Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

# Курсова робота

на тему: «**PID КОНТРОЛЕР УПРАВЛІННЯ РУХОМ АВТОНОМНОЇ СИСТЕМИ У ЗАМКНЕНОМУ ПРОСТОРІ**»

Виконав: студент 3-го року навчання,

Освітньої програми

«Комп’ютерні науки», 122

Мачехін Назар Олексійович

Керівник: Вознюк Ярослав Іванович

Київ – 2025

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики

ЗАТВЕРДЖУЮ
Зав. Кафедри
_____
*(підпис)*_____
“____” _____ 2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студента Мачехіна Назара Олексійовича факультету інформатики 3 курсу
ТЕМА PID контролер управління рухом автономної системи у замкненому просторі

Індивідуальне завдання: Розробка API на основі PID контролера для картографування замкнутого простору за допомогою лідару та подальша навігація в ньому з урахуванням віртуальних перешкод.
Зміст ТЧ до курсової роботи:
Календарний план
Зміст
Перелік умовних позначень
Вступ
Розділ 1: Аналіз даних та поставленого завдання.
Розділ 2: Теоретична частина
Розділ 3: Програмна реалізація

Розділ 4: Результати та їх обговорення

Висновки

Список використаної літератури

Додатки

Дата видачі “____” _____ 2025 р. Керівник_____

Завдання отримав _____

*(підпис)* *(підпис)*

# КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапу | Термін виконання |
|---|---|---|
| 1. | Отримання теми курсової роботи | 28.10.2024 |
| 2. | Постановка індивідуального завдання | 28.11.2024 |
| 3. | Вивчення літератури | 10.11.2024 |
| 4. | Вибір алгоритмів та засобів | 31.12.2024 |
| 5. | Ознайомлення з симулятором, побудова середовища | 20.01.2025 |
| 6. | Початок розробки API | 01.02.2025 |
| 7. | Тестування та налагодження алгоритмів | 05.03.2025 |
| 8. | Симуляція роботи та аналіз роботи API | 17.03.2025 |
| 9. | Початок написання тексту роботи | 31.03.2025 |
| 10. | Редагування роботи | 30.04.2025 |
| 11. | Створення презентації для захисту | 05.05.2025 |
| 12. | Захист роботи | 15.05.2025 |

# ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

1. PID – Пропорційно-інтегрально-диференційний регулятор, що використовується для зменшення відхилення регульованої величини від заданого значення за допомогою трьох основних складових: пропорційної, інтегральної та диференційної.
2. LIDAR (лідар) – Light Detection and Ranging, лазерний далекомір, який здатний вимірювати відстань до об’єкту за допомогою аналізу часу затримки між моментом посиланням лазерного імпульсу та зчитування його відбиття.
3. API – Application Programming Interface, інтерфейс прикладного програмування, який представляє собою набір функцій, процедур, правил та протоколів, що дозволяють одній програмі взаємодіяти з іншою.
4. Перешкода – фізичний або віртуальний об’єкт у просторі з чіткими межами та перебування в межах якого є не допустимим для системи.
5. Замкнутий простір – середовище, що чітко обмежене фізичними або штучними перешкодами, що унеможливлюють вільне проникнення об’єктів ззовні та вільне переміщення за його межі.
6. Стартове положення – точка в замкнутому просторі в якій перебуває автономна система в момент початку руху, а також напрямок в який спрямована система.
7. Поточне положення – це точка в замкненому просторі, яка характеризує просторовий стан системи в певний момент часу. Її координати визначаються відносно стартового положення, а кут напрямку руху вимірюється відносно початкового напряму, приймаючи до уваги гіпотетичний випадок, за якого система не зазнала зміщення.
8. Робот – автономна роботизована система, що функціонує на основі рухомої платформи та оснащена засобами пересування, датчиками вимірювання навколишнього середовища, механізмом розрахунку власного розташування

відносно стартової точки, а також апаратними засобами призначеними для здійснення керування системою та розгортання програмного забезпечення.

9. Розворот на місці – обертальний рух автономної системи, під час якого змінюється її орієнтація в просторі без лінійного переміщення у будь-якому напрямку.
10. Центральна точка – точка, розташована в межах корпусу робота, яка залишається відносно нерухомою під час виконання обертання на місці. Вона слугує базовою координатою для визначення поточного положення системи в просторі та розрахунку відносного розташування перешкод.
11. SLAM – Simultaneous localization and mapping, алгоритм побудови карти навколишнього середовища з відслідковуванням руху у ньому.
12. RRT – Rapidly exploring random tree, алгоритм пошуку траєкторії руху в просторі за рахунок випадкової генерації вершин та побудови на їх основі дерева можливих маршрутів.
13. BFS – Breadth-first search, алгоритм пошуку в ширину на графі.
14. Карта – віртуальна модель замкнутого простору навколо робота, що зберігає інформацію про відомі перешкоди та пусті ділянки.

# ВСТУП

Початок повномасштабної війни в Україні ознаменував нову епоху розвитку технологій як у нашій державі, так і в усьому світі. Серед іншого, війна продемонструвала, що автономні системи більше не є прерогативою виключно великих корпорацій або заможних ентузіастів: дрони та інші роботизовані платформи стали доступними широкому колу користувачів і здатні виконувати складні завдання в автономному режимі. Широка публіка нарешті усвідомила, що розробка та застосування таких систем доступна для будь-кого, хто володіє відповідними знаннями, ресурсами та мотивацією.

Однією з найактуальніших задач, що продовжує досліджуватися і розвиватися, є проблема автономної навігації як у відкритому просторі, так і всередині закритих приміщень. Сучасні алгоритми навігації застосовуються у найрізноманітніших пристроях: від роботів-пилососів і кур'єрських систем на складах до розмінувальних роботів і самокерованих безпілотників.

Одним із базових елементів систем автономного руху є PID-регулятор. Завдяки простоті математичного апарату та можливості реалізації на малопотужних обчислювальних пристроях, PID-регулятори широко застосовуються для стабілізації руху, компенсації інерційності середовища та забезпечення точності виконання маневрів. У навігаційних системах дані регулятори зазвичай використовуються для допоміжних функцій: стабілізації руху та компенсації відхилень, тоді як складніша логіка прокладання траєкторії, уникнення перешкод і прийняття рішень реалізується через окремі планувальні алгоритми. Однак використання ресурсомістких планувальників є недоліком, особливо в умовах обмежених обчислювальних ресурсів або великих навігаційних просторів.

*Актуальність проблеми* навігації в автономних системах та перспективність застосування PID-регуляторів зумовили вибір теми даної роботи.

*Метою дослідження* стала розробка API для роботизованих систем, яке здійснюватиме картографування та подальшу навігацію у замкненому просторі, використовуючи систему, засновану на PID-регуляторах для повного керування рухом до заданої позиції, базуючись виключно на даних, зібраних автономною системою під час роботи, з можливістю встановлення заборонених зон.

Мета роботи зумовила постановку наступного наукового завдання:

1. Виконати аналіз існуючих рішень та алгоритмів навігації в автономних системах.
2. Розробити систему розрахунку руху, основану на PID-регуляторах, що забезпечує рух у просторі, уникнення реальних та віртуальних перешкод, виконання поставлених завдань.
3. Обґрунтувати вибір технологій для реалізації API, що забезпечить його придатність для практичного застосування.
4. Проаналізувати працездатність та ефективність розробленого рішення через експериментальну перевірку в контрольованих умовах.
5. Зробити висновки щодо практичності та ефективності запропонованого підходу.

Об'єктом дослідження роботи є процес автономної навігації роботизованих систем у замкненому просторі, в той час як предметом дослідження — застосування PID-регуляторів для повного керування рухом автономної системи на основі даних, отриманих під час роботи.

Методи дослідження включають огляд існуючих систем і підходів, розробку математичних моделей та огляд існуючих алгоритмів, симулювання та тестування програмного забезпечення в межах віртуального середовища.

Робота складається з чотирьох основних розділів.

У першому розділі здійснюється аналіз проблеми навігації та основні моменти поставленого завдання, розглядаються існуючі рішення та роль PID-регуляторів у сучасних автономних системах.

Другий розділ присвячено теоретичним основам запропонованого підходу, опису алгоритмів і моделей.

Третій розділ містить опис програмної реалізації, обґрунтування вибору технологій та структури API.

Четвертий розділ присвячено тестуванню, оцінці результатів роботи системи та формулюванню потенційних недоліків і шляхів покращень.

# РОЗДІЛ 1: АНАЛІЗ ДАНИХ ТА ПОСТАВЛЕНОГО ЗАВДАННЯ

## 1.1 Огляд існуючих рішень з використанням PID регулятора

Перш ніж розпочинати роботу, слід ознайомитися з уже існуючими рішеннями навігації в невідомому середовищі та визначити, яку роль в них відіграє PID-контролер, щоб знайти можливі місця для покращення і продемонструвати актуальність дослідження. В межах аналізу наукових статей були знайдені наступні приклади застосування PID.

У роботі [1] автор вивчає покращення навігації робота за допомогою Hector SLAM на основі операційної системи ROS. Як зазначає автор в анотації: “На відміну від інших SLAM методів, Hector SLAM працює без використання одометрії, спираючись на дані з лідару для створення точних карт...” [1, с. 1] (переклад з англійської). Автор використовує PID для покращення точності руху в середовищі та точності картографування, а для руху та створення траєкторії застосовуються алгоритми Дейкстри та A* [1, с. 598-599].

У статті [2] автор досліджує алгоритм ухилення від перешкод за допомогою PID-регулятора для безпілотних літальних апаратів. Хоча чітку аналогію між БЛА та автоматизованими системами, розглянутими в цій роботі, провести складно, все ж варто звернути увагу на застосування PID. Як зазначено в дослідженні: “Дві моделі безпілотних літальних засобів згенеровані на початковій позиції і контролюються PID-контролером для слідування оптимальному маршруту, врешті досягаючи кінцевої точки” [2, с. 10] (переклад з англійської).

У роботі [3] автори досліджують PID з нечіткою логікою, так званий "Fuzzy PID", для керування роботом та уникнення перешкод, виявлених візуально в реальному часі. В цьому випадку, при виявленні перешкоди, робот визначає її центр і будує кругову траєкторію оминання, якою слідує [3, с. 12]. Метою роботи є розробка модифікованого PID для контролю за швидкістю [3, с. 3]. У висновку можна зазначити, що в даному випадку PID є радше допоміжним інструментом для виконання маршруту, який будується окремим планувальником.

На противагу вищезгаданим використанням, існує робота, в якій PID використовується для контролю руху до вказаного пункту призначення та ухилення від перешкод. В роботі [4] автор використовує PID для обходу перешкод та навігації розумного автомобіля на основі даних з лідара, що сканує середовище навколо системи. Під час навігації дані про розташування машини та об’єктів навколо використовуються для визначення напрямку руху, після чого PID застосовується для спрямування в потрібний бік [4, с. 34]. Хоч таке рішення й є дуже близьким до запропонованого в поточній роботі, воно має деякі недоліки, такі як відсутність контролю відстані до перешкоди, залежність від лідару, який дозволяє бачити практично в усіх напрямках, розгортання програмного забезпечення на окремій обчислювальній машині та відсутність явного використання вже розмічених перешкод.

