НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ННК "Інститут прикладного системного аналізу"
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)

"___"_____2016 р.

Дипломна робота

___першого (бакалаврського)_____**рівня вищої освіти**
(першого (бакалаврського), другого (магістерського))

зі спеціальності7.05010102, 8.05010102 Інформаційні технології проектування
_____7.05010103, 8.05010103 Системне проектування
(код та назва спеціальності)

на тему: "Алгоритми та математичні моделі рухомої платформи"

Виконав: студент _IV_ курсу, групи ___ДА-21____
(шифр групи)

_____Михалько Віталій Геннадійович_____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник _____доц., к.т.н., Харченко К.В._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Економічний ___проф., д.е.н., Семенченко Н.В.__ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____доц., к.т.н., Тимощук О.Л._____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормоконтроль _____ст. викладач Бритов О.А._____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2016 року

Національний технічний університет України
«Київський політехнічний інститут»

Факультет (інститут) ННК "Інститут прикладного системного аналізу"
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти Перший(Бакалаврський)
(перший (бакалаврський), другий (магістерський) або спеціаліста)

Спеціальність 7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)
«___»_____2016 р.

ЗАВДАННЯ
на дипломний проект (роботу) студенту
Михальку Віталію Геннадійовичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи)) Алгоритми та математичні моделі рухомої платформи
керівник проекту (роботи))Харченко К.В., к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 12 травня 2016 р. № 50-ст

2. Строк подання студентом проекту (роботи) 10.06.2016

3. Вихідні дані до проекту (роботи) _____

Операційна система Linux Mint
Частота процессору 2.0 ГГц
Середовище розробки - Intellij IDEA
Мова програмування - Java
Графічний фреймворк - JavaFX
Система збірки проекту - Maven
Система контролю версій - Git
Бібліотеки, що використовувались: Apache Common Math, Guava, JUnit 4

4. Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити)

1. Скласти математичну модель рухомої платформи.
2. Розробити алгоритм пошуку шляху.
3. Вибрати мову програмування і реалізувати математичну модель та алгоритм пошуку шляху.
4. Провести функціонально-вартісний аналіз отриманого програмного продукту.
5. Проаналізувати результати роботи, зробити висновки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів тощо)

1. Схема математичної моделі – плакат.
2. Схема і приклади роботи алгоритму пошуку шляху – плакат.
3. Архітектура системи – плакат.

6. Консультанти розділів проекту (роботи)*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Семенченко Н.В., професор		

6. Дата видачі завдання 01.02.2015

Календарний план

№ з/п	Назва етапів виконання дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	01.02.2016	
2	Ознайомлення з технічною літературою і підготовка теоретичної частини	15.02.2016	
3	Аналіз вимог завдання, вибір методів і засобів розв'язання поставленої задачі	28.02.2016	
4	Розробка математичної моделі і алгоритмів	10.03.2016	
5	Тестування розроблених алгоритмів і математичної моделі. Перевірка відповідності завданню.	15.03.2016	
6	Захист дипломної роботи	30.04.2016	
7	Оформлення дипломної роботи	31.05.2016	

* Консультантом не може бути зазначено керівника дипломного проекту (роботи).

8	Отримання допуску до захисту та подача роботи в ДЕК	10.06.2016	

Студент _____ В.Г. Михалько
(підпис) (ініціали, прізвище)

Керівник проекту (роботи) _____ К.В. Харченко
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

бакалаврської дипломної роботи Михалька Віталія Геннадійовича на тему: «Алгоритми та математичні моделі рухомої платформи»

Дипломна робота присвячена розробці математичної моделі і алгоритмів пошуку шляху і оминання перешкод для чотириколісних рухомих платформ.

В роботі описано основні принципи математичного моделювання та розглянуто існуючі підходи для вирішення задачі знаходження шляху і оминання перешкод. В результаті аналізу рухомої платформи було складено математичну модель, яка в багатьох випадках є вдалим наближенням цієї платформи. Було розроблено швидкий алгоритм для прокладання шляху і оминання перешкод, який ґрунтується на поєднанні вже існуючих алгоритмів Theta* і гібридного A*.

Тестування розробленого програмного забезпечення показали, що математична модель і алгоритм пошуку шляху дозволяють доволі швидко знаходити шлях до поставленої цілі, навіть в умовах значної кількості перешкод.

Результати роботи можуть бути застосовані в робототехніці, зокрема при розробці безпілотних автомобілів і транспортних систем.

Загальний обсяг роботи: 88 сторінок, 32 рисунків, 6 таблиць, 15 посилань

Ключові слова: математична модель, алгоритми пошуку шляху, робототехніка, Theta*, гібридний A*

АННОТАЦИЯ

бакалаврской дипломной работы Михалько Виталия Геннадьевича на тему: «Алгоритмы и математические модели подвижной платформы»

Дипломная работа посвящена разработке математической модели и алгоритмов поиска пути и обхода препятствий для четырехколесных подвижных платформ.

В работе описаны основные принципы математического моделирования и рассмотрены существующие подходы для решения задачи нахождения пути и обхода препятствий. В результате анализа подвижной платформы была составлена математическая модель, которая во многих случаях является удачным приближением этой платформы. Был разработан быстрый алгоритм для прокладки пути и обхода препятствий, основанный на сочетании существующих алгоритмов Theta * и гибридного A *.

Тестирование разработанного программного обеспечения показали, что математическая модель и алгоритм поиска пути позволяют достаточно быстро находить путь к поставленной цели, даже в условиях значительного количества препятствий.

Результаты работы могут быть применены в робототехнике, в частности при разработке беспилотных автомобилей и транспортных систем.

Общий объем работы: 88 страниц 32 рисунков, 6 таблиц, 15 ссылок

Ключевые слова: математическая модель, алгоритмы поиска пути, робототехника, Theta*, гибридный A*

ANNOTATION

For the bachelor thesis of Mykhalko Vitaly Hennadiovych on “Algorithms and mathematical models of mobile robot”

Thesis is devoted to the development of mathematical model of mobile robot and path finding algorithms.

Mathematical modeling principles and basic path finding algorithms were described. Mathematical model of mobile robot was implemented. Algorithm for effective path finding was developed. Algorithm was based on Theta* and Hybrid A* algorithms.

Tests showed that the model gives good approximation of real mobile robots and that path finding algorithms are quite effective.

Results of research can be used in robotics and autonomous car development.

Total volume of work: 88 pages, 32 figures, 6 tables, 15 links.

Keywords: mathematical model, path finding algorithms, robotics, Theta*, Hybrid A*

ЗМІСТ

ВСТУП

В наші часи все більшого поширення набувають автономні мобільні платформи і робототехніка. Їх застосування можливе в багатьох сферах, а особливо в тих, які містять підвищену небезпеку для людей. Також, все частіше робототехніка застосовується для автоматизації різних процесів і виробництва.

Однією з основних задач XXI сторіччя у галузі робототехніки постала розробка автопілотів для автомобілів і інших рухомих платформ. Ця задача є дуже перспективною, бо зважаючи на те, що методи штучного інтелекту і машинного навчання в багатьох прикладних завданнях вже перевершують людські можливості, існує значна ймовірність зменшення аварійності на дорогах і збільшення безпеки пересування в цілому.

Розробка автономних рухомих платформ і робототехніки є доволі складною задачею. Вона складається з двох головних частин: фізичне складання робота і розробка на інтегрування відповідного програмного забезпечення.

Для фізичного конструювання робота потрібно підібрати правильні компоненти, правильно їх поєднати, а також налаштувати зв'язок робота з відповідним сервером, для можливості віддаленого керування.

При розробці програмного забезпечення для робототехніки постає багато задач:

- побудова математичної моделі
- локалізація
- алгоритми прокладання шляху і оминання перешкод
- алгоритми зворотнього зв’язку і контролю

- пошук і розпізнавання предметів

Першою задачею, яку треба вирішити постає побудова математичної моделі. Математична модель описує такі параметри робота як положення, швидкість, напрям руху і дозволяє спрогнозувати зміну цих параметрів в залежності від часу і управління.

Після побудови математичної моделі потрібно вирішити задачу локалізації, що означає можливість визначення положення об’єкту в просторі в залежності від даних сенсорів і інформації про попередній стан системи. Так як неможливо дати дуже точну оцінку положення якогось предмету в просторі, тут стають у нагоді методи теорії ймовірностей і математичної статистики. Двома основними способами вирішення задачі локалізації є фільтр Калмана і частинковий фільтр.

Наступною важливою характеристикою автономних роботів і рухомих платформ є вміння прокладати шлях і оминати перешкоди, які можуть зустрітися на цьому шляху. При цьому треба враховувати математичну модель, адже рухомі платформи часто не є голономними, тобто вони не можуть рухатись в будь-якому напрямі.

Будь-яка реальна система не є ідеальною, тобто неможливо спрогнозувати точне значення її параметрів. Після того, як робот проклав шлях до цілі і визначив послідовность керувальних команд для руху по цьому шляху, існує доволі висока ймовірність відхилення від правильної траєкторії. При цьому, незначні відхилення на початку шляху можуть призвести в подальшому до серйозної помилки. Для того щоб зменшити ефекти від таких помилок, потрібно використовувати алгоритми зворотного зв’язку і виконувати керувальні команди, які дозволять зменшити відхилення. Одним із найбільш поширених алгоритмів для цього є PID-контролер.

Останньою з основних підзадач розробки програмного забезпечення для робототехніки є розпізнавання предметів, яке дозволить знаходити їх а також визначати більш точне положення рухомої платформи.

Метою цієї роботи є розробка математичної моделі і алгоритмів пошуку шляху з оминанням перешкод для чотириколісної рухомої платформи, яка є доволі поширеною у реальному світі. Шлях, який будується за допомогою даних алгоритмів має бути реальним для виконання з точки зору математичної моделі робота і при цьому бути близьким до оптимального. Результати роботи можуть бути використані при створенні автоматизованих та автоматичних систем керування для об’єктів, спрощена модель яких є подібною до даної рухомої платформи, зокрема для автомобілів.

1. МАТЕМАТИЧНА МОДЕЛЬ РУХОМОЇ ПЛАТФОРМИ

1.1 Означення математичної моделі

Математична модель це опис системи за допомогою математичних рівнянь і концепцій. Вона дозволяє симулювати поведінку реальної системи і передбачати зміну значень її параметрів в залежності від часу і керувальних сигналів. Моделювання завжди було основою інженерних наук, бо при наявності моделі об’єкта, який потрібно реалізувати, можна легко спрогнозувати його стан в довільних момент часу, поведінку при різних обставинах, і в результаті цього зробити висновки чи буде він правильно виконувати свої функції і визначити що потрібно в ньому покращити або змінити. Математична модель є часто доволі спрощеним представленням системи, в ній може бути проігноровато багато показників, але, тим не менш, правильно побудована модель в багатьох випадках все одно надасть доволі точний прогноз про стан і поведінку системи.

Математичне моделювання застосовується у двох основних випадках:

- передбачення поведінки реальної системи чи явища
- передбачення поведінки системи, яка проектується

Для передбачення поведінки вже існуючої системи (часто природньої) застосовується так званий науковий метод [1]. Він складається з трьох основних частин:

- спостереження
- моделювання
- передбачення

Науковий метод схематично зображено на рисунку 1.1

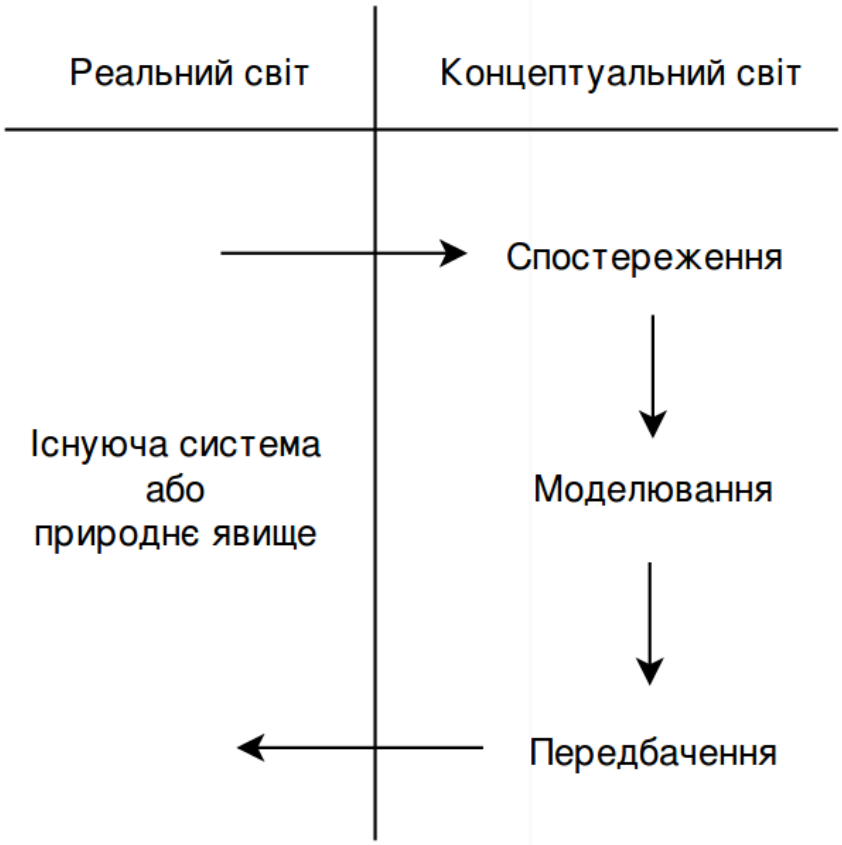

Рисунок 1.1 – Схематичне зображення наукового методу

В той час як науковці займаються описом і складанням моделей для природних явищ, інженери зацікавлені в створенні систем, яких ще не було раніше. Інженери повинні описати і проаналізувати концептуальні системи, які ще не були втілені в реальність, щоб побачити чи будуть вони правильно виконувати свої функції, тобто вони самі є творцями пристроїв чи процесів, які треба змоделювати.

1.2 Принципи математичного моделювання

При побудові математичної моделі для якогось об’єкту чи системи, в першу чергу треба дати відповіді на такі питання:

- Навіщо будується модель?
- Що ми хочемо знайти чи вияснити в результаті побудови моделі?
- Для чого можна використати модель?

- Що ми знаємо, які в нас є дані вхідні дані?
- Що можна припустити про об’єкт чи систему?
- Які передбачення повинна робити модель?
- чи є передбачення правильними і прийнятними?
- як можна покращити модель

Послідовність, в якій необхідно дати відповіді на ці питання, зображена на рисунку 1.2

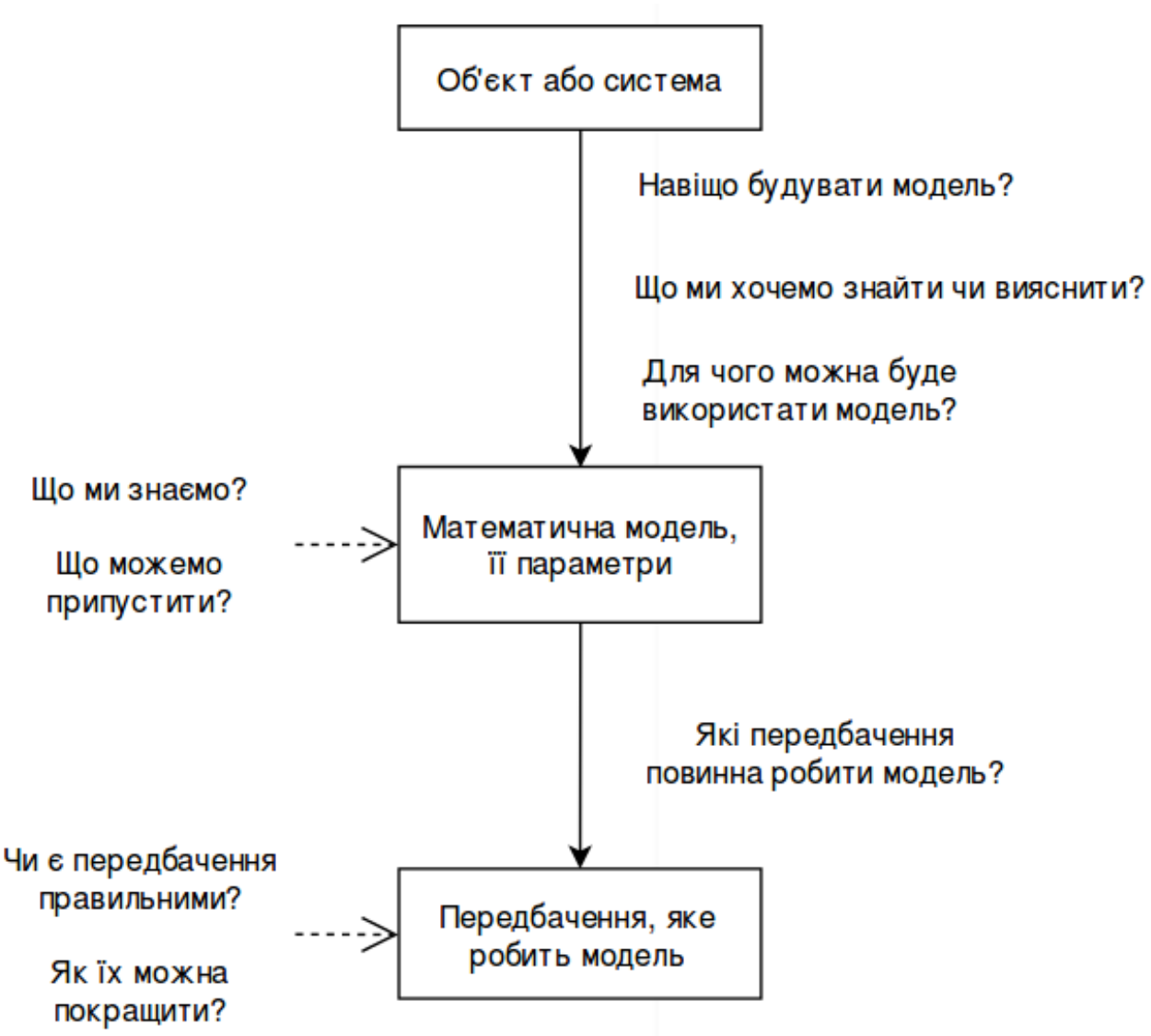

Рисунок 1.2 – Принципи математичного моделювання

1.3 Основні етапи математичного моделювання

Математичне моделювання часто є доволі складною задачею, але загалом в ньому можна виділити такі основні етапи [2]:

1. Створення якісної моделі (з'ясування законів, які діють в системі, виявлення характерних рис процеса, які визначають його особливості).
2. Створення математичної моделі (постановка математичної задачі, складання і розв'язання рівнянь, які описують систему).
3. Вивчення математичної моделі (якісна оцінка моделі, з'ясування поведінки при граничних умовах, розробка алгоритму і написання програми).
4. Отримання результатів і їх інтерпретація.
5. Використання отриманих результатів.

