МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Дальневосточный федеральный университет»

ИНЖЕНЕРНАЯ ШКОЛА

Кафедра Автоматизации и управления

Степаненко Владислав Юрьевич

РАЗРАБОТКА АЛГОРИТМА ПОСТРОЕНИЯ КАРТЫ ДЛЯ МОБИЛЬНОГО РОБОТА НА ОСНОВЕ ДАННЫХ, ПОСТУПАЮЩИХ ОТ ИНФРАКРАСНЫХ И УЛЬТРАЗВУКОВЫХ ДАТЧИКОВ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки бакалавров
15.03.06 – Мехатроника и робототехника
профиль «*Мехатроника и робототехника*»

г. Владивосток
2018

Студент _____
(подпись)
«22» июня 2018 г.

Руководитель квалификационной выпуск
работы (проекта) доцент, д.т.н.
(должность, ученое звание)
_____ Юхимец Д.
(подпись) (ФИО)
«22» июня 201

«Допустить к защите»

Руководитель ОП к.т.н.
(ученое звание)
_____ Кацурин А. А.
(подпись) (ФИО)
«22» июня 2018 г.

Зав. кафедрой д.т.н.
(ученое звание)
_____ Филаретов В. Ф.
(подпись) (ФИО)
«27» июня 2018г

Защищена в ГАК с оценкой хорошо

Секретарь ГАК
_____ Мурсалимов Т.М.
(подпись) (ФИО)
«4» июля 2018г.

В материалах данной выпускной квалификационной работы содержатся сведения, составляющие государственную тайну, и сведен подлежащие экспортному контролю.

Директор ИШ ДВФУ А.Т. Беккер

Сведений, содержащих государственную тайну, нет.
Эксперт

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Дальневосточный федеральный университет»

Инженерная школа
Кафедра автоматизации и управления

УТВЕРЖДЕНО
Руководитель ОП к.т.н., доцент.
(ученая степень, должность)
(подпись) Кацурин А.А. (ФИО)
« 5 » октября 2017 г.

Заведующий кафедрой д.т.н., профессор
(ученая степень, должность)
(подпись) Филаретов В.Ф. (ФИО)
« 5 » октября 2017 г.

З А Д А Н И Е
на выпускную квалификационную работу

Студенту **Степаненко Владислав Юрьевич** (Фамилия, Имя, Отчество) Группа **Б3421** (номер группы)

1. Наименование темы **Разработка алгоритма построения карты для мобильного робота на основе данных, поступающий от инфракрасных и ультразвуковых датчиков.**
2. Основания для разработки **Задание от руководителя выпускной работы.**
3. Источники разработки **Отсутствуют.**
4. Технические требования (параметры) **Разрабатываемый алгоритм должен обеспечить построение карты местности и определение положение робота с помощь ультразвуковых и инфракрасных датчиков.**
5. Дополнительные требования **Отсутствуют.**
6. Перечень разработанных вопросов **1. Анализ существующих подходов к реализации алгоритмов одновременного построения карты и локализации положения робота.**
2. Разработка алгоритма одновременного построения карты и локализации

положения робота, для мобильного робота с использованием ультразвуковых и инфракрасных датчиков.

3. Проверка работоспособности разработанного алгоритма в среде моделирования V-REP для мобильного двухколесного робота.

7.Перечень графического материала (с точным указанием обязательных плакатов)

Графический материал отсутствует.

КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ РАБОТЫ

№ п/п	Наименование этапов дипломного проекта (работы)	Срок выполнения этапов дипломного проекта (работы)	Примечание
1	Написание введения и постановки задачи	До 15.02.2018	
2	Написание первой главы ВКР (обзор существующих подходов)	До 15.03.2018	
2	Написание второй главы ВКР (основной раздел)	До 12.04.2018	
4	Написание третьей главы ВКР (описание результатов моделирования)	До 19.05.2018	
5	Доработка ВКР в соответствии с замечаниями руководителя	До 03.06.2018	
6	Проверка ВКР на антиплагиат	До 10.06.2018	
7	Подготовка в защите (подготовка доклада и презентация в Power Point)	До 17 .06.2018	
8	Защита ВКР в ГЭК	27.06.2018-06.07.2018	

Дата выдачи задания 5.10.2017г.

Срок представления к защите июнь 2018года

Руководитель ВКР _____ Юхимец Д.А.
(подпись) (ФИО)

Студент _____ Степаненко В.Ю.
(подпись) (ФИО)

АННОТАЦИЯ

Дипломная работа на тему: “Разработка алгоритма построения карты для мобильного робота на основе данных, поступающих от инфракрасных и ультразвуковых датчиков”.

Объем дипломной работы 66 страниц, на которых размещены 14 рисунков и 1 приложение. При написании диплома использовалось 29 источника.

Ключевые слова: SLAM, мобильный робот, фильтр частиц, фильтр Калмана.

Объектом исследования при написании работы послужила проблема построения карты неизвестной местности для мобильного робота.

Предметом исследования работы стало разработка алгоритма одновременной локализации и построения карты (SLAM) для мобильного робота.

В дипломную работу входит введение, три главы, три вывода по написанным главам, итоговое заключение.

Во введении раскрывается актуальность работы по выбранному направлению, ставится проблема, цель и задачи исследования, определяется

В первой главе предложено теоретическое обоснование проблемы построения карты местности для мобильного робота, а также рассматриваются самые современные подходы к её решению.

В выводе по ней подводятся итоги по изучению теоретического материала.

Во второй главе на основе рассмотренного теоретического материала, осуществляется разработка алгоритма для одновременной локализации и построения карты непосредственно для мобильного робота, с помощью ультразвуковых и инфракрасных датчиков.

В выводе по второй главе проводится анализ разработанного алгоритма, его эффективность и степень применимости в реальных условиях.

В третей главе осуществляется моделирование работы разработанного алгоритма, сравнение полученных результатов в зависимости от типа использованных датчиков.

В выводе по третей главе подводятся итоги по моделированию работы алгоритма SLAM, а также анализ его эффективности в зависимости от использованных датчиков.

Заключение посвящено основным выводам и предложениям по разработанному алгоритму одновременной локализации и построению карты местности, с помощью ультразвуковых или инфракрасных датчиков, а также о возможности дальнейшего его использования.

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

МР – мобильный робот

РТС - робототехническая система

РФК – расширенный фильтр Калмана

ПО – программное обеспечение

ФК – фильтр Калмана

ФЧ – фильтр частиц

GPS - Global Positioning System

IEEE - Institute of Electrical and Electronics Engineers

ROS – Robot Operating System

SLAM – Simultaneous Localization and Mapping

ВВЕДЕНИЕ

Важной задачей в современной робототехнике является разработка способов определения положения робота в окружающем пространстве. Не зная положения робота в пространстве, не зная как выглядит окружающее пространство невозможно решить даже простейшую задачу движения робота из одной точки в другую. Наиболее часто используемые способы определения положения – интегрирование перемещений робота (с помощью одометров) или применение маяков, установленных в определенных местах. Использование маяков не универсально и требует предварительного оборудования рабочих помещений, при этом маяки постоянно должны быть в зоне видимости роботом. Интегрирование показаний одометров не обеспечивает точности позиционирования из-за накопления ошибки по всем отслеживаемым координатам.

Актуальность выбранной темы обусловлена значимостью развития методов локализации робота и одновременного построения карты местности в современной робототехнике.

Целью данной работы – является разработка алгоритма SLAM на основе фильтра частиц для наземного мобильного робота, использующего простейшие ультразвуковые или инфракрасные датчики дистанции, и моделирование его работы в среде моделирования V-REP.

В соответствии с поставленной целью необходимо решить следующие задачи:

а) Изучить теоретические основы и ознакомиться с основными методами и подходами к решению задачи SLAM;

б) На основе проведенного анализа различных подходов разработать алгоритм, который позволяет определить положения робота в пространстве, и построить карту окружения робота.

в) Провести моделирование работы разработанного алгоритма и провести анализ полученных результатов.

Выпускная квалификационная работа состоит из введения, трёх глав, заключения, списка используемых источников и приложения.

В первой главе раскрываются основные подходы к реализации алгоритма одновременной локализации и построения карты, а также рассматриваются современные реализации алгоритмов, их достоинства и недостатки.

Во второй главе на основе рассмотренного теоретического материала, осуществляется разработка алгоритма для одновременной локализации и построения карты непосредственно для мобильного робота, с помощью ультразвуковых и инфракрасных датчиков.

В третей главе проводится моделирование роботы разработанного алгоритма при использовании ультразвуковых и инфракрасных датчиков в среде моделирования V-REP.

1 ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ

1.1 Актуальность и новизна

Мобильные автономные робототехнические системы в настоящее время, являются весьма обширной областью робототехники. Основным направлением исследований при разработке таких систем является разработка алгоритмов определения положения и ориентации робота в пространстве. Данную задачу можно разделить на две составляющих: построение карт (отображение) и локализация робота на местности (локализация).

Для решения первой задачи, необходимо знать максимально точное положение мобильного робота, чтобы основываясь на данных получаемых от датчиков производить построение карты. Однако тогда задача локализации становится приоритетной, и возникает вопрос, как определять положение робота?