Таким чином, в існуючих дослідженнях PID-регулятор зазвичай використовується як засіб дотримання маршруту, який генерується окремою системою з урахуванням знайдених на шляху перешкод. Хоча дослідження стосовно повного контролю руху за допомогою регулятора проводилися, вони все ще мають недоліки, що робить поточне дослідження актуальним.

## 1.2 Аналіз постановки завдання

Аналіз постановки завдання є першим кроком у розробці системи. Для подальшої роботи необхідно чітко визначити основні етапи та пункти, які потрібно реалізувати.

По-перше, важливим етапом є розробка API, що включає створення основних структур даних, конфігурацій та методів комунікації між програмним забезпеченням та автономною системою, на якій воно буде розгорнуте. Потрібно чітко визначити формат вхідних і вихідних даних, що забезпечить коректну взаємодію між компонентами системи.

Наступним кроком є розробка методів збереження та обробки зібраних даних. Це дозволить здійснювати контроль за рухом системи та забезпечити можливість відновлення роботи в уже розміченому середовищі, зберігаючи необхідну інформацію для подальшої навігації.

На основі збережених даних має бути побудована система управління рухом, яка використовуватиме один або декілька PID-регуляторів для досягнення вказаної точки призначення, враховуючи можливі перешкоди на шляху. Це буде центральною частиною API і безпосередньо впливатиме на процес керування. На цьому етапі також необхідно розробити допоміжні системи для навігації в просторі.

Завершивши розробку API, необхідно перейти до симуляцій, тестування та налагодження програмного забезпечення. Це дозволить провести детальний аналіз роботи системи та оцінити її ефективність.

## 1.3 Підсумки

У першому розділі дослідження було розглянуто реальні випадки застосування PID-регуляторів, що показало, що використання PID як основної

системи для руху в замкненому просторі є рідкісним і недосконалим, тим самим підтверджуючи актуальність виконаної роботи.

Аналіз постановки завдання та чітке визначення етапів розробки дозволили сформулювати чіткий план дій для побудови навігаційної системи. Це, в свою чергу, визначило напрямок подальших досліджень та етапів реалізації, що є основою для наступних етапів розробки.

# РОЗДІЛ 2: ТЕОРЕТИЧНА ЧАСТИНА

## 2.1 Допущення та умовності

Перед початком виконання роботи приймемо за основу наступні факти: Простір закритої системи вважається таким, що не містить перепадів висоти або складної багаторівневої структури, тобто може розглядатися як двовимірна площина. API не здійснює керування роботом безпосередньо, а лише надає інтерфейс для системи навігації. Автономна система оснащена власним програмним забезпечення, що відповідає за зчитування значень з датчиків, їх обробку та представлення в необхідному форматі, визначення поточного положення в просторі, а також взаємодію з наданим API з подальшою обробкою та виконанням отриманих від нього команд. В межах даної роботи проблеми неточності або некоректності показників датчиків, впливу несприятливих умов середовища чи збоїв в роботі апаратної частини розглядаються як такі, що не стосуються роботи API і не вирішуються ним. Відповідно, припускається, що всі отримані дані є актуальними та достатньо точними, а можлива похибка не впливає на кінцевий результат. Робота розглядається як такий, що здатний виконувати розворот на місці, центральна точка якого не значно відхиляється від геометричного центру корпусу. Також припускається, що платформа може здійснювати рух завдяки незалежному керуванню приводами з протилежних боків.

## 2.2 PID-регулятор: принцип роботи

Головною метою PID-регулятора є мінімізація похибки між поточним значенням змінної системи та чітко встановленою бажаною величиною [5, с. 1].

Така модель обчислює керуючий сигнал, що впливає на систему з метою наближення її до бажаного стану, і складається з трьох основних компонентів: пропорційної складової (P), інтегральної (I) та диференційної (D), кожна з яких має власний коефіцієнт налаштування. PID-регулятор також враховує динаміку зміни похибки, а при правильному налаштуванні забезпечує плавне коригування та зменшує коливання значень.

Існують також спрощені варіанти регуляторів, що є його окремими випадками: P, I, D, PI та PD. У більшості випадків немає потреби розглядати та реалізовувати окремі моделі для кожного з них, оскільки повнофункціональний PID дозволяє відключати окремі компоненти шляхом встановлення відповідних коефіцієнтів у нуль.

Принцип роботи PID-регулятора можна розглядати окремо для кожної з його складових.

Пропорційна складова ґрунтується на поточному значенні похибки та формує корекцію, пропорційну до її величини. Таким чином, вона сприяє зменшенню відхилення. Водночас у випадках коли похибка є незначною, ефективність пропорційного керування знижується [5, с. 2-3].

Інтегральна складова враховує інтеграл відхилення за часом, тобто накопичення похибки, і діє пропорційно до цього накопичення. Це дозволяє компенсувати систематичне відхилення та досягти бажаного значення, однак при надмірному значенні інтегрального коефіцієнта регулятор може вносити корекцію навіть тоді, коли бажаний стан досягнутий або близький до такого, спричиняючи коливання [5, с. 2-3].

Для згладжування можливих коливань застосовується диференційна складова, яка реагує на швидкість зміни похибки, тобто на її похідну за часом. Дана складова забезпечує випереджувальну корекцію, що сприяє стабілізації системи та забезпечує більш плавну реакцію [5, с. 2-3].

В підсумку, весь математичний апарат PID-регулятора можна звести до однієї формули наступного вигляду:

$$f(e,t) = k_p e(t) + k_i \int_0^t e(t)dt + k_d \frac{de(t)}{dt}$$

Де e – абсолютне значення похибки, t – період часу що пройшов з моменту початку роботи регулятора, а $k_p, k_i, k_d$ – відповідні коефіцієнти кожного з параметрів.

### 2.3 Збір даних та їх подача

Однією з ключових вимог до подачі даних для коректної взаємодії з API є стандартизована обробка інформації, що збирається та обчислюється під час виконання алгоритму.

На початку роботи програми центральна точка робота приймається за нульову позицію в закритому просторі та збігається зі стартовою точкою. У випадку подання простору як двовимірної площини координати приймаються за (0; 0) по осях абсцис та ординат відповідно, а кут напрямку руху рівний 0 радіан. Надалі положення робота визначаються відносно стартової точки та подається у вигляді координат по вказаних осях. Кут напрямку руху обчислюється в межах [0; 2π) радіан з урахуванням правила, прийнятого при визначенні терміна *поточне положення*.

Дані отримані з лідара, декількох лідарів або функціонально подібних датчиків, подаються у вигляді координат радіальної системи, [d, φ], де d – відстань до об’єкта, а φ – кут. Центром цієї системи відліку є центральна точка робота, а напрямок φ=0 радіан збігається з поточним напрямком руху. Допустимим діапазоном значень φ є проміжок [0; 2π), а відстані до d – як [0; +∞]. значення d повинно бути явним і більшим за максимальний діапазон сенсора з

найбільшим покриттям. Дотримання наведених умов дозволяє об’єднати значення з декількох датчиків в єдиний віртуальний багатоспрямований лідар, що спрощує майбутні розрахунки.

## 2.4 Моделювання та картографування навколишнього середовища

Як уже було встановлено раніше, замкнутий простір розглядається як такий, що може бути представлений у вигляді двовимірної площини. В подальшій реалізації середовище приймається як дискретна координатна площина з фіксованим кроком, розміри якої є достатніми для охоплення всього простору між двома найбільш віддаленими точками без врахування перешкод. Для такої моделі можна визначити функцію, що діятиме з площини у множину, та яка буде ставити у відповідність кожній координатній парі певний елемент множини, залежно від відомостей про конкретну локацію.  Такий підхід має низку переваг: зручність репрезентації об'єктів та визначення їх меж шляхом проекції на площину, наявність великої кількості ефективних алгоритмів, розроблених спеціально для таких моделей, а також відносна простота математичних обчислень. Серед недоліків можна відзначити зростання складності обробки, оскільки більшість алгоритмів на таких моделях за своєю складністю є квадратичними, обсяг обчислень швидко збільшується при розширенні розміру карти.

Для ефективного нанесення даних у форматі, описаному в попередньому пункті, доцільно скористатися алгоритмом, що дозволяє проектувати різноспрямовані промені в дискретну систему координат. У цьому випадку може бути використаний алгоритм Брезенхема, який широко застосовується для наближеного відображення прямих у комп’ютерній графіці, з огляду на подібність піксельної структури дисплея до обраної моделі простору. Базова

ідея алгоритму полягає у покроковому русі між двома точками $(x_0; y_0)$ та $(x_1; y_1)$ по осі абсцис, із розрахунку похибки, яка визначає необхідність зміщення по осі ординат. Початкове значення похибки задається як $2 \times |y_0 - y_1| - |x_0 - x_1|$. При кожному кроці по осі абсцис похибка зменшується на $2 \times |y_0 - y_1|$. Якщо обраховане значення стає не від'ємним, то виконується зміщення по осі ординат, після чого похибка збільшується на $2 \times |y_0 - y_1| - |x_0 - x_1|$. Крок зміщення по обох осях визначається залежно від кута нахилу прямої [6, с. 26-27]. Недоліком цього алгоритму є його обмеження на обробку ліній з кутом нахилу не більше одного октета, проте, це легко компенсується заміною схеми руху на протилежну у випадку якщо кут нахилу прямої завеликий.

Під час картографування може виникнути проблема, пов'язана з нерозміченими ізольованими точками. Цілком реалістичною є ситуація, коли окрема точка або група точок залишаються невідомими після завершення побудови карти, зокрема через неточності обчислень або обмеження в роботі сенсорів. Для вирішення цієї проблеми пропонується простий алгоритм: якщо кількість сусідніх точок навколо нерозміченої перевищує встановлений поріг і ці точки помічені як вільний простір, то відповідну точку також можна вважати вільною.

Аналогічна ситуація може виникати й під час розмітки перешкод: в їх структурі можуть з'являтися малі розриви, які не є суттєвими з практичної точки зору. Для їх усунення вводиться ще один алгоритм: якщо точка позначена як перешкода, система перевіряє наявність іншої точки з таким самим станом у межах певного околу та з'єднує їх. При цьому пошук здійснюється виключно вздовж однієї з координатних осей, що дозволяє уникнути заокруглення кутів і зберегти прямолінійність меж перешкод.

## 2.5 Алгоритми навігації в середовищі з перешкодами

Перебуваючи в частково або повністю розміченому середовищі, робот повинен мати змогу встановлювати ціль та досягати її. У попередніх пунктах було розглянуто способи подання даних і їх обробки для побудови карти середовища, тож тепер, спираючись на вже викладене, можна перейти до створення системи, відповідальної за навігацію.

Модель PID-регуляторів, як зазначалося раніше, самостійно не здатна приймати рішення чи визначати оптимальну траєкторію руху, тому потребує зовнішнього спрямування. Попри те, що відомі алгоритми побудови маршруту, такі як алгоритм Дейкстри або алгоритм A*, демонструють квадратичне зростання складності із розширенням карти, повністю відмовитися від застосування методів планування руху неможливо. Натомість доцільним є оптимізація їх використання шляхом обмеження обсягу оброблюваних даних на кожній ітерації.

Система, що проектується в межах даної роботи, передбачає здатність автономного пересування без суворого маршруту, однак складна структура середовища може зумовити потребу у створенні хоча б мінімального набору проміжних точок для підвищення надійності руху.

