МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

Институт ЭнМИ
Кафедра РМДиПМ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)

Направление 15.03.06 Мехатроника и робототехника
(код и наименование)

Образовательная программа Компьютерные технологии управления в робототехнике и мехатронике

Форма обучения очная
(очная/очно-заочная/заочная)

Тема: Мультиагентное взаимодействие роя роботов

Студент С-12б-20 Разорвин А.Д.
группа подпись фамилия и инициалы

Руководитель ВКР к.ф.-м.н. доцент Адамов Б.И.
уч. степень должность подпись фамилия и инициалы

Консультант
уч. степень должность подпись фамилия и инициалы

Внешний консультант
уч. степень должность подпись фамилия и инициалы

организация

«Работа допущена к защите»

Заведующий кафедрой д.т.н. профессор Меркурьев И.В.
уч. степень звание подпись фамилия и инициалы

Дата

Москва, 2024

МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

Институт ЭнМИ
Кафедра РМДиПМ

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)

Направление 15.03.06 Мехатроника и робототехника
(код и наименование)

Образовательная программа Компьютерные технологии управления в робототехнике и мехатронике

Форма обучения очная
(очная/очно-заочная/заочная)

Тема: Мультиагентное взаимодействие роя роботов

Студент С-12б-20 Разорвин А.Д.
группа подпись фамилия и инициалы

Руководитель ВКР к.ф.-м.н. доцент Адамов Б.И.
уч. степень должность подпись фамилия и инициалы

Консультант
уч. степень должность подпись фамилия и инициалы

Внешний консультант
уч. степень должность подпись фамилия и инициалы

организация

Заведующий кафедрой д.т.н. профессор Меркурьев И.В.
уч. степень звание подпись фамилия и инициалы

Место выполнения работы ФГБОУ ВО «НИУ «МЭИ»

СОДЕРЖАНИЕ РАЗДЕЛОВ ЗАДАНИЯ И ИСХОДНЫЕ ДАННЫЕ

ВВЕДЕНИЕ: Обзор литературы. Актуальность работы.

ГЛАВА 1 ПОСТАНОВКА ЗАДАЧИ: Описание мультиагентной задачи, кинематическая и принципиальная схема робота.

ГЛАВА 2 РАЗРАБОТКА МАТЕМАТИЧЕСКОЙ МОДЕЛИ РОЯ РОБОТОВ: Математическая модель робота-агента, модель определения положения и ориентации робота.

ГЛАВА 3 РАЗРАБОТКА ПРОГРАММЫ КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ И АЛГОРИТМОВ РОЕВОГО УПРАВЛЕНИЯ: Разработка программы компьютерного моделирования и роевых алгоритмов, сравнение результатов моделирования.

ГЛАВА 4 ПРОВЕДЕНИЕ НАТУРНОГО ЭКСПЕРИМЕНТА С РОЕМ РОБОТОВ НА МАКЕТЕ ГОРОДСКОЙ ИНТЕЛЛЕКТУАЛЬНОЙ ТРАНСПОРТНОЙ СИСТЕМЫ: Разработка программы для роботов, проведение эксперимента для алгоритмов роевого управления, выводы по работе.

ЗАКЛЮЧЕНИЕ: Результаты и анализ проделанной работы.

Исходные данные:

1. Макет городской интеллектуальной транспортной системы платформы «Робовейник»
2. Роботы на базе Raspberry Pi 4

ПЕРЕЧЕНЬ ГРАФИЧЕСКОГО МАТЕРИАЛА

Количество листов 56

Количество слайдов в презентации 15

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. В. И. Городецкий, О. В. Карсаев, В. В. Самойлов, С. В. Серебряков, // Прикладные многоагентные системы группового управления./ Искусственный интеллект и принятие решений, 2009, выпуск 2, 3–24

2. Baykasoglu A., Ozbakir L., Tapkan P.// Artificial bee colony algorithm and its application to generalized assignment problem /Swarm Intelligence: Focus on Ant and particle swarm optimization. –2007.– Т. 1.

3. M. Dorigo, M. Birattari and T. Stutzle, //Ant colony optimization / in IEEE Computational Intelligence Magazine, vol. 1, no. 4, pp. 28-39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

4. Андреев В.Д.// Теория инерциальной навигации. (Автономные системы). / Издательство «Наука», 1966.

5. Y. Cheng and T. Zhou, // UWB Indoor Positioning Algorithm Based on TDOA Technology/ pp. 777-782, 2019.

6. O. V. Glukhov, I. A. Akinfiev, A. D. Razorvin, A. A. Chugunov, D. A. Gutarev and S. A. Serov, "Loosely Coupled UWB/Stereo Camera Integration for Mobile Robots Indoor Navigation," 2023 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE), Moscow, Russian Federation, 2023, pp. 1-7, doi: 10.1109/REEPE57272.2023.10086807.

АННОТАЦИЯ

Выпускная работа бакалавра посвящена разработке алгоритмов мультиагентного взаимодействия роя роботов и их исследованию в контексте задачи нахождения кратчайшего пути для каждого робота-агента.

Работа состоит из введения, четырёх основных глав, заключения, списка используемых источников информации и приложения с листингом программ.

В первой главе описана мультиагентная задача управления роем, кинематическая и принципиальная схема робота.

Во второй главе разработана математическая модель робота-агента, модель определения положения и ориентации робота.

В третьей главе разработана программа для моделирования мультиагентной задачи и тестирования алгоритмов роевого управления. Описаны три алгоритма мультиагентного взаимодействия, проведено моделирование этих алгоритмов и выполнен анализ результатов.

В четвёртой главе представлен натурный эксперимент, его результаты и анализ. Также описан состав макет городской интеллектуальной транспортной системы платформы «Робовейник», на котором проводился эксперимент, изложена методика проведения эксперимента и сделан вывод по всей работе.

Работа содержит 56 страниц, 27 рисунков, 2 таблицы и 2 приложения.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Мультиагентная система [1] в контексте роевого управления мобильными роботами представляет собой совокупность агентов, каждый из которых обладает собственной степенью автономности, способностью к самостоятельному принятию решений и выполнению задач. Важной особенностью таких систем является их способность к кооперации, координации и распределенному решению задач без централизованного управления. Мультиагентные системы, применяемые в роевом управлении, вдохновлены природными явлениями, такими как поведение стай птиц, рыб, колоний муравьёв и других социальных насекомых, которые демонстрируют сложные формы коллективного поведения, основанные на простых правилах взаимодействия между отдельными особями.

Основные принципы роевого управления, согласно [1] включают:

− Автономность: каждый робот в рое обладает собственной степенью автономности и способен самостоятельно выполнять задачи.

− Локальное взаимодействие: роботы обмениваются информацией только с непосредственно «видимыми» соседями или через локальное взаимодействие с окружающей средой.

− Распределенное решение задач: роевая система не зависит от одного управляющего центра; вместо этого каждый агент вносит вклад в общую задачу, основываясь на локально доступной информации.

− Масштабируемость: система способна эффективно работать при увеличении или уменьшении числа агентов.

− Устойчивость к отказам: благодаря распределенной природе, система может продолжать функционировать даже при выходе из строя одного или нескольких агентов.

Обзор литературы

Рассмотрим самые распространённые описанные в литературе алгоритмы, которые реализуют принципы роевого управления.

Алгоритмы на основе метода оптимизации роя частиц (Particle Swarm Optimization, PSO) [2] моделируют социальное поведение роя, где каждая частица адаптируется, исходя из своего опыта и опыта своих соседей, для поиска оптимального решения задачи. В контексте мобильных роботов PSO можно применять для задач, таких как планирование траекторий и координация движений в пространстве с препятствиями. Алгоритм демонстрирует высокую эффективность в быстром нахождении оптимальных или близких к оптимальным решений в сложных многомерных пространствах, делая его идеальным для задач, где требуется быстродействие и точность выполнения целевой задачи. Это хорошо описано в работе [3], где PSO использует опыт каждой частицы и в отдельности каждая частица использует свой лучший опыт.

Муравьиные алгоритмы [4] (Ant Colony Optimization, ACO): вдохновлены поведением муравьёв при поиске пищи, где они оставляют феромоновые следы для обозначения оптимальных путей, что используется для решения оптимизационных задач. Применение ACO в многоагентных системах для решения задач структурной оптимизации в энергетических системах показано в статье [5]. В статье [5] проведено сравнение муравьиного алгоритма с генетическим алгоритмом на матрицах различных размерностей. Результаты показали, что муравьиный алгоритм превосходит генетический алгоритм в скорости и значении целевой функции на матрицах больших размерностей.

В работе [6] рассматривается модификация муравьиного алгоритма MMAS, связанная с ограничением феромона. Эта модификация хорошо вписывается в естественные законы природы и позволяет решать задачи оптимизации. В статье [7] представлен модернизированный алгоритм ACO,

который рассматривает принципы построения и организации мультиагентных систем на основе эволюционного проектирования.

В мультиагентных системах мобильных роботов, ACO может использоваться для оптимизации маршрутов и распределения задач между агентами, таких как доставка грузов в логистических приложениях или поиск и спасение. Алгоритм показывает отличные результаты в решении комбинаторных и маршрутных задач, предлагая эффективные пути в динамично изменяющихся условиях среды. Интересным примером использования данного семейства алгоритмов является работа [8], в которой описано решение задачи коммивояжёра мультиагентным методом, подобно муравьиной колонии.

Алгоритмы поведения косяков птиц/рыб [9] основаны на имитации поведения стай птиц или рыб, включая правила разделения, выравнивания и сгущения, для координированного движения группы как единого цеого.

Бактериальный алгоритм взаимодействия (Bacterial Foraging Optimization) [10] вдохновлён микробиологическим процессом поиска пищи бактериями, где используется стратегия для оптимизации решений путём имитации поведения бактерий.

Алгоритмы, основанные на имитации поведения пчёл [11] используют поведение пчёл при опылении и поиске пищи для решения задач оптимизации, где пчёлы "танцуют", указывая на лучшие ресурсы.

В дополнение к вышеупомянутым методам, в мультиагентных системах широко применяются алгоритмы, основанные на чёткой логике (Common sense). Эти алгоритмы используют простые, но эффективные правила для управления поведением агентов, например, правила для избежания столкновений, следования за лидером или распределения по территории. Преимущество таких алгоритмов заключается в их прозрачности и простоте реализации, что делает их подходящими для задач, где требуется надёжность и предсказуемость поведения роя. Они могут быть использованы как самостоятельно, так и в комбинации с более сложными методами,

например, для первоначального распределения агентов перед применением более сложных стратегий оптимизации. Подробнее использование комбинаций различных методов описаны в монографии [12]. В работе раскрыты основные проблемы, связанные с разработкой интеллектуального вычислителя решений задач автономных интеллектуальных мобильных систем различного назначения, способных целенаправленно функционировать в априори неописанных и недоопределенных условиях сложной проблемной среды. Интеграция этих алгоритмов в мультиагентные системы предполагает разработку программного обеспечения, которое включает в себя не только сам алгоритм управления поведением агентов, но и коммуникационные протоколы для обмена информацией между агентами.

Так же важным аспектом является разработка математической модели мультиагентной системы и агента, что позволяет точно симулировать их взаимодействие и адаптировать алгоритмы для выполнения конкретных задач. Подробная математическая модель роевой системы показана в работе [13]. В ней подробно описана модель, которая учитывает энергетические характеристики единичного агента, группы роботов и роя в целом.

Рассмотренные алгоритмы демонстрируют, как простые правила взаимодействия могут привести к возникновению сложного и целенаправленного коллективного поведения, что делает их мощными инструментами для решения широкого спектра задач в области робототехники и искусственного интеллекта.

Актуальность работы

В эпоху стремительного развития робототехники и искусственного интеллекта распределенное управление роем роботов представляет собой инновационный подход в области робототехники, поэтому актуальным становится анализ существующих решений и научных исследований в этой области. Настоящая работа посвящена поиску и анализу существующих алгоритмов роевого управления, их моделированию и сравнению скорости выполнения поставленной задачи роем роботов в программной среде и в натурных экспериментах.

Для натурных экспериментов в работе будет использоваться макет городской интеллектуальной транспортной системы (ИТС) платформы «Робовейник», основанной в Московском энергетическом институте в 2022 году [14].

ГЛАВА 1 ПОСТАНОВКА ЗАДАЧИ

В рамках работы рассматривается моделирование мультиагентной системы роботов, разработка программного обеспечения (ПО) для каждого агента и оценка скорости решения задачи передвижения роя в пространстве с препятствиями с помощью различных алгоритмов роевого управления. По результатам моделирования, более эффективный алгоритм будет реализован в реальной среде, на макете городской ИТС с колёсными роботами.

Для выполнения работы требуется:

- разработать математическую модель системы;
- разработать математическую модель робота-агента;
- разработать программу для компьютерного моделирования работы мультиагентных алгоритмов;
- реализовать мультиагентные алгоритмы;
- сравнить полученные алгоритмы по показателям качества;
- реализовать аналогичную мультиагентную систему на макете городской ИТС;
- разработать ПО для колёсных роботов и запустить роевые алгоритмы в реальной среде.

1.1 Описание мультиагентной задачи управления

Мультиагентные алгоритмы имеют большую эффективность в решении задач, которые не могут иметь централизованного управления или требуют большие вычислительные ресурсы. Рой роботов способен выполнять задачи более эффективно, чем одиночные агенты, благодаря способности к самоорганизации, распределенному принятии решений и адаптивности к изменяющимся условиям окружающей среды. Исходя из этого поставим задачи, которые должна решать наша мультиагентная система:

– переместить каждого агента роя из своего начального положения в целевое;

– избегать столкновения агентов роя друг с другом и с окружающими препятствиями;

– выполнить задачу наиболее эффективно по времени.

Для решения этой задачи каждый агент роя роботов должен определять своё местоположение, уметь планировать свой путь, учитывать положения окружающих роботов и соблюдать требования, заданные для всего роя.

Мультиагентной системой будет считаться совокупность роботов-агентов, имеющих собственные вычислители, средства связи и протокол общения с остальными агентами, навигационную систему и набор датчиков, который позволяет решать поставленную задачу (рисунок 1). Все агенты имеют равные возможности.

Рисунок 1. Схема робота-агента

Протокол общения между агентами роя выберем таким образом, чтобы он позволял масштабировать систему, имел наименьшую задержку и мог передавать всю необходимую информацию. При моделировании этот шаг будет организован внутри программы симуляции, а на реальных роботах будет использована локальная сеть WiFi [15] и протокол WebSocket [16]. На рисунке 2 изображена схема обмена сообщениями:

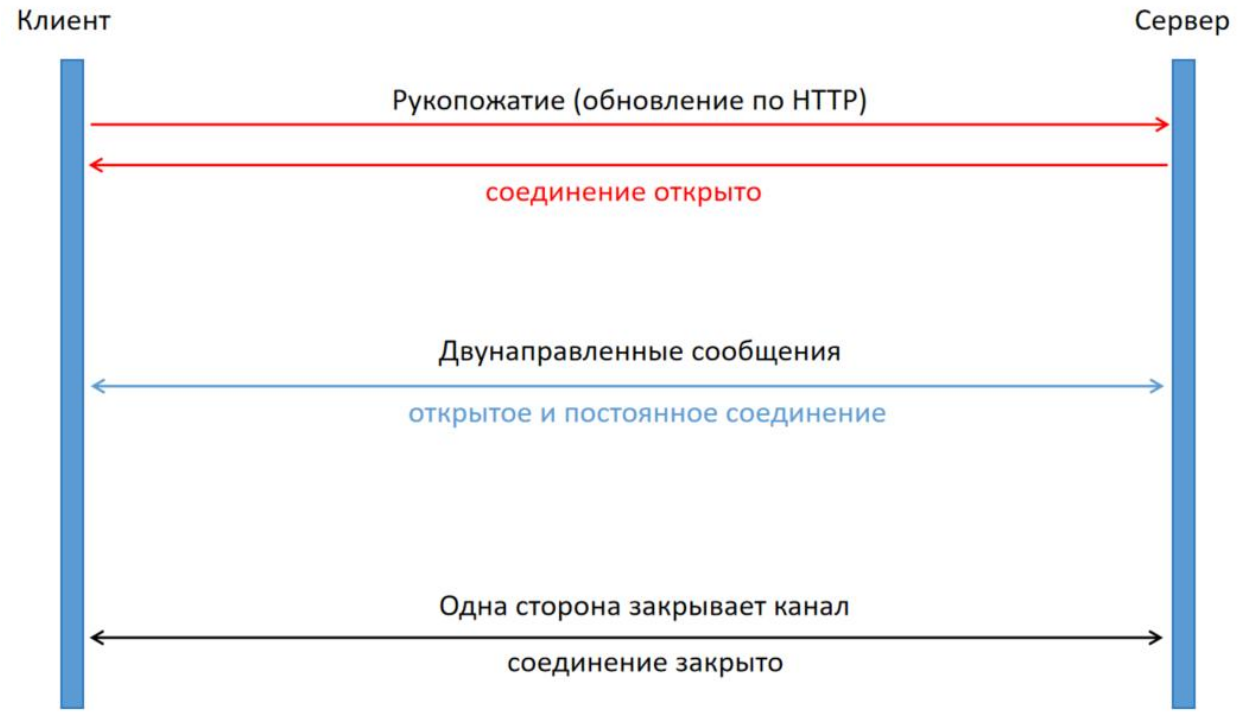

Рисунок 2. Принцип обмена сообщениями по WebSocket

Каждый робот является и сервером и клиентом одновременно, обмениваясь данными со всеми доступными в сети агентами.

Набор датчиков, который необходим для решения мультиагентной задачи управления: приемно-передающий модуль радионавигационной системы, дальномер и инерциальный измерительный блок (рисунок 3).

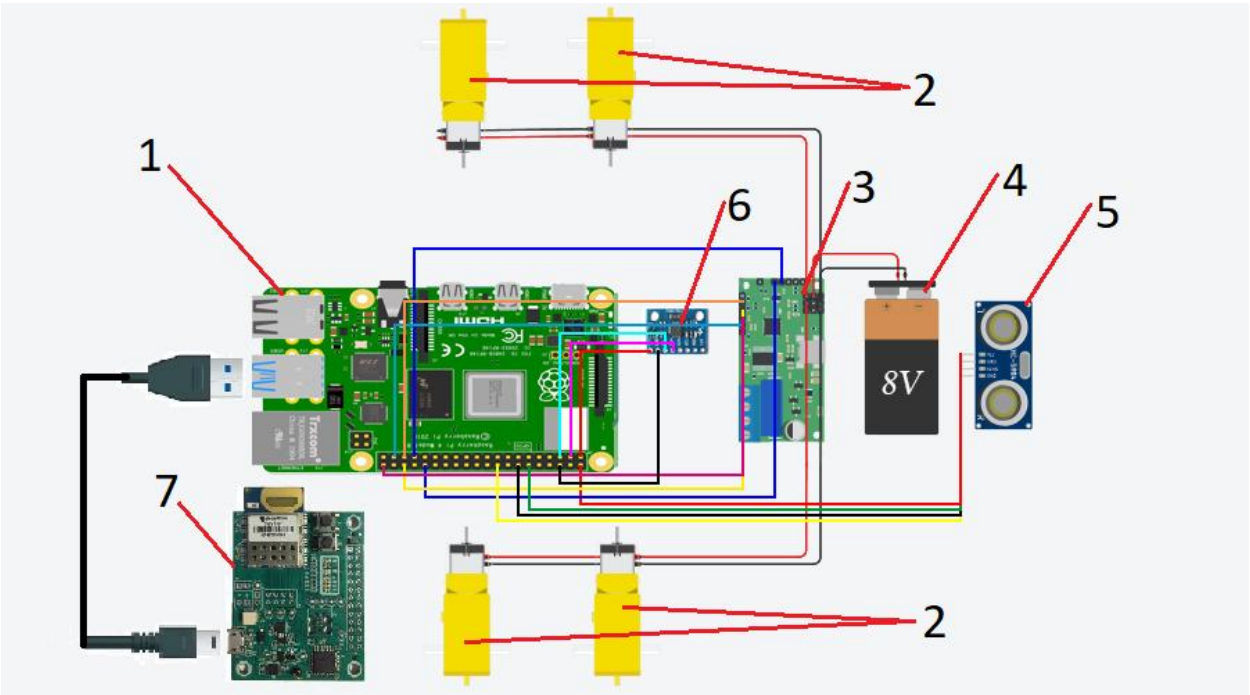

Рисунок 3. Принципиальная схема робота

На рисунке 3 цифрами обозначены: 1 - Raspberry Pi 4, 2 - моторы с редукторами, 3 - драйвер электродвигателей, 4 - элемент питания, 5 - ультразвуковой дальномер, 6 - инерциальный датчик, 7 - радиомодуль MDEK1001.

Так же робот должен иметь средства для перемещения в пространстве. Примем кинематическую схему робота как дифференциальный привод: правые и левые колёса вращаются синхронно, поэтому можно принять их линейную скорость общей. (рисунок 4).

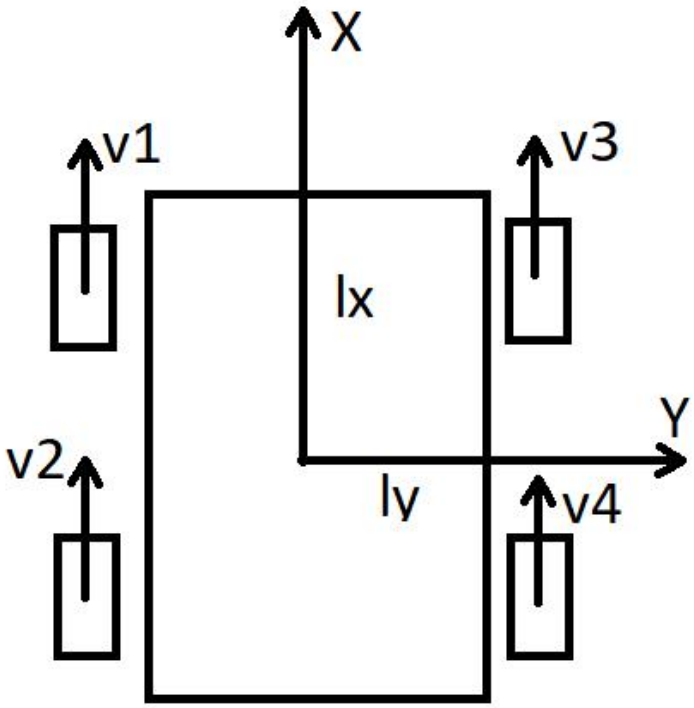

Рисунок 4. Кинематическая схема робота

Линейные скорости левой и правой стороны робота:

$$V_l = V_1 + V_2, \quad (1)$$

$$V_r = V_3 + V_4.$$

При помощи этого набора измерителей и средств для перемещения, каждый робот будет способен самостоятельно перемещаться, планировать свой путь, отслеживать препятствия и свою ориентацию в пространстве.

1.2 Выводы по главе

Поставленная мультиагентная задача управления позволит выявить алгоритм, который потратит на выполнение наименьшее количество времени. Согласно плану работы далее будет описана разработка математическая модель системы, среда симуляции для роевых задач, алгоритмы мультиагентного взаимодействия, программа для реальных роботов и проведён эксперимент на макете городской ИТС.