Использование одних одометров, хотя и является относительно простым способом определения позиции робота, этот метод не отличается высокой точностью. Использование GPS не всегда представляется возможным. Тогда разумным решением будет использование карты для определения положения робота, но откуда мы её можем получить, если окружающее робота пространство неизвестно?

В идеальном случае имеется возможность загрузить рабочую карту окружающего пространства, однако на практике такая возможность имеется не всегда, поэтому встает естественная задача: научить робота строить карту местности и одновременно определять положение в этой местности и траекторию движения. Область знаний, описывающая методы решения данной задачи, получила название SLAM (одновременная локализация и сопоставление). На данный момент существует довольно большое количество

реализаций и подходов, опирающихся на различные программные и аппаратные возможности платформ [1].

1.2 Классификация алгоритмов SLAM

Алгоритмы SLAM можно разделить по методу, используемому для расчетов на три группы [2]:

а) На основе фильтра Калмана и его модификаций,

б) На основе фильтра частиц,

в) Методы, основанные на графах.

Каждый из этих подходов можно применять для решения задачи SLAM, но у каждого из них есть свои преимущества и недостатки, рассмотрим каждый из подходов по отдельности.

1.2.1 Фильтр Калмана

Фильтр Калмана – это, наверное, самый популярный алгоритм фильтрации, используемый во многих областях науки и техники. Благодаря своей простоте и эффективности его можно встретить в обработчиках показаний датчиков, GPS-приемниках, при реализации систем управления и т.д.

Впервые использование фильтра Калмана для решения задачи SLAM было предложено 2002 году Montemerlo в своей статье [3].

Фильтры Калмана [4] базируются на дискретных по времени линейных динамических системах. Так как движение автономных роботов плохо аппроксимируется линейным движением, то фильтры Калмана не получили широкого распространения в реализациях задачи SLAM. В то же время, возможно применение фильтров Калмана для вычисления положения

неподвижных пространственных ориентиров, так как они удовлетворяют свойству линейности.

При использовании фильтра Калмана для получения оценок вектора состояния процесса по серии зашумленных измерений необходимо представить модель данного процесса в соответствии со структурой фильтра – в виде матричного уравнения определенного типа. Для каждого такта k работы фильтра необходимо в соответствии с приведенным ниже описанием определить матрицы: эволюции процесса F_k; матрицу наблюдений H_k; ковариационную матрицу процесса Q_k; ковариационную матрицу шума измерений R_k; при наличии управляющих воздействий матрицу их коэффициентов B_k. Рассмотрим каждый из этапов подробнее:

Модель системы подразумевает, что состояние в момент k получается из состояния в момент (k − 1) в соответствии с уравнением:

$$\hat{x}_k^- = F\hat{x}_{k-1} + Bu_{k-1} + w_k,$$

где:

F_k – матрица эволюции процесса/системы, которая воздействует на вектор состояния в момент (k − 1);

B_k – матрица управления, которая прикладывается к вектору управляющих воздействий u_{k-1};

w_k – нормальный случайный процесс с нулевым математическим ожиданием и ковариационной матрицей Q_k, который описывает случайный характер эволюции системы:

$$w_k \sim N(0, Q_k).$$

В момент k производится наблюдение (измерение) z_k истинного вектора состояния x_k, которые связаны между собой уравнением:

$$z_k = H_k x_k + v_k,$$

где:

H_k – матрица измерений, связывающая вектор состояния и вектор произведенных измерений,

v_k – белый гауссовский шум измерений с нулевым математическим ожиданием и ковариационной матрицей R_k:

$$v_k \sim N(0, R_k).$$

Начальное состояние и векторы случайных процессов на каждом такте $\{x_0, w_1, \ldots, w_k, v_1 \ldots v_k\}$ считаются независимыми.

Многие реальные динамические системы нельзя точно описать данной моделью. На практике неучтенная в модели динамика может серьезно испортить рабочие характеристики фильтра, особенно при работе с неизвестным стохастическим сигналом на входе. Более того, неучтенная в модели динамика может сделать фильтр неустойчивым. С другой стороны, независимый белый шум в качестве сигнала не будет приводить к расхождению алгоритма [5].

Для вычисления оценки состояния системы на текущий такт работы фильтру Калмана необходимы оценка состояния (в виде оценки состояния системы и оценки погрешности определения этого состояния) на предыдущем такте работы и измерения на текущем такте.

Далее рассмотрим работу классического алгоритма SLAM, на рисунке 1 представлена общая схема работы фильтра Калмана для задачи SLAM.

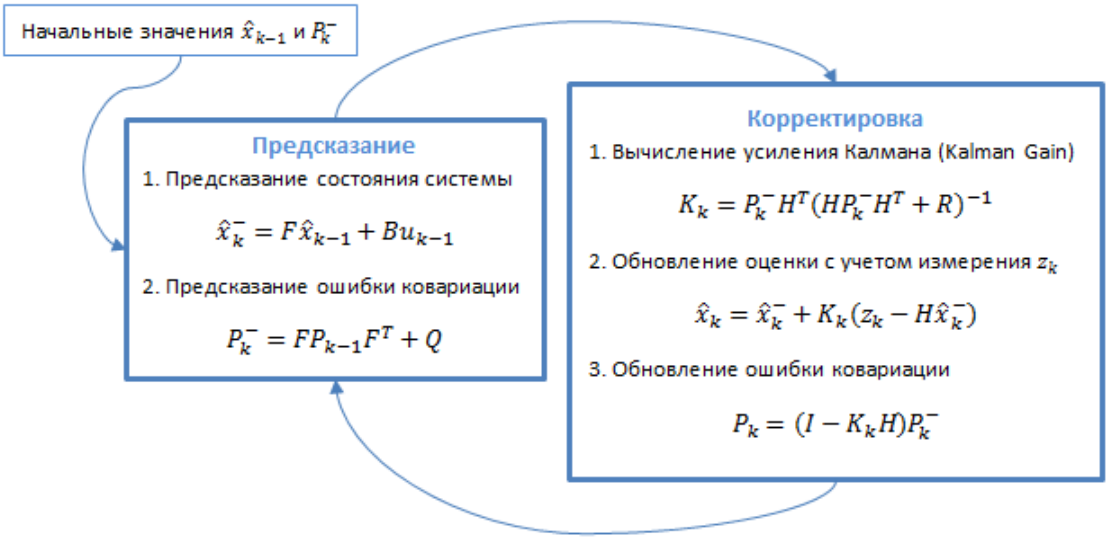

Рисунок 1 – Алгоритм работы фильтра Калмана

Как видно из рисунка 1, работу фильтра Калмана можно разделить на два больших этапа:

а) Этап экстраполяции – на данном этапе ФК “предсказывает” положение робота на основе начальных значений, если это первая итерация цикла или данных, получаемых с этапа корректировки, а также значений управляющий воздействий, для мобильного робота это данные, поступающие с одометров.

б) Этап корректировки – на данном этапе ФК используя данные, полученные с датчиков для корректировки положения

Рассмотрим каждый из этапов по отдельности.

Этап экстраполяции

Экстраполяция (предсказание) вектора состояние системы по оценке вектора состояния и примененному вектору управления с шага (k − 1) на шаг k:

$$\hat{\mathrm{x}}_{\mathrm{k}}^{-} = \mathrm{F}\hat{\mathrm{x}}_{\mathrm{k}-1} + \mathrm{Bu}_{\mathrm{k}-1}.$$

Ковариационная матрица для экстраполированного вектора состояния:

$$P_k^- = FP_{k-1}F^T + Q.$$

Этап коррекции

Отклонение полученного на шаге k наблюдения от наблюдения, ожидаемого при произведенной экстраполяции:

$$\tilde{y}_k = z_k - H\hat{x}_k^-.$$

Ковариационная матрица для вектора отклонения (вектора ошибки):

$$S_k = HP_k^-H^T + R.$$

Оптимальная по Калману матрица коэффициентов усиления, формирующаяся на основании ковариационных матриц, имеющейся экстраполяции вектора состояния и полученных измерений (посредством ковариационной матрицы вектора отклонения):

$$K_k = P_k^-H^TS_k^{-1}.$$

Коррекция ранее полученной экстраполяции вектора состояния – получение оценки вектора состояния системы:

$$\hat{x}_k = \hat{x}_k^- + K_k\tilde{y}_k.$$

Расчет ковариационной матрицы оценки вектора состояния системы:

$$P_k = (I - K_kH)P_k^-.$$

Фильтры Калмана [6, 7] базируются на дискретных по времени линейных динамических системах. Так как движение автономных роботов плохо аппроксимируется линейными функциями, то фильтры Калмана не

получили широкого распространения в реализациях задачи SLAM. В то же время, возможно применение фильтров Калмана для вычисления положения неподвижных пространственных ориентиров, так как они удовлетворяют свойству линейности.

Если стоит задача фильтрации на основе фильтра Калмана для нелинейных систем, то необходимо использовать расширенный фильтр Калмана (РФК).