1.4 Основні види рухомих платформ

1.4.1 Колісні рухомі платформи

Найбільш поширеним видом рухомих платформ є чотирьохколісна машиноподібна платформа. В ній зазвичай задні колеса приєднані до двигуна, що дозволяє рухатись вперед і назад, а передні колеса є рульовими. Однією з особливостей такої платформи є те, що вона не може розвернутись на місці. Це є доволі важливим фактором, який треба враховувати при розробці алгоритмів керування.

Система рульового керування передніх колес може бути розроблена за допомогою паралельного повороту (колеса знаходяться на одній осі) або використовуючи принцип Акермана [3]. Перший варіант є більш очевидним і легшим у реалізації, але його основним недоліком є те, що для повороту всієї осі потрібно більше вільного місця у передній частині платформи. Цей недолік відсутній, якщо будувати рульове керування, використовуючи принцип

Акермана. В цьому випадку передню вісь не потрібно повертати, але передні колеса повинні бути повернуті за різні кути, так щоб перпендикулярні до коліс лінії сходились в одній точці, яка і буде центром повороту. Схематичне зображення цих двох систем рульового керування показано на рисунках 1.3 і 1.4.

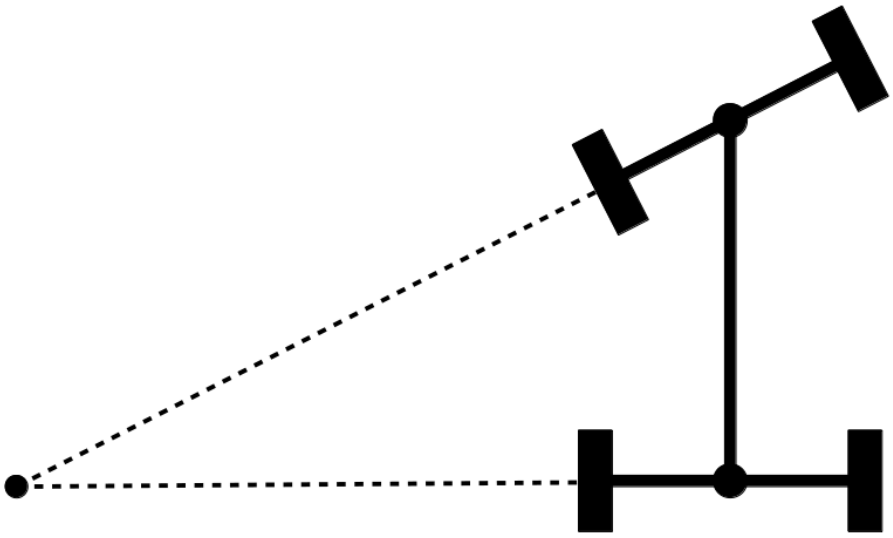

Рисунок 1.3 – Паралельне рульове керування

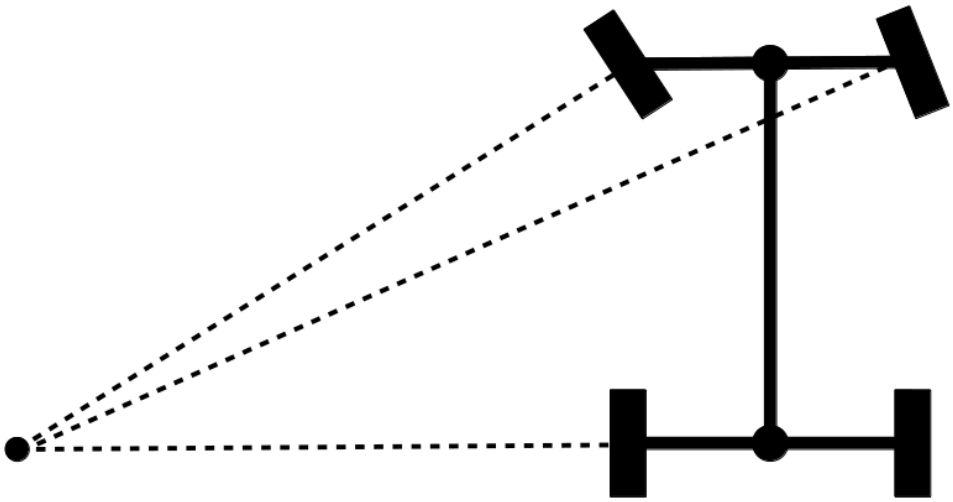

Рисунок 1.4 – Рульове керування Акермана

1.4.2 Гусеничні рухомі платформи

Іншим поширеним видом рухомих платформ є гусеничні платформи. Їхня основна перевага є в тому, що ліва і права гусениці можуть обертатись з різною швидкістю, що надає таким платформам більшу свободу рухів і, зокрема, можливість робити повороти на одному місці. Але вони є складнішими в реалізації і швидше зношуються. Приклад такої платформи наведено на рисунку 1.5.

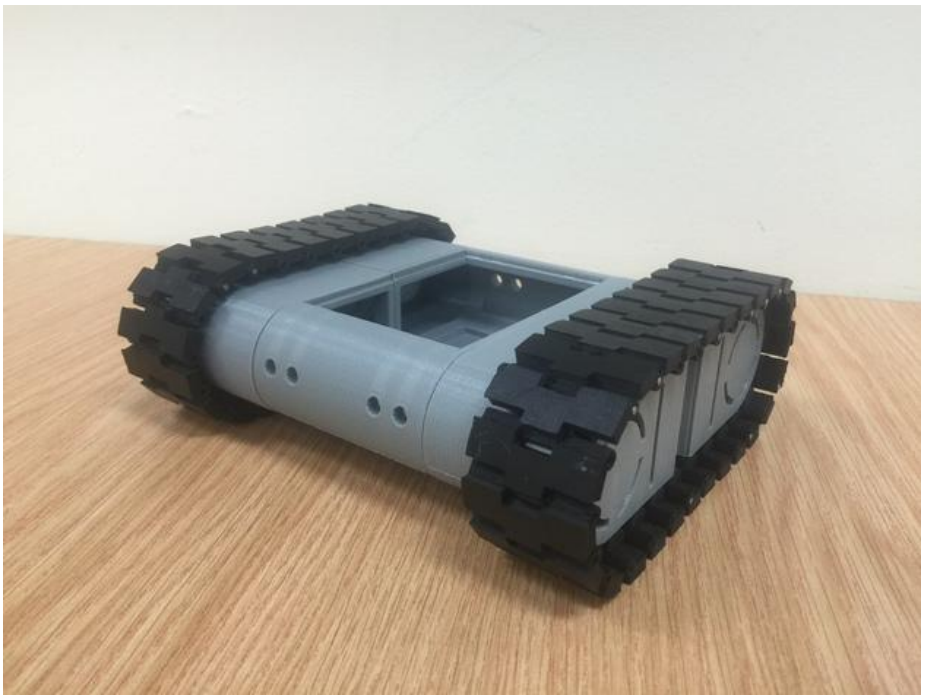

Рисунок 1.5 – Гусенична платформа [4]

1.4.3 Крокуючі рухомі платформи

Крокуючі машини рухаються, використовуючи спеціальні механізми, які нагадують лапи тварин. Крокуючи системи пересування можуть бути використані у значно більшій кількості випадків, ніж колісні, так як вони не вимагають рівної, підготовленої поверхні для пересування. Але, з іншого боку, вони є значно складнішими в реалізації, ніж колісні чи гусеничні платформи,

через це поки не набули значного поширення. Зазвичай в таких системах використовується 4 або 6 лап. Приклад крокуючого робота наведено на рисунку 1.6.

Рисунок 1.6 – Крокуюча рухома платформа[5]

1.5 Складання математичної моделі для чотириколісної рухомої платформи

1.5.1 Визначення основних параметрів математичної моделі

В цій роботі основна увага приділяється чотириколісній рухомій платформі з паралельним рульовим керуванням, але багато з описаних технік і алгоритмів можуть бути успішно застосованій і для інших видів платформ.

Основна мета побудови математичної моделі рухомої платформи полягає в тому, щоб можна було спрогнозувати її поведінку у разі застосування керувальних сигналів, і виходячи з цього правильно формувати самі керувальні сигнали для можливості зміни положення і просторової орієнтації платформи. Математична модель повинна бути доволі загальною, щоб її можна було застосувати для управління багатьох чотириколісних роботів, незалежно від їхніх розмірів і обмежень рульового керування.

Для спрощення побудови моделі зробими наступні припущення про способи впливу на систему:

1. Задавання швидкості платформи, яка може бути одним і з трьох значень: v_{pos} , v_0, і v_{neg} що відповідно означають рух вперед, стояння на місці і рух назад. Будемо вважати, що після застосування відповідного керувального сигналу, швидкість задається майже миттєво, тому прискоренням можна знехтувати.
2. Задавання кута повороту передньої осі, який також може приймати одне з трьох значень:α_{neg}, α_0, α_{pos} (бути від'ємним, нульовим і додатнім), що означає можливіть повороту праворуч, руху прямо і повороту ліворуч. Будемо вважати, що задавання кута повороту передньої осі також відбувається миттєво.

Після визначення можливостей керування платформою, можна ввести основні параметри, які будуть описувати систему:

1. l_{body} - довжина корпуса платформи.
2. w_{body} - ширина корпуса платформи.
3. $l_{chassis}$ - довжина шасі.
4. $w_{chassis}$ - ширина шасі.
5. v - скалярне значення швидкості руху.
6. x_p, y_p – координати положення платформі у просторі.
7. α - кут орієнтації платформи у просторі (кут відхилення від осі OX)
8. β – кут повороту передньої осі відносно платформи.
9. t – момент часу, в якій розглядається система.

При цьому, такі параметри як l_{body}, w_{body}, $l_{chassis}$, $w_{chassis}$ є стаціонарними, а значення параметрів v, x_p, y_p, α, і β залежать від часу. Схематичне зображення платформи і її основних параметрів наведено на рисунку 1.7.

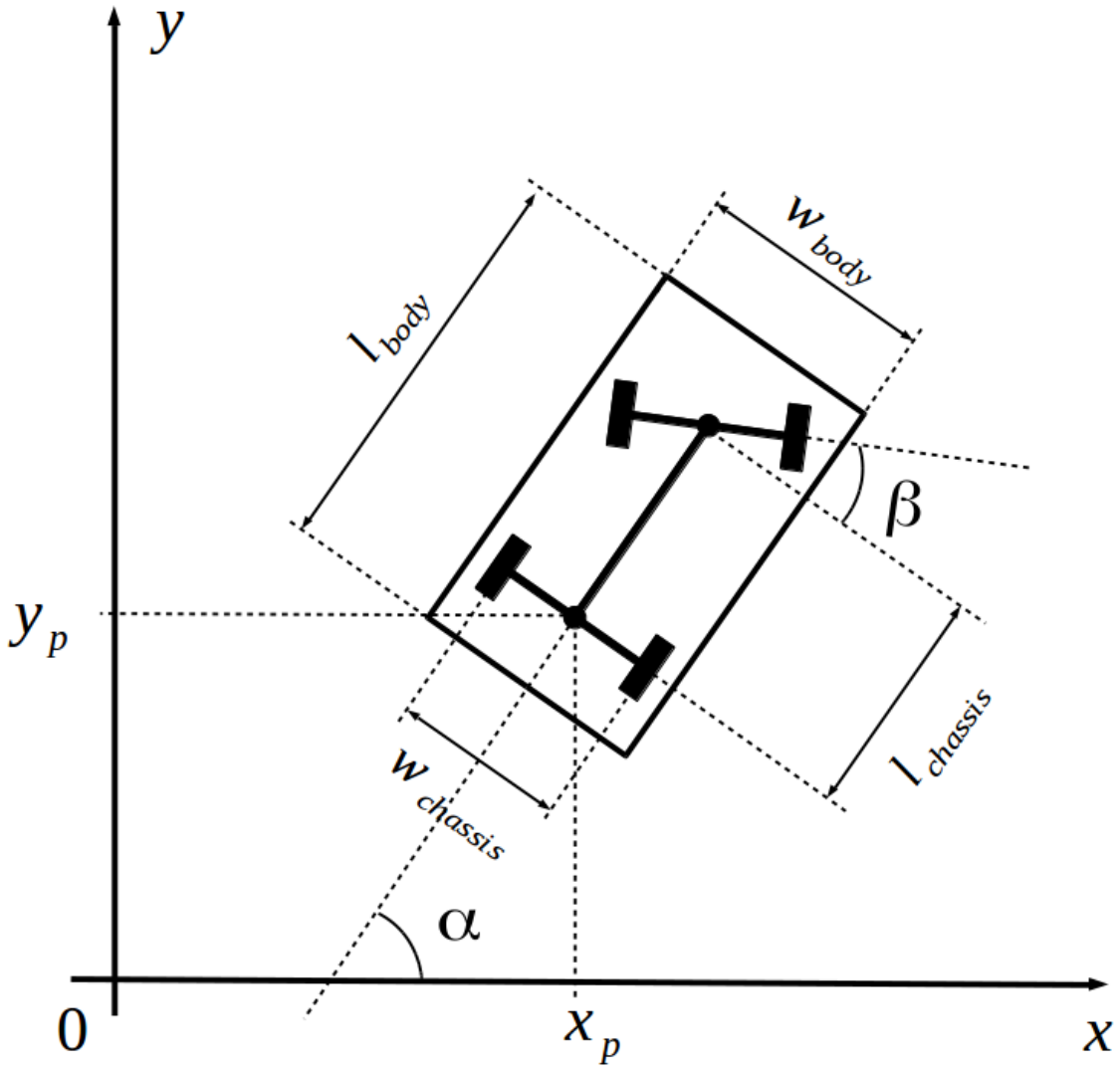

Рисунок 1.7 – Схематичне зображення платформи і її основних параметрів

1.5.2 Рівняння для руху прямо

Найпростіший випадок руху платформи - це рух прямори. В цьому випадку при заданій швидкості будуть змінюватись тільки координати x, y платформи, а решта залишатись сталими. Користуючись правилами кінематики можна записати такі рівняння для зміни координат:

$$x = x_0 + v_x \Delta t \quad (1)$$

$$y = y_0 + v_y \Delta t \quad (2)$$

Проекції швидкостей можна визначити за допомогою модуля швидкості і кута орієнтації платформи в просторі:

$$v_x = v \cos(\alpha) \quad (3)$$

$$v_y = v \sin(\alpha) \quad (4)$$

Отже, рівняння зміни координат можна переписати таким чином:

$$x = x_0 + v \cos(\alpha)\, \Delta t \quad (5)$$

$$y = y_0 + v \sin(\alpha)\, \Delta t \quad (6)$$

1.5.3 Рівняння для повороту

Коли передня вісь відхилена на якийсь кут, то при русі, рухома платформа буде робити поворот. Для того, щоб визначити зміну координат положення платформи під час повороту, спочатку треба визначити координати центру кола, по якому і буде виконуватись поворот. Ця точка знаходиться на перетину ліній, які виходять з передньої і задньої осей (Рисунок 1.8).

