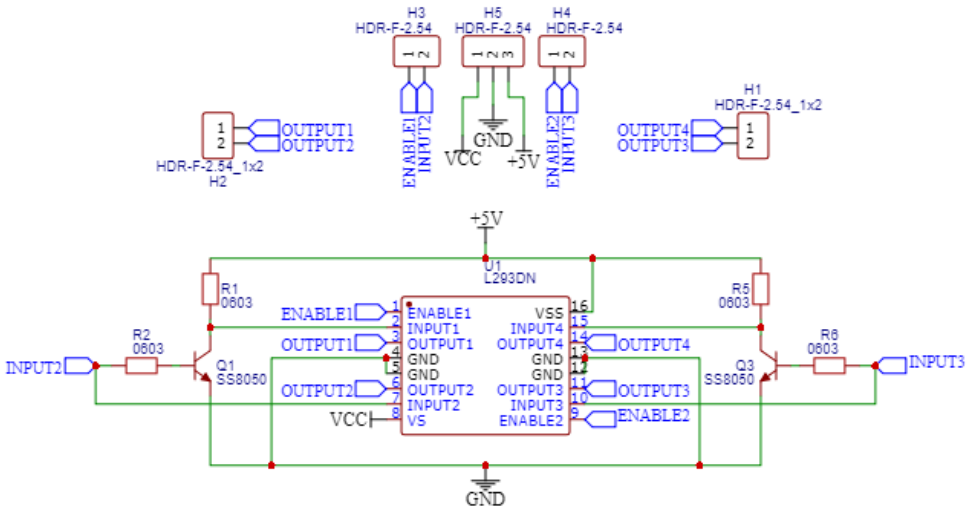

Рисунок 3.21 – Принципова схема модуля драйверу

Резистор на базі транзистора обмежує струм для захисту бази від пошкоджень, а резистор на колекторі обмежує струм колектор-емітерного переходу та підтягує високий потенціал, коли транзистор закритий. Розведено друковану плату модуля драйверу двигунів, з врахуванням вимог до охолодження (Додаток В).

Загальну принципову схему апаратної частини автоматизованої системи керування роботизованою платформою розроблено в програмному забезпеченні Proteus [20] (Рис. 3.22).

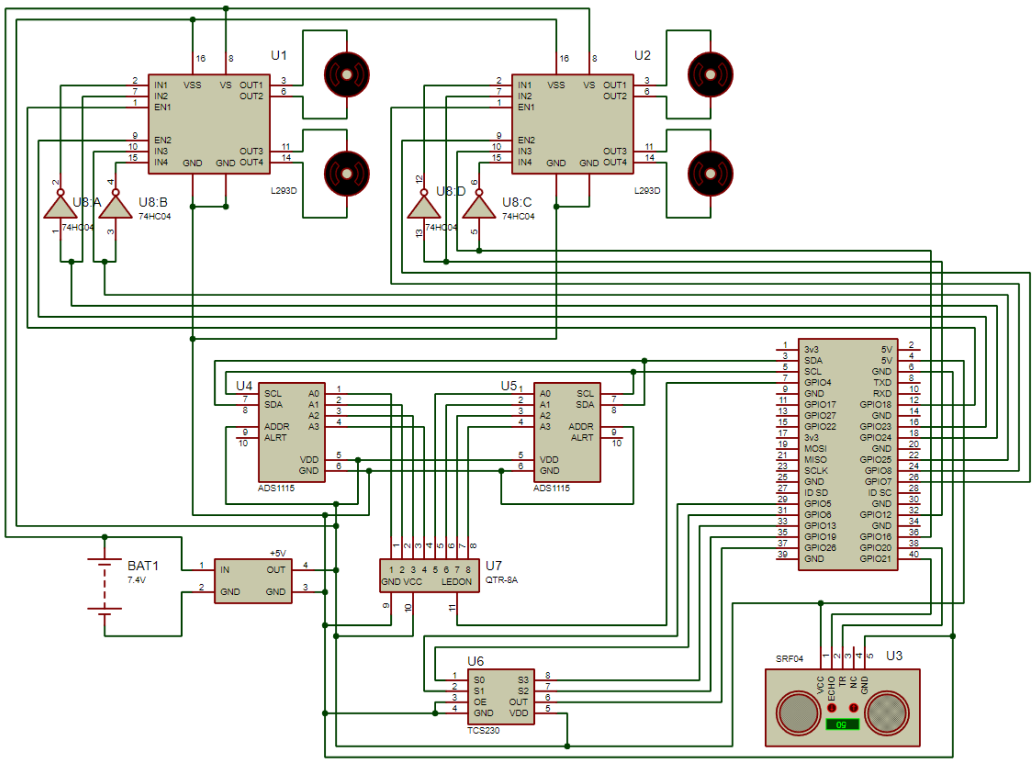

Рисунок 3.22 – Принципова схема апаратної частини

Оскільки схема містить дві ділянки з різною напругою (7,4 В та 5 В), було додано DC-DC перетворювач для пониження напруги батареї з 7,4 В до 5 В для живлення логіки. Напруга живлення 7,4 В відповідає акумуляторній збірці, що складається з двох послідовно з'єднаних елементів живлення. Пікове споживання схеми складає приблизно 6 А, але реальне споживання буде ближче до 4 А. Для живлення роботизованої платформи було обрано збірку з двох літій-полімерних елементів живлення ємністю 1000 мА*год, чого має вистачити на 15 хвилин безперервної роботи платформи.

В розділі було проведено детальний аналіз вимог до системи керування, на основі якого сформульовано та обґрунтовано технічні рішення для їх реалізації. З урахуванням розглянутих аспектів та прийнятих рішень, створено основу для переходу до етапу практичної реалізації автоматизованої системи керування.

РОЗДІЛ 4
ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ КЕРУВАННЯ

У попередніх розділах було розглянуто теоретичні основи та алгоритми, які лежать в основі автоматизованої системи керування роботизованою платформою. В даному розділі буде представлена програмна реалізація системи керування, яка включає в себе розробку симулятора за допомогою модуля pygame, що дозволить моделювати рух роботизованої платформи в реальному часі та тестувати алгоритми керування у віртуальному середовищі. Симулятор буде використовуватися для перевірки коректності розроблених алгоритмів, їх взаємодії та адаптивності до змін середовища.

4.1 Розробка віртуального середовища симуляції руху платформи

Модуль pygame надає широкий вибір варіантів візуалізації елементів графіки, як наприклад малювання простих геометричних фігур, ліній, кривих та відображення зображень. Представлено основу моделі роботизованої платформи у вигляді растрового зображення (Рис. 4.1).

Рисунок 4.1 – Роботизована платформа

Відповідно до вимог, платформа повинна містити 3 датчики. Датчики було відображено за допомогою простих геометричних фігур: лінія прямокутників зеленого кольору (інфрачервоний датчик лінії), круг зеленого кольору (ультразвуковий датчик відстані), який змінює колір на червоний, пропорційно

відстані до перешкоди, та квадрат, колір якого залежить від кольору поверхні під ним (датчик кольору). Платформа з нанесеними датчиками у відповідних місцях має наступний вигляд (Рис.4.2).

Рисунок 4.2 – Платформа з датчиками

Трек для платформи представляє собою замкнену фігуру, що складається з довільної кількості відрізків довільної довжини (Рис. 4.3).

Рисунок 4.3 – Трек для роботизованої платформи

Модель керування платформи у віртуальному середовищі враховує лінійні розміри платформи, діаметр коліс та максимальну швидкість їх обертання, для імітації руху реальної роботизованої платформи. Як було зазначено в попередньому розділі, рух платформи в симуляторі задається величиною, яка є відсотком від максимальної потужності моторів, окремо для моторів зліва і справа від центру платформи. Дані, що зчитуються з моделей сенсорів за значенням та форматом імітують дані з реальних сенсорів.

Побудовано діаграму класів реалізованої системи з симулятором в нотації UML (Рис. 4.4).

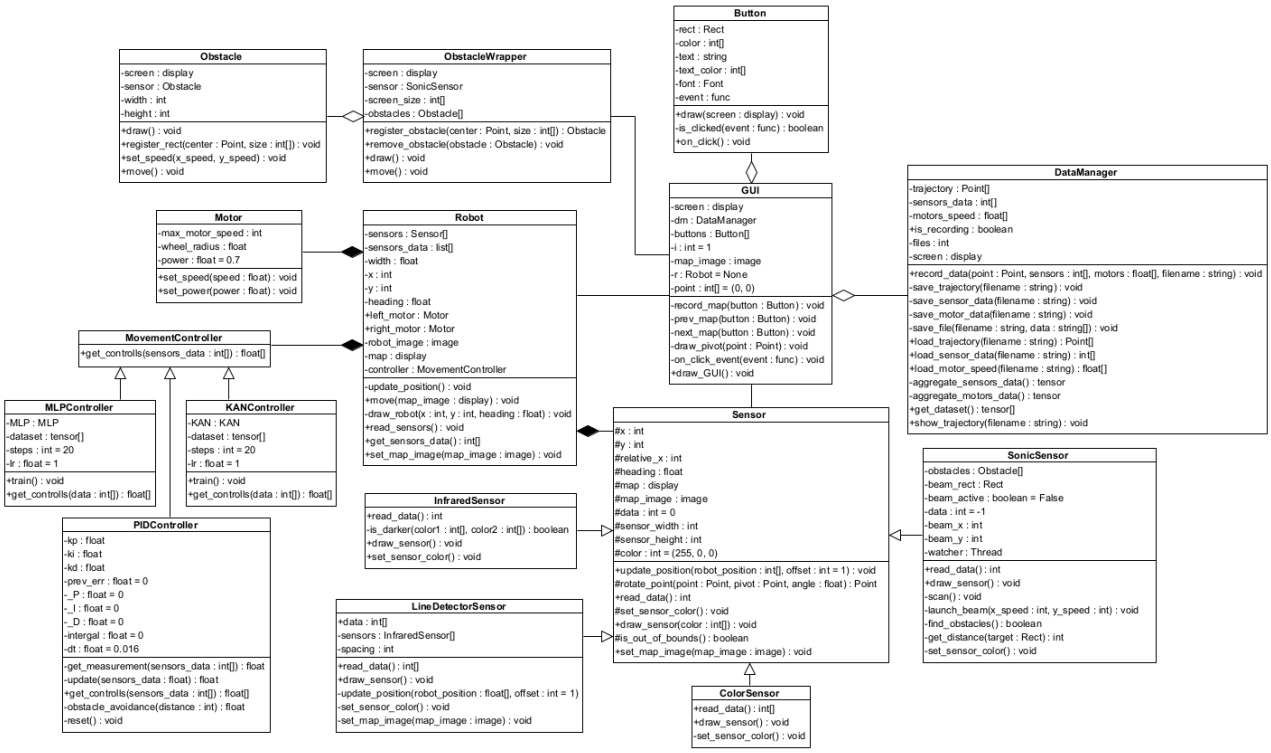

Рисунок 4.4 – Діаграма класів

Коротко описано призначення кожного з класів:

- Obstacle – клас, що представляє умовну перешкоду;
- ObstacleWrapper – клас обгортка;
- Button – кнопка, елемент графічного інтерфейсу;
- GUI – відповідає за графіку симулятора та інтерфейс;
- DataManager – клас для зберігання даних та створення вибірки;
- Sensor – абстрактний клас датчику;
- ColorSensor – датчик кольору;
- SonicSensor – ультразвуковий датчик перешкод;
- InfraredSensor – інфрачервоний датчик;
- LineDetectorSensor – датчик лінії, що складається з інфрачервоних;
- MovementController – абстрактний клас моделі керування;
- PIDController – модель керування, заснована на PID;
- MLPController – модель керування, заснована на MLP;
- KANController – модель керування, заснована на KAN;

- Motor – клас представляє двигун платформи;
- Robot – основний клас, що представляє роботизовану платформу зі всіма датчиками та двигунами.

Загальне вікно симулятора в кнопками керування виглядає наступним чином (Рис. 4.5):

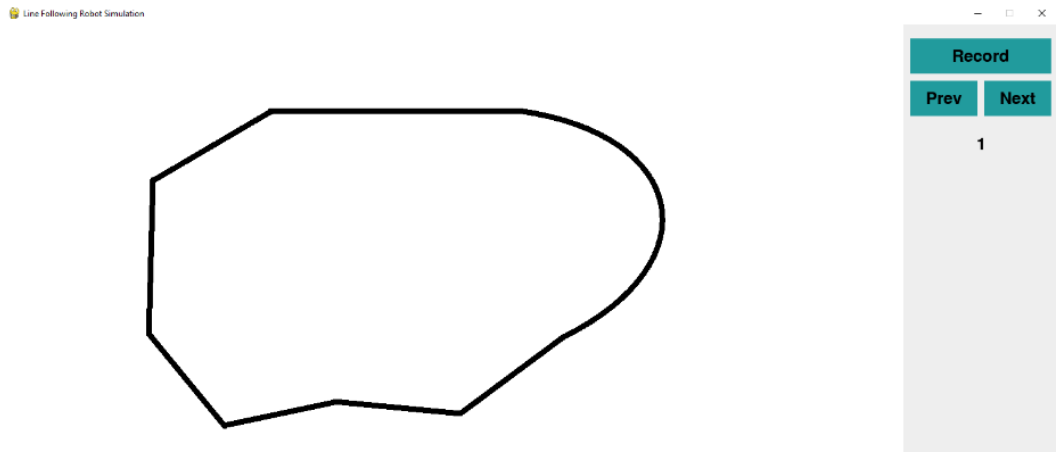

Рисунок 4.5 – Загальне вікно симулятора

Кнопка Record відповідає за записування даних з сенсорів, траєкторії руху платформи та швидкості моторів для формування навчальної вибірки. Кнопки Prev та Next дозволяють переходити до попереднього та наступного треків відповідно. Цифра під кнопками відображає номер активного треку.

4.2 Генерування навчальної вибірки

Для генерації навчальної вибірки для нейромережевих алгоритмів керування, було розроблено скрипт, що генерує задану кількість випадкових треків довільної складності (Рис. 4.6). Під складністю треку в даному випадку розуміється кількість відрізків, їх середня довжина, та кількість різких поворотів (> 60°).