ГЛАВА 2 РАЗРАБОТКА МАТЕМАТИЧЕСКОЙ МОДЕЛИ РОЯ РОБОТОВ

Математическая модель мультиагентной системы роя роботов включает в себя динамику перемещения каждого робота, механизмы избежания столкновений, и стратегии достижения целевых позиций. Модель можно описать с использованием состояний, управляющих воздействий и динамических уравнений движения, а также алгоритмов решения задач навигации.

Внешний вид робота-агента представлен на рисунке 5.

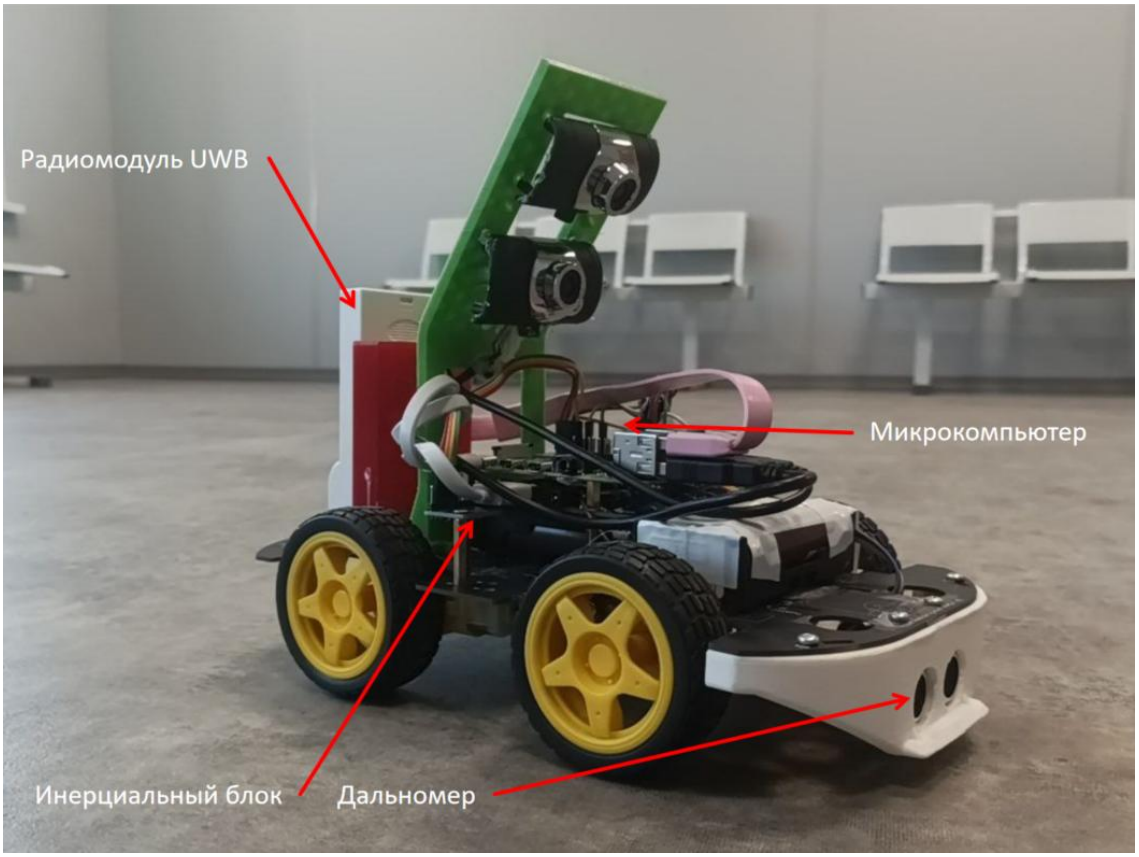

Рисунок 5. Внешний вид робота

2.1 Составление модели робота-агента

Для каждого робота определим вектор состояния:

$$\bar{x} = (x, y, \theta) , \quad (2)$$

где $x, y -$ координаты робота в двумерном пространстве, $\theta -$ угол ориентации робота относительно глобальной системы координат.

Управляющие воздействия для каждого шага описываются вектором:

$$\mathrm{u} = (v, \omega), \quad (3)$$

где $v -$ линейная скорость движения вперёд, $\omega -$ угловая скорость поворота.

Движение робота моделируется с использованием уравнений кинематики дифференциального привода уравнений:

$$x_{t+1} = x_t + v_t cos(\theta_t), \quad (4)$$
$$y_{t+1} = y_t + v_t sin(\theta_t),$$
$$\theta_{t+1} = \theta_t + \omega_t t,$$

где t обозначает текущий момент времени.

Каждый агент принимает на каждый дискретный шаг времени данные с датчиков: дальномер (дальность до объекта перед роботом), инерциальный датчик (даёт угол ориентации робота) и навигационная система (даёт координаты робота в двумерном пространстве).

Для определения препятствий робот-агент должен анализировать неподвижные (статичные элементы среды) и подвижные (другие агенты роя) объекты. Неподвижные объекты можно определить с помощью дальномера, проверяя пространство перед роботом, соблюдая заданный порог (5).

$$d_{max} = const. \quad (5)$$

Проверку на столкновение с другими агентами роя можно выполнить при помощи определения расстояния между известными координатами окружающих роботов. При помощи системы связи каждый робот имеет информацию о положении его соседей, получая от них вектора состояния (2) и управления (3) на данный временной шаг. Определить расстояние до других роботов можно по формуле (6):

$$d_r = \sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2}, \quad (6)$$

где x_0, y_0 - собственные координаты робота, а x_i, y_i - координаты i-го робота-агента.

Таким образом, выполняя проверку (5) и (6) перед началом движения робот сможет избегать столкновения с различными препятствиями.

2.2 Модель определение положения робота

Положение робота в пространстве определим при помощи локальной навигационной системы дальномерным методом как в работах [17] и [18]. По периметру рабочей области эксперимента (рисунок 6) расставлены опорные радионавигационные точки MDEK1001 (рисунок 7).

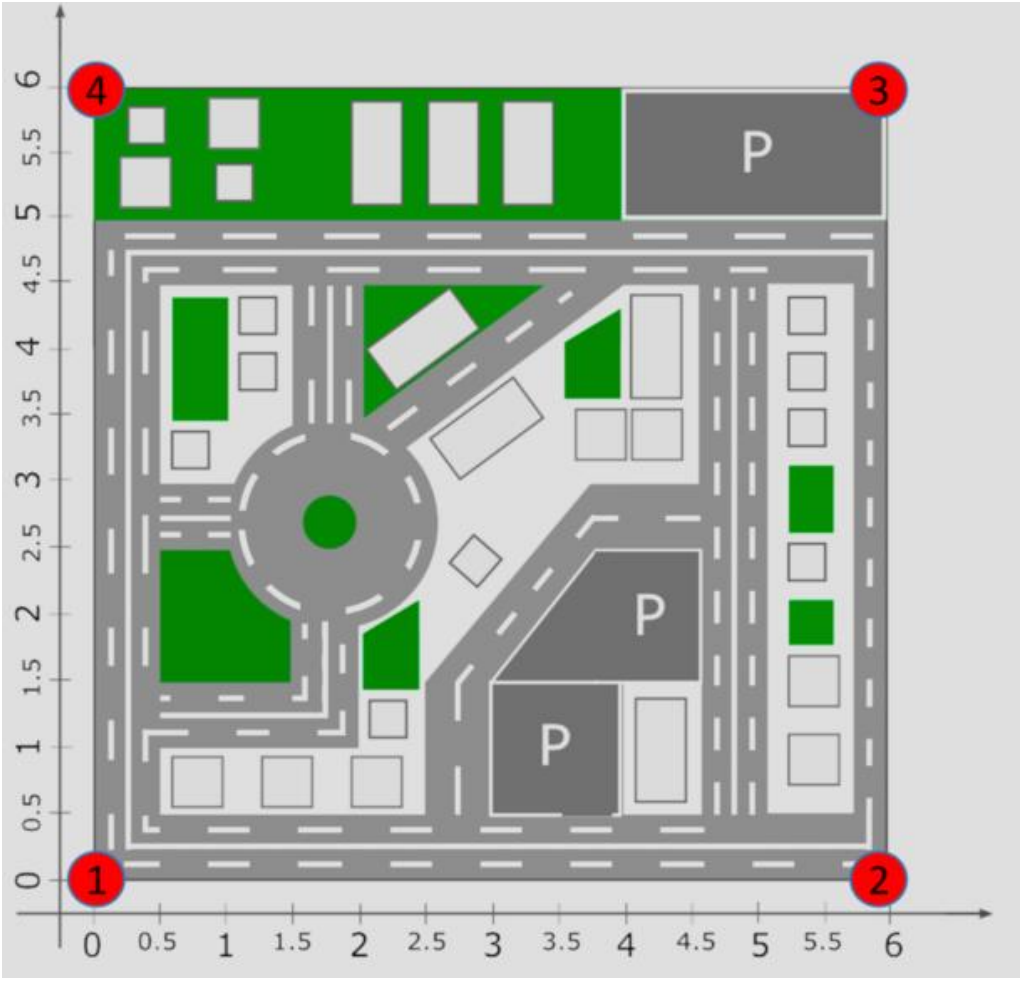

Рисунок 6. Расстановка опорных радионавигационных точек

Рисунок 7. Опорная радионавигационная точка MDEK1001

Такой же радиомодуль установлен на каждом роботе и принимает дальности от опорных точек. Получаемый вектор дальностей (7):

$$\widehat{R} = \left|\widehat{R}_1\ \widehat{R}_2\ \widehat{R}_3\ \widehat{R}_4\right|^T. \tag{7}$$

Кроме дальностей, радиомодуль имеет априорную информацию о положении опорных точек:

$$x_1 = |x_1\ y_1\ z_1|^T, \tag{8}$$
$$x_2 = |x_2\ y_2\ z_2|^T,$$
$$x_3 = |x_3\ y_3\ z_3|^T,$$
$$x_4 = |x_4\ y_4\ z_4|^T.$$

Необходимо определить координаты робота (9):

$$x = |x_0\ y_0\ z_0|^T. \tag{9}$$

Запишем функциональную связь между измеряемой дальностью и координатами объекта (10):

$$R_i = \sqrt{(\mathrm{x_i} - \mathrm{x_0})^2 + (\mathrm{y_i} - \mathrm{y_0})^2 + (\mathrm{z_i} - \mathrm{z_0})^2} = \|\mathbf{x_i} - \mathbf{x}\|, \tag{10}$$

где $\|*\|$ - квадратичная норма, отсюда (11):

$$\boldsymbol{R} = \boldsymbol{f}(\boldsymbol{x}) = \begin{vmatrix} \|\mathbf{x_1} - \mathbf{x}\| \\ \|\mathbf{x_2} - \mathbf{x}\| \\ \|\mathbf{x_3} - \mathbf{x}\| \\ \|\mathbf{x_4} - \mathbf{x}\| \end{vmatrix}. \quad (11)$$

Применим метод наименьших квадратов (МНК), который минимизирует квадратичную норму вектора невязок (12):

$$\left\|\widehat{\boldsymbol{R}} - \boldsymbol{f}(\boldsymbol{x})\right\| \to \min. \quad (12)$$

Для применения МНК найдём производную функции $\boldsymbol{f}(\boldsymbol{x})$, связывающей вектор измерений с вектором состояния, по вектору состояния **x**. Эта производная называется градиентной матрицей:

$$\frac{\partial \mathrm{R_i}}{\partial \mathrm{x_0}} = \frac{-(\mathrm{x_i} - \mathrm{x_0})}{\sqrt{(\mathrm{x_i}-\mathrm{x_0})^2+(\mathrm{y_i}-\mathrm{y_0})^2+(\mathrm{z_i}-\mathrm{z_0})^2}} = \frac{-(\mathrm{x_i} - \mathrm{x_0})}{\|\mathbf{x_i} - \mathbf{x}\|}, \quad (13)$$

отсюда:

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \boldsymbol{H}(\boldsymbol{x}) = \begin{vmatrix} \frac{-(\mathrm{x_1} - \mathrm{x})^{\mathrm{T}}}{\|\mathbf{x_1} - \mathbf{x}\|} \\ \frac{-(\mathrm{x_2} - \mathrm{x})^{\mathrm{T}}}{\|\mathbf{x_2} - \mathbf{x}\|} \\ \frac{-(\mathrm{x_3} - \mathrm{x})^{\mathrm{T}}}{\|\mathbf{x_3} - \mathbf{x}\|} \\ \frac{-(\mathrm{x_4} - \mathrm{x})^{\mathrm{T}}}{\|\mathbf{x_4} - \mathbf{x}\|} \end{vmatrix}, \quad (14)$$

где $\boldsymbol{H}(\boldsymbol{x})$ - градиентная матрица размером 4 x 3.

Находим вектор состояния **x**, пользуясь итеративным алгоритмом, суть которого и есть МНК:

$$\mathbf{x_0} = |0\ 0\ 0|^T, \quad (15)$$

$$\mathbf{x_k} = \mathbf{x_{k-1}} + ((\boldsymbol{H}(\mathbf{x_{k-1}}))^{-1}\boldsymbol{H}(\mathbf{x_{k-1}}))^{-1}(\boldsymbol{H}(\mathbf{x_{k-1}}))^{\mathrm{T}}(\widehat{\boldsymbol{R}} - \boldsymbol{f}(\boldsymbol{x})), \quad (16)$$

где k - номер итерации.

Критерий останова:

$$\|\mathbf{x_k} - \mathbf{x_{k-1}}\| \leq \boldsymbol{\varepsilon}, \quad (17)$$

где ε - требуемая точность.

Сходимость МНК представлена на рисунке 8:

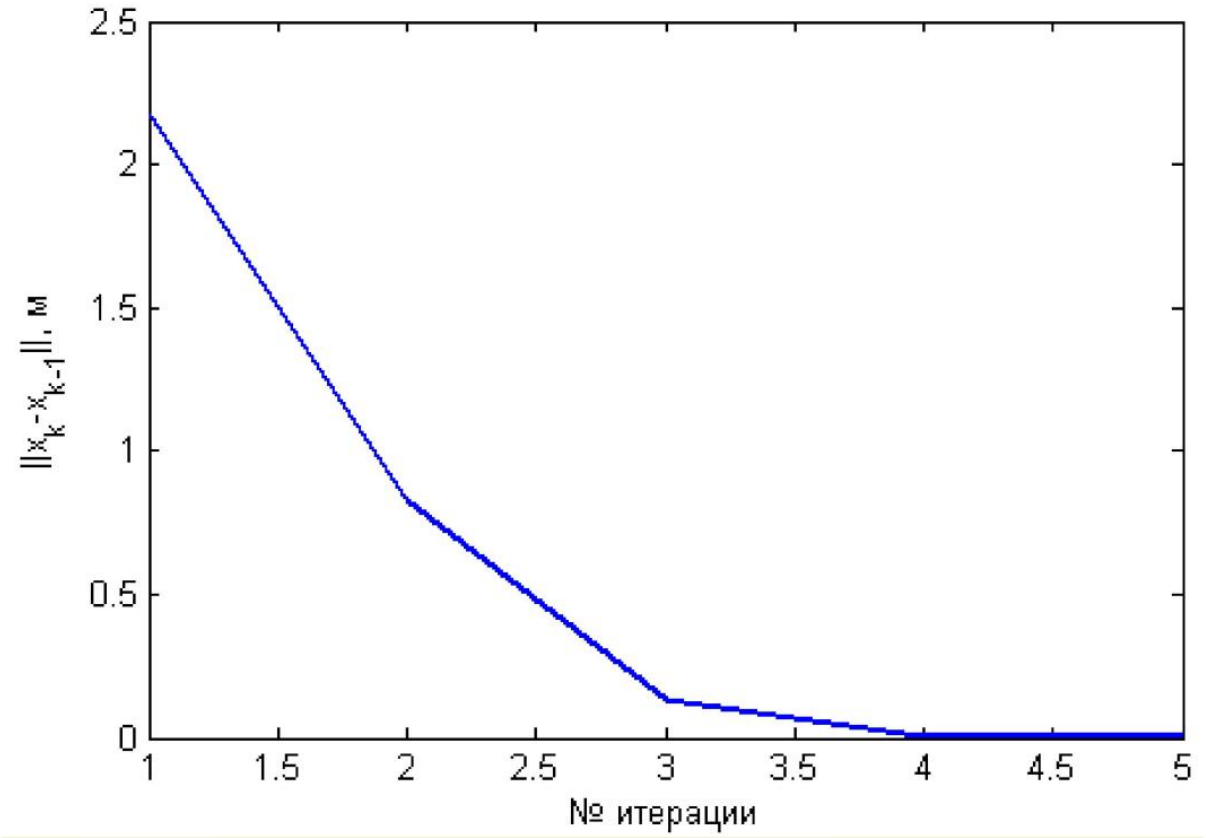

Рисунок 8. Сходимость МНК

Таким образом робот получает решение навигационной задачи для каждого дискретного момента времени, за счёт чего получает свои координаты на плоскости.

2.3 Модель определение ориентации робота

Ориентацию робота-агента в пространстве позволяет определить инерциальный блок. На роботе установлен датчик GY-85 (рисунок 9), на выходе который выдаёт угловые скорости по трём осям от гироскопа, линейные ускорения по 3 осям от акселерометра и показания магнитометра по 3 осям, аналогично работе [19].

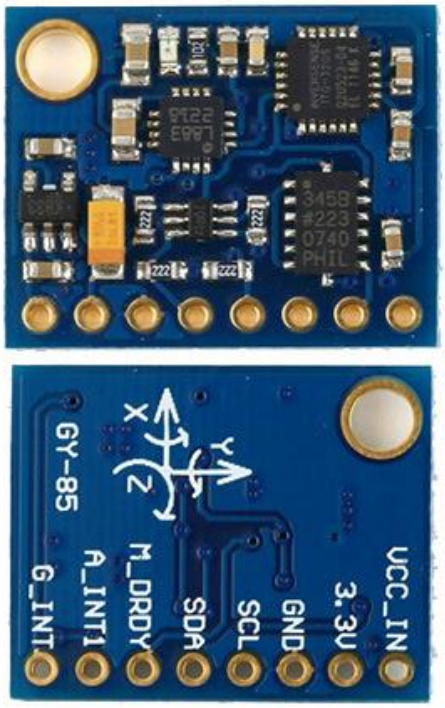

Рисунок 9. Инерциальный блок GY-85

Вычисление угла ориентации робота будем определять по угловой скорости ω, измеренной гироскопом. Вычисления угла на основе данных гироскопа получим путём интегрирования (18):

$$\theta_t = \theta_{t-1} + \omega_t \Delta t, \quad (18)$$

где Δt - временной интервал между измерениями.

Однако угловая скорость ω, измеренная гироскопом, имеет погрешность из-за скорости вращения Земли и отклонения от вертикали. Для коррекции с учётом вращения Земли интегрируем скорость вращения $\Omega_{земли}$ по осям X,Y (19), используя известные координаты (широту ψ и долготу λ) в месте проведения эксперимента:

$$\theta_{земли\ x} = \int_0^t \Omega_{земли} \cos(\psi) dt, \quad (19)$$

$$\theta_{земли\ y} = \int_0^t \Omega_{земли} \sin(\psi)\, dt.$$

Далее проинтегрируем измерения гироскопа (20) за время t - время калибровки и вычтем из полученных углов поправки на вращение Земли (21):

$$\int_0^t \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} \phi_x \\ \phi_y \\ \phi_z \end{pmatrix}, \quad (20)$$

$$\Delta\phi_x = \phi_x - \theta_{\text{земли x}}, \quad (21)$$

$$\Delta\phi_y = \phi_y,$$

$$\Delta\phi_z = \phi_z - \theta_{\text{земли z}}.$$

Вычислим углы после завершения калибровки (22):

$$\Delta_x = \Delta\phi_x t_d\,, \quad (22)$$

$$\Delta_y = \Delta\phi_y t_d\,,$$

$$\Delta_z = \Delta\phi_z t_d\,.$$

Далее с помощью акселерометра определим отклонение оси Z от вертикали, чтобы учитывать влияние ускорения свободного падения g. Для этого проинтегрируем кажущиеся ускорения (23), измеренные акселерометром, получим линейные скорости и вычисли углы отклонения от вертикали (24):

$$\int_0^t \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix}, \quad (23)$$

$$\alpha_{0x} = arctg\left(\frac{V_x}{V_z}\right), \quad (24)$$

$$\alpha_{0y} = arctg\left(\frac{V_y}{V_z}\right).$$

Далее вычислим углы, на которые нужно сделать поворот системы координат гироскопа (25):

$$\alpha_x = \Delta_x + \alpha_{0x}, \quad (25)$$

$$\alpha_y = \Delta_y - \alpha_{0y},$$

$$\alpha_z = \Delta_z.$$

После калибровки, полученные данные гироскопа интегрируем как в (20) на каждый временной промежуток и корректируем согласно вращению Земли и отклонения от вертикали (25):

$$\begin{pmatrix} \phi_x \\ \phi_y \\ \phi_z \end{pmatrix} = \begin{pmatrix} \phi_x - \alpha_x - \theta_{\text{земли x}} \\ \phi_y - \alpha_y - \theta_{\text{земли y}} \\ \phi_z - \alpha_z \end{pmatrix}. \quad (25)$$

Таким образом, робот получает информацию о направлении в каждый момент времени, отталкиваясь от начального направления.

2.4 Выводы по главе

В данной главе представлена математическая модель роя роботов, включающая динамику перемещения, механизмы избежания столкновений и стратегии достижения целевых позиций. Разработка модели основывается на векторах состояния и управления каждого робота, учитывающих координаты, угол ориентации, линейную и угловую скорости. Динамика движения описана с использованием соответствующих уравнений и данных, получаемых с датчиков.

Для избежания столкновений применяются алгоритмы анализа расстояний до статичных и подвижных объектов, используя данные дальномера и системы связи между роботами. Положение робота определяется с помощью локальной навигационной системы, включающей радионавигационные точки и метод наименьших квадратов для вычисления координат. Ориентация робота вычисляется на основе данных инерциального блока, включающего гироскоп, акселерометр и магнитометр.

Таким образом, разработанная модель позволяет описывать поведение роя роботов, обеспечивая корректное определение их положения и ориентации, а также предотвращение столкновений.

ГЛАВА 3 РАЗРАБОТКА ПРОГРАММЫ КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ И АЛГОРИТМОВ РОЕВОГО УПРАВЛЕНИЯ

Для выявления наиболее эффективного по времени алгоритма реализуем программу моделирования мультиагентной системы, состоящей из автономных роботов-агентов, которые действуют в заданной двумерной области для достижения целевых позиций. Каждый робот моделируется как независимая сущность с собственными характеристиками и поведенческой стратегией.

3.1 Разработка среды моделирования

Определим основные сущности и функции, которые должна учитывать среда моделирования:

− Робот: автономный агент, характеризующийся позицией в пространстве (x, y), углом ориентации θ, областью восприятия, ограниченной дальностью действия дальномера, и физическими размерами. Робот способен перемещаться в пространстве, изменяя свою позицию в направлении заданной целевой точки. История перемещений агента сохраняется в память.

− Целевая позиция: конечная точка, к которой должен прибыть робот. Определяется случайно внутри заданной области, что вносит элемент стохастики в поведение робота.