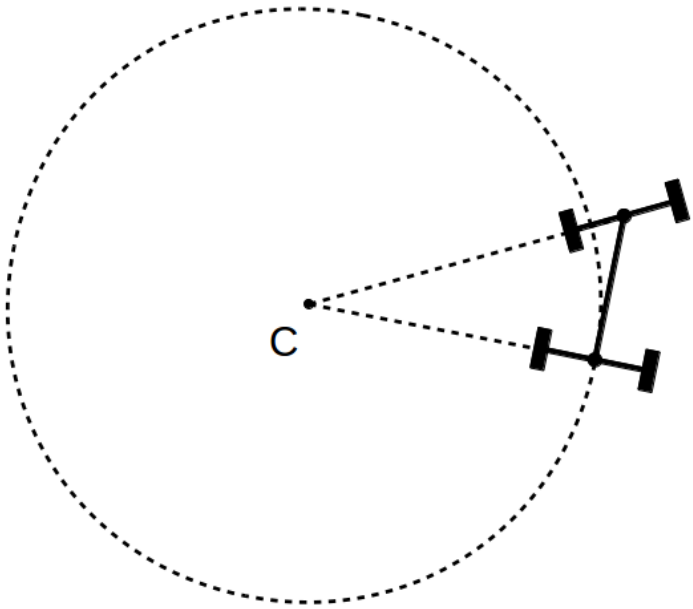

Рисунок 1.8 – Коло повороту

Для визначення координат центру кола спочатку необхідно побудувати рівняння прямих, які проходять через передню і задню вісь, а потім класти систему рівнянь, розв’язком якої і буде положення центру.

Як відомо, рівняння прямої має такий вигляд:

$$Ax + By + C = 0 \tag{7}$$

При наявності точки, через яку проходить пряма, і перпендикулярного вектору, рівняння прямої можна записати таким чином:

$$A'(x - x_0) + B'(y - y_0) = 0 \tag{8}$$

де A' і B' – це координати перпендикулярного вектору, а x_0 і y_0 – це координати точки, через яку проходить пряма.

Якщо зробити прості перетворення, то можна визначити коефіцієнти рівняння (7) через рівняння (8)

$$A = A' \tag{9}$$

$$B = B' \tag{10}$$

$$C = -(A'x_0 + B'y_0) \tag{11}$$

Побудуємо спочатку рівняння прямої для задньої осі. Нам відоме положення центру задньої осі - це параметри x_p і y_p математичної моделі. Перпендикулярний вектор (A_1, B_1) можна знайти, використовуючи кут орієнтації платформи у просторі – α:

$$A_1 = \cos(\alpha) \tag{12}$$

$$B_1 = \sin(\alpha) \tag{13}$$

Коефіцієнт C_1 обчислюється таким чином:

$$C_1 = -(A_1 x_p + B_1 y_p) \tag{14}$$

Складемо тепер рівняння прямої для передньої осі.

Координати центру передньої осі можна знайти за наступними формулами:

$$x_f = x_p + \cos(\alpha)\, l_{chassis} \tag{15}$$

$$y_f = y_p + \sin(\alpha)\, l_{chassis} \tag{16}$$

Координати перпендикулярного до осі вектора (A_2, B_2) можна обчислити, використовуючи кут орієнтації платформи у просторі і кут повороту передньої осі:

$$A_2 = \cos(\alpha + \beta) \tag{17}$$

$$B_2 = \sin(\alpha + \beta) \tag{18}$$

Коефіцієнт C_2 обчислюється наступним чином:

$$C_2 = -(A_2 x_f + B_2 y_f) \tag{19}$$

В результаті маємо таку систему рівнянь:

$$\begin{cases} A_1 x + B_1 y = -C_1 \\ A_2 x + B_2 y = -C_2 \end{cases} \tag{20}$$

розв’язком якої і будуть координати центру кола повороту.

Тепер потрібно визначити, чи платформа буде рухатись за годинниковою стрілкою, чи проти. Це можна зробити, використовуючи векторний добуток двох векторів. Перший вектор ($\vec{a} = (a_x, a_y)$) сполучає центр кола повороту і центр задньої осі платформи, а другий вектор ($\vec{b} = (b_x, b_y)$) вказує напрям орієнтації платформи у просторі (як було зазначено вище, його можна знайти, використовуючи кут орієнтації платформи у просторі (α) .

Векторний добуток обчислюється за такою формулою:

$$p = a_x b_y - a_y b_x \tag{21}$$

Якщо p > 0, то це означає, що платформа буде рухатись проти годинникової стрілки, а при p < 0 - за годинниковою стрілкою.

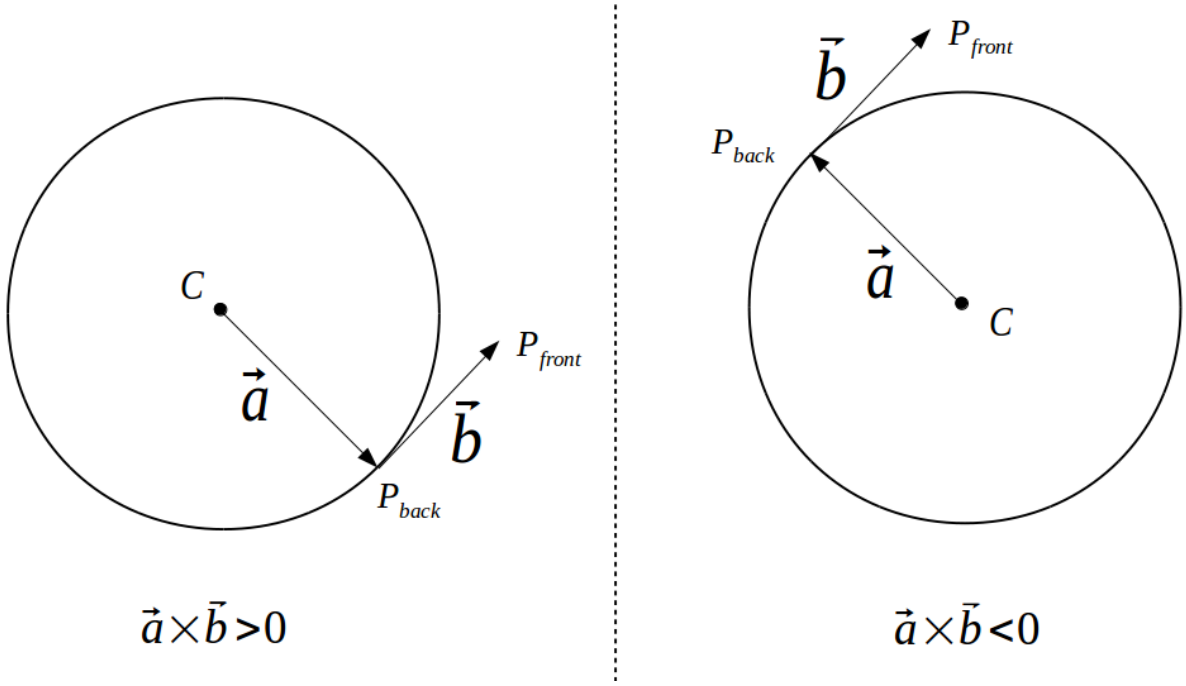

Рисунок 1.9 – Визначення напряму обертання

Далі можна визначити, як змінюються координати положення платформи під час повороту. Для цього спочатку треба обчислити кут дуги кола, яку проїжає платформа. В залежності від руху напряму обертання цей кут може бути додатнім або від’ємним:

$$\Delta\gamma = \frac{v\Delta t}{R_c}\text{, якщо p > 0} \quad (22)$$

$$\Delta\gamma = -\frac{v\Delta t}{R_c}\text{, якщо p < 0} \quad (23)$$

де R_c – радіус кола повороту.

Потім можна обчислити значення нових координат і кута орієнтації рухомої платформи:

$$x'_p = x_c + R_c\cos(\gamma + \Delta\gamma) \quad (24)$$

$$y'_p = y_c + R_c\sin(\gamma + \Delta\gamma) \quad (25)$$

$$\alpha' = \alpha + \Delta\gamma \quad (26)$$

де (x_c, y_c) – це координати точки O' (центр кола повороту), а γ – це кут між віссю $O'X'$ і вектором, що сполучає центр кола повороту із центром задньої осі рухомої платформи. Це проілюстровано на Рисунку 1.10.

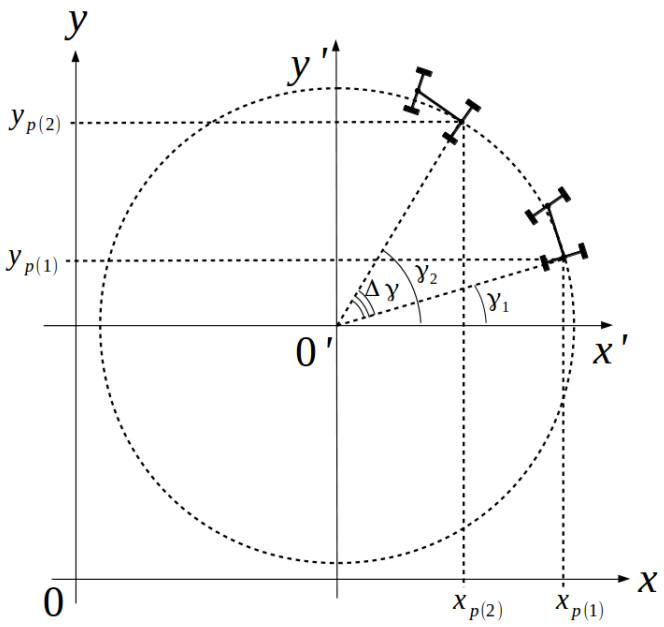

Рисунок 1.10 – Схема обчислення нових координат при повороті рухомої платформи

Використовуючи складені рівняння можна легко прогнозувати зміну положення рухомої платформи в залежності від часу та керувальних сигналів.

1.6 Висновки

Отже, в цьому розділі були розглянуті загальні принципи побудови математичних моделей і складено математичну модель для чотириколісної рухомої платформи. Рівняння побудованої математичної моделі можна використовувати для передбачення зміни координат положення рухомої платформи в просторі, а також кута орієнтації в залежності від часу та керувальних сигналів.

2. АЛГОРИТМ ПОШУКУ ШЛЯХУ

Побудова шляху до заданої цілі є дуже важливою задачою в робототехніці. Ціль цієї задачі - побудувати гарний шлях, який дозволить оминати перешкоди і при цьому мінімізувати вагу цього шляху, зокрема, найчастіше потрібно мінімізувати такі параметри, як довжина шляху, час, кількість ресурсів, які необхідно витратити на його проходження. Також часто ставляться і додаткові вимоги, такі як рух якомога далі від перешкод і тому подібне.

Типовий приклад вирішення задачі побудови шляху - це алгоритм Дейкстри. В цьому алгоритмі спочатку обирається початкова вершина графа, а всі вершини, які з’єднані з нею, заносяться в множину відкритих вершин-кандитатів. На кожному кроці розглядається вершина з цієї множини з найменшою відстанню до старту. Потім вона заноситься в іншу множину - розглянутих вершин, а всі вершини поряд з нею, які ще не були при цьому розглянуті заносяться в множину відкритих вершин. Так повторюється доти, поки не буде розглянута кінцева вершина, до якої і треба прокласти шлях.

Алгоритм Дейкстри гарантує знаходження найкоротшого шляху [6], якщо такий існує, але, з іншого боку, він є доволі повільним. Його модифікацією є алгоритм A*, який вводить додаткову евристичну функцію, що дозволяє скерувати напрям пошуку шляху і, таким чином, значно скоротити кількість часу на цей пошук.

Зазначені вище алгоритми легко застосувати для голономних роботів і рухомих платформ. Голономна рухома платформа - це така, яка може миттєво рухатись в будь-якому напрямку. У випадку неголономних рухомих платформ, таких як чотириколісна машиноподібна платформа, яка і є основним предметом для досліджень в цій роботі, побудова шляху значно ускладнюється, так як, в загальному випадку, шляхи прокладені за домогою алгоритмів Дейкстри або A*

не є можливими для виконання, бо в них робиться припущення, що об’єкт може рухатись в будь-якому напрямку. При цьому, для неголономних платформ часто ставиться задача не тільки дістатися до зазначеної точки на місцевості, а також зупинитись в ній з заданою орієнтацією (наприклад при паркуванні). В такому випадку потрібно використовувати алгоритми, які враховують кінематику руху робота. Одним із найбільш відомих є RRT (Rapidly Exploring Random Trees).

У випадку відсутності перешкод, для чотириколісної рухомої платформи з паралельним рульовим керуванням, найкоротший шлях можна знайти, використовуючи криві Дубінса (коли можливий тільки рух вперед і повороти), або криві Рідса-Шеппа (коли також можливий і рух назад).

Основними недоліками таких алгоритмів як RRT, а також кривих Дубінса і Рідса-Шеппа є те, що вони є значно складніші з обчислювальної точки зору в порівнянні з A* і модифікаціями. Тому можна спробувати поєднати ці алгоритми, щоб спочатку шлях прокладався за допомогою більш швидкого алгоритму, який не враховує кінематику рухомої платформи, а вже потім модифікувати цей шлях, щоб він став можливим для виконання неголономними платформами.

2.1 Криві Дубінса

Використання кривих Дубінса дозволяє побудувати найкоротший шлях між двома точками для машини Дубінса [7]. Машина Дубінса - це така машина, яка може рухатись прямо і робити повороти ліворуч та праворуч з заданим радіусом. При цьому можливий рух тільки вперед. Ці рухи зручно описати за допомогою наступних позначень:

1) S - рух прямо.
2) R - поворот праворуч.
3) L - поворот ліворуч.

Лестер Дубінс у своїй роботі показав, що серед шести комбінацій елементарних рухів, одна з них обов'язково описує найкоротшу можливою траєкторією для машини з зазначеними вище обмеженнями. Множина цих комбінацій виглядає наступним чином:

{RSR, LSL, RSL, LSR, RLR, LRL} (27)

Щоб знайти найкоротший серед цих шляхів, потрібно обчислити довжину кожного з них, і вибрати той, в якого довжина є найменшою.

Для того, щоб рухома платформа змогла рухатись по цій траєкторії, необхідно знайти точки на координатній площині, де потрібно змінити один елементарний рух на інший.

Знайдемо спочатку точки дотику прямої до кіл для траєкторії RSR (аналогічним чином можна знайти точки і для траєкторії LSL):

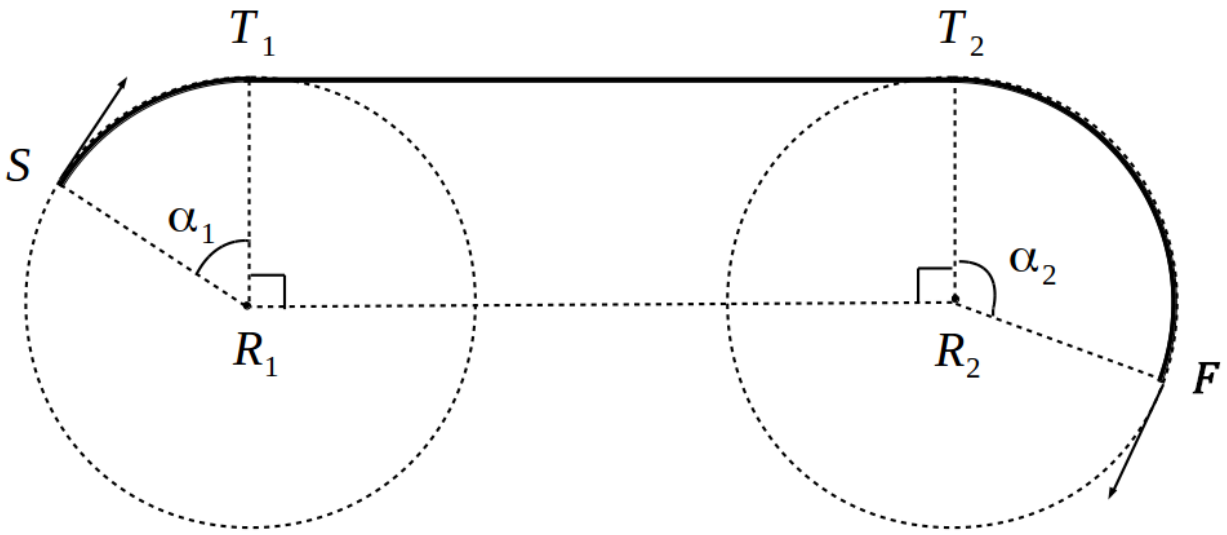

Рисунок 2.1 – Траєкторія для RSR кривої

Нам відомі координати точок S і F (початкове положення і кінцеве). Також, використовуючи алгоритм, наведений у розділі "Математична модель рухомої платформи" можна знайти координати центрів кіл поворотів (R_1 і R_2).