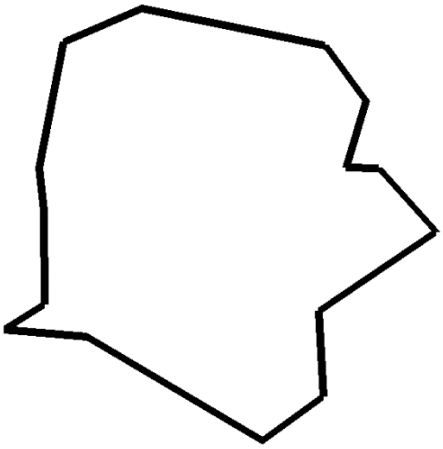

Рисунок 4.6 – Приклад згенерованого складного треку

Навчальну вибірку було згенеровано на основі 30 треків різного розміру та складності. Для досягнення найкращого результату, товщина треку повинна бути більшою, ніж відстань між двома інфрачервоними сенсорами. Значення цільової функції (потужності моторів) отримані з моделі керування на основі PID-регулятора. Було записано рух моделі контролю на кожному треку, по 5 кіл за часовою стрілкою і проти. Значення з кожного треку записано в окремий файл формату .csv. Загальна вибірка, що містить дані з усіх файлів, складається з 81071 записів – 64856 на навчальній вибірці і 16215 на тестовій.

Для забезпечення адекватної реакції системи на різкі повороти, було виставлено коригувальні коефіцієнти для інфрачервоних датчиків. Іншими словами, вплив значення з датчику на кутову швидкість платформи прямо пропорційний його відстані від центру платформи. Отримано наступний вектор пріоритетів для сенсорів від першого до останнього: [15, 12, 6, 2, 2, 6, 12, 15].

Було визначено оптимальні коефіцієнти PID-регулятора:

$$\begin{aligned} \mathrm{Kp} &= 0.025 \\ \mathrm{Ki} &= 0.000625 \\ \mathrm{Kd} &= 0.000025 \end{aligned} \tag{4.1}$$

Розмір вікна симуляції становить 1280x720 пікселів, відповідно, розмір треку обмежений цими значеннями.

4.3 Навчання нейромереж та порівняння моделей контролю

Відповідно до вимог до розроблюваної системи, згенерована вибірка має вхідний вектор розмірністю 12, що відповідає розмірності даних з датчиків платформи, та вихідний вектор розмірністю 2. Завдяки тому, що здійснюється керування потужністю моторів, вихідні дані нормовані, тобто лежать в діапазоні $[0; 1]$, що позитивно впливає на результати навчання.

Побудовано KAN мережу з архітектурою $\mathrm{KAN} = [12,2]$, кількість кроків навчання $\mathrm{step} = 200$, швидкість навчання $\mathrm{lr} = 1$, порядок сплайну $k = 3$ та кількість грід-інтервалів $G = 3$ Отримано значення функції втрат (Рис. 4.7).

```
| train_loss: 2.24e-02 | test_loss: 2.24e-02 | reg: 4.92e+00
```

Рисунок 4.7 – Значення функції втрат $\mathrm{KAN} = [12,2]$

Для даної системи, такі значення функції втрат занадто великі, оскільки значення цільової функції нормовані. Проблему помітно на графіку залежності величини середньоквадратичної похибки від кроку навчання (Рис. 4.8).

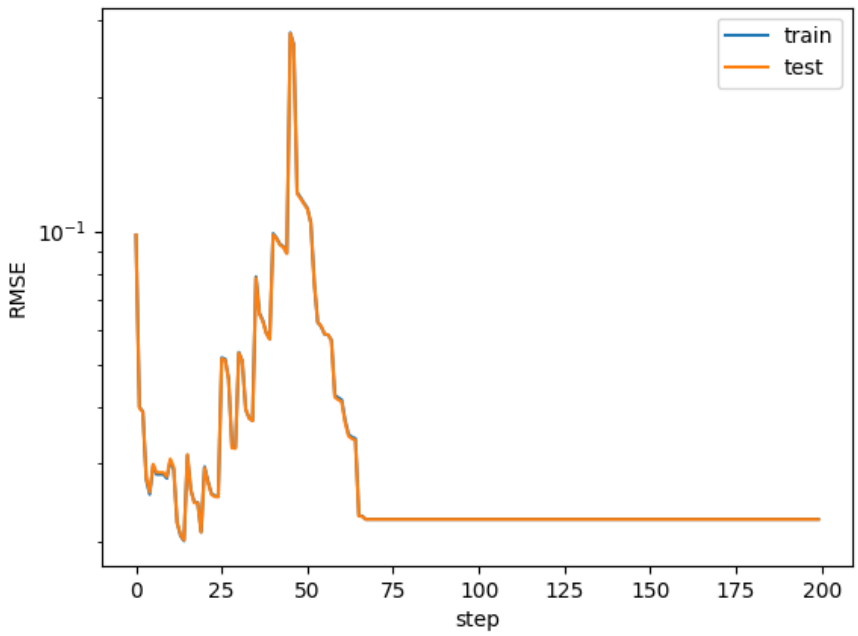

Рисунок 4.8 – Залежність похибки від часу $\mathrm{KAN} = [12,2]$

Отримані результати свідчать про те, що побудована мережа занадто «проста» та не може узагальнити вхідні дані з достатньою точністю. Необхідно додати прихований шар. Архітектура нової мережі KAN = [12,2,2]. Додання прихованого шару покращило модель, зменшивши значення функції втрат на один порядок (Рис. 4.9).

```
| train_loss: 2.10e-03 | test_loss: 2.15e-03 | reg: 9.67e+00
```

Рисунок 4.9 – Покращення функції втрат KAN = [12,2,2]

Покращення також помітно на графіку середньоквадратичної похибки (Рис. 4.10).

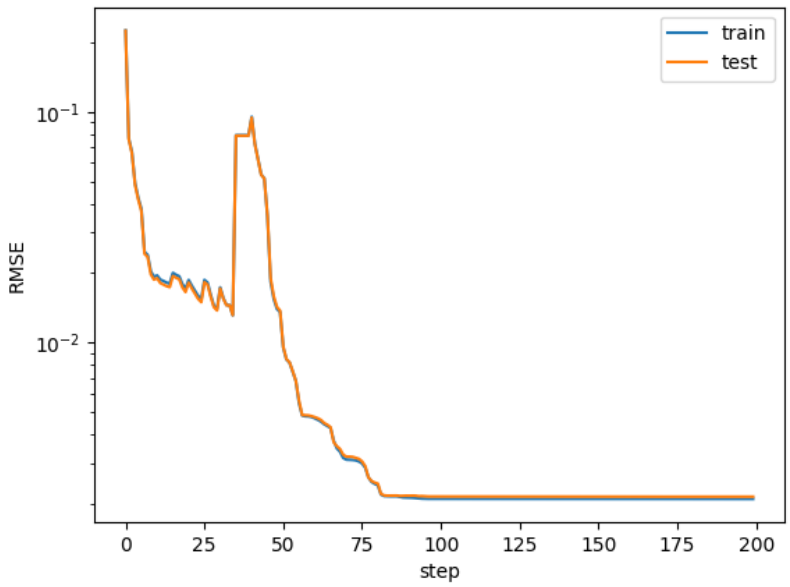

Рисунок 4.10 – Залежність похибки від часу KAN = [12,2,2]

Крива динаміки навчання виходить на плато приблизно на дев'яностому кроці навчання. Доцільно збільшити швидкість навчання моделі.

Навчену KAN мережу зі швидкістю навчання lr = 1,2 (Рис. 4.11).

```
| train_loss: 1.50e-03 | test_loss: 1.42e-03 | reg: 1.04e+01 |
```

Рисунок 4.11 – Значення функції втрат KAN = [12,2,2]

Функцію втрат зменшено майже вдвічі. Процес навчання також став більш ефективним, з виходом на плато після кроку 130 (Рис. 4.12).

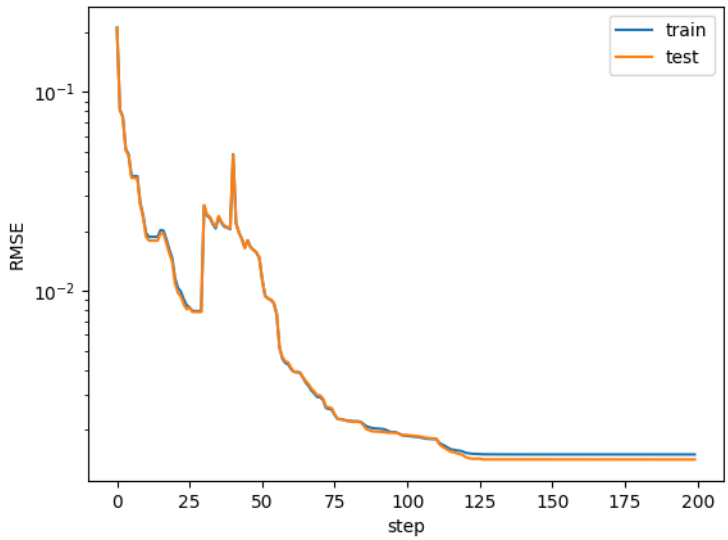

Рисунок 4.12 – Залежність похибки від часу KAN = [12,2,2]

Зміна інших гіперпараметрів, таких як: порядок сплайну k, та кількість грід-інтервалів G не призвели до покращення результатів навчання.

Навчена мережа тепер має достатню точність для застосування на практиці та порівняння з PID-регулятором (Рис. 4.13).

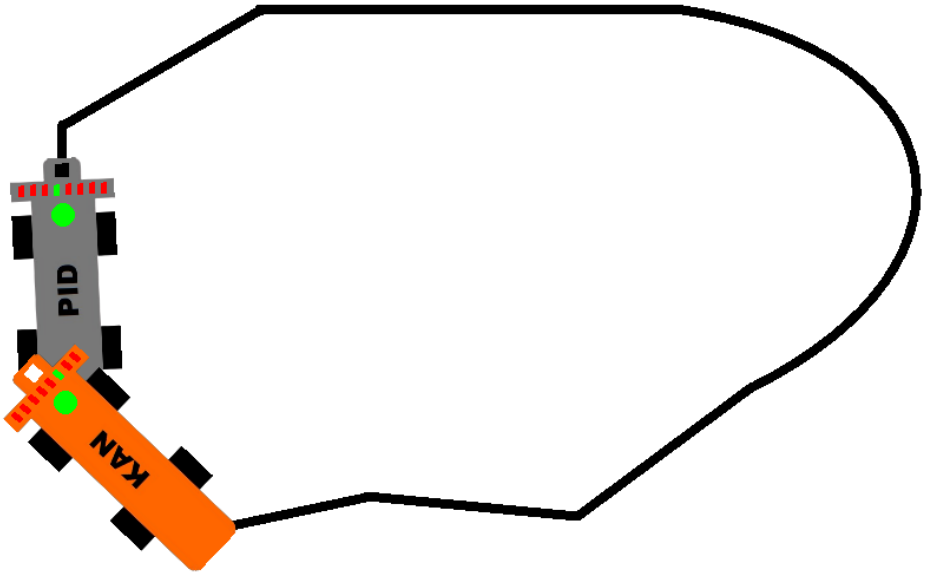

Рисунок 4.13 – Порівняння моделі KAN з PID

Помаранчева платформа керується KAN мережею, сіра – PID-регулятором. Проведений експеримент в середовищі симуляції демонструє, що точність мережі достатня для виконання поставленого завдання, а саме слідування за лінією. Однак, присутні незначні осциляції платформи, що призводить до витрат енергії на непотрібні повороти та знижує середню швидкість платформи. Це пояснює відставання від платформи під керуванням PID-регулятору, враховуючи, що платформи стартували одночасно з однієї точки.

Відображено архітектуру побудованої мережі у вигляді графу (Рис. 4.14).

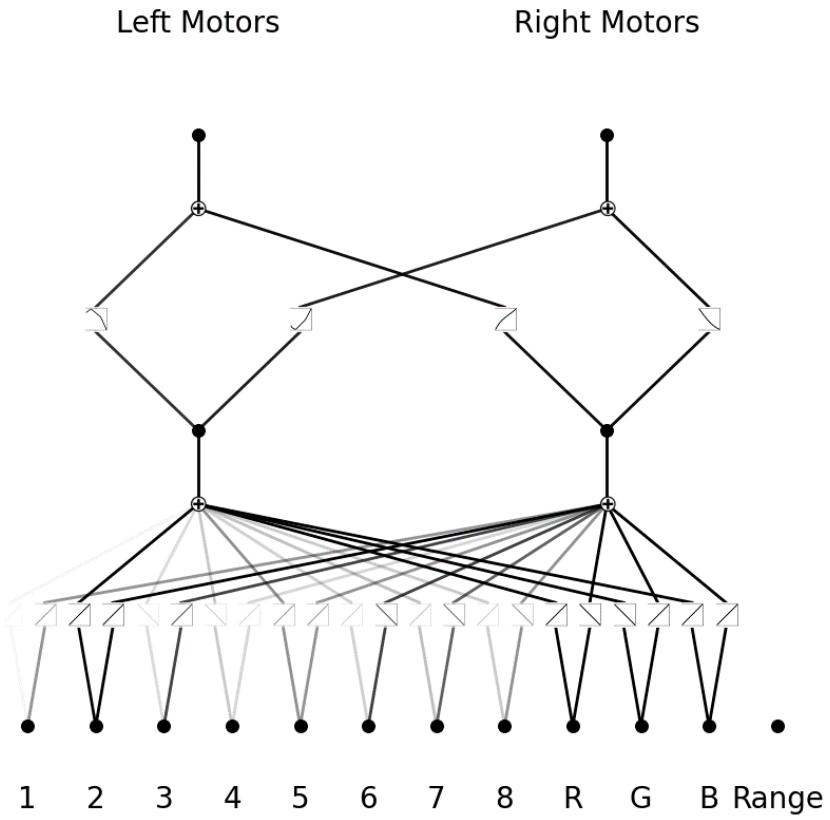

Рисунок 4.14 – Архітектура KAN мережі

З графа чітко видно, що значний вклад в значення цільової функції вносять дані з датчику кольору, але лише на основі цих даних неможливо зробити висновок про необхідний напрямок повороту платформи. Вибірка складалася з треків з великою кількістю поворотів, тому значний вплив мають інфрачервоні сенсори, віддалені від центру платформи. Даних про різкі повороти було мало, відносно всієї вибірки, тому крайні інфрачервоні сенсори мають найменший

вплив. Прихований шар вносить нелінійність в процес керування платформою, що значно покращує траєкторію руху, роблячи її більш плавною. В згенерованій вибірці відсутні перешкоди, тому дані з датчику відстані ніяк не впливають на значення цільової функції та не беруть участі в процесі керування.