− Модель движения: описывает изменение позиции робота в пространстве согласно формуле (4). В зависимости от текущего положения и угла ориентации, робот перемещается к целевой позиции. Модель может учитывать ограничения скорости и ускорения.

− Алгоритмы восприятия и обхода препятствий: с помощью дальномера робот оценивает расстояние до ближайших объектов, что позволяет ему избегать столкновений, корректируя траекторию движения. Так же робот может принимать вектора состояния других роботов роя.

Для реализации сформулированных выше функций и сущностей среды моделирования был написан код на языке программирования Python. Были применены принципы объектно ориентированного программирования (ООП) для реализации объекта робота-агента и окружающей его среды с входными параметрами:

- начальное положения каждого робота-агента;
- положения препятствий;
- конечное положение каждого робота-агента.

Для того, чтобы можно было смоделировать роботов-агентов с различными параметрами, в объекте робота заданы следующие константы:

- ширина робота;
- длинна робота;
- максимальная дальность обнаружения дальномера;
- максимальная линейная скорость;
- максимальная угловая скорость.

Расставим препятствия таким образом, чтобы роботам для достижения своих целевых позиций было необходимо изменять свои траектории, то есть сделать достаточно сложную окружающую среду. С помощью библиотеки Matplotlib [20] отобразим среду и начальные положения роботов-агентов (рисунок 10), тут синие точки - роботы-агенты, зелёные линии - лучи дальномеров, голубые прямоугольники - статичные препятствия, а красные квадраты - целевые положения роботов.

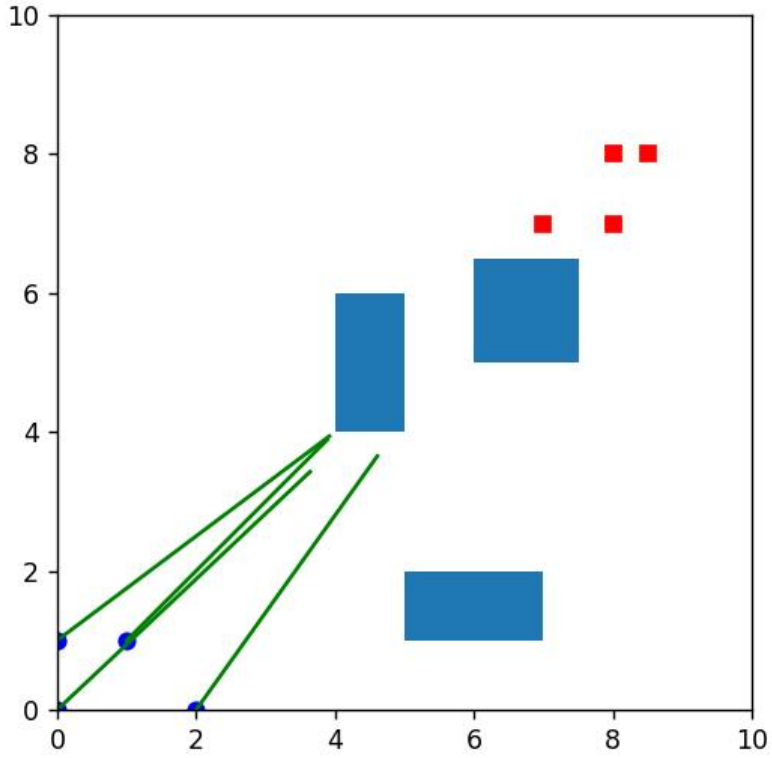

Рисунок 10. Среда моделирования. Начальное положение роботов и препятствий

Далее необходимо реализовать перемещение роботов с дискретным временем. Зададим время между каждым шагом равным 1 секунде и будем обновлять положения роботов на экране с помощью встроенной функции FuncAnimation [21] библиотеки Matplotlib. Данная функция позволяет выполнять некоторый блок программы с заданным временным промежутком. В этом блоке будет заключаться основная логика моделирования мультиагентного взаимодействия.

Реализуем обновление положения всех роботов-агентов по ходу анимации с помощью вызова функции с различными алгоритмами. Таким образом, возможно сравнить время выполнения целевой задачи каждым алгоритмом, так как у всех одинаковые начальные условия. Блок-схема работы программы симуляции представлена на рисунке 11.

Рисунок 11. Блок-схема работы программы симуляции

Красной рамкой выделен блок кода, в котором будет заключён проверяемый алгоритм. Таким образом, разработанная среда симуляции позволит сравнивать различные алгоритмы управления роем и выявить наиболее эффективные.

3.2 Разработка алгоритма, основанного на здравом смысле

Реализуем первый алгоритм, основой которого будет простая логика или здравый смысл. Такое семейство алгоритмов называют Common sense (COM): каждый робот действует по простому алгоритму, продвигаясь к

целевой точке по прямой и объезжая препятствия и других роботов. Положения других роботов алгоритм учитывает только при сближении для избежания столкновений. Такие алгоритмы обычно хорошо работают в простых условиях, не требующих нахождения эффективного решения. На рисунке 12 изображена блок-схема работы алгоритма.

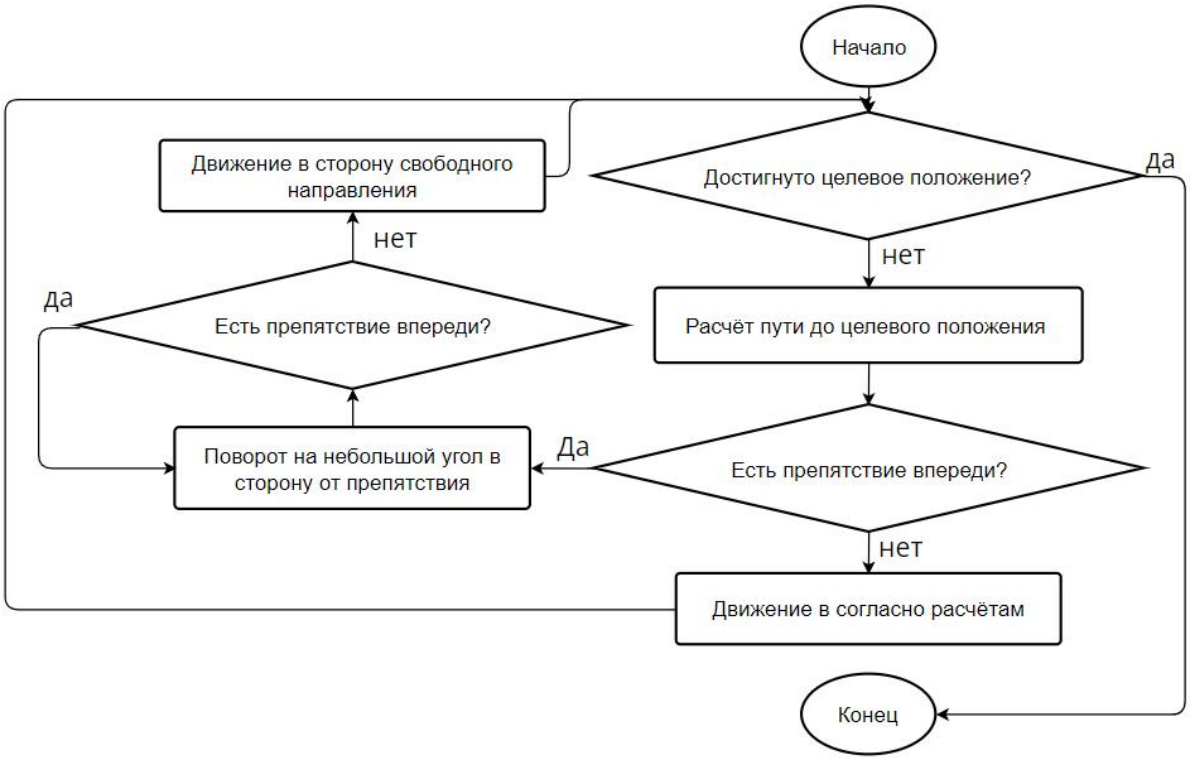

Рисунок 12. Блок-схема работы COM алгоритма

Этот алгоритм встраивается в программу симуляции, описанную выше, на место обозначенное красной рамкой на рисунке 11.

Один из основных аспектов данного алгоритма это блок расчёта пути к целевой позиции. Он разбивается на серию шагов, которые включают в себя движение вперёд и повороты для коррекции направления. Движение можно разложить на две составляющие: поворот с заданной угловой скоростью и движение вперёд с заданной линейной скоростью. Целевая угловая и линейная скорость рассчитывается по формулам (26) и (27):

$$\theta_{target} = arctg\left(\frac{(x_{target} - x_{i-1})}{(y_{target} - y_{i-1})}\right), \quad (26)$$

$$\omega_i = \theta_{target} t_d,$$

$$\omega_i \leq \omega_{max}\,,$$
$$d_{xy} = \sqrt{(x_{target} - x_i)^2 + (y_{target} - y_i)^2}\,, \tag{27}$$
$$\mathrm{V_i} = \frac{d_{xy}}{t_d},$$
$$V_i \leq V_{max},$$
$$x_i = x_{i-1} + V_i cos(\theta_{target}),$$
$$y_i = y_{i-1} + V_i sin(\theta_{target}),$$

где t_d - дискретное время между итерациями симуляции.

После расчёта пути по шагам формируется массив скоростей: линейная и угловая скорость. Следующим шагом является проверка на столкновение с неподвижными и подвижными объектами среды. Проводиться проверка дальности, полученной от дальномера и вычисляется расстояние до других агентов, исходя из их положения на предыдущем шаге согласно формулам (5) и (6). Если проверка показала возможность столкновения, проверяется другое направление движения, сначала близкое с целевым направлением, если они тоже заняты, направление меняем сильнее до того момента, пока не найдётся свободное. Далее меняем целевое направление робота-агента на данном шаге на свободное и проводим расчёт пути заново.

Далее запускаем движение с заданными скоростями согласно формуле (4) на время dt, до следующей итерации цикла симуляции.

Робот-агент останавливается, если достигает целевой позиции или если не может изменить свою позицию на протяжении определённого количества шагов, что указывает на застревание.

Ниже на рисунке 13 представлены траектории движения роя из 4 роботов.

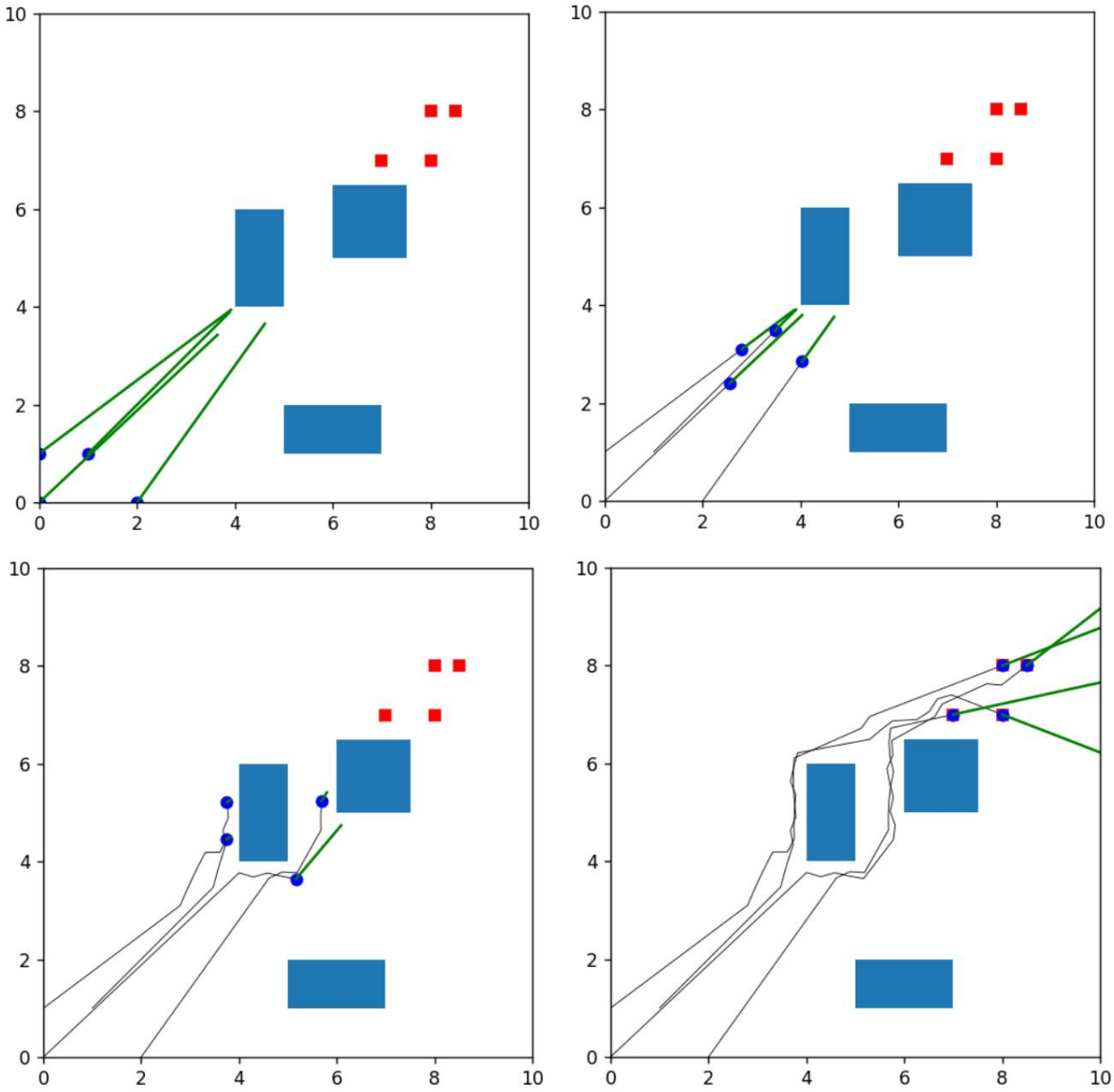

Рисунок 13. Траектории движение роя роботов по COM алгоритму

Время, затраченное на выполнения задачи перемещения всех роботов из начальных положений в целевые COM алгоритмом, составило 13.26 секунд. По траекториям роботов можно заметить, что сначала роботы двигаются по прямой к своим целевым позициям и при достижении первого препятствия расходятся в разные стороны, в зависимости от близости целевой точки для каждого робота-агента. Так же видно, что траектории при обходе препятствий у соседних роботов не совпадают, что может так же влиять на скорость общего выполнения задания. На конечном участке

движения роботы начинают сталкиваться друг с другом, так как некоторые уже закончили движение, а другие все ещё продолжают перемещение. На этом участке роботам-агентам приходиться так же изменять свои траектории, затрачивая дополнительное время.

3.3 Разработка алгоритма, основанного на методе роя частиц

Другим подходом к решению задачи эффективного перемещения роя роботов является создание мультиагентного взаимодействия между роботами. Одним из таких алгоритмов является метод роя частиц - particle swarm optimization (PSO). Алгоритм PSO, применяемый в данной программе, представляет собой вариацию метода роя частиц, адаптированную для оптимизации пути движения роботов в пространстве с препятствиями. В основе метода лежит концепция социального взаимодействия и обмена информацией между частицами (в данном случае роботами-агентами), что позволяет каждой частице адаптироваться и улучшать своё положение в пространстве в соответствии с опытом других частиц. Блок-схема работы алгоритма представлена на рисунке 14.

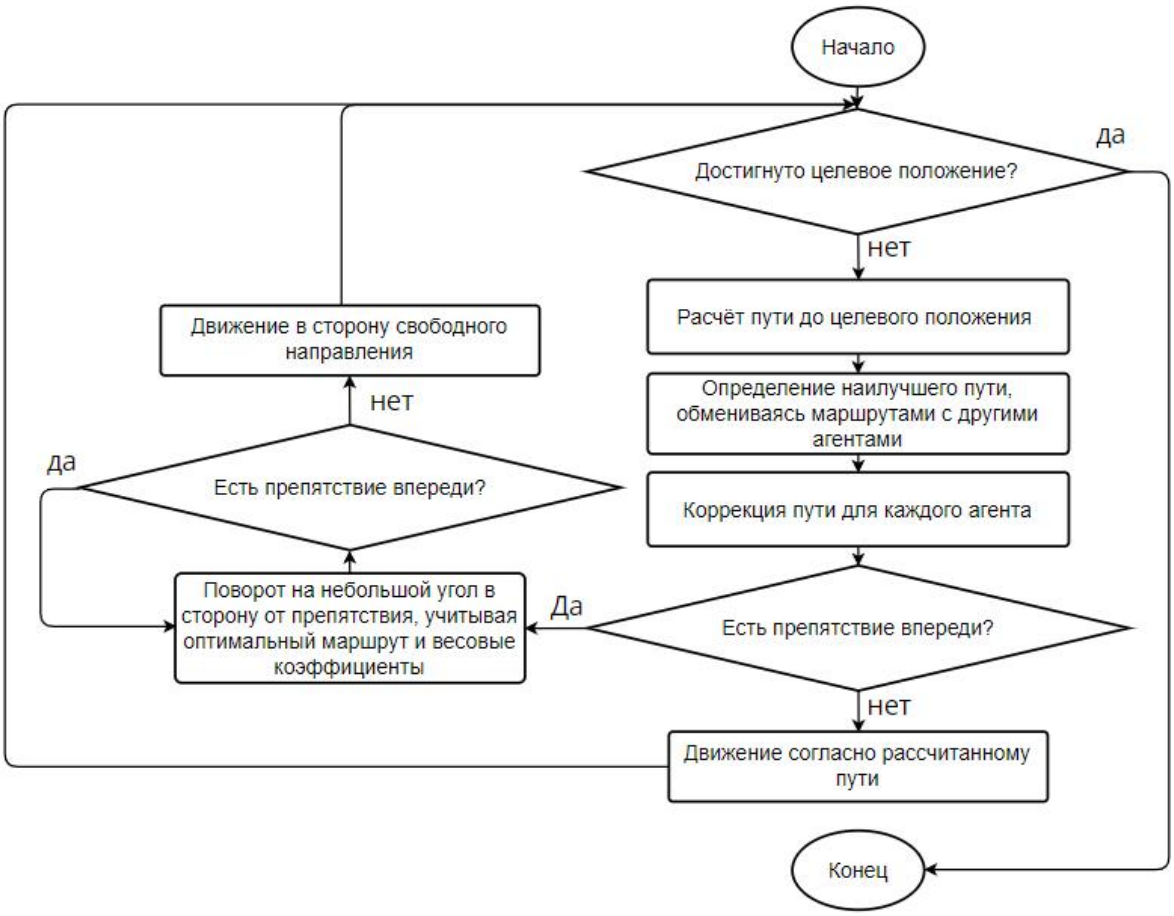

Рисунок 14. Блок-схема работы алгоритма на основе метода роя частиц

Каждый робот-агент в системе описывается своим положением и скоростью, которые динамически обновляются на основе локально и глобально оптимальных найденных решений. Это является ключевым моментом, обеспечивающим эффективность и адаптивность поведения роя в динамически изменяющейся среде. Процесс обновления положения робота базируется на обновлении его скорости, после чего на основе новой скорости вычисляется новое положение. Положение робота обновляется согласно формуле (28):

$$p_{i+1} = p_i + V_{i+1}\Delta t, \tag{28}$$

где p_i - текущее положение робота, V_{i+1} - скорость робота на следующем шаге, которая была обновлена на основании нескольких факторов, Δt - дискретный шаг времени.

Скорость V_{i+1} каждого робота обновляется по следующей формуле (29):

$$V_{i+1} = qV_i + \varphi_p R_p(p_{best} - p_i) + \varphi_g R_g(g_{best} - p_i), \quad (29)$$

где:

– q - коэффициент инерции, который помогает сохранять предыдущую скорость, добавляя стабильность в движение робота;

– φ_p - когнитивный коэффициент, определяющий степень влияния личного опыта робота (лучшее локальное положение p_{best}, к которому робот стремился ранее);

– R_p - случайное число в диапазоне [0, 1], обеспечивающее стохастическую компоненту в векторе к личному лучшему положению;

– φ_g - социальный коэффициент, показывающий влияние глобального опыта роя (лучшее глобальное положение g_{best});

– R_g - случайное число в диапазоне [0, 1], вносящее случайность в вектор к глобальному лучшему положению.

Этот процесс обновления позволяет роботам-агентам адаптироваться к изменениям в среде и реагировать на препятствия, других роботов и динамические изменения в распределении целей. Когнитивная и социальная составляющие обеспечивают сбалансированное сочетание индивидуальной стратегии поиска пути и коллективного поведения, что позволяет рою быстро адаптироваться и находить оптимальные пути в сложных условиях.

Программа также включает механизмы обнаружения столкновений и избежания препятствий. Каждый робот проверяет потенциальные столкновения с препятствиями и другими роботами, что реализовано через функцию, которая определяет, пересекает ли предполагаемая позиция робота какое-либо из препятствий. В случае обнаружения препятствия программа корректирует траекторию, выбирая альтернативное направление движения, чтобы минимизировать вероятность столкновения.

Результатом является то, что каждый робот не только следует за лучшим решением в рое, но и сохраняет способность к самостоятельной адаптации и изучению пространства, что существенно повышает шансы на успешное достижение всех поставленных перед роем целей. На рисунке 15

представлен результат работы алгоритма PSO в виде траекторий роботов-агентов.