Використовуючи ці дані, потрібно знайти координати точок дотику прямої до кіл (T_1 і T_2).

Як відомо з геометрії, пряма, яка є дотичною до кола, утворює прямий кут з радіусом, проведеним у точку дотику. Зважаючи на це, а також на те, що радіуси обох кіл однакові, можна зробити висновок, що $R_1T_1T_2R_2$ - прямокутник, а отже пряма, яка проходить через точки R_1 та R_2 є паралельною до прямої T_1T_2.

Введемо наступні позначення:

$$\vec{u} = (u_x, u_y) = \overrightarrow{R_1T_1} \tag{28}$$

$$\vec{z} = (z_x,\ z_y) = \overrightarrow{R_1R_2} \tag{29}$$

Координати вектора $\vec{z}$ відомі. Вектор $\vec{u}$ є перпендикулярним до вектора $\vec{z}$, тому його координати можна знайти, використовуючи наступні формули:

$$u_x = -\frac{R_c}{|\vec{z}|} z_y \tag{30}$$

$$u_y = \frac{R_c}{|\vec{z}|} z_x \tag{31}$$

де R_c – радіус кола повороту.

Координати точки T_1 можна визначити таким чином:

$$T_{1x} = R_{1x} + u_x \tag{32}$$

$$T_{1y} = R_{1y} + u_y \tag{33}$$

Аналогічно можна визначити і координати точки T_2. При наявності координат точок T_1 і T_2 можна в потрібний момент часу задати платформі необхідні керувальні команди для слідування траєкторії RSR.

Використовуючи схожі техніки, можна також розрахувати і інші криві Дубінса.

2.2 Криві Рідса-Шеппа

Криві Рідса-Шеппа будуються за схожим принципом, як і криві Дубінса, але в них ще припускається, що платформа може також рухатись назад. Тобто в цьому випадку можна виділити шість базових рухів:

1) S^+ (рух вперед прямо)
2) R^+ (рух вперед з поворотом праворуч)
3) L^+ (рух вперед з поворотом ліворуч)
4) S^- (рух назад прямо)
5) R^- (рух назад з поворотом праворуч)
6) L^- (рух назад з поворотом ліворуч)

Внаслідок можливості руху назад, криві Рідса-Шеппа є більш практичними і часто дозволяють знайти значно коротший шлях в порівнянні з кривими Дубінса.

Рідс і Шепп у своїй роботі показали, що найкоротший шлях є однією із 48 можливих комбінацій елементарних рухів (пізніше було доведено, що буде достатньо використовувати 46 комбінацій) [8].

2.3 Алгоритм A*

A* - це інформований алгоритм пошуку, який при виборі наступної вершини для розгляду використовує спеціальну евристичну функцію, що надає оптимістичну оцінку шляху [9]. Цей алгоритм вперше був описаний Пітером Хартом, Нільсом Нільсоном і Бертрамом Рафаелем у 1968 році і позиціонувався як розширення для алгоритму Дейкстри.

В алгоритмі A* послідовно вибираются вершини для прокладення шляху, поки не буде досягнута цільова вершина. При цьому для вибору вершини використовується спеціальна функція:

$$f(n) = g(n) + h(n) \quad (34)$$

де g(n) - це відстань від стартової позиції до поточної, а h(n) - евристика, яка дає оптимістичний прогноз довжини шляху від поточної вершини до цільової. На кожному кроці вибирається вершина графа, у які функція f(n) приймає найменше значення.

В більшій частині випадків, завдяки використанню евристичної функції значно скорочується кількість вершин, які необхідно розглянути, в порівнянні з алгоритмом Дейкстри. Приклад наведено на Рисунку 2.2.

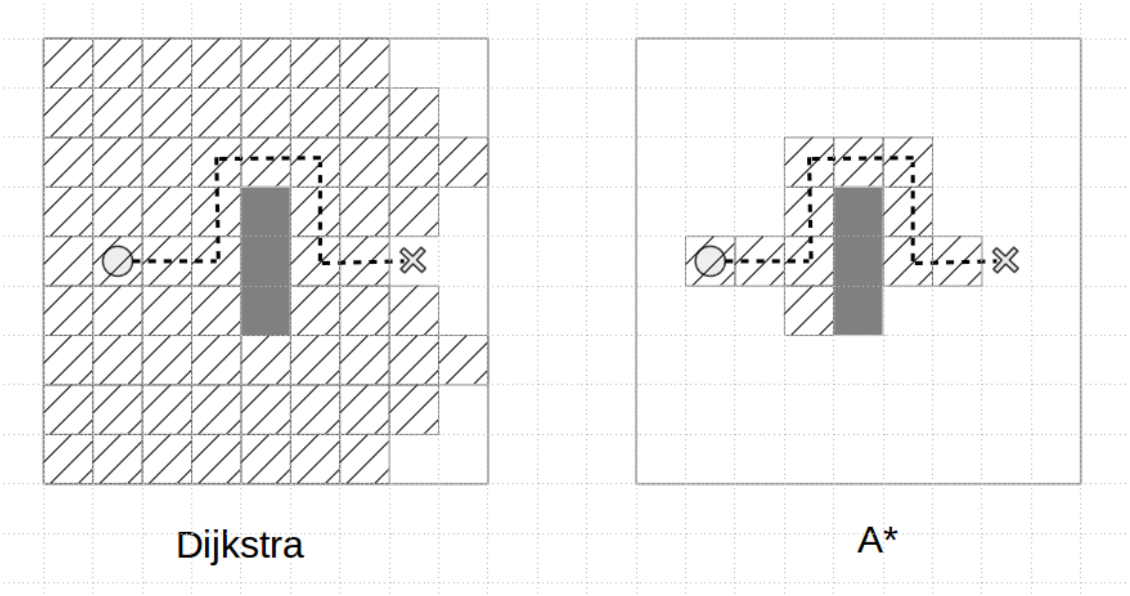

Рисунок 2.2 – Порівняння роботи алгоритмів пошуку Дейкстри і A* на однаковій карті (заштрихованим позначені розглянуті вузли)

Нище наведено псевдокод для алгоритму A*:

```
function A*(start, finish, grid):
```

```
    // створити дві множини: для відкритих вузлів і
закритих
    open_nodes = new Set()
    closed_nodes = new Set()

    // створити початковий вузол і додати його у множину
відкритих
    heuristic = compute_heuristic(start, finish)
    current_node = create_node(null, start, heuristic)
    open_nodes.add(current_node)

    while open_nodes is not empty:
        // пройтись по всіх сусідніх клітинках поточного
        //вузла
        for each neighbor of current_node.cell:


            // брати до уваги тільки ті клітинки,
            // які ще не були розглянуті
            if neighbor is not in open_nodes and
closed_nodes:
                heuristic = compute_heuristic(neighbor)
                node = create_node(current_node,
neighbor)
                open_nodes.add(node)

        // додати поточний вузол у закриті
        closed_nodes.add(current_node)
```

```
        // знайти і видалити вузол з множини відкритих
вузлів,
        // у якого мінімальним значенням gValue + hValue
        current_node = pop node with min(gValue + hValue)
from open_nodes

        // якщо новий поточний вузол і є цільовим -
        // побудувати і повернути шлях
        if (current_node.cell == finish):
            return reconstruct_path(current_node)


    return failure
}

// створити вузоз
function create_node(previous_node, cell, heuristic):
    gValue = previous_node.gValue + 1
    hValue = heuristic
    return new Node(previous_node, cell, gValue, hValue)

// обчислити евристику
// в даному випадку використовується
// манхеттенська відстань
function compute_heuristic(cell, finish):
    return |cell.x - finish.x| + |cell.y - finish.y|

// відновити шлях, використовуючи вузли
function reconstruct_path(last_node):
    path = new List()
```

```
    current_node = last_node
    while current_node != null:
        path.add_to_beginning(current_node.cell)
    return path

// у вузлі міститься інформація про попередній вузол,
// клітинку
// а також gValue (пройдена відстань)
// і hValue (евристика, оптимістична оцінка відстані до
// цілі)
class Node:
    Node previous
    Cell cell
    double gValue
    double hValue
```

Алгоритм A* дуже добре себе проявляє в задачах, у яких використовується манхеттенська відстань, згідно якої, відстань між двома точками дорівнює сумі модулів різниць її координат. Це ідеально підходить для багатьох комп'ютерних ігр і для тих випадків, коли рухомий об'єкт, для якого потрібно прокласти шлях, може рухатись тільки в чотирьох напрямках: вперед, назад, ліворуч і праворуч. Але реальні рухомі платформи і роботи часто можуть рухатись в будь-якому напрямку, тобто під будь-яким кутом. В цьому випадку потрібно використовувати евклідову відстань, тому шлях, прокладений алгоритмом A*, загалом не є оптимальним і часто виглядає неприродньо. Приклад такого шляху наведено на Рисунку 2.3.

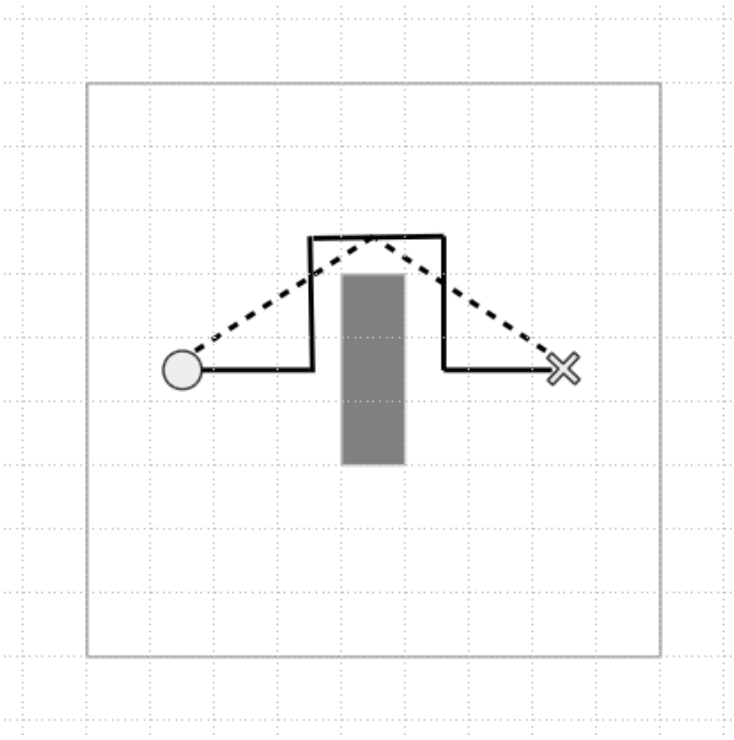

Рисунок 2.3 – Суцільною лінією позначений шлях, прокладений за допомогою алгоритму A*, пунктирною – шлях, який є значно коротшим при вимірюванні за допомогою евклідової відстані

Одним із варіантів вирішення цієї проблеми є використання евристичної функції, яка ґрунтується на евклідовій відстані і введення можливості діагональних рухів, а також подальше згладження траєкторії за допомогою спеціальних алгоритмів. Але побудова таких алгоритмів часто є доволі складною задачею, бо A* в загальному випадку не гарантує побудову траєкторії, яка після згладжування буде близькою до оптимальної при використанні евклідової відстані.

2.4 Алгоритм Theta*

При можливості руху під будь-яким кутом можна спробувати використовувати алгоритм Theta* [10], який є модифікацією A*. Основна різниця цього алгоритму з A* полягає в тому, що при додаванні наступної вершини, його батьківською вершиною може бути будь-яка з попередньо

побудованого шляху, а не тільки сусідня. Тобто, функція **create_node()**, з наведеного вище псевдокоду для A*, має виглядати таким чином:

```
function create_node(previous_node, cell, heuristic):
    chosen_parent = previous_node

    // попередній вузол попереднього вузла
    previous_node_parent = previous_node.previous
    if previous_node_parent != null:
        if line_of_sight(cell,
previous_node_parent.cell):
            chosen_parent = previous_node_parent

    gValue = chosen_parent.gValue +
euclidean_distance(chosen_parent.cell, cell)
    hValue = heuristic
    return new Node(chosen_parent, cell, gValue, hValue)
```

При цьому, функція **compute_heuristic()** тепер повинна використовувати евклідову відстань:

```
function compute_heuristic(cell, finish):
    return euclidean_distance(cell, finish)

function euclidean_distance(first_cell, second_cell):
    return sqrt((first_cell.x - second_cell.x)^2
            + (first_cell.y - second_cell.y)^2)
```

Важливим моментом в алгоритмі Theta* є функція **line_of_sight()**, яка визначає чи можна дістатися з однієї клітинки поля до іншої по прямій. Її

обчислювальна складність напряму впливає на час роботи алгоритму Theta*, так як вона викликається при додаванні кожного нового вузла. Тому дуже важливо, щоб реалізація цієї функції була ефективною.

Можна помітити, що функція `line_of_sight()` нагадує алгоритм для малювання лінії на растровому дисплеї. Тобто можна спочатку визначити клітинки, через які буде проходити ця лінія, і для кожної з них перевірити чи нема там перешкоди. Це показано на Рисунку 2.4.

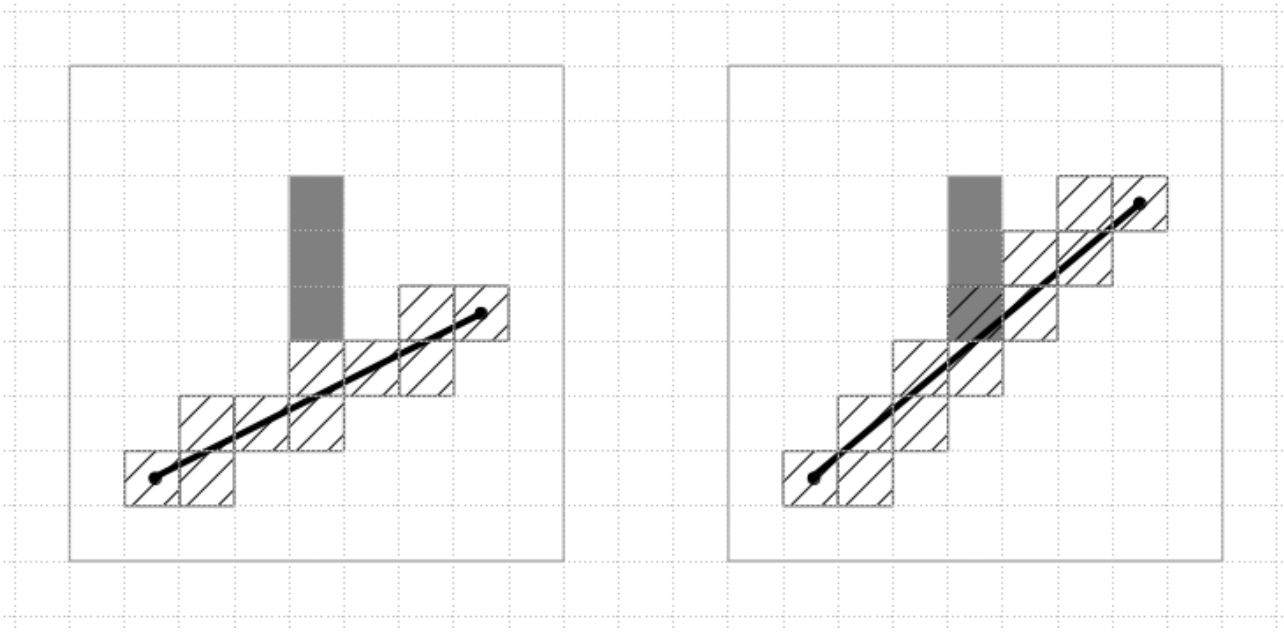

Рисунок 2.4 – На малюнку праворуч між двома точками існує прямий шлях, на малюнку ліворуч - не існує, бо пряма лінія, яка сполучає точки перетинає через перешкоду

На Рисунку 2.4 наведено випадок, коли товщина цієї лінії дорівнює розміру перешкоди, тобто одиниці. Це підходить тоді, коли розмір рухомої платформи або робота дорівнює розміру перешкод. Але в загальному випадку розміри перешкоди можуть бути значно менші за самого робота - і це бажано враховувати. Тому, часто є доцільним вважати, що розмір робота може дорівнювати декільком клітинкам. В таких випадках, для коректності роботи алгоритму Theta*, потрібно малювати лінію, товщина якої буде дорівнювати ширині рухомої платформи.

Лінію можна описати рівнянням $y = kx + b$. При цьому, клітинки, через які проходить лінія, відносно просто визначити за умови, що коефіцієнт $k <= 1$, що еквівалентно наступномій нерівності:

$$|x_2 - x_1| >= |y_2 - y_1| \quad (35)$$

де (x_1, y_1) - координати початкової клітинки $pixel_1$, (x_2, y_2) - координати кінцевої клітинки $pixel_2$.