Побудовано MLP мережу з аналогічною архітектурою $\mathrm{MLP} = [12,2,2]$, кількість кроків навчання $\mathrm{step} = 200$, швидкість навчання $\mathrm{lr} = 1{,}2$ (Рис. 4.15).

```
| train_loss: 1.47e-03 | test_loss: 1.48e-03 | reg: 1.13e+01
```

Рисунок 4.15 – Значення функції втрат $\mathrm{MLP} = [12,2,2]$

Значення функції втрат на навчальній та тренувальній вибірках MLP мережі співставні з KAN, при цьому швидкість навчання значно вища, але модель швидко виходить на плато (Рис. 4.16).

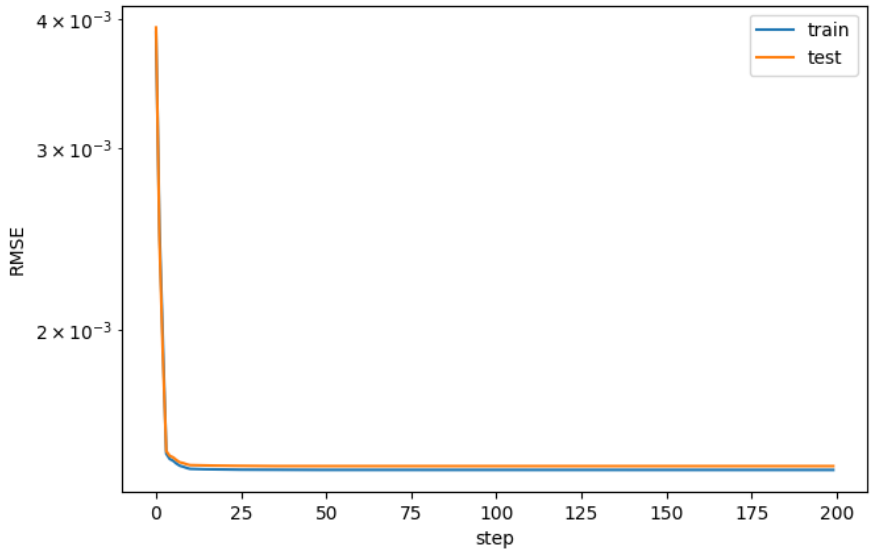

Рисунок 4.16 – Залежність похибки від часу $\mathrm{MLP} = [12,2,2]$

При заданих гіперпараметрах, MLP мережа приходить до субоптимального рішення за 15 кроків і значно не покращує значення середньоквадратичної похибки при подальшому навчанні. Але, на практиці, дана модель керування ігнорує дані з датчиків та зривається з треку при повороті (Рис. 4.17).

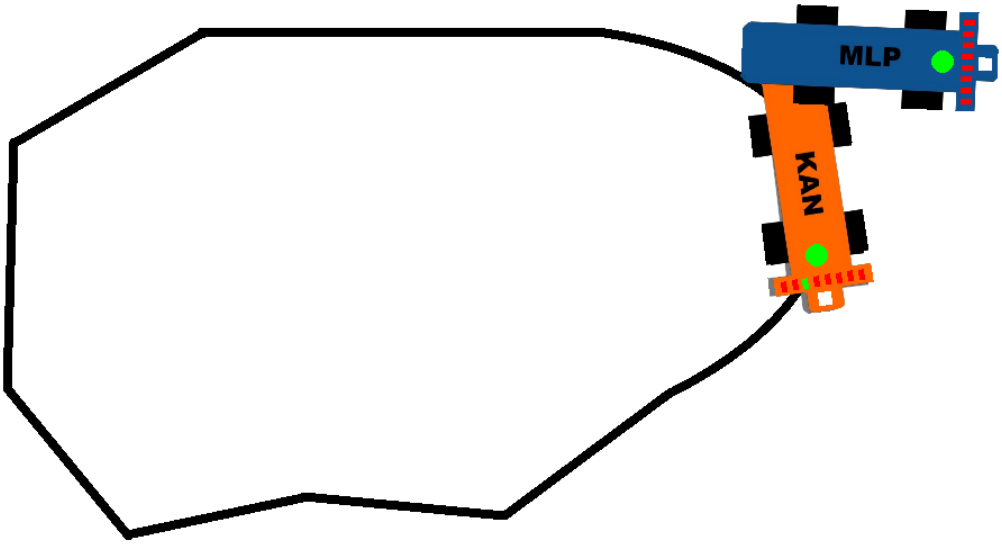

Рисунок 4.17 – Порівняння моделей керування

Помітне недонавчання MLP мережі через недостатню кількість параметрів та велику вибірку. Побудовано мережу з архітектурою MLP = [12,10,7,2] (Рис. 4.18-4.19).

```
| train_loss: 3.80e-04 | test_loss: 3.81e-04 | reg: 4.94e+01
```

Рисунок 4.18 – Значення функції втрат MLP = [12,10,7,2]

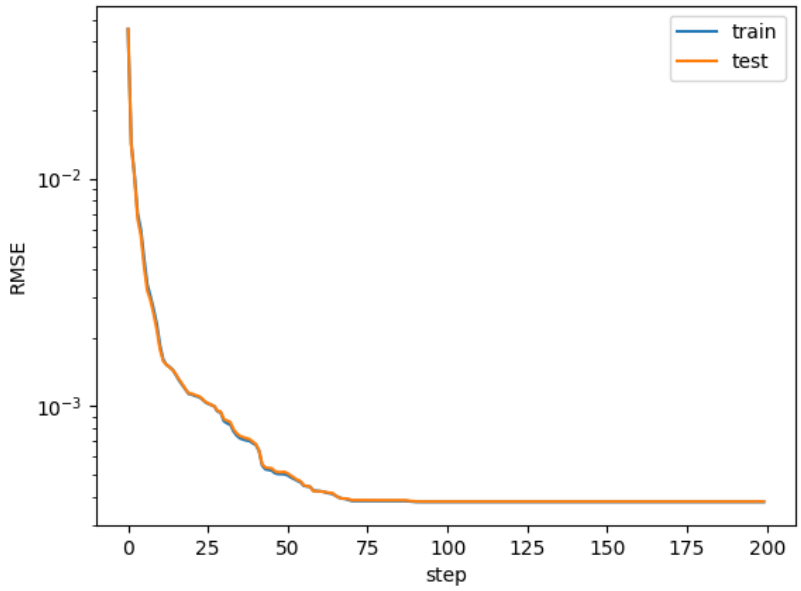

Рисунок 4.19 – Залежність похибки від часу MLP = [12,10,7,2]

Мережа з даною архітектурою значно краще узагальнює вибірку, значення функції втрат менше на один порядок. Процес навчання тривав помітно довше, величина середньоквадратичної похибки зменшувалась приблизно до дев'яностого кроку навчання. Отже, підібрано оптимальну складність мережі, при якій вдається уникнути проблем недонавчання та перенавчання MLP мережі. Архітектуру мережі відображено на графі (Рис. 4.20).

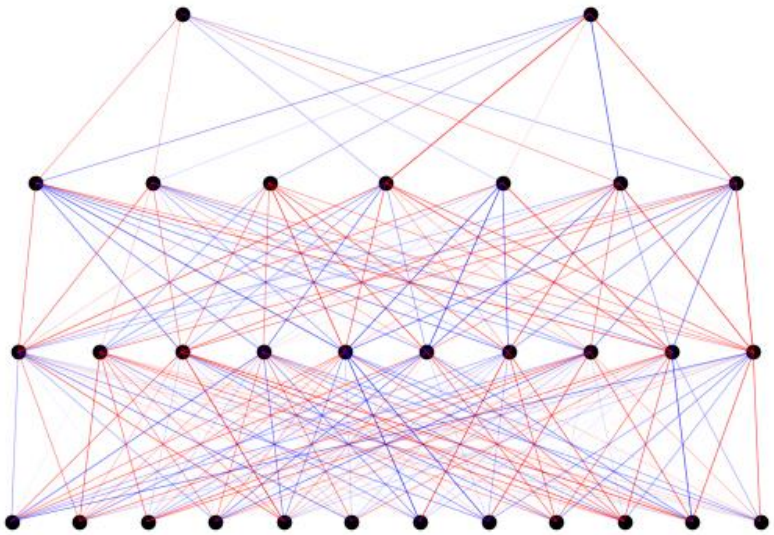

Рисунок 4.20 – Архітектура MLP мережі

Поведінка даної моделі керування та траєкторія руху платформи значно ближче до моделі PID-контролеру, ніж модель MLP = [12,2,2] (Рис. 4.21).

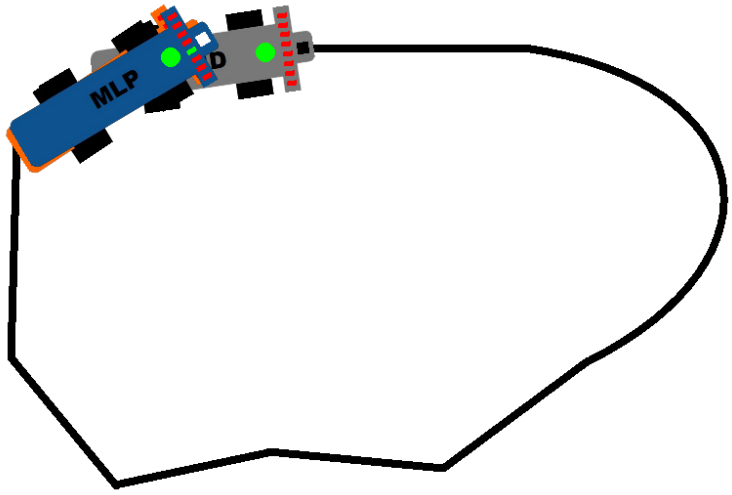

Рисунок 4.21 – Порівняння PID, MLP та KAN моделей

Модель MLP (синя платформа) подібна до KAN, але гірше повторює повороти. На складних треках з різкими поворотами, різниця між моделями більш помітна (Рис. 4.22-4.23).

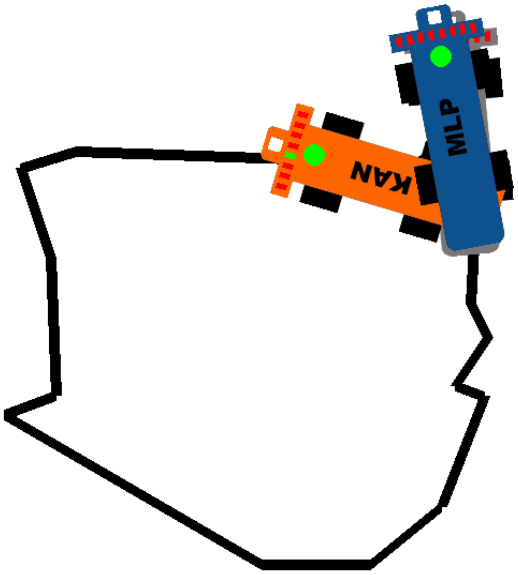

Рисунок 4.22 – Порівняння моделей на складних треках

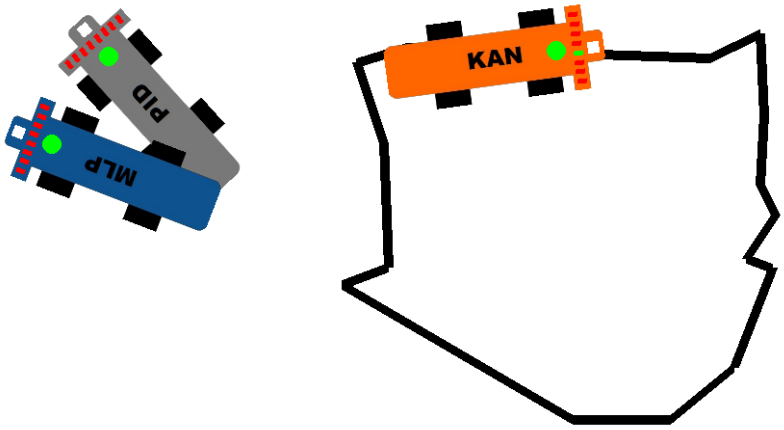

Рисунок 4.23 – Порівняння моделей на складних треках

Трек містить один складний поворот як при русі по часовій стрілці, так і проти. Серед моделей керування, лише KAN здатна стабільно проходити даний трек в обох напрямках, тоді як PID та MLP зазвичай зриваються на першому або другому колі. Така особливість зумовлена нелінійністю на вихідному шарі KAN

моделі, що дозволяє краще узагальнювати вхідні дані. Хоча MLP також використовує нелінійну функцію активації (сигмоїду), вона гірше апроксимує цільову функцію в даній задачі.

Проведені експерименти підтверджують раніше висунуті тези про те, що KAN моделі навчаються довше ніж MLP, проте потребують значно меншої кількості параметрів. Враховуючи простоту побудованої KAN моделі, розмір навчальної вибірки можна зменшити без суттєвого погіршення результатів. Для перевірки даної гіпотези, було згенеровано вибірку, що складається з 12832 записів, яка була розбита на навчальну та тестову вибірку з 10265 та 2567 записів відповідно. Навчено KAN мережу (Рис. 4.24).

```
| train_loss: 3.32e-04 | test_loss: 3.77e-04 | reg: 6.85e+00 |
```

Рисунок 4.24 – Значення функції втрат KAN на меншій вибірці

Значення функції втрат для KAN мережі на меншій вибірці знизилось на порядок, що підтверджує висунуту гіпотезу про непогіршення результатів навчання. Функція втрат на тестовій вибірці майже не відрізняється, що свідчить про відсутність проблеми перенавчання. Також покращився графік залежності значення середньоквадратичної похибки від кроку навчання (Рис. 4.25).