На етапі картографування середовища запропоновано використання алгоритму пошуку в ширину (BFS). Цей алгоритм є одним із базових методів обходу графів і характеризується рівномірним заглибленням у граф із фіксацією пройдених вершин для уникнення зациклення [7, с. 594–597]. Оскільки карта території в межах даної роботи подається у вигляді дискретної координатної площини, кожну її точку можна розглядати як вершину графа, а сусідні точки — як суміжні вершини. Таким чином, застосування BFS дозволяє рівномірно досліджувати навколишній простір і зменшувати обсяг обчислень за рахунок локалізованого пошуку.

На наступному етапі розглянемо побудову навігації по повністю розміченій карті замкнутого середовища. Необхідно обрати алгоритм, який дозволить сформувати маршрут до цільової точки, розбитий на декілька ключових контрольних точок, без надмірної деталізації шляху. Такий підхід обумовлений особливостями навігаційної системи, яка базується на каскаді PID-регуляторів: надто щільне розташування контрольних точок може призвести до виникнення небажаних маневрів під час слідування маршрутом. Крім того, слід уникати побудови маршрутів, які проходять у безпосередній близькості до перешкод, оскільки це може спричинити конфлікт між алгоритмами слідування за маршрутом та ухилення від перешкод. З огляду на ці вимоги, у межах даної роботи розглядається застосування алгоритму швидкоростучих випадкових дерев (Rapidly-Exploring Random Trees, RRT). Суть роботи алгоритму RRT полягає у побудові дерева на карті середовища: коренева вершина розташовується у стартовій точці, а нові вершини додаються у випадкових позиціях карти та приєднуються до найближчої існуючої вершини за умови наявності прямого незаблокованого шляху між ними. Завершення роботи алгоритму настає при додаванні вершини в районі цільової точки [8, с. 13]. Попри стохастичний характер побудови, що не гарантує оптимальності маршруту, алгоритм RRT має суттєві переваги: він дозволяє швидко знаходити прохідний шлях без повного алгоритмічного обходу карти та без потреби обчислення складних метрик. Реалізація алгоритму потребує наявності процедури перевірки можливості прокладання цілісного відрізку між вершинами, однак вимоги до точності такої перевірки можуть бути не строгими: недостатньо великі для розпізнання перешкоди можуть бути обійдені системою на етапі руху.

## 2.6 Застосування PID-регуляторів для руху автономної системи

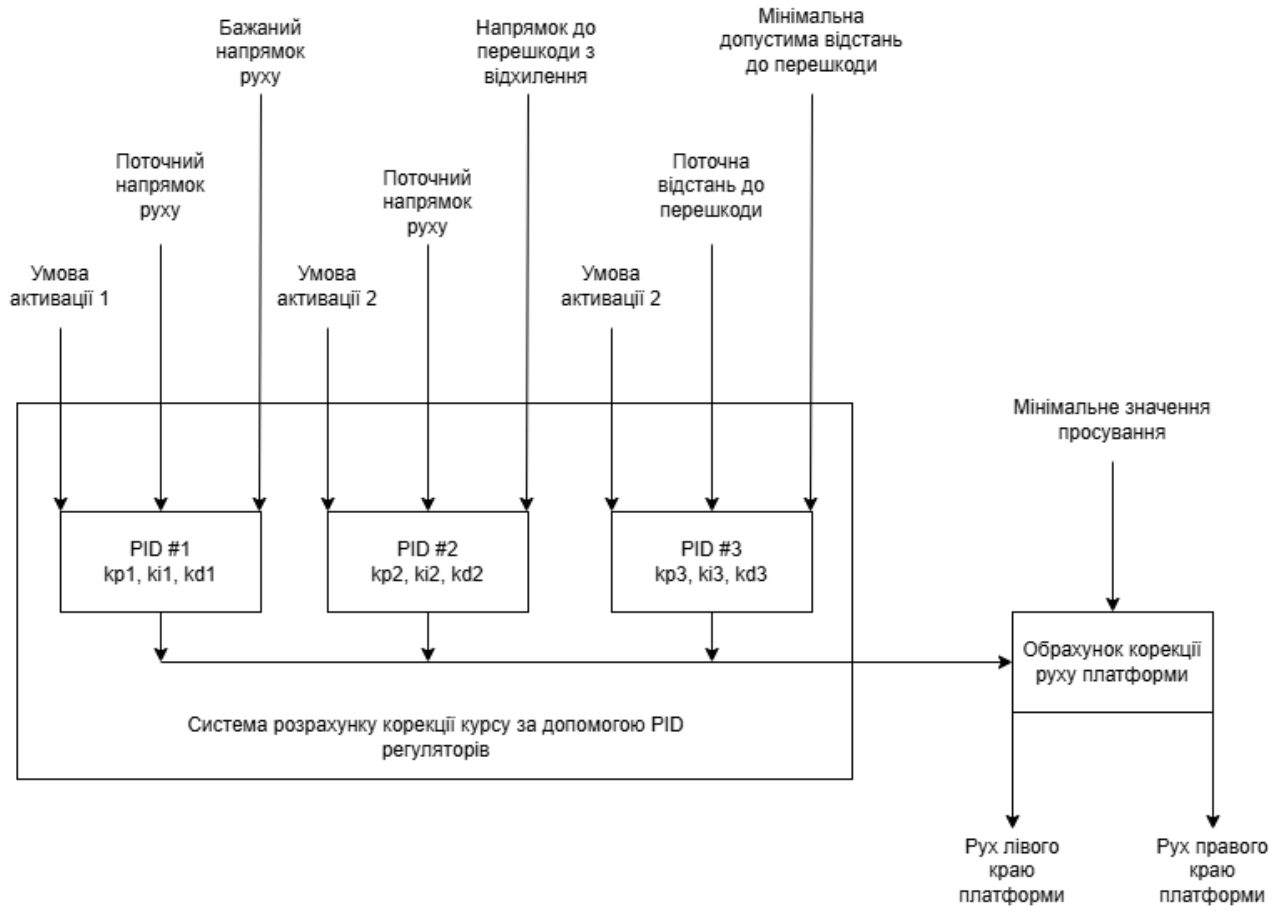
Центральною частиною дослідження є розробка системи руху автономної платформи з використанням каскаду PID-регуляторів. Як зазначалося раніше, PID-регулятор здійснює контроль одного змінного параметра. Відтак, для забезпечення руху до заданої цілі в умовах наявності перешкод необхідно побудувати каскад із декількох регуляторів. У даній роботі запропоновано рішення на основі трьох PID-регуляторів.

Перший регулятор відповідає за рух у напрямку до цілі. Бажаним значенням для нього є кут напряму на ціль, тоді як регульованою змінною є кут поточного напряму руху робота. Варто зауважити, що цей регулятор працюватиме не постійно: у критичних ситуаціях для запобігання зіткненням його буде вимкнено.

Другий регулятор забезпечує обхід перешкод. Він активується лише за наявності перешкод у безпосередній близькості до платформи. У такому випадку бажаним значенням є кут, що відповідає відхиленню на $\frac{\pi}{2}$ радіан від напряму, перпендикулярного до перешкоди. Регульованою змінною є кут поточного напряму руху. Завданням другого регулятора є підтримка руху вздовж межі перешкоди та запобігання прямого зіткнення.

Третій регулятор працює у парі з другим та виконує контроль відстані до перешкоди. Головною проблемою другого регулятора є неможливість самостійно підтримувати безпечну дистанцію, особливо у складних конфігураціях навколишнього середовища або під впливом інших регуляторів. Тому третій PID здійснює моніторинг поточної відстані до перешкоди та коригує траєкторію для підтримання мінімальної безпечної дистанції. Бажаним значенням є мінімально допустима відстань до перешкоди, тоді як контрольованою змінною — поточна відстань.

Усі три регулятори здійснюють вплив на рух шляхом керування швидкістю руху лівої та правої сторін платформи. Щоб забезпечити поступальний рух навіть за відсутності активної корекції з боку регуляторів, до керуючих сигналів додається певне мінімальне базове значення швидкості.




Діаграма 1: принципова схема системи корекції курсу

На Діаграмі 1 наведено принципову схему запропонованої системи керування. Як видно з діаграми, вихідні сигнали трьох регуляторів підсумовуються у загальне коригуюче значення, яке визначає подальший рух платформи. Варто звернути увагу, що другий та третій PID мають однакову умову активації, адже вони обидва працюють лише у випадку виявлення перешкоди. Кожен з регуляторів має власні параметри налаштування, що дозволяє

оптимізувати їхню взаємодію для підвищення ефективності роботи всієї системи.

Дані про поточний напрямок руху передаються у вигляді пари чисел {a, b}, де кожне число має значення в діапазоні [-1; 1]. Значення -1 відповідає максимальній швидкості руху відповідної сторони платформи у зворотному напрямку, 0 – відсутності руху, а 1 – максимальній швидкості руху вперед. Допускаються проміжні значення, що відповідають руху в заданому напрямку з пропорційно меншою швидкістю.

## 2.7 Підсумки теоретичної частини

У межах другого розділу були розглянуті основні допущення та умовності, які необхідно враховувати в подальшій роботі для запобігання суперечностям та для додаткового пояснення принципів і термінів, закладених у дослідження.

Було визначено чітку структуру даних, що подаються на вхід до API, а також структуру даних, які API генерує для керування рухом автономної системи. Окрім цього, було розглянуто основні алгоритми, використані при побудові API, та обґрунтовано їх застосування.

Також було здійснено огляд та пояснення математичного апарату PID-регулятора, а також визначено особливості його використання в контексті навігації автономної системи. На основі даного аналізу було запропоновано побудову каскадної системи, що складається з трьох PID-регуляторів, для забезпечення контролю за рухом з урахуванням наявності перешкод у середовищі.

# 3. ПРОГРАМНА РЕАЛІЗАЦІЯ

## 3.1 Вибір середовища моделювання та засобів реалізації

Обговоривши теоретичну частину задачі, переходимо до її практичної реалізації. Однак перед початком важливо обрати відповідні засоби для виконання цього завдання. Потрібно звернути увагу як на мову програмування, якою буде реалізовано API, так і на середовище, в якому буде здійснюватися тестування програмного забезпечення та проведення симуляцій для подальшого аналізу рішення.

Щодо вибору мови програмування, то було обрано мову C. Це компільована мова програмування зі стандартизованими та відкритими компіляторами, що дозволяє напряму працювати з пам’яттю пристрою, на якому виконується код. Хоча сучасні тенденції в робототехніці свідчать, що мова C не є основною для розробки роботизованих систем, вона активно використовується для написання ефективних та швидкодіючих бібліотек, що застосовуються у поєднанні з мовами більш вищого рівня [9, с. 259-260]. Отже, мова C є оптимальним вибором для реалізації цієї задачі завдяки її швидкості, ефективності та широкому застосуванню для виконання подібних проектів.

Симуляція роботи алгоритму є ключовою складовою для успішної розробки, відлагодження та аналізу програмного забезпечення. У рамках цієї роботи для виконання цього завдання було обрано середовище Webots. Це симулятор, спеціально розроблений для моделювання роботів та їх програмування з відкритим вихідним кодом. Webots дозволяє моделювати оточення, роботів та різноманітні їх датчики чи компоненти, а також надає програмні інтерфейси для взаємодії з ними. Webots підтримує кілька мов програмування, зокрема мову C, і має вбудовані компілятори для зручності розробки [10, розділи Foreword, Compiling with the Source Code Editor, Assets].