Ця умова виконується не завжди. В тих випадках, коли нерівність хибна, можна спочатку зробити спеціальну трансформацію - замінити в початковій і кінцевій клітинках координати x і y місцями перед початком виконання алгоритму, і для визначених пікселів провести зворотню трансформацію. Також варто зазначити, що спочатку зручніше побудувати пікселі для лінії товщиною в одну клітинку, а вже потім додати ще пікселів зверху і знизу від лінії, щоб була потрібна товщина.

Таким чином, роботу алгоритму можна поділити на чотири етапи:

1) якщо потрібно, трансформувати координати початкової і кінцевої клітино, щоб виконувалась нерівність, зазначена вище
2) визначити клітинки (або пікселі) для лінії шириною в одну клітинку
3) збільшити товщину лінії при необхідності
4) якщо була проведена трансформація під час першого етапу - для отриманих клітинок, через які проходить лінія, провести зворотню трансформацію координат x і y.

Розглянемо детальніше пункт 2), а саме алгоритм для побудови ліній, товщиною в одну клітинку, при виконанні умови (35). Зробимо також припущення, що $x_2 > x_1$ (якщо це не виконується, то можна просто змінити початкову і кінцеву клітинки місцями). Для визначення клітинок, через які

проходить лінія необхідно здійснити $|x_2 - x_1|$ ітерацій, тобто пройтись по всім проміжним цілочисельним значенням осі x, які належать проміжку $[x_1, x_2]$

Будемо вважати, що лінія має проходити через середини початкового і кінцевого пікселів. На кожній ітерації спочатку необхідно визначити координати точок x_i і y_i:

$$x_i = x_{i-1} + 1 \quad (36)$$

$$y_i = y_{i-1} + k \quad (37)$$

де k - це коефіцієнт з рівняння прямої $y = kx + b$, x_i - це ціле число, яке знаходиться на межі двох клітинок, y_i - дробове. Початкові значення x_0 і y_0 можна визначити таким чином:

$$x_0 = pixel_{1x} \quad (38)$$

$$y_0 = pixel_{1y} \quad (39)$$

де (x_0, y_0) - це точка, яка лежить на прямій $y = kx + b$, і при цьому знаходиться на лівій межі початкової клітинки $pixel_1$

На кожній ітерації можливе виконання однієї із умов:

1) y_i знаходиться далеко від меж клітинок
2) y_i знаходиться на межі клітинок, тобто y_i є цілим числом.

В першому випадку до множити клітинок, через яку проходить лінія, необхідно додати дві точки, як зображено на Рисунку 2.5.

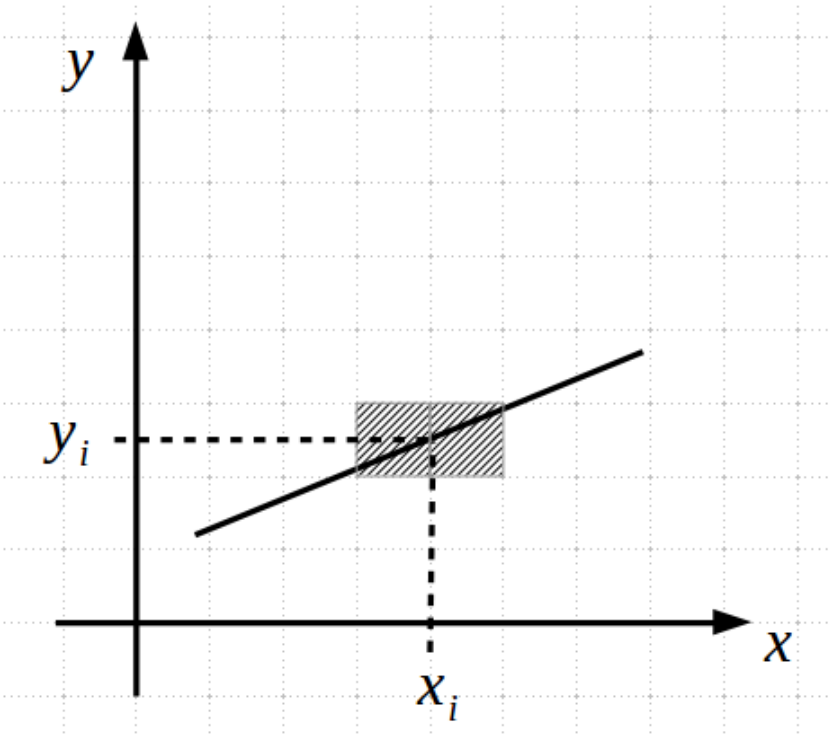

Рисунок 2.5 – Ітерація, на якій необхідно додати дві клітинки

В другому випадку необхідно додати чотири клітинки, як зображено на рисунку 2.6.

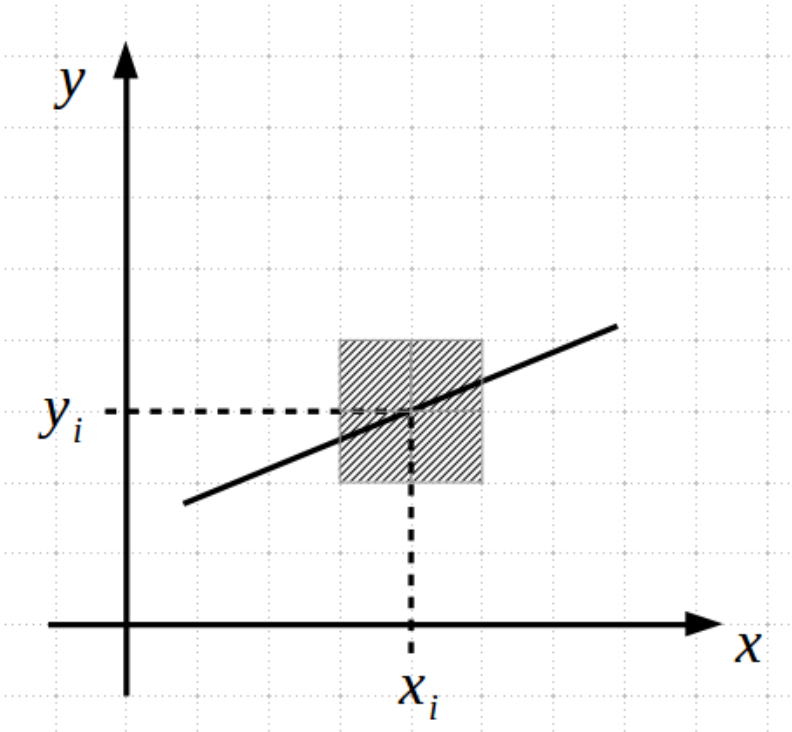

Рисунок 2.6 – Ітерація, на якій потрібно додати чотири клітинки

При цьому, деякі клітинки будуть додаватись по декілька разів на різних ітераціях. Цього можна уникнути, якщо для зберігання пікселів використовувати структуру даних, яка ігнорує повтори.

Псевдокод для даного алгоритму:

```
function draw_line_pixels(from, to, width):
        abs_delta_x = |to.x - from.x|;
        abs_delta_y = |to.y - from.y|;

        // клітинки зберігатимуться в структурі даних,
яка ігнорує повтори
        pixels = new Set()

        // клітинка "to" має бути праворуч від клітинки
"from"
        if (from.x > to.x) {
            temp = from;
            from = to;
            to = temp;
        }

        // точки будуть визначатись за допомогою рівняння
прямої y = kx + b
        // ratio == k
        ratio = (to.y - from.y) / abs_delta_x;

        // початкове значення y (на лівій межі клітинки
"from")
        y = from.y + 0.5 - ratio * 0.5;
```

```
        // кількість ітерацій
        iters = to.x - from.x;

        for (i = 0; i < iters; i++) {
            y += ratio;

            x_left_integer = from.x + i;
            x_right_integer = x_left_integer + 1;
            if (almost_integer(y)) {
                // необхідно додати чотити клітинки
                y_upper_integer = round(y);
                y_bottom_integer = y_upper_integer - 1;
                pixels.add(new Pixel(x_left_integer,
y_bottom_integer));
                pixels.add(new Pixel(x_left_integer,
y_upper_integer));
                pixels.add(new Pixel(x_right_integer,
y_bottom_integer));
                pixels.add(new Pixel(x_right_integer,
y_upper_integer));
            } else {
                // необхідно додати дві клітинки
                int y_integer = (int) floor(y);
                pixels.add(new Pixel(x_left_integer,
y_integer));
                pixels.add(new Pixel(x_right_integer,
y_integer));
            }
        }
    return adjust_line_width(pixels, width);
```

Після того, коли визначені пікселі для лінії з товщиною “1”, необхідно додати ще пікселів зверху і знизу від лінії, щоб була досягнута необхідна товщина. Для цього необхідно спочатку згрупувати отримані пікселі за координатою x (при цьому в кожній групі буде або один або два пікселі), і після цього додати зверху і знизу ще пікселів, користуючись наступними правилами:

1) якщо $w = 1$, то нічого не робити, бо лінія і так має товщину “1”

2) якщо w - непарне, додати зверху і знизу по $\frac{(w-1)}{2}$ пікселів

3) якщо w - парне і $n_x = 1$ - додати зверху і знизу по $\frac{w}{2}$ пікселів

4) якщо w - парне і $n_x = 2$ - додати зверху і знизу по $\frac{(w-2)}{2}$ пікселів

де w - це потрібна товщина лінії ($w >= 1$),

n_x - кількість побудованих пікселів алгоритмом наведеним нище для заданого x

Псевдокод для цієї функції вигладає наступним чином:

```
function adjust_line_width(input_pixels, width)

    // згрупувати пікселі по координаті x
    // для легшого визначення додаткових пікселів
    pixels_grouped_by_x = groupe_and_order_by_x(piexles)

    // пікселі для лінії з правильною шириною
    output_pixels = new List()
    for each vertical_group in pixels_grouped_by_x:
        min_y
        max_y
        x = vertical_group.get(0).x
```

```
        if vertical_group.size == 1
            // в цьому випадку в групі тільки один
піксель
            pixel = vertical_group.get(0)
            adjusted_width = width % 2 == 0 ? width + 1 :
width
            min_y = pixel.y - (adjustedWidth - 1) / 2
            max_y = min_y + adjustedWidth - 1

        else:
            // в цьому випадку в групі два пікселі:
нижній і верхній
            bottom_pixel = vertical_group.get(0)
            adjusted_width = width % 2 == 0 ? width :
width + 1
            min_y = bottom_pixel.y - (adjustedWidth - 1)
/ 2
            max_y = min_y + adjustedWidth - 1;

    // додавання пікселів зверху та знизу від лінії для
поточної групи
    // щоб досягнути заданої товщини
    for (y = min_y; y <= max_y; y++)
        output_pixels.add(new Pixel(x, y))

    return pixelsWithProperLineWidth;
```

Запропонована реалізація алгоритму побудови лінії працює за лінійний час, що є прийнятним.

Після визначення клітинок, через які проходить лінія, що сполучає дві точки, необхідно для кожної з цих клітинок визначити чи нема в ній перешкод. Якщо для зберігання карти перешкод використовувати структуру даних, яка виконує пошук елементів за константний час, то сумарна складність алгоритму визначення чи існує прямий шлях між двома клітинками також буде $O(n)$. При використанні двовимірного масиву для зберігання перешкод, пошук по двох індексах (координатах $x\ i\ y$) буде константним, що задольняє вимоги. Але, так як кількість перешкод в більшій частині випадків значно менша ніж кількість пустих клітинок, то краще використовувати якусь множину, яка застосовує хешування, бо в такому випадку час визначення чи є на клітинці перешкода також буде лінійним, але при цьому буде потрібно значно менше оперативної пам’яті.

2.5 Гібридний A*

Алгоритми A* і Theta* є доволі швидкими, але вони підходять для рухомих платформ, які здатні розвернутись на місці. Але для неголономних рухомих платформ, в яких відсутня можливість розвертунись на місці і є специфічна кінематика руху, ці алгоритми в загальному випадку не можна застосувати, бо вони будують неможливі для виконання траєкторії. Приклад порівнянн траєкторії, побудованої алгоритмом Theta* і реальної траєкторії платформи, в якій враховані особливості кінематики руху цієї платформи наведено на Рисунку 2.7.

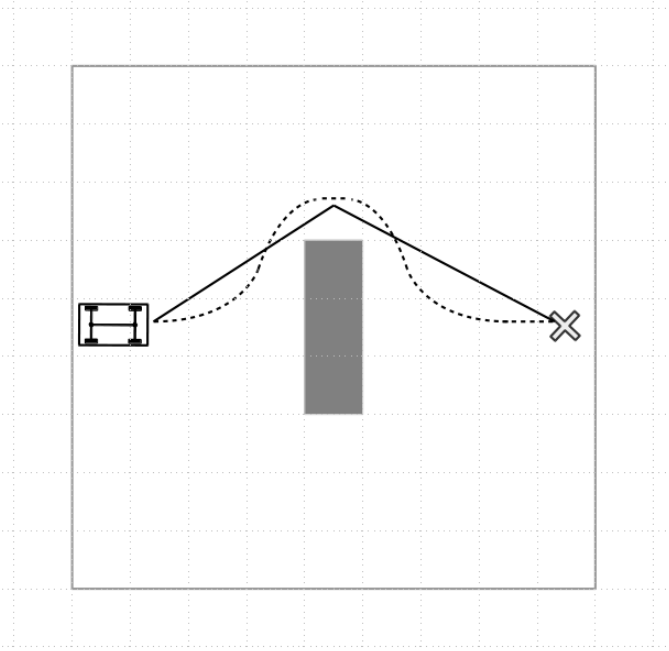

Рисунок 2.7 – Суцільної лінією позначена траєкторія, побудована алгоритмом A*, пунктирною – траєкторія, реальна для виконання рухомою платформою

У випадку неголономних платформ потрібно застосовувати алгоритм, який враховує їхню кінематику. Гібридний A* є саме таким алгоритмом [11]. Його основна ідея полягає в тому, що рухи платформи можуть виконуватись тільки відповідно до математичної моделі, і кожна клітинка зберігає інформацію не тільки про те, чи була вона відвідана, чи ні, а також кут орієнтації платформи, під яким вона відвідала цю клітинку. При цьому для кожної ітерації потрібно зберігати також повноцінний стан платформи, а саме дійсні координати x і y (а не тільки координати клітинку) а також точний кут орієнтації платформи. Таким чином, в кожному стані алгоритм «Гібридний A*» розглядає всі можливі рухи об’єкту і оцінює кожен із них. Для чотириколісної рухомої платформи розгляд цих рухів схожий на побудову дерев. Приклад такого дерева, побудованого за допомогою алгоритму «Гібридний A» показано на Рисунку 2.8.

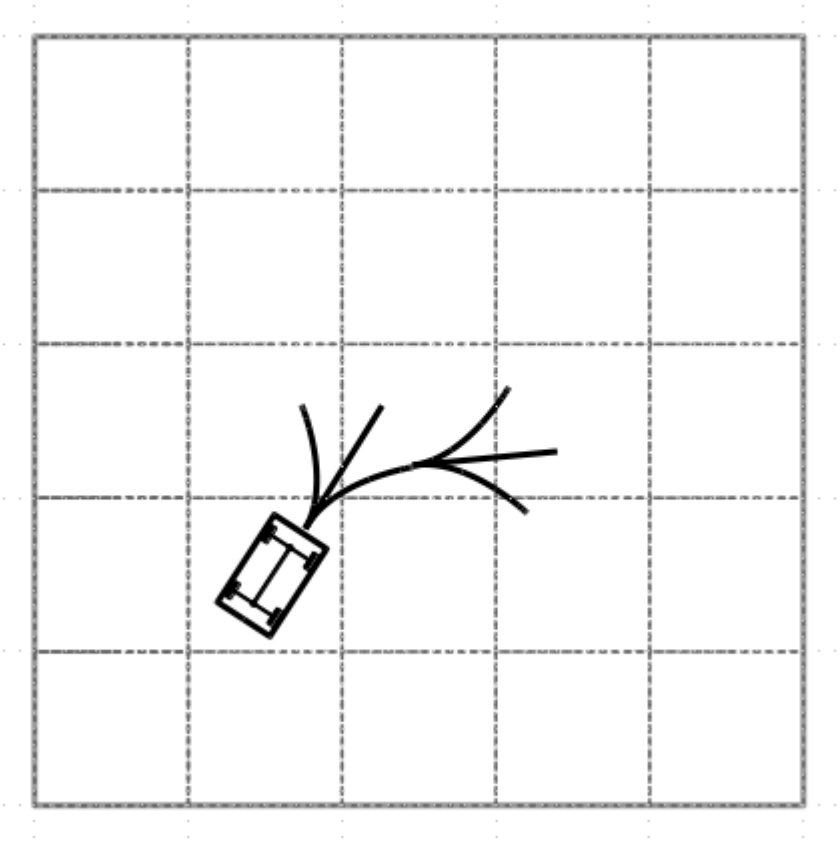

Рисунок 2.8 – Дерево можливих рухів платформи, побудоване алгоритмом «Гібридний А*»

При цьому, змінюється також функція для обчислення ваги (cost) кожного стану, яка впливає на вибір наступного стану для прокладання шляху (з відкритої множини станів наступним вибирається той, в якого найменша вага). Одним із найпростіших варіантів реалізації цієї функції є такий:

$$cost = g\_value + heuristic + cell\_weight \quad (40)$$

де g_value - це відстань, яка вже пройдена, щоб досягти поточного стану, $heuristic$ - оптимістична оцінка відстані до цілі, $cell\_weight$ - вага клітинки, яка є великою, якщо платформа вже була в цій клітинці під таким самим кутом і маленька якщо платформа була в клітинці під іншим кутом, або не була взагалі.