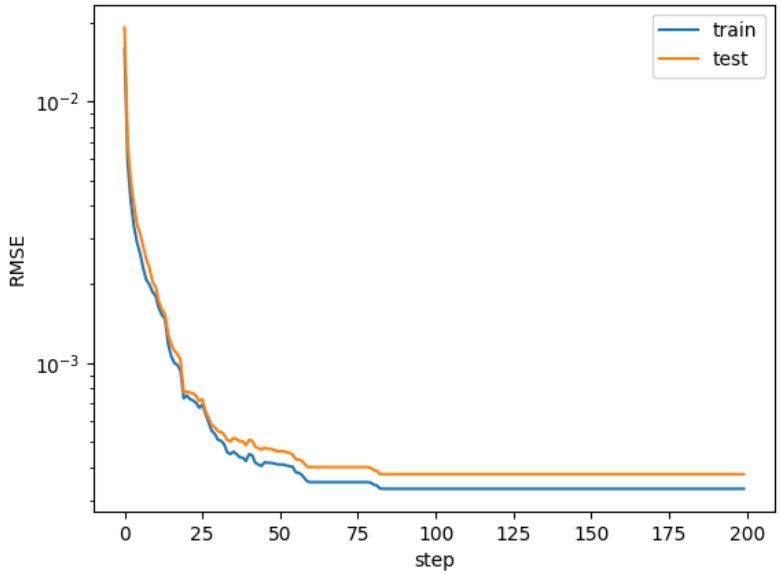

Рисунок 4.25 – Залежність похибки від часу на меншій вибірці

Для моделі MLP результат навпаки погіршився (Рис. 4.26-4.27).

```
| train_loss: 2.63e-03 | test_loss: 3.12e-03 | reg: 5.26e+01 |
```

Рисунок 4.26 – Значення функції втрат MLP на меншій вибірці

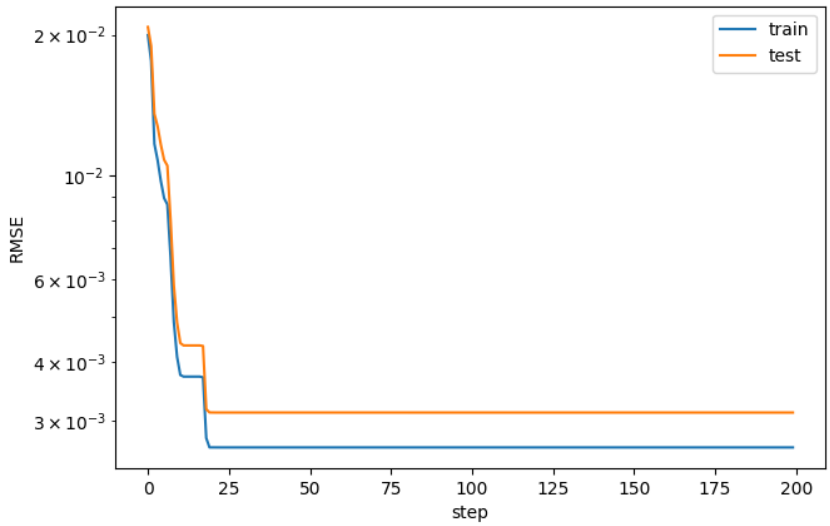

Рисунок 4.27 – Залежність похибки від часу на меншій вибірці

Значення похибки на тестовій вибірці помітно вище, мережа застрягла в локальному мінімумі.

Моделі керування на меншій вибірці перевірено на практиці (Рис. 4.28).

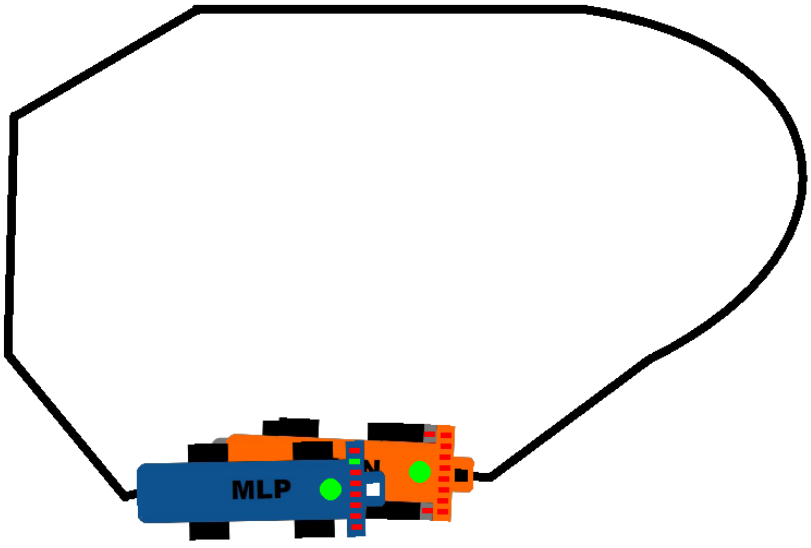

Рисунок 4.28 – Порівняння моделей керування на меншій вибірці

Модель KAN на меншій вибірці значно краще апроксимує цільову функцію. Осциляції відсутні, траєкторія руху більш плавна, на деяких треках модель випереджає PID модель та краще проходить складні ділянки, на яких PID модель швидко зривається з треку. MLP, навчена на даній вибірці, незастосовна. Присутні суттєві осциляції, різкі рухи, модель неспроможна втримати трек та часто зривається на поворотах. Така поведінка пояснюється тим, що вибірка здебільшого містить дані про рух по прямій лінії, даних про повороти недостатньо, однак даних про повороти вліво було більше, тому модель краще справляється з рухом проти годинникової стрілки (Рис. 4.29).

Рисунок 4.29 – Рух проти годинникової стрілки

Іншою важливою особливістю KAN моделей є адаптація до змін в навколишньому середовищі. PID модель не адаптується до змін, тому необхідно змінити модель керування таким чином, щоб вона змінювала швидкість платформи в залежності від відстані до перешкод. До навчальної вибірки було додано дані про відстань до перешкод та максимальну швидкість, в залежності від цієї відстані. KAN мережа не втрачає здатності проходити трек після донавчання, тоді як MLP поводить себе непередбачувано та неадекватно реагує на перешкоди (Рис. 4.30).

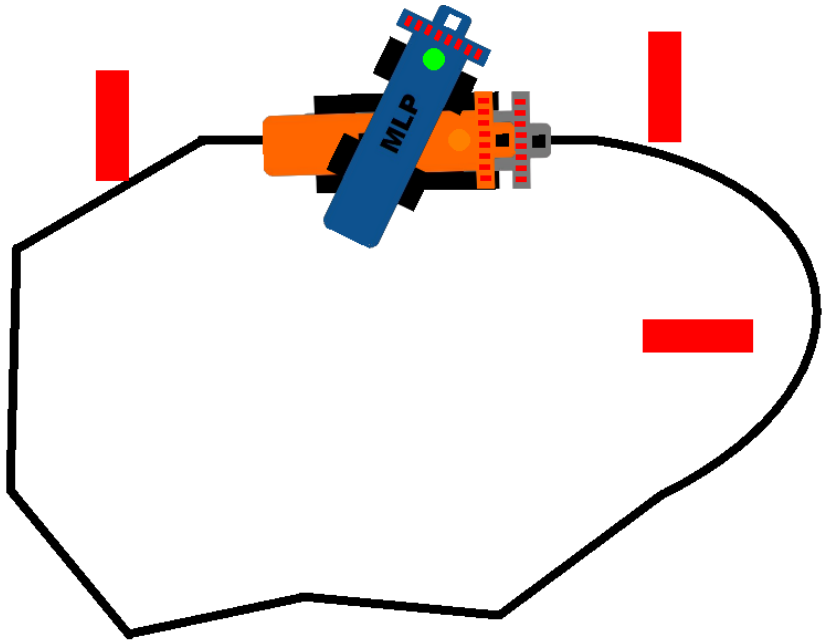

Рисунок 4.30 – Реакція моделей керування на перешкоди

Перешкоди в середовищі симуляції відображено у вигляді червоних прямокутників, що довільно рухаються по екрану. KAN модель сповільнюється, коли датчик відстані реєструє перешкоду, та повністю зупиняється, якщо відстань до перешкоди мала. MLP модель при реєстрації перешкоди різко змінює напрямок руху та зривається з треку.

Порівняно траєкторії руху платформ на складному треку з великою кількістю поворотів (Рис. 4.31).

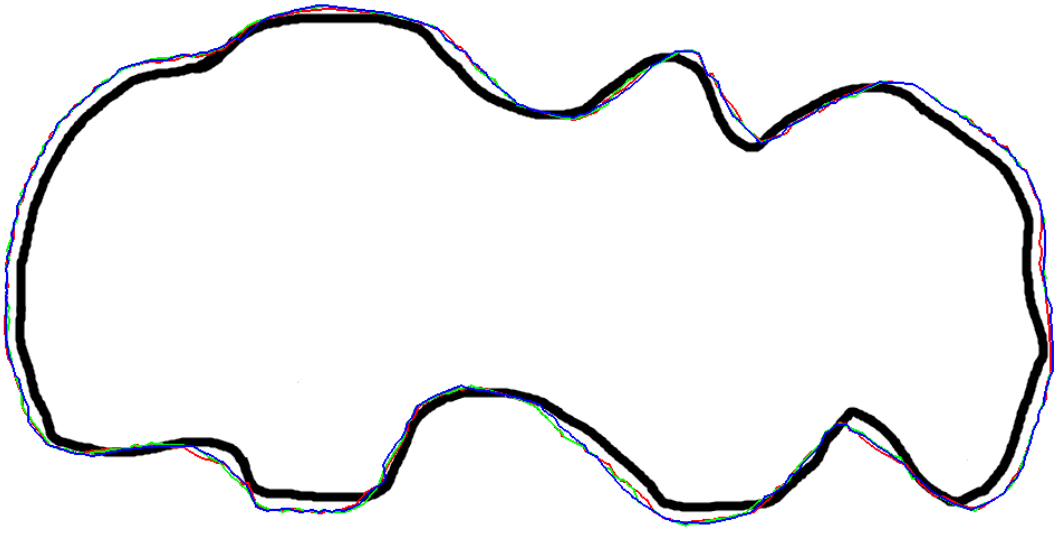

Рисунок 4.31 – Порівняння траєкторії руху платформ

Червоною лінією відображено положення центральної точки датчику лінії платформи під керуванням PID-регулятору, зеленою – KAN, та синьою – MLP. Як видно, траєкторії збігаються майже повністю та добре наближають оригінальний трек. Осциляції мінімальні, KAN мережа краще повторює траєкторію PID моделі. Також відображено положення точки обертання – центру платформи (Рис. 4.32).

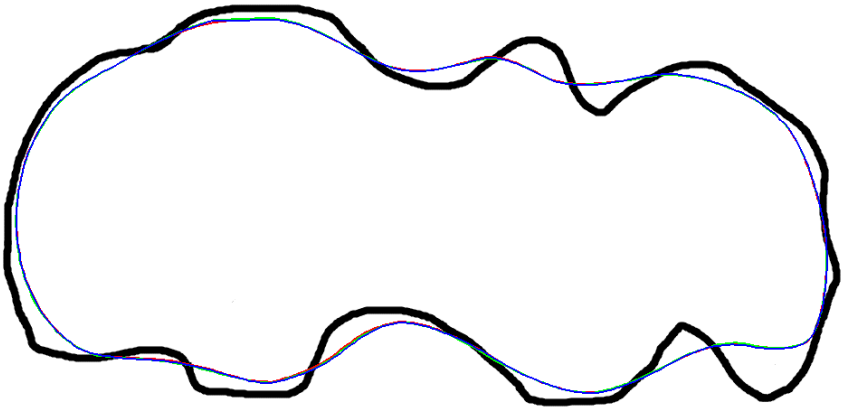

Рисунок 4.32 – Траєкторія точки обертання платформ

Осциляції переднього краю платформ не відображаються на точках обертання, траєкторії центрів платформ плавно повторюють трек.

Порівняно значення потужності обертання моторів лівої частини платформи, отримані з моделей керування (Рис. 4.33).

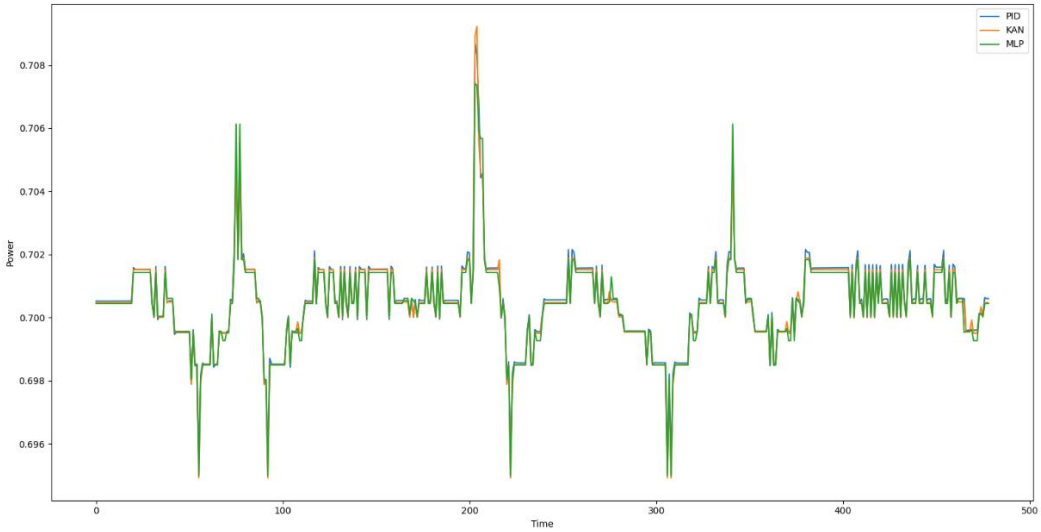

Рисунок 4.33 – Команди на мотори від моделей керування

Команди на обертання моторів від нейромережевих моделей керування здебільшого повторюють команди PID-регулятору, але помітні трохи нижчі піки.

