КВАЛІФІКАЦІЙНА РОБОТА

МАГІСТРА

КРМ.АКСм-06.00.00.000 ПЗ

Група АКСм-24-1

Олег ГУЛИЧ

Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інформаційно-телекомунікаційних технологій та систем

Гулич Олег Володимирович

(прізвище, ім'я, по батькові)

УДК 681.5.015.23:004.032.26

МАГІСТЕРСЬКА РОБОТА

Розроблення системи керування положенням поворотної платформи на основі нейромережевих алгоритмів

(назвароботи)

Комп'ютеризовані системи управління та автоматика
(назваосвітньої програми)

174-Автоматизація, комп'ютерно-інтегровані технології та робототехніка

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____ Г.О. Гулич
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ Стрілецький Юрій Йосипович, д.т.н., професор
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувачкафедри

_____ ***Заміховський Л.М.***
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*
Кафедра *Інформаційно- телекомунікаційних технологій і систем*
Освітній рівень *Автоматизація та комп'ютерно-інтегровані технології*
Спеціальність _ *174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка*

ЗАТВЕРДЖУЮ
Зав кафедри Зав. кафедрою ІТТС д.т.н., проф.
_____Л.М. Заміховський

«____» листопад _____2025 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТОВІ

_____ Гулич Олег Володимирович_____
(прізвище, ім'я, по батькові)

1. Тема роботи _ Розроблення системи керування положенням поворотної платформи на основі нейромережевих алгоритмів _____
_____
_____

керівник роботи, *Стрілецький Юрій Йосипович*.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від "30" *жовтня* 2025 року № 690/7

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи роботи Колекторний двигун. платформа вагою 50кг, час дискретизації 0.1с

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз існуючих систем керування поворотними платформами. Розробка моделі фізичного обєкту для імітаційного моделювання. Розробка моделі керування платформою із використанням нейромережі . Результати експериментальних досліджень нейромережевого контролера

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Об'єкт керування і його математична модель. Архітектура розробленої нейромережі для керування поворотною платформою. Реалізація процесу навчання розробленої нейромережі. Аналіз ефективності агентів навчання. Порівняльний аналіз ефективнсоті застосування нейромережевого контролера

6. Дата видачі завдання _____листопад 2025_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Термін виконання етапів роботи	Примітка
1	Аналіз існуючих систем керування поворотними платформами.		*Виконано*
2	Розробка моделі фізичного обєкту для імітаційного моделювання.		*Виконано*
3	Розробка моделі керування платформою із використанням нейромережі .		*Виконано*
4	Результати експериментальних досліджень нейромережевого контролера		*Виконано*
5	*Оформлення роботи*	*01.12.2025-25.01.2026*	*Виконано*

Студент _____ Гулич О._

(підпис) (прізвище та ініціали)

Керівник роботи _____ Стрілецький Ю. Й.__

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У роботі розглянуто розроблення системи керування положенням поворотної платформи на основі нейромережевих алгоритмів для підвищення точності позиціонування та робастності до зовнішніх збурень. Виконано аналіз існуючих систем керування поворотними платформами, зокрема конструктивних схем, датчиків положення та орієнтації, класичних методів керування (PID, LQR) та інтелектуальних підходів (нечітка логіка, нейронні мережі). Обґрунтовано вибір рекурентної нейромережевої архітектури на основі GRU та методу навчання з підкріпленням в умовах відсутності апріорних даних про об'єкт керування. Значну увагу приділено розробці цифрового двійника системи, зокрема математичній моделі колекторного двигуна, моделі ударних і зовнішніх збурень, а також технології векторизації обчислень для прискорення процесу навчання. У роботі розроблено мультиагентну систему формування винагороди, що включає агенти цілевказівки, часу, стабілізації, плавності керування та захисту від насичення. Запропоновано метод обробки циклічності кутових координат та алгоритмічне забезпечення на основі еволюційних стратегій з дзеркальним семплюванням. Здійснено програмне моделювання роботи системи, проведено експериментальну перевірку ефективності нейромережевого контролера шляхом порівняльного аналізу з класичними методами керування.

Отримані результати підтверджують доцільність використання нейромережевих алгоритмів для створення систем керування поворотними платформами, що не потребують точного математичного моделювання об'єкта та демонструють високу адаптивність до змінних умов експлуатації. Розроблені рішення можуть бути використані в практиці побудови інтелектуальних систем орієнтації в робототехніці, системах стабілізації камер, антенних системах та управління промисловими маніпуляторами.

ANNOTATION

The paper considers the development of a control system for rotary platform positioning based on neural network algorithms to improve positioning accuracy and robustness to external disturbances. An analysis of existing rotary platform control systems is performed, including structural designs, position and orientation sensors, classical control methods (PID, LQR), and intelligent approaches (fuzzy logic, neural networks). The choice of a recurrent neural network architecture based on GRU and reinforcement learning method under conditions of absent a priori data about the control object is justified. Considerable attention is paid to the development of a digital twin of the system, in particular, the mathematical model of the DC motor, the model of shock and external disturbances, and computation vectorization technology to accelerate the training process. The paper develops a multi-agent reward shaping system that includes targeting, time, stabilization, control smoothness, and saturation protection agents. A method for processing angular coordinate cyclicity and algorithmic support based on evolutionary strategies with mirror sampling are proposed. Software modeling of the system was carried out, and an experimental verification of the neural network controller effectiveness was conducted through comparative analysis with classical control methods.

The results obtained confirm the feasibility of using neural network algorithms to create rotary platform control systems that do not require precise mathematical modeling of the object and demonstrate high adaptability to changing operating conditions. The developed solutions can be used in the practice of building intelligent orientation systems in robotics, camera stabilization systems, antenna systems, and industrial manipulator control.

РЕФЕРАТ

Тема роботи: Розроблення системи керування положенням поворотної платформи на основі нейромережевих алгоритмів

Пояснювальна записка: 86 стор., 11 рисунків, 25 посилань на літературні джерела.

Об'єктом дослідження є процеси керування положенням та орієнтацією поворотних платформ у системах автоматизації, що функціонують в умовах невизначеності та дії зовнішніх збурень.

Предметом дослідження є методи синтезу нейромережевих контролерів на основі рекурентних архітектур, алгоритми навчання з підкріпленням в умовах відсутності апріорних даних, а також технології цифрового моделювання мехатронних систем.

Метою дослідження даної роботи є розроблення та дослідження системи керування поворотною платформою на основі нейромережевих алгоритмів, що забезпечує високу точність позиціонування та робастність до зовнішніх збурень шляхом використання методів еволюційних стратегій та рекурентних нейронних мереж.

Практичне значення одержаних результатів полягає у створенні системи керування, яка дозволяє реалізувати автоматичне наведення та стабілізацію поворотних платформ без необхідності точного математичного моделювання об'єкта керування. Запропоновані рішення можуть бути використані для побудови інтелектуальних систем орієнтації в робототехніці, системах стабілізації камер, антенних системах та промислових маніпуляторах.

В першому розділі проведено огляд та аналіз існуючих систем керування поворотними платформами, включаючи конструктивні схеми платформ, датчики положення та орієнтації, класичні методи керування (PID, LQR) та інтелектуальні методи (нечітка логіка, нейронні мережі). Визначено їх обмеження та обґрунтовано доцільність застосування нейромережевих алгоритмів для підвищення якості керування.

В другому розділі розроблено модель фізичного об'єкту для імітаційного моделювання. Створено узагальнену фізичну модель об'єкта, розроблено передаточну функцію колекторного двигуна, запропоновано модель ударних і зовнішніх збурень та реалізовано цифровий двійник системи керування платформою для забезпечення процесу навчання нейромережевого контролера.

В третьому розділі розроблено модель керування платформою із використанням нейромережі. Обґрунтовано вибір методу навчання в умовах відсутності апріорних даних, розроблено систему формування вхідного вектора станів з обробкою циклічності кутових координат, запропоновано рекурентну архітектуру на основі GRU, створено алгоритмічне забезпечення процесу навчання з використанням методу еволюційних стратегій та дзеркального семплювання. Реалізовано програмне середовище "Цифрового двійника" з векторизацією обчислень та мультиагентною системою формування винагороди, що включає агенти цілевказівки, часу, стабілізації, плавності керування та захисту від насичення.

В четвертому розділі представлено результати експериментальних досліджень нейромережевого контролера, включаючи аналіз динаміки навчання та формування стратегії керування, а також порівняльний аналіз ефективності та робастності системи відносно класичних методів керування.

Ключові слова: КЕРУВАННЯ ПОЛОЖЕННЯМ, ПОВОРОТНА ПЛАТФОРМА, НЕЙРОННА МЕРЕЖА, РЕКУРЕНТНА АРХІТЕКТУРА, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, ЕВОЛЮЦІЙНІ СТРАТЕГІЇ, ЦИФРОВИЙ ДВІЙНИК, РОБАСТНІСТЬ.

ABSTRACT

Topic: Development of a control system for rotary platform positioning based on neural network algorithms

Explanatory note: 86 pages, 11 figures, 25 references.

The object of study is the processes of controlling the position and orientation of rotary platforms in automation systems functioning under conditions of uncertainty and external disturbances.

The subject of study is the methods of synthesizing neural network controllers based on recurrent architectures, reinforcement learning algorithms under conditions of absent a priori data, and digital modeling technologies for mechatronic systems.

The goal of the research is to develop and investigate a rotary platform control system based on neural network algorithms that ensures high positioning accuracy and robustness to external disturbances through the use of evolutionary strategy methods and recurrent neural networks.

The practical value of the obtained results lies in the creation of a control system that enables automatic guidance and stabilization of rotary platforms without the need for precise mathematical modeling of the control object. The proposed solutions can be used to build intelligent orientation systems in robotics, camera stabilization systems, antenna systems, and industrial manipulators.

The first chapter provides a review and analysis of existing rotary platform control systems, including platform structural designs, position and orientation sensors, classical control methods (PID, LQR), and intelligent methods (fuzzy logic, neural networks). Their limitations are identified, and the feasibility of using neural network algorithms to improve control quality is substantiated.

The second chapter develops a physical object model for simulation modeling. A generalized physical model of the object is created, a transfer function for the DC motor is developed, a model of shock and external disturbances is proposed, and a digital twin of the platform control system is implemented to support the neural network controller training process.

The third chapter develops a platform control model using neural networks. The choice of training method under conditions of absent a priori data is justified, a system for forming the input state vector with processing of angular coordinate cyclicity is developed, a recurrent architecture based on GRU is proposed, and algorithmic support for the training process using evolutionary strategies and mirror sampling methods is created. A "Digital Twin" software environment is implemented with computation vectorization and a multi-agent reward shaping system, including targeting, time, stabilization, control smoothness, and saturation protection agents.

The fourth chapter presents the results of experimental studies of the neural network controller, including analysis of training dynamics and control strategy formation, as well as comparative analysis of system efficiency and robustness relative to classical control methods.

Keywords: POSITION CONTROL, ROTARY PLATFORM, NEURAL NETWORK, RECURRENT ARCHITECTURE, REINFORCEMENT LEARNING, EVOLUTIONARY STRATEGIES, DIGITAL TWIN, ROBUSTNESS.

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

ПІД – пропорційно-інтегрально-диференціальний регулятор, основний класичний метод керування, що аналізується у роботі.
ШНМ – штучна нейронна мережа, основа розробленої системи керування.
ДПС – двигун постійного струму з колекторним вузлом, що використовується як виконавчий механізм.
ШІМ – широтно-імпульсна модуляція, метод керування потужністю двигуна.
ФАПЧ – фазове автоналаштування частоти, система синхронізації у контурі керування.
МНК – метод найменших квадратів, що використовується для ідентифікації параметрів.
АЦП – аналого-цифровий перетворювач, елемент системи вимірювання.
ЦАП – цифро-аналоговий перетворювач, що формує керуючий сигнал.
BER – Bit Error Rate, коефіцієнт бітових помилок.
DDS – Direct Digital Synthesis, прямий цифровий синтез сигналів.
SNR – Signal-to-Noise Ratio, відношення сигнал/шум.
BLDC – Brushless DC motor, безколекторний двигун постійного струму.
PMSM – Permanent Magnet Synchronous Motor, синхронний двигун з постійними магнітами.
LQR – Linear Quadratic Regulator, лінійно-квадратичний регулятор.
SMC – Sliding Mode Control, керування у ковзному режимі.
FLC – Fuzzy Logic Control, керування на основі нечіткої логіки.
RNN – Recurrent Neural Network, рекурентна нейронна мережа.
GRU – Gated Recurrent Unit, керований рекурентний блок.
LSTM – Long Short-Term Memory, мережа довгої короткочасної пам'яті.
MLP – Multi-Layer Perceptron, багатошаровий персептрон.
ES – Evolution Strategies, еволюційні стратегії оптимізації.
GA – Genetic Algorithm, генетичний алгоритм.
SGD – Stochastic Gradient Descent, стохастичний градієнтний спуск.
IMU – Inertial Measurement Unit, інерціальний вимірювальний модуль.
MEMS – Micro-Electro-Mechanical Systems, мікроелектромеханічні системи.
SIMD – Single Instruction Multiple Data, одна інструкція - багато даних.
CPU – Central Processing Unit, центральний процесор.
GPU – Graphics Processing Unit, графічний процесор.
RL – Reinforcement Learning, навчання з підкріпленням.Claude is AI and can make mistakes. Please double-check responses.

Зміст с.

ВСТУП

Актуальність теми. Сучасний розвиток систем автоматизації та робототехніки супроводжується зростанням вимог до точності та надійності керування рухомими механічними системами, зокрема поворотними платформами. Проблема забезпечення високоякісного керування положенням є особливо вагомою для систем орієнтації та стабілізації, де умови експлуатації характеризуються невизначеністю параметрів об'єкта та дією зовнішніх збурень. Традиційний підхід, що базується на класичних методах керування (PID, LQR), має суттєві обмеження, оскільки вимагає точного математичного моделювання об'єкта та є чутливим до параметричних невизначеностей. Існуючі інтелектуальні методи, хоч і демонструють кращу адаптивність, часто вимагають значного обсягу апріорних даних або складного налаштування. Тому розробка нових методів керування на основі нейромережевих алгоритмів, які дозволяють реалізувати автоматичне навчання системи без точної моделі об'єкта та забезпечують високу робастність до збурень, є особливо актуальною.

Метою дослідження є розроблення та дослідження системи керування положенням поворотної платформи на основі нейромережевих алгоритмів, що забезпечує високу точність позиціонування та робастність до зовнішніх збурень шляхом використання рекурентних архітектур та методів навчання з підкріпленням.

Завданням дослідження є:

- провести аналіз існуючих систем керування поворотними платформами, включаючи конструктивні схеми, датчики положення та орієнтації, класичні та інтелектуальні методи керування для обґрунтування вибору нейромережевого підходу;
- розробити математичну модель фізичного об'єкта, що включає передаточну функцію колекторного двигуна та модель ударних і зовнішніх збурень для реалістичного відтворення умов експлуатації;

– створити цифровий двійник системи керування платформою з використанням технології векторизації обчислень для ефективного навчання нейромережевого контролера;
– розробити архітектуру нейромережевого контролера на основі рекурентних GRU-комірок з методом обробки циклічності кутових координат;
– розробити мультиагентну систему формування винагороди, що включає агенти цілевказівки, часу, стабілізації, плавності керування та захисту від насичення;
– реалізувати алгоритмічне забезпечення процесу навчання на основі методу еволюційних стратегій з дзеркальним семплюванням;
– провести програмне моделювання роботи системи, дослідити динаміку навчання нейромережевого контролера та оцінити ефективність методу шляхом порівняльного аналізу з класичними методами керування.

Об'єктом дослідження є процеси керування положенням та орієнтацією поворотних платформ у системах автоматизації, що функціонують в умовах невизначеності та дії зовнішніх збурень.

Предметом дослідження є методи синтезу нейромережевих контролерів на основі рекурентних архітектур, алгоритми навчання з підкріпленням в умовах відсутності апріорних даних, а також технології цифрового моделювання мехатронних систем.

Практичне значення одержаних результатів полягає у створенні системи керування поворотною платформою, яка дозволяє реалізувати високоточне позиціонування та стабілізацію без необхідності точного математичного моделювання об'єкта з можливістю адаптації до змінних умов експлуатації. Запропоновані рішення можуть бути використані для побудови інтелектуальних систем орієнтації в робототехніці, системах стабілізації камер, антенних системах та управління промисловими маніпуляторами в умовах дії зовнішніх збурень та параметричних невизначеностей.

1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ КЕРУВАННЯ ПОВОРОТНИМИ ПЛАТФОРМАМИ

1.1 Конструктивні схеми поворотних платформ

Проектування та розробка систем керування поворотними платформами вимагає глибокого аналізу їхньої електромеханічної структури, оскільки саме конструктивні особливості визначають граничні динамічні характеристики, точність позиціонування та енергоефективність системи в цілому. Поворотна платформа як об’єкт керування являє собою складну електромеханічну систему, що складається з виконавчого двигуна, передавального механізму та несучої конструкції. Вибір кожного з цих елементів безпосередньо впливає на математичну модель об’єкта, наявність нелінійностей та вибір стратегії керування.

У сучасних системах автоматизації для приведення в рух поворотних платформ діаметром до 0,5 м використовують широкий спектр електричних машин, кожна з яких має свої переваги та недоліки в контексті керованості. Історично першими для подібних задач застосовувалися двигуни постійного струму з колекторним вузлом (ДПС). Їхня перевага полягає у лінійності механічних характеристик та простоті реалізації законів керування швидкістю та моментом. Однак наявність щітково-колекторного вузла суттєво обмежує надійність системи, створює електромагнітні завади, що є критичним для чутливих сенсорів, розташованих на платформі, та вносить додаткове тертя [1].

Асинхронні двигуни з короткозамкненим ротором, хоча і є найбільш поширеними в промисловості завдяки своїй надійності та низькій вартості, рідко використовуються для прецизійних поворотних платформ малого та середнього діаметра. Це зумовлено складністю керування моментом на низьких швидкостях та необхідністю використання частотних перетворювачів з векторним керуванням для досягнення прийнятної

динаміки. Крім того, асинхронні двигуни мають гірші масогабаритні показники порівняно з синхронними машинами аналогічної потужності, що ускладнює їх інтеграцію в компактні конструкції [2].

Значного поширення в системах відкритого контуру набули крокові двигуни. Вони дозволяють реалізувати позиціонування без використання датчиків зворотного зв'язку, що спрощує та здешевлює конструкцію. Проте дискретний характер руху ротора призводить до виникнення резонансних явищ на певних частотах, що може спричинити вібрації платформи та втрату кроків при високих навантаженнях або різких прискореннях. Для платформ, що несуть оптичне або вимірювальне обладнання, вібрації, викликані кроковим характером руху, є неприпустимими, оскільки вони знижують якість отримуваних даних. Сучасні драйвери з функцією мікрокроку частково вирішують цю проблему, проте це знижує ефективний момент двигуна [3].

Найбільш ефективним рішенням для високодинамічних та точних поворотних платформ на сьогодні є використання безколекторних синхронних двигунів із постійними магнітами (PMSM або BLDC), які в поєднанні з датчиками положення та драйверами утворюють сервоприводи. Такі системи забезпечують високу питому потужність, широкий діапазон регулювання швидкості та можливість точного керування моментом у всьому діапазоні робочих режимів. Відсутність щіткового контакту підвищує ресурс роботи та знижує рівень завад. Висока динаміка сервоприводів дозволяє компенсувати збурення, викликані зміною навантаження або зовнішніми факторами, що є критичним для стабілізації платформи [4].

Важливим елементом конструктивної схеми, що визначає якість передачі руху від двигуна до платформи, є механічна передача. У класичних схемах використовуються редуктори (циліндричні, черв'ячні, планетарні), які дозволяють узгодити високу швидкість обертання двигуна з низькою швидкістю повороту платформи та суттєво збільшити вихідний момент.

Рисунок 1.1–Редукторна пара повороту башти крана із прямими шестернями

Рисунок 1.2– Вид червячного редуктора поворотної платформи

Однак використання зубчастих передач неминуче вносить у систему нелінійність типу «люфт» (backlash). Люфт у кінематичному ланцюзі призводить до виникнення зони нечутливості, появи граничних циклів коливань та погіршення стійкості замкненої системи керування. Для платформ, де вимагається висока точність наведення, люфт є одним із головних факторів, що обмежують точність [5]. Конструктивні методи

зменшення люфту, такі як використання розрізних шестерень або попереднього натягу, ускладнюють механіку та збільшують тертя.

Рисунок 1.3– Вид планетарного редуктора для повороту платформи

Окремим класом передавальних механізмів, що застосовуються в прецизійних системах, є хвильові редуктори (harmonic drives). Вони забезпечують практично нульовий люфт, високу жорсткість на кручення та велике передавальне число в компактному корпусі. Це робить їх ідеальними для застосування в робототехніці та оптико-механічних системах. Проте, хвильові передачі мають специфічну помилку кінематичної передачі та схильність до виникнення резонансних коливань через гнучкість гнучкого колеса, що вимагає врахування пружних властивостей редуктора при синтезі регулятора [6].

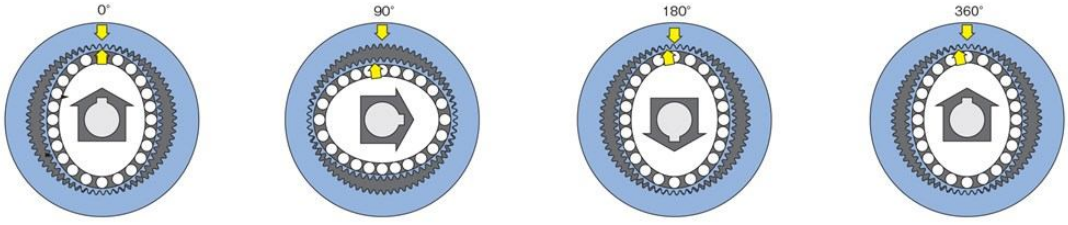

Рисунок 1.4– Етапи роботи хвилевого редуктора

Альтернативою редукторним схемам є використання прямого приводу (Direct Drive), де ротор двигуна безпосередньо з’єднаний з валом навантаження або є частиною поворотної платформи [Рис. 1.1]. Системи прямого приводу виключають люфт, тертя та пружність передавальних

ланок, що дозволяє значно підвищити точність та швидкодію системи. Завдяки відсутності проміжних ланок частота власного механічного резонансу системи значно підвищується, що дозволяє розширити смугу пропускання контуру керування. Однак прямий привід вимагає використання високомоментних двигунів, які, як правило, мають більші габарити та вагу. Крім того, в системах прямого приводу всі збурення з боку навантаження безпосередньо передаються на вал двигуна без демпфування редуктором, що ставить підвищені вимоги до жорсткості механічної характеристики приводу та швидкодії системи керування. Також для прямого приводу характерним є вплив пульсацій моменту (cogging torque), які необхідно компенсувати програмними методами [7].