1.2.2 Расширенный фильтр Калмана

Расширенный фильтр Калмана [8, 9] является нелинейной версией фильтра Калмана. Одно время РФК мог считаться стандартом в теории нелинейного вычисления положения, навигационных систем и GPS. Однако с появлением фильтра частиц и графа аналитических методов его популярность пошла на убыль.

В расширенном фильтре Калмана модели системы и наблюдения не обязаны быть линейными функциями, а могут быть дифференцируемыми:

$$x_k = f(x_{k-1}, u_{k-1}) + w_{k-1},$$
$$z_k = h(x_k) + v_k.$$

Основная идея, применяемая в расширенном фильтре Калмана, состоит в приближении функций состояния и наблюдения с использованием их первых производных.

Этап экстраполяции

Экстраполяция (предсказание) вектора состояния системы по оценке вектора состояния и примененному вектору управления с шага (k-1) на шаг k:

$$\hat{x}_k^- = F\hat{x}_{k-1} + Bu_{k-1}.$$

Ковариационная матрица для экстраполированного вектора состояния:

$$P_k^- = FP_{k-1}F^T + Q.$$

Этап коррекции

Отклонение полученного на шаге k наблюдения от наблюдения, ожидаемого при произведенной экстраполяции:

$$\tilde{y}_k = z_k - H\hat{x}_k^-.$$

Ковариационная матрица для вектора отклонения (вектора ошибки) ():

$$S_k = HP_k^-H^T + R.$$

Оптимальная по Калману матрица коэффициентов усиления, формирующаяся на основании ковариационных матриц, имеющейся экстраполяции вектора состояния и полученных измерений (посредством ковариационной матрицы вектора отклонения):

$$K_k = P_k^-H^TS_k^{-1}.$$

Коррекция ранее полученной экстраполяции вектора состояния – получение оценки вектора состояния системы:

$$\hat{x}_k = \hat{x}_k^- + K_k\tilde{y}_k.$$

Расчет ковариационной матрицы оценки вектора состояния системы:

$$P_k = (I - K_kH)P_k^-.$$

Матрицы изменения состояния системы и наблюдения определяются Якобианами:

$$F_{k-1} = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_{k-1}^{-},\, u_{k-1}},$$

$$H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_{k-1}}.$$

Недостатки расширенного фильтра Калмана:

а) Расширенный фильтр Калмана (РФК) в общем случае не является оптимальным, в отличие от линейного аналога [10].

б) Если начальное вычисление состояния системы ошибочно, или процесс смоделирован некорректно, результаты могут быстро расходиться из-за линеаризации [10].

в) Вычисляемая матрица ковариации склонна к недооценке реальной ковариации и, поэтому, риски становятся мало связанными с реальной ситуацией без добавления «стабилизирующего шума» [11].

1.2.3 Фильтр частиц

Как говорилось выше, расширенный фильтра Калмана, не является универсальным способом решения задачи SLAM. В том случае, когда шумы не являются Гауссовскими, расширенный фильтр Калмана применять нельзя. В этом случае обычно применяют фильтр частиц, в котором используются численные методы взятия интегралов на основе методов Монте Карло.

При данном подходе, каждая частица характеризуется как возможное положение робота в данный момент времени. Будучи математическими моделями, эти фильтры описывают распределение вероятности в виде дискретного набора частиц в пространстве состояний, для задачи SLAM.

Первым шагом итерации локализации фильтра частиц является генерация нового частичного распределения для данной модели движения и примененного управления.

Следующим шагом, рассчитывается вес каждого значения. Частицам, для которых прогнозируемые значения соответствуют измеренным, даются большие веса.

И последний этап — это повторная выборка частиц. Она основывается на весах частиц из предыдущего распределения, которые берутся случайным образом, создавая новое распределения.

Главной проблемой является логарифмическая сложность фильтра, которая сильно зависит от числа частиц. При их малом количестве, в течение длительного времени работы алгоритма могут возникать серьезные ошибки, тем самым искажая полученный образ пространства. Однако, чем большим числом частиц мы будем располагать, тем больше времени потребуется для вычислений.

К самым распространённым алгоритмам, базирующимся на фильтре частиц можно отнести:

а) **Алгоритм FastSLAM.** FastSLAM – один из подходов к решению задачи SLAM. В основе алгоритма лежит фильтр частиц и применение Байесовской сети. В первые такой подход для решения задачи SLAM предложили в своей статье Monterlo, Trum, Kollerm, Wedbreit [12], а также его последующее улучшение в статье о FastSLAM 2.0 [13].

FastSLAM разделяет задачу локализации и картографии на множество подзадач, используя независимость состояния отдельных элементов модели SLAM. На рисунке 2 представлена диаграмма процесса SLAM, где V – текущее положение робота, Z – показания датчиков и U – сигнал управления.

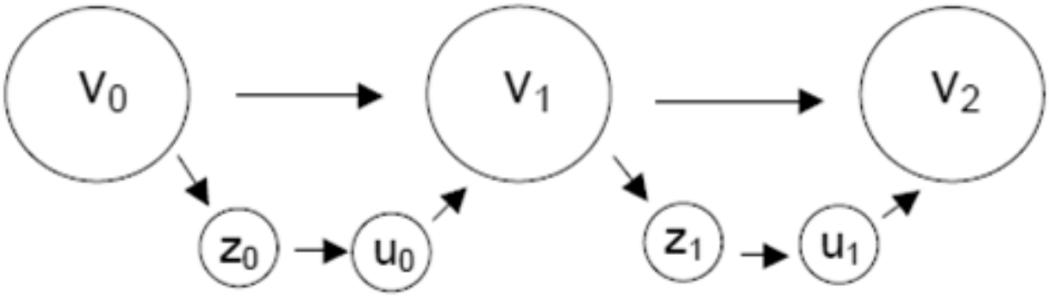

Рисунок 2 – Диаграмма процесса SLAM

Из диаграммы следует, что все наблюдения не зависят друг от друга, и единственное, что их связывает – это ошибка определения положения робота E(). Таким образом, если положение робота определяется абсолютно точно, то никаких зависимостей между отдельными наблюдениями быть не может. Конечно, в реальности положение робота определить абсолютно точно нельзя, в этом и заключается сущность проблемы задачи SLAM [14], но сам факт независимости ориентиров друг от друга, мотивировал создателей FastSLAM обрабатывать каждый ориентир отдельно.

Предположим, что робот находится в одномерном пространстве, и его положение характеризуется одной переменной x. Тогда $p(x)$ будет распределением вероятности x. Если x отражает положение робота и ориентиров в многомерном пространстве, тогда распределение вероятности $p(x)$ будет определять вероятность всех возможных переменных состояния.[15] Следовательно, $p(x \mid \{u_o, u_1, \dots, u_i\}, \{z_o, z_1, \dots, z_i\}$ описывает вероятность всех величин текущего состояния системы, таких как показания датчиков и информация о положении робота, полученные в момент времени i.

Для удобства обозначим $U_i = \{u_o, u_1, \dots, u_i\}$ и $Z_i = \{z_o, z_1, \dots, z_i\}$, а x в свою очередь содержит положение робота v и положение ориентиров $p_0, p_1, \dots, p_m$. $p(x \mid U_i Z_i)$ можно представить с следующем виде:

$$p(x \mid U_i Z_i) = p(v, p_0, p_1, \dots, p_m | U_i Z_i).$$

Воспользуемся основами теории вероятности для упрощения задачи SLAM. Если предположить, что A и B две независимые случайные величины, тогда можно сказать, что $p(A, B) = p(A) * p(B)$. Но это выражение несправедливо, если A зависит от B. В этом случае оно будет иметь вид $p(A, B) = p(A) * p(B|A)$. Как известно, оценка положения ориентиров зависит от оценки положения робота, а это значит, что $p(v, p_0, p_1, \dots, p_m|U_i Z_i)$ можно представить, как:

$$p(v, p_0, p_1, \dots, p_m|U_i Z_i) = p(v|\, U_i Z_i) * p(p_0, p_1, \dots, p_m|U_i Z_i v).$$

В силу независимости наблюдаемых ориентиров друг от друга можно разделить $p(p_0, p_1, \dots, p_m|U_i Z_i v)$ на m независимых выражений:

$$p(v|\, U_i Z_i) * p(p_0, p_1, \dots, p_m|U_i Z_i v) =$$
$$= p(v|\, U_i Z_i) * p(p_0|\, U_i Z_i v) * \; p(p_1|\, U_i Z_i v) * \dots * p(p_m|\, U_i Z_i v)$$

В итоге мы получаем результирующее выражение для распределения вероятности, имеющее следующий вид:

$$p(x|\, U_i Z_i) = p(x|\, U_i Z_i) * \prod\nolimits_m p\,(p_m|U_i Z_i v).$$

Если посмотреть на полученное выражение, то становиться очевидно, что проблема SLAM разделена на $m + 1$ задач, а также ни одна из оценок положения ориентиров не зависти от других. [16] Это позволяет решить проблему полиномиальной сложности алгоритма присущей РФК, и избежать её в FastSLAM. Цена, которую приходится платить за такое упрощение – это

падение точности, связанное с игнорированием корреляции ошибок оценки положения робота [15].