Але в такій реалізації є один недолік - побудована траєкторія може мати дуже багато непотрібних поворотів, як це зображено на Рисунку 2.9

Рисунок 2.9 – Траєкторія, побудована алгоритмом «Гібридний A*» з простою cost-функцією

Для уникнення цього, при обчисленні ваги треба також враховувати скільки було зроблено поворотів, а також змін напрямку руху на протилежний. Окрім цього, можна збільшувати вагу при русі назад, адже в основній частині випадків рух вперед є більш доцільним.

Також потрібно зауважити, що коли платформа знаходиться далеко від цілі, і на шляху до цілі практично немає перешкод, то найбільш оптимальною є стратегія, коли рух відбувається вздовж лінії, яка сполучає початкову точку (старт) і кінцеву точку. Для досягнення цього можна ввести ще два доданка для обчислення ваги стану - перший є пропорційним відстані між платформою і прямою лінією, що сполучає початкову і кінцеву точки, а другий - пропорційний куту між орієнтацією платформи і тією ж лінією. При цьому, коли платформа знаходиться близько біля цілі, то ці два доданки вже враховувати не потрібно, адже вони будуть тільки заважати зупинитись рухомій платформі під правильним кутом, тобто припаркуватися. В свою чергу,

цього можна досягти, якщо зробити вагу описаних вище додатків пропорційною до відстані між поточним положенням платформи і кінцевою точкою.

Це все враховується в псевдокоді для обчислення ваги стану платформи:

```
function compute_state_cost(car, movement,
previous_state, start, target)
        // відстань до цілі
        distance_to_target = distance_between(car,
target)

        // вага клітинки, яка залежить від того,
        // чи була платформа в цій клітинці
        // і під яким кутом
        cell_weight = compute_cell_weight(car)

        // якщо платформа рухається назад, то додати вагу
        move_backward_weight = movement.is_backward() ?
                                BACKWARD_MOVE_WEIGHT : 0

        // якщо платформа змінила напрям руху, то додати
вагу
        change_steering_weight = movement !=
previous_state.movement ?
                                CHANGE_STEERING_WEIGHT :
0

        // додаткова вага, яка залежить від того,
        // наскільки далеко знаходиться платформа
```

```
        // від прямої лінії, що сполучає початкову та
кінцеву точки
        // при цьому, ця вага майже не береться до уваги,
        // якщо платформа знаходиться близько до цілі
        shift_error = |distance_between(car, line(start,
target))
        shift_error_weight = SHIFT_ERROR_COEF * distance
to target

        // додаткова вага, яка залежить від того,
        // наскільки відрізняється кут орієнтації
платформи
        // від кута прямої лінії, що сполучає початкову
та кінцеву точки
        // при цьому, ця вага майже не береться до уваги,
        // якщо платформа знаходиться близько до цілі
        orientation_error = |angle_between(car,
line(start, target))
        orientation_error_weight = ORIENTATION_ERROR_COEF
* distance_to_target
        // сумарна вага для стану
        cost = = distance_to_target + previous_distance +
cell_weight
                    + move_backward_weight +
change_steering_weight
                    + shift_error * shift_error_weight
                    + orientation_error *
orientation_error_weight
        return cost;
```

В результаті використання такої функції для обчислення ваги стану, шлях, прокладений за допомогою алгоритму “Гібридний A*”, стає дуже подібним на оптимальний. Це зображено на Рисунку 2.10.

Рисунок 2.10 – Шлях, прокладений алгоритмом «Гібридний A*» з покращеною cost-функцією.

Важливим моментом для роботи наведеної вище функції є обчислення ваги клітинки. Для визначення цієї ваги потрібно зберігати масив клітинок, в яких буде інформація про те, скільки разів вже платформа була в цій клітинці і під яким кутом вона тут була. При цьому вага клітинки буде складатись з двох компонент, перший з яких не залежить від орієнтації платформи в клітинці (цей компонент не повинен сильно впливати на результат) і компоненту який залежить від орієнтації (його вплив має бути значним, щоб не допускати однакових станів рухомої платформи у клітинці, бо вони тільки ускладнюють обчислення). Другий компонент можна обчислювати двома шляхами:

1) Повністю дискретизувати всі можливі кути орієнтації платформи на відповідні проміжки, і при додаванні нового стану зберігати не справжній кут орієнтації платформи, а дискретизований (визначати проміжок, до якого входить цей кут) При цьому треба правильно визначити на кількість цих дискретних проміжків.

2) Іншим варіантом є зберігання справжніх значень кутів орієнтації платформи, і при обчисленні ваги клітинки використовувати неперервну вагову функцію для визначення наскільки близько лежить новий кут орієнтації платформи до вже існуючих. Як вагову функцію можна використовувати функцію розподілу Гауса з центром у точці зі значенням нового кута. При цьому попередні кути потрібно модифікувати таким чином, щоб вони лежали у проміжку $[new\_angle - \pi, new\_angel + \pi]$ Схематично це зображено на Рисунку 2.11.

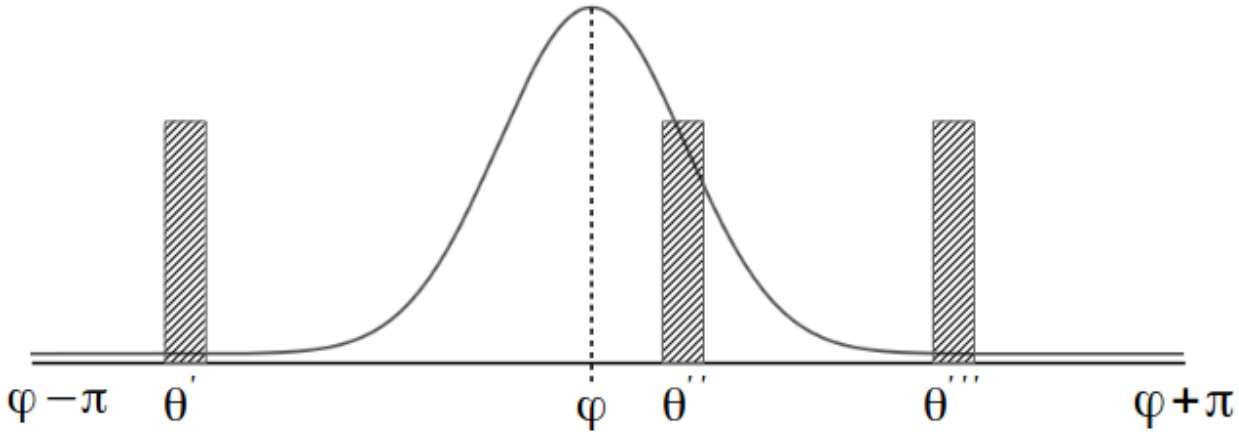

Рисунок 2.11 – $\theta', \theta'', \theta'''$ - кути орієнтації платформи, під якими вона була в клітинці в попередні рази. ϕ - поточний кут орієнтації платформи в клітинці. Як видно з графіку розподілу Гауса - на вагу сильно вплине θ'' і майже не вплинуть θ' і θ'''

Псевдокод для обчислення ваги другим способом наведено нище:

```
function compute_cell_weight(car)
    // спочатку треба трансформувати попередні кути,
    // щоб вони були у проміжку [car.angle - PI,
car.angle + PI]
    previous_angles = previous_angles_in_range(car.angle
- PI, car.angle + PI)
```

```
    // функція розподілу Гауса для визначенн ваги
    // попередніх кутів орієнтації платформи.
    //
    // мат. сподівання цієї функції дорівнює поточному
куту
    // середньоквадратичне відхилення задається
константою STD_DEV
    //
    // попередні кути які лежать в межах [mean - stddev,
mean + stddev]
    // мають значний вплив, а інші - менший. при цьому
чим далі
    // знаходиться попередній кут від поточного - тим
менший він має вплив
    //
    // чим менше значення має константа STD_DEV, - тим
більше дозволено
    // нових положень в клітинці, які не будуть мати
велику вагу
    gauss = new Gauss(mean = car.angle, stddev = STD_DEV)

    weight = 0;
    for each angle in previous_angles:
        weight += gauss.of(angle)

    return weight.
```

Ще однією значною перевагою алгоритму “Гібридний A*” є те, що він дозволяє рухомій платформі зупинятись у кінцевій точці під правильном

напрямком. Іншими словами, його можна застосовувати для вирішення задачі паркування. Приклад цього зображено на Рисунку 2.12.

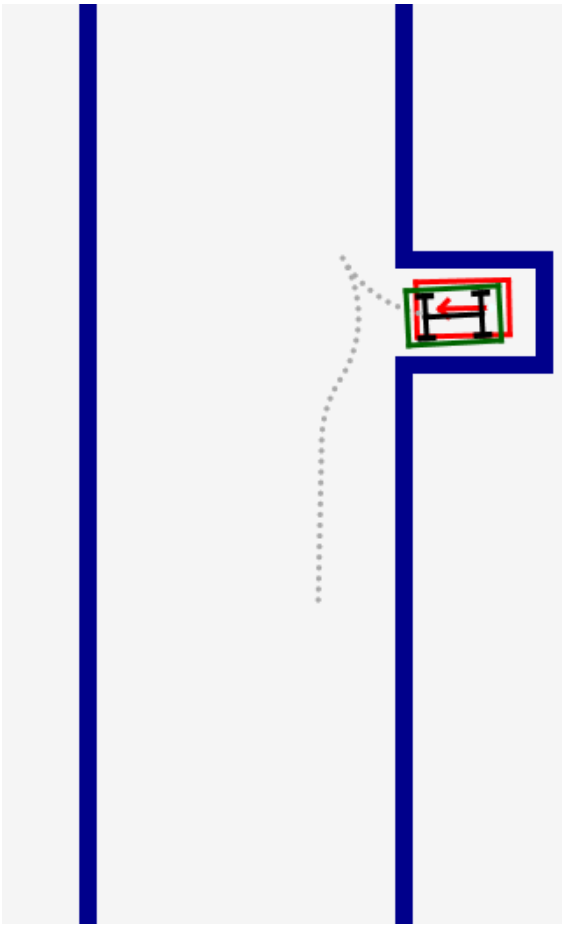

Рисунок 2.12 – Паркування

Гібридний A* будує траєкторії, які є можливими для виконання для рухомих платформ, але він виконує значну кількість обчислень. У випадках великих карт, а особливо, якщо при цьому на них наявні U-подібні перешкоди, даних алгоритм буде працювати дуже повільно, що часто є неприйнятним.

2.6 Поєднання алгоритмів Theta* і гібридного A*

Алгоритм Theta* працює доволі швидко, але не враховує кінематики рухомої платформи. З іншого боку гібридний A* працює відносно повільно, але будує траєкторії, які є можливими для виконання. Тому, можна спробувати

поєднати ці алгоритми, щоб в результаті мати можливість швидкої побудови шляху, в якому, при цьому, враховується кінематика платформи.

Це поєднання можна зробити таким чином:

1) Спочатку визначити траєкторію (множину сполучених прямих, які уникають перешкод і ведуть від старту до кінцевої позиції) за допомогою алгоритму Theta*
2) Запускати алгоритм гібридний A* на кожній прямій з побудованої траєкторії. Кожна з цих прямих не перетинає перешкоди, тому алгоритм “Гібридний A*” має працювати відносно швидко.

Також варто зазначити, що при досягненні наступної точки з траєкторії, побудованої за допомогою Theta*, бажано щоб в рухомої платформи був зручний кут орієнтації. Для того, щоб траєкторіє була більш плавною, бажано щоб вектор орієнтації платформи у будь-якій точці траєкторії був направлений в бік наступної точки. З іншого боку, найлегше досягнути цієї точки у напрямком вектору, який сполучає попередню точку з поточною. Тому, для досягнення усередненого напряму можна просто додати ці два вектори. Цього можна досягти просто додавши два вектори. Це зображено на Рисунку 2.13.

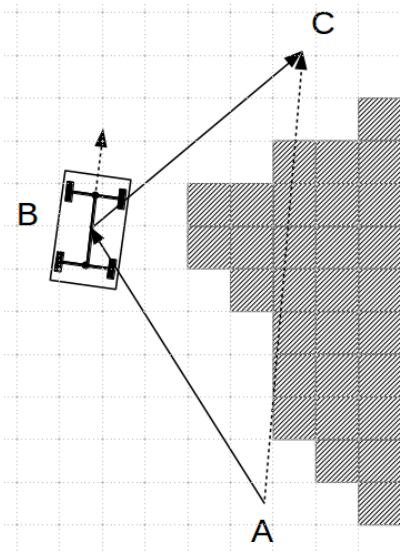

Рисунок 2.13 – Визначення потрібного напряму в точці B

Ще одним нюансом, який треба розглянути, є те, що при великій довжині кроку і високій точності для кінцевої позиції, алгоритм “Гібридний A*” може виконувати дуже багато ітерацій, які дуже повільно будуть наближувати до правильного розв’язку задачі пошуку шляху. Ефект від цього можна зменшити, реалізувавши два додатковий кроки:

1) Для проміжних точок шляху вказувати меншу точність для позиції і орієнтації платформи, адже в них цей критерій значно менше важливий, ніж для кінцевої точки.
2) Накладати обмеження на кількість ітерацій в алгоритмі “Гібридний A*”. Коли кількість ітерацій перевищує задану, необхідно зробити випадковий рух платформою і заново перезапустити алгоритм. В такому випадку збільшується ймовірність вдалого початкового розташування для роботи A*.

Псевдокод для поєднання алгоритмів Theta* і “Гібридний A*” наведено нище:

```
function resolve_path(start, target)

    // список з визначеними рухама рухомої платформи
    // для досягнення заданої цілі
    movements = new List()

    max_attempts_to_restart = 10
    attempts_to_restart = 0

    current_start = start;

    while attempts_to_restart < max_attempts_to_restart:

        try:
```

```
            // визначення шляху за допомогою алгоритму
Theta*
            theta_star_path =
compute_theta_star_path(current_start, target);


            for (i = 1; i < theta_star_path.size - 1;
i++):

                // попередня, поточна і наступна точки з
шляху theta*
                previous = theta_star_path.get(i - 1)
                current = theta_star_path.get(i)
                next = theta_star_path.get(i + 1)

                // визначення поточної цільової позиції і
орієнтації
                target_direction = new Vector(previous,
current)
                                        + new Vector(current,
next))

                target_position = current

                current_movements =
hybrid_a_star(current_start,
                                              target_position,
target_direction,

INTERMEDIATE_POSITION_ERROR,
```

```
INTERMEDIATE_DIRECTION_ERROR,
                                        MAX_ITERATIONS)

                movements.add(current_movements)

                // оновити значення поточної позиції
                current_start =
update_current_start(movements)


            target_position = target.position
            target_direction = target.direction

            // для кінцевого запуску алгоритму Гібридний
A*
            // помилка в позиції і напрямку має бути
меншою
            // ніж для проміжних запусків
            final_movements =
hybrid_a_star(current_start, target_position,
target_direction, FINAL_POSITION_ERROR,
FINAL_DIRECTION_ERROR, MAX_ITERATIONS)

            movements.add(final_movements)

        catch TooManyIterationsException:

            // якщо була перевищена максимальна кількість
ітерацій
```

```
                // потрібно зробити випадковий рух
                // і перезапустити пошук шляху
                current_start =
make_random_movement(current_start)

                attempts_to_restart++

    return movements
```

В результаті поєднання цих двох алгоритмів, пошук шляху відбувається доволі швидко навіть на великих картах і при наявності U-подібних перешкод.