На основі проведених експериментів, сформовано порівняльну таблицю моделей керування (Таблиця 4.1).

Таблиця 4.1 – Порівняння моделей керування

	PID	**KAN**	**MLP**
Кількість параметрів	3	84	204
Швидкість навчання	–	43 с	20 с
Час виконання алгоритму	< 1 мс	6 мс	2 мс
Розмір вибірки	–	малий	великий
Проходження складних треків	нестабільно	стабільно	нестабільно
Адаптація до нових умов	ні	так	частково

В даному розділі було реалізовано та протестовано три підходи до побудови автоматизованої системи керування роботизованою платформою: на основі PID-регулятору, багатошарового перцептрону MLP та мережі Колмогорова-Арнольда KAN. Описано віртуальне середовище для тестування побудованих систем. Систему на основі PID-регулятору використано для генерації навчальної вибірки для нейромережевих алгоритмів. Завдяки використанню нелінійних функцій активації, нейромережеві алгоритми згладжують траєкторію руху платформи, роблячи її більш плавною. Узагальнення даних з сенсорів дозволяє їм краще проходити складні ділянки треку з різкими поворотами, на яких PID модель часто зривається. Завдяки тому, що KAN модель навчає не зв’язки між вузлами, а одновимірні функції активації, цільова функція апроксимується точніше, потребуючи при цьому меншої кількості параметрів та розміру навчальної вибірки, ніж MLP модель. Здатність KAN мереж до безперервного навчання дозволяє їм краще адаптуватися до змін в середовищі, що є суттєвою перевагою в автоматизованих системах керування в реальному часі.

ВИСНОВКИ

В роботі було досліджено застосування KAN мереж в автоматизованій системі керування роботизованою платформою. Описано теоретичні основи керування, розглянуто готові рішення та оцінено різні підходи. Викладено вимоги до розроблюваного рішення. Обґрунтовано прийняті програмні та апаратні рішення, детально описано особливості KAN мережі та математичні принципи, що дозволяють мережі апроксимувати багатовимірні функції з достатньою точністю. Платформу спроектовано за модульним принципом. Розроблено та виготовлено друковані плати для моделі роботизованої платформи. Описано схему підключення модулів платформи.

Розроблено віртуальне середовище симуляції руху платформи з графічним інтерфейсом за допомогою бібліотеки Pygame. Налаштовано коефіцієнти PID-регулятору для побудови лінійної моделі керування, на основі якої згенеровано навчальну вибірку для нейромережевих алгоритмів. Найкраще себе показали моделі з архітектурами $\mathrm{KAN} = [12,2,2]$ та $\mathrm{MLP} = [12,10,7,2]$, швидкість навчання була однакова і становила $\mathrm{lr} = 1{,}2$, кількість кроків $\mathrm{step} = 200$.

Проведено аналіз особливостей побудови KAN, її архітектурних переваг і можливостей для реалізації точного та адаптивного керування в умовах змінного середовища. Порівняння KAN із традиційними підходами, такими як PID-регулятори та нейронні мережі типу MLP, показало, що запропонований підхід має суттєві переваги в умовах обмежених ресурсів і складних динамічних середовищ. Найбільшим недоліком KAN мереж є відносно повільна швидкість навчання, в середньому до десяти раз повільніше, ніж MLP, але це компенсується тим, що KAN потребують меншої кількості параметрів та розміру вибірки. Завдяки тому, що навчаються одновимірні функції активації, точність апроксимації цільової функції значно вища, ніж у MLP. Нелінійність функції активації дозволяє KAN моделі керування проходити складні ділянки, де лінійний PID зазвичай зривається з треку. Головною перевагою використання

KAN в автоматизованих системах керування є можливість безперервного навчання та висока здатність до адаптації до змін в середовищі, що є важливим фактором в динамічних системах реального часу.

Технічні обмеження та швидкодія алгоритму на основі KAN є скоріше інженерною проблемою та пов’язані з неоптимальною програмною реалізацією алгоритму. Очікується значне покращення швидкості навчання KAN та збільшення складності мереж в майбутньому.

Таким чином, робота підтверджує доцільність і перспективність використання мереж Колмогорова-Арнольда в сучасних автоматизованих системах керування роботизованими платформами. Отримані результати можуть слугувати основою для подальших досліджень у цій галузі, спрямованих на розширення можливостей алгоритмів керування й інтеграцію таких систем у більш складні робототехнічні рішення. Такі дослідження, зокрема, можуть бути зосереджені на розширені можливостей керування платформою. Наприклад, наявність датчику кольору дозволяє задавати максимальну швидкість руху платформи, змінюючи колір лінії. Використання KAN в більш складних алгоритмах обробки зображень та розпізнавання об’єктів дозволяє інтеграцію цифрової камери для отримання детальної інформації про середовище, в якому знаходиться платформа, що є основою для розробки складніших систем керування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. N.S. Nise, Control Systems Engineering, 4th ed. (John Wiley and Sons, Inc., Chichester, 2004).

2. The PID Controller & Theory Explained. URL: https://www.ni.com/en/shop/labview/pid-theory-explained.html?srsltid=AfmBOoqq8HQRzm0V4XQ5iOpX2x0zNUAMupifXa5EKgq-BMFSElYULNPR (дата звернення: 20.11.2024).

3. Multilayer Perceptrons in Machine Learning: A Comprehensive Guide. URL: https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning (дата звернення: 20.11.2024).

4. Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, KAN: Kolmogorov–Arnold Networks – The NSF Institute for Artificial Intelligence and Fundamental Interactions, 2024.

5. Robotic Automation Systems. URL: https://www.roboticautomationsystems.com/company/ (дата звернення: 20.11.2024).

6. Amazon Robotics LLC. URL: https://www.aboutamazon.com/news/operations/amazon-introduces-new-robotics-solutions (дата звернення: 20.11.2024).

7. Temerland. URL: https://temerland.com (дата звернення: 20.11.2024).

8. Python. URL: https://uk.wikipedia.org/wiki/Python (дата звернення: 20.11.2024).

9. Pygame. URL: https://www.pygame.org/wiki/GettingStarted (дата звернення: 20.11.2024).

10. Welcome to PyYAML. URL: https://pyyaml.org (дата звернення: 20.11.2024).

11. A Beginner-friendly Introduction to Kolmogorov Arnold Networks (KAN). URL: https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-arnold-networks-kan/ (дата звернення: 20.11.2024).

12. QTR-8A and QTR-8RC Reflectance Sensor Array User's Guide. URL: https://www.pololu.com/docs/0J12 (дата звернення: 20.11.2024).

13. ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs. URL: https://www.ti.com/lit/ds/symlink/ads1113.pdf?ts=1734879414682&ref_url=https%253A%252F%252Fwww.google.com%252F (дата звернення: 20.11.2024).

14. Ultrasonic Ranging Module HC - SR04. URL: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf (дата звернення: 20.11.2024).

15. TCS230 Color Light-To-Frequency Converter. URL: https://www.alldatasheet.com/datasheet-pdf/view/239919/TAOS/TCS230.html (дата звернення: 20.11.2024).

16. EasyEDA. URL: https://easyeda.com (дата звернення: 20.11.2024).

17. JLCPCB. URL: https://jlcpcb.com (дата звернення: 20.11.2024).

18. Raspberry Pi Zero 2W. URL: https://evo.net.ua/mikrokomputer-raspberry-pi-zero-2-w/?srsltid=AfmBOopRMEUGmB30fUVZPJknxKROxLtiRNwvzWHE28poAYLbxBpY-Lf3 (дата звернення: 20.11.2024).

19. L293x Quadruple Half-H Drivers. URL: https://www.ti.com/lit/ds/symlink/l293.pdf (дата звернення: 20.11.2024).

20. Proteus. URL: https://www.labcenter.com (дата звернення: 20.11.2024).

ДОДАТОК А
ВИХІДНИЙ КОД СЕРЕДОВИЩА СИМУЛЯЦІЇ

1. Абстрактний контролер:

```python
from abc import ABC, abstractmethod

class MovementController(ABC):
    @abstractmethod
    def get_controlls(self, sensors_data):
        pass
```

2. MLP контролер:

```python
from Classes.Controllers.MovementController import *
from kan import *
from kan.MLP import MLP

class MLPController(MovementController):
    def __init__(self, width, seed, dataset, steps=20, lr=1):
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.MLP = MLP(width=width, seed=seed, device=device)
        self.dataset = dataset
        self.steps = steps
        self.lr = lr

    def train(self):
        self.MLP.fit(self.dataset, lr=self.lr, steps=self.steps)

    def get_controlls(self, data):
        data = [data]
        data.append([0] * 12)
        data = torch.tensor(data).float().to("cuda")
        controls = self.MLP.forward(data)

        return (float(controls[0][0]), float(controls[0][1]))
```

3. KAN контролер:

```python
from Classes.Controllers.MovementController import *
from kan import *

class KANController(MovementController):
```

```
    def __init__(self, width, grid, k, seed, dataset, steps=20, lr=1):
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.KAN = KAN(width=width, grid=grid, k=k, seed=seed, device=device)
        self.dataset = dataset
        self.steps = steps
        self.lr = lr

    def train(self):
        self.KAN.fit(self.dataset, lr=self.lr, steps=self.steps)

    def get_controlls(self, data):
        data = [data]
        data.append([0] * 12)
        data = torch.tensor(data).float().to("cuda")
        controls = self.KAN.forward(data)

        return (float(controls[0][0]), float(controls[0][1]))
```

4. PID контролер:

```
from Classes.Controllers.MovementController import *
class PIDController(MovementController):
    def __init__(self, Kp, Ki, Kd):
        """
        Initialize the PID controller.
        Args:
            Kp: Proportional coeficient.
            Ki: Integral coeficient.
            Kd: Derivative coeficient.
            csv_logger (CsvLogger): CsvLogger object from higher controller. Default to None
            name (str): Name for csv logging. Default to "pid"
        """

        self.kp = Kp
        self.ki = Ki
        self.kd = Kd
        self.prev_err = 0
        self._P = 0
        self._I = 0
        self._D = 0
        self.integral = 0
        self.dt = 1 / 60

    def get_measurement(self, sensors_data):
        data = sensors_data[:8]
        prioriti_table = [15, 12, 6, 2, 2, 6, 12, 15]
        data = [a * b for a, b in zip(data, prioriti_table)]

        return sum(data[:4]) - sum(data[4:8])
```

```
def update(self, sensors_data):
    """
    Returns the updated value of a controlled signal.
    Args:
        err: Measured error.
    Returns:
        float: New signal value.
    """

    err = self.get_measurement(sensors_data)
    P = err
    self.integral += err * self.dt
    D = (err - self.prev_err) / self.dt
    self.prev_err = err

    self._P = P * self.kp
    self._I = self.integral * self.ki
    self._D = D * self.kd

    return self._P + self._I + self._D

def get_controlls(self, sensors_data):
    value = self.update(sensors_data)
    value = value / 100
    base_power = self.obstacle_avoidance(sensors_data[-1])
    if base_power == 0:
        return (0, 0)

    return (max(0, min(1, base_power + value)), max(0, min(1, base_power - value)))

def obstacle_avoidance(self, distance):
    base_power = 0.7

    if distance > -1:
        if distance < 100:
            base_power = 0
        elif distance < 300:
            base_power = 0.3

    return base_power

def reset(self):
    self._P = 0
    self._I = 0
    self._D = 0
    self.integral = 0
    self.prev_err = 0
```

5. Клас кнопки:

```
import pygame

class Button:
    def __init__(self, x, y, width, height, color, event, text='', text_color=(0,0,0)):
        self.rect = pygame.Rect(x, y, width, height)
        self.color = color
        self.text = text
        self.text_color = text_color
        self.font = pygame.font.Font(None, 36)  # Default font, size 36
        self.event = event

    def draw(self, screen):
        # Draw the button rectangle
        pygame.draw.rect(screen, self.color, self.rect)

        # Draw the text (if any)
        if self.text:
            text_surf = self.font.render(self.text, True, self.text_color)
            text_rect = text_surf.get_rect(center=self.rect.center)
            screen.blit(text_surf, text_rect)

    def is_clicked(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            if self.rect.collidepoint(event.pos):
                return True
        return False

    def on_click(self):
        self.event(self)
```

6. Графічний інтерфейс користувача:

```
from Classes.GUI.Button import *

BUTTON_COLOR = (33, 155, 157)
BUTTON_CLICKED = (255, 128, 0)
GREY = (238, 238, 238)

class GUI:
    def __init__(self, screen, dm, map_image):
        self.screen = screen
        self.dm = dm
        self.buttons = []
        self.i = 1
        self.map_image = map_image
        self.r = None
        self.point = (0, 0)
```

```
        self.buttons.append(Button(1290, 20, 200, 50, BUTTON_COLOR, self.record_map, "Record"))
        self.buttons.append(Button(1290, 80, 95, 50, BUTTON_COLOR, self.prev_map, "Prev"))
        self.buttons.append(Button(1395, 80, 95, 50, BUTTON_COLOR, self.next_map, "Next"))

    def record_map(self, button):
        self.dm.is_recording = not self.dm.is_recording
        button.text = "Stop" if self.dm.is_recording else "Record"
        self.point = (self.r.x, self.r.y)

    def prev_map(self, button):
        self.i -= 1
        self.map_image = pygame.image.load(f"Dataset/Images/{self.i}.png")

    def next_map(self, button):
        self.i += 1
        self.map_image = pygame.image.load(f"Dataset/Images/{self.i}.png")

    def draw_pivot(self, point):
        pygame.draw.circle(self.screen, BUTTON_CLICKED, point, 8)

    def on_click_event(self, event):
        for b in self.buttons:
            if b.is_clicked(event):
                b.on_click()

    def draw_GUI(self):
        self.screen.fill(GREY)
        self.screen.blit(self.map_image, (0, 0))

        self.dm.record_data((self.r.x, self.r.y), self.r.get_sensors_data(),
(self.r.left_motor.power, self.r.right_motor.power), str(self.i))

        if self.dm.is_recording:
            self.draw_pivot(self.point)

        for b in self.buttons:
            b.draw(self.screen)

        font = pygame.font.Font(None, 36)
        text_surf = font.render(str(self.i), True, (0,0,0))
        text_rect = text_surf.get_rect(center=(1390, 170))
        self.screen.blit(text_surf, text_rect)
```