### 3.2 Структури для роботи з API

Як було встановлено раніше у даній роботі, API приймає на вхід стандартизовані дані певного вигляду. Для їх обробки було реалізовано дві основні структури даних (див. Додаток api.c).

struct Laser – зберігає інформацію про один скануючий промінь лідара і містить кут відносно напрямку руху і відстань до виявленого об’єкта. У випадку, якщо перешкоду не виявлено, відстань встановлюється більшою за максимальне поле зору лідара.

struct MotorPair – використовується для формування відповіді API на виклик функції обчислення напрямку руху. Структура містить керуючі значення для лівої та правої сторін робота відповідно, представлені у вигляді, описаному раніше.

### 3.3 Внутрішні структури та змінні

В даному пункті розглянемо структури і змінні які використовуються при внутрішніх обчисленнях API (див. Додаток api.c, global_types_def.c)

enum MapStat – тип даних enum що використовується для полегшення роботи з картою за рахунок призначення імен для різних індексів стану комірок карти.

struct PID – Структура потрібна для реалізації та обчислення PID-регуляторів, зберігає 3 головні коефіцієнти, попередню похибку та накопичену похибку.

struct PID pid – PID відповідальний за спрямування робота до точки призначення.

struct PID pid_obstacle – PID відповідальний за контролем кута відхилення від виявленої перешкоди.

struct PID pid_distance – PID відповідальний за дотримання дистанції до виявленої перешкоди.

struct Position – структура метою якої є зберігання позиції робота в просторі, містить в собі координати та кут напрямку руху.

enum RobotState – тип даних enum призначений для полегшення роботи зі змінною поточного стану алгоритму, за рахунок призначення імен для різних індексів, що відображають поточний етап.

enum RobotState  global_state – змінна що відображає поточний стан роботи API.

struct Coord queue[FIELD_SIZE*FIELD_SIZE] – масив що зберігає чергу яка генерується при роботі алгоритму BFS.

struct Position global_position – змінна що зберігає поточне розташування роботав просторі. Координати вносяться в неї тільки за допомогою функції, що записує їх певним чином щоб за протреби методом простого округлення мати змогу звернутися до відповідної клітинки на карті, яку зберігає API.

enum MapStat map[FIELD_SIZE][FIELD_SIZE] – двовимірний масив що містить в собі карту, складену за допомогою роботи алгоритму картографування або завантажену з постійної пам’яті.

int possibility_map[FIELD_SIZE][FIELD_SIZE] – масив що застосовується при роботі алгоритму пошуку в ширину, може містити як обрахунок метрик так і дані про обхід комірки, залежно від модифікації алгоритму.

struct Coord – структура, що зберігає координати на карті.

const struct Coord EMPTY_COORD ={-1, -1}; - константа, яка є аналогом null для структури Coord, якщо змінна є рівною цій константі, то це означає що вона не визначена.

struct Coord target – змінна, що містить поточну ціль алгоритму, коли він перебуває на етапі руху по уже розміченій карті.

struct Node – структура призначена для побудови дерева в процесі роботи алгоритму RRT. Зберігає координати вершина на карті та індекс батьківської

вершини. У випадку, якщо поточна вершина є батьківською то зберігається власний індекс.

struct Node rrt_tree[RRT_CUTOFF_LEN]; - масив, призначений для збереження дерева згенерованого в процесі роботи алгоритму RRT.

int tree_len - зберігає кількість згенерованих вершин дерева.

int current_route_node – використовується в процесі слідування побудованого маршруту, зберігає індекс поточної вершини дерева.

bool found_route – змінна, що містить інформацію чи був знайдений шлях до цілі алгоритмом RRT.

void save_map(const char *filename, int rows, int cols, enum MapStat array[rows][cols]) – функція збереження карти у постійну пам’ять, може бути перевизначена залежно від архітектури.

bool load_map(const char *filename, int rows, int cols, enum MapStat array[rows][cols]) – функція завантаження карти з постійної пам’яті, повертає інформацію про успішність операції. Може бути перевизначена залежно від архітектури.

### 3.4 Функції ініціалізації

Функції ініціалізації – це функції які викликаються один раз на початку програми. Їх задача встановити початкові значення необхідних змінних та підготувати API до подальшої роботи. (див. Додаток api.c, global_types_def.c)

void init(bool use_saved_map) – дана функція має бути викликана першою. Вхідний параметер визначає чи буде відбуватися картографування місцевості, чи буде використана збережена карта. Якщо збережену карту зчитати не вийде, то

автоматично почнеться картографування. Варто зазначити, що при завантаженні карти робот автоматично вважатиме що знаходиться в точці, звідки починалося картографування, тому якщо позиція робота з нею не збігається, варто врахувати це при виклику set_new_position та set_new_angle (див. Пункт 3.5) для коректної роботи API.

void add_forbidden_zone(int x1, int y1, int x2, int y2) – Дана функція викликається після init та може бути викликана декілька разів. Результатом її виконання є створення прямокутної віртуальної перешкодами між двома точками. Слід зазначити, що точки в даному випадку задаються відносно стартової позиції робота, що має координати {0;0}. Додавати перешкоди можна необмежену кількість разів.

### 3.5 Функції головного циклу програми

Робота API побудована таким чином, що деякі її функції мають постійно викликатися з головного циклу програмного забезпечення робота, для внесення актуальних даних та здійснення поправок напрямку руху. Наведені в цьому пункті функції розраховані саме на багаторазовий повторюваний виклик. (див. Додаток api.c).

void set_new_angle(float angle) – функція призначення для передавання актуального напрямку руху до API і має викликатися якомога частіше.

float get_current_angle() – ця функція є допоміжною і повертає кут який наразі збережений в API, є допоміжною і може викликатися лише у разі виникнення потреби в цих даних.

void set_new_position(float x, float y) – ця функція використовується для оновлення позиції робота в просторі. Як і функцію оновлення кута напрямку руху її слід викликати якомога частіше.

struct MotorPair get_directions(struct Laser *beams, int length, float time_step) – ця функція є ключовою для отримання вказівок стосовно руху робота. На вхід вона приймає актуальні та стандартизовані дані віртуального лідару, що, як було зазначено раніше, можуть бути обробленими даними з одного або декількох датчиків. Решта аргументів це кількість променів віртуального лідару та час в секундах що пройшов з моменту останнього виклику. Вимірювання часу є необхідним, адже на нього спирається робота PID регуляторів.

void set_target(int x, int y) – ця функція не має постійно викликатися, на відміну від уже перерахованих. Вона може бути застосована у внутрішній логіці робота і викликатися коли йому потрібно досягти конкретної точки карти. Якщо API знаходиться в процесі картографування – рух почнеться в момент повного розмічення території. У іншому випадку – рух почнеться відразу після виклику функції. Як і у випадку виклику функції add_forbidden_zone, координати пункту призначення вказуються відносно стартового положення робота, координатами якого є {0;0}.

### 3.6 Реалізація PID та каскаду регуляторів

Реалізація та застосування PID регуляторів є центральною частиною даної роботи, тому в цьому пункті програмний код, відповідальний за цей функціонал, буде розглянуто більш детально. (див. Додаток api.c).

```
float calculate_pid(struct PID* p, float error, float dt)
{
  float res=error*p->kp;
  res+=(error-p->prev_error)/dt*p->kd;
  p->integral_sum+=error*dt;
  if(fabs(p->integral_sum)>=PID_OVERFLOW_ABS) p->integral_sum=0;
  p->prev_error=error;
  res+=p->integral_sum*p->ki;
```

```
 return clamp(res, -1, 1);
}
```

Для коректної роботи calculate_pid потрібно передати посилання на структуру, що містить конфігурації регулятора, поточне значення похибки та часовий проміжок в секундах, що пройшов з часу останнього виклику API. Перший аргумент обов’язково має передаватися за посиланням, це забезпечує оновлення внутрішніх змінних PID. Обрахунок пропорційної складової є простим, і не потребує додаткових пояснень. Інші ж складові, слід розглянути детальніше: через дискретну природу даних, в даному випадку використовуються наближення диференціальної та інтегральної складових. Швидкість приросту похибки обчислюється за допомогою збереження попереднього її значення, та обрахунку різниці з поточною. Для наближення інтегралу використовується сума усіх попередніх похибок, домножених на часовий проміжок між розрахунками. Варто зазначити, що дана функція призначена для зведення похибки до 0, таким чином дозволяючи роботу з більш складним математичним апаратом її обчислення, що, на приклад, може бути корисним при роботі з градусними мірами, значення яких є циклічними.

```
struct MotorPair drive(struct Position pos, float time_step, struct Coord
current_target)
{
 if(are_coords_same(current_target, EMPTY_COORD)) //condition of stop
 {
  if(global_state==MAPPING)
  {
   global_state=DRIVING;
   save_map("map.txt", FIELD_SIZE, FIELD_SIZE, map);

  }else global_state=STOP;
  return (struct MotorPair){0,0};
 }
```

```
 //calculating error of angle towards target
 float angle=-atan2(pos.y-current_target.j, current_target.i-pos.x);
 float angle_error=absp(-angle+pos.angle);
 struct Laser obstacle=check_direction_safety_on_map(pos, 2, (float[2][2])
{{3.0*Pi/2.0, P2}, {0, Pi/2}});
 if(angle_error>Pi)angle_error=angle_error-P2;
 float correction=calculate_pid(&pid, angle_error, time_step);

 //if no obstacle detected using only first pid correction
 if(obstacle.dist<0)
 {
  return (struct MotorPair){clamp(0.5+correction,-1,1), clamp(0.5-correction, -1,
1)};
 }

 //calculating error of angle towards obstacle
 float obstacle_error=0;
 if(obstacle.angle<Pi)obstacle_error=Pi-obstacle.angle;
 else obstacle_error=3.0*Pi/2.0-obstacle.angle;
 obstacle_error=clamp(obstacle_error, -Pi/2, Pi/2);

 float obstacle_correction=-calculate_pid(&pid_obstacle, obstacle_error,  time_step);

 //calculating distance error to the obstacle
 //clamp is used to ban robot for going closer but allowing to drive off the wall
 float distance_correction=0;
 if(obstacle.angle<Pi)distance_correction=calculate_pid(&pid_distance, clamp(-
(SAFE_ZONE-obstacle.dist)/SAFE_ZONE, -1, 0), time_step);
 else distance_correction=calculate_pid(&pid_distance, clamp((SAFE_ZONE-
obstacle.dist)/SAFE_ZONE,0, 1), time_step);

 //getting total sum of all corrections and
 float total_correction;
 if(obstacle.dist<=MINIMUM_SAFE_ZONE) return (struct
MotorPair){clamp(0.2+obstacle_correction+distance_correction*0.5,-1,1),
clamp(0.2-obstacle_correction-distance_correction*0.5, -1, 1)};
```

```
  else total_correction=clamp(correction+(obstacle_correction+distance_correction),
-1, 1);
  return (struct MotorPair){clamp(0.5+total_correction,-1,1), clamp(0.5-
total_correction, -1, 1)};
}
```

Функція drive відповідає за розрахунок вказівок руху, та реалізовує описаний в даній роботі каскад з трьох PID регуляторів для цього. На вхід функції подаються наступні аргументи: поточне положення, часовий проміжок з останнього виклику в секундах, координата точки призначення. Перш за все, проводиться перевірка умова зупинки: точка призначення є невизначеною. У цьому випадку залежно від стану API приймається рішення або про повну зупинку, або про збереження поточної карти і перехід в режим навігації. Якщо ж умова не справджується, то відбувається розрахунок корекцій руху. За замовчуванням, для руху обом сторонам робота дається команда руху вперед на половині від максимальної швидкості, що дозволяє скорегувати її як у верхню сторону, так і в нижню, дозволяючи давати вказівку про рух назад. Після знаходження напрямку до найближчої перешкоди, пошук якої ведеться в діапазонах $[2\pi/3; 2\pi)$ та $[0; \pi/2]$ для ігнорування тих, що знаходяться позаду, починається робота каскаду. Призначення кожного регулятора було описано в теоретичній частині роботи. Перед знаходженням коригуючого значення відбувається обрахунок похибки. Слід звернути увагу, що при обрахунку третього регулятора похибка обмежується, щоб не завадити роботу віддалятися від перешкоди. Існує 3 головні сценарії застосування PID:

1. Якщо перешкод поблизу не виявлено: працює тільки перший регулятор.
2. Якщо перешкода виявлена, але не на критичній відстані: працюють усі 3 регулятори, відбувається рух до точки призначення з ухиленням.
3. Якщо перешкода критично близько: працюють лише другий та третій регулятори, щоб запобігти зіткненню.