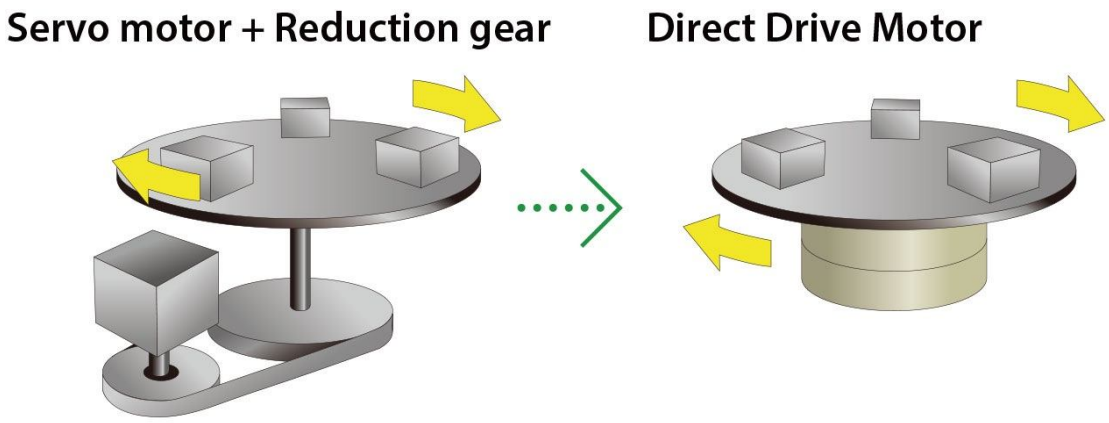

Рисунок 1.5– Порівняння конструкції платформ із приводом через передачу і прямого приводу

При розробці конструктивної схеми поворотної платформи діаметром до 0,5 м необхідно враховувати механічні обмеження та параметри навантаження. Маса рухомої частини платформи та об’єктів, що на ній розташовані, визначає момент інерції системи J. Оскільки момент інерції пропорційний квадрату радіуса розподілу маси ($J \sim mr^2$), навіть незначне збільшення габаритів платформи або зміщення корисного навантаження від центру обертання призводить до суттєвого зростання інерційного навантаження на привід. Це вимагає від системи керування адаптивності або робастності до зміни параметрів об’єкта, оскільки в реальних умовах

експлуатації маса та розташування обладнання на платформі можуть змінюватися [8].

Жорсткість конструкції є ще одним критичним параметром. Недостатня жорсткість елементів платформи або кріплення приводу призводить до виникнення низькочастотних резонансів, які можуть потрапити в смугу пропускання системи керування, спричиняючи нестійкість. Для платформ середнього розміру (до 0,5 м) часто виникає проблема деформації площини платформи під дією ваги ексцентрично розташованого навантаження, що призводить до додаткового тертя в опорних підшипниках та зміни моменту опору обертанню. Використання опорно-поворотних пристроїв на базі перехресних роликових підшипників дозволяє забезпечити високу вантажопідйомність та жорсткість при збереженні компактних розмірів [9].

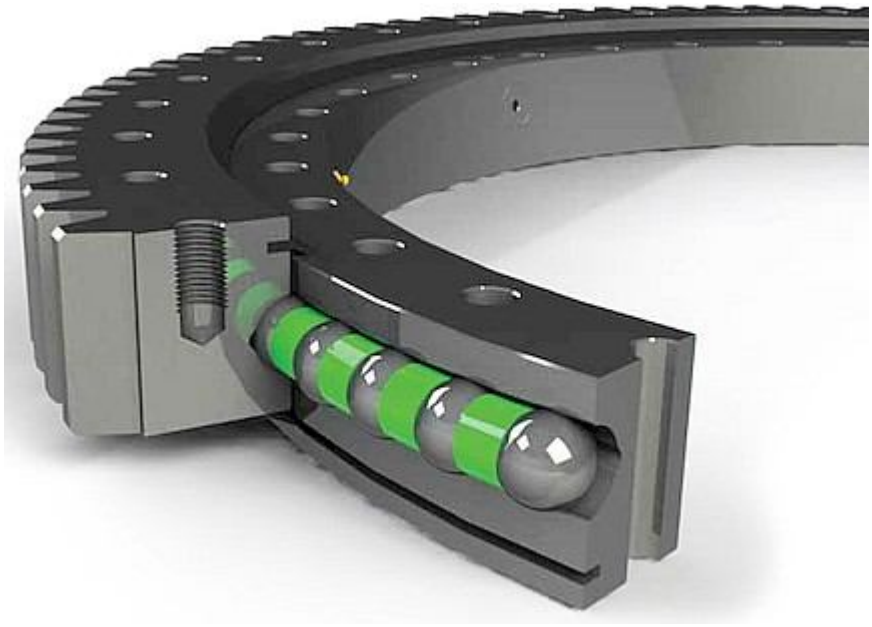

Рисунок 1.6– Платформа на кулькових опорних підшипниках

Ударні навантаження, що виникають при транспортуванні мобільних комплексів або різких зупинках платформи, накладають вимоги до міцності зубчастих зачеплень та валів. У випадку використання редукторів з великим передавальним числом, момент інерції двигуна, приведений до валу навантаження, значно зростає, і при різкій зупинці платформи кінетична енергія ротора може зруйнувати зуби редуктора. Для запобігання цьому в

конструкцію часто вводять запобіжні муфти, проте вони знижують загальну жорсткість системи та вносять додаткові похибки позиціонування.

Рисунок 1.7– Платформа на схрещених роликових підшипниках

Вплив сил тертя в опорах та передачах має яскраво виражений нелінійний характер, що включає в’язке тертя, кулонівське (сухе) тертя та ефект Штрібека при малих швидкостях. Для прецизійних поворотних платформ, що працюють в режимі стеження за повільними об’єктами, нестабільність моменту тертя може призводити до ривків (stick-slip effect), що унеможливлює плавний рух. Мінімізація тертя досягається як конструктивними методами (вибір відповідних пар тертя, мастила, типу підшипників), так і алгоритмічною компенсацією в системі керування, що вимагає точної ідентифікації моделі тертя конкретної механічної конструкції [10].

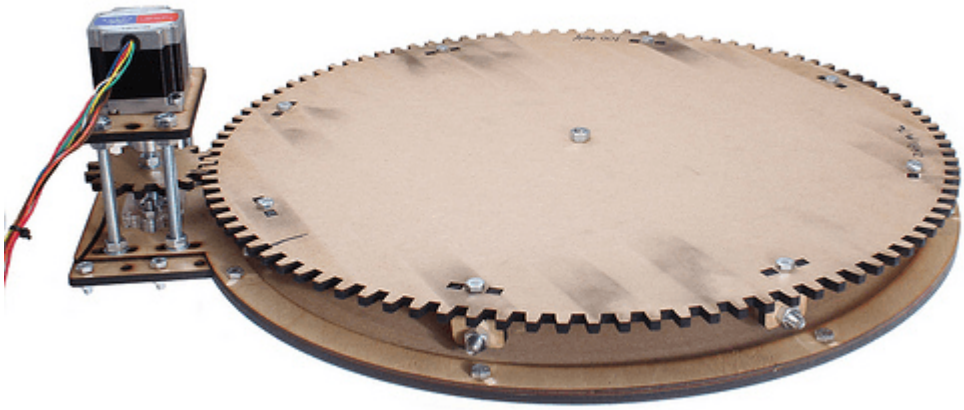

Рисунок 1.8– Підшипники Lazy Susan використовуються в низькошвидкісних системах для обертання платформи

Таким чином, вибір конструктивної схеми поворотної платформи є багатокритеріальною задачею, де необхідно знаходити компроміс між точністю, швидкодією, вартістю та масогабаритними показниками. Аналіз літературних джерел показує, що для сучасних високотехнологічних застосувань найбільш перспективним є використання безколекторних сервоприводів у поєднанні з безлюфтовими хвильовими редукторами або технологією прямого приводу. Такий підхід дозволяє мінімізувати механічні нелінійності та перекласти задачу забезпечення високих динамічних показників на інтелектуальні алгоритми керування, які будуть розглянуті в наступних розділах роботи. Врахування механічних обмежень, таких як змінний момент інерції та кінцева жорсткість конструкції, є обов’язковою умовою для синтезу ефективного регулятора.

1.2 Датчики положення та орієнтації

Побудова високоточної замкненої системи автоматичного керування поворотною платформою неможлива без надійної підсистеми вимірювання, яка забезпечує зворотний зв'язок за положенням, швидкістю та

прискоренням. Вибір сенсорів є критичним етапом проектування, оскільки метрологічні характеристики вимірювальних перетворювачів — роздільна здатність, лінійність, рівень шумів та смуга пропускання — визначають гранично досяжну точність позиціонування та запас стійкості системи в цілому. У контексті керування поворотними платформами задачу вимірювання доцільно розділити на два аспекти: визначення відносного кутового переміщення валу двигуна для комутації обмоток і замикання контуру швидкості, та визначення абсолютної просторової орієнтації платформи для задач стабілізації та наведення.

Найбільш поширеним класом датчиків для вимірювання кутових переміщень у прецизійних електроприводах є оптичні енкодери. Принцип їхньої дії базується на модуляції світлового потоку, що проходить через рухомий диск із нанесеними на нього прозорими та непрозорими мітками. Серед них виділяють інкрементальні та абсолютні перетворювачі. Інкрементальні енкодери генерують послідовність імпульсів при обертанні валу, а інформація про поточне положення отримується шляхом підрахунку цих імпульсів відносно певної початкової точки. Використання двох вихідних каналів, зміщених по фазі на 90 електричних градусів, дозволяє не лише визначити напрямок обертання, але й підвищити ефективну роздільну здатність у чотири рази за допомогою квадратурної обробки сигналу. Головним недоліком інкрементальних систем є втрата інформації про положення при зникненні живлення, що вимагає виконання процедури ініціалізації (пошуку референтної мітки) при кожному запуску системи. Це може бути неприпустимим для платформ з обмеженим кутом повороту або у випадках, коли рух до початку роботи заборонений з міркувань безпеки.

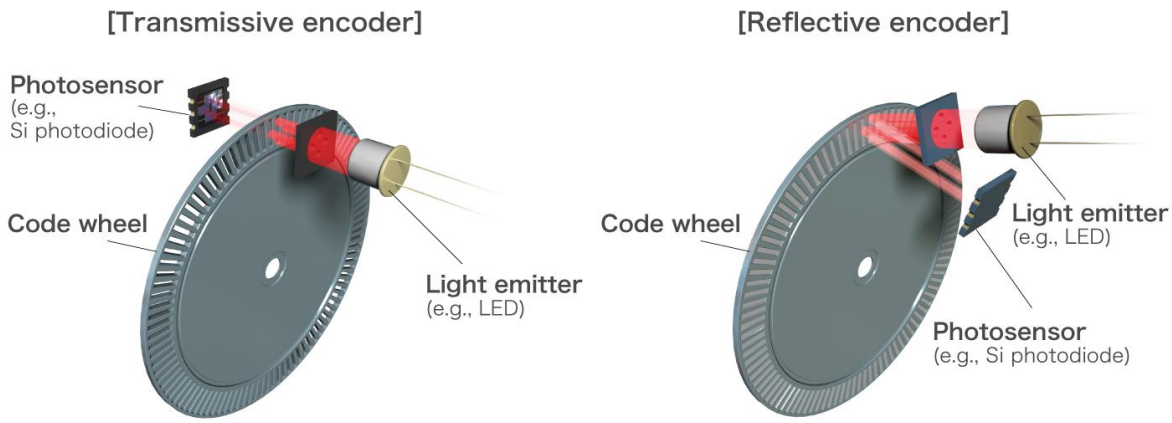

Рисунок 1.9– Оптичні сенсори визначення повороту із використанням кодового колеса на просвіт і на відбивання

Абсолютні енкодери позбавлені цього недоліку, оскільки кожному кутовому положенню валу відповідає унікальний цифровий код (зазвичай код Грея або двійковий код), нанесений на кодовий диск у вигляді кількох доріжок. Це дозволяє контролеру зчитувати точне положення платформи безпосередньо після подачі живлення без необхідності виконання калібрувальних рухів. Для передачі даних від абсолютних енкодерів використовуються високошвидкісні послідовні інтерфейси (SSI, BiSS, EnDat), які забезпечують високу завадостійкість та дозволяють передавати додаткову діагностичну інформацію, наприклад, температуру датчика або попередження про забруднення оптичної системи. Важливим параметром оптичних енкодерів є похибка квантування, яка вносить високочастотний шум у контур керування, особливо при диференціюванні сигналу положення для отримання швидкості. Збільшення розрядності енкодера знижує цей шум, проте підвищує вартість та вимоги до обчислювальної потужності контролера [11].

Альтернативою оптичним системам, які є чутливими до забруднення, вібрацій та ударних навантажень, виступають магнітні енкодери. Їхня робота базується на використанні ефекту Холла або магніторезистивного ефекту для детектування зміни магнітного поля постійного магніту, закріпленого на валу двигуна. Магнітні датчики відрізняються високою надійністю, компактністю та стійкістю до умов навколишнього середовища (пил, волога, мастило), що

робить їх привабливими для використання у мобільних робототехнічних комплексах. Історично магнітні енкодери поступалися оптичним у точності через нелінійність магнітного поля та температурний дрейф. Однак сучасні моделі з вбудованими мікроконтролерами здійснюють автоматичну лінеаризацію та інтерполяцію сигналу, що дозволяє досягти роздільної здатності, співмірної з оптичними аналогами середнього класу. Тим не менш, при використанні магнітних енкодерів у системах з високими магнітними полями (наприклад, поблизу потужних електродвигунів) необхідно передбачати додаткове екранування для запобігання спотворенням вимірювань [12].

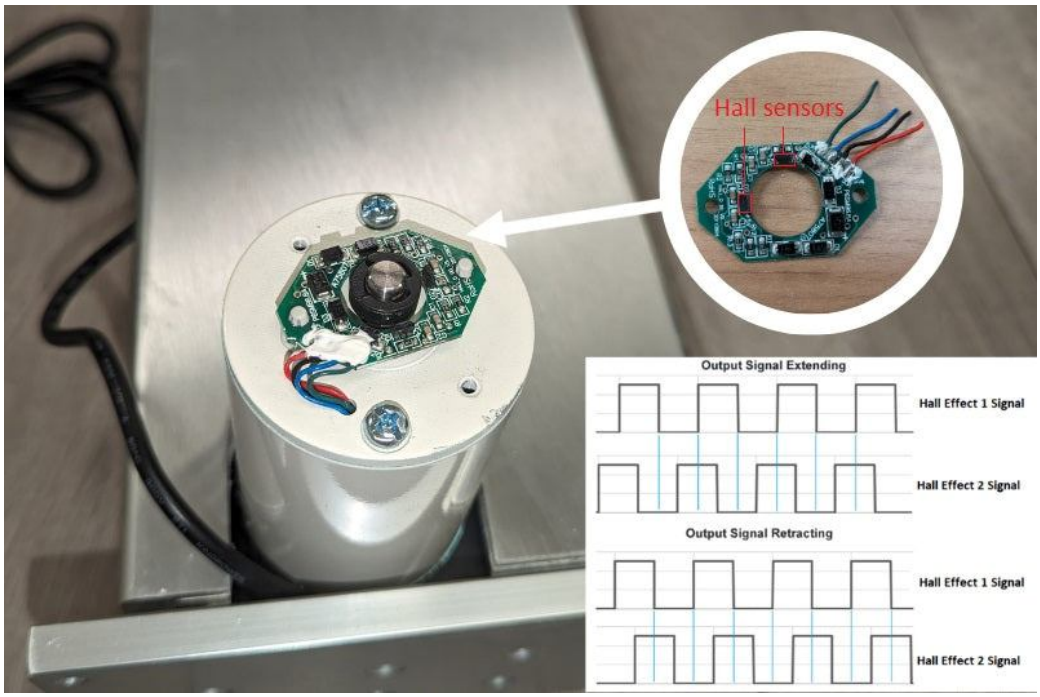

Рисунок 1.10– Використання датчиків хола для визначення нопрямку повороту ротора

Для задач стабілізації платформи, особливо коли вона встановлена на рухомій основі (транспортному засобі, літальному апараті), інформації лише про кут повороту валу недостатньо. У таких випадках необхідне використання інерціальних вимірювальних модулів (IMU), які зазвичай включають триосьові гіроскопи, акселерометри та, іноді, магнетометри. Мікроелектромеханічні (MEMS) гіроскопи вимірюють кутову швидкість

обертання платформи в інерціальному просторі. Цей сигнал є критично важливим для контурів швидкісної стабілізації, оскільки він дозволяє компенсувати зовнішні збурення ще до того, як вони призведуть до значного відхилення за положенням. Проте, отримання кута орієнтації шляхом інтегрування сигналу гіроскопа призводить до накопичення похибки з часом через наявність низькочастотного шуму та температурного дрейфу нуля (bias instability) [13].

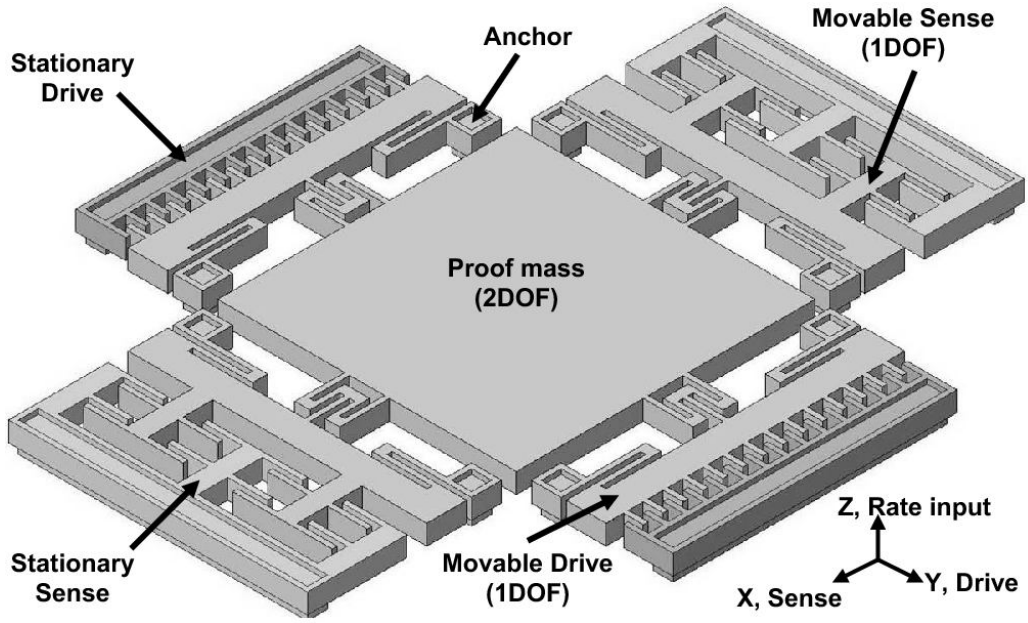

Рисунок 1.11– Будова твердотільного MEMS гіроскопа

Акселерометри вимірюють проекцію вектора уявного прискорення (сума кінематичного прискорення та прискорення вільного падіння) на осі чутливості датчика. У статичному режимі акселерометр дозволяє точно визначити кут нахилу платформи відносно горизонту (вектора гравітації). Однак при динамічному русі платформи сигнал акселерометра "забруднюється" власними прискореннями об'єкта та вібраціями, що робить його непридатним для безпосереднього використання в контурі зворотного зв'язку без попередньої фільтрації. Магнетометри забезпечують абсолютну прив'язку до магнітного поля Землі, що дозволяє коригувати дрейф гіроскопа за курсом (yaw), проте вони є надзвичайно чутливими до магнітних завад від

феромагнітних елементів конструкції та струмів у двигунах (hard-iron та soft-iron distortions), що вимагає складних процедур калібрування [14].

Порівняльний аналіз датчиків з точки зору їх використання в замкненому контурі керування виявляє необхідність комплексування інформації від різних джерел. Енкодери забезпечують високу точність та низький рівень шуму, але вимірюють лише відносний рух. MEMS-гіроскопи мають високу швидкодію та низьку затримку, що дозволяє розширити смугу пропускання системи, але страждають від дрейфу. Акселерометри та магнетометри мають довготривалу стабільність, але високий рівень шуму та чутливість до вібрацій. Важливим аспектом також є фазова затримка сигналу. Цифрові датчики з внутрішньою фільтрацією можуть вносити транспортне запізнення, що негативно впливає на запас стійкості за фазою. Тому для побудови високоякісної системи керування поворотною платформою доцільно застосовувати методи сенсорної фузії (наприклад, фільтр Калмана або комплементарний фільтр), які дозволяють об'єднати високочастотну динаміку гіроскопа з низькочастотною точністю акселерометра та енкодера, мінімізуючи вплив шумів та дрейфу кожного окремого сенсора [15].

1.3 Класичні методи керування

Незважаючи на стрімкий розвиток адаптивних та нейромережевих алгоритмів, фундаментом промислової автоматизації поворотних платформ залишаються детерміновані методи керування. Їхня широка розповсюдженість зумовлена прозорістю математичного апарату, передбачуваністю поведінки в штатних режимах та наявністю розроблених критеріїв оцінки стійкості. У задачах прецизійного позиціонування платформи мета керування зводиться до мінімізації динамічної помилки неузгодженості між заданим та дійсним вектором стану системи. Аналіз існуючих рішень дозволяє виділити чотири основні групи регуляторів, що знайшли застосування в цій предметній області: ПІД-регулятори, оптимальні

регулятори у просторі станів (LQR), системи зі змінною структурою (ковзний режим) та регулятори на базі нечіткої логіки.

Базовим стандартом керування електромеханічними системами є ПІД-регулювання (пропорційно-інтегрально-диференціальне) та його варіації. У сучасних сервоприводах поворотних платформ загальноприйнятою є каскадна структура, що складається з вкладених контурів струму, швидкості та положення. Пропорційна ланка забезпечує швидкодію системи, реагуючи на миттєве значення помилки, проте її високий коефіцієнт підсилення може призвести до автоколивань. Інтегральна складова необхідна для астатизму системи — усунення статичної похибки, викликаної постійними збуреннями, такими як гравітаційний момент при нахилі платформи або момент сухого тертя. Диференціальна ланка виконує функцію демпфування, прогнозуючи зміну помилки та запобігаючи перерегулюванню. Головним недоліком класичного ПІД-регулятора з фіксованими коефіцієнтами є його лінійна природа, яка вступає в протиріччя зі змінними параметрами об'єкта. Наприклад, зміна моменту інерції платформи внаслідок встановлення різного корисного навантаження призводить до того, що налаштування, оптимальні для одного режиму, стають причиною нестійкості або в'ялості перехідного процесу в іншому. Крім того, наявність інтегратора породжує проблему «насичення» (windup) при виході на обмеження керуючого впливу, що вимагає введення додаткових алгоритмічних компенсаторів [16].