Стоит отметить, что FastSLAM одновременно отслеживает несколько возможных маршрутов, в то время как расширенный фильтра Калмана не хранит даже одного, а лишь работает с положением робота – последним шагом текущей маршрута. В оригинальном виде, FastSLAM сохраняет маршрут, но в вычислениях использует только предыдущий шаг.

б) **Алгоритм DP-SLAM.** DP-SLAM - метод с принципиально другим подходом к представлению окружающей среды. В этой постановке робот находится в среде с препятствиями сложной формы, какими являются, например, помещения жилых домов [17]. В таких условиях практически невозможно выделить отдельные ориентиры.

Вместо положения ориентиров строится полная детализированная карта, для получения достоверных результатов одновременно поддержка нескольких гипотетических карт, по которым и происходит фильтрация с применением последовательных методов Монте-Карло.

Для хранения и структурирования большого числа промежуточных карт авторами в своей работе [18] была реализована древовидная структура, представленная на рисунке 3.

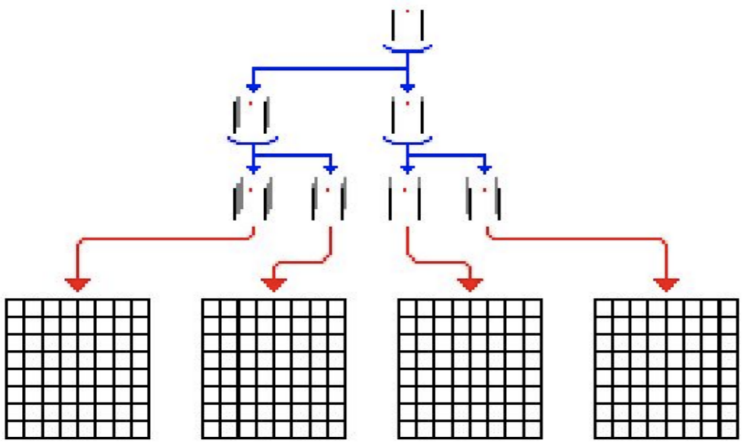

Рисунок 3 – Система хранения “гипотетических” карт в алгоритме SLAM

Стоит обратить внимание, что, несмотря на большую общность в отношении среды, этот метод рассчитан на определенный тип сенсоров - точные лазерные дальномеры и может потребовать отдельные усовершенствований для других типов.

в) **Алгортим HectorSlam.** Описание алгоритма HectorSlam представлено в работе [13]. Данный алгоритм использует в качестве модели рабочей среды всю накопленную карту. Карта представлена в виде сетки занятости, где значения функции $M(P_m) = 1$, если соответствующая ячейка занята, и $M(P_m) = 0$, если ячейка свободна. Чтобы смоделировать вероятностное распределение точек, относящихся к объектам на карте, авторы используют билинейную интерполяцию. При этом функции карты становятся непрерывными, и легко вычислить её градиент в любой точке. В качестве меры совпадения скана и карты используется среднеквадратичная ошибка всех точек скана. Тогда для оценки положения ξ^* мобильного робота необходимо воспользоваться методом наименьших квадратов:

$$\xi^* = \arg\min \sum_{i=1}^{n} [1 - M(S_i(\xi))]^2,$$

где $S_i(\xi)$ – координаты i-й точки скана, преобразованной в систему координат помещения, если считать, что скан получен из положения ξ.

Учитывая простоту вычислений градиента, минимизация этой функции выполняется методом Гаусса-Ньютона. Однако оптимизируемая функция имеет локальные минимумы, поэтому авторы предложили искать решение в нескольких этапов, каждый из которых имеет дело с картой различного разрешения. Начиная с карты с большим разрешением, алгоритм находит неточную оценку положения, близкую к истинному. Затем выбирая все меньшие разрешения, эта оценка улучшается до тех пор, пока не достигнет необходимой точности, либо поиск останавливается на карте с наилучшим

разрешением. Наконец, применительно к задаче SLAM, оценка положения мобильного робота обновляется с использованием ФК.

Рассмотренный алгоритм позволяет получить точную оценку положения робота, причем, накапливаемая ошибка мала для использования всей карты при сопоставлении скана. В качестве недостатка можно выделить необходимость в построении и обновлении нескольких карт с помощью различных разрешений, что обеспечивает эффективность алгоритма.

г) **Алгоритм Gmapping**. Данный алгоритм является одним из основных алгоритмов SLAM в операционной системе ROS, что обусловлено его относительной эффективностью и простотой подхода.

В основе данного алгоритма лежит метод рекурсивной фильтрации частиц Рао-Блэквелла на сетчатой карте [19, 20]. Каждая частица является потенциальной траекторией робота и содержит в себе, как потенциальную траекторию робота, так и информацию о состоянии карты рабочей среды в текущий момент времени. При наивном подходе на каждой итерации алгоритма генерируются несколько новых частиц из каждой предыдущей, при этом их число растет экспоненциально, что приводит к невозможности реализации данного алгоритма.

Авторы предложили два подхода, которые позволяют увеличить производительность фильтра для возможности решения задачи SLAM в реальном времени, это:

а) вспомогательная функция распределения, которая учитывает точность сенсоров, таким образом, позволяя создавать частицы весьма точным способом;

б) техника адаптивного ресемплинга, которая поддерживает разумное число частиц для достаточной точности локализации и, в то же время, снижения риска опустошения множества частиц.

Вспомогательная функция распределения вычисляется путем оценки степени похожести скана и карты при помощи процедуры регистрации и одометрической информации.

Ключевой идеей данного фильтра является оценка условной вероятности каждой частицы, которую можно разложить на множители, согласно теореме Рао-Блэквелла-Колмогорова:

$$p(x_{1:t}, m|z_{1:t},\ u_{1:t-1}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}),$$

где $x_{1:t}$ – траектория робота, из последовательных положений, m – карта помещений, $z_{1:t}$ – последовательность вектора наблюдения, $u_{1:t-1}$ – измерения одометрии.

Работу фильтра можно описать следующим алгоритмом

а) Семплирование – формирование выборки частиц следующего поколения $\{x_t^{(i)}\}$ из вспомогательной функции распределения π на основе предыдущей выборки $\{x_{t-1}^{(i)}\}$ и вероятностной модели одометрии.

б) Взвешивание степени важности каждой частицы $w_t^{(i)}$, основываясь на её апостериорной вероятности и значении вспомогательной функции распределения.

в) Ресемплин – частицы выбираются с заменой, пропорционально их весу, вследствие чего веса оставшихся частиц выравниваются.

г) Для каждой частицы вычисляется соответствующая оценка карты m на основе её траектории $x_{1:t}$ и истории наблюдений $z_{1:t}$.

Реализация рассмотренного алгоритма описана в [21]. По сути, рассмотренный алгоритм предназначен для фильтрации оценки положения робота, которая вычисляется некоторым методом регистрации сканов, и служит для повышения точности локализации. Отсюда видно, что достоинством данного алгоритма является высокая точность построения карты, однако при использовании метода регистрации сканов с низкой погрешностью целесообразность данного алгоритма ставится под сомнение, учитывая его достаточно высокую вычислительную сложность.

1.2.4 Методы основанные на графах

Методы SLAM основанные на использовании графов, являются одними из самых современных, подходом к задаче одновременной локализации и построения карты.

Одним из самых популярных методов, основанных на графах, является GraphSlam. Предпосылкой к появлению этого метода была возможность представления задачи SLAM в виде разреженного графа и связей между его узлами [22]. Узлы графа представляют собой положение робота и элементы карты. Относительные перемещения робота и положения элементов карты (относительно робота), выражаются на графе как связи между соответствующими узлами.

На рисунке 4, показан пример поэтапного представления процесса SLAM. Каждый узел в графике соответствует позиции робота. Некоторые позиции связаны между собой ребрами, которые соответствуют положению препятствий относительно позиции робота, возникающие в результате измерений. В то время как другие ребра соответствуют изменению позиции робота по измерению одометра робота.

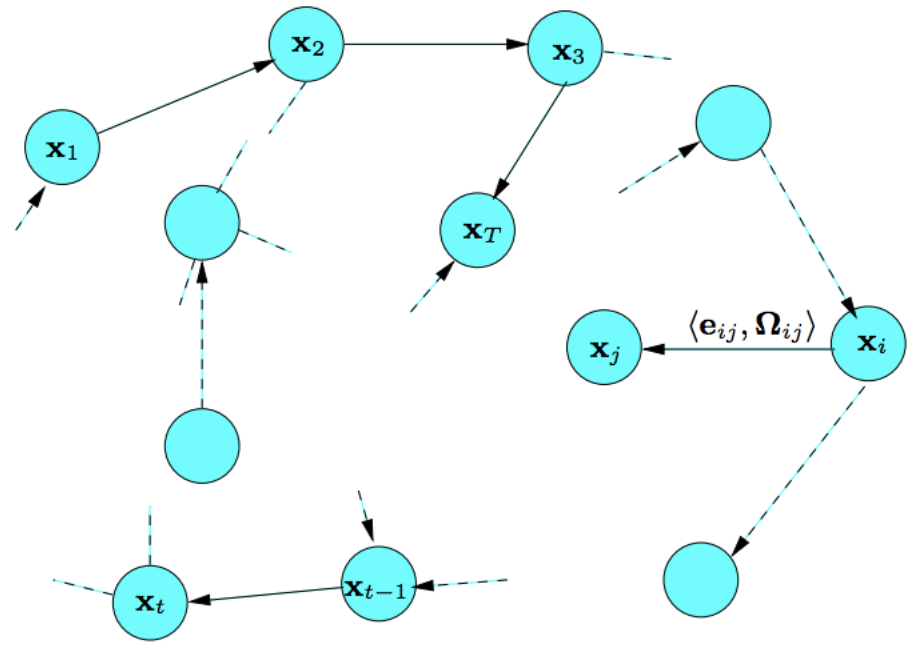

Рисунок 4 - Модель графа

Таким образом, решение SLAM может быть найдено вычислением состояния с минимальной энергией этой сетки.