### 3.7 Допоміжні функції

Окрім уже згаданих основних функцій, API також містить низку допоміжних функцій, які виконують обчислення, необхідні для коректної роботи основних алгоритмів. У цьому пункті наведено короткий опис кожної з них. (див. Додаток api.c).

void build_rrt(struct Position pos, struct Coord t) – функція містить алгоритм RRT, що будує випадкове дерево на карті оточення починаючи з пункту призначення. Робота алгоритму вважається завершеною, якщо в місце поточного розташування робота можна поставити нову вершину і приєднати до будь-якої з уже згенерованих.

struct Coord get_to_unknown(int starti, int startj) – дана функція знаходить найближчу точку, що ще не була розмічена, до поточної позиції робота за рахунок алгоритму BFS, що рухається по ділянках карти що розмічені як порожні, для уникнення знаходження недосяжних точок.

void place_beam(struct Laser beam, struct Position pos) – функція обробки даних з лідару, де за допомогою алгоритму Брезенхема на карті оточення відображається шлях кожного променя. Варто зазначити, що для уникнення спотворення меж уже розмічених перешкод через неточності лідара, їх перезапис відбувається лише у випадку безпосередньої близькості робота до них.

float absp(float angle) – простий медот до дозволяє отримати аналог міри кута в діапазоні [0; 2π) рад.

float distance_sqrt(float x1, float y1, float x2, float y2) – функція що обчислює відстань між двома точками на карті.

bool in_bounds_single(int i), bool in_bounds(int i, int j), bool in_bounds_coord(struct Coord c) – три функції що перевіряють чи занходиться координата або пара координат в межах карти.

bool are_coords_same(struct Coord a, struct Coord b) – функція що порівнює дві точки на предмет їх збігу.

void clear_possibility_map() – функція очищення даних в змінній possibility_map.

float clamp(float value, float min, float max) – функція обмеження, призначена повертати значення параметру value, якщо воно знаходиться в межах (min; max). Якщо дане значення більше або менше, то функція повертає параметр max або min відповідно.

void gap_compensation(int i, int j) – функція компенсації проміжків. Якщо для даної точки, яка помічена як перешкода, вдається знайти сусіда по горизонталі чи вертикалі в межах певної відстані, що теж помічений як перешкода, то усі точки між ними заповнюються відповідним значенням для закриття прогалини.

bool empty_spots_compensation(int i, int j) – функція що дозволяє заповнювати окремі нерозмічені ділянки. Вона перевіряє вісім сусідів навколо даної точки, якщо навколо неї 6 або більше порожніх комірок то вона набуває відповідного значення, повертаючи інформацію про результат перевірки.

void push_node(int i, int j, int parent_id) – функція, що додає нову вершину до дерева RRT.

int find_closest_node(int i, int j) – функція, що проходить списком вершин дерева RRT і повертає ту, що знаходиться найближче до даної.

bool is_free(int x1, int y1, int x2, int y2) – функція, що бігло перевіряє відрізок між двома вершинами і повертає інформацію чи знаходиться на ньому хоч одна перешкода. Відслідковування відрізку не є дуже точним, на відміну від функції обробки даних лідара, але в межах навігації за допомогою RRT це не є критичним.

struct Laser check_direction_safety_on_map(struct Position pos, int ranges_len, float ranges[ranges_len][2]) – функція, що проводить пошук перешкоди на карті

поблизу поточної позиції робота. На вхід функція отримує діапазони допустимих значень кута до перешкоди відносно напрямку руху. У відповідь на виклик повертається промінь, що вказує відстань та кут до знайденої перешкоди.

### 3.8 Константи конфігурацій

Важливою частиною API є набір констант, що безпосередньо впливають на роботу усіх функцій та алгоритмів. При розгортанні коду, обов'язково слід приділити увагу встановленню правильних значень для системи, на якій буде виконуватися програмне забезпечення.

FIELD_SIZE – цілочисельне значення що визначає розмір карти, задається в сантиметрах.

LIDAR_DIST – максимальна довжина променя лідару в сантиметрах, що подається на вхід до API.

SAFE_ZONE – максимальна відстань в сантиметрах від робота до перешкоди, при якій перешкода враховується при здійсненні руху.

MINIMUM_SAFE_ZONE – максимальна відстань в сантиметрах від робота до перешкоди, при якій наближення вважається критичним.

MAX_GAP_SIZE – максимальний розмір прогалини в перешкоді яку можна заповнити алгоритмічно, задається в сантиметрах.

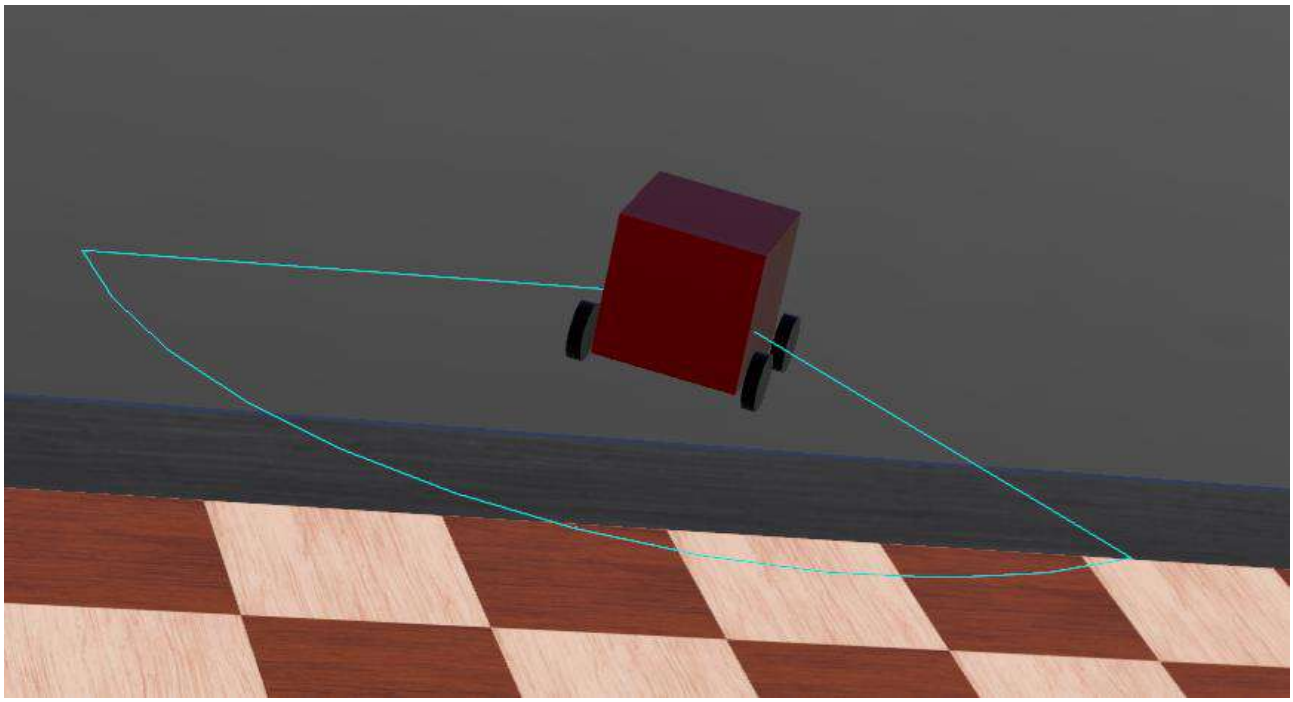
RRT_CUTOFF_LEN – максимальна кількість згенерованих вершин при роботі алгоритму RRT, при досягненні якої вважається що маршрут до цілі не можливо побудувати.

PID_OVERFLOW_ABS – абсолютне значення, при якому вважається, що змінна накопиченої похибки PID переповнена і її потрібно встановити на 0.