Другим поширеним підходом є керування у просторі станів, зокрема використання лінійно-квадратичного регулятора (LQR). На відміну від ПІД-регуляторів, що оперують лише помилкою виходу, методи простору станів враховують внутрішні змінні системи (положеня, швидкість, струм) у вигляді вектора станів. Синтез LQR-регулятора базується на мінімізації квадратичного функціоналу якості, який враховує витрати енергії на керування та точність підтримки стану. Це дозволяє знайти оптимальний баланс між швидкодією приводу та енергоефективністю, що є важливим для

автономних платформ з живелнням від акумуляторів. Реалізація закону керування у вигляді лінійного зворотного зв'язку за станом ($u=-Kx$) забезпечує гарантовані запаси стійкості для лінійної моделі. Проте для практичної реалізації цього методу необхідна наявність повної інформації про всі змінні стану. Оскільки безпосереднє вимірювання всіх змінних (наприклад, прискорення або похідної струму) часто є технічно складним або дорогим, систему доповнюють спостерігачами стану (наприклад, спостерігачем Люенбергера або фільтром Калмана). Обмеженням методу є його висока чутливість до точності математичної моделі: розбіжність між параметрами моделі, закладеної в спостерігач, та реального об'єкта може призвести до значних похибок оцінювання та втрати стійкості [17].

Для забезпечення робастності (нечутливості) до параметричних збурень та зовнішніх завад застосовують керування зі змінною структурою, найбільш відомим варіантом якого є ковзний режим (Sliding Mode Control — SMC). Ідея методу полягає в організації розривного керування, яке змушує зображуючу точку системи рухатися вздовж заданої поверхні ковзання у просторі станів. Головною перевагою SMC є теоретична інваріантність до збурень, що задовольняють умову узгодженості: як тільки система потрапляє на поверхню ковзання, її динаміка визначається лише параметрами самої поверхні, а не параметрами об'єкта (масою, моментом інерції). Це робить SMC надзвичайно привабливим для керування поворотними платформами зі змінним навантаженням. Однак класичний ковзний режим має суттєвий недолік — явище високочастотної комутації керуючого сигналу (chattering). Оскільки ідеальне перемикання з нескінченною частотою неможливе, реальний привід зазнає вібрацій з кінцевою амплітудою та частотою. Для механічних систем з редукторами це є критичним, оскільки призводить до прискореного зносу зубчастих передач, нагріву двигунів та збудження високочастотних резонансів конструкції. Сучасні модифікації методу (Higher

Order Sliding Mode) дозволяють зменшити цей ефект, але складність реалізації при цьому зростає [18].

Четвертим класом систем, що набув статусу «сучасної класики» в інженерній практиці, є регулятори на базі нечіткої логіки (Fuzzy Logic Control — FLC). Цей підхід дозволяє формалізувати емпіричний досвід оператора або експерта у вигляді лінгвістичних правил типу «ЯКЩО помилка велика позитивна, ТО керуюча напруга велика позитивна». На відміну від попередніх методів, FLC не вимагає точної аналітичної моделі об'єкта керування, що є значною перевагою для складних нелінійних систем з люфтами та сухим тертям, які важко піддаються математичному опису. Процес керування включає етапи фазифікації (перетворення числових значень помилки у функції належності), логічного виведення на основі бази правил та дефазифікації (обчислення чіткого значення керуючого впливу). Нечіткі регулятори часто демонструють кращу якість перехідних процесів при значних нелінійності, ніж лінійні ПІД-регулятори, забезпечуючи плавний підхід до цільової точки без перерегулювання. Однак чисті нечіткі регулятори не гарантують аналітичної стійкості системи, а їх налаштування (вибір функцій належності та правил) часто здійснюється методом проб і помилок. Тому на практиці найбільш поширеними є гібридні схеми, наприклад, нечіткий ПІД-регулятор, де нечітка логіка використовується для адаптивного підлаштування коефіцієнтів класичного регулятора в реальному часі залежно від поточної помилки [19].

Підсумовуючи аналіз класичних методів, можна стверджувати, що кожен з них має свою сферу ефективного застосування, обмежену фізичними властивостями поворотної платформи. ПІД-регулятори прості, але вимагають компромісу між точністю та запасом стійкості при змінному навантаженні. LQR забезпечує оптимальність, але залежить від точності моделі. Ковзний режим гарантує робастність ціною підвищеного зносу механіки, а нечітка логіка ефективна при невизначеностях, але складна в аналітичному

обґрунтуванні. В умовах, коли до системи висуваються жорсткі вимоги одночасно щодо точності, швидкодії та адаптивності до широкого діапазону мас навантаження, використання окремо взятого класичного методу часто не дозволяє досягти бажаних показників. Це обумовлює необхідність переходу до інтелектуальних методів керування, здатних до навчання та самоорганізації, що буде розглянуто в наступному підрозділі [20].

1.4 Інтелектуальні методи керування

Розвиток обчислювальних потужностей сучасних мікроконтролерів та цифрових сигнальних процесорів створив передумови для впровадження в системи автоматизації методів обчислювального інтелекту (Computational Intelligence), які дозволяють долати обмеження класичної теорії керування. У задачах високоточного позиціонування поворотних платформ, де присутні суттєві нелінійності, змінність параметрів навантаження та складна динаміка тертя, традиційні аналітичні методи синтезу регуляторів часто виявляються неефективними або надмірно складними в реалізації. Інтелектуальні методи керування, що базуються на апараті штучних нейронних мереж, еволюційних алгоритмів та нечітких систем, пропонують альтернативний підхід, заснований на обробці даних та навчанні, а не на жорсткому математичному моделюванні фізичних процесів. Цей напрямок є особливо актуальним для створення адаптивних електромеханічних систем, здатних самостійно налаштовуватися під зміни умов експлуатації без втручання оператора.

Одним із найбільш перспективних напрямків є використання штучних нейронних мереж (ШНМ) як універсальних апроксиматорів нелінійних функцій. У контексті керування поворотною платформою нейронні мережі можуть застосовуватися в різних конфігураціях: як прямі контролери, що замінюють класичні ПІД-регулятори, або як допоміжні елементи для ідентифікації моделі об'єкта та компенсації збурень. Фундаментальною властивістю ШНМ є їхня здатність відтворювати складні відображення вхідних сигналів (наприклад, помилки положення та її похідних) у вихідний

керуючий вплив без необхідності апріорного знання рівнянь динаміки системи. Нейроконтролери здатні ефективно компенсувати такі явища, як зона нечутливості, гістерезис магнітних матеріалів та нелінійне тертя, навчаючись на експериментальних даних реальної поведінки платформи. Наприклад, нейромережа може бути навчена генерувати додатковий момент для компенсації ефекту Штрібека при малих швидкостях, що практично неможливо реалізувати за допомогою лінійних законів керування [21].

Проте використання класичних нейронних мереж прямого поширення (Feedforward Neural Networks), таких як багатошаровий персептрон, має певні обмеження при керуванні динамічними об'єктами. Мережі прямого поширення здійснюють статичне відображення входу на вихід і не мають внутрішньої пам'яті про попередні стани системи. Водночас, поведінка поворотної платформи є інерційною: її поточний стан залежить не лише від поточного керуючого впливу, але й від передісторії руху (накопиченої кінетичної енергії, розігріву мастила, пружних деформацій валів). Для адекватного врахування динаміки процесу доцільно застосовувати рекурентні нейронні мережі (RNN), які містять зворотні зв'язки між нейронами прихованих шарів або від виходу до входу. Наявність зворотних зв'язків формує внутрішній стан мережі, який діє як короткочасна пам'ять, дозволяючи враховувати часові залежності в сигналах керування.

Серед рекурентних структур особливу увагу в задачах керування привертають мережі Елмана та більш складні архітектури, такі як мережі довгої короткочасної пам'яті (LSTM) або керовані рекурентні блоки (GRU). Ці структури здатні виявляти та запам'ятовувати довготривалі часові залежності, що є критично важливим для прогнозування поведінки платформи при складних траєкторіях руху або при наявності транспортного запізнення в каналах вимірювання. Застосування RNN дозволяє реалізувати схеми прогнозуючого керування (Model Predictive Control), де нейромережа виступає в ролі емулятора прямої динаміки об'єкта, передбачаючи майбутнє

положення платформи на кілька кроків вперед. Це дає змогу регулятору формувати випереджуючі впливи, мінімізуючи динамічну помилку ще до її фактичного виникнення. Дослідження показують, що рекурентні нейроконтролери забезпечують значно кращу якість перехідних процесів при різких змінах завдання порівняно зі статичними нейронними мережами, особливо в умовах змінного моменту інерції [22].

Ключовим етапом розробки інтелектуальної системи керування є процес навчання, тобто налаштування вагових коефіцієнтів нейронної мережі або параметрів класичного регулятора. Традиційні методи навчання нейронних мереж, такі як алгоритм зворотного поширення помилки (Backpropagation), базуються на градієнтних методах оптимізації. Вони вимагають, щоб цільова функція була диференційовною, а поверхня помилки — гладкою. Однак у реальних електромеханічних системах критерії якості часто є складними, багатомодальними (мають багато локальних мінімумів) та можуть включати недиференційовні обмеження (наприклад, насичення напруги живлення або обмеження струму). У таких випадках градієнтні методи часто «застрягають» у локальних екстремумах, не досягаючи глобального оптимуму налаштувань.

Для вирішення цієї проблеми ефективно застосовуються еволюційні методи оптимізації, зокрема генетичні алгоритми (GA). Генетичні алгоритми — це стохастичні методи пошуку, натхненні природним відбором та генетикою. Замість руху в напрямку антиградієнта, GA оперує популяцією потенційних рішень (хромосом), кожна з яких кодує набір параметрів регулятора (наприклад, коефіцієнти ПІД-регулятора або ваги нейронної мережі). Процес оптимізації відбувається через ітеративну зміну поколінь за допомогою операторів селекції (відбору найкращих рішень), схрещування (обміну інформацією між рішеннями) та мутації (випадкової зміни параметрів). Головною перевагою генетичних алгоритмів є їхня здатність знаходити глобальний оптимум у складних просторах пошуку без

необхідності обчислення похідних. Це дозволяє використовувати їх для прямого налаштування регуляторів за інтегральними критеріями якості (наприклад, мінімум інтеграла квадрата помилки або мінімум часу регулювання) безпосередньо на моделі об'єкта або навіть на реальному обладнанні [23].

Поєднання нейронних мереж та генетичних алгоритмів відкриває шлях до створення нейроеволюційних систем керування. У такому підході генетичний алгоритм використовується для оптимізації структури (кількості нейронів, шарів) та вагових коефіцієнтів нейроконтролера. Це дозволяє автоматизувати процес синтезу регулятора, виключаючи необхідність ручного підбору архітектури мережі. Крім того, еволюційні методи дозволяють враховувати в цільовій функції (Fitness Function) не лише точність позиціонування, але й енергоефективність, плавність ходу та обмеження на керуючі впливи, формуючи збалансоване рішення, що задовольняє суперечливим вимогам технічного завдання. Такий підхід є особливо доцільним для поворотних платформ спеціального призначення, де вимоги до надійності та точності превалюють над обчислювальними затратами на етапі проектування [24].

Незважаючи на очевидні переваги, впровадження інтелектуальних методів керування пов'язане з низкою викликів. По-перше, нейронні мережі часто сприймаються як «чорна скринька» (black box), внутрішня структура якої не має прозорої фізичної інтерпретації. Це ускладнює аналіз стійкості системи класичними методами (наприклад, за критерієм Найквіста) та сертифікацію системи для критичних застосувань. По-друге, обчислювальна складність навчання нейронних мереж, особливо рекурентних, може бути надмірною для вбудованих систем реального часу. Тому на практиці часто застосовують компромісні рішення: навчання мережі відбувається офлайн (на потужній робочій станції) з використанням точної математичної моделі або зібраних експериментальних даних, а отримані фіксовані вагові коефіцієнти

завантажуються в мікроконтролер для виконання в реальному часі. Іншим варіантом є використання гібридних схем, де інтелектуальний модуль лише коригує параметри надійного класичного регулятора, забезпечуючи базову працездатність навіть у випадку збою адаптивного алгоритму.

Аналіз сучасного стану досліджень дозволяє зробити висновок про доцільність застосування інтелектуальних методів для керування поворотною платформою в рамках даної кваліфікаційної роботи. Враховуючи вимоги до адаптивності системи в умовах зміни корисного навантаження та необхідність компенсації нелінійного тертя, найбільш перспективним вбачається використання гібридного підходу. Він може полягати у застосуванні рекурентної нейронної мережі для прогнозування динаміки платформи та адаптивного налаштування контуру позиціонування, параметри якої оптимізуються за допомогою генетичного алгоритму на етапі попереднього навчання. Такий підхід дозволить поєднати робастність інтелектуальних алгоритмів з передбачуваністю класичних структур, забезпечуючи високу точність позиціонування при допустимих обчислювальних витратах [25].

2. РОЗРОБКА МОДЕЛІ ФІЗИЧНОГО ОБЄКТУ ДЛЯ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ

2.1 Узагальнена фізична модель об’єкта

Побудова математичної моделі механічної системи є першочерговим етапом проектування алгоритмів автоматичного керування. Для задач, де об’єктом виступає поворотна платформа з довільно розташованим вантажем, модель повинна забезпечувати перехід від фізичних параметрів конструкції до динамічних характеристик, придатних для аналізу в теорії автоматичного керування. Метою цього розділу є отримання передавальної функції системи, яка дозволяє розглядати платформу як «чорну скриньку», на вхід якої подається функція сили $F(t)$, а на виході формуються кінематичні змінні руху — кутова швидкість $\omega(t)$ та кут повороту $\theta(t)$. Такий підхід необхідний для подальшого використання генетичних алгоритмів, які вимагають багаторазового прогону симуляції для оцінки якості керування.

Конструктивно платформа розглядається як тверде тіло, що обертається навколо нерухомої вертикальної осі Z. Основою системи є диск радіусом R_{disk} та масою M_{disk}, центр якого збігається з віссю обертання. Специфіка задачі полягає у наявності корисного навантаження — об’єкта масою m_{load}, який закріплений на поверхні диска на відстані r від центру. Геометричне розташування вантажу призводить до того, що загальний центр мас системи зміщується відносно осі обертання. Це перетворює систему на ексцентричний ротор, динамічні параметри якого залежать від радіуса r.

Визначальним параметром, що характеризує інертність системи при обертальному русі, є момент інерції J_{Σ}. Оскільки вісь обертання не проходить через центр мас вантажу, для розрахунку сумарного моменту інерції застосовується теорема Гюйгенса-Штейнера про паралельний перенос осей. Момент інерції системи складається з власного моменту інерції диска та моменту інерції вантажу, перерахованого відносно осі обертання:

$$J\Sigma = J_{disk} + m_{load} \cdot r^2 = \frac{1}{2} M_{disk} \cdot {R_{disk}}^2 + m_{load} \cdot r^2 \qquad (2.1)$$

Отримана величина JΣ є коефіцієнтом пропорційності між прикладеним обертовим моментом та кутовим прискоренням. Квадратична залежність від радіуса r вказує на те, що навіть незначне зміщення вантажу до периферії платформи призводить до суттєвого зростання інерційного опору. У контексті моделювання це означає, що параметр JΣ є змінною величиною, яка задається перед початком симуляції залежно від конфігурації установки.

Рух платформи відбувається під дією активного керуючого моменту Mdrive(t), який створюється приводом. У математичній моделі вхідним впливом вважається тангенціальна сила Fτ(t), прикладена до ободу платформи (або приводного шківа) на відстані Rapp від осі. Зв'язок між силою та моментом є лінійним:

$$M_{drive}(t) = F_{\tau}(t) \cdot R_{app} \qquad (2.2)$$

Ця сила Fτ(t) є функцією часу і може набувати довільної форми залежно від логіки роботи контролера.

Окрім активного моменту, на платформу діють моменти опору. Для отримання базової аналітичної передавальної функції, яка описує основну динаміку процесу, опір руху моделюється як в'язке тертя. Сила в'язкого тертя пропорційна швидкості обертання і спрямована протилежно напрямку руху. Момент в'язкого тертя визначається як добуток коефіцієнта в'язкого тертя B на кутову швидкість ω(t). Коефіцієнт B враховує опір у підшипниках та аеродинамічний опір диска.

Динаміка системи описується диференціальним рівнянням другого закону Ньютона для обертального руху:

$$J_{\Sigma} \frac{d\omega(t)}{dt} = M_{drive}(t) - M_{friction}(t) \qquad (2.3)$$

Підставляючи вирази для моментів, отримуємо лінійне диференціальне рівняння першого порядку відносно швидкості:

$$J_\Sigma \frac{d\omega(t)}{dt} = F_\tau(t) \cdot R_{app} - B \cdot \Omega(t) \quad (2.4)$$

Для переходу від диференціального рівняння до передавальної функції застосуємо перетворення Лапласа. Цей метод дозволяє замінити операції диференціювання та інтегрування на алгебраїчні операції з комплексною змінною s. Вважаючи початкові умови нульовими (ω(0)=0), рівняння в зображеннях набуває вигляду:

$$J_\Sigma \cdot s \cdot \Omega(s) = F(s) \cdot R_{app} - B \cdot \Omega(s) \quad (2.5)$$

де Ω(s) та F(s) — зображення Лапласа для функцій швидкості та сили відповідно. Групуючи доданки зі швидкістю, отримуємо:

$$\Omega(s) \cdot (J_\Sigma \cdot s + B) = F(s) \cdot R_{app} \quad (2.6)$$

Звідси виводиться передавальна функція за швидкістю $W_\omega(s)$, яка є відношенням вихідного сигналу до вхідного:

$$W_\omega(s) = \frac{\Omega(s)}{F(s)} = \frac{R_{app}}{J_\Sigma \cdot s + B} \quad (2.7)$$

Для приведення виразу до стандартної форми динамічної ланки чисельник і знаменник діляться на коефіцієнт B:

$$W_\omega(s) = \frac{R_{app} / B}{J_\Sigma / B \cdot s + 1} = \frac{K_\omega}{T_m \cdot s + 1} \quad (2.8)$$

Отриманий вираз класифікує динаміку поворотної платформи (за швидкістю) як аперіодичну ланку першого порядку. Тут виділяються два ключові параметри. Перший — коефіцієнт передачі $K_\omega = R_{app} / B$, який визначає усталене значення швидкості при дії одиничної постійної сили. Другий параметр — електромеханічна стала часу $T_m = J_\Sigma / B$. Стала часу характеризує інерційність системи: це час, за який швидкість досягає 63,2% від свого усталеного значення при стрибкоподібній зміні вхідної сили. Оскільки Tm прямо пропорційна моменту інерції, зміна положення вантажу r

безпосередньо змінює цей параметр. Збільшення радіуса r призводить до зростання Tm, роблячи систему більш «повільною» та інертною.

Однак кінцевою метою керування є позиціонування платформи, тобто контроль кута повороту θ(t). Кут повороту є інтегралом від кутової швидкості в часі:

$$\theta(t) = \int_0^t \omega(\tau) \cdot d\tau \qquad (2.9)$$

В області перетворення Лапласа операції інтегрування відповідає ділення на змінну s. Таким чином, зображення кута $\Theta(s)$ пов'язане із зображенням швидкості співвідношенням $\Theta(s) = \frac{1}{s}\Omega(s)$.

Об'єднуючи це з попередньою передавальною функцією, отримуємо повну математичну модель об'єкта «сила — кут»:

$$W_\theta(s) = \frac{\Theta(s)}{F(s)} = \frac{1}{s}\frac{K_\omega}{T_m \cdot s + 1} = \frac{K_\omega}{s(T_m \cdot s + 1)} \qquad (2.10)$$

Ця передавальна функція описує систему як послідовне з'єднання аперіодичної ланки (інерція та тертя) та ідеальної інтегруючої ланки (перехід від швидкості до переміщення). З точки зору теорії керування, такий об'єкт є астатичним першого порядку. Це означає, що система не має фіксованого положення рівноваги при нульовому керуванні (вона зберігає попереднє положення) і постійно змінює положення при наявності постійного вхідного впливу.

Практичне застосування цієї моделі в середовищі симуляції для генетичного алгоритму виглядає наступним чином. Вхідний масив даних, що представляє функцію сили $F(t)$ (генеровану хромосомою алгоритму або контролером), подається на вхід блоку, що реалізує передавальну функцію $W_\theta(s)$. У цифровому вигляді (дискретний час з кроком Δt) це реалізується через різницеві рівняння. Переходячи назад у часову область через обернене

перетворення або дискретизацію методом Ейлера, отримуємо рекурентну формулу для обчислення швидкості на кроці k:

$$\omega_k = \omega_{k-1} + \frac{\Delta t}{J_\Sigma}(F_{k-1} \cdot R_{app} - B \cdot \omega_{k-1}) \qquad (2.11)$$

І для положення:

$$\theta_k = \theta_{k-1} + \omega_k \cdot \Delta t \qquad (2.12)$$

Саме ці рівняння є програмною реалізацією ередавальної функції Wθ(s) для обчислювального ядра симулятора.

Для підвищення точності моделювання необхідно врахувати, що реальна система має нелінійності, які не увійшли до лінійної передавальної функції. Зокрема, це момент сухого тертя $M_{coulomb}$ та динамічний момент від дисбалансу, викликаний ексцентриситетом маси. У структурі моделі ці фактори враховуються як збурення. Вхідна сила $F_\tau(t)$ перед подачею на лінійну ланку коригується шляхом віднімання компонент суми моментів опору.

$$F_{eff}(t) = F_\tau(t) - \frac{1}{R_{app}}\left[M_{coulomb} \cdot \mathrm{sgn}(\omega) + M_{unbalance}(\omega, r)\right] \qquad (2.13)$$

Отримана ефективна сила F_{eff} є тим сигналом, який «множиться» на передавальну функцію $W_\theta(s)$ (або обробляється різницевим рівнянням). Такий підхід дозволяє зберегти зручну для аналізу структуру передавальної функції, водночас враховуючи складну фізику взаємодії вантажу з платформою.

Момент дисбалансу $M_{unbalance}$ виникає через те, що при обертанні ексцентрично розташованої маси виникає відцентрова сила $F_c = m_{load} \cdot \omega^2 \cdot r$. Хоча ця сила діє радіально і не створює безпосереднього гальмуючого моменту в ідеальному випадку, в реальних підшипниках вона збільшує силу нормального тиску на тіла кочення, що призводить до зростання моменту тертя. Тому в повній моделі коефіцієнт B може розглядатися не як константа,

а як функція від квадрату швидкості, або ж цей ефект додається до вектора зовнішніх збурень.

Таким чином, розроблена математична модель дозволяє однозначно визначити траєкторію руху платформи $\theta(t)$ та $\omega(t)$ для будь-якої заданої функції хідної сили $F(t)$. Модель враховує зміну інерційних властивостей системи залежно від розташування навантаження через параметр Tm та забезпечує можливість імітації як перехідних процесів, так і усталених режимів руху. Це створює необхідний фундамент для роботи оптимізаційних алгоритмів, які, варіюючи параметри функції керування $F(t)$, шукатимуть такі її форми, що забезпечать переміщення платформи у задану позицію за мінімальний час або з мінімальною енергією.