Теперь рассмотрим преимущества и недостатки данного подхода.

Существенное преимущество GraphSlam состоит в том, что по сравнению с фильтром Калмана количество вычислений и памяти, которые требуются для обновления и сохранения ковариационной матрицы, не растет квадратично с числом элементов [23].

Главным недостатком методов SLAM основанных на графах, это требования больших затрат на вычисление, если робот прошел достаточно длинный путь [24].

1.3 Выводы по первой главе

В первой главе выпускной квалификационной работы были рассмотрены основные методы и подходы к решению задачи одновременной локализации и построения карты. Обзор показал, что существует несколько работоспособных реализаций алгоритмов SLAM, однако различные подходы и реализации имеют свои преимущества и недостатки, такие как:

а) Проблема сходимости,

б) Применимость алгоритма в реальных условиях,

в) Вычислительная сложность алгоритма,

г) Точность и эффективность.

В результате проведенного обзора, для реализации алгоритма SLAM для данной работы, был выбран подход на основе фильтра частиц. Этот подход был выбран из-за современности данного подхода, его эффективности для решения задачи SLAM для мобильных роботов и большого количества литературы по различным алгоритмам SLAM основанным на использовании фильтра частиц.

2 ОПИСАНИЕ РАЗРАБОТАННОГО АЛГОРИТМА SLAM

Как было сказана выше для разработки алгоритма на основе данных получаемых с помощью одометров и ультразвуковых или инфракрасных датчиков, был выбран подход, базирующийся на использовании фильтра частиц. Данный подход является одним из самых современных и самым распространённых подходом к решению задачи SLAM, при этом он лишен недостаток ФК и РФК.

В качестве модели мобильного робота будем использовать модель двухколесного мобильного робота. Данная модели была выбрана для простоты и наглядности, а также по причине распространенности данной схемы среди мобильных роботов.

2.1 Модель объекта управления

Как говорилось выше, в качестве объекта управления для апробирования работы алгоритма SLAM мы будем использовать модель двухколесного мобильного робота. Уравнение кинематики, описывающее перемещение двухколесного мобильного робота имеет следующий вид:

$$\begin{cases} \dot{x} = v \cdot \sin\varphi\,, \\ \dot{y} = v \cdot \sin\varphi\,, \\ \dot{\varphi} = \omega, \\ v = \frac{1}{2a_0}(U_1 + U_2), \\ \omega = \frac{1}{2a_0 l}(U_1 + U_2), \end{cases}$$

где x и y – координаты объекта на плоскости, v и ω – линейная и угловая скорость объекта $(0 \le v \le v_{max};\ -\omega_{max} \le \omega \le \omega_{max})$, φ – угол,

характеризующий направление движения объекта относительно оси ординат. l – половина расстояния между колесами, r – радиус колеса.

На рисунке 5 приведено схематическое изображение мобильного робота.

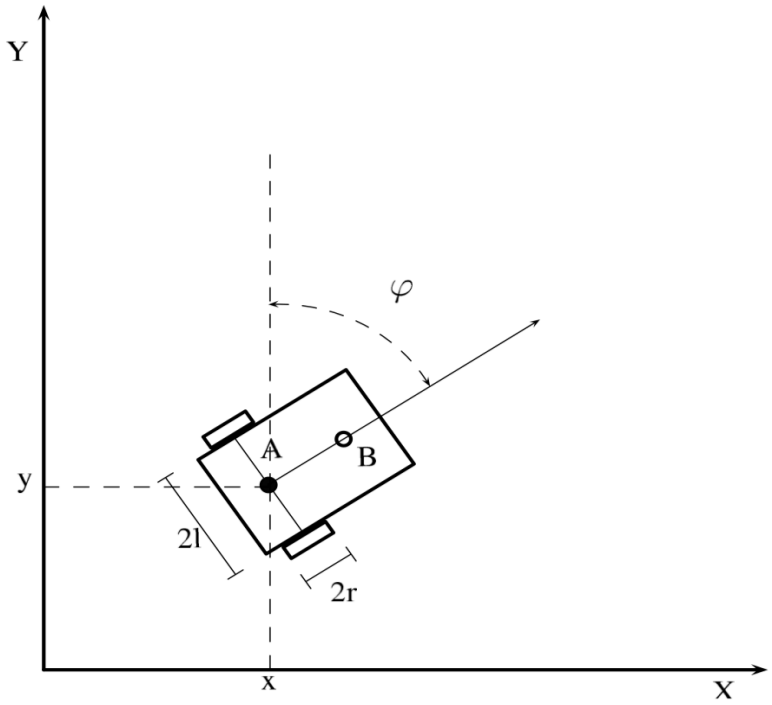

Рисунок 5 – Кинематическая модель двухколесного мобильного робота

На каждом шаге квантования времени Δt, мы будем вычислить угол поворота каждого колеса ($\Delta\alpha_R, \Delta\alpha_L$ - угол поворота правого и левого колеса робота соответственно). Определить перемещение (ΔS) и угол поворота (Δφ) робота относительно точки А можно по формулам (1).

$$\begin{cases} \Delta S = \frac{r(\Delta\alpha_R + \Delta\alpha_L)}{2}, \\ \Delta\varphi = \frac{r(\Delta\alpha_R - \Delta\alpha_L)}{2l}. \end{cases} \quad (1)$$

2.2 Датчики робота

В данной работе в качестве дальномеров мы будем использовать ультразвуковые или инфракрасные датчики.

Стоит отметить, что никакой разницы какие датчики использовать для большинства алгоритмов SLAM нет, однако стоить учитывать, что при использовании ультразвуковых датчиков получаемые значения могут быть серьезно искажены [26]. Это может вносить определенные проблемы в работу алгоритма SLAM.

Также стоит отметить, что в большинстве алгоритмов SLAM в качестве датчиков определения расстояния, используются датчики, позволяющие за одну итерацию цикла алгоритма сформировать двумерную карту окружающего пространства, такие как лидар, кинект и прочие. Эти датчики отличаются высокой эффективности для задачи SLAM так, как они позволяют отслеживать положения множества ориентиров, но минус этих датчиков заключается в высокой стоимости из-за чего их применение ограничено.

Из этого следует, что при использовании обычных датчиков мы столкнемся с несколькими проблемами:

а) При использовании обычных жестко закрепленных датчиков мы будем получать не более одного значения с одного датчика, а возможно и вообще ни одного (если, например препятствия будут находиться вне диапазона датчиков), из-за чего нам придётся полагаться только на данные одометров и как следствие это будет приводить к накоплению ошибок в определении положения робота.

б) Большинство алгоритмов SLAM в качестве ориентиров используют так называемые особые точки: углы, выступы, некоторые характерны объекты на местности и др. При использовании обычных датчиков такой подход невозможен, что приводит к необходимости использовать другие подходы, например сетчатую карту.

Для решения указанных проблем с использованием двух обычных датчиков, воспользуемся следующим подходом.

Прикрепим датчики на сервоприводе так чтобы они могли вращаться на 90 градусов в плоскости XY, и расположим их как показано на рисунке 6.

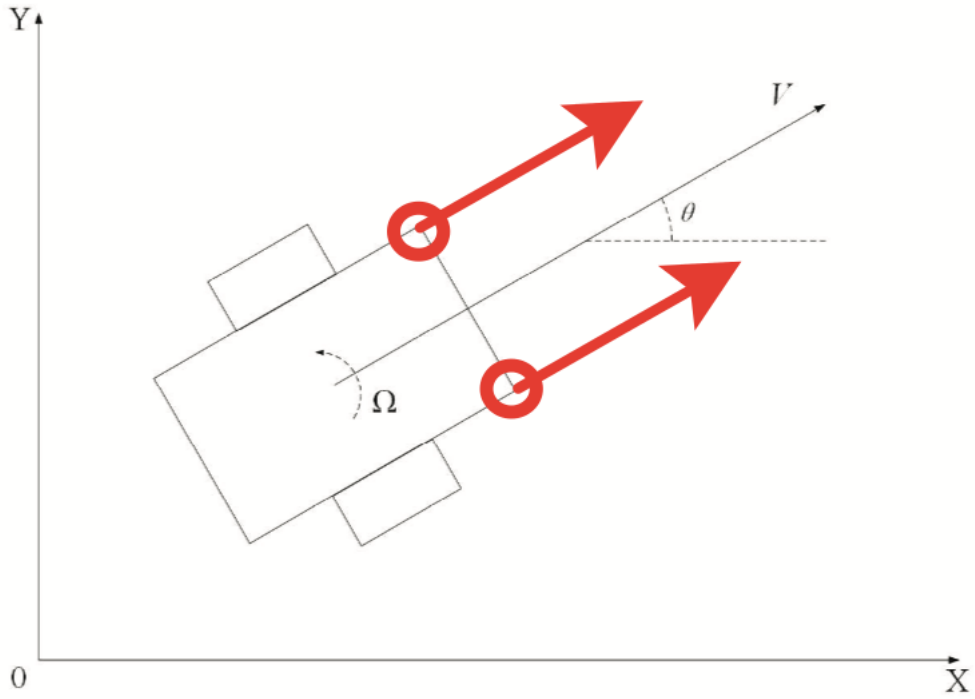

Рисунок 6 – Схема расположения датчиков на мобильном роботе

Данный подход не является универсальным, но он позволяет нам полностью сканировать всю переднюю полусферу робота, с помощью всего двух датчиков.