7. Клас перешкоди:

```
import pygame

class Obstacle:
```

```
    def __init__(self, screen, sonic_sensors, center, size, width, height):
        self.screen = screen
        self.sensors = sonic_sensors

        self.width = width
        self.height = height

        self.register_rect(center, size)


    def draw(self):
        pygame.draw.rect(self.screen, (255, 0, 0), self.rect)

    def register_rect(self, center, size):
        self.rect = pygame.Rect(0, 0, size[0], size[1])
        self.rect.center = center

        for sensor in self.sensors:
            sensor.obstacles.append(self.rect)

    def set_speed(self, x_speed, y_speed):
        self.x_speed = x_speed
        self.y_speed = y_speed

    def move(self):
        x = self.rect.center[0]
        y = self.rect.center[1]

        x += self.x_speed
        y += self.y_speed

        if x > self.width or x < 0:
            self.x_speed *= -1
        if y > self.height or y < 0:
            self.y_speed *= -1

        self.rect.center = (x, y)
```

8. Клас-обгортка перешкод:

```
import pygame
from Classes.Obstacles.Obstacle import *

class ObstacleWrapper:
    def __init__(self, screen, sonic_sensors, screen_size):
        self.screen = screen
        self.sensors = sonic_sensors
        self.screen_size = screen_size
```

```
        self.obstacles = []

    def register_obstacle(self, center, size):
        obstacle = Obstacle(self.screen, self.sensors, center, size, self.screen_size[0],
self.screen_size[1])
        self.obstacles.append(obstacle)

        return obstacle

    def remove_obstacle(self, obstacle):
        self.obstacles.remove(obstacle)

    def draw(self):
        for i in self.obstacles:
            i.draw()

    def move(self):
        for i in self.obstacles:
            i.move()
```

9. Абстрактний датчик:

```
import numpy as np
import pygame
from abc import ABC, abstractmethod
import math

class Sensor(ABC):
    """Line sensor. Composes the Robot class."""

    def __init__(self, sensor_relative_position, robot_initial_position, sensor_width,
sensor_height, map, map_image):
        """Sensor class constructor. Positions the sensor relative to the robot's initial
position.

        Args:
            sensor_relative_position (tuple): sensor position (x, y) relative to the robot, in
meters.
            robot_initial_position (tuple): robot initial position (x, y, heading), in meters and
radians.
        """

        # Adds the relative position vector (rotated) to the robot's initial position
        self.x = robot_initial_position[0] + sensor_relative_position[0]
        self.y = robot_initial_position[1] + sensor_relative_position[1]
        self.relative_x = sensor_relative_position[0]
        self.heading = robot_initial_position[2]
        self.map = map
        self.map_image = map_image
```

```
        self.data = 0
        self.sensor_width = sensor_width
        self.sensor_height = sensor_height
        self.color = (255, 0, 0)

    def update_position(self, robot_position, offset=1):
        """Updates the sensor's position according to the robot's position.

        Args:
            robot_position (tuple): robot current position (x, y, heading), in meters and
radians.
        """

        line_center = self.rotate_point((robot_position[0] + self.relative_x, robot_position[1]),
robot_position, robot_position[2])

        # Calculate the perpendicular angle to the radius
        perpendicular_angle = robot_position[2] + math.pi / 2

        # Calculate the positions of the rectangles along the perpendicular line
        x = line_center[0] + offset * math.cos(perpendicular_angle) - robot_position[0]
        y = line_center[1] + offset * math.sin(perpendicular_angle) - robot_position[1]

        # Adds the relative position vector to the robot's position to obtain the sensor's real
position
        self.x = robot_position[0] + x
        self.y = robot_position[1] + y
        self.heading = robot_position[2]

    def rotate_point(self, point, pivot, angle):
        """Rotate a point around a pivot by a given angle."""

        s, c = math.sin(angle), math.cos(angle)
        px, py = point[0] - pivot[0], point[1] - pivot[1]
        xnew = px * c - py * s
        ynew = px * s + py * c
        return xnew + pivot[0], ynew + pivot[1]

    @abstractmethod
    def read_data(self):
        """Reads the sensor data.

        Args:
            map_image (pygame.Surface): arena image.
        """
        pass

    @abstractmethod
    def set_sensor_color(self):
        pass
```

```
    def draw_sensor(self, color):
        """Draws a sensor on the screen.

        Args:
            color: sensor color
        """

        sensor_rect = pygame.Rect(0, 0, self.sensor_width, self.sensor_height)
        sensor_rect.center = (self.x, self.y)
        pygame.draw.rect(self.map, color, sensor_rect)

    def is_out_of_bounds(self):
        """Checks if the object is out of bounds.

        Args:
            object (Robot or Sensor): object to be checked.

        Returns:
            bool: True if the object is out of bounds, False otherwise.
        """

        # Checks if the robot is within the arena limits
        if (self.x < 0 or
            self.x > self.map.get_width() or
            self.y < 0 or
            self.y > self.map.get_height()):
            return True
        else:
            return False

    def set_map_image(self, map_image):
        self.map_image = map_image
```

10. Датчик кольору:

```
from Classes.Sensors.Sensor import *
import math

class ColorSensor(Sensor):

    def read_data(self):
        """Reads the sensor data.

        Args:
            map_image (pygame.Surface): arena image.
        """

        if self.is_out_of_bounds():
            return self.data
```

```
        # Sensor reads the color of the arena pixel at the sensor position
        color = self.map_image.get_at((int(self.x), int(self.y)))[:-1]
        self.set_sensor_color()

        self.data = color

        return self.data

    def draw_sensor(self):
        """Draw a rotated rectangle on the screen."""
        rect = pygame.Rect(0, 0, self.sensor_width, self.sensor_height)
        rect.center = (self.x, self.y)

        # Rotate the rectangle
        rotated_surface = pygame.Surface((self.sensor_width, self.sensor_height),
pygame.SRCALPHA)
        rotated_surface.fill(self.color)
        rotated_surface = pygame.transform.rotate(rotated_surface, -math.degrees(self.heading))
        rotated_rect = rotated_surface.get_rect(center=rect.center)

        # Blit the rotated surface onto the screen
        self.map.blit(rotated_surface, rotated_rect.topleft)

    def set_sensor_color(self):
        self.color = self.data
```

11. Інфрачервоний датчик:

```
from Classes.Sensors.Sensor import *
import math

class InfraredSensor(Sensor):

    def read_data(self):
        """Reads the sensor data. The sensor returns 0 if it reads a dark color and 1 if it reads
a light color.

        Args:
            map_image (pygame.Surface): arena image.
        """

        if self.is_out_of_bounds():
            return self.data

        # Sensor reads the color of the arena pixel at the sensor position
        color = self.map_image.get_at((int(self.x), int(self.y)))[:-1]

        # Returns 1 if the color is lighter than medium gray and 0 otherwise.
```

```
        self.data = 0 if self.is_darker(color, (255/2, 255/2, 255/2)) else 1
        self.set_sensor_color()

        return self.data

    def is_darker(self, color1, color2):
        """Checks if color1 is darker than color2.

        Args:
            color1, color2 (tuple): Colors in RGB format (R, G, B).

        Returns:
            bool: True if color1 is darker than color2, False otherwise.
        """
        # Calculates the average of the RGB values of each color (grayscale)
        gray1 = sum(color1) / len(color1)
        gray2 = sum(color2) / len(color2)

        return gray1 < gray2

    def draw_sensor(self):
        """Draw a rotated rectangle on the screen."""
        rect = pygame.Rect(0, 0, self.sensor_width, self.sensor_height)
        rect.center = (self.x, self.y)

        # Rotate the rectangle
        rotated_surface = pygame.Surface((self.sensor_width, self.sensor_height),
pygame.SRCALPHA)
        rotated_surface.fill(self.color)
        rotated_surface = pygame.transform.rotate(rotated_surface, -math.degrees(self.heading))
        rotated_rect = rotated_surface.get_rect(center=rect.center)

        # Blit the rotated surface onto the screen
        self.map.blit(rotated_surface, rotated_rect.topleft)

    def set_sensor_color(self):
        if self.data == 1:
            self.color = (255, 0, 0)
        else:
            self.color = (0, 255, 0)
```

12. Датчик лінії:

```
from Classes.Sensors.InfraredSensor import *
from Classes.Sensors.Sensor import *
import math

class LineDetectorSensor(Sensor):
```

```
    def __init__(self, sensor_relative_position, spacing, robot_initial_position, map,
map_image):
        super().__init__(sensor_relative_position, robot_initial_position, 10, 5, map, map_image)

        self.data = []
        self.sensors = []
        self.spacing = spacing

        line_center = self.rotate_point((robot_initial_position[0] + sensor_relative_position[0],
robot_initial_position[1]), robot_initial_position, robot_initial_position[2])

        # Calculate the perpendicular angle to the radius
        perpendicular_angle = robot_initial_position[2] + math.pi / 2

        # Calculate the positions of the rectangles along the perpendicular line
        for i in range(8):
            offset = (i - 7 / 2) * self.spacing
            x = line_center[0] + offset * math.cos(perpendicular_angle) -
robot_initial_position[0]
            y = line_center[1] + offset * math.sin(perpendicular_angle) -
robot_initial_position[1]

            self.sensors.append(InfraredSensor((x, y), robot_initial_position, 10, 5, map,
map_image))

    def read_data(self):
        """Reads the sensor data.

        Args:
            map_image (pygame.Surface): arena image.
        """

        self.data.clear()

        for i in self.sensors:
            self.data.append(i.read_data())

        return self.data

    def draw_sensor(self):
        """Draws a sensor on the screen.

        Args:
            color: sensor color
        """

        for i in self.sensors:
            i.draw_sensor()

    def update_position(self, robot_position, offset=1):
        i = 0
```

```
        for s in self.sensors:
            offset = (i - 7 / 2) * self.spacing

            s.update_position(robot_position, offset)
            i += 1

    def set_sensor_color(self):
        pass

    def set_map_image(self, map_image):
        for i in self.sensors:
            i.set_map_image(map_image)
```

13. Датчик відстані до перешкод:

```
from Classes.Sensors.Sensor import *
from threading import Thread
import time
import math

class SonicSensor(Sensor):
    def __init__(self, sensor_relative_position, robot_initial_position, sensor_width,
sensor_height, map, map_image):
        super().__init__(sensor_relative_position, robot_initial_position, sensor_width,
sensor_height, map, map_image)

        self.obstacles = []
        self.beam_rect = pygame.Rect(sensor_relative_position[0], sensor_relative_position[1],
15, 15)
        self.beam_active = False
        self.data = -1
        self.beam_x = sensor_relative_position[0]
        self.beam_y = sensor_relative_position[1]
        watcher = Thread(target=self.scan)
        watcher.start()

    def read_data(self):
        """Reads the sensor data.

        Args:
            map_image (pygame.Surface): arena image.
        """
        self.set_sensor_color()

        return self.data

    def draw_sensor(self):
        """Draw a circle on the screen."""
```

```
            pygame.draw.circle(self.map, self.color, (self.x, self.y), self.sensor_width)

        def scan(self):
            running = True
            while running:
                self.beam_active = True
                self.beam_x = self.x
                self.beam_y = self.y

                x_speed = 100*np.cos(self.heading)/2
                y_speed = 100*np.sin(self.heading)/2

                while self.beam_active:
                    try:
                        self.launch_beam(x_speed, y_speed)
                        time.sleep(0.001)
                    except:
                        running = False
                        break

        def launch_beam(self, x_speed, y_speed):
            if self.beam_active:
                self.beam_x += x_speed
                self.beam_y += y_speed

                rect = pygame.Rect(0, 0, 50, 25)
                rect.center = (self.beam_x, self.beam_y)
                self.beam_rect = rect

                if self.find_obstacles():
                    self.data = self.get_distance(self.beam_rect.center)
                    self.beam_active = False

                    return

            if self.beam_y < 0 or self.beam_x < 0 or self.beam_y > self.map.get_height() or
    self.beam_x > self.map.get_width():
                self.data = -1
                self.beam_active = False

        def find_obstacles(self):
            for i in self.obstacles:
                if self.beam_rect.colliderect(i):
                    return True

            return False

        def get_distance(self, target):
            return math.dist(target, (self.x, self.y))
```

```
    def set_sensor_color(self):
        if self.data == -1 or self.data > 300:
            self.color = (0, 255, 0)
        elif self.data < 100:
            self.color = (255, 0, 0)
        elif self.data < 300:
            self.color = (255, 128, 0)
```

14. Контролер даних:

```
import csv
import torch
import pygame
from kan.utils import create_dataset_from_data

class DataManager:
    def __init__(self, screen, files=30, is_recording=False):
        self.trajectory = []
        self.sensors_data = []
        self.motors_speed = []
        self.is_recording = is_recording
        self.files = files
        self.screen = screen

    def record_data(self, point, sensors, motors, filename):
        if self.is_recording:
            self.trajectory.append(point)
            self.sensors_data.append(sensors)
            self.motors_speed.append(motors)
        else:
            if self.trajectory:
                self.save_trajectory(filename)
                self.save_sensor_data(filename)
                self.save_motor_speed(filename)

            self.trajectory.clear()
            self.sensors_data.clear()
            self.motors_speed.clear()

    def save_trajectory(self, filename):
        filename = f"Trajectory/{filename}_tra.csv"
        self.save_file(filename, self.trajectory)

    def save_sensor_data(self, filename):
        filename = f"Data/{filename}_X.csv"
        self.save_file(filename, self.sensors_data)

    def save_motor_speed(self, filename):
        filename = f"Data/{filename}_y.csv"
```

```
        self.save_file(filename, self.motors_speed)

    def save_file(self, filename, data):
        with open(f"Dataset/{filename}", 'w', newline="") as file:
            csv_writer = csv.writer(file)
            csv_writer.writerows(data)

    def load_trajectory(self, filename):
        trajectory = []
        with open(f"Dataset/Trajectory/{filename}_tra.csv", 'r') as file:
            csv_reader = csv.reader(file)
            trajectory = [(float(row[0]), float(row[1])) for row in csv_reader]

        return trajectory

    def load_sensor_data(self, filename):
        with open(f"Dataset/Data/{filename}_X.csv", 'r') as file:
            csv_reader = csv.reader(file)
            sensors = [[float(value) for value in row] for row in csv_reader]
            sensors = torch.tensor(sensors).float()

        return sensors

    def load_motor_speed(self, filename):
        with open(f"Dataset/Data/{filename}_y.csv", 'r') as file:
            csv_reader = csv.reader(file)
            motors = [[float(row[0]), float(row[1])] for row in csv_reader]
            motors = torch.tensor(motors).float()

        return motors

    def aggregate_sensors_data(self):
        data = []

        for i in range(1, self.files + 1):
            data.append(self.load_sensor_data(str(i)))

        return torch.cat(data, dim=0)

    def aggregate_motors_data(self):
        data = []

        for i in range(1, self.files + 1):
            data.append(self.load_motor_speed(str(i)))

        return torch.cat(data, dim=0)

    def get_dataset(self):
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        sensors = self.aggregate_sensors_data()
```

```
        motors = self.aggregate_motors_data()

        return create_dataset_from_data(sensors, motors, device=device)

    def show_trajectory(self, filename, color):
        trajectory = self.load_trajectory(filename)
        pygame.draw.polygon(self.screen, color, trajectory, 2)
```