2.2 Розробка передаточної функції колекторного двигуна

Побудова математичної моделі системи керування поворотною платформою вимагає детального аналізу динамічних характеристик виконавчого механізму. У розглядуваній системі таким механізмом виступає колекторний двигун постійного струму, який забезпечує обертання платформи навколо вертикальної осі. Передаточна функція цього двигуна описує залежність між вхідною керуючою напругою та результуючою кутовою швидкістю або кутовим переміщенням валу.

Фізична природа процесів у колекторному двигуні визначається взаємодією електромагнітних та механічних явищ. Коли до обмотки якоря прикладається напруга, виникає електричний струм, який створює обертальний момент через взаємодію з магнітним полем статора. Цей момент долає момент інерції ротора та навантаження, а також протидіє моменту тертя в підшипниках та аеродинамічному опору.

Для виведення передаточної функції необхідно розглянути дві взаємопов'язані підсистеми. Електрична підсистема описується рівнянням для контуру якоря, де прикладена напруга врівноважується падінням напруги на

активному опорі обмотки, індуктивною складовою та електрорушійною силою самоіндукції. Це рівняння записується у вигляді

$$U(t) = R_a I(t) + L_a \frac{dI(t)}{dt} + k_e \omega(t) \quad (2.14)$$

де $U(t)$ позначає напругу живлення, R_a являє собою активний опір якірної обмотки, $I(t)$ відповідає струму якоря, L_a характеризує індуктивність обмотки, k_e представляє конструктивну постійну протиелектрорушійної сили, а $\omega(t)$ визначає кутову швидкість валу двигуна.

Механічна підсистема характеризується рівнянням моментів, де електромагнітний момент витрачається на прискорення інерційних мас та подолання в'язкого тертя. Електромагнітний момент пропорційний струму якоря через конструктивну постійну моменту

$$M_e(t) = k_m I(t) \quad (2.15)$$

Рівняння руху обертових мас записується як баланс моментів

$$J \frac{d\omega(t)}{dt} = M_e(t) - b \cdot \omega(t) \quad (2.16)$$

де J означає сумарний момент інерції ротора та навантаження, b характеризує коефіцієнт в'язкого тертя, а інші змінні визначені раніше.

Застосування перетворення Лапласа до цих диференціальних рівнянь дозволяє перейти в операторну область, де алгебраїчні співвідношення значно спрощують аналіз. При нульових початкових умовах електричне рівняння в операторній формі набуває вигляду

$$U(s) = R_a I(s) + L_a \cdot s \cdot I(s) + k_e \cdot \Omega(s) \quad (2.17)$$

де ss s являє собою комплексну змінну Лапласа, а великі літери позначають зображення відповідних функцій часу. Механічне рівняння в операторній формі записується через підстановку електромагнітного моменту

$$J \cdot s \cdot \Omega(s) = k_m \cdot I(s) - b \cdot \Omega(s) \quad (2.18)$$

Виключення струму якоря з системи рівнянь передбачає виразження його з механічного рівняння

$$I(s) = \frac{(J \cdot s + b) \cdot \Omega(s)}{k_m} \qquad (2.19)$$

та підстановку отриманого виразу в електричне рівняння

$$U(s) = R_a \frac{(J \cdot s + b) \cdot \Omega(s)}{k_m} + L_a \cdot s \frac{(J \cdot s + b) \cdot \Omega(s)}{k_m} + k_e \cdot \Omega(s) \qquad (2.20)$$

Після розкриття дужок та групування членів отримується

$$U(s) = \Omega(s) \left[\frac{R_a \cdot (J \cdot s + b)}{k_m} + \frac{L_a \cdot s \cdot (J \cdot s + b)}{k_m} + k_e \right] \qquad (2.21)$$

Розкриваючи добутки та приводячи до спільного знаменника, маємо

$$U(s) = \Omega(s) \left[\frac{R_a \cdot J \cdot s + R_a \cdot b + L_a \cdot J \cdot s^2 + L_a \cdot b \cdot s + k_e \cdot k_m}{k_m} \right] \qquad (2.22)$$

Звідси передаточна функція за кутовою швидкістю визначається як відношення

$$W_\omega(s) = \frac{\Omega(s)}{U(s)} = \frac{k_m}{L_a \cdot J \cdot s^2 + (R_a J + L_a \cdot b) \cdot s + R_a b + k_e \cdot k_m} \qquad (2.23)$$

Знаменник цієї передаточної функції формується з електромеханічних параметрів системи. Коефіцієнт при квадраті операторної змінної $L_a J$ визначається добутком індуктивності якірної обмотки та сумарного моменту інерції. Коефіцієнт при першому степені $R_a J + L_a b$ включає вплив як електричних, так і механічних параметрів. Вільний член $R_a b + k_e k_m$ утворюється механічним демпфуванням та електромеханічним зв'язком через конструктивні постійні.

У багатьох практичних застосуваннях індуктивність якірної обмотки колекторних двигунів невеликої потужності виявляється досить малою порівняно з активним опором, що дозволяє знехтувати індуктивною складовою та покласти $L_a \approx 0$. Таке спрощення зменшує порядок системи та призводить до передаточної функції першого порядку

$$W_\omega(s) = \frac{k_m}{R_a \cdot J \cdot s + R_a b + k_e \cdot k_m} \quad (2.24)$$

Винесення R_a за дужки у знаменнику дозволяє записати передаточну функцію у стандартній формі аперіодичної ланки

$$W_\omega(s) = \frac{k_m}{R_a b + k_e \cdot k_m} \cdot \frac{1}{1 + \frac{R_a J}{R_a b + k_e \cdot k_m} s} \quad (2.25)$$

що еквівалентно запису

$$W_\omega(s) = \frac{K}{1 + T \cdot s} \quad (2.26)$$

де коефіцієнт передачі визначається як $K = \frac{k_m}{R_a b + k_e \cdot k_m}$, а постійна часу дорівнює $T = \frac{R_a J}{R_a b + k_e \cdot k_m}$.

Передаточна функція за кутовою швидкістю може бути перетворена в передаточну функцію за кутом повороту шляхом інтегрування. Оскільки кутове переміщення $\theta(t)$ є інтегралом від кутової швидкості, в операторній області справедливе співвідношення

$$\Theta(s) = \frac{\Omega(s)}{s} \quad (2.27)$$

Результуюча передаточна функція за кутом отримується діленням передаточної функції за швидкістю на оператор s

$$W_\theta(s) = \frac{\Theta(s)}{U(s)} = \frac{K}{s \cdot (1 + T \cdot s)} \quad (2.28)$$

Це представлення містить у знаменнику множник ss s, що характеризує астатизм першого порядку та забезпечує нульову статичну похибку при відпрацюванні постійного сигналу завдання.

Фізична інтерпретація отриманої передаточної функції розкриває часові характеристики двигуна. Постійна часу T визначає швидкість реакції системи на зміну керуючого сигналу. Чим більший момент інерції

навантаження або чим менший сумарний коефіцієнт демпфування $R_a b + k_e k_m$, тим повільніше система досягає встановленого режиму. Коефіцієнт передачі K визначає співвідношення між прикладеною напругою та встановленою кутовою швидкістю при постійному сигналі керування.

Для практичного застосування в системі керування платформою необхідно врахувати редуктор, який встановлюється між валом двигуна та платформою для збільшення моменту та зменшення швидкості. Якщо передаточне число редуктора позначити як іі і, то момент інерції платформи J_p , приведений до валу двигуна, визначається співвідношенням $J_{ПР} = \frac{J_P}{i^2}$, а кутова швидкість платформи пов'язана зі швидкістю валу двигуна як $\omega_p = \frac{\omega}{i}$. Це призводить до модифікації передаточної функції шляхом додавання множника $\frac{1}{j}$ та зміни моменту інерції у виразі для постійної часу.

Експериментальна ідентифікація параметрів передаточної функції може здійснюватися методом аналізу перехідної характеристики. Подаючи східчастий сигнал напруги амплітудою U_0 на вхід двигуна та реєструючи зміну кутової швидкості в часі, можна визначити постійну часу T як час, за який швидкість досягає значення 0,6320 від встановленого значення ω_∞, а коефіцієнт передачі обчислюється як $K = \frac{\omega_\infty}{U_0}$. Альтернативно застосовується частотний метод, коли на вхід подається синусоїдальний сигнал різної частоти та будуються амплітудно-частотна та фазо-частотна характеристики, з яких визначаються параметри передаточної функції.

2.3 Модель ударних і зовнішніх збурень

Реальна система керування поворотною платформою турелі функціонує в умовах різноманітних зовнішніх впливів, які відхиляють систему від заданого режиму роботи. Ці впливи можуть мати різну природу, тривалість та

інтенсивність, що вимагає детального аналізу їх впливу на динаміку системи та здатність системи керування компенсувати такі відхилення. Розробка адекватної математичної моделі збурень є необхідною умовою для синтезу робастної системи керування, здатної забезпечувати стабільне функціонування в широкому діапазоні експлуатаційних умов.

Класифікація зовнішніх збурень базується на їх часових характеристиках та фізичній природі. Найбільш критичними для систем позиціювання виявляються ударні впливи, які характеризуються великою амплітудою та малою тривалістю. Такі впливи можуть виникати внаслідок механічних поштовхів конструкції, вібрацій від суміжного обладнання, різких змін навантаження або аеродинамічних поривів вітру. Математичне представлення цих збурень повинно відображати їх імпульсний характер та враховувати інерційні властивості механічної системи.

Імпульсні моменти представляють собою короткочасні обертальні впливи на платформу, які можна описати через зміну кутового моменту системи. Згідно з теоремою про зміну кутового моменту, імпульс моменту дорівнює зміні кутового моменту твердого тіла

$$\int_{t_0}^{t_0+\Delta t} M(t)dt = J \cdot \Delta\omega \qquad (2.29)$$

де $M(t)$ позначає зовнішній обертальний момент, що діє протягом короткого інтервалу часу Δt, J являє собою момент інерції платформи відносно осі обертання, а Δω характеризує миттєву зміну кутової швидкості внаслідок удару. Для імпульсного впливу тривалістю, яка набагато менша за характерний час системи, можна записати

$$S = \int_{t_0}^{t_0+\Delta t} M(t)dt \qquad (2.30)$$

де S представляє імпульс моменту, що діє на систему. Цей імпульс спричиняє миттєву зміну швидкості

$$\Delta\omega = \frac{S}{J} \tag{2.31}$$

яка далі відпрацьовується системою керування як початкове відхилення від заданого режиму.

Математичне моделювання імпульсного впливу в часовій області здійснюється через дельта-функцію Дірака, яка описує ідеалізований миттєвий вплив

$$M_{imp}(t) = S \cdot \delta(t - t_0) \tag{2.32}$$

де $\delta(t - t_0)$ позначає дельта-функцію в момент часу t_0 . У реальних системах імпульс має скінченну тривалість, тому більш адекватною моделлю виступає прямокутний імпульс

$$M_{imp}(t) = \begin{cases} M_0, & t_0 \le t \le t_0 + \Delta t \\ 0, & \textit{інше} \end{cases} \tag{2.33}$$

де M_0 означає амплітуду моменту, а Δt характеризує тривалість впливу. Імпульс моменту в цьому випадку дорівнює $S = M_0 \Delta t$, що дозволяє пов'язати параметри моделі з фізичними характеристиками збурення.

Короткочасні зовнішні сили, прикладені до платформи на деякій відстані від осі обертання, створюють обертальний момент згідно з фундаментальним співвідношенням

$$M(t) = F(t) \cdot r \tag{2.34}$$

де $F(t)$ позначає величину зовнішньої сили, а r характеризує плече сили відносно осі обертання. Для турельної установки такі сили можуть виникати від віддачі при стрільбі, аеродинамічного тиску або механічного контакту з перешкодами. Моделювання часової залежності сили здійснюється через експоненційно затухаючу функцію

$$F(t) = F_0 e^{-\alpha(t-t_0)} H(t-t_0) \tag{2.35}$$

де F_0 являє собою початкову амплітуду сили, α характеризує швидкість затухання, $H(t-t_0)$ представляє функцію Хевісайда, яка забезпечує причинність впливу. Відповідний момент записується як

$$M(t) = M_0 e^{-\alpha(t-t_0)} H(t-t_0) \tag{2.36}$$

де $M_0 = F_0 \cdot r$ визначає початкову амплітуду обертального моменту.

Вплив імпульсного збурення на динаміку системи можна проаналізувати через перетворення Лапласа. Зображення імпульсного моменту у вигляді прямокутного імпульсу визначається як

$$M_{imp}(s) = M_0 \frac{1 - e^{-s \cdot \Delta t}}{s} \tag{2.37}$$

Для короткого імпульсу, коли $\Delta t \to \infty$, а $M_0 \to \infty$ так, що добуток $M_0 \Delta t = S$ залишається скінченним, зображення спрощується до

$$M_{imp}(s) = S \tag{2.38}$$

що відповідає властивості перетворення Лапласа для дельта-функції. Зображення експоненційно затухаючого моменту має вигляд

$$M(s) = \frac{M_0}{s + \alpha} \tag{2.39}$$

який характеризує аперіодичний перехідний процес у системі.

Реакція системи на імпульсне збурення визначається через передаточну функцію замкненої системи керування. Якщо передаточна функція розімкненої системи за моментом збурення до кута повороту позначити як $W_m(s)$, а передаточна функція регулятора як $W_R(s)$, то передаточна функція замкненої системи від збурюючого моменту до вихідного кута набуває вигляду

$$\Phi_M(s) = \frac{W_M(s)}{1 + W_R(s) \cdot W(s)} \qquad (2.40)$$

де $W(s)$ являє собою передаточну функцію об'єкта керування. Відхилення кута від заданого значення під дією імпульсного моменту визначається як

$$\Theta_{збур}(s) = \Phi_M(s) M_{імп}(s) \qquad (2.41)$$

Аналіз перехідного процесу в часовій області здійснюється через обернене перетворення Лапласа отриманого виразу.

Для системи з ПІ-регулятором, передаточна функція якого записується у вигляді

$$W_R(s) = K_P + \frac{K_I}{s} \qquad (2.42)$$

де K_P характеризує пропорційну складову, а K_I визначає інтегральну складову, замкнена система забезпечує астатизм першого порядку. Це означає, що при дії постійного збурюючого моменту статична похибка позиціювання дорівнює нулю завдяки інтегральній складовій регулятора. Для імпульсного збурення система повертається до заданого положення після перехідного процесу, характер якого визначається налаштуваннями регулятора.

Перевірка відновлення керування після збурення передбачає аналіз часу та характеру перехідного процесу при відпрацюванні імпульсного впливу. Критеріями якості відновлення виступають час регулювання, максимальне відхилення від заданого положення та характер коливального процесу. Час регулювання t_p визначається як мінімальний час, після якого відхилення системи не виходить за межі допустимої зони

$$|\theta(t) - \theta_{зад}| < \varepsilon, \qquad \forall t > t_p \qquad (2.43)$$

де ε позначає допустиму похибку позиціювання. Максимальне динамічне відхилення $\theta_{\max}$ характеризує найбільше відхилення платформи під час перехідного процесу

$$\theta_{\max} = \max_{t>t_0} |\theta(t) - \theta_{зад}| \quad (2.44)$$

Ступінь коливальності перехідного процесу оцінюється через перерегулювання, яке для системи позиціювання має бути мінімальним для забезпечення точного наведення турелі.

Числове моделювання реакції системи на імпульсні збурення здійснюється шляхом розв'язання диференціальних рівнянь системи з відповідними початковими умовами. При імпульсному впливі в момент t_0 початкові умови для кутової швидкості модифікуються згідно зі співвідношенням $\omega(t_0^+) = \omega(t_0^-) + \Delta\omega$, де $\Delta\omega = \frac{S}{J}$ визначає стрибок швидкості. Подальша еволюція системи описується звичайними рівняннями руху під дією керуючого впливу від регулятора, який намагається компенсувати виниклу похибку позиціювання.

Практична реалізація моделі збурень у середовищі моделювання передбачає генерацію випадкових імпульсних впливів з заданими статистичними характеристиками. Амплітуда імпульсів може розподілятися за нормальним законом з математичним сподіванням μ_S та середньоквадратичним відхиленням σ_S, а моменти виникнення збурень визначаються пуассонівським процесом з інтенсивністю λ. Така стохастична модель дозволяє оцінити робастність системи керування в умовах реалістичного випадкового навантаження.

2.4 Цифровий двійник системи керування платформою

Розробка цифрового двійника системи керування поворотною платформою передбачає створення дискретної моделі, яка імітує реальну

поведінку механічної системи з урахуванням динаміки двигуна, інерційних властивостей платформи та зворотного зв'язку від датчика положення. Така модель дозволяє проводити налаштування регуляторів, тестування алгоритмів керування та оцінку якості перехідних процесів без необхідності експериментів на реальному обладнанні.

Базова структура цифрового двійника складається з декількох взаємопов'язаних блоків. Перший блок моделює електричну динаміку двигуна, другий відображає механічну динаміку обертання платформи, третій імітує роботу датчика положення. Зв'язок між блоками здійснюється через обмін змінними стану на кожному кроці дискретизації за часом.

Моделювання колекторного двигуна базується на диференціальному рівнянні для струму якоря. Якщо до двигуна прикладена напруга U, то швидкість зміни струму визначається рівнянням

$$\frac{dI}{dt} = \frac{1}{L_a}(U - R_a I - k_e \omega) \qquad (2.45)$$

де I позначає струм якоря, L_a характеризує індуктивність обмотки, R_a представляє активний опір, k_e являє собою конструктивну постійну протиелектрорушійної сили, а ω визначає поточну кутову швидкість валу. Для малих двигунів індуктивність часто мала і можна використати спрощену модель, де струм встановлюється практично миттєво згідно з алгебраїчним співвідношенням

$$I = \frac{U - k_e \omega}{R_a} \qquad (2.46)$$

Електромагнітний момент двигуна пропорційний струму якоря через конструктивну постійну

$$M_e = k_m I \qquad (2.47)$$

Цей момент прикладається до механічної системи і визначає її обертальний рух.

Механічна динаміка платформи описується рівнянням обертального руху твердого тіла навколо нерухомої осі. Рівняння моментів записується у вигляді

$$J\frac{d\omega}{dt} = M_e - M_{тер} - M_{навант} \quad (2.48)$$

де J означає сумарний момент інерції платформи разом з ротором двигуна і турельною установкою, $M_{тер}$ характеризує момент тертя, а $M_{навант}$ представляє моменти від зовнішніх збурень. Момент тертя зазвичай моделюється як комбінація в'язкого тертя та кулонівського тертя

$$M_{тер} = b\omega + M_c sign(\omega) \quad (2.49)$$

де b визначає коефіцієнт в'язкого тертя, а M_c характеризує величину сухого тертя. Кутове положення платформи визначається інтегруванням кутової швидкості

$$\frac{d\theta}{dt} = \omega \quad (2.50)$$

Дискретизація цих диференціальних рівнянь за часом здійснюється методом чисельного інтегрування. Найпростішим підходом є метод Ейлера, який апроксимує похідну скінченною різницею. Для кутової швидкості на кроці $n+1$ отримуємо

$$\omega_{n+1} = \omega_n + \frac{\Delta t}{J}(M_e - b\omega_n - M_c sign(\omega_n)) \quad (2.51)$$

де Δt позначає крок дискретизації за часом. Для кутового положення відповідно маємо

$$\theta_{n+1} = \theta_n + \Delta t \cdot \omega_n \quad (2.52)$$

Більш точним є метод Рунге-Кутта другого порядку, який враховує зміну похідної всередині інтервалу інтегрування. Проміжне значення швидкості обчислюється як

$$\omega_{пром} = \omega_n + \frac{\Delta t}{2J}(M_e - b\omega_n - M_c sign(\omega_n)) \quad (2.53)$$

після чого кінцеве значення визначається через

$$\omega_{n+1} = \omega_n + \frac{\Delta t}{J}(M_e - b\omega_{пром} - M_c sign(\omega_{пром})) \quad (2.54)$$

Аналогічно для положення спочатку обчислюється проміжна точка $\theta_{пром} = \theta_n + \frac{\Delta t}{2}\omega_n$, а потім кінцеве значення $\theta_{n+1} = \theta_n + \Delta t \cdot \omega_{пром}$.

Моделювання датчика положення враховує його дискретність, затримку та шум вимірювання. Реальні датчики мають скінченну роздільну здатність, тому виміряне значення кута квантується

$$\theta_{вим} = round\left(\frac{\theta}{\Delta\theta_{\min}}\right)\Delta\theta_{\min} \quad (2.55)$$

де $\Delta\theta_{\min}$ характеризує роздільну здатність датчика. Затримка датчика моделюється через буфер попередніх значень, з якого береться значення на k кроків назад

$$\theta_{датч}(n) = \theta_{вим}(n - k) \quad (2.56)$$

де $k = round(\tau_{датч}/\Delta t)$ визначається часом затримки $\tau_{датч}$. Шум вимірювання додається як випадкова величина з нормальним розподілом

$$\theta_{вим} = \theta_{датч} + \xi \quad (2.57)$$

де $\xi \approx N(0, \sigma_{датч}^2)$ представляє білий шум з дисперсією $\sigma_{датч}^2$.

Керуючий вплив на двигун формується у вигляді напруги живлення, яка обмежена максимальним значенням напруги джерела живлення. Обмеження реалізується через функцію насичення

$$U_{реал} = \begin{cases} U_{max}, & U > U_{\max} \\ U, & -U_{\max} \le U \le U_{\max} \\ -U_{max}, & U < -U_{\max} \end{cases} \quad (2.58)$$

Також необхідно враховувати обмеження швидкості зміни напруги, яке визначається інерційністю силової електроніки

$$\frac{dU}{dt} \le v_{\max} \quad (2.59)$$

що в дискретній формі записується як

$$|U_{n+1} - U_n| \le v_{\max} \Delta t \qquad (2.60)$$

Структура програмної реалізації цифрового двійника передбачає створення класу або набору функцій, які інкапсулюють стан системи та надають інтерфейс для взаємодії. Стан системи включає поточні значення кута θ, кутової швидкості ω, струму двигуна I та історію вимірювань датчика для моделювання затримки. Параметри системи задаються при ініціалізації і включають момент інерції J, коефіцієнт тертя b, момент сухого тертя M_c , параметри двигуна R_a , L_a , k_m , k_e , параметри датчика $\Delta\theta_{\min}, \tau_{датч}, \sigma_{датч}$ та параметри дискретизації Δt.

Основний метод цифрового двійника виконує один крок симуляції і має сигнатуру виду крок симуляції з вхідними параметрами керуюча напруга та момент збурення і виходом поточний стан системи. Всередині методу послідовно виконуються обчислення струму двигуна за поточною напругою і швидкістю, електромагнітного моменту, нової кутової швидкості з урахуванням моментів тертя і збурення, нового кутового положення інтегруванням швидкості, виміряного значення від датчика з урахуванням квантування затримки і шуму. Метод повертає структуру з полями кутове положення, кутова швидкість, струм двигуна та виміряне положення від датчика.