Приклад оминання U-подібних перешкод наведено на Рисунку 2.14

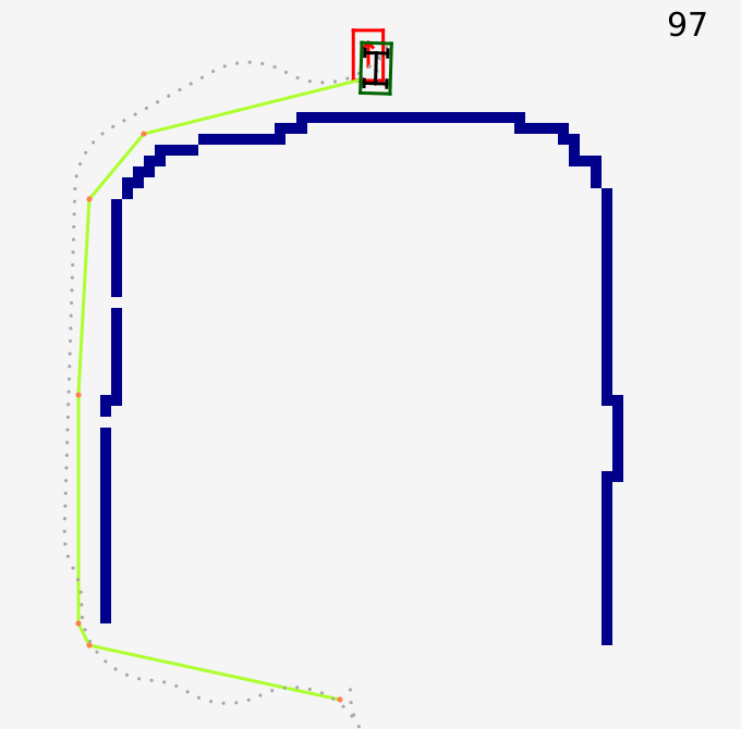

Рисунок 2.14 – Приклад оминання U-подібних перешкод. Зеленим кольором позначена траєкторія, побудована за допогомогою алгоритму Theta*. Пунктирною лінією позначено реальну траєкторію рухомої платформи

Також алгоритм швидко працює і на більш складних картах з великою кількістю перешкод. Приклад цього наведено на Рисунках 2.15 і 2.16.

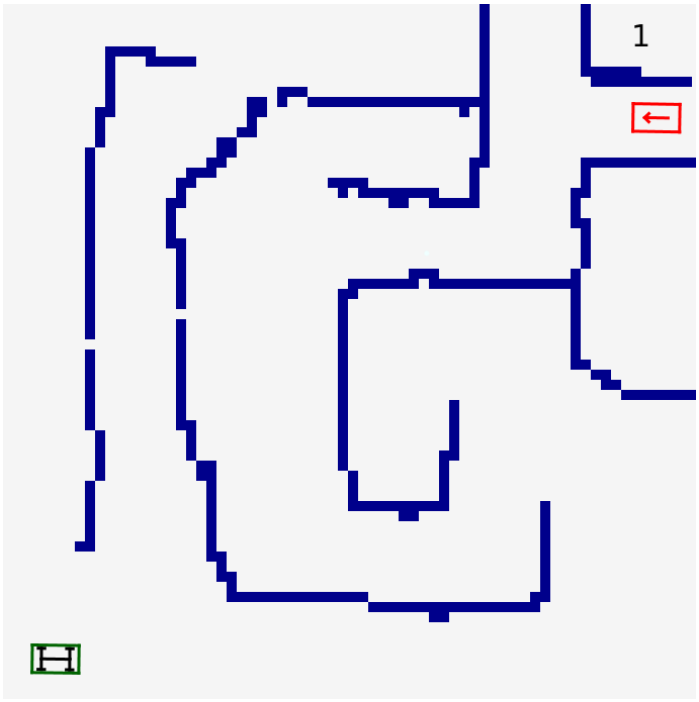

Рисунок 2.15 – Більш складна карта

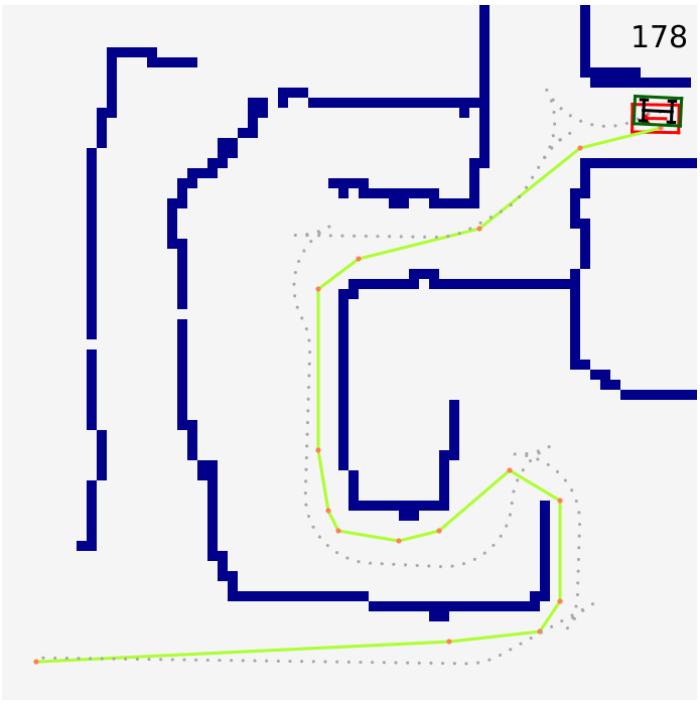

Рисунок 2.16 – Знайдений шлях на більш складній карті

2.7 Висновки

Отже, в цьому розділі було досліджено основні алгоритми пошуку шляху і оминання перешкод. Класичні алгоритми пошуку, такі як A* і Theta* будують траєкторії, неможливі для виконання рухомою платформою. З іншого боку, алгоритм «Гібридний A*» враховує кінематику платформи, але працює досить повільно на складних картах. В результаті, було прийнято рішення поєднати два алгоритми: Theta* і гібридний A* для збільшення продуктивності пошуку шляху і будування траєкторій, в яких врахована кінематика платформи.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ І МАТЕМАТИЧНОЇ МОДЕЛІ РУХОМОЇ ПЛАТФОРМИ

Реалізацію математичної моделі рухомої платформи та алгоритмів пошуку і прокладання шляху можна поділити та пункти:

1) Вибір мови програмування на необхідних бібліотек.
2) Проектування високорівневої архітектури.
3) Написання коду.
4) Тестування програмної системи.

Важливим є кожен з пунктів, але для правильної реалізації кожного з них необхідно керуватись відповідними принципами.

3.1 Вибір мови програмування, графічного фреймворку та бібліотек

При виборі мови програмування необхідно керуватись такими характеристиками:

- легкість написання коду
- швидкодія мови
- необхідна кількість ресурсів.

Так як описані вище алгоритми і математична модель потребує значної кількості обчислень, то однією з найважливіших характеристик мови є швидкодія. З іншого боку, необхідно щоба мова програмування була достатньо високорівнева, щоб можна було сконцентруватись на самих алгоритмах і математичній моделі рухомої платформи, без необхідності витрачати багато

часу на сторонні задачі, пов’язані з самостійним виділенням і очищенням пам’яті, роботи з вказівниками і іншими низькорівневими задачами. Також бажано, щоб мова була компільована, адже це дозволяє униктути значної кількості помилок. Окрім цього, значною перевагою буде наявність стандартних графічних фреймворків, а також велика кількість бібліотек, адже це може значно пришвидшити розробку програмного продукту.

Необхідними характеристиками володіє мова програмування Java. Java - це високорівнева, об’єктно-орієнтована мова прогамування, вперше випущена компанією Sun Microsystems у 1995 році. Java працює на віртуальній машині, що робить цю мову кросплатформовою [12].

У Java є два основних графічних фреймворка: Swing і JavaFX. JavaFX є значно новішою технологією, яка надає сучасні елементи графічного інтерфейсу і зручне API для їх використання. Тому в якості графічного фреймворку для реалізація програмної системи була обрана саме технологія JavaFX.

Для збирання проекту було обрано Maven. В порівнянні з конкуруючими системами Maven є доволі швидким і надає зручні засоби для підключення сторонніх бібліотек, а також керуваннями фазами життя проекту (такими як компілювання, очищення, тестування, запуск і т.д.)

У якості системи контролю версій було обрато Git, а також GitHub у якості віддаленого репозиторію. Git є розподіленою системою контролю версій, яка є доволі легкою в освоєнні, але при цьому надає велику кількість функціоналу. Основним поняттям в системі контролю версій Git є коміт, що означає “знімок” файлів проекту в якийсь певний момент часу.

Для пришвидшення і полегшення розробки програмного коду було також використано інтегроване середовище розробки Intellij IDEA, яке в порівняння з конкуруючими системами містить розумну систему для автодоповнення коду,

значну кількість вбудованих механізмів для автогенерації коду, який часто повторюється, а також велике число засобів для автоматичного рефакторінку коду.

Окрім цього було використано три сторонні бібліотеки:

- Guava
- Apache Commons Math
- JUnit 4

3.2 Принципи розробки програмного забезпечення

При розробці програмної реалізації алгоритмів та математичної моделі рухомої платформи були застосовані такі основні принципи і підходи:

- Використання MVC-архітектури (тобто розділення компонент на модель, представлення і контролер.
- При можливості написання незмінних (immutable) класів. Тобто поля класів мають ініціалізуватись тільки один раз при створенні [13].
- Дотримання загальних принципів ООП:
    - інкапсуляція,
    - наслідування,
    - поліморфізм.
- Застосування принципів SOLID [14]:
    - принцип єдиного обов’язку,
    - принцип відкритості/закритості,
    - принцип підстановки Барбари Лісков,
    - принцип розділення інтерфейсу,
    - принцип інверсії залежностей
- Використання принципу DRY (код не повинен повторюватись).

- Використання TDD (Test Driven Development). Тобто спочатку потрібно писати тести, а вже потім реалізацію класів і функцій. Це дозволяє зробити програмне забезпечення значно більш надійним, а також більш легким для розуміння [15].
- Написання коду, який сам себе документує

3.3 Високорівнева діаграма класів та складових компонент

Перед початком написання програмного коду, необхідно спроектувати високорівневу архітектуру. Це дозволяє уникнути багатьох поширених помилок, а також отримати цілісне уявлення, як програмна система повинна виглядати. Після розробки загальної архітектури необхідно перейти до проектування конкретних класів, а вже після цього - до написання коду.

Високорівнева діаграма основних класів і складових компонент зображена на Рисунку 3.1

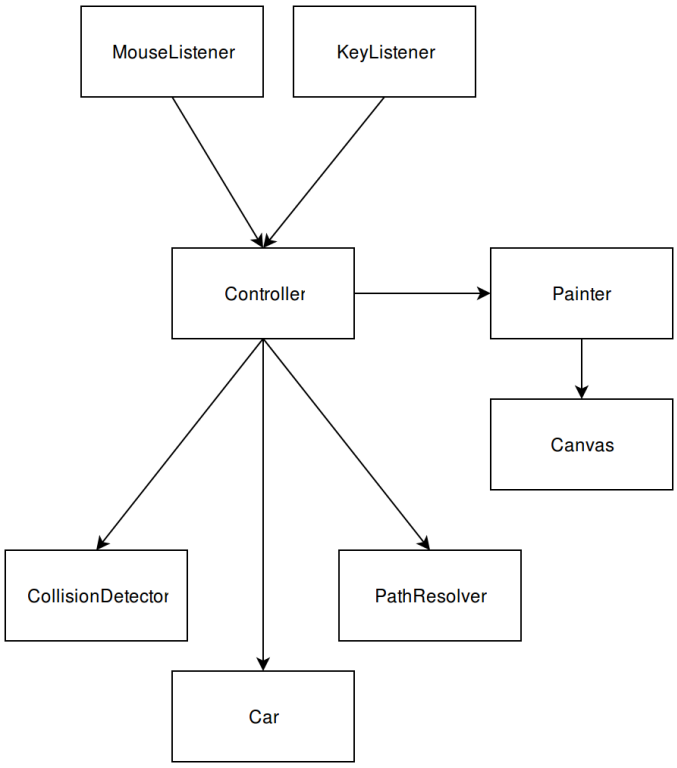

Рисунок 3.1 - Високорівнева діаграма класів і складових компонент

3.4 Тестування програмного продукту

Для тестування програмної системи було використано бібліотеку JUnit 4, яка надає зручні засоби для написання модульних тестів, а також систему збірки проектів Maven, яка дозволяє легко запускати потрібні тести.

Розробка більшості класів була виконана по методології TDD (Test Driven Development), основним правилом якої є те, що написання тестів має передувати написанню реалізації функціоналу.

Були написані модульні тести для математичної моделі, а також алгоритмів пошуку і виявлення колізій. Після програмної реалізації математичної моделі і основних алгоритмів всі тести були пройдені успішно.

Також було проведено мануальне тестування роботи програмної системи в цілому. Це тестування показало, що рухома платформа рухається коректно, а алгоритми пошуку шляху і прокладання траєкторії в більшій частині випадків працюють доволі швидко.

3.5 Висновки

Отже, в цьому розділі були досліджені основні принципи на підходи до розробки програмного забезпечення. Перед написанням програмного коду, було складено високорівневу діаграму основних класів і складових компонент. Після реалізації програмної системи її було протестовано.

4. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка основних характеристик програмного продукту – реалізації математичної моделі і алгоритмів пошуку для рухомої платформи Інтерфейс користувача був розроблений за допомогою мови програмування Java у середовищі розробки IntellijIDEA. Інтерфейс користувача створений за допомогою технології JavaFX.

Програмний продукт призначено для використання на персональних комп’ютерах під управлінням будь-якої операційної системи.

Нижче наведено аналіз різних варіантів реалізації модулю з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. Як прямі, так і побічні витрати розподіляються по продуктам та послугам у залежності від потрібних на кожному етапі виробництва обсягів ресурсів. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

– визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам:

ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

– для кожної функції визначаються повні річні витрати й кількість робочих часів.

– для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.

– після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

4.1 Постановка задачі

У роботі застосовується метод ФВА для проведення техніко-економічний аналізу розробки. Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

– програмний продукт повинен функціонувати на персональних комп’ютерах із стандартним набором компонент;

– забезпечувати високу швидкість обробки великих об’ємів даних у реальному часі;

– забезпечувати зручність і простоту взаємодії з користувачем або з розробником програмного забезпечення у випадку використовування його як модуля;

– передбачати мінімальні витрати на впровадження програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0– розробка програмного продукту, який аналізує процес за вхідними даними та будує його модель для подальшого

прогнозування. Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

F_1 – вибір мови програмування;

F_2 – вибір оптимальної СКБД;

F_3 – інтерфейс користувача.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) мова програмування Python;

б) мова програмування Java;

Функція F_2:

а) SQLite;

б) MySQL.

Функція F_3:

а) інтерфейс користувача, створений за технологією PyQt;

б) інтерфейс користувача, створений за технологією JavaFX.

4.3 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 4.1).

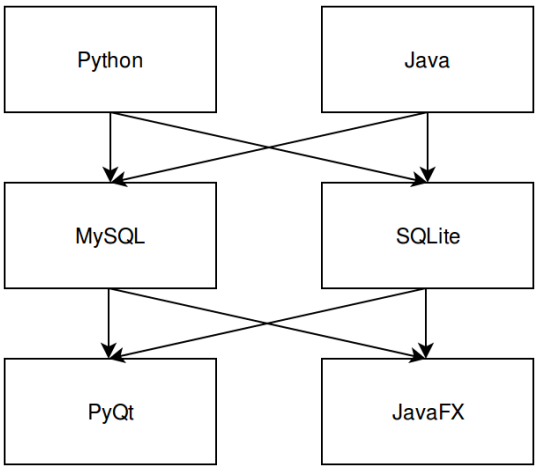

Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

Таблиця 4.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Кросплатформений	Низька швидкодія
	Б	Займає менше часу при написанні коду	Більший час на виконання операцій
F2	А	Безкоштовність	Відсутність вкладених запитів
	Б	Надійність, внесення змін без перезапуску, безкоштовність	Необхідність додаткової інсталяції, низький рівень користувацької підтримки
F3	*А*	Простота створення	Відсутність кросплатформеності
	Б	Простота створення,	Кросплатформеність

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція *F1:*

Оскільки розрахунки проводяться з великими об’ємами вхідних даних, то час виконання програмного коду є дуже необхідним, тому варіант а) має бути відкинутий.

Функція *F2:*

Вибір СКБД не відіграє велику роль у даному програмному продукту, тому вважаємо варіанти а) та б) гідними розгляду.

Функція *F3:*

Оскільки, програмний продукт реалізується на мові Java, використовуємо варіант Б як єдиний можливий.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3б
2. F1б – F2б – F3б

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.4 Обґрунтування системи параметрів ПП

4.4.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

– *X1* – швидкодія мови програмування;

– *X2* – об’єм пам’яті для збереження даних;

– *X3* – час обробки даних;

– *X4* – потенційний об’єм програмного коду.

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає об’єм пам’яті в оперативній пам’яті персонального комп’ютера, необхідний для збереження та обробки даних під час виконання програми.

X3: Відображає час, який витрачається на дії.

X4: Показує розмір програмного коду який необхідно створити безпосередньо розробнику.

4.4.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 4.2.