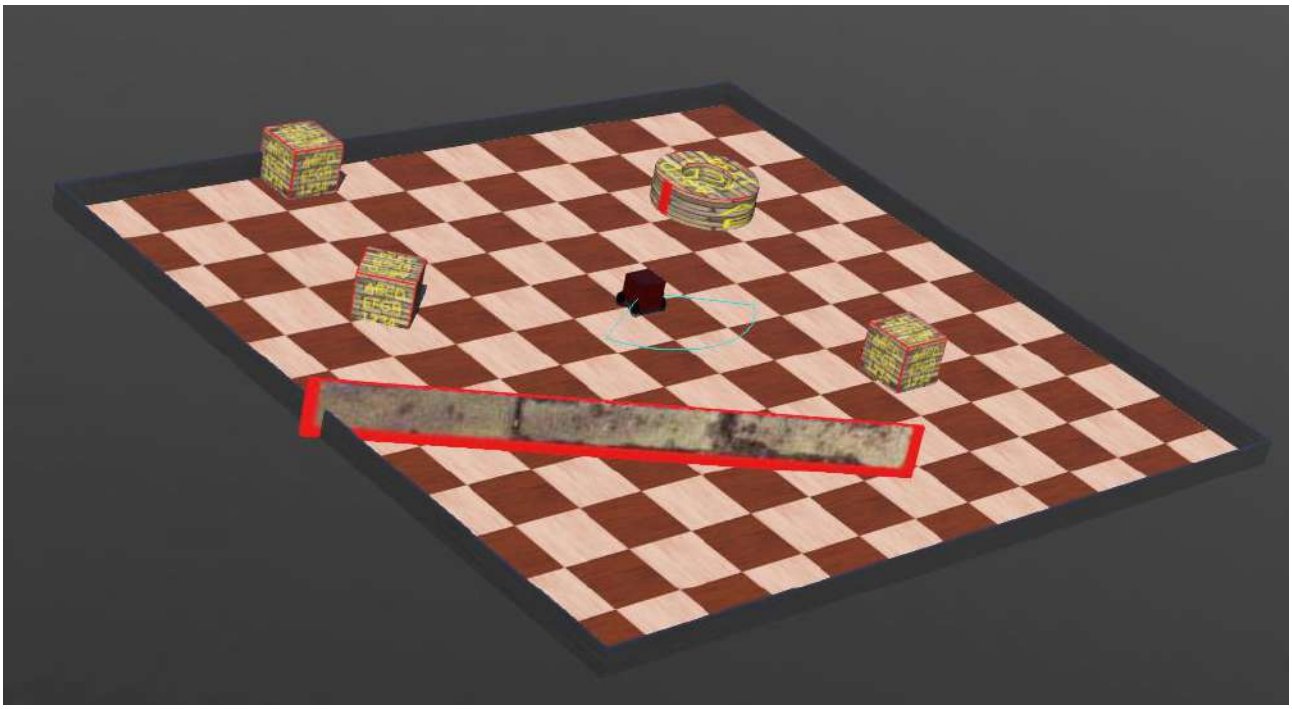
# 4. РЕЗУЛЬТАТИ ТА ЇХ ОБГОВОРЕННЯ

## 4.1 Практичне тестування у віртуальному середовищі

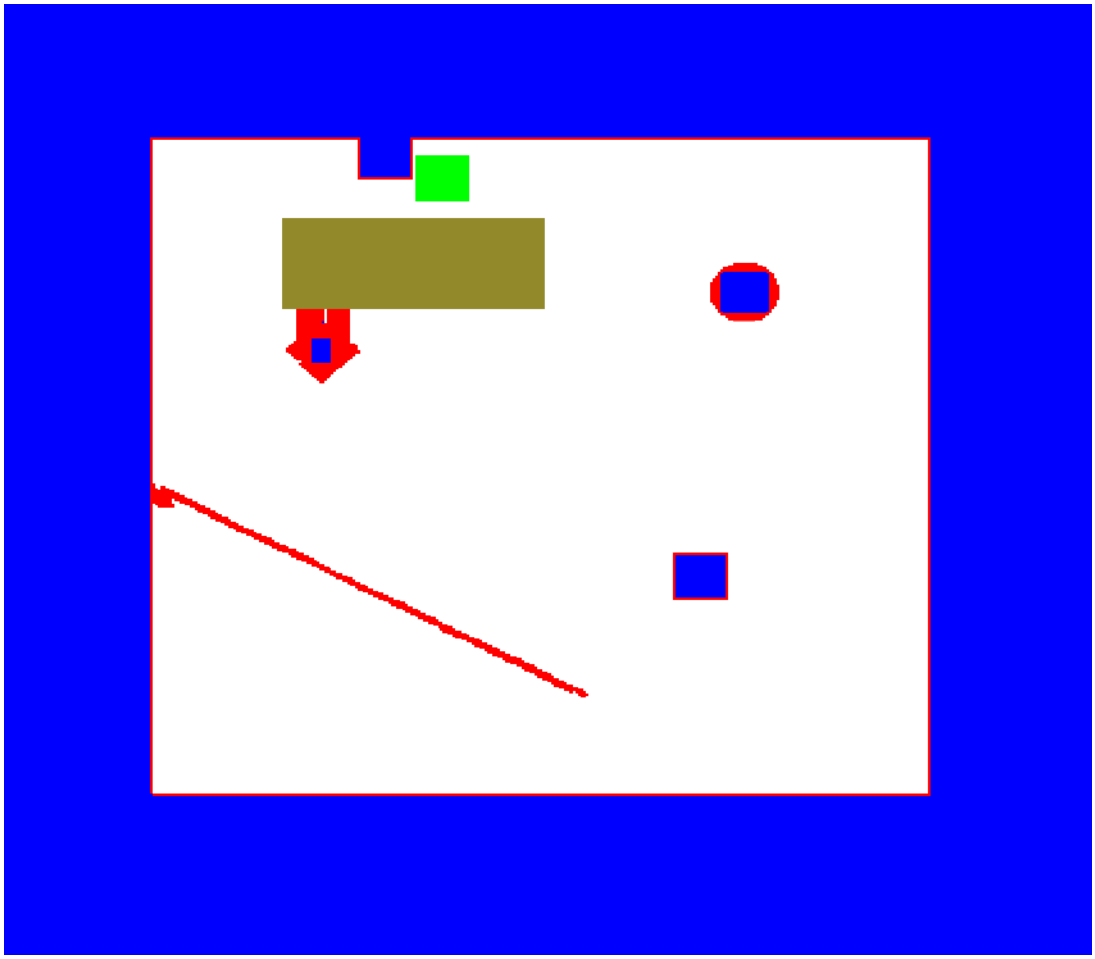
Для проведення симуляції на платформі Webots було створено макет простого робота, оснащеного такими датчиками: GPS, компасом та лідаром. Усі вони розміщені в центрі робота та забезпечують можливість відстеження його актуального положення й напрямку руху. Така конфігурація була обрана для полегшення приведення даних до раніше визначених форматів і мінімізації ризику помилок на стороні програмного забезпечення. Макет має чотири колеса та може одночасно задіювати обидва розташовані на вказаній стороні.



Малюнок 1



Малюнок 2



Малюнок 3

На Малюнку 1 зображено структуру робота в симуляторі Webots, де синіми лініями позначено поле зору лідару. Оточення в даному прикладі було змодельовано на полі розмірами 3 на 3 метри. Окремо в симуляторі було розгорнуто засіб відображення стану карти, яку будує API, за допомогою вбудованого в Webots пристосування Display. Статичні перешкоди були реалізовані у вигляді об'єктів різних форм, розставлених у різних конфігураціях для перевірки здатності системи обробляти приміщення з різними видами та положеннями перешкод. Приклад обстановки поля можна побачити на Малюнку 2.

Для перевірки роботи функціоналу обминання віртуальних перешкод та навігації по розміченій місцевості було додано додаткові різні конфігурації. На Малюнку 3 представлено приклад завершеної карти для обстановки, зображеної на Малюнку 2. На карті кольорами позначено:

1. червоним — виявлені фізичні перешкоди.
2. білим — пусті ділянки.
3. синім — не розмічені ділянки.
4. брудно-жовтим — віртуальні перешкоди, задані на початку роботи алгоритму.
5. зеленим — область навколо точки призначення, якої має досягти робот після завершення картографування. Варто зазначити що окіл в даному випадку є лише інструментом покращення видимості для користувача і не збігається з реальним.

Також, увагу було приділено симулюванню ситуації з уже готовою картою замкнутого простору та перевірці спроможності робота досягти заданого положення.

## 4.2 Аналіз результатів симуляцій: досягнення та недоліки

Проаналізувавши роботу API за різних конфігурацій, можна виокремити такі позитивні аспекти: за умови правильного налаштування коефіцієнтів PID-каскаду та супутніх параметрів робота у більшості випадків успішно здійснював картографування місцевості та досягав заданої цілі. Алгоритм пошуку нерозмічених ділянок дозволяв роботу не лише сканувати оточення прямолінійним рухом, а й виконувати повороти, що забезпечувало отримання більш повної інформації про навколишню область. Це особливо важливо при використанні датчиків із обмеженим полем зору. За умови надходження валідних вхідних даних, побудована карта виявилася доволі точною, за винятком окремих місць, де дрібні проміжки були алгоритмічно компенсовані. Рух за готовою картою також демонстрував прийнятні результати: алгоритм RRT у більшості випадків забезпечував валідні маршрути, що запобігало втраті орієнтації, хоча ефективність прокладених шляхів залишала бажати кращого. Робот успішно оминав як фізичні, так і віртуальні перешкоди, не допускаючи зіткнень.

Щодо недоліків, перш за все варто відзначити необхідність частого переналаштування коефіцієнтів PID-регуляторів. Налаштування, які працювали у середовищах із подібними характеристиками, виявлялися неефективними при суттєвій зміні умов, що іноді призводило до тривалого блукання робота в окремих ділянках без прогресу. Баланс між регуляторами виявився вкрай чутливим: за надто високих коефіцієнтів регулятора, що забезпечував рух до поточної цілі, робот затримувався перед перешкодами, безрезультатно обертаючись на місці замість того, щоб просто оминути заблоковану ділянку, тоді як при занадто м’яких налаштуваннях він міг рухатись колами навколо нерозмічених зон, не спроможний достатньо змінити орієнтацію для сканування їх лідаром. Очікувано, алгоритм RRT іноді генерував неоптимальні маршрути, однак траплялися випадки, коли шлях був настільки нераціональним, що робот

об’їжджав майже всю мапу, щоб досягти цілі, розташованої неподалік від поточної позиції. Алгоритм нанесення даних на карту також виявив недоліки: хоча розмітка окремих точок перешкод виконувалась коректно, алгоритм компенсації проміжків працював ненадійно для об’єктів, розташованих під кутом до умовної горизонтальної осі карти. Попри спроби обмежити пошук заповнення лише горизонтальними й вертикальними напрямками, здійсненими під час розробки, проблеми залишалися на похилих перешкодах, що добре видно на прикладі, наведеному на Малюнку 3.

## 4.3 Аналіз недоліків, пропозиції стосовно їх усунення та подальшого розвитку API

На жаль, слід визнати, що проблема постійної потреби в корекції коефіцієнтів PID-регуляторів навряд чи може бути легко усунута. Цей тип регулятора є вкрай чутливим до характеристик системи, в якій він функціонує, а використання каскадної схеми лише підсилює цю залежність. Відповідно, різні оточення та модифікації роботів, імовірно, вимагатимуть індивідуального налаштування параметрів.

Водночас проблему блукання робота в окремих ділянках можна потенційно вирішити шляхом впровадження механізму відстеження прогресу руху до заданої позиції. У випадку відсутності помітного прогресу має сенс використати алгоритм RRT або подібний для перенаправлення робота у правильному напрямку.

Алгоритм RRT може бути модифікований або замінений альтернативним підходом, що забезпечить побудову більш ефективних маршрутів у межах замкненого простору.

Попри недоліки, алгоритм компенсації проміжків у перешкодах не можна повністю відкинути: у разі його відсутності та через обмеження лідара при скануванні поверхонь під кутом до напрямку руху виникає значна кількість дрібних прогалин. Це призводить до того, що робот може намагатися пройти через них, витрачаючи надмірно багато часу та маневрів, або навіть зациклюватися навколо перешкод в намаганнях повністю їх розмітити. Наразі оптимального вирішення цієї проблеми не знайдено, однак на подальших етапах розвитку API її усунення є критично важливим.

У перспективі, після виправлення виявлених недоліків, API може бути розширене для обробки динамічних перешкод під час руху по вже розміченій карті, а також оптимізоване щодо використання постійної та оперативної пам’яті. Таким чином, після внесення відповідних доопрацювань, API має потенціал стати конкурентоспроможним рішенням для роботизованих систем.

# ВИСНОВКИ

В результаті проведеного дослідження було проаналізовано застосування PID-регуляторів у системах SLAM. Встановлено, що PID-регулятори зазвичай виконують допоміжну роль, забезпечуючи чіткий рух автономної системи по вже спланованій траєкторії. В ході роботи було розроблено API, що використовує каскад PID-регуляторів для картографування закритого простору та навігації в ньому.

Актуальність роботи полягала у дослідженні можливості використання PID-регулятора як основного інструменту для керування рухом до заданої точки та ухилення від перешкод. Хоча результати показали, що для ефективної роботи не обійтися без планувальника маршрутів, використання PID-регуляторів дозволяє зменшити необхідність в деталізації та чіткості побудованого шляху, зменшуючи навантаження на обчислювальні потужності.