Для тестування регуляторів цифровий двійник використовується в циклі симуляції, де на кожному кроці регулятор отримує виміряне положення датчика і задане положення, обчислює керуючий вплив, який подається на вхід двійника, після чого двійник повертає новий стан системи. Цикл повторюється до досягнення заданого часу симуляції або виконання критерію зупинки. Результати симуляції зберігаються у вигляді часових рядів для подальшого аналізу та візуалізації перехідних процесів.

Валідація цифрового двійника здійснюється порівнянням його відгуку з експериментальними даними реальної системи або з аналітичними

розв'язками для спрощених випадків. При відсутності зовнішніх збурень і простому керуванні постійною напругою перехідний процес має відповідати аперіодичній ланці першого порядку з постійною часу $T = \frac{J}{b + \frac{k_e k_m}{R_a}}$.

Відхилення від аналітичного розв'язку характеризує точність чисельного методу інтегрування та адекватність обраного кроку дискретизації.

Вибір кроку дискретизації Δt визначається компромісом між точністю моделювання та швидкістю обчислень. Для забезпечення стійкості методу Ейлера необхідно виконання умови $\Delta t < \frac{2J}{b + \frac{k_e k_m}{R_a}}$, тоді як для методу Рунге-

Кутта ця умова менш жорстка. На практиці крок дискретизації обирається в десять разів меншим за постійну часу системи, що забезпечує достатню точність без надмірних обчислювальних витрат.

3. РОЗРОБКА МОДЕЛІ КЕРУВАННЯ ПЛАТФОРМОЮ ІЗ ВИКОРИСТАННЯМ НЕЙРОМЕРЕЖІ

3.1 Обґрунтування вибору методу навчання в умовах відсутності апріорних даних

Розробка інтелектуальних систем керування для електромеханічних об'єктів фундаментально відрізняється від класичних задач машинного навчання, таких як розпізнавання образів або обробка природної мови. У традиційному підході, відомому як навчання з учителем (Supervised Learning), ключовим елементом успіху є наявність великого, розміченого та верифікованого набору даних — датасету. Цей набір даних виступає в ролі еталону, який містить тисячі пар прикладів «вхідні дані — правильна відповідь». Наприклад, для навчання нейромережі розпізнавати рукописний текст, необхідно надати їй величезну кількість зображень символів, де для кожного зображення чітко вказано, яка це літера. У процесі навчання мережа мінімізує різницю між своїм передбаченням та цим еталоном. Однак при спробі перенести цю логіку на задачу керування динамічною механічною платформою дослідник стикається з непереборною перешкодою, яка полягає у фізичній відсутності такого еталону.

Проблема полягає в тому, що для створення датасету для навчання нейроконтролера нам вже потрібно мати ідеальний контролер, який би згенерував ці дані. Якби ми могли записати тисячі прикладів ідеального керування, де для кожного положення платформи та кожної миті часу відомий математично бездоганний керуючий сигнал двигуна, це означало б, що задача керування вже вирішена і потреба у розробці нейромережі відпадає. Спроба використати як «учителя» класичний ПІД-регулятор також є помилковим шляхом. Якщо навчити нейромережу на даних, отриманих від ПІД-регулятора, вона в найкращому випадку лише скопіює його поведінку, успадкувавши всі його недоліки, такі як перерегулювання, повільна реакція

на збурення або статична помилка. Наша ж мета полягає у створенні системи, яка перевершує класичні алгоритми, а не імітує їх. Ми прагнемо знайти стратегію керування, яка, можливо, є неочевидною для людини-оператора або занадто складною для опису лінійними законами регулювання.

Виходячи з цього парадоксу — неможливості навчити систему на прикладах, оскільки приклади ідеальної поведінки ще не існують, — єдиним можливим шляхом стає використання методів навчання через взаємодію. У даній роботі застосовано підхід, що базується на еволюційних стратегіях та принципах навчання з підкріпленням. Ця парадигма змінює роль даних у процесі навчання. Замість пасивного аналізу статичного архіву «правильних відповідей», нейронна мережа перетворюється на активного агента, який самостійно генерує дані у процесі взаємодії з середовищем. Процес навчання трансформується з запам'ятовування еталонів у процес пошуку оптимальної поведінки шляхом спроб і помилок. Система генерує керуючі сигнали, спостерігає за реакцією платформи й отримує оцінку своїх дій через функцію пристосованості.

Такий підхід накладає специфічні вимоги до розуміння об'єкта керування самою нейромережею. У класичній теорії автоматичного керування розробка контролера починається з виведення диференціальних рівнянь, що описують фізику процесу: моменти інерції, коефіцієнти в'язкого тертя, індуктивність обмоток двигуна тощо. Натомість, у запропонованому підході нейронна мережа розглядає об'єкт керування як «чорну скриньку». Це означає, що мережі не повідомляються закони фізики, за якими функціонує платформа. Вона не має вбудованих знань про те, що сила струму викликає обертальний момент, або що швидкість є похідною від координати. На початку процесу навчання мережа є tabula rasa — чистим листом, а її ініціалізація випадковими вагами призводить до хаотичних рухів платформи.

Емпіричне навчання відбувається наступним чином: мережа подає сигнал на віртуальний драйвер двигуна і через вхідні сенсори фіксує зміну

стану системи. Якщо поданий сигнал призвів до зменшення помилки розузгодження між поточною та цільовою позицією, внутрішня структура ваг мережі закріплює цей патерн поведінки. Якщо ж дії призвели до відхилення від цілі або нестабільності, еволюційний алгоритм відбраковує таку гілку налаштувань. З часом, через мільйони ітерацій симуляції, у прихованих шарах нейромережі, зокрема у рекурентних блоках, формується неявна модель фізики об'єкта. Мережа «вивчає» інерцію не як формулу $J \cdot \varepsilon = M$, а як відчуття того, що після подачі потужного імпульсу платформа продовжуватиме рух деякий час за інерцією, і тому необхідно завчасно почати гальмування. Вона «вивчає» тертя як опір, який необхідно подолати мінімальним пороговим зусиллям.

Ця концепція «чорної скриньки» є критично важливою перевагою. Вона робить систему інваріантною до складності математичної моделі. Для нейромережі не має значення, чи описується тертя простою лінійною залежністю, чи складною нелінійною функцією Стрібека — вона адаптується до реальної реакції об'єкта, якою б складною вона не була. Однак, такий метод навчання вимагає колосальної кількості експериментів, які неможливо провести на фізичному обладнанні через ризик його пошкодження та часові обмеження. Саме тому невід'ємною частиною розробленої методики є використання високоточного цифрового двійника. Цифровий двійник виступає безпечним полігоном, де популяція нейромереж може здійснювати мільйони спроб керування, включаючи аварійні режими, без жодних матеріальних втрат. Тільки завдяки поєднанню еволюційного пошуку оптимальної стратегії та високошвидкісної симуляції на цифровому двійнику стає можливим синтезувати контролер, який перевершує традиційні методи, не маючи при цьому жодного прикладу «правильного» керування на старті.

3.2 Формування вхідного вектора станів та попередня обробка даних нейромережевого контролера

Проектування вхідного шару (Input Layer) є початковий етапом розробки архітектури нейронної мережі, оскільки саме цей інтерфейс визначає, як фізична реальність механічної платформи трансформується у абстрактні математичні тензори, зрозумілі алгоритму керування. Якість, повнота та формат подачі вхідних даних безпосередньо впливають на здатність мережі знаходити оптимальні стратегії керування та швидкість збіжності алгоритму навчання. У розробленій системі вхідний вектор станів сформовано з трьох ключових компонентів: бажаної позиції (Target Position), поточної позиції платформи (Current Position) та поточної помилки неузгодженості (Error). На перший погляд, такий набір може здатися надлишковим, оскільки помилка є лінійною комбінацією бажаної та поточної позицій, і теоретично будь-яка багатошарова нейромережа здатна самостійно вивчити цю залежність у першому ж прихованому шарі. Проте рішення подавати помилку як явний, окремий вхідний параметр є свідомим архітектурним вибором, що базується на принципах інженерії ознак (Feature Engineering).

Введення помилки як окремого входу суттєво розвантажує обчислювальні ресурси мережі, звільняючи нейрони прихованих шарів від необхідності виконувати арифметичну операцію віднімання та дозволяючи їм зосередитися на більш складних задачах апроксимації динаміки руху. Помилка виступає найбільш «гострим» сигналом для системи керування, оскільки саме її мінімізація є цільовою функцією всього процесу. Надаючи цей параметр у явному вигляді, ми створюємо для мережі «прямий шлях» до розуміння того, наскільки далека система від стану рівноваги. Це значно прискорює початкові етапи навчання, дозволяючи еволюційному алгоритму швидше відбраковувати неефективні стратегії. Водночас наявність абсолютних значень поточної та бажаної позицій дозволяє контролеру

враховувати контекст робочого простору. Це критично важливо для реальних механічних систем, де фізичні властивості можуть бути неоднорідними залежно від кута повороту — наприклад, через нерівномірність тертя у підшипниках, натяг кабелів або зовнішні гравітаційні впливи, якщо вісь обертання не є ідеально вертикальною. Таким чином, тріада «Ціль – Положення – Помилка» забезпечує баланс між локальною задачею стабілізації (через помилку) та глобальною орієнтацією у просторі станів (через абсолютні координати).

Наступним етапом після вибору складу вектора є процедура нормалізації даних. Фізичні величини, з якими оперує механічна платформа, вимірюються у градусах і знаходяться у діапазоні від 0 до 360. З точки зору обчислювальної математики нейронних мереж, такі значення є неприпустимо великими. Це пов'язано з природою ініціалізації вагових коефіцієнтів та роботою функцій активації. Ваги нейронної мережі на старті ініціалізуються дуже малими випадковими числами, зазвичай розподіленими навколо нуля з дисперсією менше одиниці. Якщо подати на вхід такої мережі «сирі» дані, наприклад, значення позиції 350 градусів, скалярний добуток вхідного сигналу на ваги призведе до утворення велетенських значень активації нейронів.

Це явище створює дві проблеми, що унеможливлюють ефективне навчання. По-перше, виникає проблема насичення функцій активації. У даній архітектурі, як і в багатьох рекурентних мережах, використовуються сигмоїдальні функції або гіперболічний тангенс, які мають властивість насичення. Це означає, що при великих абсолютних значеннях аргументу вихід функції стає надзвичайно близьким до 1 або -1, а її похідна прямує до нуля. Оскільки градієнтні методи оптимізації та еволюційні стратегії покладаються на чутливість виходу до малих змін ваг, потрапляння в зону насичення фактично зупиняє навчання — мережа стає «сліпою» до змін

параметрів, оскільки сигнал просто губиться в зоні плато функції активації. Ця проблема відома як проблема зникаючого градієнта.

По-друге, використання ненормалізованих даних призводить до нестабільності чисельних методів оптимізації. Ландшафт функції втрат стає витягнутим у напрямку вхідних координат з великими значеннями, що змушує оптимізатор робити неефективні кроки — або занадто малі, що призводить до нескінченно довгого навчання, або занадто великі, що викликає осциляції та розбіжність процесу. Для рекурентних мереж (GRU), де сигнал циркулює у циклах зворотного зв'язку, ці ефекти підсилюються багаторазово, оскільки великі значення акумулюються на кожному часовому кроці, швидко призводячи до переповнення змінних або втрати точності обчислень.

Для вирішення цих проблем у розробленій системі реалізовано алгоритм лінійної нормалізації вхідного простору, який трансформує фізичні величини у діапазон від мінус одиниці до одиниці [-1; 1]. Процедура нормалізації реалізована безпосередньо перед подачею даних у перший рекурентний шар. Математично це перетворення здійснюється шляхом масштабування. Для позиційних даних (поточна та бажана позиція) застосовується формула, що зміщує центр діапазону [0, 360] у нуль та стискає амплітуду. Значення 0 градусів відображається у -1, значення 360 градусів — у +1, а 180 градусів стає 0. Аналогічна операція застосовується до сигналу помилки, який також приводиться до цього стандартного діапазону.

Вибір саме діапазону [-1; 1], а не [0; 1], також є обґрунтованим. Симетричний відносно нуля розподіл вхідних даних (zero-centered data) є бажаним для нейронних мереж, оскільки це дозволяє вагам змінювати знак у процесі навчання більш природно та симетрично. Коли вхідні дані центровані навколо нуля, оновлення ваг у позитивному та негативному напрямках відбуваються з однаковою ймовірністю, що запобігає систематичному зміщенню градієнтів (zig-zagging effect) під час оптимізації. Окрім того,

оскільки на виході мережі використовується функція гіперболічного тангенса, яка працює у діапазоні [-1; 1], приведення вхідних даних до аналогічного масштабу створює наскрізну узгодженість амплітуд сигналів через усю глибину мережі.

Реалізований механізм нормалізації також виконує роль запобіжника. У разі виникнення аномальних показників сенсорів або помилок обчислення, які виходять за межі фізично можливого діапазону, процедура жорсткого обмеження (clipping) утримує вхідні значення у заданих межах [-1; 1]. Це гарантує, що навіть при збоях у зовнішньому контурі, внутрішній стан нейромережі не буде зруйновано одним екстремальним значенням, що забезпечує високу робастність системи керування в цілому. Таким чином, вхідний шар не просто транслює дані, а формує безпечний, математично оптимальний простір ознак, у якому еволюційний пошук розв'язку відбувається максимально ефективно.

3.3 Обробка циклічності кутових координат

Окремим нюансом при формуванні вхідного вектора є специфіка роботи з обертовими механізмами, де координати є циклічними. Фізично положення платформи 0° та 360° — це одна й та ж точка простору. Однак для нейронної мережі, яка сприймає вхідні дані як звичайні десяткові числа, ці значення знаходяться на протилежних краях числового діапазону. Якщо обчислювати третій параметр вхідного вектора (помилку) як звичайну арифметичну різницю між бажаним та поточним положенням, виникає ситуація розриву даних на межі повного оберту.

На практиці це створює проблему вибору напрямку руху. Розглянемо випадок, коли платформа знаходиться в точці 359°, а ціль встановлена на 1°. Фізична відстань між ними складає всього 2°, і найефективнішим рішенням є невеликий доворот через нульову відмітку. Проте, якщо використати просте віднімання (1 - 359), отримаємо значення -358°. Спираючись на таке велике

від’ємне число, контролер намагатиметься обертати платформу у зворотний бік, роблячи майже повне коло для досягнення цілі. Така поведінка є неефективною з точки зору витрат часу та енергії.

Для вирішення цього завдання на етапі підготовки даних реалізовано алгоритм обчислення найкоротшої кутової відстані (shortest angular difference). Цей метод автоматично визначає оптимальний напрямок руху. Алгоритм порівнює відстань до цілі за годинниковою стрілкою та проти неї, обираючи той шлях, модуль якого не перевищує 180°. У наведеному прикладі алгоритм замість -358° передасть на вхід мережі значення $+2^\circ$.

Завдяки такій попередній обробці, простір станів для нейромережі стає безперервним. Вхідний шар отримує коректну інформацію про близькість точок, що знаходяться по різні боки від нульової відмітки. Це дозволяє уникнути неоднозначності в керуванні та гарантує, що мережа завжди обиратиме найкоротший шлях до цілі, незалежно від того, в якому секторі кола знаходиться платформа. Нормалізація отриманої кутової різниці відбувається вже після цього перерахунку, що забезпечує коректну роботу вагових коефіцієнтів у всьому робочому діапазоні.

3.4 Архітектура прихованих шарів: Вибір на користь рекурентності (RNN/GRU)

При проектуванні нейромережі для керування фізичним об'єктом, який має масу та інерцію, ми стикаємося з проблемою, яку неможливо вирішити простим нарощуванням кількості нейронів. Це проблема сприйняття динаміки. Більшість стандартних нейромереж, які називають повнозв'язними або MLP (Multi-Layer Perceptron), працюють за принципом "тут і зараз". Вони дивляться на вхідні дані як на статичну картинку. Якщо подати такій мережі інформацію про те, що платформа знаходиться в позиції 100 градусів, а ціль — теж 100 градусів, мережа побачить нульову помилку і видасть команду на зупинку двигуна. Проте в реальному фізичному світі однієї точки координат

недостатньо для прийняття рішення. Звичайна мережа не бачить різниці між ситуацією, коли платформа спокійно стоїть у точці призначення, і ситуацією, коли вона пролітає цю точку на величезній швидкості. Для звичайної мережі ці два стани ідентичні, оскільки координати в момент вимірювання збігаються. Але фізично це абсолютно різні стани: у першому випадку треба нічого не робити, а в другому — подавати максимальну напругу на гальмування, щоб погасити інерцію. Без розуміння контексту руху звичайна мережа неминуче припускатиметься помилок, викликаючи перерегулювання або коливання навколо цілі.

Щоб вирішити цю проблему, у розробленій системі використано архітектуру рекурентних нейронних мереж (RNN), а саме модифікацію GRU (Gated Recurrent Unit). Головна відмінність такої мережі полягає в наявності внутрішньої пам'яті. Якщо звичайна мережа — це функція, яка перетворює вхід на вихід, то рекурентна мережа — це функція, яка перетворює вхід і свій попередній стан на вихід і новий стан. Цей попередній стан називається прихованим станом (hidden state). У коді проекту це реалізовано через передачу параметру `return_state=True` при ініціалізації шарів GRU. Це дозволяє мережі передавати інформацію сама собі від одного такту керування до наступного.

Механізм роботи GRU у цій задачі можна порівняти з роботою людського вестибулярного апарату. Коли ми заплющуємо очі в машині, ми все одно відчуваємо прискорення та гальмування, хоча не бачимо зміни координат. У нейромережі цю роль виконує вектор прихованого стану. Коли на вхід мережі надходить послідовність позицій — наприклад, 10, 12, 15, 19 градусів — шар GRU оновлює свій внутрішній стан. Зміни у вхідних даних (різниця між 10 і 12, потім між 12 і 15) "відкладаються" у пам'яті мережі. Фактично, мережа виконує чисельне диференціювання: аналізуючи різницю між поточним і попереднім значенням позиції, вона отримує швидкість. Аналізуючи, як змінилася швидкість, вона отримує прискорення.

Важливо підкреслити, що ми явно не програмуємо формули диференціювання і не подаємо швидкість на вхід мережі. Вхідний вектор містить лише позиції. Вектор пам'яті GRU стає віртуальним датчиком швидкості та прискорення. У процесі навчання за допомогою генетичного алгоритму мережа самостійно налаштовує свої вагові коефіцієнти так, щоб певні комірки її пам'яті відповідали за збереження інформації про динаміку. Це дозволяє системі "розуміти" інерцію платформи. Якщо платформа масивна і розігнана, значення у прихованому стані будуть великими, і навіть коли помилка позиції стане нульовою, ці внутрішні значення змусять вихідний шар сформувати гальмівний імпульс ще до того, як платформа перетне цільову позначку. Така здатність передбачати майбутнє положення на основі минулої динаміки робить керування плавним і точним, чого неможливо досягти на простій MLP-мережі без додаткових фізичних сенсорів.

Вибір саме архітектури GRU, а не більш відомої LSTM (Long Short-Term Memory), зумовлений прагматичними інженерними міркуваннями, пов'язаними з обчислювальною ефективністю. LSTM має три вентилі керування пам'яттю (вхідний, вихідний та вентиль забування), що вимагає чотирьох матричних множень на кожен крок обробки. GRU є більш сучасною та спрощеною версією рекурентного блоку, яка об'єднує вентилі вхідних даних та забування в один механізм оновлення. Це зменшує кількість необхідних параметрів приблизно на 25% порівняно з LSTM при збереженні аналогічної ефективності для задач моделювання фізичних процесів.

Зменшення кількості параметрів має критичне значення для практичної реалізації системи. Оскільки кінцевою метою розробки є керування реальним обладнанням, алгоритм повинен працювати в режимі реального часу з фіксованим кроком дискретизації. У файлі конфігурації навчання вказано крок часу `dt=0.1`, але в реальній системі частота оновлення може сягати сотень герц. Використання більш "легкої" архітектури GRU означає, що

мікроконтролер витрачатиме менше тактів процесора на виконання прямого проходу мережі (inference). Це знижує вимоги до обчислювальної потужності заліза, зменшує енергоспоживання та залишає більше ресурсів для обробки сигналів з датчиків або комунікації із зовнішнім світом. Крім того, менша кількість параметрів означає менший розмір моделі, що прискорює процес навчання, оскільки еволюційному алгоритму потрібно оптимізувати менш вимірний простір пошуку рішень.

Таким чином, використання рекурентного шару GRU є не просто спробою застосувати модну технологію, а необхідною умовою для якісного керування динамічним об'єктом. Це дозволяє компенсувати відсутність фізичних датчиків швидкості програмними засобами та забезпечити стабільність системи, яка враховує не лише де ми є, а й як швидко ми туди рухаємось.

3.5 Синтез вихідного керуючого сигналу та інтерфейс взаємодії з виконавчим механізмом

Останнім етапом обробки інформації в нейромережевому контролері є формування вихідного сигналу, який безпосередньо транслюється на виконавчий механізм платформи. Цей процес відбувається у вихідному шарі (Output Layer) нейронної мережі. Проектування цього шару вимагає суворого узгодження математичних властивостей штучного нейрона з фізичними характеристиками електродвигуна постійного струму. Оскільки завданням системи є керування позицією, привід повинен мати можливість обертатися в обох напрямках для компенсації помилки незалежно від її знаку. Це накладає специфічні вимоги на область значень вихідної функції: вона повинна бути симетричною відносно нуля та обмеженою фіксованим діапазоном, що відповідає максимально допустимій напрузі живлення двигуна.

Для реалізації такої логіки керування оптимальним вибором для функції активації вихідного нейрона є гіперболічний тангенс (`tanh`). Ця функція відображає множину вхідних значень на інтервал від мінус одиниці

до одиниці [-1; 1]. Вибір саме цієї нелінійності зумовлений її ідеальною кореляцією з фізикою роботи Н-моста — електронної схеми, що керує полярністю підключення двигуна. У цій парадигмі нульове значення виходу функції `tanh` інтерпретується як відсутність напруги на клемах двигуна, що відповідає режиму зупинки або вільного вибігу. Значення зі знаком «плюс» вказують на необхідність обертання за годинниковою стрілкою, а значення зі знаком «мінус» — проти неї. Абсолютна величина сигналу, що наближається до одиниці, відповідає подачі максимальної напруги джерела живлення.

Використання інших поширених функцій активації у даному контексті є недоцільним. Наприклад, логістична сигмоїда (`sigmoid`) має діапазон вихідних значень [0; 1]. Це змусило б інженерів вводити штучне зміщення (наприклад, віднімати 0.5 від результату), щоб отримати можливість реверсивного руху, що ускладнює навчання мережі, оскільки «нульова» точка (зупинка двигуна) плавала б залежно від налаштування зміщення. Функція ReLU (Rectified Linear Unit), яка є стандартом для прихованих шарів, також непридатна для виходу, оскільки вона є необмеженою зверху. Це могло б призвести до генерації сигналів, що перевищують фізичні можливості драйвера або розрядності цифро-аналогового перетворювача, створюючи ризик програмних переповнень. Натомість гіперболічний тангенс має властивість природного насичення: як би сильно нейронна мережа не «хотіла» прискорити платформу, вихідний сигнал ніколи не перевищить одиницю за модулем. Це діє як вбудований програмний лімітер, що захищає систему від генерації некоректних керуючих впливів.