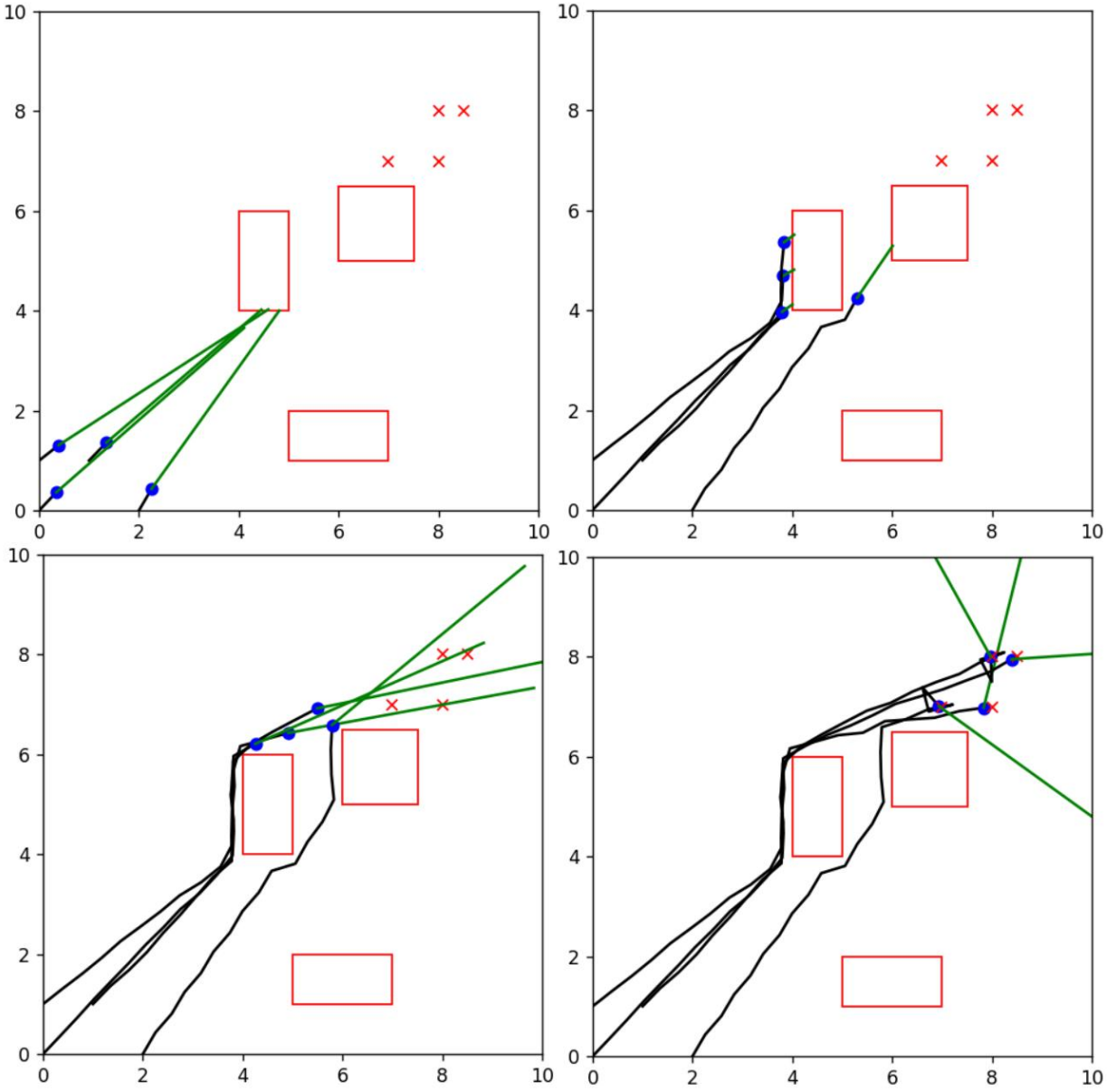

Рисунок 15. Траектории движение роя роботов по PSO алгоритму

Время, затраченное на выполнения задачи перемещения всех роботов из начальных положений в целевые PSO алгоритмом, составило 11.11 секунд. По траекториям роботов-агентов видно, что большинство старается держаться ближе друг к другу, выбирая наиболее эффективный путь для всех. Их траектории стали более сглаженные по сравнению с первым алгоритмом, на что влияет коэффициент инерции робота, делая его более стабильным. Обход препятствий большинство роботов так же делает по одной и той же траектории, что сокращает время, относительно COM алгоритма, где роботы

обходили препятствия не всегда с наиболее эффективной стороны. На последнем участке движения роботы-агенты выбирают наиболее эффективный путь для себе, стараясь не пересекаться с другими. Однако при финишировании роботы начинают делать движения в сторону соседей, считая, что они имеют более эффективный маршрут, что показывают выбросы в траектории около целевых точек. Разработанный алгоритм имеет выигрыш по времени, относительно предыдущего, но тоже имеет недостатки на конечном промежутке пути.

3.4 Разработка муравьиного алгоритма

На основе алгоритма Ant Colony Optimization (ACO), основаного на поведении муравьев в природе, особенно на их способности находить кратчайшие пути от колонии к источнику пищи. В контексте управления роем роботов, этот алгоритм адаптируется для оптимизации маршрутов движения в заданной среде с препятствиями, используя концепцию феромонов для маркировки и выбора оптимальных путей.

Роботы взаимодействуют с феромоновой картой, которая обновляется динамически, отражая текущее состояние истории перемещений всех роботов. На рисунке 16 показана блок-схема алгоритма.

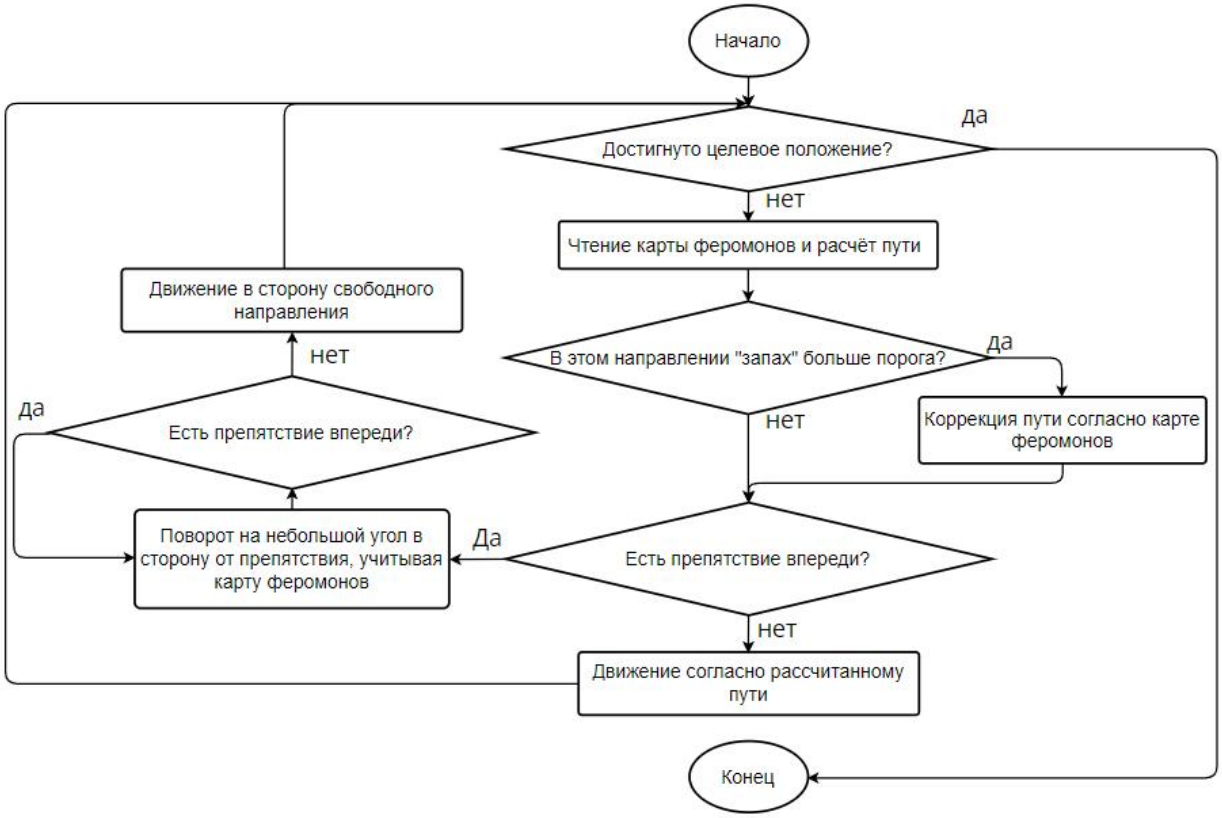

Рисунок 16. Блок-схема муравьиного алгоритма

Метод выбора следующего шага является критически важным аспектом, так как он напрямую влияет на эффективность и оптимальность пути, который прокладывают роботы.

Принятие решений о направлении движения на основе уровней феромонов и расстояния до цели. Робот оценивает возможные позиции для следующего шага, применяя комбинацию двух ключевых факторов: уровень феромонов на потенциальных путях и расстояние до целевой точки. Взаимодействие этих факторов можно выразить следующей формулой (30):

$$S(x,y) = \alpha F(x,y) + \beta \frac{1}{D(x,y)}, \quad (30)$$

где:

– $S(x,y)$ - оценочный балл для клетки с координатами (x, y);

– $F(x,y)$ - уровень феромонов в данной клетке;

– $D(x,y)$- евклидово расстояние от клетки до целевой точки;

– α и β - весовые коэффициенты для феромонов и расстояния соответственно.

Феромоны $F(x, y)$ играют ключевую роль, так как они предоставляют информацию о предыдущих успешных маршрутах, пройденных роботами. Высокий уровень феромонов указывает на предпочтительные пути, которые часто использовались или были определены как эффективные. Это создаёт положительную обратную связь, поощряющую роботов следовать за уже исследованными и успешными маршрутами.

Расстояние $D(x, y)$ обеспечивает баланс, добавляя стратегическую составляющую выбора на основе геометрической близости к цели. Включение обратного значения расстояния обеспечивает тем больший вес этому компоненту, чем меньше расстояние до цели. Это помогает направлять роботов по более короткому пути, когда феромоновые следы отсутствуют или недостаточно сильны.

Выбор значения α и β зависит от конкретных задач и условий окружающей среды. Высокое значение α подчёркивает важность следования за установленными маршрутами, тогда как высокое значение β стимулирует более исследовательское и рискованное поведение, направленное на поиск новых, возможно, более коротких путей к цели.

Процесс оптимизации может включать настройку параметров α и β на основе обратной связи о производительности роя, что позволяет динамически адаптировать поведение роботов под изменяющиеся условия или задачи. Использование методов машинного обучения для адаптации этих параметров в реальном времени представляет собой перспективное направление развития алгоритмов роя. Таким образом, метод выбора следующего шага в ACO обеспечивает сложное взаимодействие между исследовательским поведением и использованием знаний о среде, что делает его мощным инструментом для управления роем роботов в сложных и динамически меняющихся условиях.

После каждого перемещения робот добавляет феромон в текущее местоположение, увеличивая вероятность выбора этого местоположения другими роботами в будущем. Уровень феромонов в каждой точке со

временем уменьшается, что позволяет алгоритму адаптироваться к изменяющимся условиям и избегать чрезмерной фиксации на старых путях.

Перед выбором следующего шага робот проверяет, не приведёт ли перемещение в выбранную точку к столкновению с препятствиями или другими роботами. Если обнаруживается потенциальное столкновение, робот ищет альтернативные маршруты, минимизирующие риск столкновения. На рисунке 17 представлен результат работы алгоритма ACO в виде траекторий роботов-агентов.

Время, затраченное на выполнения задачи перемещения всех роботов из начальных положений в целевые PSO алгоритмом, составило 6.94 секунд. По траекториям движения роботов-агентов видно, что они наиболее эффективны для данной среды. Роботы следуют своему маршруту, пока не замечают «запах» феромонов впереди идущих, после чего корректируют свои маршруты, опираясь на наилучший опыт предыдущих агентов. При обходе препятствий роботы так же выбирают оптимальный путь, что сокращает время относительно предыдущих алгоритмов. На последнем участке роботы-агенты так же двигаются вместе только до определённого момента, после чего заворачивают на свои целевые точки. В отличии от алгоритма PSO, на финише роботы не колеблются вокруг целевых точек, что тоже сокращает время выполнения задачи.

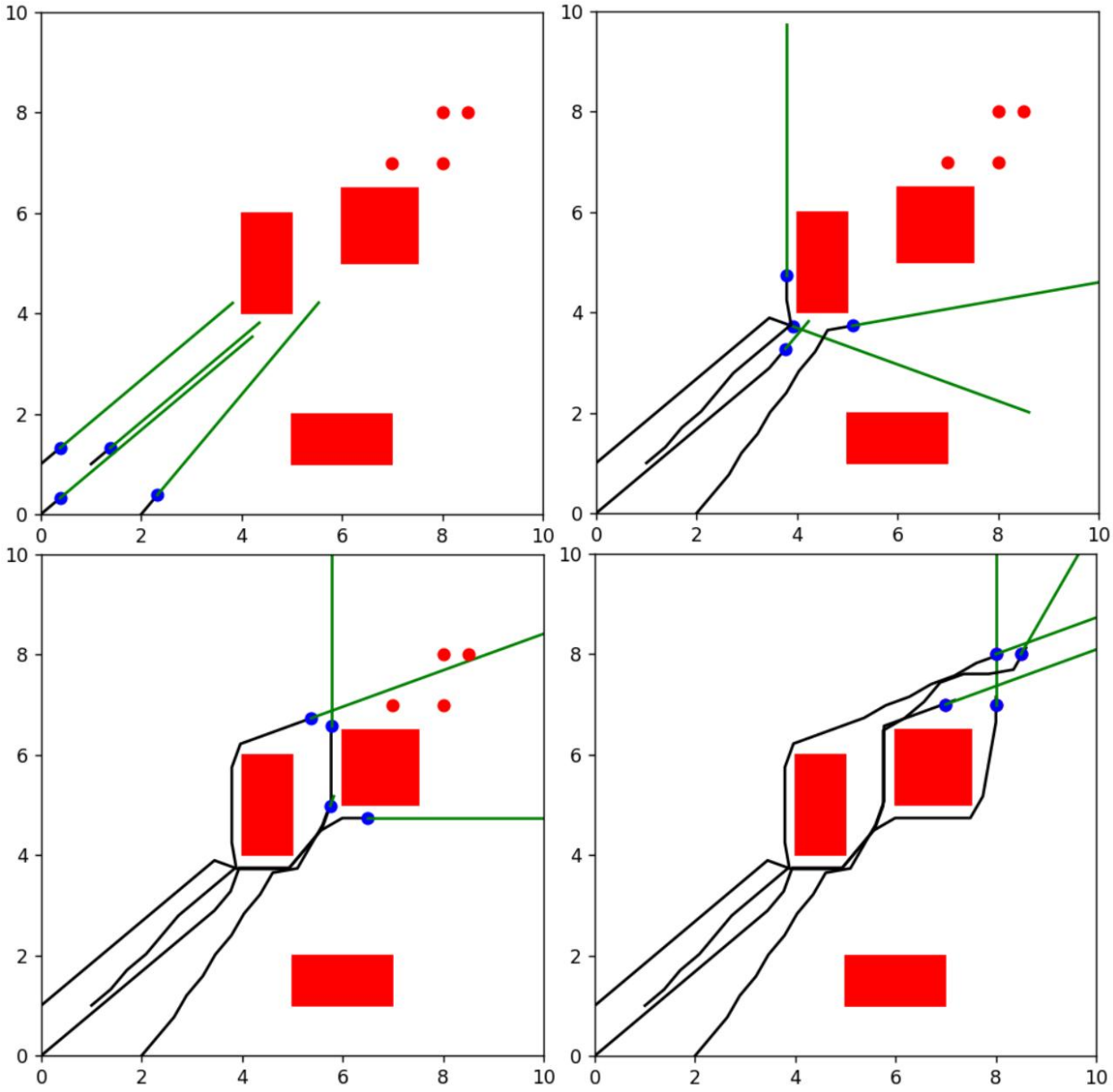

Рисунок 17. Траектории движение роя роботов по ACO алгоритму

Этот подход позволяет рою роботов динамически адаптироваться к изменениям в среде, учитывая как недавние перемещения других роботов, так и статические и динамические препятствия. ACO эффективен в средах, где требуется нахождение оптимальных путей в условиях высокой неопределённости и динамических изменений, предоставляя роботам механизм для оптимизации совместных маршрутов на основе общих опытов и «памяти» окружающей среды.

Для анализа поведения роя при разном количестве роботов-агентов, проведём моделирование для каждого алгоритма с 4, 8 и 16 роботами. Результаты приведены в таблице 1.

Таблица 1. Сравнительная таблица времени выполнения задания при разном количестве роботов-агентов

Количество роботов	Алгоритм		
	COM	PSO	ACO
4 робота	13.26 с	11.11 с	6.94 с
8 роботов	16.03 с	12.90 с	8.62 с
16 роботов	24.56 с	21.03 с	10.84 с

На рисунке 18 изображены траектории роботов при работе алгоритмов с 8 и 16 роботами-агентами.

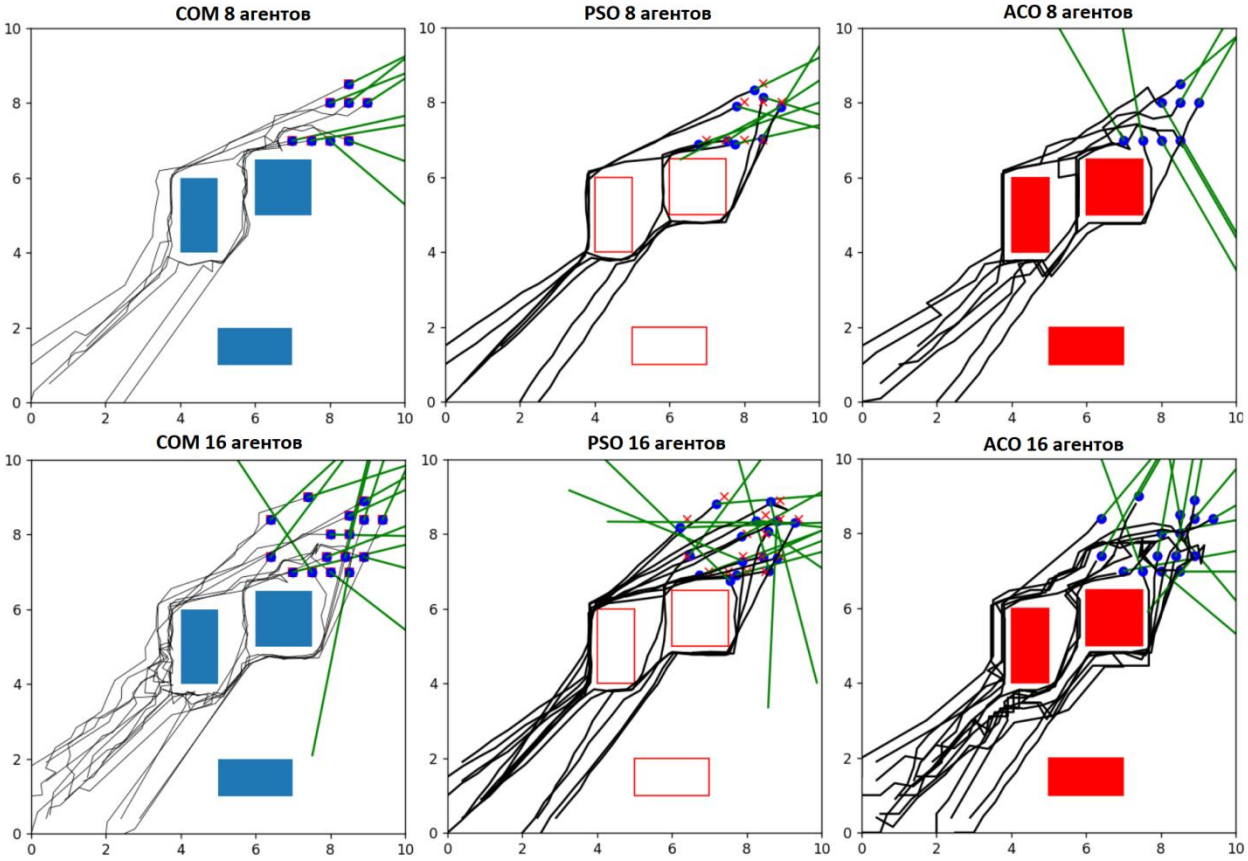

Рисунок 18. Траектории роботов при работе алгоритмов с 8 и 16 роботами-агентами

По полученным данным можно сделать вывод, что при увеличении количества агентов в моделировании время общего выполнения задания увеличивается, при этом лучшим по времени алгоритмом остаётся ACO, дальше идёт PSO и медленнее остальных COM. Зависимость времени выполнения от количества агентов приведена на рисунке 19.

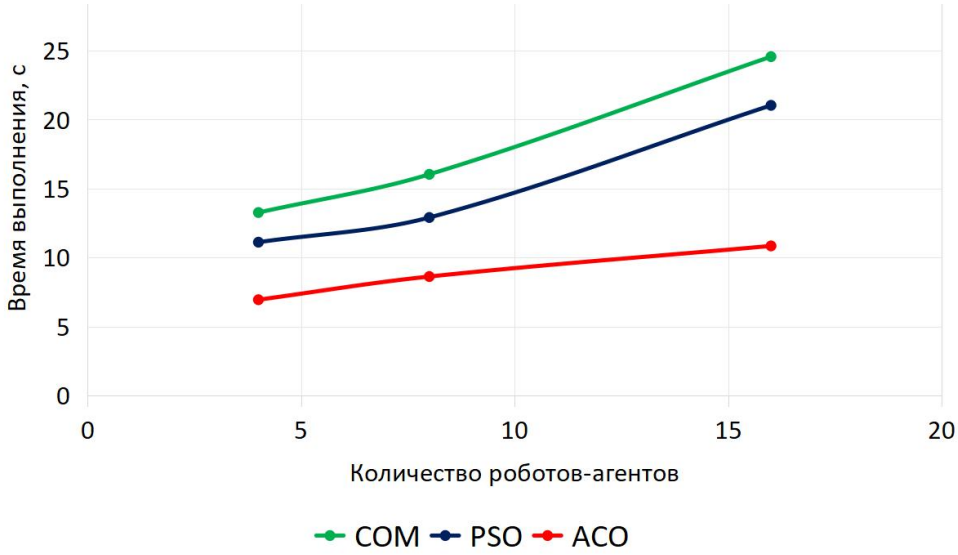

Рисунок 19. Зависимость времени выполнения задания от количества агентов

Для анализа скоростей в каждом алгоритме построим зависимость средней линейной и угловой скорости всех агентов от времени. Возьмём вариант с 16 агентами. Графики зависимости приведены на рисунках 20, 21, 22.