2.3 Карта пространства

В большинстве алгоритмов SLAM, данные об окружающем пространстве хранится в так называемой карте-сетке (grid map), которая представляет собой матрицу, каждый элемент которой описывает прямоугольный участок пространства $M_{i,j}$, где i – изменяется от 0 до N_y и j – от 0 до N_x.

В самой простой реализации карты-сетки, если элемент матрицы равен 1, то это означает, что в соответствующем участке находится объект, если элемент равен 0, то участок свободен. Данное представление сравнимо с бинарным растровым изображением.

В более сложной модели карта-сетка пространства включает в себя информацию о вероятности нахождения какого-либо объекта в каждой ячейку карты и, следовательно, каждому элементу присваивается значение от 0 до 1. Где 1 означает, что достоверное нахождение объекта помещения, соответствующая карта, а 0 отсутствие каких-либо объектов. Такой подход позволяет учесть статистические характеристики измерений, что, в свою очередь, повышает детализировалось получаемой карты. При этом сравнение скана с картой проводится путем сложения плотностей вероятности, содержащихся в ячейках карты, в которые попали точки текущего скана. В отличие от предыдущего варианта, здесь можно более точно оценить степень совпадения, и как следствие определить преобразование координат с меньшей погрешностью.

На рисунке 7 приведена графическая интерпретация некоторого участка помещения в виде карты-сетки:

а) с бинарным представлением,

б) с вероятностным.

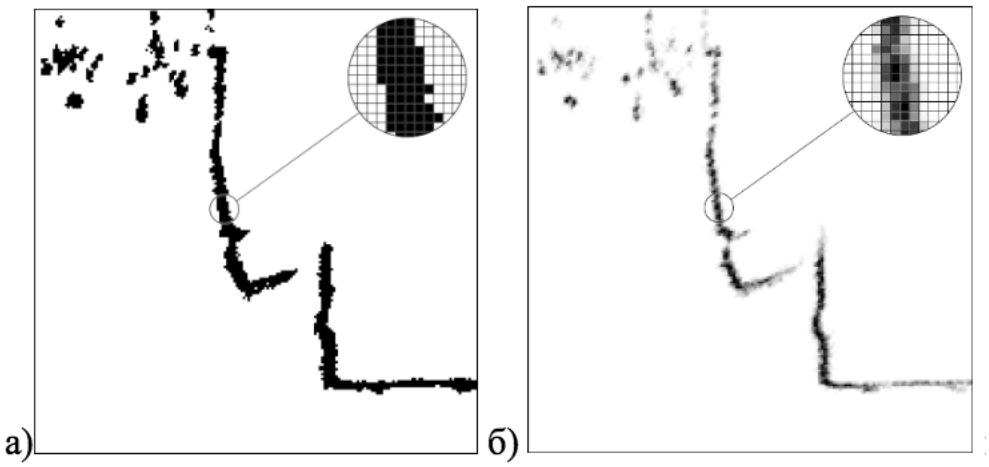

Рисунок 7 – Графическая интерпретация карты

В данном подходе есть существенный недостаток, чтобы обеспечить высокую детализацию карты, необходимо выбрать минимальный размер ячейки, сопоставимый с требуемой точностью локализации. Это естественно увеличивает объем информации для хранения карты.

В данной работе для расчета веса частиц мы будем использовать карту-сетку с вероятностным представлением, это позволит с большей точностью вычислять вес частиц, а следовательно, точнее определять положение робота. А для отображения препятствий непосредственно на самой карте, которую мы будем строить для отображения результатов моделирования, мы будем использовать подход с бинарным представлением.

2.4 Фильтр частиц

Фильтр частиц, или многочастотный фильтр - в задачах одновременной локализации и построения карты, является одним из самых распространённых методов для нахождения положения робота.

Фильтр частиц позволяет получить оценку (приближенное значение) параметров системы или объекта (параметры А) которые нельзя измерить напрямую. Для построения этой оценки фильтр использует измерения других параметров (параметры Б) связанных с первыми. Покажем это на рисунке 8.

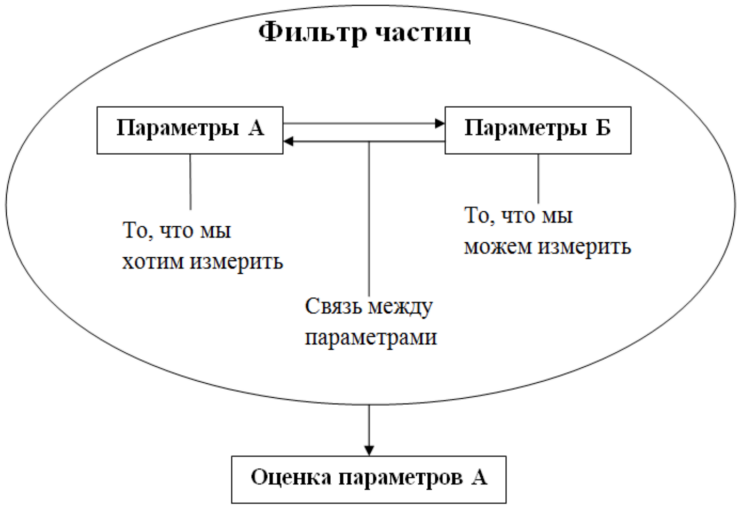

Рисунок 8 – Общая схема работы фильтра частиц

Для оценки параметров А фильтр создает множество гипотез (частиц) о текущем значении этих параметров. В начальный момент времени эти гипотезы абсолютно случайны, но на каждой итерации цикла фильтрации фильтр будет убирать гипотезы, которые не пройдет проверку достоверности, основанную на измерениях параметров Б.

На рисунке 8 представлен более подробный алгоритм определения положения робота по положению ориентиров. По нему видно, что для определения положения робота на каждом этапе необходимо фиксировать хотя бы по одному ориентиру, иначе невозможно применять алгоритм. Так как в данной работе для расчета показаний, мы будем использовать сетчатую карту.

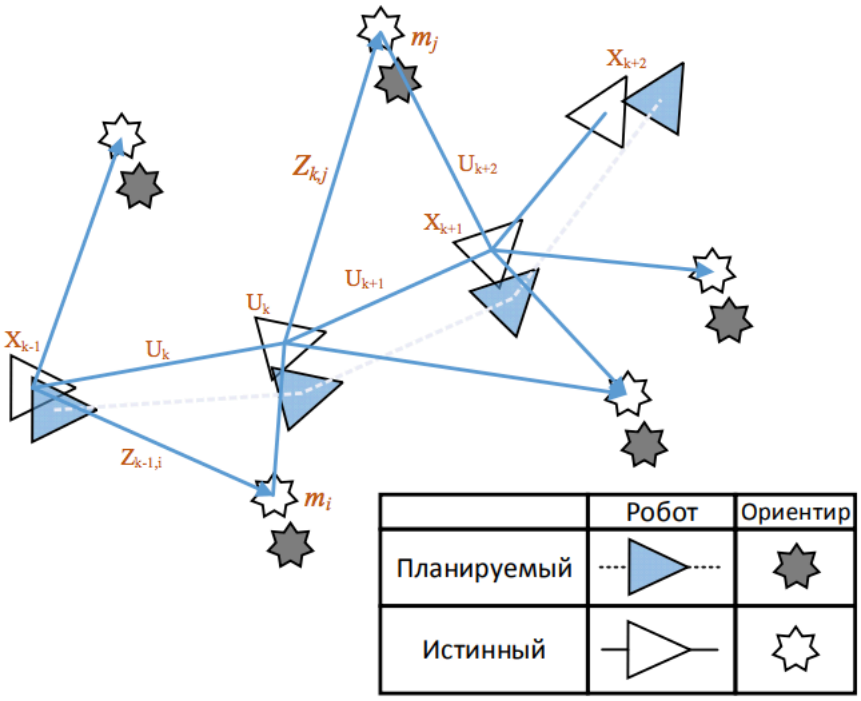

Рисунок 9 – Схема работы алгоритма SLAM

В данной работе рассмотрен один из простейших вариантов применения фильтра частиц. Робот движется в двумерном пространстве и может измерять дальность до определенных объектов в этом пространстве (ориентиров).