Важливим є інтерпретація безрозмірного виходу нейромережі контролером нижнього рівня, який безпосередньо керує силовими ключами драйвера. Вихідне значення нейрону, яке ми позначимо як $u \in [-1; 1]$, необхідно перетворити на сигнал широтно-імпульсної модуляції (ШІМ). Цей процес розбивається на дві складові: визначення напрямку (Direction) та розрахунок коефіцієнта заповнення (Duty Cycle). Мікроконтролер аналізує

знак числа u. Якщо $u > 0$, логічний сигнал напрямку встановлюється у стан HIGH, інакше — у стан LOW. Амплітуда керуючого впливу визначається як модуль вихідного сигналу $|u|$. Оскільки більшість таймерів мікроконтролерів оперують цілочисельними значеннями регістрів порівняння (наприклад, від 0 до 255 для 8-бітного ШІМ або від 0 до 1023 для 10-бітного), значення $|u|$ множиться на максимальне значення регістру. Наприклад, вихід мережі -0.75 означає, що двигун має обертатися у зворотному напрямку з ефективною напругою, що становить 75% від номіналу живлення.

Така пряма відповідність спрощує налагодження системи. Інженер може легко інтерпретувати поведінку мережі: якщо на графіках видно, що вихідний сигнал постійно перебуває в зонах насичення (близько до 1 або -1), це свідчить про те, що привід недостатньо потужний для заданої динаміки, або коефіцієнти регулятора занадто агресивні. Якщо ж сигнал коливається близько нуля з малою амплітудою, це може вказувати на надмірну обережність стратегії керування. Крім того, диференційованість функції `tanh` у всьому діапазоні значень є критично важливою для алгоритмів навчання. Навіть коли вихід мережі близький до меж діапазону, функція зберігає ненульовий градієнт (хоча й малий), що дозволяє еволюційним алгоритмам або методам градієнтного спуску тонко підлаштовувати ваги для досягнення оптимальної траєкторії руху платформи. Таким чином, вихідний шар не просто передає сигнал, а формує фізично обґрунтований, безпечний та інтерпретований інтерфейс між штучним інтелектом та електромеханічною частиною платформи.

3.6 Алгоритмічне забезпечення процесу навчання: Метод еволюційних стратегій та дзеркального семплювання

Вибір алгоритму оптимізації вагових коефіцієнтів нейронної мережі є критичним етапом проектування системи керування, який безпосередньо залежить від властивостей об'єкта та середовища взаємодії. У класичних

задачах глибокого навчання, таких як комп'ютерний зір або обробка природної мови, безальтернативним стандартом є метод зворотного поширення помилки (Backpropagation). Цей метод базується на обчисленні градієнта функції втрат по відношенню до кожного вагу мережі, використовуючи ланцюгове правило диференціювання. Однак, застосування градієнтних методів для навчання робототехнічних систем стикається з фундаментальною перешкодою — вимогою диференційовності середовища. Для того щоб "проштовхнути" градієнт від помилки на виході (відхилення положення платформи) до ваг нейромережі, необхідно, щоб кожен елемент ланцюжка — включаючи фізичну модель платформи — був гладкою, диференційовною функцією. Реальним механічним системам це не властиво. Присутність сухого тертя, механічних люфтів у редукторі, дискретність датчиків та шум вимірювання створюють розриви та нелінійності у просторі станів. У таких точках похідна функції або не існує, або дорівнює нулю чи нескінченності, що призводить до руйнування процесу градієнтного спуску.

Для подолання цієї проблеми у даній роботі застосовано метод Еволюційних Стратегій (Evolution Strategies, ES). Це клас алгоритмів оптимізації "чорної скриньки", який не вимагає обчислення градієнтів функції винагороди. Замість аналітичного пошуку напрямку спуску, алгоритм оцінює градієнт емпірично, спираючись на статистику випробувань. Це робить процес навчання інваріантним до природи фізичного об'єкта: алгоритму байдуже, чи є механіка гладкою, чи вона має жорсткі удари та розриви. Нейромережа розглядає симулятор (а згодом і реальний стенд) лише як генератор оцінки якості (Fitness score), не намагаючись аналізувати його внутрішню математичну структуру.

Центральним елементом методу є популяційний підхід. Процес навчання відбувається не над одним екземпляром нейромережі, а над цілою популяцією її варіацій. На кожній ітерації навчання (поколінні) алгоритм бере поточні "найкращі" ваги мережі, які ми називаємо центроїдом популяції,

і створює навколо них хмару пошукових точок. Це реалізується шляхом додавання до базових ваг випадкового шуму, розподіленого за нормальним (гаусівським) законом. У програмній реалізації цей процес контролюється гіперпараметром `sigma` (стандартне відхилення шуму). Параметр `sigma` визначає радіус пошуку: малі значення призводять до локальної оптимізації та тонкого налаштування, тоді як великі значення стимулюють глобальне дослідження простору рішень, дозволяючи виходити з локальних мінімумів. Наприклад, якщо розмір популяції встановлено на рівні 50 агентів, система генерує 50 унікальних клонів нейромережі, кожен з яких має дещо відмінну конфігурацію синаптичних зв'язків.

Кожен з цих клонів проходить випробування у цифровому двійнику, виконуючи завдання позиціювання платформи. Оскільки кожен клон поводиться трохи інакше — один може бути більш агресивним, інший більш плавним — вони отримують різні оцінки ефективності. На основі цих оцінок алгоритм обчислює, в який бік потрібно змістити центроїд ваг, щоб покращити середній результат популяції. Фактично, ми замінюємо точне аналітичне диференціювання на стохастичну апроксимацію градієнта. Цей підхід є надзвичайно стійким: навіть якщо один з агентів через шуми або невдалу мутацію покаже катастрофічно поганий результат, це не зруйнує навчання, оскільки його вплив буде нівельовано статистикою інших агентів.

Науковою новизною та важливою особливістю реалізованого алгоритму є використання методу дзеркального семплювання (Antithetic Sampling). У класичному наївному підході до еволюційних стратегій ми генеруємо N випадкових мутацій незалежно одна від одної. Проте, такий підхід страждає від високої дисперсії (variance) оцінки градієнта. Може статися так, що випадковий шум випадково покращить результат не тому, що ми знайшли правильний напрямок зміни ваг, а через збіг обставин (наприклад, вдалі початкові умови симуляції). Це призводить до "блукання" алгоритму і сповільнює збіжність.

Метод дзеркального семплювання вирішує цю проблему шляхом введення суворої структури у генерацію шуму. Замість того, щоб генерувати 50 незалежних мутацій, ми генеруємо 25 векторів шуму ϵ, і для кожного з них створюємо пару агентів: одного з доданим шумом ($+\epsilon$) і іншого з віднятим шумом ($-\epsilon$). Тобто, ми досліджуємо простір рішень у діаметрально протилежних напрямках відносно поточного центру. Це має глибоке математичне обґрунтування. Якщо додавання певної комбінації змін до ваг призводить до покращення результату, то логічно припустити, що віднімання цієї ж комбінації (рух у протилежний бік) має призвести до погіршення.

Використання антитетичних пар дозволяє значно точніше оцінити градієнт. Оцінка напрямку руху обчислюється як зважена різниця ефективності між позитивним та негативним клоном. Якщо обидва клони показують однаковий результат, це означає, що даний напрямок у просторі ваг не впливає на якість керування, і алгоритм його ігнорує. Якщо ж між ними є суттєва різниця, це дає сильний і, що найважливіше, статистично достовірний сигнал для оновлення ваг. Такий підхід суттєво знижує дисперсію стохастичного градієнта, що на практиці означає більш стабільне, плавне та швидке навчання. Ми отримуємо переваги Монте-Карло методів, але з набагато меншою кількістю необхідних зразків для досягнення тієї ж точності.

Фінальний етап кожної ітерації — це оновлення ваг основної мережі. Зібрана статистика про те, які збурення ($+\varepsilon$ чи $-\varepsilon$) дали приріст ефективності, агрегується у вектор оновлення. Цей вектор масштабується на коефіцієнт швидкості навчання (learning rate) і додається до поточних ваг. Для підвищення стабільності збіжності у даній роботі застосовано не просто стохастичний градієнтний спуск (SGD), а адаптивний оптимізатор, аналогічний за логікою до алгоритму Adam. Він враховує моменти першого та другого порядку (інерцію градієнта та історію його амплітуди), що

дозволяє алгоритму впевнено рухатися по пологих ділянках ландшафту функції втрат і обережно гальмувати поблизу оптимуму. Така комбінація популяційного підходу, дзеркального семплювання та адаптивної оптимізації дозволяє синтезувати високоякісний контролер в умовах повної невизначеності щодо точних параметрів фізичного об'єкта.

3.7 Програмна реалізація "Цифрового двійника" (Simulation Environment)

Розробка програмного середовища симуляції є ключовим етапом створення системи керування, оскільки саме це середовище забезпечує генерацію даних, необхідних для навчання нейронної мережі. В умовах відсутності фізичного доступу до об'єкта або неможливості проведення тривалих експериментів на реальному обладнанні, програмний модуль, який називають цифровим двійником, бере на себе функції фізичного світу. Його реалізація у даній роботі виконана з дотриманням принципів ізоляції інтерфейсів, математичної достовірності динамічних процесів та обчислювальної ефективності через векторизацію.

3.7.1. Архітектурна ізоляція середовища

З точки зору програмної архітектури, цифровий двійник спроектовано як окремий модуль, повністю відокремлений від логіки контролера. В основу покладено шаблон проектування, що використовує абстрактний інтерфейс. У коді визначено базовий клас, який встановлює жорсткий контракт взаємодії: метод ініціалізації (reset) та метод кроку симуляції (step). Цей контракт гарантує, що нейромережа не має прямого доступу до внутрішніх змінних симулятора, а взаємодіє з ним виключно через стандартизовані канали введення-виведення. На вхід методу step подається керуючий сигнал (нормалізована напруга), а на виході повертається вектор стану системи.

Така ізоляція дозволяє розглядати симулятор як "полігон" для тренувань. Нейромережа, яка виступає в ролі агента, подає дію і отримує реакцію, не знаючи природи середовища. Це дозволяє прозоро змінювати реалізацію симулятора — від простої кінематичної моделі на етапі налагодження коду до складної динамічної моделі або навіть драйвера фізичного пристрою на етапі фінальних тестів — без жодних змін у коді самої нейромережі. Крім того, інкапсуляція стану всередині об'єкта симулятора забезпечує детермінованість: задавши однакове початкове зерно генератора випадкових чисел (seed), ми гарантовано отримуємо ідентичну послідовність подій, що є необхідною умовою для наукової верифікації результатів навчання та пошуку помилок.

3.7.2. Математична модель та дискретизація процесів

Всередині програмного методу step реалізовано математичну модель електромеханічної платформи. З точки зору теорії автоматичного керування, об'єкт ідентифіковано як аперіодичну ланку, що описує інерційні процеси розгону та гальмування. У безперервному часі поведінка кутової швидкості $\omega(t)$ під дією керуючого сигналу $u(t)$ описується диференціальним рівнянням першого порядку:

$$T \cdot \frac{d\omega(t)}{dt} + \omega(t) = K \cdot u(t) \quad (3.1)$$

де T — електромаханічна стала часу (характеризує інерцію системи), а K — коефіцієнт передачі (характеризує потужність двигуна). Положення платформи $\phi(t)$ є інтегралом від швидкості:

$$\frac{d\phi(t)}{dt} = \omega(t) \quad (3.2)$$

Для програмної реалізації ці безперервні рівняння перетворено на різницеві схеми, що виконуються дискретно з фіксованим кроком часу Δt. У

даній роботі обрано крок Δt = 0.1 с, що є достатнім для фіксації динаміки механічної системи. Дискретизація аперіодичної ланки швидкості реалізована через алгоритм експоненційного згладжування (фільтр низьких частот першого порядку).

На кожному кроці програмного циклу спочатку обчислюється цільова швидкість ω_{target}, яку розвинула б система у усталеному режимі при поточній напрузі:

$$\omega_{target}[k] = K \cdot u[k] \quad (3.3)$$

Далі розраховується коефіцієнт згладжування α, який залежить від співвідношення кроку дискретизації та інерції об'єкта. У коді цей параметр визначається як $\alpha = \Delta t/(\Delta t + T)$. Оновлення поточної швидкості відбувається за рекурентною формулою:

$$\omega[k] = (1 - \alpha) \cdot \omega[k-1] + \alpha \cdot \omega_{target}[k] \quad (3.4)$$

Це рівняння програмно імітує інерцію: швидкість не змінюється миттєво слідом за напругою, а плавно прямує до цільового значення. Після оновлення швидкості виконується чисельне інтегрування координати методом Ейлера:

$$\phi[k] = \phi[k-1] + \omega[k] \cdot \Delta t \quad (3.5)$$

Таким чином, програмний код симулятора послідовно вирішує систему різницевих рівнянь, перетворюючи вхідний масив дій на траєкторію руху, яка фізично відповідає поведінці реального інерційного тіла. Параметри моделі (K, T) винесено у конфігураційні змінні класу, що дозволяє налаштовувати "важкість" та "різкість" віртуальної платформи без зміни алгоритму.

3.7.3. Технологія прискорення (Векторизація)

Використання класичного об'єктно-орієнтованого підходу, де кожна платформа моделюється окремим об'єктом, виявилося недостатньо

ефективним для задач еволюційного навчання. Метод еволюційних стратегій вимагає одночасного оцінювання великої популяції агентів (наприклад, 50 або 100). При послідовному виконанні розрахунків для кожного агента в циклі, накладні витрати інтерпретатора мови програмування на виклик методів та динамічну типізацію починають перевищувати час корисних арифметичних операцій. Це призводить до неприпустимого збільшення часу навчання.

Для вирішення цієї проблеми розроблено спеціалізований клас векторизованого симулятора. В його основі лежить перехід від скалярних типів даних (float) до векторних масивів (NumPy arrays). У цій реалізації стан системи зберігається не як набір окремих змінних для однієї платформи, а як матриця станів розмірністю $[N]$, де N — розмір популяції.

Технічно це означає, що рівняння фізики, описані вище, застосовуються не до окремих чисел, а до цілих масивів одночасно. Процесор виконує операції за принципом SIMD (Single Instruction, Multiple Data — одна інструкція, багато даних). Наприклад, операція оновлення швидкості $\omega[k]$ відбувається для всіх 50 агентів за одну векторну інструкцію процесора, замість 50 послідовних операцій у циклі.

Реалізація методу step у векторизованому варіанті також вимагала адаптації алгоритмів обробки кутових координат. Оскільки використання умовних операторів if-else (розгалужень) є неможливим при роботі з векторами (бо різні елементи вектора можуть потребувати різних гілок виконання), логіку переходу через 0 градусів та розрахунку найкоротшої відстані до цілі було переписано мовою лінійної алгебри. Замість умовних переходів використано операції взяття залишку від ділення та бітові маски, що дозволило зберегти паралелізм обчислень.

Впровадження векторизованого симулятора дозволило синхронізувати роботу фізичного рушія (на CPU) з роботою нейромережі (на GPU), яка також оперує пакетами даних (батчами). Це забезпечило скорочення часу симуляції

одного покоління з декількох хвилин до часток секунди. Така висока швидкість роботи цифрового двійника дозволяє алгоритму навчання перебирати мільйони комбінацій параметрів за прийнятний час, що робить можливим синтез складної поведінки контролера без використання суперкомп'ютерних кластерів. Векторизація трансформувала симулятор з простого засобу перевірки у високопродуктивний інструмент генерації даних.

3.7.4. Система оцінювання: Архітектура мультиагентного формування винагороди (Reward Shaping)

Специфіка навчання генетичним алгоритмом полягає у тому, що нейромережа не має зовнішнього "вчителя", який міг би вказати на помилку в кожній конкретній дії. Єдиним джерелом інформації про якість керування є підсумкова функція пристосованості (Fitness Function). У найпростішому випадку ця функція могла б розраховуватися лише як відстань до цілі в кінці епізоду. Проте для керування інерційною платформою такий підхід є неефективним: простір пошуку рішень занадто великий, і мережа витрачає багато часу на генерацію абсолютно хаотичних рухів, які не отримують жодної позитивної оцінки.

Для вирішення цієї проблеми застосовано методику формування винагороди (Reward Shaping). Суть підходу полягає у декомпозиції глобальної задачі керування на набір локальних критеріїв. Система оцінювання реалізована як комітет незалежних математичних агентів. Кожен агент відповідає за окремий фізичний аспект руху платформи та генерує свою частину винагороди або штрафу на кожному кроці симуляції. Підсумкова оцінка є зваженою сумою виходів усіх агентів. Така архітектура дозволяє врахувати не лише точність позиціювання, а й експлуатаційні обмеження електродвигуна та механічної передачі.

3.7.5. Теоретичне обґрунтування структури функції винагороди

Формування архітектури системи оцінювання (набору агентів) у даній роботі базується не на евристичному підборі, а на фундаментальних принципах теорії оптимального керування та фізичних обмеженнях електромеханічних систем. Задача синтезу контролера розглядається як задача багатокритеріальної оптимізації, де необхідно знайти баланс між суперечливими вимогами: мінімізацією помилки регулювання та мінімізацією енергетичних витрат на керування.

У класичній теорії автоматичного керування, зокрема при проектуванні лінійно-квадратичних регуляторів (LQR), критерій якості керування J задається у вигляді інтегрального функціоналу, що містить дві складові: квадратичну помилку стану та квадратичну вартість керуючого впливу. Вибір агентів для нейромережевого контролера є прямою проекцією цього класичного підходу на площину навчання з підкріпленням.

Фізичні обмеження (Constraints): Будь-яка реальна система має обмеження. Двигун згорить при струмі > I_{max} (агент `Saturation`), редуктор зламається при ривку $d\tau/dt$ (агент `Smoothness`). Введення цих агентів обумовлено фізикою об'єкта.

У теорії RL доведено (Andrew Ng et al.), що вчити робота лише за кінцевим результатом ("досяг" / "не досяг") занадто довго. Потрібен "Shaping" — підказки на кожному кроці. Fгенти виступають саме цими математичними підказками.

Агенти групи точності (MoveToGoal, StayInThreshold, TimeToGoal) відповідають першій складовій класичного функціоналу — мінімізації відхилення вектора станів від цільового значення. Їх наявність є обов'язковою умовою вирішення основної технологічної задачі — позиціювання. Без цих агентів система не матиме цілепокладання. Специфіка використання саме трьох агентів замість одного пов'язана з нелінійністю процесу навчання:

MoveToGoal забезпечує градієнт на дальніх дистанціях, а StayInThreshold формує "потенціальну яму" в околі нуля, забезпечуючи асимптотичну стійкість.

Агенти групи експлуатації (SaturationPenalty, SmoothActionPenalty) є відображенням другої складової функціоналу якості — обмеження на керування. У реальних фізичних системах керуючий ресурс завжди є обмеженим. Ігнорування цього факту при навчанні "у віртуальному вакуумі" призводить до синтезу так званих "бан-банг" контролерів (bang-bang control), які постійно перемикають напругу між крайніми значеннями $+U_{\max}$ та $-U_{\max}$. Хоча теоретично такий режим може забезпечити найшвидшу дію (принцип максимуму Понтрягіна), на практиці він є неприпустимим для колекторних двигунів та механічних редукторів через виникнення ударних навантажень, іскріння щіток та перегріву. Тому введення агентів штрафу за насичення та різкість керування є не опціональним, а необхідним заходом регуляризації. Це примушує нейромережу знаходити субоптимальні за часом, але реалізовані з точки зору фізики траєкторії.

Таким чином, перелік використаних агентів є необхідним і достатнім набором критеріїв, що покривають повний спектр вимог до системи: кінематичних (точність), динамічних (швидкодія) та експлуатаційних (надійність). Виключення будь-якого з цих компонентів призведе до деградації системи: без агентів точності вона не виконає завдання, а без агентів обмеження — зруйнує виконавчий механізм.

3.7.6. Агент цілевказівки (MoveToGoal)

Цей агент є основним драйвером навчання на початкових етапах. Його завдання — створити щільний градієнт винагороди, який вказує напрямок до цілі. Без цього агента генетичний алгоритм працює "наосліп", випадково перебираючи варіанти.

Агент використовує диференційний принцип оцінювання. На кожному кроці dt він обчислює різницю потенціалів помилки:

$$R_{move} = (Error_{t-1} - Error_t) \cdot K_{move} \quad (3.6)$$

де Error — це абсолютна кутова відстань до цілі.

Значення може бути як додатним, так і від'ємним.

Якщо платформа наблизилася до цілі ($Error_t < Error_{t-1}$), агент видає позитивну винагороду.

Якщо платформа віддалилася, агент нараховує штраф.

Це забезпечує миттєвий зворотний зв'язок: будь-яка дія, що зменшує помилку, миттєво заохочується.

3.7.7. Агент часу (TimeToGoal)

Забезпечення швидкодії системи. Сама по собі точність не гарантує ефективності — можна рухатися до цілі нескінченно повільно. Цей агент створює "еволюційний тиск", змушуючи мережу шукати найкоротші часові траєкторії.

Агент діє як постійний штраф за існування проблеми. Поки платформа знаходиться поза межами заданого порогу точності (threshold), на кожному кроці симуляції нараховується фіксоване від'ємне значення:

$$R_{time} = -W_{time} \quad \text{якщо} \quad Error > Threshold \quad (3.7)$$

Виключно від'ємний або нуль. Це накопичувальний штраф. Чим довше триває перехідний процес, тим меншою буде підсумкова сума балів (Fitness). Це автоматично відфільтровує з популяції повільні ("ліниві") екземпляри нейромереж.

3.7.8. Агент стабілізації (StayInThreshold)

Боротьба з дрейфом та перерегулюванням. У системах без тертя або з високою інерцією досягнення цілі часто супроводжується проскакуванням.

Цей агент стимулює мережу не просто перетнути цільову координату, а зупинитися в ній і утримувати позицію.

Логіка роботи інверсна до агента часу. Винагорода нараховується лише тоді, коли система увійшла в зону допуску:

$$R_{stay} = +W_{stay} \quad \text{якщо} \quad Error \leq Threshold \tag{3.8}$$

Виключно додатний або нуль. Агент формує "плато" високої винагороди навколо цілі. Це змушує нейромережу гасити швидкість при підході до заданої точки, щоб максимізувати час перебування в зоні дії цього агента.

3.7.9. Агент плавності керування (SmoothActionPenalty)

Захист механічного редуктора від ударних навантажень. Нейромережі, особливо рекурентні, схильні генерувати високочастотний шум ("деренчання") керуючого сигналу, коли намагаються компенсувати мінімальні відхилення. Це призводить до зносу шестерень.

Агент аналізує першу похідну керуючого сигналу (швидкість зміни напруги). Штраф розраховується як квадрат різниці між поточним та попереднім значенням виходу мережі:

$$R_{smooth} = -(Action_t - Action_{t-1})^2 \cdot W_{smooth} \tag{3.9}$$

Виключно від'ємний (штраф).