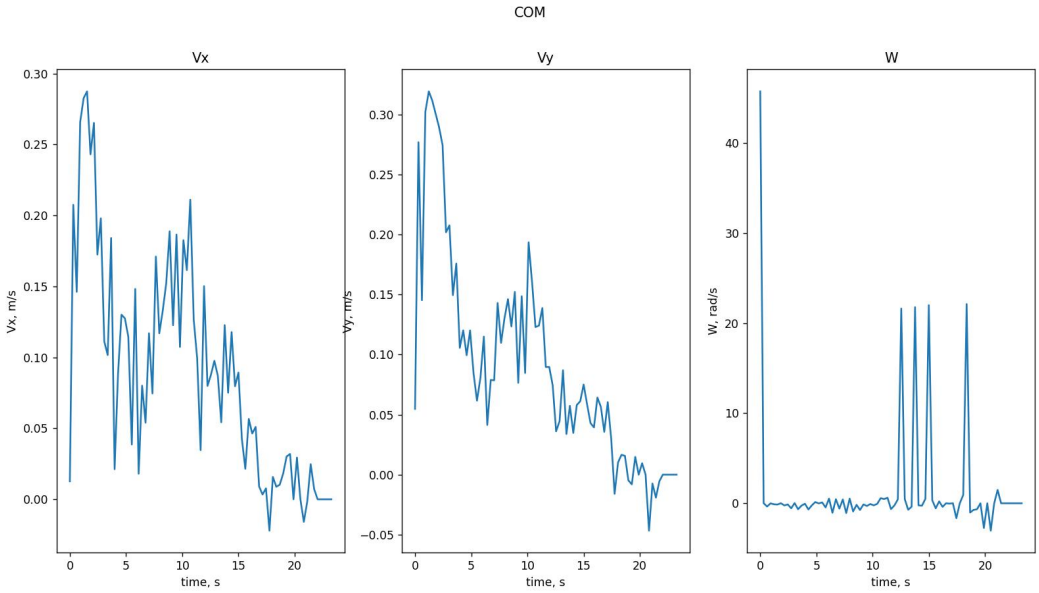

Рисунок 20. Зависимость средних скоростей роботов-агентов в COM алгоритме

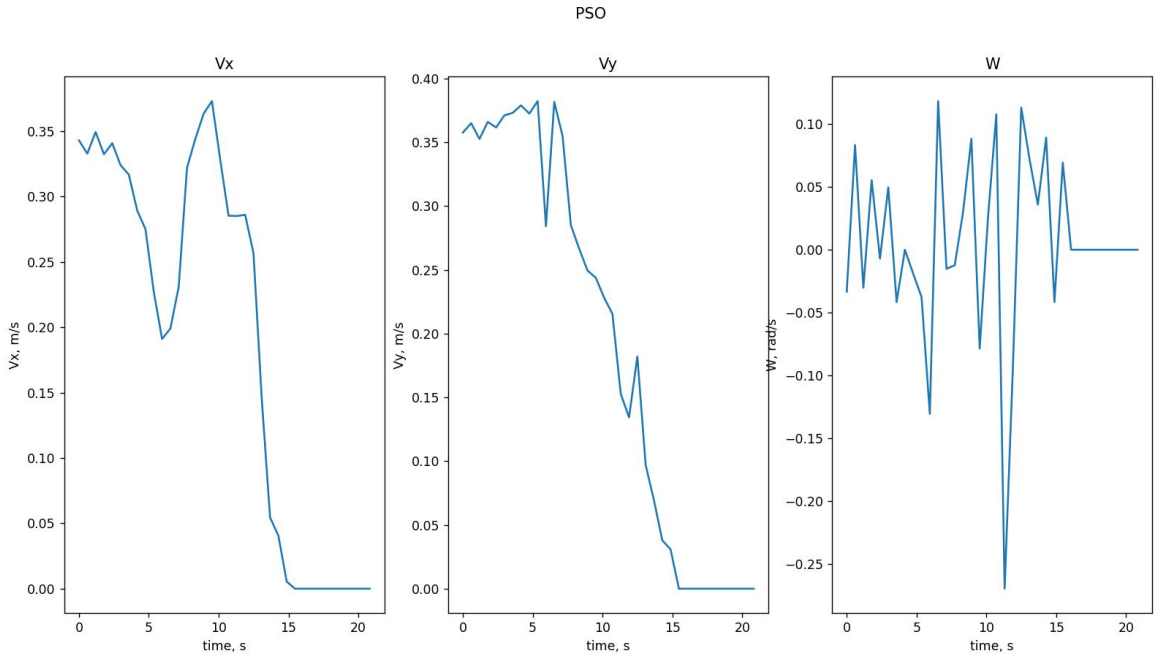

Рисунок 21. Зависимость средних скоростей роботов-агентов в PSO алгоритме

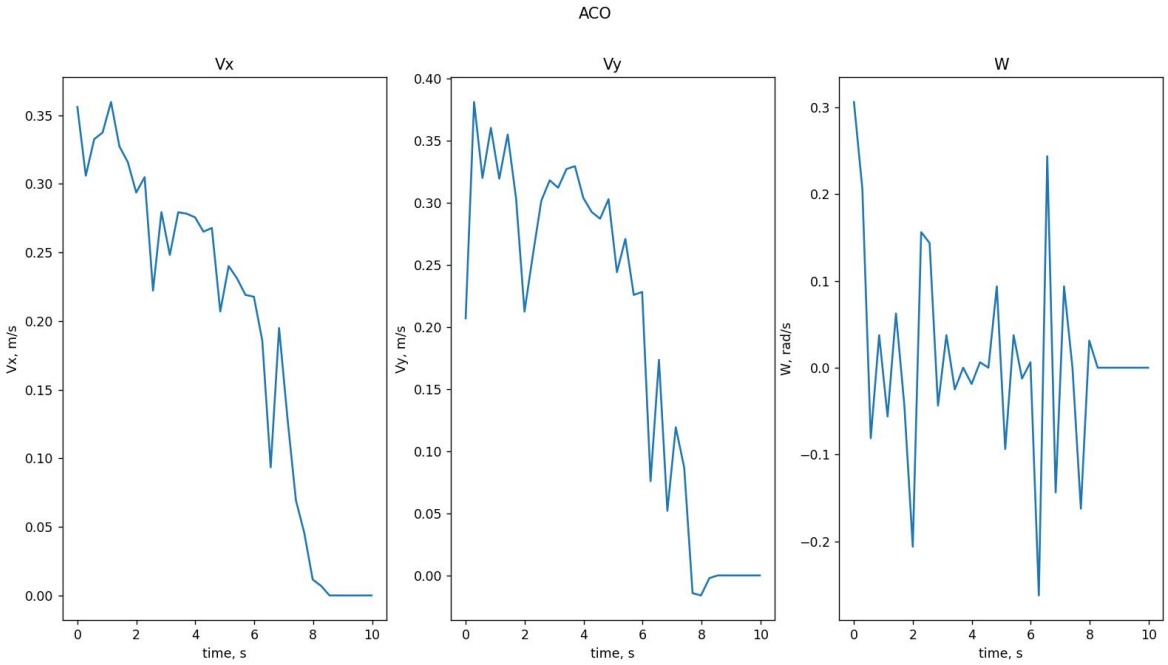

Рисунок 22. Зависимость средних скоростей роботов-агентов в ACO алгоритме

По графикам можно сделать вывод, что сначала скорости всех роботов близки к максимальной, дальше при обходе препятствий скорость начинает колебаться и к финишу снижается до нуля. В алгоритме COM график наиболее ломаный, что подтверждает наличие лишних действий у роботов, за

счёт чего время выполнения задания у этого алгоритма больше остальных. Графики линейных скоростей у алгоритмов PSO и ACO схожи, но колебания угловых скоростей больше у алгоритма ACO. По этим данным можно сделать вывод, что в алгоритме ACO роботы быстрее выбирают нужное направление, что подтверждает наименьшее время выполнения задания.

3.5 Выводы по главе

В третьей главе дипломной работы рассмотрена разработка программы компьютерного моделирования и алгоритмов роевого управления для анализа эффективности мультиагентных систем, состоящих из автономных роботов-агентов, действующих в двумерной области для достижения целевых позиций. Листинги программ симуляции по трём алгоритмам представлены в приложении А. Основное внимание уделено разработке среды моделирования, определению основных сущностей и функций, таких как роботы, их целевые позиции и модель движения, а также алгоритмы восприятия и обхода препятствий. Реализация программы на Python с использованием принципов ООП позволила визуализировать среду и процессы взаимодействия роботов, что существенно облегчило сравнение и анализ эффективности различных алгоритмов управления.

В ходе работы были разработаны и протестированы три алгоритма: алгоритм на основе здравого смысла (COM), метод роя частиц (PSO) и муравьиный алгоритм (ACO). Каждый из алгоритмов демонстрирует различные подходы к решению задачи управления движением роботов, отражая как простые стратегии обхода препятствий и движения к цели, так и сложные взаимодействия на основе социальных и когнитивных компонентов.

Анализ мультиагентных алгоритмов демонстрирует значительные различия в эффективности разработанных алгоритмов управления. Сравнительный анализ показал, что для 4 агентов алгоритмы PSO и ACO значительно превосходят COM алгоритм по времени выполнения задачи: PSO сократил время на 16%, а ACO — на 47%, сравнительно с COM

алгоритмом, который затратил 13.26 секунд на выполнение задачи. Также, алгоритмы PSO и ACO продемонстрировали более гладкие и координированные траектории движения роботов, что свидетельствует о их способности эффективно реагировать на динамические изменения в распределении препятствий и целей в среде. Муравьиный алгоритм ACO показал наилучшие результаты не только в скорости, но и в стабильности маршрутов, минимизируя необходимость частых корректировок траектории, что делает его предпочтительным выбором для сложных динамических сред. Эти выводы аналогичны так же и для другого количества агентов в моделировании.

По графикам зависимостей линейных и угловых скоростей можно сделать вывод, что наиболее эффективными являются алгоритмы PSO и ACO, так как они имеют меньшие колебания по сравнению с COM алгоритмом.

ГЛАВА 4 ПРОВЕДЕНИЕ НАТУРНОГО ЭКСПЕРИМЕНТА С РОЕМ РОБОТОВ НА МАКЕТЕ ГОРОДСКОЙ ИНТЕЛЛЕКТУАЛЬНОЙ ТРАНСПОРТНОЙ СИСТЕМЫ

Для подтверждения результатов моделирования был проведён натурный эксперимент на роботах макета городской ИТС платформы «Робовейник». Внешний вид роботов представлен на рисунке 5. Эксперимент включал в себя адаптацию программного обеспечения, используемого в симуляции, под условия реального мира, что потребовало внесения изменений в обмен сообщениями между роботами, навигационную систему, инерциальные датчики, а также процессы запуска программы и записи времени.

4.1 Разработка алгоритмов роевого управления для натурного эксперимента

В отличие от симуляции, в реальной среде была реализована более сложная система обмена сообщениями между роботами, обеспечивающая передачу данных о положении и состоянии каждого робота, схема обмена сообщениями представлена на рисунке 2. Это включало в себя интеграцию протоколов связи WebSocket для уменьшения задержек и повышения надёжности обмена информацией. Навигационная система была усовершенствована для работы в условиях реальной среды: использовались UWB - радиомодули для точного определения координат роботов на макете. Инерциальные датчики, включая акселерометры и гироскопы, обеспечивали стабильное определение ориентации роботов в пространстве. Алгоритмы навигации и определения ориентации роботов, описанные в пункте 2.2, были реализованы на языке Python. Полный листинг программы представлен в приложении Б. Запуск программы и запись времени выполнялись с использованием бортовых микрокомпьютеров Raspberry Pi 4, что позволило

запускать код всех роботов-агентов одновременно, а также автоматически фиксировать время выполнения общей задачи.

Эксперимент проводился на макете городской ИТС (схема представлена на рисунке 6), который представлял собой уменьшенную копию городской среды с препятствиями в виде макетов домов. Внешний вид макета и роботов представлен на рисунке 23.

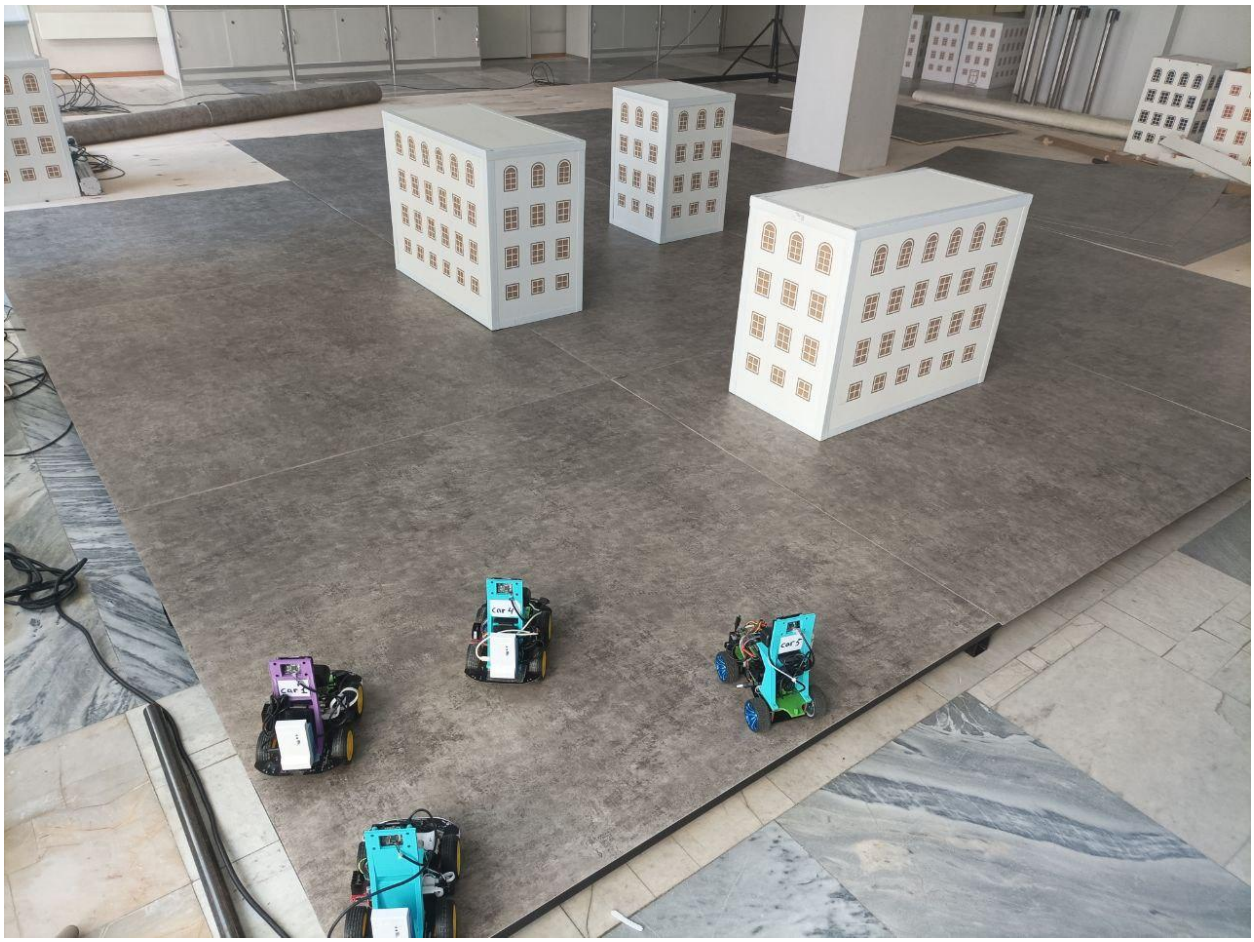

Рисунок 23. Внешний вид макета ИТС и роботов

Для отслеживания траектории движения роботов их координаты записывались в файл, для того чтобы после выполнения мультиагентной задачи их визуализировать для дальнейшего анализа. Аналогично координатам записывались так линейные и угловые скорости каждого робота-агента. Для сравнения с результатами моделирования был проведён эксперимент с 4 роботами. Результаты представлены в сравнительной таблице 2.

Таблица 2. Сравнительная таблица результатов натурного эксперимента

Количество роботов	Алгоритм		
	COM	PSO	ACO
4 робота	14.06 с	12.51 с	10.28 с

Результаты эксперимента показали, что алгоритмы PSO и ACO, как и в моделировании, продемонстрировали наилучшие результаты по времени выполнения задачи и качеству траекторий. Время выполнения задачи алгоритмом PSO составило 12.51 секунд, а алгоритмом ACO – 10.28 секунды, что подтверждает данные, полученные в симуляции. Алгоритм на основе здравого смысла (COM) показал наихудшие результаты, затратив 14.06 секунд на выполнение задачи.

Траектории движения роботов (рисунок 24) и графики зависимости скоростей от времени (рисунки 25, 26, 27), полученные в результате натурного эксперимента, аналогичны траекториям и графикам, полученным в процессе моделирования. Роботы, работающие по алгоритму COM, демонстрировали более резкие и неоптимальные траектории, часто изменяя направление движения при обнаружении препятствий, что показывают графики зависимости скоростей. В то время как роботы, использующие алгоритмы PSO и ACO, двигались по более плавным и оптимизированным траекториям, эффективно избегая препятствия и минимизируя необходимость частых корректировок маршрута.

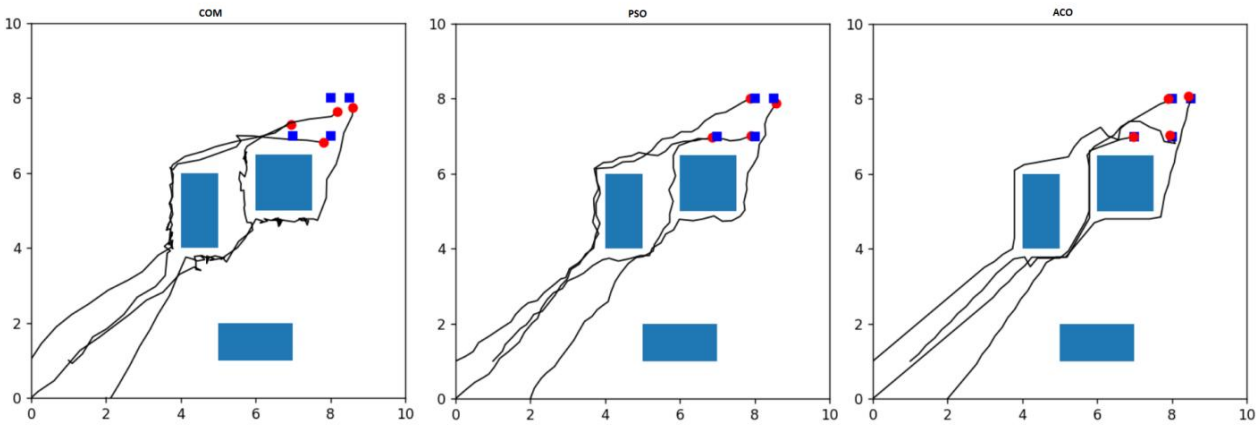

Рисунок 24. Траектории роботов-агентов, полученные экспериментально

Остановка роботов не точно в целевых точках обусловлена погрешностью навигационной системы, порядок погрешности UWB радиомодуля 0.2 метра.

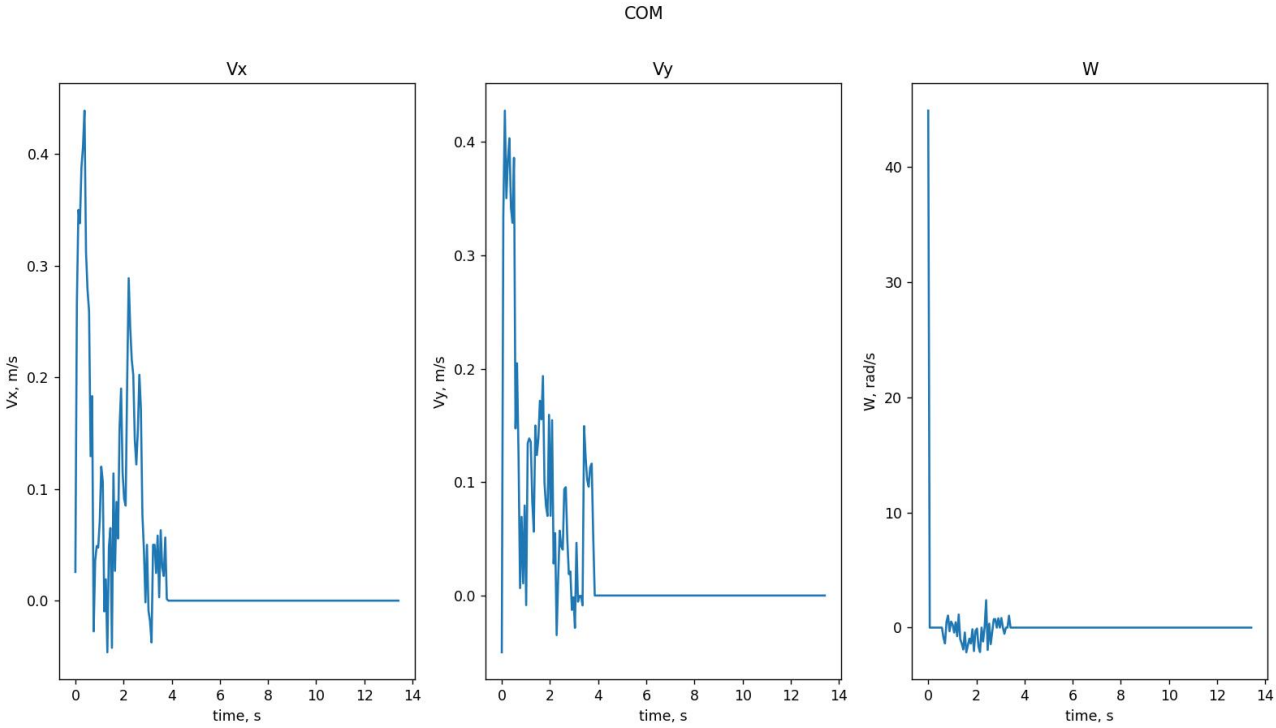

Рисунок 25. Графики зависимостей скоростей от времени COM алгоритма, полученные экспериментально

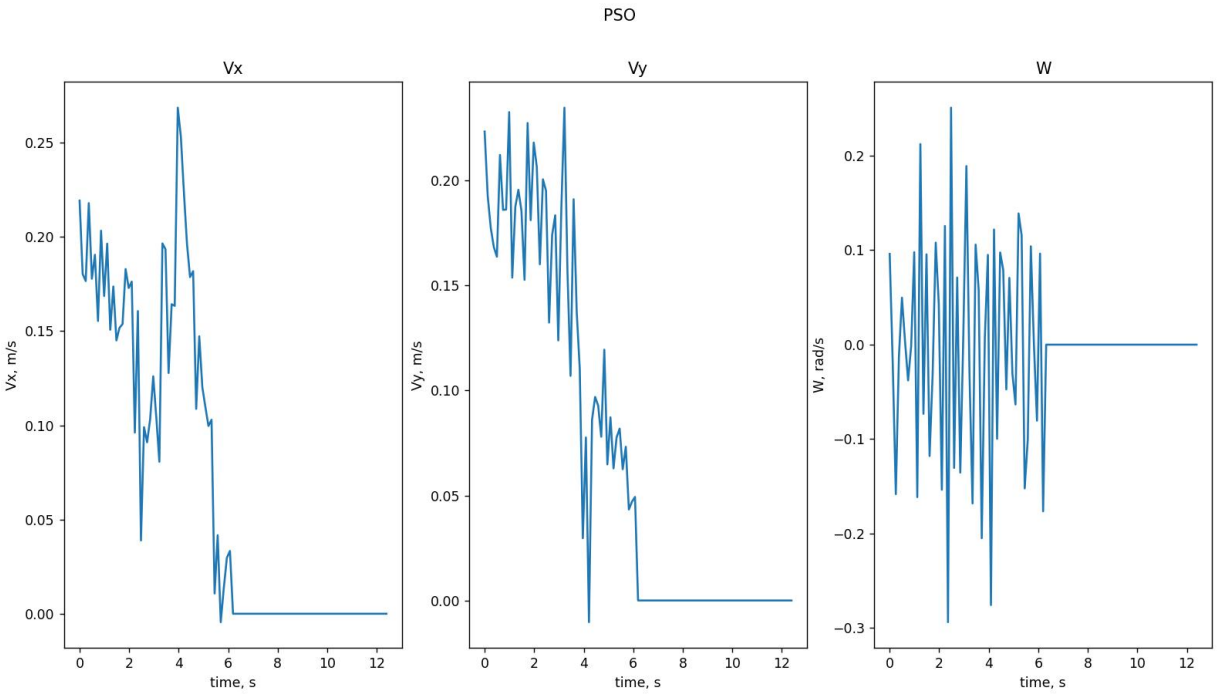

Рисунок 26. Графики зависимостей скоростей от времени PSO алгоритма, полученные экспериментально

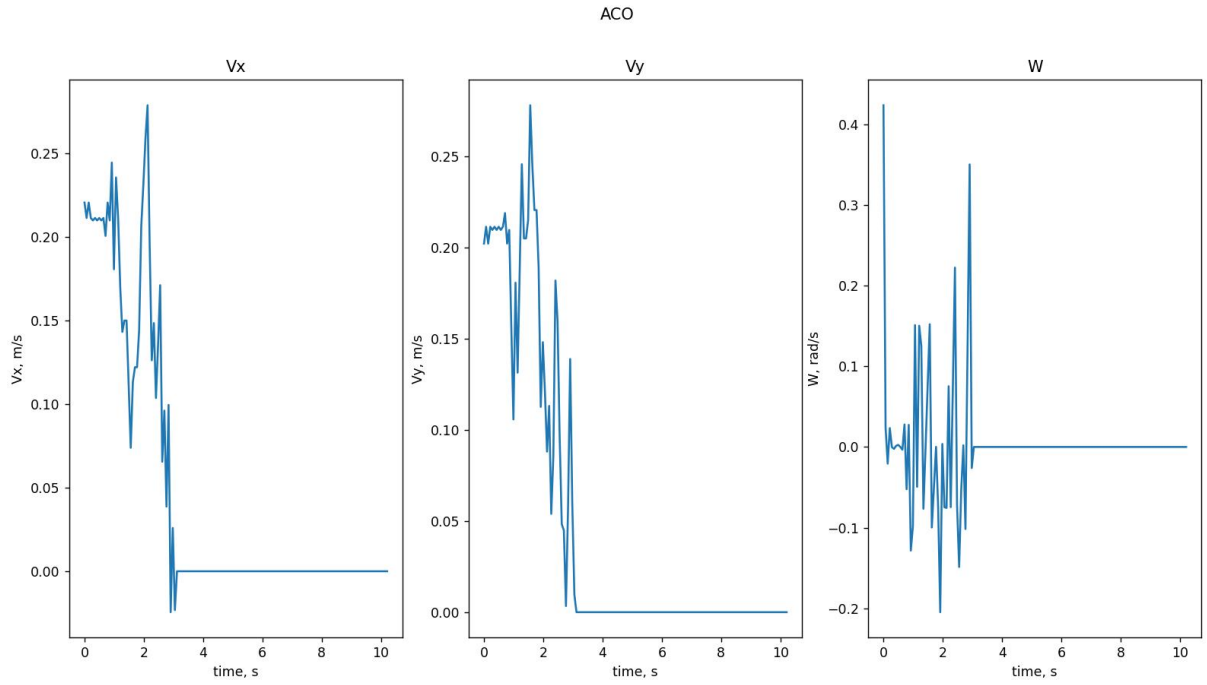

Рисунок 27. Графики зависимостей скоростей от времени ACO алгоритма, полученные экспериментально.