Таблиця 4.2 – Основні параметри ПП

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			Гірші	середні	кращі
Швидкодія мови програмування	X1	Оп/мс	19000	11000	2000
Об'єм пам'яті для збереження даних	X2	Мб	32	16	8
Час обробки запитів користувача	X3	мс	1000	420	60
Потенційний об'єм програмного коду	X4	кількість строк коду	2000	1500	1000

За даними таблиці 4.2 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

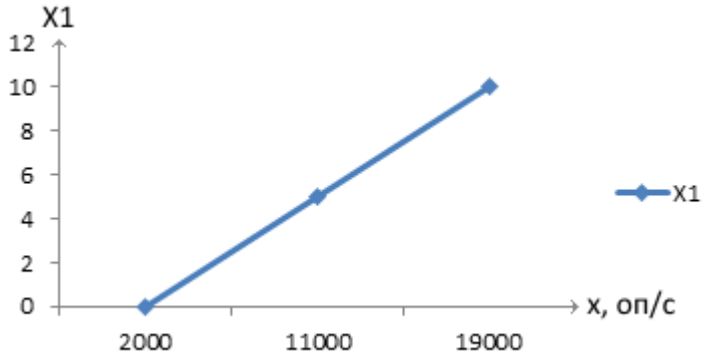

Рисунок 4.2 – X1, швидкодія мови програмування

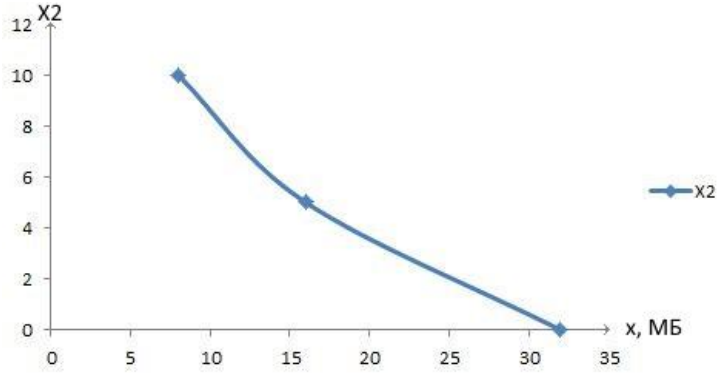

Рисунок 4.3 – X2, об'єм пам'яті для збереження даних

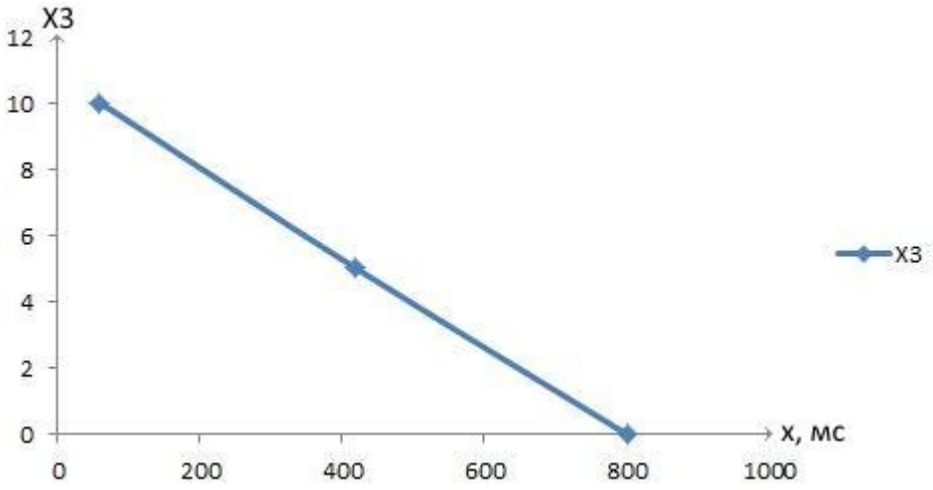

Рисунок 4.4 – X3, час виконання запитів користувача

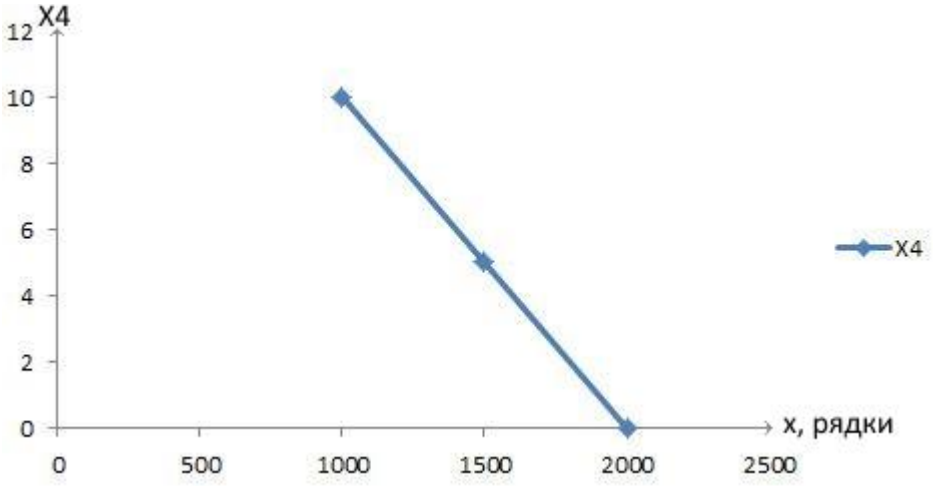

Рисунок 4.5 – X4, потенційний об’єм програмного коду

4.4.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;

– перевірку придатності експертних оцінок для подальшого використання;

– визначення оцінки попарного пріоритету параметрів;

– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Таблиця 4.3 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	4	3	4	4	4	4	4	27	0,75	0,56
X2	Об’єм пам’яті для збереження даних	Мб	4	4	4	3	4	3	3	25	-1,25	1,56
X3	Час обробки запитів користувача	Мс	2	2	1	2	1	2	2	12	-14,25	203,06
X4	Потенційний об’єм програмного коду	кількість строк коду	5	6	6	6	6	6	6	41	14,75	217,56
	Разом		15	15	15	15	15	15	15	105	0	420,75

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 105 \qquad (4.1)$$

де N – число експертів, n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26{,}25 \qquad (4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T \qquad (4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 420{,}75 \qquad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 420{,}75}{7^2(5^3 - 5)} = 1{,}03 > W_k = 0{,}67$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	=	>	=	<	=	<	<	<	0,5
X1 і X3	<	<	<	<	<	<	<	<	0,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	<	<	<	<	<	<	<	<	0,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	>	>	>	>	>	>	>	>	1,5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1{,}5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}$$

З отриманих числових оцінок переваги складемо матрицю А=$\| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i}$,де $b_i = \sum_{i=1}^{N} a_{ij}$.

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%).На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}$,де $b_i' = \sum_{i=1}^{N} a_{ij} b_j$.

Як видно з таблиці 5.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметриx_i	Параметриx_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1,0	0,5	0,5	1,5	3,5	0,219	22,25	0,216	100	0,215
X2	1,5	1,0	0,5	1,5	4,5	0,281	27,25	0,282	124,25	0,283
X3	1,5	1,5	1,0	1,5	5,5	0,344	34,25	0,347	156	0,348
X4	0,5	0,5	0,5	1,0	2,5	0,156	14,25	0,155	64,75	0,154
Всього:					16	1	98	1	445	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2*(об’єм пам’яті для збереження даних) та *X1* (швидкодія мови програмування) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра *X3 (*час обробки даних) обрано не найгіршим (не максимальним), тобто це значення відповідає або варіанту а) 1000 мс або варіанту б) 80мс.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j} \qquad (4.5)$$

де n – кількість параметрів;$K_{вi}$– коефіцієнт вагомості i–го параметра;B_i – оцінка i–го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	11000	3,6	0,215	0,774
F2(X2)	А	16	3,4	0,283	0,962
F3(X3,X4)	А	800	2,4	0,348	0,835
	Б	80	1	0,154	0,154

За даними з таблиці 4.6 за формулою

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

K_{K1} = 0,774 + 0,962 + 0,835 = 2,57

K_{K2} = 0,774 + 0,962 + 0,154 = 1,89

Як видно з розрахунків, кращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М},$$

де T_P – трудомісткість розробки ПП;$K_П$ – поправочний коефіцієнт; $K_{СК}$ – коефіцієнт на складність вхідної інформації; K_M – коефіцієнт рівня мови програмування; $K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм; $K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $K_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ}$ = 0.8. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4$ людино-днів.

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $T_Р$ =27 людино-днів, $К_П$ =0.9,$К_{СК}$ = 1,$К_{СТ}$ =0.8:

$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44$ людино-днів.

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$T_I = (122.4 + 19.44 + 4.8 + 19.44) \cdot 8 = 1328{,}64$ людино-годин;

$T_{II} = (122.4 + 19.44 + 6.91 + 19.44) \cdot 8 = 1345.52$ людино-годин;

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 7000 грн., один фінансовий аналітик з окладом 9500грн. Визначимо зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m \cdot t} \text{грн.},$$

де М – місячний оклад працівників;T_m – кількість робочих днів тиждень;t – кількість робочих годин в день.

$$С_ч = \frac{7000 + 7000 + 9500}{3 \cdot 21 \cdot 8} = 46{,}62 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою

$$С_{ЗП} = С_ч \cdot T_i \cdot К_Д,$$

де $С_ч$– величина погодинної оплати праці програміста;T_i – трудомісткість відповідного завдання; $К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 46{,}62 \cdot 1328.64 \cdot 1.2 = 74340{,}57$ грн.

II. $С_{ЗП} = 46{,}62 \cdot 1345.52 \cdot 1.2 = 75285{,}04$ грн.

Відрахування на єдиний соціальний внесок в залежності від групи професійного ризику (II клас) становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 74340{,}57*0.22 = 16354.93$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 75285{,}04*0.22 = 16562.71$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 7000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 7000 \cdot 0,2 = 16800$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1 + К_З) = 16800 \cdot (1 + 0.2) = 20160$ грн.

Відрахування на єдиний соціальний внесок:

$С_{ВІД} = С_{ЗП} \cdot 0.22 = 20160 * 0,22 = 4435.20$ грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 10000 грн.

$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 10000 = 2875$ грн.,

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$– річна норма амортизації; $Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 10000 \cdot 0.05 = 575$ грн.,

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 16) \cdot 8 \cdot 0.9 = 1706.4$ годин,

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t –кількість робочих годин в день; $К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706,4 \cdot 0,156 \cdot 0,9733 \cdot 2,0218 = 523.83$ грн.,

де $N_С$ – середньо-споживча потужність приладу; $К_З$– коефіцієнтом зайнятості приладу; $Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$С_Н = Ц_{ПР} \cdot 0.67 = 10000 \cdot 0,67 = 6700$ грн.

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$С_{ЕКС} = 20160 + 4435.20 + 2875 + 575 + 523.83 + 6700 = 35269.03$ грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$С_{М-Г} = С_{ЕКС} / Т_{ЕФ} = 35269.03 / 1706,4 = 20,67$ грн/час.

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$С_М = С_{М-Г} \cdot Т$$

I. $С_М = 20,67 * 1328,64 = 27462.99$ грн.;

II. $С_М = 20,67 * 1345.52 = 27811.9$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0,67$$

I. $С_Н = 74340,57 * 0,67 = 49808,18$ грн.;

II. $С_Н = 75285,04 * 0,67 = 50440,98$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н$$

I. $С_{ПП} = 74340,57 + 16354.93 + 27462.99 + 49808,18 = 167966.67$ грн.;

II. $С_{ПП} = 75285,04 + 16562.71 + 27811.9 + 50440,98 = 170100.63$ грн.;

4.7 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{Фj},$$

$К_{ТЕР1} = 2,57 / 167966.67 = 0,15 \cdot 10^{-4}$;

$К_{ТЕР2} = 1,89 / 183561,43 = 0,1 \cdot 10^{-4}$;

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 0,15 \cdot 10^{-4}$.

4.8 Висновки

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 0,15 \cdot 10^{-4}$.

Цей варіант реалізації програмного продукту має такі параметри:

– мова програмування – Java;

– СКБД MySQL;

– інтерфейс користувача, створений за технологією JavaFX.

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, функціонал і швидкодію.

ВИСНОВКИ

Метою роботи була розробка математичної моделі і алгоритмів пошуку шляху і оминання перешкод для чотириколісної рухомої платформи, яка є досить поширеною у реальному світі.

В першу чергу була побудована математична модель, яка описує основні параметри рухомої платформи, такі як положення, швидкість, довжина і ширина шасі, довжина і ширина корпуса, кут орієнтації платформи у просторі і кут повороту передньої осі. Математична модель також описує зв’язки і взаємодію між основними параметрами системи, зокрема вплив швидкості на зміну положення платформи, зв’язок довжини і ширини шасі з радіусом кола повороту, вплив куту повороту передньої осі на траєкторію руху платформи. Окрім цього, завдяки таким параметрам, як довжина і ширина корпуса можна визначити можливі колізії з перешкодами. Завдяки використанню складених рівнянь математичної моделі можна доволі точно спрогнозувати зміну положення і траєкторію рухомої платформи в залежності від часу і заданих керувальних дій, таких швидкість і кут повороту передньої осі.

Після складання математичної моделі було досліджено існуючі алгоритми пошуку шляху і оминання перешкод. Було виявлено, що класичні алгоритми пошуку шляху, такі як A* і Theta* не можна напряму застосувати для неголономних об’єктів, зокрема і для чотириколісної рухомої платформи, яка і досліджувалась в роботі, а також те, що алгоритми, які враховують кінематику рухі об’єктів, зокрема “Гібридний A*”, є доволі повільними і неприйнятними для вирішення задачі пошуку шляху в реальному часі. Тому було прийнято рішення поєднати швидкий алгоритм Theta* з алгоритмом “Гібридним A*” таким чином, щоб спочатку прокладався шлях, який являє собою набір сполучених відрізків, за допомогою алгоритму Theta*, а потім для

кожного з цих відрізків складалась можлива для виконання траєкторія за допомогою алгоритму A*.

Для програмної реалізації математичної моделі рухомої платформи і алгоритмів пошуку шляху і оминання перешкод було обрано мову Java, яка з одного боку є доволі високорівневою, але в той же час швидкою, тобто її характеристики вдало підходять для реалізація поставлених задач. Спочатку було складено високорівневу архітектуру програмної системи, потім були спроектовані класи і функція, а вже після цього слідувало написання коду. Після написання програмна компоненти програмної системи були протестовані за допомогою методик модульного тестування, а також проведено тестування всієї системи в цілому.

В результаті роботи була реалізована математична модель, яка є вдалою абстракцією для чотириколісних рухомих платформ, а також швидкий алгоритм пошуку шляху і оминання перешкод, який оснований на поєднанні алгоритмів Theta* і алгоритму “Гібридний A*.

Реалізовану програмну систему можна використовувати для вирішення реальних задач робототехніки, зокрема при побудові автопілоту для рухомих платформ.

ПЕРЕЛІК ПОСИЛАНЬ

1. Clive Dym. Principles of Mathematical Modeling, Second Edition / Clive Dym. - San Diegl: Elsevier Academic Press, 2004. – 303 p.

2. Боголюбов А.Н. Основы математического моделирования / Боголюбов А.Н. - М.: МГУ им. Ломоносова, Физический факультет. - 137 с.

3. rcTek - Ackerman Steering Principle. - Режим доступа: http://www.rctek.com/technical/handling/ackerman_steering_principle.html. - Дата доступа: 15.05.2016.

4. Drogerdy - Raspberry Pi Controlled Tank Bot. - Режим доступа: http://www.thingiverse.com/thing:652851. - Дата доступа: 23:05.2016.

5. Big dog military robots. - Режим доступа: https://en.wikipedia.org/wiki/BigDog#/media/File:Big_dog_military_robots.jpg. Дата доступа: 23.05.2016.

6. Stuard Russell, Peter Norvig. Artificial Intelligence: A Modern Approach (3rd ed.) / Stuard Russell, Peter Norvig - Prentice Hall., 2009. – 1104 p.

7. Dubins, L.E. (July 1957). "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". / Dubins, L.E. // - American Journal of Mathematics 79 (3): pp.497–516.

8. Reeds, J.A. and L.A. Shepp, "Optimal paths for a car that goes both forwards and backwards," / Reeds, J.A. and L.A. Shepp // Pacific J. Math., 145 (1990), pp. 367–393

9. Introduction to A*. - Режим доступа: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. - Дата доступа: 12.05.2016.

10. Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments. - Режим доступа: http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/. - Дата доступа: 18.05.2016.

11. Practical Search Techniques in Autonomous Driving. Режим доступа: http://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf. - Дата доступа: 19.05.2016.

12. Java Platform, Standard Edition (Java SE) 8. Режим доступа: http://docs.oracle.com/javase/8/javase-clienttechnologies.htm. - Дата доступа: 3.05.2016.

13. Joshua Bloch. Effective Java (2nd Edition) / Joshua Bloch. - Addison-Wesley Professional, 2008. - 346 p.

14. James Martin. Principles of object-oriented analysis and design. / James Martin. - New Jersey, Prentice Hall, 1993. - 515 p.

15. Steve Freeman, Nat Pryce. Growing Object-Oriented Software, Guided by Tests / Steve Freeman, Nat Pryce. - Addison-Wesley, 2010. – 384 p.