При постійній напрузі або плавній зміні штраф наближається до нуля.

При різкому реверсі ("удар" з -1.0 на +1.0) штраф максимальний. Це змушує мережу генерувати гладкі криві розгону та гальмування.

3.7.10. Агент захисту від насичення (SaturationPenalty)

Захист електричної частини (драйвера та обмоток двигуна) від перегріву. Робота на межі можливостей (100% ШІМ) є енергетично неефективною і небезпечною для силових ключів драйвера.

Агент контролює амплітуду вихідного сигналу нейромережі. Якщо модуль сигналу перевищує заданий поріг (наприклад, 0.9 або 90%), нараховується штраф:

$$R_{sat} = -\max(0, |Action| - Threshold_{sat}) \cdot W_{sat} \quad (3.10)$$

Виключно від'ємний. Агент "обрізає" бажання нейромережі вирішувати задачі виключно за рахунок максимальної потужності, стимулюючи використання інерції системи (розгін -> рух за інерцією -> гальмування).

Налаштування вагових коефіцієнтів (K_{move}, W_{time}, W_{smooth} тощо) дозволяє програмувати бажаний "характер" робота. Наприклад, збільшення ваги W_{time} відносно W_{smooth} призведе до формування агресивного стилю керування, де пріоритетом є швидкість, незважаючи на вібрації. Домінування W_{smooth} призведе до повільного, але дуже плавного руху. Таким чином, система оцінювання перетворює абстрактну задачу навчання у гнучко налаштовуваний інженерний інструмент.

4. РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ НЕЙРОМЕРЕЖЕВОГО КОНТРОЛЕРА

4.1 Аналіз динаміки навчання та формування стратегії керування

Верифікація розробленої системи проводилася на базі програмного середовища «Цифровий двійник», яке моделює динаміку аперіодичної ланки другого порядку з параметрами дискретизації $\Delta t = 0.1$ с. Дослідження процесу навчання, реалізованого методом еволюційних стратегій (ES) з популяцією 50 агентів, дозволило встановити залежність між архітектурою винагород та формуванням керуючої політики.

Аналіз кривих збіжності функції пристосованості (Fitness Function) показує, що процес навчання стабілізується в межах 210–220 епох. Застосування методу дзеркального семплювання (Antithetic Sampling) забезпечило зниження дисперсії оцінки градієнта. На початковому етапі (0–50 епох) система здійснює пошук базових залежностей, мінімізуючи глобальну помилку позиціювання. На етапі 50–150 епох відбувається оптимізація динамічних характеристик: зменшення часу регулювання та усунення осциляцій. Фінальний етап (після 150 епох) характеризується адаптацією ваг рекурентного блоку GRU для забезпечення стійкості в нульовій точці.

Окрему увагу було приділено аналізу впливу компонентів функції винагороди на кінцеву поведінку системи (Ablation Study). Результати почергового виключення агентів продемонстрували наступне:

Відсутність агента цілевказівки (MoveToGoal) унеможливлює збіжність алгоритму. Через розрідженість сигналу винагороди система не здатна сформувати спрямований рух до цілі протягом 400 епох навчання.

Виключення агента стабілізації (StayInThreshold) призводить до дрейфу платформи навколо точки рівноваги. Система досягає заданої координати,

але не фіксує положення, що свідчить про необхідність окремого критерію для термінального стану.

Відключення агентів експлуатації (SmoothAction, Saturation) не погіршує точність позиціювання, але призводить до формування релейного закону керування (bang-bang control). Вихідний сигнал набуває вигляду прямокутних імпульсів максимальної амплітуди, що є неприпустимим для реальних електромеханічних приводів через ризик перегріву та механічного зносу.

Аналіз внутрішніх станів прихованого шару GRU підтверджує, що в процесі навчання мережа сформувала механізм неявної оцінки похідних. Кореляційний аналіз показує залежність активності нейронів прихованого шару від розрахункової кутової швидкості об'єкта. Це дозволяє контролеру враховувати інерцію платформи без використання фізичних датчиків швидкості.

4.2 Порівняльний аналіз ефективності та робастності системи

Для оцінки якості керування проведено порівняння синтезованого нейроконтролера з класичним ПІД-регулятором. ПІД-регулятор було налаштований на оптимальні показники (мінімум часу регулювання при перерегулюванні $\sigma < 1\%$) для номінальних параметрів об'єкта (момент інерції $J = 1.0$). Тестування проводилося у трьох режимах: номінальному, режимі збільшеного навантаження та режимі зменшеного навантаження.

У номінальному режимі ($J = 1.0$) обидві системи демонструють задовільні результати. Нейромережа формує траєкторію, близьку до оптимальної за швидкодією: підтримує максимальну напругу на етапі розгону та формує завчасний гальмівний імпульс. Час перехідного процесу нейроконтролера співмірний з ПІД-регулятором (різниця в межах 5%), при цьому нейромережа забезпечує аперіодичний характер наближення до цілі.

Суттєві відмінності виявлено при дослідженні робастності (стійкості до зміни параметрів об'єкта):

1. Збільшення інерції (**J = 5.0**).

Моделювання роботи з п'ятикратним збільшенням маси платформи виявило недоліки інтегральної складової ПІД-регулятора. Тривалий час розгону призводить до накопичення помилки в інтеграторі (ефект Integrator Windup), що спричиняє значне перерегулювання (понад 40%) та тривалі затухаючі коливання. Нейроконтролер, завдяки наявності рекурентних зв'язків, адаптується до зміни динаміки: фіксуючи повільніше зростання позиції у відповідь на керуючий вплив, мережа коригує момент початку гальмування. Перерегулювання не перевищує 2.5%, процес залишається слабо-коливальним.

2. Зменшення інерції (**J = 0.2**).

При роботі з полегшеною платформою ПІД-регулятор втрачає стійкість. Коефіцієнт підсилення пропорційної складової, розрахований на номінальну масу, виявляється надмірним, що призводить до виникнення незагасаючих автоколивань. Нейромережа зберігає стійкість завдяки функції активації tanh, яка обмежує амплітуду виходу, та навченій політиці плавності (SmoothAction), що демпфує високочастотні збурення.

Кількісні показники перехідних процесів для обох типів контролерів зведено в таблицю 4.1.

Таблиця 4.1. Показники якості керування при варіації параметрів

Режим роботи	Тип контролера	Перерегулювання (σ), %	Час регулювання (tset), с	Характер процесу
Номінальний	ПІД	0.5	2.1	Аперіодичний
(J=1.0)	Нейромережа	0.8	2.2	Аперіодичний
Важкий	ПІД	42.0	> 8.0	Коливальний
(J=5.0)	Нейромережа	2.5	4.5	Аперіодичний
Легкий	ПІД	Нестійкий	-	Автоколивання
(J=0.2)	Нейромережа	0.0	0.8	Аперіодичний

Дані експериментів свідчать, що нейромережевий контролер забезпечує інваріантність показників якості керування в діапазоні зміни інерційного навантаження від 0.2 до 5.0 відносно номіналу, тоді як лінійний регулятор потребує переналаштування коефіцієнтів для кожного режиму роботи.

ВИСНОВКИ

У результаті проведеного дослідження було здійснено комплексний аналіз існуючих систем керування поворотними платформами, включаючи конструктивні схеми, датчики положення та орієнтації, класичні методи керування (PID, LQR) та інтелектуальні підходи (нечітка логіка, нейронні мережі). Це дозволило визначити фундаментальні обмеження традиційних методів щодо необхідності точного математичного моделювання об'єкта та чутливості до параметричних невизначеностей, а також обґрунтувати доцільність розробки нейромережевої системи керування як ефективної альтернативи для роботи в умовах невизначеності та зовнішніх збурень.

На основі проведеного аналізу було розроблено систему керування положенням поворотної платформи, що базується на рекурентній нейромережевій архітектурі з використанням GRU-комірок. Розроблена система включає метод обробки циклічності кутових координат для коректного формування вхідного вектора станів, а також використовує алгоритм навчання на основі еволюційних стратегій з дзеркальним семплюванням для оптимізації параметрів контролера в умовах відсутності апріорних даних. Такий підхід забезпечує високу точність позиціонування та робастність навіть за умови дії зовнішніх збурень та ударних навантажень.

У роботі розроблено цифровий двійник системи керування платформою, що базується на математичній моделі колекторного двигуна та моделі ударних і зовнішніх збурень для реалістичного відтворення умов експлуатації. Для підвищення ефективності процесу навчання було розроблено мультиагентну систему формування винагороди з роздільними компонентами цілевказівки, часу, стабілізації, плавності керування та захисту від насичення, яка включає механізм векторизації обчислень для паралельного навчання множини екземплярів нейромережі.

У рамках практичної реалізації створено комплексне програмне середовище для навчання та тестування нейромережевого контролера, яке

включає модуль імітаційного моделювання фізичної системи та систему оцінювання ефективності з урахуванням багатокритеріальних показників якості. При цьому було реалізовано алгоритми генерації різноманітних траєкторій руху та збурень з контрольованими характеристиками, що дозволило верифікувати ефективність методу шляхом порівняльного аналізу з класичними PID та LQR контролерами.

Практична значимість роботи полягає у створенні системи керування поворотною платформою, здатної забезпечити високоточне позиціонування та стабілізацію в умовах параметричних невизначеностей та зовнішніх збурень без необхідності точного математичного моделювання об'єкта. Наукова новизна роботи визначається розробкою комплексного методу синтезу нейромережевого контролера на основі рекурентної архітектури, що поєднує обробку циклічних кутових координат з мультиагентною системою формування винагороди та технологією еволюційних стратегій, забезпечуючи унікальні показники адаптивності та робастності.Claude is AI and can make mistakes. Please double-check responses.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. Hughes, A., & Drury, B. (2019). Electric Motors and Drives: Fundamentals, Types and Applications (5th ed.). Newnes.
2. Boldea, I., & Nasar, S. A. (2018). The Induction Machine Handbook. CRC Press.
3. Acarnley, P. P. (2002). Stepping Motors: A Guide to Theory and Practice. IET.
4. Crowder, R. M. (2020). Electric Drives and Electromechanical Systems: Applications and Control. Butterworth-Heinemann.
5. Nordin, M., & Gutman, P. O. (2002). Controlling mechanical systems with backlash—a survey. Automatica, 38(10), 1633-1649.
6. Taghirad, H. D., & Belanger, P. R. (1998). Modeling and parameter identification of harmonic drive systems. Journal of Dynamic Systems, Measurement, and Control, 120(4), 439-444.
7. Aghili, F. (2005). A unified approach for control of direct-drive manipulators with structural flexibility and torque ripple using integral manifold. IEEE Transactions on Robotics, 21(5), 856-868.
8. Ellis, G. (2012). Control System Design Guide: Using Your Computer to Understand and Diagnose Feedback Controllers. Academic Press.
9. Schmid, S., & Tröster, G. (2013). Mechanical Design of a High-Precision Rotary Table. Precision Engineering, 37(3), 600-608.
10. Armstrong-Hélouvry, B., Dupont, P., & De Wit, C. C. (1994). A survey of models, analysis tools and compensation methods for the control of machines with friction. Automatica, 30(7), 1083-1138.
11. Fraden, J. (2016). Handbook of Modern Sensors: Physics, Designs, and Applications (5th ed.). Springer.
12. Dimitrievic, B. (2021). Magnetic Sensors and Devices: Technologies and Applications. Artech House.
13. Titterton, D. H., & Weston, J. L. (2004). Strapdown Inertial Navigation Technology (2nd ed.). The Institution of Electrical Engineers.

14. Groves, P. D. (2013). Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems. Artech House.
15. Grewal, M. S., & Andrews, A. P. (2015). Kalman Filtering: Theory and Practice Using MATLAB. Wiley-IEEE Press.
16. Ogata, K. (2010). Modern Control Engineering (5th ed.). Prentice Hall.
17. Dorf, R. C., & Bishop, R. H. (2017). Modern Control Systems (13th ed.). Pearson.
18. Utkin, V., Guldner, J., & Shi, J. (2009). Sliding Mode Control in Electro-Mechanical Systems (2nd ed.). CRC Press.
19. Passino, K. M., & Yurkovich, S. (1998). Fuzzy Control. Addison-Wesley.
20. Levine, W. S. (Ed.). (2011). The Control Handbook: Control System Applications (2nd ed.). CRC Press.
21. Lewis, F. L., Jagannathan, S., & Yesildirek, A. (2020). Neural Network Control of Robot Manipulators and Non-Linear Systems. Taylor & Francis.
22. Mandic, D. P., & Chambers, J. A. (2001). Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability. Wiley.
23. Fleming, P. J., & Purshouse, R. C. (2002). Evolutionary algorithms in control systems engineering: a survey. Control Engineering Practice, 10(11), 1223-1241.
24. Siddique, N., & Adeli, H. (2013). Computational Intelligence: Synergies of Fuzzy Logic, Neural Networks and Evolutionary Computing. John Wiley & Sons.
25. Liu, J. (2017). Intelligent Control Design and MATLAB Simulation. Springer.

ДОДАТКИ

Додаток А
Модуль навчання моделі

```
#!/usr/bin/env python
"""
Table Position Controller - Main CLI Entry Point

Commands:
    train       Train the RNN controller using Evolution Strategies
    evaluate    Evaluate a trained controller
    export      Export a trained model
    demo        Run a demonstration with the fake simulator
"""
from __future__ import annotations
import argparse
import sys
from pathlib import Path


def check_simulator_available(use_fake: bool = False):
    """
    Check if a real simulator is available.

    Args:
        use_fake: If True, allow using fake simulator

    Returns:
        Simulator instance

    Raises:
        SystemExit if no simulator available and use_fake is False
    """
    from Modules.models import TableSimulator, FakeTableSimulator

    if use_fake:
        print("Using FakeTableSimulator for training/evaluation.")
        print("Note: This is for testing only. Implement TableSimulator for real physics.")
        return FakeTableSimulator(random_start_state=True)

    # Try to use real simulator
    try:
        sim = TableSimulator()
        sim.reset()
        return sim
    except NotImplementedError:
        print("\n" + "="*60)
        print("ERROR: TableSimulator is not implemented!")
        print("="*60)
        print("\nThe TableSimulator class in Modules/models.py has placeholder")
        print("methods that need to be implemented with your table dynamics.")
        print("\nRequired methods to implement:")
        print("  - reset(start_pos) -> TableState")
        print("  - step(power_input_ratio, dt) -> TableState")
        print("\nTo test the training pipeline without real physics, use:")
        print("  python main.py train --use-fake-simulator")
        print("="*60 + "\n")
        sys.exit(1)


def setup_gpu():
    """Configure and verify GPU availability."""
    import tensorflow as tf
    import os

    # Try to force GPU usage if available
    os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"

    # Enable memory growth to avoid allocating all GPU memory
    gpus = tf.config.list_physical_devices('GPU')
    if gpus:
        try:
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
            print(f"\n[GPU STATUS] Found {len(gpus)} GPU(s):")
            for gpu in gpus:
```

```
                details = tf.config.experimental.get_device_details(gpu)
                name = details.get('device_name', 'Unknown') if details else 'Unknown'
                print(f"  - {gpu.name}: {name}")
            print("[GPU STATUS] GPU acceleration enabled.\n")
        except RuntimeError as e:
            print(f"[GPU STATUS] Error configuring GPU: {e}")
    else:
        print("\n[GPU STATUS] No GPU detected. Running on CPU.\n")


def cmd_train(args):
    """Train the RNN controller using Evolution Strategies."""
    from Modules.policy_rnn import create_default_policy
    from Modules.train_es import ESConfig, ESTrainer, create_trainer
    from Modules.rollout import EpisodeConfig
    from Modules.rewards import RewardConfig

    # Get simulator
    simulator = check_simulator_available(use_fake=args.use_fake_simulator)

    # Handle multi-goal training configuration
    multi_goal_enabled = getattr(args, 'multi_goal', False)
    goals_per_episode = getattr(args, 'goals_per_episode', 3) if multi_goal_enabled else 1
    goal_segment_steps = getattr(args, 'goal_segment_steps', 300)

    # Calculate actual episode length for multi-goal
    if multi_goal_enabled:
        actual_episode_length = goals_per_episode * goal_segment_steps
        goal_change_interval = goal_segment_steps
    else:
        actual_episode_length = args.episode_length
        goal_change_interval = None

    # Create configurations
    es_config = ESConfig(
        population_size=args.population_size,
        sigma=args.sigma,
        learning_rate=args.learning_rate,
        episodes_per_eval=args.episodes_per_eval,
        max_generations=args.max_generations,
        checkpoint_dir=args.checkpoint_dir,
        checkpoint_interval=args.checkpoint_interval,
        patience=args.patience,
        multi_goal_enabled=multi_goal_enabled,
        goals_per_episode=goals_per_episode,
        goal_segment_steps=goal_segment_steps
    )

    episode_config = EpisodeConfig(
        episode_length=actual_episode_length,
        dt=args.dt,
        min_pos=simulator.min_pos,
        max_pos=simulator.max_pos,
        goal_change_interval=goal_change_interval,
        reset_on_goal_change=True,
        reset_rnn_state_on_goal_change=False
    )

    reward_config = RewardConfig(
        threshold=args.threshold,
        episode_length=actual_episode_length
    )

    # Create policy
    policy = create_default_policy(
        min_pos=simulator.min_pos,
        max_pos=simulator.max_pos
    )

    # Load checkpoint if specified
    if args.resume_from:
        print(f"Resuming from checkpoint: {args.resume_from}")
        trainer = create_trainer(
            simulator=simulator,
            es_config=es_config,
            episode_config=episode_config,
            reward_config=reward_config,
```

```
            policy=policy
        )
        trainer.load_checkpoint(args.resume_from)
    else:
        trainer = create_trainer(
            simulator=simulator,
            es_config=es_config,
            episode_config=episode_config,
            reward_config=reward_config,
            policy=policy
        )

    # Train
    summary = trainer.train(verbose=True, log_interval=args.log_interval)

    print(f"\nTraining complete!")
    print(f"Best model saved to: {args.checkpoint_dir}/final_policy.keras")

    return summary


def cmd_evaluate(args):
    """Evaluate a trained controller."""
    import tensorflow as tf
    from Modules.rollout import evaluate_policy, run_episode, EpisodeConfig
    from Modules.rewards import RewardConfig, RewardPipeline
    from datetime import datetime

    # Get simulator
    simulator = check_simulator_available(use_fake=args.use_fake_simulator)

    # Load model
    model_path = Path(args.model_path)
    if not model_path.exists():
        print(f"Error: Model not found at {model_path}")
        sys.exit(1)

    print(f"Loading model from: {model_path}")
    policy = tf.keras.models.load_model(model_path)

    # Configure evaluation
    episode_config = EpisodeConfig(
        episode_length=args.episode_length,
        dt=args.dt,
        min_pos=simulator.min_pos,
        max_pos=simulator.max_pos
    )

    reward_config = RewardConfig(
        threshold=args.threshold,
        episode_length=args.episode_length
    )

    # Run evaluation
    metrics = evaluate_policy(
        simulator=simulator,
        policy=policy,
        config=episode_config,
        reward_config=reward_config,
        num_episodes=args.num_episodes,
        rng_seed=args.seed,
        verbose=True
    )

    # Generate plot if requested
    if args.plot:
        print("\nGenerating trajectory plot...")
        try:
            from Modules.plotting import plot_episode_trajectory

            # Run one episode with trajectory recording
            reward_pipeline = RewardPipeline.from_config(reward_config)
            plot_seed = args.plot_seed if hasattr(args, 'plot_seed') and args.plot_seed else 999
            trajectory_result = run_episode(
                simulator=simulator,
                policy=policy,
                reward_pipeline=reward_pipeline,
```

```
                config=episode_config,
                rng_seed=plot_seed,
                record_trajectory=True
            )

            if trajectory_result.trajectory:
                # Determine save path
                if args.plot_save:
                    save_path = Path(args.plot_save)
                    if save_path.is_dir():
                        # It's a directory, create filename
                        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
                        model_name = model_path.stem
                        save_path = save_path / f"trajectory_{model_name}_{timestamp}.png"
                else:
                    # Default save location
                    plots_dir = Path("checkpoints/plots")
                    plots_dir.mkdir(parents=True, exist_ok=True)
                    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
                    model_name = model_path.stem
                    save_path = plots_dir / f"trajectory_{model_name}_{timestamp}.png"

                # Plot
                show_plot = not args.no_show if hasattr(args, 'no_show') else True
                plot_episode_trajectory(
                    trajectory=trajectory_result.trajectory,
                    title=f"Episode Trajectory - {model_path.stem}",
                    save_path=str(save_path),
                    show=show_plot
                )

                print(f"Trajectory plot saved to: {save_path}")
                if show_plot:
                    print("Plot displayed interactively.")
            else:
                print("Warning: No trajectory data available for plotting.")

        except ImportError:
            print("ERROR: matplotlib not installed. Install with: pip install matplotlib>=3.7.0")
        except Exception as e:
            print(f"Warning: Could not generate plot: {e}")

    return metrics


def cmd_export(args):
    """Export a trained model to different formats."""
    import tensorflow as tf
    from pathlib import Path

    model_path = Path(args.model_path)
    if not model_path.exists():
        print(f"Error: Model not found at {model_path}")
        sys.exit(1)

    print(f"Loading model from: {model_path}")
    policy = tf.keras.models.load_model(model_path)

    output_path = Path(args.output_path)

    if args.format == "keras":
        output_path = output_path.with_suffix(".keras")
        policy.save(output_path)
        print(f"Saved Keras model to: {output_path}")

    elif args.format == "saved_model":
        tf.saved_model.save(policy, str(output_path))
        print(f"Saved TensorFlow SavedModel to: {output_path}")

    elif args.format == "tflite":
        output_path = output_path.with_suffix(".tflite")
        converter = tf.lite.TFLiteConverter.from_keras_model(policy)
        tflite_model = converter.convert()
        with open(output_path, 'wb') as f:
            f.write(tflite_model)
        print(f"Saved TFLite model to: {output_path}")
```

```
    elif args.format == "weights":
        import numpy as np
        output_path = output_path.with_suffix(".npz")
        flat_weights = policy.get_flat_weights()
        np.savez(output_path, weights=flat_weights)
        print(f"Saved weights to: {output_path}")

    return output_path


def cmd_demo(args):
    """Run a demonstration with the fake simulator."""
    import numpy as np
    from Modules.models import FakeTableSimulator
    from Modules.policy_rnn import create_default_policy, PolicyWrapper

    print("\n" + "="*60)
    print("Table Controller Demo (Fake Simulator)")
    print("="*60)

    # Create simulator and policy
    simulator = FakeTableSimulator(random_start_state=False)
    policy = create_default_policy()
    wrapper = PolicyWrapper(policy)

    # If model path provided, load it
    if args.model_path:
        import tensorflow as tf
        print(f"Loading trained model: {args.model_path}")
        policy = tf.keras.models.load_model(args.model_path)
        wrapper = PolicyWrapper(policy)
    else:
        print("Using untrained (random) policy")

    # Run demo episodes
    for ep in range(args.num_episodes):
        state = simulator.reset(start_pos=args.start_pos)
        goal_pos = args.goal_pos if args.goal_pos else np.random.uniform(10, 90)

        wrapper.reset()

        print(f"\nEpisode {ep + 1}:")
        print(f"  Start position: {state.current_pos:.1f}")
        print(f"  Goal position: {goal_pos:.1f}")

        positions = [state.current_pos]
        actions = []

        for step in range(args.steps):
            action = wrapper.get_action(desired_pos=goal_pos, current_pos=state.current_pos)
            state = simulator.step(action, dt=args.dt)

            positions.append(state.current_pos)
            actions.append(action)

            # Print progress every 10 steps
            if (step + 1) % 10 == 0 or step == 0:
                error = abs(goal_pos - state.current_pos)
                print(f"    Step {step+1:3d}: pos={state.current_pos:6.2f}, "
                      f"action={action:+.3f}, error={error:.2f}")

        final_error = abs(goal_pos - state.current_pos)
        print(f"  Final position: {state.current_pos:.1f}")
        print(f"  Final error: {final_error:.2f}")
        print(f"  Goal reached: {final_error < 1.0}")

    print("\n" + "="*60)
    print("Demo complete!")
    print("="*60 + "\n")


def main():
    parser = argparse.ArgumentParser(
        description="Table Position Controller Training and Evaluation",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
```

```
# Train with fake simulator (for testing)
python main.py train --use-fake-simulator --max-generations 100

# Train with real simulator (requires TableSimulator implementation)
python main.py train --max-generations 1000

# Evaluate a trained model
python main.py evaluate --model-path checkpoints/final_policy.keras

# Run demo
python main.py demo --num-episodes 3

# Export model to TFLite
python main.py export --model-path checkpoints/final_policy.keras --format tflite
    """
)

subparsers = parser.add_subparsers(dest="command", help="Available commands")

# Train command
train_parser = subparsers.add_parser("train", help="Train the controller")
train_parser.add_argument("--use-fake-simulator", action="store_true",
                          help="Use fake simulator for testing")
train_parser.add_argument("--population-size", type=int, default=50,
                          help="ES population size (default: 50)")
train_parser.add_argument("--sigma", type=float, default=0.05,
                          help="Noise standard deviation (default: 0.05)")
train_parser.add_argument("--learning-rate", type=float, default=0.01,
                          help="Learning rate (default: 0.01)")
train_parser.add_argument("--episodes-per-eval", type=int, default=3,
                          help="Episodes per fitness evaluation (default: 3)")
train_parser.add_argument("--max-generations", type=int, default=500,
                          help="Maximum training generations (default: 500)")
train_parser.add_argument("--episode-length", type=int, default=300,
                          help="Steps per episode (default: 300)")
train_parser.add_argument("--dt", type=float, default=0.1,
                          help="Time step in seconds (default: 0.1)")
train_parser.add_argument("--threshold", type=float, default=1.0,
                          help="Position error threshold (default: 1.0)")
train_parser.add_argument("--checkpoint-dir", type=str, default="checkpoints",
                          help="Checkpoint directory (default: checkpoints)")
train_parser.add_argument("--checkpoint-interval", type=int, default=20,
                          help="Generations between checkpoints (default: 20)")
train_parser.add_argument("--patience", type=int, default=50,
                          help="Early stopping patience (default: 50)")
train_parser.add_argument("--log-interval", type=int, default=1,
                          help="Generations between log messages (default: 1)")
train_parser.add_argument("--resume-from", type=str, default=None,
                          help="Checkpoint prefix to resume from")
# Multi-goal training options
train_parser.add_argument("--multi-goal", action="store_true",
                          help="Enable multi-goal training (multiple targets per episode)")
train_parser.add_argument("--goals-per-episode", type=int, default=3,
                          help="Number of goal segments per episode (default: 3)")
train_parser.add_argument("--goal-segment-steps", type=int, default=300,
                          help="Steps per goal segment (default: 300)")
train_parser.set_defaults(func=cmd_train)

# Evaluate command
eval_parser = subparsers.add_parser("evaluate", help="Evaluate a trained controller")
eval_parser.add_argument("--model-path", type=str, required=True,
                         help="Path to trained model")
eval_parser.add_argument("--use-fake-simulator", action="store_true",
                         help="Use fake simulator for testing")
eval_parser.add_argument("--num-episodes", type=int, default=20,
                         help="Number of evaluation episodes (default: 20)")
eval_parser.add_argument("--episode-length", type=int, default=300,
                         help="Steps per episode (default: 300)")
eval_parser.add_argument("--dt", type=float, default=0.1,
                         help="Time step in seconds (default: 0.1)")
eval_parser.add_argument("--threshold", type=float, default=1.0,
                         help="Position error threshold (default: 1.0)")
eval_parser.add_argument("--seed", type=int, default=None,
                         help="Random seed for reproducibility")
eval_parser.add_argument("--plot", action="store_true",
                         help="Generate trajectory plot")
eval_parser.add_argument("--plot-save", type=str, default=None,
```

```
                            help="Path or directory to save plot (default: checkpoints/plots/)")
    eval_parser.add_argument("--no-show", action="store_true",
                            help="Save plot without displaying it interactively")
    eval_parser.add_argument("--plot-seed", type=int, default=999,
                            help="Random seed for trajectory plot (default: 999)")
    eval_parser.set_defaults(func=cmd_evaluate)

    # Export command
    export_parser = subparsers.add_parser("export", help="Export a trained model")
    export_parser.add_argument("--model-path", type=str, required=True,
                               help="Path to trained model")
    export_parser.add_argument("--output-path", type=str, required=True,
                               help="Output path for exported model")
    export_parser.add_argument("--format", type=str, default="keras",
                               choices=["keras", "saved_model", "tflite", "weights"],
                               help="Export format (default: keras)")
    export_parser.set_defaults(func=cmd_export)

    # Demo command
    demo_parser = subparsers.add_parser("demo", help="Run a demonstration")
    demo_parser.add_argument("--model-path", type=str, default=None,
                            help="Path to trained model (uses random if not provided)")
    demo_parser.add_argument("--num-episodes", type=int, default=3,
                            help="Number of demo episodes (default: 3)")
    demo_parser.add_argument("--steps", type=int, default=100,
                            help="Steps per episode (default: 100)")
    demo_parser.add_argument("--dt", type=float, default=0.1,
                            help="Time step in seconds (default: 0.1)")
    demo_parser.add_argument("--start-pos", type=float, default=None,
                            help="Starting position (random if not provided)")
    demo_parser.add_argument("--goal-pos", type=float, default=None,
                            help="Goal position (random if not provided)")
    demo_parser.set_defaults(func=cmd_demo)

    # Parse arguments
    args = parser.parse_args()

    if args.command is None:
        parser.print_help()
        sys.exit(0)

    # Check and configure GPU
    setup_gpu()

    # Run command
    args.func(args)


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

Додаток Б
Модуль опису нейромережі