4.2 Выводы по главе

В четвёртой главе дипломной работы проведён натурный эксперимент, отличия между результатами моделирования и эксперимента были минимальными и связаны в основном с непредвиденными задержками в обмене сообщениями и неточностями навигационных датчиков в реальной среде. Эти факторы были учтены при разработке, и их влияние было минимизировано в рамках возможностей аппаратного обеспечения. Основное влияние оказывала неточность поворотов по ИНС. По траекториям всех трёх алгоритмов видно, что роботы делают более неточные повороты, за счёт чего увеличивается время выполнения задания.

На основании проведённых экспериментов можно сделать вывод, что предложенные алгоритмы мультиагентного взаимодействия демонстрируют высокую эффективность в реальных условиях, аналогично результатам моделирования. Алгоритм ACO показал наилучшую стабильность маршрутов и минимальное время выполнения задачи, что делает его предпочтительным для использования в сложных динамических средах. Алгоритм PSO также продемонстрировал хорошие результаты, уступая ACO в стабильности, но превосходя COM по всем параметрам.

Таким образом, проведённый натурный эксперимент подтвердил эффективность разработанных алгоритмов и открыл перспективы для их дальнейшего совершенствования и применения в различных областях робототехники и автоматизации. Данная работа показала, что применение алгоритмов роевого управления позволяет значительно повысить эффективность и координацию работы роя роботов, что является важным шагом на пути к созданию более автономных и интеллектуальных систем.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы была выполнена разработка и исследование алгоритмов управления роем роботов для решения задачи нахождения кратчайшего пути каждым роботом-агентом.

В работе была сформулирована мультиагентная задача управления, представлена кинематическая и принципиальная схема робота. Была создана математическая модель робота-агента, включающая описание его положения и ориентации. Рассматривались механизмы избежания столкновений, стратегии достижения целевых позиций и алгоритм решения навигационной задачи. Была разработана программная среда для моделирования мультиагентной задачи и тестирования алгоритмов роевого управления. Были рассмотрены три алгоритма мультиагентного взаимодействия: алгоритм на основе здравого смысла (COM), метод роя частиц (PSO) и муравьиный алгоритм (ACO). Проведено моделирование этих алгоритмов и анализ их результатов.

Заключительная часть работы содержит описание натурного эксперимента, проведённого на макете городской интеллектуальной транспортной системы платформы «Робовейник», на основании которого можно сделать вывод, что разработанные и протестированные алгоритмы мультиагентного взаимодействия успешно решают задачу нахождения кратчайшего пути для роя роботов.

Таким образом, в ходе работы были решены все поставленные задачи: разработана математическая модель системы, программная среда для моделирования и проведён натурный эксперимент. Настоящая работа открывает перспективы для дальнейших исследований и совершенствования алгоритмов управления роями роботов, что может найти широкое применение в различных областях робототехники и автоматизации.

СПИСОК ЛИТЕРАТУРЫ

1 В. И. Городецкий, О. В. Карсаев, В. В. Самойлов, С. В. Серебряков // Прикладные многоагентные системы группового управления, Искусственный интеллект и принятие решений, 2009, выпуск 2, 3–24.

2 Federico Marini, Beata Walczak Particle swarm optimization // A tutorial, Chemometrics and Intelligent Laboratory Systems. Volume 149, Part B, 2015, Pages 153-165, ISSN 0169-7439, https://doi.org/10.1016/j.chemolab.2015.08.020.

3 Е. Н. Шварева, Л. В. Еникеева, И. М. Губайдуллин Оптимизация роем частиц // Уфа : Уфимский государственный нефтяной технический университет, 2021. – 82 с. – ISBN 978-5-7831-2105-0. – EDN JIVVRU.

4 M. Dorigo, M. Birattari T., Stutzle Ant colony optimization // IEEE Computational Intelligence Magazine, vol. 1, no. 4, pp. 28-39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

5 Еременко Ю.И., Цуканов М.А., Соловьев А.Ю. О применении мультиагентных алгоритмов муравьиных колоний для решения задачи структурной оптимизации в энергетических системах // Фундаментальные исследования. – 2013. – № 10-15. – С. 3316-3320.

6 Dorigo M., Stutzle T. Ant Colony Optimization // MIT Press, Cambridge (2004) A Critical Analysis of Parameter Adaptation in Ant Colony Optimization. IRIDIA – Technical Report Series

7 Гладков Л. А., Гладкова Н. В. Эволюционирующие многоагентные системы и эволюционное проектирование // Известия ЮФУ. Технические науки. 2020. №4 (214). URL: https://cyberleninka.ru/article/n/evolyutsioniruyuschie-mnogoagentnye-sistemy-i-evolyutsionnoe-proektirovanie (дата обращения: 09.04.2024).

8 Жданкина, С. А. Применение алгоритмов роевого интеллекта для решения задачи коммивояжёра // Наука. Технологии. инновации : Сборник научных трудов XVII Всероссийской научной конференции молодых ученых. В 11-ти частях, Новосибирск, 04–08 декабря 2023 года. – Новосибирск:

Новосибирский государственный технический университет, 2024. – С. 54-58. – EDN LXHUDE.

9 Aadesh Neupane, Michael A., Goodrich Eric G. Mercer Grammatical evolution algorithm for evolution of swarm behaviors // In Proceedings of the Genetic and Evolutionary Computation Conference 2018 GEESE. Association for Computing Machinery, New York, NY, USA, 999–1006. https://doi.org/10.1145/3205455.3205619.

10 Md. Arafat Hossain, Israt Ferdous, Autonomous robot path planning in dynamic environment using a new optimization technique inspired by bacterial foraging technique // Robotics and Autonomous Systems, Volume 64,2015,Pages 137-141, ISSN 0921-8890, https://doi.org/10.1016/j.robot.2014.07.002.

11 Baykasoglu A., Ozbakir L., Tapkan P. Artificial bee colony algorithm and its application to generalized assignment problem // Swarm Intelligence: Focus on Ant and particle swarm optimization. –2007.– Т. 1.

12 В. Б. Мелехин, М. В. Хачумов Планирование поведения автономных интеллектуальных мобильных систем в условиях неопределенности // Санкт-Петербург : Издательство "Политехника", 2022. – 274 с. – ISBN 978-5-7325-1193-2. – DOI 10.25960/7325-1193-2. – EDN HLSQHR.

13 К. Д. Крестовников, А. Р. Шабанова, А. Д. Ковалев Математическая модель роевой робототехнической системы с беспроводной двусторонней передачей энергии // Труды Научно-исследовательского института радио. – 2020. – № 1-2. – С. 64-73. – DOI 10.34832/NIIR.2020.1.1.007. – EDN ZBUEVH.

14 Официальный сайт платформы «Робовейник» https://landing.hackmpei.ru/ (дата обращения: 22.05.2024)

15 Педжман Рошан, Джонатан Лиэри Основы построения беспроводных локальных сетей стандарта 802.11. Руководство Cisco = 802.11 Wireless Local-Area Network Fundamentals. — М.: «Вильямс», 2004. — С. 304. — ISBN 5-8459-0701-2

16 I. Fette, A. Melnikov, The WebSocket Protocol // Internet Engineering Task Force (IETF) Google, Inc. Category: Standards Track ISSN: 2070-1721Isode Ltd. December 2011

17 Y. Cheng T. Zhou UWB Indoor Positioning Algorithm Based on TDOA Technology. pp. 777-782, 2019.

18 O. V. Glukhov, I. A. Akinfiev, A. D. Razorvin, A. A. Chugunov, D. A. Gutarev and S. A. Serov Loosely Coupled UWB/Stereo Camera Integration for Mobile Robots Indoor Navigation // 2023 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE), Moscow, Russian Federation, 2023, pp. 1-7, doi: 10.1109/REEPE57272.2023.10086807.

19 Андреев В.Д. Теория инерциальной навигации. (Автономные системы). // Издательство «Наука», 1966.

20 Mathplotlib Официальный сайт https://matplotlib.org/ (дата обращения: 22.05.2024)

21 FuncAnimation Mathplotlib Официальный сайт https://matplotlib.org/stable/api/animation_api.html#funcanimation (дата обращения: 22.05.2024)

ПРИЛОЖЕНИЕ А

Листинг программы моделирования COM алгоритма

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib.patches as patches
import time

pos = []
sec = 0

class Rectangle(patches.Rectangle):
    def distance(self, point):
        closest_x = max(self.get_x(), min(point[0], self.get_x() + self.get_width()))
        closest_y = max(self.get_y(), min(point[1], self.get_y() + self.get_height()))
        return np.sqrt((point[0] - closest_x) ** 2 + (point[1] - closest_y) ** 2)

    def intersects(self, other_rect):
        return self.get_x() < other_rect.get_x() + other_rect.get_width() and \
               self.get_x() + self.get_width() > other_rect.get_x() and \
               self.get_y() < other_rect.get_y() + other_rect.get_height() and \
               self.get_y() + self.get_height() > other_rect.get_y()

class Robot:
    def __init__(self, x, y, angle, target, obstacles, robots=None):
        self.x = x
        self.y = y
        self.x0 = x
        self.y0 = y
        self.ang0 = 0
        self.angle = angle
        self.lidar_range = 5
        self.width = 0.4
        self.max_speed_per_step = 0.5  # максимальная скорость 0.5 метра в секунду
        self.path = [(x, y)]
        self.target_position = target
        self.obstacles = obstacles
        self.robots = robots if robots is not None else []  # Добавляем проверку на robots
        self.steps = self.calculate_path_steps(self.target_position)
        self.step_index = 0
        self.stop = False
        self.stuck_steps = 0
        self.last_position = (x, y)

    def move(self):
        if self.step_index < len(self.steps) and not self.stop:
            distance, angle_change = self.steps[self.step_index]
            self.angle = (self.angle + angle_change) % 360
            rad_angle = np.deg2rad(self.angle)
            proposed_x = self.x + distance * np.cos(rad_angle)
            proposed_y = self.y + distance * np.sin(rad_angle)

            if not self.check_collision(proposed_x, proposed_y, self.robots):
                if (proposed_x, proposed_y) == self.last_position:
                    self.stuck_steps += 1
                else:
```

```
                self.stuck_steps = 0
            self.x, self.y = proposed_x, proposed_y
            self.path.append((self.x, self.y))
            self.step_index += 1
            self.last_position = (self.x, self.y)
        else:
            self.avoid_collision()

        if self.stuck_steps > 10:
            self.stop = True

    def check_collision(self, x, y, robots):
        for obstacle in self.obstacles:
            if obstacle.intersects(Rectangle((x - self.width/2, y - self.width/2), self.width, self.width)):
                return True
        for obstacle in [(r.x, r.y) for r in robots if r != self]:
            if np.hypot(x - obstacle[0], y - obstacle[1]) <= self.width / 2 + 0.1:
                return True
        return False

    def avoid_collision(self):
        for angle_offset in [30, -30, 60, -60, 90, -90, 120, -120, 180]:
            rad_angle = np.deg2rad(self.angle + angle_offset)
            proposed_x = self.x + 0.3 * np.cos(rad_angle)
            proposed_y = self.y + 0.3 * np.sin(rad_angle)
            if not self.check_collision(proposed_x, proposed_y, self.robots):
                self.x, self.y = proposed_x, proposed_y
                self.path.append((self.x, self.y))
                self.steps = self.calculate_path_steps(self.target_position)
                self.step_index = 0
                return
        self.stuck_steps += 1

    def measure_distance(self):
        lidar_range = self.lidar_range
        lidar_angle = self.angle
        min_distance = lidar_range  # Устанавливаем максимальное расстояние дальномера как минимальное
найденное расстояние до препятствия
        for angle_step in range(-5, 6):  # Проверяем углы от -5 до 5 градусов относительно текущего
направления робота для учета небольших расхождений
            check_angle = np.deg2rad(self.angle + angle_step)
            for step in np.linspace(0, lidar_range, num=int(lidar_range/0.1)):
                check_x = self.x + np.cos(check_angle) * step
                check_y = self.y + np.sin(check_angle) * step
                for obstacle in self.obstacles:
                    if obstacle.distance((check_x, check_y)) < self.width / 2:
                        min_distance = min(min_distance, step)  # Обновляем минимальное расстояние, если найдено
более близкое препятствие
                        break
                else:
                    continue
                break

        lidar_end_x = self.x + np.cos(np.deg2rad(lidar_angle)) * min_distance
        lidar_end_y = self.y + np.sin(np.deg2rad(lidar_angle)) * min_distance

        return lidar_end_x, lidar_end_y

    def calculate_path_steps(self, target_position):
        steps = []
        current_x, current_y = self.x, self.y
        current_angle = np.deg2rad(self.angle)
        target_x, target_y = target_position
```

```
        while True:
            dx = target_x - current_x
            dy = target_y - current_y
            distance = np.sqrt(dx**2 + dy**2)
            if distance < 0.01:
                break
            target_angle = np.arctan2(dy, dx)
            angle_diff = np.rad2deg(np.mod(target_angle - current_angle + np.pi, 2 * np.pi) - np.pi)
            if abs(angle_diff) > 1:
                steps.append((0, angle_diff))
                distance = 0
            else:
                if distance <= self.max_speed_per_step:
                    steps.append((distance, 0))
                    break
                else:
                    steps.append((self.max_speed_per_step, 0))
            current_angle += np.deg2rad(angle_diff)
            current_x += np.cos(current_angle) * min(self.max_speed_per_step, distance)
            current_y += np.sin(current_angle) * min(self.max_speed_per_step, distance)

        return steps

    def avoid_obstacles_and_robots(self):
        for other_robot in self.robots:
            if other_robot != self:
                # Проверяем расстояние между текущим и другим роботом
                dx = other_robot.x - self.x
                dy = other_robot.y - self.y
                distance = np.sqrt(dx ** 2 + dy ** 2)
                if distance < 0.3:  # Учитываем положение другого робота только если он на расстоянии меньше
0.3 метра
                    self.avoid_robot_collision(other_robot)
                    self.avoid_collision()
                    # Обновляем угол движения после избегания столкновений с другими роботами
                    self.angle = np.degrees(np.arctan2(self.steps[self.step_index][1], 1))

    def avoid_robot_collision(self, other_robot):
        min_distance = 0.3  # Минимальное расстояние между роботами
        dx = other_robot.x - self.x
        dy = other_robot.y - self.y
        distance = np.sqrt(dx ** 2 + dy ** 2)
        if distance < min_distance:
            # Вычисляем новый угол, чтобы роботы не сталкивались
            target_angle = np.arctan2(dy, dx)
            angle_diff = np.rad2deg(np.mod(target_angle - np.deg2rad(self.angle) + np.pi, 2 * np.pi) - np.pi)
            self.angle = (self.angle + angle_diff) % 360
            # Корректируем позицию робота, чтобы он не наезжал на другого робота
            proposed_x = self.x - min_distance * np.cos(target_angle)
            proposed_y = self.y - min_distance * np.sin(target_angle)
            if not self.check_collision(proposed_x, proposed_y, self.robots):
                self.x, self.y = proposed_x, proposed_y

class AnimatedSimulation:
    def __init__(self, init_positions, target_positions, obstacle_params):
        self.robots = []  # Создаем пустой список для роботов
        for (x, y), target in zip(init_positions, target_positions):
            robot = Robot(x, y, np.random.rand() * 360, target, self.create_obstacles(obstacle_params))
            robot.robots = self.robots  # Передаем список роботов каждому роботу
            self.robots.append(robot)  # Добавляем созданный робот в список self.robots
        self.fig, self.ax = plt.subplots()
        self.width, self.height = 10, 10
        self.ax.set_aspect('equal')  # Устанавливаем одинаковый масштаб осей
```

```
        self.ax.set_xlim(0, self.width)
        self.ax.set_ylim(0, self.height)
        self.start_time = time.time()
        self.all_robots_stopped = False
        self.steps_without_movement = 0

    def create_obstacles(self, obstacle_params):
        obstacles = []
        for obstacle in obstacle_params:
            obstacles.append(Rectangle((obstacle['x'], obstacle['y']), obstacle['width'], obstacle['height']))
        return obstacles

    def animate(self):
        def update(frame):
            global pos, sec
            nonlocal active_robots
            active_robots = 0
            for robot in self.robots:
                if not robot.stop:
                    robot.move()
                    if robot.step_index < len(robot.steps):
                        active_robots += 1
            self.ax.clear()
            self.ax.set_aspect('equal')  # Сохраняем аспект при обновлении
            self.ax.set_xlim(0, self.width)
            self.ax.set_ylim(0, self.height)

            for obstacle in robot.obstacles:
                self.ax.add_patch(obstacle)
            i = 0
            Vx = []
            Vy = []
            W = []
            for robot in self.robots:
                path_x, path_y = zip(*robot.path)
                self.ax.plot(*robot.target_position, 'rs')
                self.ax.plot(path_x, path_y, 'k-', linewidth=0.5)
                self.ax.plot(robot.x, robot.y, 'bo')
                Vx.append((robot.x - robot.x0)/1)
                robot.x0 = robot.x
                Vy.append((robot.y - robot.y0)/1)
                robot.y0 = robot.y
                W.append((robot.angle - robot.ang0)/1)
                robot.ang0 = robot.angle
                lidar_end_x, lidar_end_y = robot.measure_distance()
                self.ax.plot([robot.x, lidar_end_x], [robot.y, lidar_end_y], 'g-')
                # Проверка столкновений с другими роботами
                robot.avoid_obstacles_and_robots()
                i += 1
            pos.append([sum(Vx)/i, sum(Vy)/i, sum(W)/i])
            if active_robots == 0 and not self.all_robots_stopped:
                end_time = time.time()
                sec = end_time - self.start_time
                print(f"Все роботы остановились. Время выполнения: {end_time - self.start_time:.2f} секунд.")
                self.all_robots_stopped = True

        active_robots = len(self.robots)
        anim = FuncAnimation(self.fig, update, frames=np.arange(300), repeat=False)
        plt.show()

init_positions = [(0, 1), (2, 0), (1, 1), (0, 0), (0, 1.5), (2.5, 0), (1.5, 1.5), (0.5, 0.5),
                  (0.4, 1.4), (2.4, 0.4), (1.4, 1.4), (0.4, 0.4), (0.4, 1.9), (2.9, 0.4), (1.9, 1.9), (0.9, 0.9)]
target_positions = [(8, 7), (7, 7), (8, 8), (8.5, 8), (8.5, 7), (7.5, 7), (8.5, 8.5), (7.4, 9),
```