15. Клас мотору платформи:

```
class Motor:
    """Robot motor. Composes the Robot class."""

    def __init__(self, max_motor_speed, wheel_radius):
        """Motor class constructor. Determines the maximum speed and wheel radius.

        Args:
            max_motor_speed (float): maximum motor speed, in rpm.
            wheel_radius (float): wheel radius, in meters.
        """

        self.max_motor_speed = max_motor_speed
        self.wheel_radius = wheel_radius
        self.power = 0.7

    def set_speed(self, speed):
        """Sets the motor speed.

        Args:
            speed (float): motor speed, in rpm.
        """

        self.speed = speed

    def set_power(self, power):
        self.power = power
        self.set_speed(self.max_motor_speed * self.power)
```

16. Клас платформи:

```
from Classes.Motor import *
from Classes.Sensors.LineDetectorSensor import *
from Classes.Sensors.ColorSensor import *
from Classes.Sensors.SonicSensor import *
import numpy as np
import pygame
```

```
class Robot:
    """Differential robot model. Composed by the Motor class and the Sensor class."""

    def __init__(self, initial_position, width, map, map_image, robot_image, controller,
                 initial_motor_speed=500, max_motor_speed=50000, wheel_radius=0.04):
        """Robot class constructor. Initializes the robot's position and speed.

        Args:
            initial_position (tuple): robot initial position (x, y, heading), where "x" and "y" \
                are the robot's coordinates in meters and "heading" is the angle in radians.
            width (float): robot width, in meters.
            initial_motor_speed (float, optional): initial motor speed, in rpm. \
                Defaults to 500.
            max_motor_speed (float, optional): maximum motor speed, in rpm. \
                Defaults to 1000.
            wheel_radius (float, optional): wheel radius, in meters. Defaults to 0.04.
        """

        # List of the robot sensors
        self.sensors = []
        self.sensors_data = []

        self.sensors.append(LineDetectorSensor((87,0), 10, initial_position, map, map_image))
        self.sensors.append(ColorSensor((103, 0), initial_position, 12, 12, map, map_image))
        self.sensors.append(SonicSensor((65, 0), initial_position, 10, 10, map, map_image))

        # Scale factor from meters to pixels
        self.meters_to_pixels = 3779.52

        # Robot dimensions
        self.width = width

        # Robot initial position
        self.x = initial_position[0]
        self.y = initial_position[1]
        self.heading = initial_position[2]

        # Motor construction
        self.left_motor = Motor(max_motor_speed, wheel_radius)
        self.right_motor = Motor(max_motor_speed, wheel_radius)

        # Initial motor speed
        self.left_motor.set_speed(initial_motor_speed)
        self.right_motor.set_speed(initial_motor_speed)

        self.robot_image = pygame.image.load(f"{robot_image}")
        self.map = map

        self.controller = controller
```

```
    def update_position(self):
        """Updates the robot's position according to the wheel speeds.

        Args:
            dt (float): time elapsed since the last iteration, in seconds."""

        dt = 32/1000
        # Linear wheel speeds
        left_wheel_linear_speed = 2*np.pi*self.left_motor.wheel_radius*self.left_motor.speed/60
        right_wheel_linear_speed =
2*np.pi*self.right_motor.wheel_radius*self.right_motor.speed/60

        # Differential movement
        #
        # Horizontal, vertical and angular movement speeds
        x_speed = (left_wheel_linear_speed + right_wheel_linear_speed)*np.cos(self.heading)/2
        y_speed = (left_wheel_linear_speed + right_wheel_linear_speed)*np.sin(self.heading)/2
        heading_speed = (right_wheel_linear_speed - left_wheel_linear_speed)/self.width
        #
        # Update position and angle according to the elapsed time
        self.x += x_speed*dt
        self.y += y_speed*dt # y-axis is inverted
        self.heading -= heading_speed*dt

        # Adjust the angle to the interval [-2pi, 2pi]
        if (self.heading > 2*np.pi) or (self.heading < -2*np.pi):
            self.heading = 0

        for i in self.sensors:
            i.update_position((self.x, self.y, self.heading))

    def move(self, map_image):
        self.set_map_image(map_image)
        self.draw_robot(self.x, self.y, self.heading)

        controls = self.controller.get_controlls(self.get_sensors_data())
        self.left_motor.set_power(controls[0])
        self.right_motor.set_power(controls[1])

        self.update_position()

    def draw_robot(self, x, y, heading):
        """Draws the robot on the screen.

        Args:
            x (float): robot horizontal position, in meters.
            y (float): robot vertical position, in meters.
            heading (float): robot angle, in radians.
        """

        # Applies the rotation to the robot image according to the "heading" angle
```

```
        rotated_robot = pygame.transform.rotozoom(self.robot_image, -np.degrees(heading), 0.5)

        # Creates a rectangle with the robot image size and positions it at the center of the
robot
        rect = rotated_robot.get_rect(center=(x, y))

        # Draws the robot on the screen at the rectangle position
        self.map.blit(rotated_robot, rect)

        for i in self.sensors:
            i.draw_sensor()

        self.read_sensors()

    def read_sensors(self):
        self.sensors_data.clear()

        for i in self.sensors:
            self.sensors_data.append(i.read_data())

    def get_sensors_data(self):
        flattened = [item for sublist in self.sensors_data for item in (sublist if
isinstance(sublist, (list, tuple)) else [sublist])]
        return flattened

    def set_map_image(self, map_image):
        for i in self.sensors:
            i.set_map_image(map_image)
```

17. Підключення залежностей:

```
import pygame
from pygame.locals import *
from Classes.Sensors.InfraredSensor import *
from Classes.Sensors.LineDetectorSensor import *
from Classes.Sensors.ColorSensor import *
from Classes.Robot import *
from Classes.Sensors.SonicSensor import *
from Classes.Controllers.PIDController import *
from Classes.Controllers.KANController import *
from Classes.Controllers.MLPController import *
from Classes.GUI.Button import *
from Classes.DataManager import *
from Classes.GUI.GUI import *
from Classes.Obstacles.Obstacle import *
from Classes.Obstacles.ObstacleWrapper import *
```

18. Основний потік виконання:

```
from dependencies import *
import csv

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 1500, 720
pygame.display.set_caption("Line Following Robot Simulation")
screen = pygame.display.set_mode((WIDTH, HEIGHT))
map_image = pygame.image.load("Dataset/Images/1.png")

dm = DataManager(screen)
gui = GUI(screen, dm, map_image)

kan = KANController([12,2,2], grid=3, k=3, seed=1, lr=1.2, steps=200, dataset=dm.get_dataset())
#kan.train()
kan.KAN = kan.KAN.loadckpt('./model/0.1')

mlp = MLPController([12,10,7,2], seed=1, lr=1.2, steps=200, dataset=dm.get_dataset())
mlp.train()

pos = 1000

Kp = 0.025
Ki = 0.000625
Kd = 0.000025
pid = PIDController(Kp, Ki, Kd)

r1 = Robot((250, 350, 0), 0.5, screen, gui.map_image, "Assets/RobotPID.png", pid)
r2 = Robot((250, 350, 0), 0.5, screen, gui.map_image, "Assets/RobotKAN.png", kan)
r3 = Robot((250, 350, 0), 0.5, screen, gui.map_image, "Assets/RobotMLP.png", mlp)

gui.r = r1

obw = ObstacleWrapper(screen, [r1.sensors[2], r2.sensors[2], r3.sensors[2]], (1280, 720))

ob1 = obw.register_obstacle((300, 300), (30, 100))
ob1.set_speed(0, 10)
ob2 = obw.register_obstacle((800, 300), (30, 100))
ob2.set_speed(0, 15)
ob3 = obw.register_obstacle((300, 300), (100, 30))
ob3.set_speed(10, 0)

# Main loop
running = True
draw = False

def save_file(filename, data):
```

```
    with open(f"Dataset/{filename}", 'w', newline="") as file:
        csv_writer = csv.writer(file)
        csv_writer.writerows(data)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            cursor_position = event.pos

            if cursor_position[0] < 1280 and cursor_position[1] < 720:
                r1.x = r2.x = r3.x = cursor_position[0]
                r1.y = r2.y = r3.y = cursor_position[1]
                r1.heading = r2.heading = r3.heading = 0

            gui.on_click_event(event)


    # Draw buttons
    gui.draw_GUI()
    obw.draw()
    obw.move()

    r1.move(gui.map_image)
    r2.move(gui.map_image)
    r3.move(gui.map_image)

    # Update display
    pygame.display.flip()
    pygame.time.delay(16)  # Roughly 60 FPS

# Quit Pygame
pygame.quit()
```

19. Генератор треків:

```
import pygame
import math
import random

# Initialize Pygame
pygame.init()
# Screen setup
WIDTH, HEIGHT = 1280, 720
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Random Loop Track Generator")
clock = pygame.time.Clock()
```

```
# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)

width = 10
def random_track(center, radius, points_count):
    """Generates a looped track with random straight and curved segments."""
    angles = sorted(random.uniform(0, 2 * math.pi) for _ in range(points_count))
    track = []
    for angle in angles:
        # Randomize straight or curved segment
        distance = radius + random.uniform(-20, 100)
        x = center[0] + distance * math.cos(angle)
        y = center[1] + distance * math.sin(angle)
        track.append((x, y))
    return track

def draw_track(surface, track, color, width=5):
    """Draws the generated track."""
    for i in range(len(track) - 1):
        pygame.draw.line(surface, color, track[i], track[i + 1], width)

def main():
    running = True
    center = (WIDTH // 2, HEIGHT // 2)
    radius = 200
    points_count = 15
    track = random_track(center, radius, points_count)
    i = 7
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            '''elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_r:  # Press 'R' to regenerate the track'''

        # Drawing
        screen.fill(WHITE)
        #draw_track(screen, track, BLACK, 10)
        pygame.draw.polygon(screen, BLACK, track, width)
        pygame.display.flip()
        pygame.image.save(screen, f"Dataset/{i}.png")
        i += 1
        track = random_track(center, radius, points_count)
        if i >= 30:
            running = False
        clock.tick(30)
    pygame.quit()
if __name__ == "__main__":
    main()
```