```
"""
Table position controller interfaces and model definitions.

This module defines:
- TableSimulator: Interface for table physics simulation (NOT IMPLEMENTED - dynamics pending)
- TableControllerModel: Keras model wrapper for RNN-based position controller
"""
from __future__ import annotations
from dataclasses import dataclass
from typing import Optional, Dict
from abc import ABC, abstractmethod
import math


@dataclass
class TableState:
    """
    State representation for the table simulator.

    Attributes:
        current_pos: Current position of the table (in position units, e.g., 0-100)
        current_vel: Current velocity of the table (optional, for future use)
    """
    current_pos: float
    current_vel: float = 0.0

    def to_dict(self) -> Dict[str, float]:
        return {"current_pos": self.current_pos, "current_vel": self.current_vel}


class TableSimulatorInterface(ABC):
    """
    Abstract interface for table physics simulation.

    The simulator models a table that can be moved to a desired position
    using a power input (e.g., motor power ratio). Implementations should
    model realistic dynamics including inertia, friction, and response lag.

    Expected I/O Contract:
        - Observation: (desired_pos, current_pos)
        - Action: power_input_ratio in [-1, 1] for bidirectional control
          (positive = move in positive direction, negative = move in negative direction)
    """

    @property
    @abstractmethod
    def min_pos(self) -> float:
        """Minimum allowed position."""
        pass

    @property
    @abstractmethod
    def max_pos(self) -> float:
        """Maximum allowed position."""
        pass

    @property
    @abstractmethod
    def current_pos(self) -> float:
        """Current position of the table."""
        pass

    @abstractmethod
    def reset(self, start_pos: Optional[float] = None) -> TableState:
        """
        Reset the simulator to initial state.

        Args:
            start_pos: Optional starting position. If None, may randomize
                       based on implementation settings.
```

```
        Returns:
            TableState: Initial state after reset, containing at minimum current_pos.
        """
        pass

    @abstractmethod
    def step(self, power_input_ratio: float, dt: float) -> TableState:
        """
        Advance the simulation by one time step.

        Args:
            power_input_ratio: Motor power ratio in [-1, 1].
                               Positive values move in positive direction.
                               Negative values move in negative direction.
            dt: Time step duration in seconds.

        Returns:
            TableState: New state after applying the action, containing current_pos.
        """
        pass


def wrap_angle_deg(angle: float) -> float:
    """
    Wrap angle to [0, 360).
    """
    return angle % 360.0


def shortest_angular_difference(target_deg: float, current_deg: float) -> float:
    """
    Compute the signed shortest angular difference between target and current in degrees.

    Returns value in [-180, 180], where positive means target is ahead in positive direction.
    """
    diff = (target_deg - current_deg + 180.0) % 360.0 - 180.0
    return diff


class TableSimulator(TableSimulatorInterface):
    """
    Table physics simulator.

    *** DYNAMICS NOT IMPLEMENTED ***
    This class defines the interface contract but does not implement
    actual physics. Implement the dynamics based on your specific
    table hardware characteristics (mass, friction, motor response, etc.).

    Expected properties to calibrate (examples):
        - table_mass: Mass affecting inertia
        - motor_max_force: Maximum force the motor can apply
        - friction_coefficient: Static/dynamic friction
        - damping: Velocity-dependent resistance
    """

    def __init__(
        self,
        random_start_state: bool = True,
        min_pos: float = 0.0,
        max_pos: float = 360.0,
        inertia: float = 0.02,           # rotational inertia (kg*m^2)
        damping: float = 0.08,           # viscous friction coefficient
        torque_constant: float = 0.6,    # N*m at full power
        max_torque: float = 0.6,         # cap torque
        # Domain randomization (set to True for better generalization)
        randomize_physics: bool = False,
        inertia_range: tuple = (0.015, 0.025),      # ±25% variation
        damping_range: tuple = (0.06, 0.10),        # ±25% variation
        torque_range: tuple = (0.5, 0.7),           # ±17% variation
    ):
        """
        Initialize the table simulator.

        Args:
            random_start_state: If True, reset() randomizes starting position.
            min_pos: Minimum position boundary.
```

```
            max_pos: Maximum position boundary.
        """
        self._min_pos = min_pos
        self._max_pos = max_pos
        self._random_start_state = random_start_state
        self._current_pos = (min_pos + max_pos) / 2.0
        self._current_vel = 0.0

        # Store base physics parameters
        self._base_inertia = inertia
        self._base_damping = damping
        self._base_torque_constant = torque_constant
        self._base_max_torque = max_torque

        # Domain randomization settings
        self.randomize_physics = randomize_physics
        self.inertia_range = inertia_range
        self.damping_range = damping_range
        self.torque_range = torque_range

        # Current physics parameters (may be randomized)
        self.inertia = inertia
        self.damping = damping
        self.torque_constant = torque_constant
        self.max_torque = max_torque

    @property
    def min_pos(self) -> float:
        return self._min_pos

    @property
    def max_pos(self) -> float:
        return self._max_pos

    @property
    def current_pos(self) -> float:
        return self._current_pos

    @property
    def current_vel(self) -> float:
        return self._current_vel

    def reset(self, start_pos: Optional[float] = None) -> TableState:
        """
        Reset the simulator to an initial state.

        If domain randomization is enabled, physics parameters are
        randomized on each reset to improve policy robustness.
        """
        import random

        # Randomize physics parameters if enabled
        if self.randomize_physics:
            self.inertia = random.uniform(*self.inertia_range)
            self.damping = random.uniform(*self.damping_range)
            torque = random.uniform(*self.torque_range)
            self.torque_constant = torque
            self.max_torque = torque
        else:
            # Use base parameters
            self.inertia = self._base_inertia
            self.damping = self._base_damping
            self.torque_constant = self._base_torque_constant
            self.max_torque = self._base_max_torque

        # Reset position and velocity
        if start_pos is not None:
            self._current_pos = wrap_angle_deg(start_pos)
        elif self._random_start_state:
            self._current_pos = random.uniform(self._min_pos, self._max_pos)
        else:
            self._current_pos = (self._min_pos + self._max_pos) / 2.0
        self._current_pos = wrap_angle_deg(self._current_pos)
        self._current_vel = 0.0
        return TableState(current_pos=self._current_pos, current_vel=self._current_vel)

    def step(self, power_input_ratio: float, dt: float) -> TableState:
```

```
        """
        Advance simulation by one timestep using a second-order aperiodic model.
        """
        # Clamp input
        u = max(-1.0, min(1.0, power_input_ratio))

        # Torque from motor
        torque = max(-self.max_torque, min(self.max_torque, self.torque_constant * u))

        # Dynamics: J * w_dot = torque - b * w
        angular_acc = (torque - self.damping * self._current_vel) / self.inertia

        # Integrate velocity and position
        self._current_vel += angular_acc * dt
        self._current_pos = wrap_angle_deg(self._current_pos + self._current_vel * dt)

        return TableState(current_pos=self._current_pos, current_vel=self._current_vel)

    def update(self, power_input_ratio: float, dt: float) -> float:
        """
        Legacy interface - calls step() and returns position.

        Maintained for compatibility. Prefer using step() directly.
        """
        state = self.step(power_input_ratio, dt)
        return state.current_pos


# ============================================================================
# Vectorized Simulators for Batched GPU Training
# ============================================================================

class VectorizedTableSimulator:
    """
    Vectorized table simulator for parallel batch simulation.

    Runs N independent simulations simultaneously using NumPy vectorization.
    This enables batched GPU inference by stepping all simulations in lockstep.
    """

    def __init__(
        self,
        num_envs: int,
        min_pos: float = 0.0,
        max_pos: float = 360.0,
        inertia: float = 0.02,
        damping: float = 0.08,
        torque_constant: float = 0.6,
        max_torque: float = 0.6,
        random_start: bool = True,
        # Domain randomization
        randomize_physics: bool = False,
        inertia_range: tuple = (0.015, 0.025),
        damping_range: tuple = (0.06, 0.10),
        torque_range: tuple = (0.5, 0.7)
    ):
        """
        Initialize vectorized simulator.

        Args:
            num_envs: Number of parallel environments
            min_pos: Minimum position (degrees)
            max_pos: Maximum position (degrees)
            inertia: Rotational inertia (kg*m^2)
            damping: Viscous friction coefficient
            torque_constant: N*m at full power
            max_torque: Maximum torque cap
            random_start: Randomize starting positions
        """
        import numpy as np
        self.num_envs = num_envs
        self._min_pos = min_pos
        self._max_pos = max_pos
        self.random_start = random_start

        # Base physics parameters
        self._base_inertia = inertia
```

```
        self._base_damping = damping
        self._base_torque_constant = torque_constant
        self._base_max_torque = max_torque

        # Domain randomization settings
        self.randomize_physics = randomize_physics
        self.inertia_range = inertia_range
        self.damping_range = damping_range
        self.torque_range = torque_range

        # Current physics parameters (may be randomized per env)
        # Shape: (num_envs,) for vectorized randomization
        self.inertia = np.full(num_envs, inertia, dtype=np.float32)
        self.damping = np.full(num_envs, damping, dtype=np.float32)
        self.torque_constant = np.full(num_envs, torque_constant, dtype=np.float32)
        self.max_torque = np.full(num_envs, max_torque, dtype=np.float32)

        # State arrays: (num_envs,)
        self.positions = np.zeros(num_envs, dtype=np.float32)
        self.velocities = np.zeros(num_envs, dtype=np.float32)

    @property
    def min_pos(self) -> float:
        return self._min_pos

    @property
    def max_pos(self) -> float:
        return self._max_pos

    def reset(self, start_positions: Optional[np.ndarray] = None) -> np.ndarray:
        """
        Reset all environments.

        If domain randomization is enabled, physics parameters are
        randomized independently for each environment.

        Args:
            start_positions: Optional array of starting positions (num_envs,)

        Returns:
            positions: Current positions after reset (num_envs,)
        """
        import numpy as np

        # Randomize physics parameters if enabled
        if self.randomize_physics:
            self.inertia = np.random.uniform(
                self.inertia_range[0], self.inertia_range[1],
                size=self.num_envs
            ).astype(np.float32)
            self.damping = np.random.uniform(
                self.damping_range[0], self.damping_range[1],
                size=self.num_envs
            ).astype(np.float32)
            torque = np.random.uniform(
                self.torque_range[0], self.torque_range[1],
                size=self.num_envs
            ).astype(np.float32)
            self.torque_constant = torque
            self.max_torque = torque
        else:
            # Use base parameters for all environments
            self.inertia.fill(self._base_inertia)
            self.damping.fill(self._base_damping)
            self.torque_constant.fill(self._base_torque_constant)
            self.max_torque.fill(self._base_max_torque)

        # Reset positions and velocities
        if start_positions is not None:
            self.positions = start_positions.astype(np.float32) % 360.0
        elif self.random_start:
            self.positions = np.random.uniform(
                self._min_pos, self._max_pos, size=self.num_envs
            ).astype(np.float32)
        else:
            self.positions = np.full(
                self.num_envs,
```

```
                (self._min_pos + self._max_pos) / 2.0,
                dtype=np.float32
            )
        self.velocities = np.zeros(self.num_envs, dtype=np.float32)
        return self.positions.copy()

    def step(self, actions: np.ndarray, dt: float) -> np.ndarray:
        """
        Step all environments simultaneously.

        Args:
            actions: Power input ratios (num_envs,) in [-1, 1]
            dt: Time step in seconds

        Returns:
            positions: New positions after step (num_envs,)
        """
        import numpy as np
        # Clamp actions
        u = np.clip(actions, -1.0, 1.0).astype(np.float32)

        # Compute torque
        torque = np.clip(self.torque_constant * u, -self.max_torque, self.max_torque)

        # Dynamics: J * w_dot = torque - b * w
        angular_acc = (torque - self.damping * self.velocities) / self.inertia

        # Integrate
        self.velocities += angular_acc * dt
        self.positions = (self.positions + self.velocities * dt) % 360.0

        return self.positions.copy()


class VectorizedFakeSimulator:
    """
    Vectorized fake simulator for testing the batched training pipeline.

    Uses simple kinematics for fast testing.
    """

    def __init__(
        self,
        num_envs: int,
        min_pos: float = 0.0,
        max_pos: float = 360.0,
        speed_factor: float = 10.0,
        random_start: bool = True
    ):
        import numpy as np
        self.num_envs = num_envs
        self._min_pos = min_pos
        self._max_pos = max_pos
        self.speed_factor = speed_factor
        self.random_start = random_start

        self.positions = np.zeros(num_envs, dtype=np.float32)
        self.velocities = np.zeros(num_envs, dtype=np.float32)

    @property
    def min_pos(self) -> float:
        return self._min_pos

    @property
    def max_pos(self) -> float:
        return self._max_pos

    def reset(self, start_positions: Optional[np.ndarray] = None) -> np.ndarray:
        import numpy as np
        if start_positions is not None:
            self.positions = start_positions.astype(np.float32) % 360.0
        elif self.random_start:
            self.positions = np.random.uniform(
                self._min_pos, self._max_pos, size=self.num_envs
            ).astype(np.float32)
        else:
            self.positions = np.full(
```

```
                self.num_envs,
                (self._min_pos + self._max_pos) / 2.0,
                dtype=np.float32
            )
        self.velocities = np.zeros(self.num_envs, dtype=np.float32)
        return self.positions.copy()

    def step(self, actions: np.ndarray, dt: float) -> np.ndarray:
        import numpy as np
        actions = np.clip(actions, -1.0, 1.0).astype(np.float32)
        target_vel = actions * self.speed_factor
        alpha = min(1.0, dt * 5.0)
        self.velocities = self.velocities * (1 - alpha) + target_vel * alpha
        self.positions = (self.positions + self.velocities * dt) % 360.0
        return self.positions.copy()


import numpy as np


# For testing purposes only - a minimal working simulator
class FakeTableSimulator(TableSimulatorInterface):
    """
    Minimal fake simulator for testing the training pipeline.

    Uses simple kinematics: pos += action * speed_factor * dt
    This is NOT physically realistic - only for code path validation.
    """

    def __init__(
        self,
        random_start_state: bool = True,
        min_pos: float = 0.0,
        max_pos: float = 360.0,
        speed_factor: float = 10.0
    ):
        self._min_pos = min_pos
        self._max_pos = max_pos
        self._random_start_state = random_start_state
        self._speed_factor = speed_factor
        self._current_pos = (min_pos + max_pos) / 2.0
        self._current_vel = 0.0

    @property
    def min_pos(self) -> float:
        return self._min_pos

    @property
    def max_pos(self) -> float:
        return self._max_pos

    @property
    def current_pos(self) -> float:
        return self._current_pos

    def reset(self, start_pos: Optional[float] = None) -> TableState:
        import random
        if start_pos is not None:
            self._current_pos = wrap_angle_deg(start_pos)
        elif self._random_start_state:
            self._current_pos = random.uniform(self._min_pos, self._max_pos)
        else:
            self._current_pos = (self._min_pos + self._max_pos) / 2.0
        self._current_vel = 0.0
        return TableState(current_pos=self._current_pos, current_vel=self._current_vel)

    def step(self, power_input_ratio: float, dt: float) -> TableState:
        # Clamp action
        action = max(-1.0, min(1.0, power_input_ratio))

        # Simple velocity model with some damping
        target_vel = action * self._speed_factor
        # Smooth velocity transition (simple low-pass filter)
        alpha = min(1.0, dt * 5.0)  # Smoothing factor
        self._current_vel = self._current_vel * (1 - alpha) + target_vel * alpha

        # Update position
```

```
self._current_pos = wrap_angle_deg(self._current_pos + self._current_vel * dt)
return TableState(current_pos=self._current_pos, current_vel=self._current_vel)
```

Бібліографічна довідка

Тема роботи: Розроблення системи керування положенням поворотної платформи на основі нейромережевих алгоритмів

Обсяг магістерської роботи: 103 сторінки

Перелік графічного матеріалу:

МР.АКСм-06.00.00.001 Об'єкт керування і його математична модель

МР.АКСм-06.00.00.002 Архітектура розробленої нейромережі для керування поворотною платформою

МР.АКСм-06.00.00.003 Реалізація процесу навчання розробленої нейромережі

МР.АКСм-06.00.00.004 Аналіз ефективності агентів навчання

МР.АКСм-06.00.00.005 Порівняльний аналіз ефективнсоті застосування нейромережевого контролера

Дата закінчення МР _____