```
                (8.4, 7.4), (6.4, 7.4), (6.4, 8.4), (8.9, 8.4), (8.9, 7.4), (7.9, 7.4), (8.9, 8.9), (9.4, 8.4)]
obstacles = [{'x': 4, 'y': 4, 'width': 1, 'height': 2}, {'x': 5, 'y': 1, 'width': 2, 'height': 1}, {'x': 6, 'y': 5, 'width': 1.5, 'height':
1.5}]
simulation = AnimatedSimulation(init_positions, target_positions, obstacles)
simulation.animate()
time_ = []
Vx = []
Vy = []
W = []
old_min, old_max = 0, len(pos)
new_min, new_max = 0, sec

old_range = old_max - old_min
new_range = new_max - new_min


for i in range(len(pos)):
    converted = ((i - old_min) * new_range / old_range) + new_min
    time_.append(converted)
    Vx.append(pos[i][0])
    Vy.append(pos[i][1])
    W.append(pos[i][2])

plt.suptitle("COM")
plt.subplot(131)
plt.title("Vx")
plt.xlabel("time, s")
plt.ylabel("Vx, m/s")
plt.plot(time_, Vx)

plt.subplot(132)
plt.title("Vy")
plt.xlabel("time, s")
plt.ylabel("Vy, m/s")
plt.plot(time_, Vy)

plt.subplot(133)
plt.title("W")
plt.xlabel("time, s")
plt.ylabel("W, rad/s")
plt.plot(time_, W)

plt.show()
```

Листинг программы моделирования PSO алгоритма

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib.patches as patches
import time

pos = []
sec = 0

class PSORobot:
    def __init__(self, x, y, target, obstacles):
        self.position = np.array([x, y], dtype=float)
        self.x0 = x
        self.y0 = y
        self.velocity = np.random.rand(2) * 0.1  # Инициализируем небольшую случайную начальную скорость
```

```
        self.angle = np.arctan2(self.velocity[1], self.velocity[0])
        self.ang0 = self.angle
        self.target = np.array(target, dtype=float)
        self.path = [tuple(self.position)]
        self.max_speed = 0.5
        self.max_angular_speed = np.pi / 2
        self.obstacles = obstacles
        self.lidar_range = 5
        self.all_robots = None
        self.at_target = False

    def update_all_robots(self, robots):
        self.all_robots = robots

    def update_velocity(self, global_best_position, delta_time=1):
        if not self.at_target:
            inertia = 0.5
            cognitive = 2.5
            social = 1
            r1, r2 = np.random.rand(), np.random.rand()
            cognitive_velocity = cognitive * r1 * (self.target - self.position)
            social_velocity = social * r2 * (global_best_position - self.position)
            velocity = inertia * self.velocity + cognitive_velocity + social_velocity
            speed = np.linalg.norm(velocity)
            if speed > self.max_speed:
                velocity = velocity * (self.max_speed / speed)
            self.velocity = velocity
            self.update_angle(delta_time)

    def update_angle(self, delta_time):
        if np.linalg.norm(self.velocity) > 0:  # Убеждаемся, что скорость не нулевая
            target_angle = np.arctan2(self.velocity[1], self.velocity[0])
            angle_diff = (target_angle - self.angle + np.pi) % (2 * np.pi) - np.pi
            max_angle_change = self.max_angular_speed * delta_time
            angle_diff = np.clip(angle_diff, -max_angle_change, max_angle_change)
            self.angle += angle_diff

    def move(self, delta_time=1):
        self.check_if_at_target()
        if not self.at_target:
            if self.avoid_obstacles_and_robots() or True:  # Убедитесь, что условие для движения правильное
                self.position += self.velocity * delta_time
                self.path.append(tuple(self.position))

    def check_if_at_target(self):
        target_radius = 0.3
        if np.linalg.norm(self.position - self.target) < target_radius:
            self.velocity = np.zeros(2)
            self.at_target = True

    def avoid_obstacles_and_robots(self):
        if self.at_target:
            return False
        buffer = 0.15  # Буферное расстояние для избежания столкновений
        num_checks = 1  # Количество проверочных точек вдоль предполагаемой траектории
        best_direction = None
        best_score = float('-inf')
        safe_distance = self.max_speed / num_checks
        # Итерация по разным направлениям в поисках безопасного пути
        angles = np.linspace(-np.pi/2, np.pi/2, 18)
        for angle in angles:
            test_direction = np.array([np.cos(self.angle + angle), np.sin(self.angle + angle)])
            collision = False
```

```
            for step in range(1, num_checks + 1):
                test_position = self.position + test_direction * safe_distance * step
                if self.check_collision(test_position, buffer):
                    collision = True
                    break
            if not collision:
                # Оценка направления по его близости к целевому
                score = np.dot(test_direction, self.target - self.position)
                if score > best_score:
                    best_score = score
                    best_direction = test_direction
        # Применение найденного лучшего направления
        if best_direction is not None:
            self.velocity = best_direction * self.max_speed
            return True
        return False

    def check_collision(self, position, buffer):
        """ Проверяет, пересекается ли данная позиция с каким-либо препятствием, учитывая буфер. """
        for obs in self.obstacles:
            expanded_rect = patches.Rectangle((obs['x'] - buffer, obs['y'] - buffer), obs['width'] + 2*buffer, obs['height']
+ 2*buffer)
            if expanded_rect.contains_point(position):
                return True
        return False

    def find_alternative_path(self):
        target_direction = (self.target - self.position) / np.linalg.norm(self.target - self.position)
        min_angle = float('inf')
        best_direction = self.velocity
        for angle in np.linspace(-np.pi, np.pi, 36, endpoint=True):
            direction = np.array([np.cos(angle), np.sin(angle)])
            angle_diff = np.arccos(np.clip(np.dot(direction, target_direction), -1, 1))
            if angle_diff < min_angle:
                test_position = self.position + direction * self.max_speed
                collision = False
                for obs in self.obstacles:
                    rect = patches.Rectangle((obs['x'], obs['y']), obs['width'], obs['height'])
                    if rect.contains_point(test_position):
                        collision = True
                        break
                if not collision:
                    min_angle = angle_diff
                    best_direction = direction
        self.velocity = best_direction * self.max_speed

    def measure_distance(self):
        lidar_direction = np.array([np.cos(self.angle), np.sin(self.angle)])
        for test_point in np.linspace(0, self.lidar_range, 100):
            test_pos = self.position + lidar_direction * test_point
            for obs in self.obstacles:
                rect = patches.Rectangle((obs['x'], obs['y']), obs['width'], obs['height'])
                if rect.contains_point(test_pos):
                    return test_point
        return self.lidar_range

    def avoid_obstacle(self):
        # Попытка изменить направление на +/- 30, 60, 90 градусов от текущего угла
        for angle_offset in [np.pi / 6, -np.pi / 6, np.pi / 3, -np.pi / 3, np.pi / 2, -np.pi / 2]:
            new_angle = self.angle + angle_offset
            test_direction = np.array([np.cos(new_angle), np.sin(new_angle)])
            test_position = self.position + test_direction * self.max_speed
            if not any(patches.Rectangle((obs['x'], obs['y']), obs['width'], obs['height']).contains_point(test_position) for
```

```
obs in self.obstacles):
                self.angle = new_angle
                self.velocity = test_direction * self.max_speed
                break

    def lidar_visual(self):
        lidar_direction = np.array([np.cos(self.angle), np.sin(self.angle)])
        end_point = self.position + lidar_direction * self.measure_distance()
        return self.position, end_point

class AnimatedPSOSimulation:
    def __init__(self, init_positions, target_positions, obstacles):
        self.robots = [
            PSORobot(x, y, target, obstacles)
            for (x, y), target in zip(init_positions, target_positions)]
        for robot in self.robots:
            robot.update_all_robots(self.robots)
        self.global_best_position = self.find_global_best()
        self.width, self.height = 10, 10
        self.fig, self.ax = plt.subplots()
        self.ax.set_aspect('equal')  # Сохраняем аспект при обновлении
        self.ax.set_xlim(0, self.width)
        self.ax.set_ylim(0, self.height)
        self.obstacles = obstacles
        self.start_time = time.time()  # Засекаем время старта
        self.all_robots_stopped = False

    def find_global_best(self):
        best_value = float('inf')
        best_position = None
        for robot in self.robots:
            value = np.linalg.norm(robot.position - robot.target)
            if value < best_value:
                best_value = value
                best_position = robot.position
        return best_position

    def animate(self):
        def update(frame):
            global pos, sec
            self.ax.clear()
            self.ax.set_aspect('equal')  # Сохраняем аспект при обновлении
            self.ax.set_xlim(0, self.width)
            self.ax.set_ylim(0, self.height)

            # Draw obstacles
            for obs in self.obstacles:
                rect = patches.Rectangle((obs['x'], obs['y']), obs['width'], obs['height'], linewidth=1, edgecolor='r',
facecolor='none')
                self.ax.add_patch(rect)
            # Update global best position
            current_global_best = self.find_global_best()
            if current_global_best is not None:
                self.global_best_position = current_global_best
            i = 0
            Vx = []
            Vy = []
            W = []
            all_at_target = True
            for robot in self.robots:
                robot.update_velocity(self.global_best_position)
                robot.move()
                path_x, path_y = zip(*robot.path)
```

```
                self.ax.plot(path_x, path_y, 'k-')
                self.ax.plot(robot.position[0], robot.position[1], 'bo')
                self.ax.plot(robot.target[0], robot.target[1], 'rx')
                Vx.append((robot.position[0] - robot.x0)/1)
                robot.x0 = robot.position[0]
                Vy.append((robot.position[1] - robot.y0)/1)
                robot.y0 = robot.position[1]
                W.append((robot.angle - robot.ang0)/1)
                robot.ang0 = robot.angle
                # Lidar visualization
                start, end = robot.lidar_visual()
                self.ax.plot([start[0], end[0]], [start[1], end[1]], 'g-')
                if not robot.at_target:
                    all_at_target = False
                i += 1
            pos.append([sum(Vx)/i, sum(Vy)/i, sum(W)/i])
            if all_at_target and not self.all_robots_stopped:
                end_time = time.time()
                sec = end_time - self.start_time
                print(f"Все роботы остановились. Время выполнения: {end_time - self.start_time:.2f} секунд.")
                self.all_robots_stopped = True
                # plt.close(self.fig)  # Закрываем фигуру, чтобы остановить анимацию
        anim = FuncAnimation(self.fig, update, frames=np.arange(100), repeat=False)
        plt.show()

init_positions = [(0, 1), (2, 0), (1, 1), (0, 0), (0, 1.5), (2.5, 0), (1.5, 1.5), (0.5, 0.5),
                  (0.4, 1.4), (2.4, 0.4), (1.4, 1.4), (0.4, 0.4), (0.4, 1.9), (2.9, 0.4), (1.9, 1.9), (0.9, 0.9)]
target_positions = [(8, 7), (7, 7), (8, 8), (8.5, 8), (8.5, 7), (7.5, 7), (8.5, 8.5), (7.4, 9),
                    (8.4, 7.4), (6.4, 7.4), (6.4, 8.4), (8.9, 8.4), (8.9, 7.4), (7.9, 7.4), (8.9, 8.9), (9.4, 8.4)]
obstacles = [{'x': 4, 'y': 4, 'width': 1, 'height': 2}, {'x': 5, 'y': 1, 'width': 2, 'height': 1}, {'x': 6, 'y': 5, 'width': 1.5, 'height':
1.5}]
simulation = AnimatedPSOSimulation(init_positions, target_positions, obstacles)
simulation.animate()
time_ = []
Vx = []
Vy = []
W = []
old_min, old_max = 0, len(pos)
new_min, new_max = 0, sec
old_range = old_max - old_min
new_range = new_max - new_min

for i in range(len(pos)):
    converted = ((i - old_min) * new_range / old_range) + new_min
    time_.append(converted)
    Vx.append(pos[i][0])
    Vy.append(pos[i][1])
    W.append(pos[i][2])

plt.suptitle("PSO")
plt.subplot(131)
plt.title("Vx")
plt.xlabel("time, s")
plt.ylabel("Vx, m/s")
plt.plot(time_, Vx)

plt.subplot(132)
plt.title("Vy")
plt.xlabel("time, s")
plt.ylabel("Vy, m/s")
plt.plot(time_, Vy)

plt.subplot(133)
```

```
plt.title("W")
plt.xlabel("time, s")
plt.ylabel("W, rad/s")
plt.plot(time_, W)

plt.show()
```

Листинг программы моделирования ACO алгоритма

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib.patches as patches
import time

pos = []
sec = 0

class Rectangle(patches.Rectangle):
    # Класс для описания препятствий
    def intersects(self, rect):
        return not (self.get_x() + self.get_width() < rect.get_x() or
                    self.get_x() > rect.get_x() + rect.get_width() or
                    self.get_y() + self.get_height() < rect.get_y() or
                    self.get_y() > rect.get_y() + rect.get_height())

class PheromoneMap:
    def __init__(self, width, height, decay_rate=0.1):
        self.map = np.zeros((height, width))
        self.decay_rate = decay_rate
    def update(self):
        self.map *= (1 - self.decay_rate)

    def add_pheromone(self, x, y, amount=1.0):
        if 0 <= int(x) < self.map.shape[1] and 0 <= int(y) < self.map.shape[0]:
            self.map[int(y), int(x)] += amount

    def get_pheromone_level(self, x, y):
        if 0 <= int(x) < self.map.shape[1] and 0 <= int(y) < self.map.shape[0]:
            return self.map[int(y), int(x)]
        return 0

class Robot:
    def __init__(self, x, y, target, obstacles, pheromone_map, robots=None):
        self.x = x
        self.y = y
        self.x0 = x
        self.y0 = y
        self.ang0 = 0
        self.target_position = target
        self.obstacles = obstacles
        self.pheromone_map = pheromone_map
        self.robots = robots if robots is not None else []
        self.path = [(x, y)]
        self.stop = False
        self.max_speed_per_step = 0.5
        self.width = 0.33
        self.angle = 0
        self.prev_x = x  # Предыдущее положение по x
        self.prev_y = y  # Предыдущее положение по y
```

```
def move(self):
    if self.stop:
        return
    if np.hypot(self.x - self.target_position[0], self.y - self.target_position[1]) < self.max_speed_per_step / 2:
        self.x, self.y = self.target_position  # Перемещаем робота точно в целевую позицию
        self.stop = True
        return
    self.prev_x, self.prev_y = self.x, self.y  # Обновляем предыдущее положение
    self.choose_next_step()

def choose_next_step(self):
    best_score = float('inf')
    best_move = (0, 0)
    # Увеличение детализации направлений движения
    step_angles = np.linspace(0, 360, 36, endpoint=False)  # 36 направлений
    for angle in step_angles:
        rad = np.radians(angle)
        dx = self.max_speed_per_step * np.cos(rad)
        dy = self.max_speed_per_step * np.sin(rad)
        nx, ny = self.x + dx, self.y + dy
        if 0 <= nx < 10 and 0 <= ny < 10 and not self.check_collision(nx, ny):
            distance_to_target = np.hypot(self.target_position[0] - nx, self.target_position[1] - ny)
            if distance_to_target < best_score:
                best_score = distance_to_target
                best_move = (dx, dy)
    if best_move != (0, 0):
        self.x += best_move[0]
        self.y += best_move[1]
        self.path.append((self.x, self.y))
        self.pheromone_map.add_pheromone(self.x, self.y, 1.0)
    # Рассчитываем угол направления робота на основе изменения положения
    dx = self.x - self.prev_x
    dy = self.y - self.prev_y
    self.angle = np.degrees(np.arctan2(dy, dx))

def try_random_step(self):
    attempts = 0
    while attempts < 10:
        angle = np.radians(np.random.randint(0, 360))
        dx = self.max_speed_per_step * np.cos(angle)
        dy = self.max_speed_per_step * np.sin(angle)
        nx, ny = self.x + dx, self.y + dy
        if 0 <= nx < 10 and 0 <= ny < 10 and not self.check_collision(nx, ny):
            self.x = nx
            self.y = ny
            self.path.append((self.x, self.y))
            self.pheromone_map.add_pheromone(self.x, self.y, 1.0)
            break
        attempts += 1

def check_collision(self, x, y):
    robot_rect = patches.Rectangle((x - self.width / 2, y - self.width / 2), self.width, self.width)
    if any(ob.intersects(robot_rect) for ob in self.obstacles):
        return True
    return any(np.hypot(x - other.x, y - other.y) <= self.width for other in self.robots if other != self)

def measure_distance(self):
    lidar_range = 5  # максимальный диапазон дальномера в метрах
    lidar_angle = np.radians(self.angle)
    min_distance = lidar_range  # начальное минимальное расстояние устанавливаем максимальным
    # Проверяем расстояние до препятствий в направлении дальномера
    for step in np.linspace(0, lidar_range, num=int(lidar_range/0.1)):
        check_x = self.x + np.cos(lidar_angle) * step
```

```
            check_y = self.y + np.sin(lidar_angle) * step
            for obstacle in self.obstacles:
                # Проверяем, пересекается ли точка с препятствием
                if obstacle.intersects(patches.Rectangle((check_x - self.width / 2, check_y - self.width / 2), self.width,
self.width)):
                    min_distance = min(min_distance, step)  # Обновляем минимальное расстояние, если нашли
ближайшее препятствие
                    break
            else:
                continue
            break
        lidar_end_x = self.x + np.cos(lidar_angle) * min_distance
        lidar_end_y = self.y + np.sin(lidar_angle) * min_distance

        return lidar_end_x, lidar_end_y

class AnimatedSimulation:
    def __init__(self, init_positions, target_positions, obstacle_params):
        self.start_time = time.time()  # Начальное время для замера времени выполнения
        self.all_robots_stopped = False
        self.width, self.height = 10, 10
        self.pheromone_map = PheromoneMap(self.width, self.height)
        self.obstacles = [Rectangle((obstacle['x'], obstacle['y']), obstacle['width'], obstacle['height']) for obstacle in
obstacle_params]
        self.robots = [Robot(x, y, target, self.obstacles, self.pheromone_map) for (x, y), target in zip(init_positions,
target_positions)]
        for robot in self.robots:
            robot.robots = self.robots
        self.fig, self.ax = plt.subplots()
        self.ax.set_aspect('equal')  # Устанавливаем одинаковый масштаб осей
        self.ax.set_xlim(0, self.width)
        self.ax.set_ylim(0, self.height)

    def animate(self):
        def update(frame):
            global pos, sec
            self.ax.clear()
            self.ax.set_aspect('equal')  # Сохраняем аспект при обновлении
            self.ax.set_xlim(0, self.width)
            self.ax.set_ylim(0, self.height)
            for obstacle in self.obstacles:
                self.ax.add_patch(patches.Rectangle((obstacle.get_x(), obstacle.get_y()), obstacle.get_width(),
obstacle.get_height(), fill=True, color='red'))
            active_robots = 0
            i = 0
            Vx = []
            Vy = []
            W = []
            for robot in self.robots:
                robot.move()
                path_x, path_y = zip(*robot.path)
                self.ax.plot(*robot.target_position, 'ro')
                self.ax.plot(path_x, path_y, 'k-')
                self.ax.plot(robot.x, robot.y, 'bo')
                Vx.append((robot.x - robot.x0)/1)
                robot.x0 = robot.x
                Vy.append((robot.y - robot.y0)/1)
                robot.y0 = robot.y
                W.append((robot.angle - robot.ang0)/1)
                robot.ang0 = robot.angle
                lidar_end_x, lidar_end_y = robot.measure_distance()
                self.ax.plot([robot.x, lidar_end_x], [robot.y, lidar_end_y], 'g-')  # Рисуем луч дальномера
                if not robot.stop:
```

```
                active_robots += 1
            i += 1
        pos.append([sum(Vx)/i, sum(Vy)/i, sum(W)/i])
        self.pheromone_map.update()
        if active_robots == 0 and not self.all_robots_stopped:
            self.all_robots_stopped = True
            end_time = time.time()
            total_time = end_time - self.start_time
            sec = total_time
            print(f"Все роботы остановились. Время выполнения: {total_time:.2f} секунд.")
    anim = FuncAnimation(self.fig, update, frames=np.arange(300), repeat=False)
    plt.show()

init_positions = [(0, 1), (2, 0), (1, 1), (0, 0), (0, 1.5), (2.5, 0), (1.5, 1.5), (0.5, 0.5),
                  (0.4, 1.4), (2.4, 0.4), (1.4, 1.4), (0.4, 0.4), (0.4, 1.9), (2.9, 0.4), (1.9, 1.9), (0.9, 0.9)]
target_positions = [(8, 7), (7, 7), (8, 8), (8.5, 8), (8.5, 7), (7.5, 7), (8.5, 8.5), (7.4, 9),
                    (8.4, 7.4), (6.4, 7.4), (6.4, 8.4), (8.9, 8.4), (8.9, 7.4), (7.9, 7.4), (8.9, 8.9), (9.4, 8.4)]
obstacles = [{'x': 4, 'y': 4, 'width': 1, 'height': 2}, {'x': 5, 'y': 1, 'width': 2, 'height': 1}, {'x': 6, 'y': 5, 'width': 1.5, 'height':
1.5}]
simulation = AnimatedSimulation(init_positions, target_positions, obstacles)
simulation.animate()
time_ = []
Vx = []
Vy = []
W = []
old_min, old_max = 0, len(pos)
new_min, new_max = 0, sec
old_range = old_max - old_min
new_range = new_max - new_min

for i in range(len(pos)):
    converted = ((i - old_min) * new_range / old_range) + new_min
    time_.append(converted)
    Vx.append(pos[i][0])
    Vy.append(pos[i][1])
    W.append(pos[i][2])

plt.suptitle("ACO")
plt.subplot(131)
plt.title("Vx")
plt.xlabel("time, s")
plt.ylabel("Vx, m/s")
plt.plot(time_, Vx)

plt.subplot(132)
plt.title("Vy")
plt.xlabel("time, s")
plt.ylabel("Vy, m/s")
plt.plot(time_, Vy)

plt.subplot(133)
plt.title("W")
plt.xlabel("time, s")
plt.ylabel("W, rad/s")
plt.plot(time_, W)

plt.show()
```

ПРИЛОЖЕНИЕ Б

Листинг программы робота-агента для эксперимента по COM алгоритму

```
import RPi.GPIO as GPIO
from i2c_itg3205 import *
from time import sleep
import atexit
import math
import socket
import threading
import json
import time
from zeroconf import ServiceBrowser, ServiceInfo, Zeroconf


# Конфигурация для удаленного запуска
SERVER_IP = "192.168.1.100"  # IP адрес сервера
SERVER_PORT = 12345  # Порт сервера


def listen_for_start():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((SERVER_IP, SERVER_PORT))
        print("Connected to the server. Waiting for the start command...")
        start_command = s.recv(1024).decode('utf-8')
        init_positions = {"Robot1": (0, 1), "Robot2": (2, 0), "Robot3": (1, 1), "Robot4": (0, 0)}
        target_positions = {"Robot1": (8, 7), "Robot2": (7, 7), "Robot3": (8, 8), "Robot4": (8.5, 8)}

        robot_name = "Robot2"  # Изменить имя робота
        robot = RealRobot(robot_name, init_positions[robot_name], target_positions[robot_name])
        if start_command == "START":
            print("Received start command. Starting the robot...")
            start_time = time.time()
            try:
                robot.move_to_target()
            except KeyboardInterrupt:
                robot.stop()
            end_time = time.time()
            duration = end_time - start_time
            s.sendall(f"FINISHED {robot.name} {duration:.2f}".encode('utf-8'))
            print(f"Sent completion message to the server: {duration:.2f} seconds")


# ----------------------------------------Constants---------------------------------

ENA = 13
ENB = 12
IN1 = 26
IN2 = 21
IN3 = 16
IN4 = 19

# Set the type of GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
# Motor initialized to LOW
```

```
GPIO.setup(ENA, GPIO.OUT)
GPIO.setup(IN1, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN2, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(ENB, GPIO.OUT)
GPIO.setup(IN3, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN4, GPIO.OUT, initial=GPIO.LOW)
pwmA = GPIO.PWM(ENA, 100)
pwmB = GPIO.PWM(ENB, 100)
pwmA.start(0)
pwmB.start(0)


# ----------------------------------------Close function---------------------------------
def MotorStop():
    print('motor stop')
    pwmA.ChangeDutyCycle(0)
    pwmB.ChangeDutyCycle(0)
    GPIO.output(IN1, False)
    GPIO.output(IN2, False)
    GPIO.output(IN3, False)
    GPIO.output(IN4, False)


def MotorOn(speed):
    print('motor forward')
    pwmA.ChangeDutyCycle(speed)
    pwmB.ChangeDutyCycle(speed)
    GPIO.output(IN1, False)
    GPIO.output(IN2, True)
    GPIO.output(IN3, False)
    GPIO.output(IN4, True)


def on_esc():
    print("end")
    MotorStop()


atexit.register(on_esc)


# ----------------------------------------Base function----------------------------------

def turn_by_angle(angle_g):
    itg3205 = i2c_itg3205(1)
    angle = angle_g*math.pi/180
    dt = 0.2
    spe = 60
    pwmA.ChangeDutyCycle(spe)
    pwmB.ChangeDutyCycle(spe)
    x, y, z = itg3205.getDegPerSecAxes()
    start_error = z
    this_angle = 0
    while abs(angle - this_angle) > 0.3:
        if angle - this_angle > 0:
            GPIO.output(IN4, True)
            GPIO.output(IN3, False)
            GPIO.output(IN1, True)
            GPIO.output(IN2, False)
        else:
            GPIO.output(IN3, True)
            GPIO.output(IN4, False)
            GPIO.output(IN2, True)
```