Варто зазначити, що розроблене API не є ідеальним і має низку недоліків, виявлених під час симуляцій реальної роботизованої системи в невідомому оточенні. Однак більшість цих недоліків не стосуються основних принципів, закладених під час розробки, тому їх виправлення є нормальним етапом покращення роботи API. У майбутньому необхідно зосередитись на оптимізації коду, покращенні алгоритмів планування руху та виправленні алгоритму компенсації прогалин. Після вирішення цих проблем API матиме перспективу для впровадження додаткового функціоналу та розвитку.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


[1] Fahmizal, F., Pratikno, M., Isnianto, H., Mayub, A., Maghfiroh, H., Anugrah, P. Control and navigation of differential drive mobile robot with PID and Hector SLAM: Simulation and implementation. // *Jurnal Ilmiah Teknik Elektro Komputer dan Informatika*. 2024. Vol. 10. P. 594-607.

[2] Wang, H. UAV obstacle avoidance with PID control based on improved sparrow search algorithm. *4th International Conference on Signal Processing and Machine Learning*, 15-22.01.2024 / Chicago, IL, United States, 2024.

[3] Sabrina, M., Mellah, R., Fekik, A., Azar, A. Real-time fuzzy-PID for mobile robot control and vision-based obstacle avoidance. // *International Journal of Service Science, Management, Engineering, and Technology*. 2022. Vol. 13. P. 1-32.

[4] Hu H., Li Z. Research on intelligent car PID autonomous navigation system based on ROS and Lidar. // *Journal of Simulation*. – 2022. – Vol. 10, No. 2. P. 31–35.

[5] Zurich Instruments. Principles of PID controllers. // *White Paper*. 2023.

[6] Bresenham, J. E. Algorithm for computer control of a digital plotter. // *IBM Systems Journal*. 1965 Vol. 4. No. 1. P. 25-30.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.

[8] Karaman, S., Frazzoli, E. Sampling-based algorithms for optimal motion planning. // *International Journal of Robotic Research*. 2011. Vol. 30 No. 8. P. 846-894.

[9] Haviland, J., Corke, P. Robotics software: Past, present, and future. // *Annual Review of Control, Robotics, and Autonomous Systems*. 2024. Vol. 7, P. 253-283.

[10] Офіційна документація Webots. Cyberbotics Ltd. URL: Cyberbotics. (n.d.). *Guide foreword*. https://cyberbotics.com/doc/guide/foreword (Дата звернення: 27.04.2025).

# ДОДАТКИ

**global_types_def.c**

```
#pragma once
//global structure that defines a map cells status
enum MapStat {
    FULL=8,
    FORBIDDEN=6,
    EMPTY=4,
    UNDISCOVERED=0
  };
```

**save_manager.c**

```
#include <stdio.h>
#include "global_types_def.c"

//all following functions can be redefined considering a platform that the code is
running on

void save_map(const char *filename, int rows, int cols, enum MapStat
array[rows][cols]) {
    FILE *file = fopen(filename, "w");
    if (!file) {
        perror("Cannot open file to write");
        return;
    }

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            fprintf(file, "%d ", (int)array[i][j]);
        }
        fprintf(file, "\n");
```

```
    }

    fclose(file);
}

bool load_map(const char *filename, int rows, int cols, enum MapStat
array[rows][cols]) {
  FILE *file = fopen(filename, "r");
  if (!file) {
      perror("Cannot open file to read");
      return false;
  }
  int value;
  for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
          if (fscanf(file, "%d", &value) != 1) {

              printf("Error reading element [%d][%d]\n", i, j);
              fclose(file);
              return false;
          }
          array[i][j]=(enum MapStat) value;
      }
  }
  fclose(file);
  return false;
}
```

**api.c**

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include "save_manager.c"
#include "global_types_def.c"

#define FIELD_SIZE 801 //field has a shape of squarem it is it's size
```

```
#define CENTER FIELD_SIZE/2 //central coordinate of the field
#define LIDAR_DIST 40 // max distance of unified lidar input
#define SAFE_ZONE 30 //max distance at which the obstacle would be detected
#define MINIMUM_SAFE_ZONE 15 //max distance for the obstacle to be
considered dangerous
#define MAX_GAP_SIZE 20 //max size of the gap that will be covered by gap
compensation
#define P2 6.28318530718
#define Pi P2/2
#define RRT_CUTOFF_LEN 500 //max len of rrt at which the target will be
considered unreacheble
#define PID_OVERFLOW_ABS 1e9 //absolute value for pid accumulated error to be
considered overflown and set back to 0

struct PID{
  float kp, ki, kd;
  float prev_error;
  float integral_sum;
};

//pid parameters
struct PID pid={2.1, 0.03, 0.3, 0, 0}; //target following pid
struct PID pid_obstacle={1.5,0,0.6, 0, 0}; //wall angle pid
struct PID pid_distance={0.8,0,0.3, 0, 0}; //wall distance pid

struct Laser{
  float dist;
  float angle;
};

struct Position{
  float x;
  float y;
  float angle;
};

enum RobotState{
  MAPPING=0,
```

```
  DRIVING=1,
  CHECKING_POSITION=2,
  STOP=3
} global_state=MAPPING; //state of global algorythm

//current position of robot
struct Position global_position;

enum MapStat map[FIELD_SIZE][FIELD_SIZE] ={{UNDISCOVERED}}; //map
of the current area

int possibility_map[FIELD_SIZE][FIELD_SIZE]; //map for storing data during the
bfs

struct Coord {
  int i;
  int j;
} queue[FIELD_SIZE*FIELD_SIZE]; //queue for bfs algorythms

struct Coord new_solids_queue[FIELD_SIZE]; //array of newly created obstacle
points for further gap compensation
int solid_queue_len=0;

const struct Coord EMPTY_COORD ={-1, -1};
struct Coord target=EMPTY_COORD;

//structure that is given as a movement command
struct MotorPair{
  float left;
  float right;
};

//node struct for rrt tree
struct Node{
  short i, j;
  short parent;
};
```

```
//structure for tandom tree and other values for navigation with rrt algorythm
struct Node rrt_tree[RRT_CUTOFF_LEN];
int tree_len=0;
int current_route_node=0;
bool found_route=false;

//to keep angle from 0 to 2P rad.
float absp(float angle)
{
 while(angle<0) angle+=P2;
 while(angle>=P2)angle-=P2;
 return angle;
}

float distance_sqrt(float x1, float y1, float x2, float y2)
{
 return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
}

bool in_bounds_single(int i)
{
 return i>=0 && i<FIELD_SIZE;
}

bool in_bounds(int i, int j)
{
 return i>=0 && i<FIELD_SIZE && j>=0 && j<FIELD_SIZE;
}

bool in_bounds_coord(struct Coord c)
{
 return c.i>=0 && c.i<FIELD_SIZE && c.j>=0 && c.j<FIELD_SIZE;
}

//call once on start
void init(bool use_saved_map)
{
 srand(time(NULL));
```

```
 if(use_saved_map && load_map("map.txt", FIELD_SIZE, FIELD_SIZE, map))
 {
  global_state=DRIVING;
 }else
 {
  global_state=MAPPING;
  for(int i=-SAFE_ZONE; i<=SAFE_ZONE; i++)
  {
   for(int j=-SAFE_ZONE; j<=SAFE_ZONE; j++)
    map[CENTER+i][CENTER+j]=EMPTY;
  }
  for(int i=0; in_bounds_single(i); i++)
  {
   map[i][0]=FORBIDDEN;
   map[i][FIELD_SIZE-1]=FORBIDDEN;
   map[0][i]=FORBIDDEN;
   map[FIELD_SIZE-1][i]=FORBIDDEN;
  }
 }
 global_position = (struct Position){CENTER,CENTER,0};
}

//function to set a new destination for robot to go (will do so only when mapping is
complete)
// otherwise target will be followed after finishing the map
void set_target(int x, int y)
{
 target.i=CENTER+y;
 target.j=CENTER-x;
 if(global_state==STOP)global_state=DRIVING;
}

//set current angle of the movement direction
void set_new_angle(float angle)
{
 global_position.angle=absp(angle);
}
```

```
//in case the angle of movement direction is needed
float get_current_angle()
{
 return global_position.angle;
}

//initial point is considered to be (0,0)
void set_new_position(float x, float y)
{
 global_position.x=CENTER+y;
 global_position.y=CENTER-x;
}

//function that creates a rectangular virtual obstacle that robot will avoid
void add_forbidden_zone(int x1, int y1, int x2, int y2)
{
 x1=CENTER-x1;
 y1=CENTER+y1;
 x2=CENTER-x2;
 y2=CENTER+y2;
 int j_start=fmin(x1, x2);
 int i_start=fmin(y1, y2);
 int j_end=fmax(x1, x2);
 int i_end=fmax(y1, y2);
 for(int i=i_start; i<=i_end; i+=1)
 {
  for(int j=j_start; j<=j_end; j+=1)
  {
   map[i][j]=FORBIDDEN;
  }
 }
}

bool are_coords_same(struct Coord a, struct Coord b)
{
 return (a.i==b.i) && (a.j==b.j);
}
```

```
void clear_possibility_map()
{
 for(int i=0; in_bounds_single(i); i++)
 {
  for(int j=0; in_bounds_single(j); j++)
  {
   possibility_map[i][j]=0;
  }
 }
}

struct Coord get_to_unknown(int starti, int startj)
{
 clear_possibility_map();
 int len=1;
 queue[0].i=starti;
 queue[0].j=startj;
 int i, j;
 int current=0;
 int mini=0, minj=0;
 float min_dist=0;
 bool min_found=false;
 while(current<len)
 {
    i=queue[current].i;
    j=queue[current].j;
    if(!in_bounds(i, j))
    {
     current++;
     continue;
    }
    if(possibility_map[i][j]==1 || map[i][j]==FULL || map[i][j]==FORBIDDEN)
    {
     current++;
     continue;
    }
    possibility_map[i][j]=1;
    if(map[i][j]==UNDISCOVERED)
```

```
      {
       int new_dist=distance_sqrt(starti, startj, i, j);
       if(!min_found || new_dist<min_dist)
       {
         mini=i;
         minj=j;
         min_found=true;
         min_dist=new_dist;
       }
       current++;
       continue;
      }
      queue[len].i=i+1;
      queue[len+1].i=i;
      queue[len+2].i=i-1;
      queue[len+3].i=i;

      queue[len].j=j;
      queue[len+1].j=j+1;
      queue[len+2].j=j;
      queue[len+3].j=j-1;
      len+=4;
      current++;
  }
  if(min_found)return (struct Coord){mini, minj};
  return EMPTY_COORD;

}

float clamp(float value, float min, float max) {
  if (value < min) return min;
  if (value > max) return max;
  return value;
}

//this pid implementation tries to make error=0, so it can be used more universal on
diffrent data models
//and to allow custom error calculations
```

```
float calculate_pid(struct PID* p, float error, float dt)
{
 float res=error*p->kp;
 res+=(error-p->prev_error)/dt*p->kd;
 p->integral_sum+=error*dt;
 if(fabs(p->integral_sum)>=PID_OVERFLOW_ABS) p->integral_sum=0;
 p->prev_error=error;
 res+=p->integral_sum*p->ki;
 return clamp(res, -1, 1);
}