ДОДАТОК Б
3Д МОДЕЛІ КРІПЛЕНЬ ДАТЧИКІВ

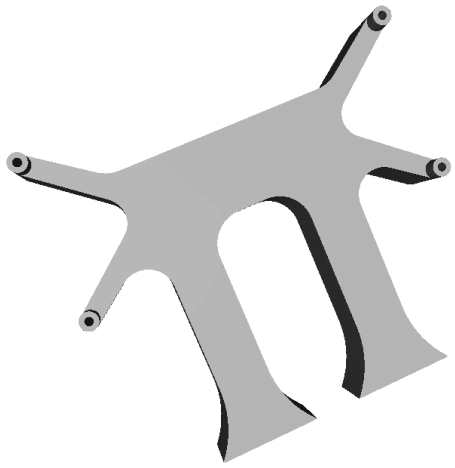

Рисунок Б.1 – Кріплення ультразвукового датчику

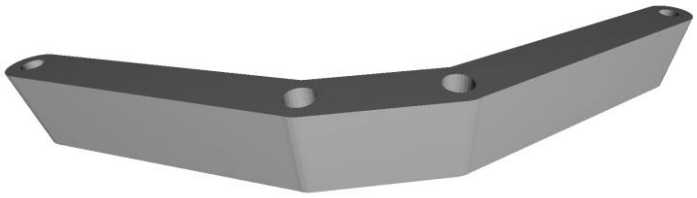

Рисунок Б.2 – Кріплення датчику лінії

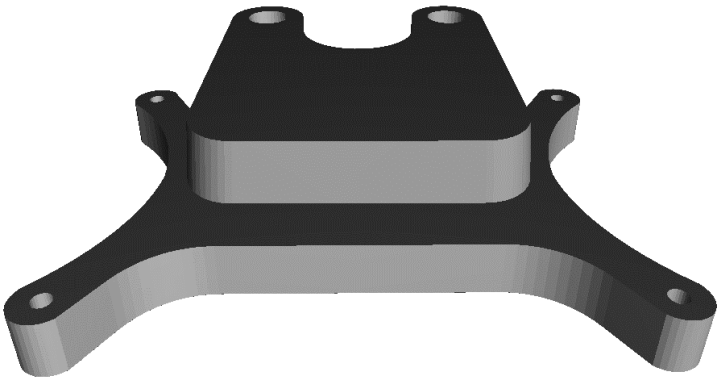

Рисунок Б.3 – Кріплення датчику кольору

ДОДАТОК В
ДРУКОВАНІ ПЛАТИ МОДУЛІВ

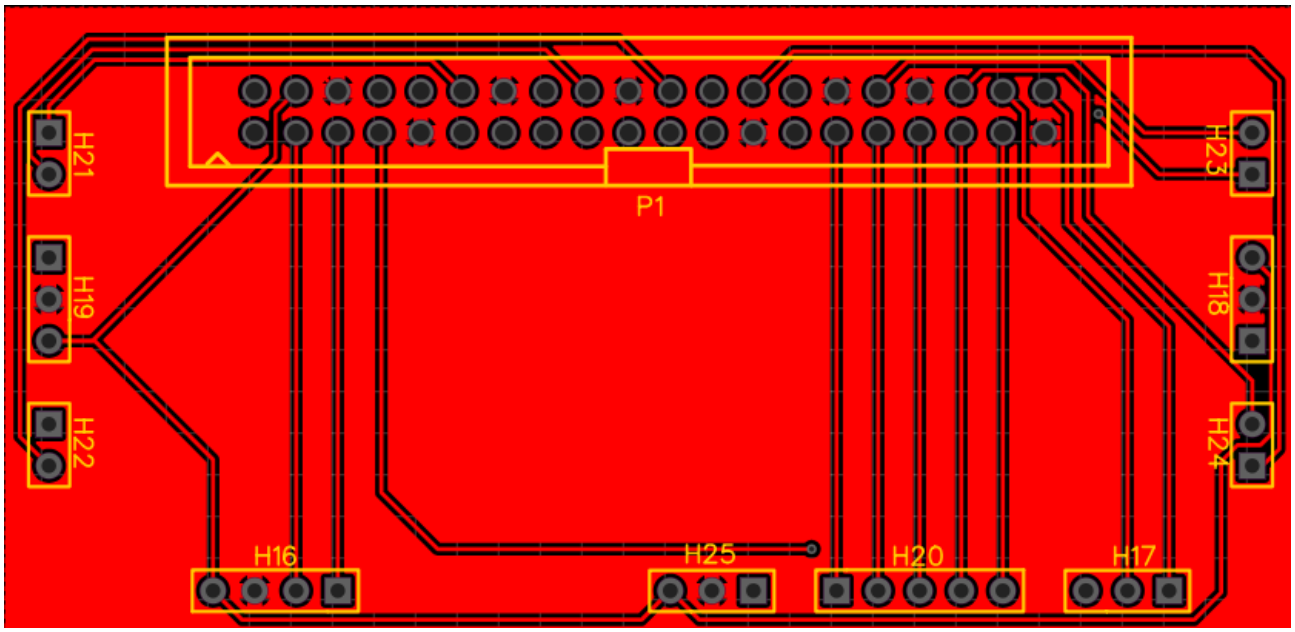

Рисунок В.1 – Модуль керування, вид зверху

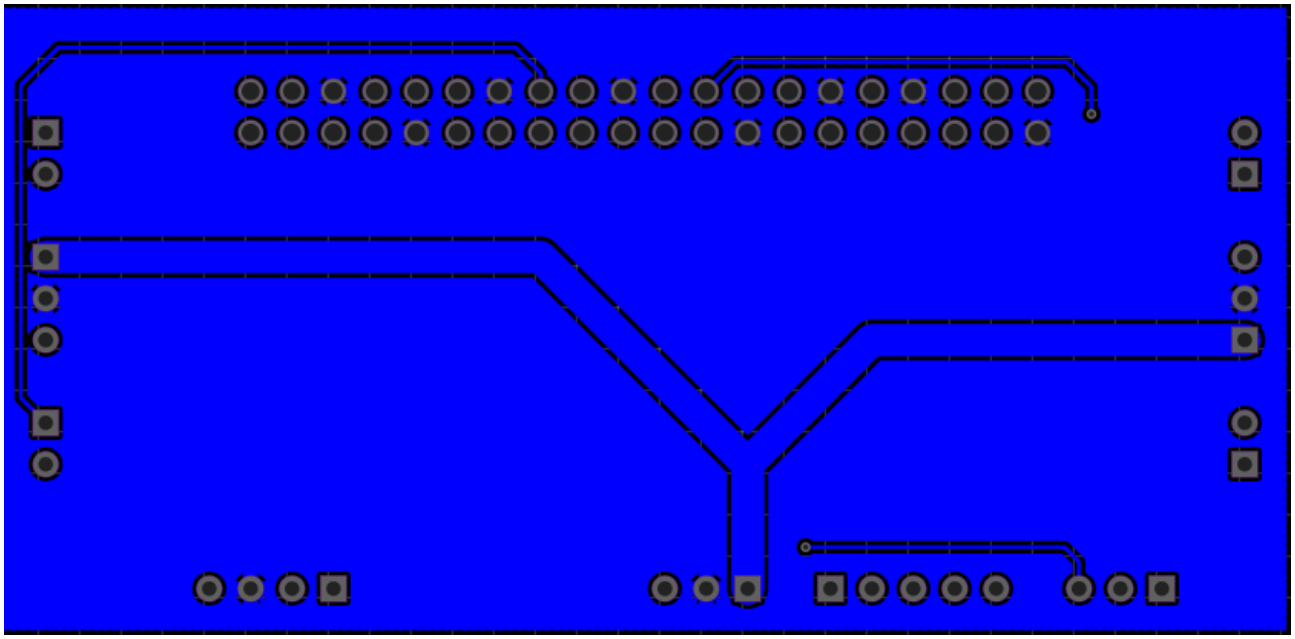

Рисунок В.2 – Модуль керування, вид знизу

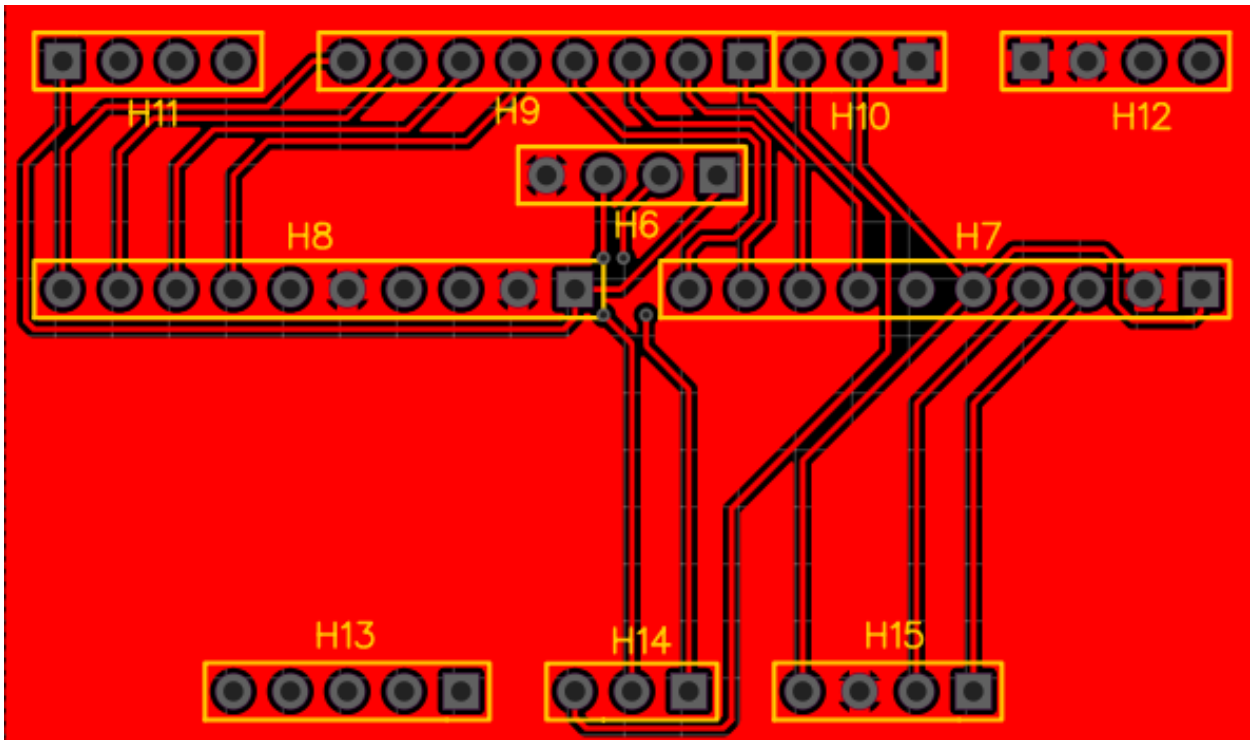

Рисунок В.3 – Модуль сенсорів, вид зверху

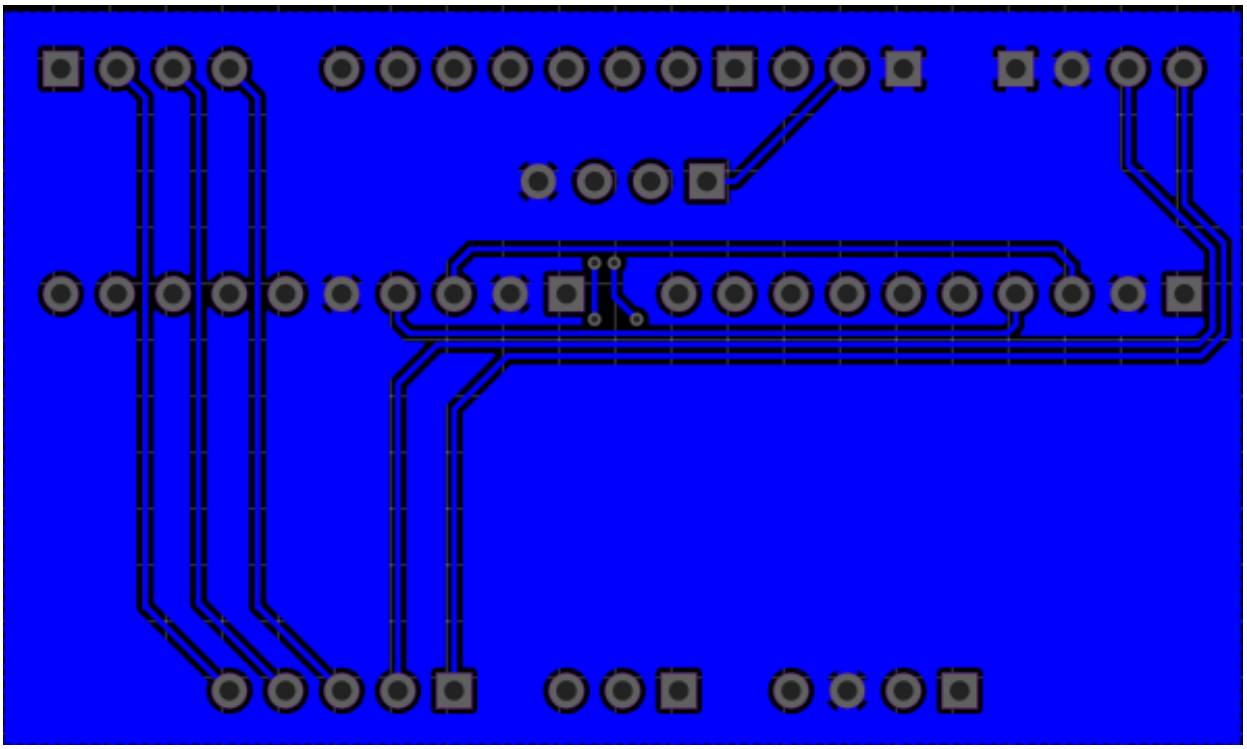

Рисунок В.4 – Модуль сенсорів, вид знизу

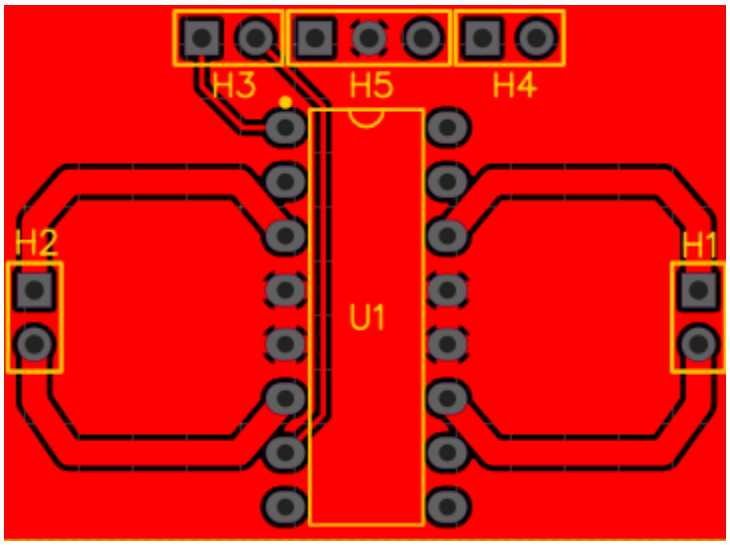

Рисунок В.5 – Модуль драйверу двигунів, вид зверху

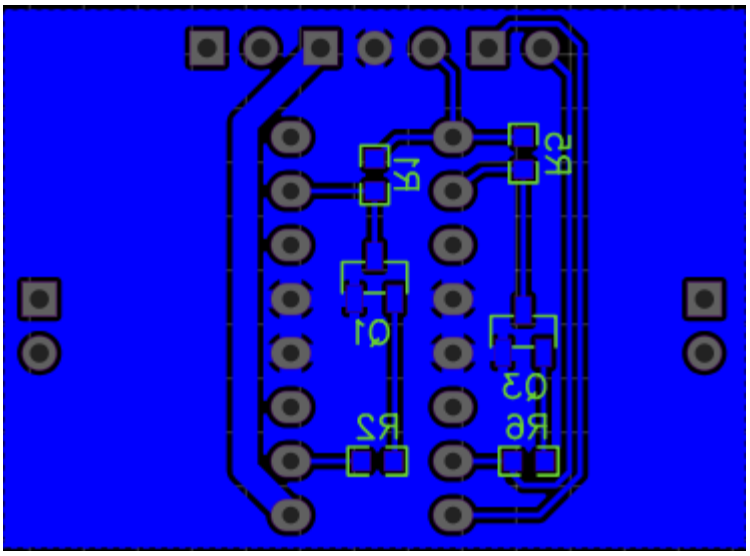

Рисунок В.6 – Модуль драйверу двигунів, вид знизу