Алгоритм фильтра частиц разделим на две части: инициализация и основной цикл фильтрации.

Инициализация. Прежде чем приступать к фильтрации, фильтр частиц нужно инициализировать — задать параметры, начальное распределение и другие условия работы фильтра.

Основной параметр фильтра частиц — число этих самых частиц. Чем больше частиц — тем точнее фильтр, и тем больше вычислений нужно проводить на каждой итерации основного цикла.

Начальное распределение частиц зависит от априорной информации. Например, если нам известно, что робот находится в клетке с координатами (0, 0), то все частицы должны быть случайным образом распределены внутри этой клетки. А также ориентация каждой частицы должна быть равна 0.

Основной цикл фильтрации. Основной цикл фильтрации разделен на три фазы:

а) **Движение**. На этом этапе робот совершает движение и, так как движение робота происходит с погрешностями, теряет информацию о своем местоположении. Для большинства мобильных роботов этот этап соответствует расчету одометров.

б) **Измерение.** На этом этапе робот совершает измерения и получает новую информацию о своем местоположении.

в) **Отсев.** На этом этапе происходит выборка частицы с максимальным весом, которая соответствует рассчитанному положению и ориентации робота.

Рассмотрим каждый из этапов по отдельности:

В начальный момент времени робот производить опрос с помощью своих вращающихся датчиков, это необходимо для формирования начальной карты, на основе которой в дальнейшем будет рассчитываться веса частиц.

2.4.1 Движение

Как говорилось выше, на этом шаге происходит расчет положения робота на основе данных, получаемых с одометров, этот этап необходим для предсказания положения робота, а следовательно, и положения частиц. Расчет положения двухколесного мобильного робота по данным с одометров был приведен в главе 2.1.

2.4.2 Измерение

Перед началом работы датчиков робота, ему необходимо остановится. Это необходимо для того, чтобы робот смог собрать достаточно информации о своем перемещении и на основе расчета своего положения по одометрам и собранно информации, робота производит расчет своего положения на основе фильтра частиц.

Каждая частица представляет собой возможное положение и ориентацию для мобильного робота.

Предположим, что за первое перемещение робота, его одометры рассчитали его положение и ориентацию с некоторой ошибкой, тогда после остановки робота, мы производим опрос датчика и на основе полученных координат рассчитываем для каждой частицы её вес, при этом позиция каждой частицы определяется из смещения роботом, т.е. происходит разброс каждой частицы на основе пройденного робот расстояния и изменения его ориентации определяемой по одометрам. Поэтому важно, чтобы перемещения робота были минимальными, в противном случае разброс частиц будет слишком большим и будут наблюдаться большие рывки в определении положения робота и как следствие получаемая карта может быть серьёзно искажена.

2.4.3 Отсев

На данном этапе происходит отсев частиц с маленьким весов, и выбирается частица с самым большим весов, координаты и ориентация этой частицы и становится новыми координатами положения и ориентации робота и расчет в следующей итерации по одометрам начнется от этих координат и с рассчитанной ориентацией

2.5 Выводы по второй главе

Во второй главе выпускной квалификационной работы была проведена разработка алгоритма одновременной локализации и построения карты, с использованием выбранного подходе на основе фильтра частиц. В ходе работы были рассмотрены следующие проблемы:

а) Рассмотрена модель мобильного робота.

б) Рассмотрены используемые датчики.

в) Взамен ориентиров, которые невозможно использовать из-за ограничения датчиков, было рассмотрено использование сетчатой карты.

г) Предложена реализации алгоритма SLAM на основе фильтра частиц.

3 МОДЕЛИРОВАНИЕ РАБОТЫ АЛГОРИТМА

3.1 Требования к моделированию

В данной работе для моделирования работы алгоритма была использована среда моделирования V-REP. Среда V-REP представляет собой среду симулирования работы различных роботов, при этом пользователю нет необходимости иметь доступ к реальной машине, что позволяет проводить исследования и моделирования без привязки к роботу.

С описанием работы среды и документацией по ней можно ознакомиться в статье V-REP [28].

В качестве мобильного робота для моделирования алгоритма SLAM была использована модель робота Pioneer. Этот робот позволяет полностью реализовать разработанный алгоритм SLAM, а также для этого робота можно применить алгоритм Брайтенберга, который позволит обеспечить передвижения робота в автоматическом режиме.

В качестве языка программирования был выбран язык программирования Python 3.6 [29]. Данный язык был использован благодаря своей простоте, а также развитой системе API, позволяющий с легкостью связать программу с со средой моделирования V-REP, а также получать всю необходимую информацию о процессе моделирования.

Реализация разработанного алгоритма SLAM, на языке программирования Python 3.6 представлен в приложении А.

В качестве пространства для проведения моделирования в среде V-REP создадим сцену, как показа на рисунке 10.

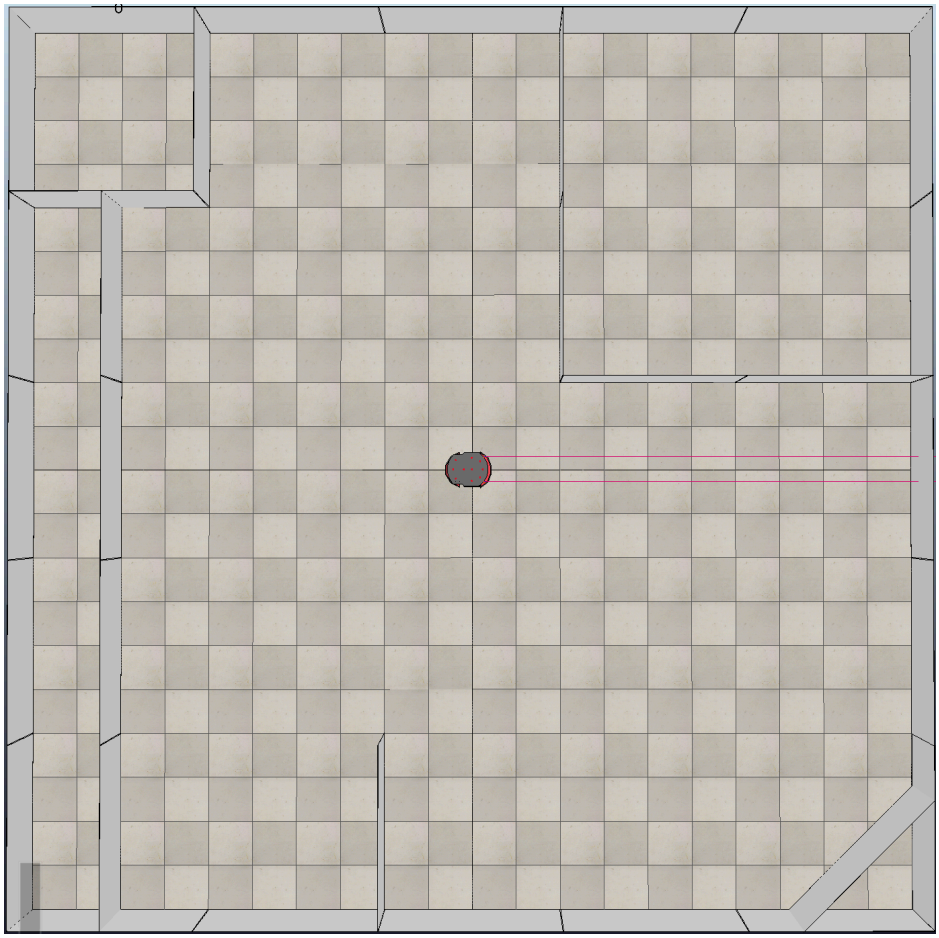

Рисунок 10 – Сцена в среде V-REP

Теперь нам необходимо определиться с количеством частиц, которое мы будем использовать при моделирования. Как говорилось раньше от числа частиц зависит точность в определении положения мобильного робота и соответственно сложность вычисления. В данной работе для всех случаев мы будем использовать 50 частиц, этого количества достаточно для определения положения робота. А также покажет эффективность при использовании разных датчиков для решения задачи SLAM.

На рисунке 11 показано расположение препятствий в начальный момент времени: зеленым цветом показано положения препятствий, белым пустое пространство.