```
            GPIO.output(IN1, False)
        sleep(dt)
        try:
            itgready, dataready = itg3205.getInterruptStatus()
            if dataready:
                x, y, z = itg3205.getDegPerSecAxes()
                this_angle += -(z - start_error) / 180 * math.pi * dt
        except OSError:
            pass
    MotorStop()


class RobotP2P:
    def __init__(self, robot_name, service_type="_robot._tcp.local.", port=12345):
        self.robot_name = robot_name
        self.service_type = service_type
        self.port = port
        self.zeroconf = Zeroconf()
        self.address = self._get_local_ip()
        self.info = ServiceInfo(
            service_type,
            f"{robot_name}.{service_type}",
            addresses=[socket.inet_aton(self.address)],
            port=port,
            properties={"name": robot_name.encode("utf-8")},
        )
        self.known_robots = {}

    def _get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        try:
            s.connect(('10.255.255.255', 1))
            IP = s.getsockname()[0]
        except Exception:
            IP = '127.0.0.1'
        finally:
            s.close()
        return IP

    def update_service(self, zeroconf, type, name, info):
        pass

    def start(self):
        self.zeroconf.register_service(self.info)
        self.browser = ServiceBrowser(self.zeroconf, self.service_type, listener=self)
        threading.Thread(target=self.listen_for_messages).start()
        threading.Thread(target=self.send_updates).start()

    def add_service(self, zeroconf, type, name):
        info = zeroconf.get_service_info(type, name)
        if info and info.properties[b'name'] != self.robot_name.encode("utf-8"):
            robot_name = info.properties[b'name'].decode("utf-8")
            self.known_robots[robot_name] = (socket.inet_ntoa(info.addresses[0]), info.port)
            print(f"Discovered robot: {robot_name} at {self.known_robots[robot_name]}")

    def remove_service(self, zeroconf, type, name):
        robot_name = name.split('.')[0]
        if robot_name in self.known_robots:
            del self.known_robots[robot_name]
            print(f"Robot {robot_name} left the network")

    def listen_for_messages(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
            s.bind(('', self.port))
            s.listen()
            while True:
                conn, addr = s.accept()
                with conn:
                    while True:
                        data = conn.recv(1024)
                        if not data:
                            break
                        message = data.decode('utf-8')
                        print(f"Message from {addr}: {message}")

    def send_message(self, ip, port, message):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                sock.connect((ip, port))
                sock.sendall(bytes(message, 'utf-8'))
        except ConnectionRefusedError:
            pass

    def send_updates(self):
        while True:
            for robot_name, (ip, port) in self.known_robots.items():
                message = json.dumps({"name": self.robot_name, "location": "x:100, y:200"})  # Customize this message
                self.send_message(ip, port, message)
            time.sleep(1)

    def stop(self):
        self.zeroconf.unregister_service(self.info)
        self.zeroconf.close()


# ------------------------- Robot Movement Control ----------------------------
class RealRobot:
    def __init__(self, name, init_pos, target_pos):
        self.name = name
        self.x, self.y = init_pos
        self.target_x, self.target_y = target_pos
        self.angle = 0  # Начальный угол движения
        self.speed = 50  # Скорость движения

        # Инициализация P2P связи
        self.p2p = RobotP2P(self.name)
        self.p2p.start()

    def move_forward(self, duration):
        MotorOn(self.speed)
        sleep(duration)
        MotorStop()

    def turn_to(self, target_angle):
        current_angle = self.angle
        angle_diff = (target_angle - current_angle) % 360
        if angle_diff > 180:
            angle_diff -= 360
        turn_by_angle(angle_diff)
        self.angle = target_angle

    def move_to_target(self):
        while not self.is_at_target():
            target_angle = self.calculate_target_angle()
            self.turn_to(target_angle)
            distance = self.calculate_distance_to_target()
```

```
            self.move_forward(distance / self.speed)
            self.update_position()
            self.send_position_update()

    def calculate_target_angle(self):
        dx = self.target_x - self.x
        dy = self.target_y - self.y
        return math.degrees(math.atan2(dy, dx)) % 360

    def calculate_distance_to_target(self):
        dx = self.target_x - self.x
        dy = self.target_y - self.y
        return math.sqrt(dx ** 2 + dy ** 2)

    def is_at_target(self):
        return self.calculate_distance_to_target() < 0.1

    def update_position(self):
        self.x = self.target_x
        self.y = self.target_y

    def send_position_update(self):
        message = json.dumps({"name": self.name, "position": (self.x, self.y)})
        for robot_name, (ip, port) in self.p2p.known_robots.items():
            self.p2p.send_message(ip, port, message)

    def stop(self):
        self.p2p.stop()


# Запуск прослушивания команды старта в отдельном потоке
threading.Thread(target=listen_for_start).start()
```

Листинг программы робота-агента для эксперимента по PSO алгоритму

```
import numpy as np
import time
import json
import threading
from zeroconf import ServiceBrowser, ServiceInfo, Zeroconf
import socket
from i2c_itg3205 import *
import RPi.GPIO as GPIO
from time import sleep

# Конфигурация для удаленного запуска
SERVER_IP = "192.168.1.100"  # IP адрес сервера
SERVER_PORT = 12345  # Порт сервера


def listen_for_start():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((SERVER_IP, SERVER_PORT))
        print("Connected to the server. Waiting for the start command...")
        start_command = s.recv(1024).decode('utf-8')
        init_positions = {"Robot1": (0, 1), "Robot2": (2, 0), "Robot3": (1, 1), "Robot4": (0, 0)}
        target_positions = {"Robot1": (8, 7), "Robot2": (7, 7), "Robot3": (8, 8), "Robot4": (8.5, 8)}

        robot_name = "Robot2"  # Изменить имя робота
        robot = PSORobot(robot_name, init_positions[robot_name], target_positions[robot_name])
```

```
        if start_command == "START":
            print("Received start command. Starting the robot...")
            start_time = time.time()
            try:
                robot.move_to_target()
            except KeyboardInterrupt:
                robot.stop()
            end_time = time.time()
            duration = end_time - start_time
            s.sendall(f"FINISHED {robot.name} {duration:.2f}".encode('utf-8'))
            print(f"Sent completion message to the server: {duration:.2f} seconds")


# ----------------------------------------Constants---------------------------------
ENA = 13
ENB = 12
IN1 = 26
IN2 = 21
IN3 = 16
IN4 = 19

# Set the type of GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
# Motor initialized to LOW
GPIO.setup(ENA, GPIO.OUT)
GPIO.setup(IN1, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN2, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(ENB, GPIO.OUT)
GPIO.setup(IN3, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN4, GPIO.OUT, initial=GPIO.LOW)
pwmA = GPIO.PWM(ENA, 100)
pwmB = GPIO.PWM(ENB, 100)
pwmA.start(0)
pwmB.start(0)


def MotorStop():
    pwmA.ChangeDutyCycle(0)
    pwmB.ChangeDutyCycle(0)
    GPIO.output(IN1, False)
    GPIO.output(IN2, False)
    GPIO.output(IN3, False)
    GPIO.output(IN4, False)


def MotorOn(speed):
    pwmA.ChangeDutyCycle(speed)
    pwmB.ChangeDutyCycle(speed)
    GPIO.output(IN1, False)
    GPIO.output(IN2, True)
    GPIO.output(IN3, False)
    GPIO.output(IN4, True)


def turn_by_angle(angle_g):
    itg3205 = i2c_itg3205(1)
    angle = angle_g * math.pi / 180
    dt = 0.2
    spe = 60
    pwmA.ChangeDutyCycle(spe)
    pwmB.ChangeDutyCycle(spe)
    x, y, z = itg3205.getDegPerSecAxes()
```

```
    start_error = z
    this_angle = 0
    while abs(angle - this_angle) > 0.3:
        if angle - this_angle > 0:
            GPIO.output(IN4, True)
            GPIO.output(IN3, False)
            GPIO.output(IN1, True)
            GPIO.output(IN2, False)
        else:
            GPIO.output(IN3, True)
            GPIO.output(IN4, False)
            GPIO.output(IN2, True)
            GPIO.output(IN1, False)
        sleep(dt)
        try:
            itgready, dataready = itg3205.getInterruptStatus()
            if dataready:
                x, y, z = itg3205.getDegPerSecAxes()
                this_angle += -(z - start_error) / 180 * math.pi * dt
        except OSError:
            pass
    MotorStop()


class RobotP2P:
    def __init__(self, robot_name, service_type="_robot._tcp.local.", port=12345):
        self.robot_name = robot_name
        self.service_type = service_type
        self.port = port
        self.zeroconf = Zeroconf()
        self.address = self._get_local_ip()
        self.info = ServiceInfo(
            service_type,
            f"{robot_name}.{service_type}",
            addresses=[socket.inet_aton(self.address)],
            port=port,
            properties={"name": robot_name.encode("utf-8")},
        )
        self.known_robots = {}

    def _get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        try:
            s.connect(('10.255.255.255', 1))
            IP = s.getsockname()[0]
        except Exception:
            IP = '127.0.0.1'
        finally:
            s.close()
        return IP

    def update_service(self, zeroconf, type, name, info):
        pass

    def start(self):
        self.zeroconf.register_service(self.info)
        self.browser = ServiceBrowser(self.zeroconf, self.service_type, listener=self)
        threading.Thread(target=self.listen_for_messages).start()
        threading.Thread(target=self.send_updates).start()

    def add_service(self, zeroconf, type, name):
        info = zeroconf.get_service_info(type, name)
        if info and info.properties[b'name'] != self.robot_name.encode("utf-8"):
```

```
            robot_name = info.properties[b'name'].decode("utf-8")
            self.known_robots[robot_name] = (socket.inet_ntoa(info.addresses[0]), info.port)
            print(f"Discovered robot: {robot_name} at {self.known_robots[robot_name]}")

    def remove_service(self, zeroconf, type, name):
        robot_name = name.split('.')[0]
        if robot_name in self.known_robots:
            del self.known_robots[robot_name]
            print(f"Robot {robot_name} left the network")

    def listen_for_messages(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.bind(('', self.port))
            s.listen()
            while True:
                conn, addr = s.accept()
                with conn:
                    while True:
                        data = conn.recv(1024)
                        if not data:
                            break
                        message = data.decode('utf-8')
                        print(f'Message from {addr}: {message}")

    def send_message(self, ip, port, message):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                sock.connect((ip, port))
                sock.sendall(bytes(message, 'utf-8'))
        except ConnectionRefusedError:
            pass

    def send_updates(self):
        while True:
            for robot_name, (ip, port) in self.known_robots.items():
                message = json.dumps(
                    {"name": self.robot_name, "position": (self.x, self.y)})  # Update with real position
                self.send_message(ip, port, message)
            time.sleep(1)

    def stop(self):
        self.zeroconf.unregister_service(self.info)
        self.zeroconf.close()


class PSORobot:
    def __init__(self, name, init_pos, target_pos):
        self.name = name
        self.position = np.array(init_pos, dtype=float)
        self.target = np.array(target_pos, dtype=float)
        self.velocity = np.random.rand(2) * 0.1
        self.angle = np.arctan2(self.velocity[1], self.velocity[0])
        self.max_speed = 0.5
        self.max_angular_speed = np.pi / 2
        self.path = [tuple(self.position)]
        self.at_target = False
        self.p2p = RobotP2P(self.name)
        self.p2p.start()

    def move_forward(self, duration):
        MotorOn(self.max_speed * 100)
        sleep(duration)
        MotorStop()
```

```
def turn_to(self, target_angle):
    current_angle = self.angle
    angle_diff = (target_angle - current_angle) % (2 * np.pi)
    if angle_diff > np.pi:
        angle_diff -= 2 * np.pi
    turn_by_angle(np.degrees(angle_diff))
    self.angle = target_angle

def update_velocity(self, global_best_position):
    if not self.at_target:
        inertia = 0.5
        cognitive = 2.5
        social = 1
        r1, r2 = np.random.rand(), np.random.rand()

        cognitive_velocity = cognitive * r1 * (self.target - self.position)
        social_velocity = social * r2 * (global_best_position - self.position)

        velocity = inertia * self.velocity + cognitive_velocity + social_velocity
        speed = np.linalg.norm(velocity)

        if speed > self.max_speed:
            velocity = velocity * (self.max_speed / speed)
        self.velocity = velocity
        self.update_angle()

def update_angle(self):
    if np.linalg.norm(self.velocity) > 0:
        target_angle = np.arctan2(self.velocity[1], self.velocity[0])
        angle_diff = (target_angle - self.angle + np.pi) % (2 * np.pi) - np.pi
        angle_diff = np.clip(angle_diff, -self.max_angular_speed, self.max_angular_speed)
        self.angle += angle_diff

def move(self):
    if not self.at_target:
        target_angle = np.arctan2(self.velocity[1], self.velocity[0])
        self.turn_to(target_angle)
        distance = np.linalg.norm(self.velocity)
        self.move_forward(distance / self.max_speed)
        self.position += self.velocity
        self.path.append(tuple(self.position))
        self.check_if_at_target()
        self.send_position_update()

def check_if_at_target(self):
    target_radius = 0.3
    if np.linalg.norm(self.position - self.target) < target_radius:
        self.velocity = np.zeros(2)
        self.at_target = True

def send_position_update(self):
    message = json.dumps({"name": self.name, "position": self.position.tolist()})
    for robot_name, (ip, port) in self.p2p.known_robots.items():
        self.p2p.send_message(ip, port, message)

def move_to_target(self):
    try:
        while not self.at_target:
            global_best_position = self.find_global_best()
            self.update_velocity(global_best_position)
            self.move()
            time.sleep(0.1)
```

```
        except KeyboardInterrupt:
            self.stop()

    def find_global_best(self):
        best_value = float('inf')
        best_position = None
        for robot_name, (ip, port) in self.p2p.known_robots.items():
            value = np.linalg.norm(self.position - self.target)
            if value < best_value:
                best_value = value
                best_position = self.position
        return best_position

    def stop(self):
        self.p2p.stop()
        MotorStop()


# Запуск прослушивания команды старта в отдельном потоке
threading.Thread(target=listen_for_start).start()
```

Листинг программы робота-агента для эксперимента по ACO алгоритму

```
import numpy as np
import time
import json
import threading
from zeroconf import ServiceBrowser, ServiceInfo, Zeroconf
import socket
from i2c_itg3205 import *
import RPi.GPIO as GPIO
from time import sleep

# Конфигурация для удаленного запуска
SERVER_IP = "192.168.1.100"  # IP адрес сервера
SERVER_PORT = 12345  # Порт сервера

def listen_for_start():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((SERVER_IP, SERVER_PORT))
        print("Connected to the server. Waiting for the start command...")
        start_command = s.recv(1024).decode('utf-8')
        init_positions = {"Robot1": (0, 1), "Robot2": (2, 0), "Robot3": (1, 1), "Robot4": (0, 0)}
        target_positions = {"Robot1": (8, 7), "Robot2": (7, 7), "Robot3": (8, 8), "Robot4": (8.5, 8)}
        robot_name = "Robot2"  # Изменить имя робота
        robot = ACORobot(robot_name, init_positions[robot_name], target_positions[robot_name])
        if start_command == "START":
            print("Received start command. Starting the robot...")
            start_time = time.time()
            try:
                robot.move_to_target()
            except KeyboardInterrupt:
                robot.stop()
            end_time = time.time()
            duration = end_time - start_time
            s.sendall(f"FINISHED {robot.name} {duration:.2f}".encode('utf-8'))
            print(f"Sent completion message to the server: {duration:.2f} seconds")

# ----------------------------------------Constants---------------------------------
ENA = 13
```

```
ENB = 12
IN1 = 26
IN2 = 21
IN3 = 16
IN4 = 19

# Set the type of GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
# Motor initialized to LOW
GPIO.setup(ENA, GPIO.OUT)
GPIO.setup(IN1, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN2, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(ENB, GPIO.OUT)
GPIO.setup(IN3, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(IN4, GPIO.OUT, initial=GPIO.LOW)
pwmA = GPIO.PWM(ENA, 100)
pwmB = GPIO.PWM(ENB, 100)
pwmA.start(0)
pwmB.start(0)

def MotorStop():
    pwmA.ChangeDutyCycle(0)
    pwmB.ChangeDutyCycle(0)
    GPIO.output(IN1, False)
    GPIO.output(IN2, False)
    GPIO.output(IN3, False)
    GPIO.output(IN4, False)

def MotorOn(speed):
    pwmA.ChangeDutyCycle(speed)
    pwmB.ChangeDutyCycle(speed)
    GPIO.output(IN1, False)
    GPIO.output(IN2, True)
    GPIO.output(IN3, False)
    GPIO.output(IN4, True)

def turn_by_angle(angle_g):
    itg3205 = i2c_itg3205(1)
    angle = angle_g * math.pi / 180
    dt = 0.2
    spe = 60
    pwmA.ChangeDutyCycle(spe)
    pwmB.ChangeDutyCycle(spe)
    x, y, z = itg3205.getDegPerSecAxes()
    start_error = z
    this_angle = 0
    while abs(angle - this_angle) > 0.3:
        if angle - this_angle > 0:
            GPIO.output(IN4, True)
            GPIO.output(IN3, False)
            GPIO.output(IN1, True)
            GPIO.output(IN2, False)
        else:
            GPIO.output(IN3, True)
            GPIO.output(IN4, False)
            GPIO.output(IN2, True)
            GPIO.output(IN1, False)
        sleep(dt)
        try:
            itgready, dataready = itg3205.getInterruptStatus()
            if dataready:
                x, y, z = itg3205.getDegPerSecAxes()
```

```
            this_angle += -(z - start_error) / 180 * math.pi * dt
        except OSError:
            pass
    MotorStop()

class RobotP2P:
    def __init__(self, robot_name, service_type="_robot._tcp.local.", port=12345):
        self.robot_name = robot_name
        self.service_type = service_type
        self.port = port
        self.zeroconf = Zeroconf()
        self.address = self._get_local_ip()
        self.info = ServiceInfo(
            service_type,
            f"{robot_name}.{service_type}",
            addresses=[socket.inet_aton(self.address)],
            port=port,
            properties={"name": robot_name.encode("utf-8")},
        )
        self.known_robots = {}

    def _get_local_ip(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        try:
            s.connect(('10.255.255.255', 1))
            IP = s.getsockname()[0]
        except Exception:
            IP = '127.0.0.1'
        finally:
            s.close()
        return IP

    def update_service(self, zeroconf, type, name, info):
        pass

    def start(self):
        self.zeroconf.register_service(self.info)
        self.browser = ServiceBrowser(self.zeroconf, self.service_type, listener=self)
        threading.Thread(target=self.listen_for_messages).start()
        threading.Thread(target=self.send_updates).start()

    def add_service(self, zeroconf, type, name):
        info = zeroconf.get_service_info(type, name)
        if info and info.properties[b'name'] != self.robot_name.encode("utf-8"):
            robot_name = info.properties[b'name'].decode("utf-8")
            self.known_robots[robot_name] = (socket.inet_ntoa(info.addresses[0]), info.port)
            print(f"Discovered robot: {robot_name} at {self.known_robots[robot_name]}")

    def remove_service(self, zeroconf, type, name):
        robot_name = name.split('.')[0]
        if robot_name in self.known_robots:
            del self.known_robots[robot_name]
            print(f"Robot {robot_name} left the network")

    def listen_for_messages(self):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.bind(('', self.port))
            s.listen()
            while True:
                conn, addr = s.accept()
                with conn:
                    while True:
                        data = conn.recv(1024)
```

```
                if not data:
                    break
                message = data.decode('utf-8')
                print(f"Message from {addr}: {message}")

    def send_message(self, ip, port, message):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                sock.connect((ip, port))
                sock.sendall(bytes(message, 'utf-8'))
        except ConnectionRefusedError:
            pass

    def send_updates(self):
        while True:
            for robot_name, (ip, port) in self.known_robots.items():
                message = json.dumps(
                    {"name": self.robot_name, "position": (self.x, self.y)})  # Update with real position
                self.send_message(ip, port, message)
            time.sleep(1)

    def stop(self):
        self.zeroconf.unregister_service(self.info)
        self.zeroconf.close()

class PheromoneMap:
    def __init__(self, width, height, decay_rate=0.1):
        self.map = np.zeros((height, width))
        self.decay_rate = decay_rate

    def update(self):
        self.map *= (1 - self.decay_rate)

    def add_pheromone(self, x, y, amount=1.0):
        if 0 <= int(x) < self.map.shape[1] and 0 <= int(y) < self.map.shape[0]:
            self.map[int(y), int(x)] += amount

    def get_pheromone_level(self, x, y):
        if 0 <= int(x) < self.map.shape[1] and 0 <= int(y) < self.map.shape[0]:
            return self.map[int(y), int(x)]
        return 0

class ACORobot:
    def __init__(self, name, init_pos, target_pos, pheromone_map):
        self.name = name
        self.position = np.array(init_pos, dtype=float)
        self.target = np.array(target_pos, dtype=float)
        self.pheromone_map = pheromone_map
        self.velocity = np.random.rand(2) * 0.1
        self.angle = np.arctan2(self.velocity[1], self.velocity[0])
        self.max_speed = 0.5
        self.width = 0.33
        self.path = [tuple(self.position)]
        self.at_target = False
        self.p2p = RobotP2P(self.name)
        self.p2p.start()

    def move_forward(self, duration):
        MotorOn(self.max_speed * 100)
        sleep(duration)
        MotorStop()

    def turn_to(self, target_angle):
```

```
        current_angle = self.angle
        angle_diff = (target_angle - current_angle) % (2 * np.pi)
        if angle_diff > np.pi:
            angle_diff -= 2 * np.pi
        turn_by_angle(np.degrees(angle_diff))
        self.angle = target_angle

    def update_velocity(self):
        if not self.at_target:
            pheromone_levels = [self.pheromone_map.get_pheromone_level(self.position[0] + dx, self.position[1] + dy)
                                for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]]
            best_direction = np.argmax(pheromone_levels)
            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
            self.velocity = np.array(directions[best_direction]) * self.max_speed
            self.update_angle()

    def update_angle(self):
        if np.linalg.norm(self.velocity) > 0:
            target_angle = np.arctan2(self.velocity[1], self.velocity[0])
            angle_diff = (target_angle - self.angle + np.pi) % (2 * np.pi) - np.pi
            angle_diff = np.clip(angle_diff, -np.pi / 2, np.pi / 2)
            self.angle += angle_diff

    def move(self):
        if not self.at_target:
            target_angle = np.arctan2(self.velocity[1], self.velocity[0])
            self.turn_to(target_angle)
            distance = np.linalg.norm(self.velocity)
            self.move_forward(distance / self.max_speed)
            self.position += self.velocity
            self.path.append(tuple(self.position))
            self.pheromone_map.add_pheromone(self.position[0], self.position[1], 1.0)
            self.check_if_at_target()
            self.send_position_update()

    def check_if_at_target(self):
        target_radius = 0.3
        if np.linalg.norm(self.position - self.target) < target_radius:
            self.velocity = np.zeros(2)
            self.at_target = True

    def send_position_update(self):
        message = json.dumps({"name": self.name, "position": self.position.tolist()})
        for robot_name, (ip, port) in self.p2p.known_robots.items():
            self.p2p.send_message(ip, port, message)

    def move_to_target(self):
        try:
            while not self.at_target:
                self.update_velocity()
                self.move()
                time.sleep(0.1)
        except KeyboardInterrupt:
            self.stop()

    def stop(self):
        self.p2p.stop()
        MotorStop()


# Запуск прослушивания команды старта в отдельном потоке
threading.Thread(target=listen_for_start).start()
```