//ranges is a set of intervasd for spotting an obstacle only in defined areas arround
robot
struct Laser check_direction_safety_on_map(struct Position pos, int ranges_len, float
ranges[ranges_len][2])
{
 float mind=0;
 float mina=0;
 bool found=false;
 bool in_range=false;
 for(int i=pos.x-SAFE_ZONE; i<pos.x+SAFE_ZONE; i++)
 {
  for(int j=pos.y-SAFE_ZONE; j<pos.y+SAFE_ZONE; j++)
  {
   if(!in_bounds(i,j)) continue;
   if (map[i][j]!=EMPTY && map[i][j]!=UNDISCOVERED)
   {
    float angle=-atan2(pos.y-j, -pos.x+i);
    float dist=distance_sqrt(pos.x, pos.y, i, j);
    in_range=false;
    for(int k=0; k<ranges_len; k++)
    {
     if(absp(-angle+pos.angle)<=ranges[k][1] && absp(-
angle+pos.angle)>=ranges[k][0])
      {
       in_range=true;
       break;
      }
```

```
        }
        if( (!found || mind>dist) && dist<SAFE_ZONE && in_range)
        {

          mind=dist;
          mina=absp(-angle+pos.angle);
          found=true;

        }
       }
     }
   }
  if (found) return (struct Laser) {mind, mina};
  return (struct Laser) {-1, -1};

}

struct MotorPair drive(struct Position pos, float time_step, struct Coord
current_target)
{
  if(are_coords_same(current_target, EMPTY_COORD)) //condition of stop
  {
    if(global_state==MAPPING)
    {
      global_state=DRIVING;
      save_map("map.txt", FIELD_SIZE, FIELD_SIZE, map);
    }else global_state=STOP;
    return (struct MotorPair){0,0};
  }


  //calculating error of angle towards target
  float angle=-atan2(pos.y-current_target.j, current_target.i-pos.x);
  float angle_error=absp(-angle+pos.angle);
  struct Laser obstacle=check_direction_safety_on_map(pos, 2, (float[2][2])
{{3.0*Pi/2.0, P2}, {0, Pi/2}});
  if(angle_error>Pi)angle_error=angle_error-P2;
  float correction=calculate_pid(&pid, angle_error, time_step);
```

```
 //if no obstacle detected using only first pid correction
 if(obstacle.dist<0)
 {
  return (struct MotorPair){clamp(0.5+correction,-1,1), clamp(0.5-correction, -1,
1)};
 }


 //calculating error of angle towards obstacle
 float obstacle_error=0;
 if(obstacle.angle<Pi)obstacle_error=Pi-obstacle.angle;
 else obstacle_error=3.0*Pi/2.0-obstacle.angle;
 obstacle_error=clamp(obstacle_error, -Pi/2, Pi/2);

 float obstacle_correction=-calculate_pid(&pid_obstacle, obstacle_error,  time_step);

 //calculating distance error to the obstacle
 //clamp is used to ban robot for going closer but allowing to drive off the wall
 float distance_correction=0;
 if(obstacle.angle<Pi)distance_correction=calculate_pid(&pid_distance, clamp(-
(SAFE_ZONE-obstacle.dist)/SAFE_ZONE, -1, 0), time_step);
 else distance_correction=calculate_pid(&pid_distance, clamp((SAFE_ZONE-
obstacle.dist)/SAFE_ZONE,0, 1), time_step);

 //getting total sum of all corrections and
 float total_correction;
 if(obstacle.dist<=MINIMUM_SAFE_ZONE) return (struct
MotorPair){clamp(0.2+obstacle_correction+distance_correction*0.5,-1,1),
clamp(0.2-obstacle_correction-distance_correction*0.5, -1, 1)};
 else total_correction=clamp(correction+(obstacle_correction+distance_correction),
-1, 1);
 return (struct MotorPair){clamp(0.5+total_correction,-1,1), clamp(0.5-
total_correction, -1, 1)};
}

void gap_compensation(int i, int j)
{
```

```
  for(int di=-MAX_GAP_SIZE; di<=MAX_GAP_SIZE; di++)
  {
   if(!in_bounds_single(i+di)) continue;
   if (map[i+di][j]==FULL || map[i+di][j]==FORBIDDEN)
   {
    for(int k=fmin(i, i+di); k<=fmax(i, i+di); k++)
    {
     if(map[k][j]!=FORBIDDEN)map[k][j]=FULL;
    }
    break;
   }
  }
  for(int dj=-MAX_GAP_SIZE; dj<=MAX_GAP_SIZE; dj++)
  {
   if(!in_bounds_single(j+dj)) continue;
   if (map[i][j+dj]==FULL || map[i][j+dj]==FORBIDDEN)
   {
    for(int k=fmin(j, j+dj); k<=fmax(j, j+dj); k++)
    {
     if(map[i][k]!=FORBIDDEN) map[i][k]=FULL;
    }
    break;
   }
  }
}

bool empty_spots_compensation(int i, int j)
{
  short cnt=0;
  for(int di=-1; di<=1; di++)
  {
   for(int dj=-1; dj<=1; dj++)
   {
    if(!in_bounds(i+di, j+dj)) continue;
    if(map[i+di][j+dj]!=UNDISCOVERED) cnt++;
   }
  }
  if(cnt>=6 && map[i][j]!=FORBIDDEN)
```

```
  {
   map[i][j]=EMPTY;
   return true;
  }
  return false;
}

void place_beam(struct Laser beam, struct Position pos) {
  float angle = absp(pos.angle - beam.angle);
  int x0 = (int)round(pos.x);
  int y0 = (int)round(pos.y);

  if (beam.dist < 0) return;

  bool too_close=beam.dist<SAFE_ZONE;
    int x1 = (int)round(pos.x + fmin(LIDAR_DIST, beam.dist) * cos(angle));
    int y1 = (int)round(pos.y +fmin(LIDAR_DIST, beam.dist) * sin(angle));

    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;
    int err = 2*dy-dx;
    if(dy>dx)err=2*dx-dy;

    while(x0!=x1 || y0!=y1) {
       if (x0 >= 0 && x0 < FIELD_SIZE && y0 >= 0 && y0 < FIELD_SIZE)
          if(map[x0][y0]==UNDISCOVERED || (map[x0][y0]==FULL &&
too_close))map[x0][y0] = EMPTY;
       if(dy>dx)
       {
         y0+=sy;
         if(err>=0)
         {
           x0+=sx;
           err+=2*dx-2*dy;
         }else err+=2*dx;
       }else{
```

```
        x0+=sx;
        if(err>=0)
        {
          y0+=sy;
          err+=2*dy-2*dx;
        }else err+=2*dy;
       }

    }


  if(beam.dist<LIDAR_DIST)
  {
    int x_hit = (int)round(pos.x + beam.dist * cos(angle));
    int y_hit = (int)round(pos.y + beam.dist * sin(angle));

    if (x_hit >= 0 && x_hit < FIELD_SIZE && y_hit >= 0 && y_hit < FIELD_SIZE)
    {
      map[x_hit][y_hit] = FULL;
      new_solids_queue[solid_queue_len]=(struct Coord) {x_hit, y_hit};
      solid_queue_len++;
    }
  }

}

void push_node(int i, int j, int parent_id)
{
  rrt_tree[tree_len].i=i;
  rrt_tree[tree_len].j=j;
  rrt_tree[tree_len].parent=parent_id;
  tree_len++;
}

int find_closest_node(int i, int j)
{
  int closest_id=0;
  float distance=0;
```

```
  bool found=false;
  for(int k=0; k<tree_len; k++)
  {
    float current_dist=distance_sqrt(rrt_tree[k].i, rrt_tree[k].j, i, j);
    if(!found || current_dist<distance)
    {
      distance=current_dist;
      found=true;
      closest_id=k;
    }
  }
  return closest_id;
}

//simple function that looks for obstacles between two points
bool is_free(int x1, int y1, int x2, int y2) {
  int steps = (int)distance_sqrt(x1, y1, x2, y2);
  int dx=x2 - x1;
  int dy=y2 - y1;
  for (int i = 0; i <= steps; i++) {
    int x = x1 + dx * i / steps;
    int y = y1 + dy * i / steps;
    if(x < 0 || x >= FIELD_SIZE || y < 0 || y >= FIELD_SIZE) return false;
    if (map[x][y] != EMPTY)
      return false;
  }
  return true;
}

void build_rrt(struct Position pos, struct Coord t)
{
      int new_x = rand() % (FIELD_SIZE);
      int new_y = rand() % (FIELD_SIZE);
      int closest = find_closest_node(new_x, new_y);
      if(rrt_tree[closest].i==new_x && rrt_tree[closest].j==new_y) return;
      if (!in_bounds(new_x, new_y) || map[new_x][new_y]!=EMPTY) return;
      if (!is_free(rrt_tree[closest].i, rrt_tree[closest].j, new_x, new_y)) return;
      push_node(new_x, new_y, closest);
```

```
    int bot_closest=find_closest_node(pos.x, pos.y);

    if (is_free(rrt_tree[bot_closest].i, rrt_tree[bot_closest].j, pos.x, pos.y)) {
      push_node(pos.x, pos.y, bot_closest);
      found_route=true;
      int current = tree_len - 1;
      while (current !=0) {
        current = rrt_tree[current].parent;
      }
      current_route_node=tree_len-1;

      return;
    }

}

//main movement handler, time_step should be a time interval in millisecconds from
last call
struct MotorPair get_directions(struct Laser *beams, int length, float time_step)
{

  if(global_state==MAPPING) //mapping the area
  {

    solid_queue_len=0;
    for (int i=0; i<length; i++)
      place_beam(beams[i], global_position);
    for(int i=0; i<solid_queue_len; i++)
    {
      gap_compensation(new_solids_queue[i].i, new_solids_queue[i].j);
    }

    struct Coord unknown_area=get_to_unknown(global_position.x,
global_position.y);
    while(empty_spots_compensation(unknown_area.i, unknown_area.j))
    {
      unknown_area=get_to_unknown(global_position.x, global_position.y);
    }
```

```
   return drive(global_position, time_step, unknown_area);
  }else if(global_state==DRIVING && !are_coords_same(target,
EMPTY_COORD)) //if target is set and robot is driving towards it
  {
   if(tree_len==0) //setting initial node
   {
    push_node(target.i, target.j, 0);
   }
   else if(!found_route && tree_len<RRT_CUTOFF_LEN &&
map[target.i][target.j]==EMPTY) //building rrt and checking if starget is in mapped
area
    build_rrt(global_position, target);
   else if(!found_route) //if route not found and one of safety requirements failed then
stopping
   {
    target=EMPTY_COORD;
    global_state=STOP;
   }
   else if(found_route) //if route is found then follow it
   {
    struct Coord local_target=(struct Coord){rrt_tree[current_route_node].i,
rrt_tree[current_route_node].j};
    if(distance_sqrt(global_position.x, global_position.y, local_target.i,
local_target.j)<=MINIMUM_SAFE_ZONE)
    {
     if(current_route_node==rrt_tree[current_route_node].parent) return (struct
MotorPair){0,0};
     current_route_node=rrt_tree[current_route_node].parent;
    }
    return drive(global_position, time_step, local_target);
   }
  }
  return (struct MotorPair){0,0}; //in all other cases sending a signal to stop any
movement
}
```