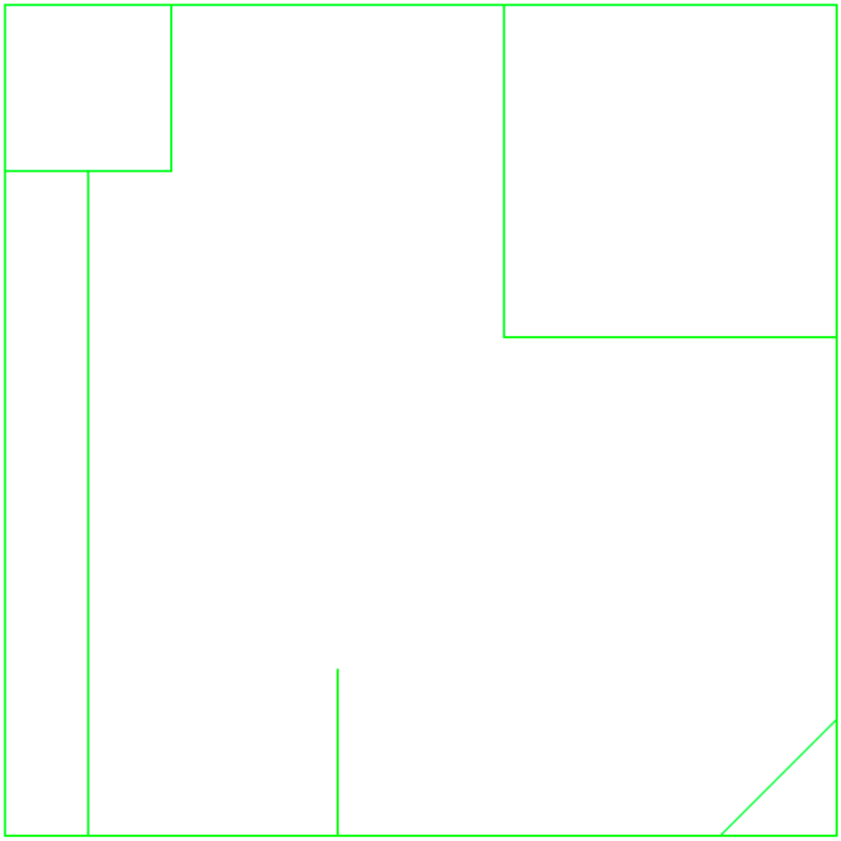

Рисунок 11 – Расположение препятствий

3.1 Результаты моделирования

Основной задачей для проведения моделирования является получение наглядных результатов и проверка правильности работы алгоритма.

На рисунке 12 представлен результат работы программы по построению карты и определения положения робота при использовании только данных, получаемых с одометров мобильного робота.

Как видно из рисунка 12 общая карта которую строит мобильный робот, как и положение робота, совершенно не соответствует действительности, и чем больше времени будет работать робот, тем хуже будут становиться результаты.

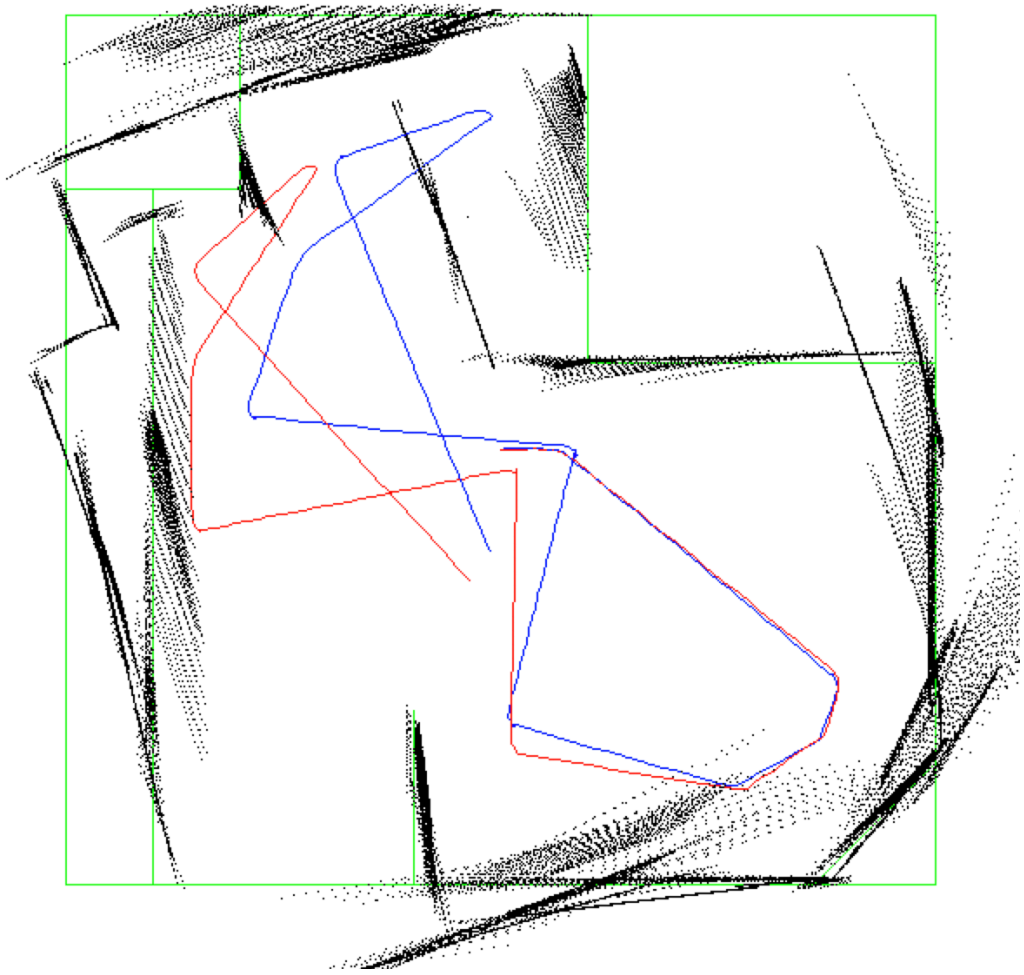

Рисунок 12 – Результат построение карты с помощью только одометров робота

На рисунке 13 представлен результат работы программы построение карты и определения положения робота с использованием данных, получаемых с одометров, а также ультразвуковых датчиков.

Данные, получаемые при использовании ультразвуковых датчиков в качестве дополнительной коррекции в разработанном алгоритме SLAM, показали свою пригодность для решения задачи построения карты местности.

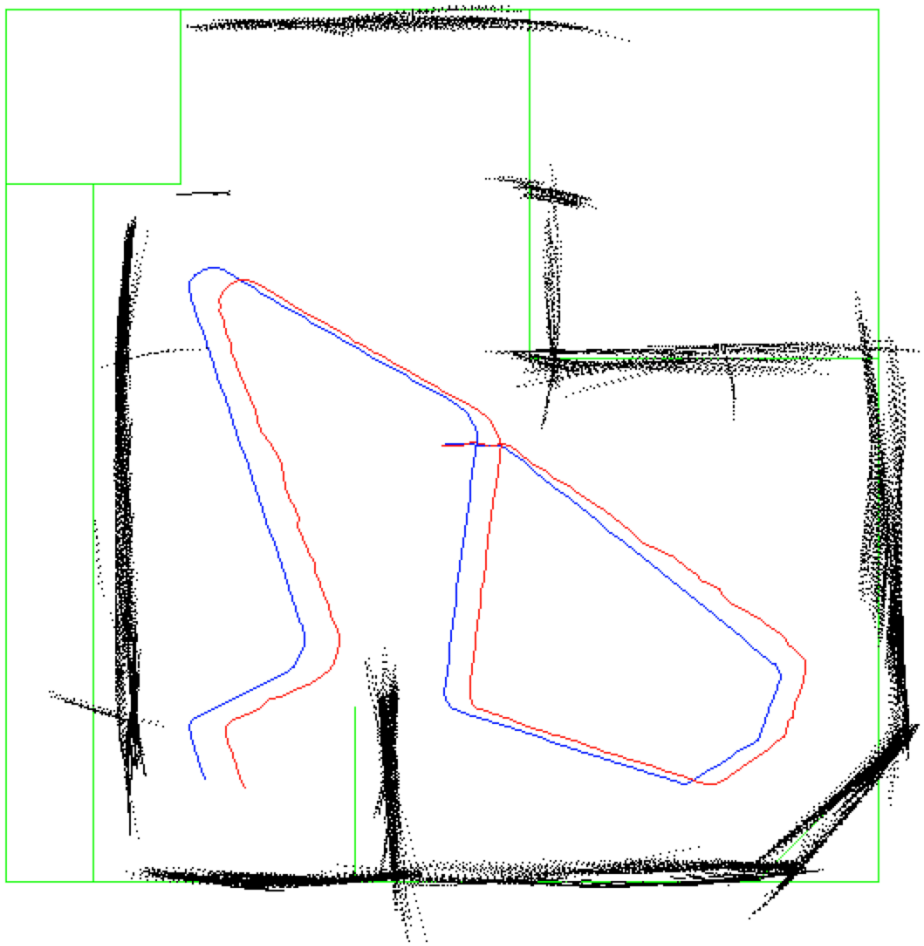

Рисунок 13 – Результат построения карты с помощью данных, поступающих с одометров робота и ультразвуковых датчиков

Хотя результаты, приведенные на рисунке 13 и заметно лучше, по сравнению с определением положения робота только по данным одометров, все же наблюдается значительное смещение рассчитанного положения и ориентации робота, от реальных, и как следствие серьезные смещения в определении препятствий на карте. Это обуславливается большими погрешностями в измерениях ультразвуковых датчиков, что вводят в алгоритм SLAM множество ложных препятствий, влияющих на его работу.

На рисунке 14 представлен результат работы программы построение карты и определения положения робота с помощью данных, получаемых от одометров и лазерных датчиков.

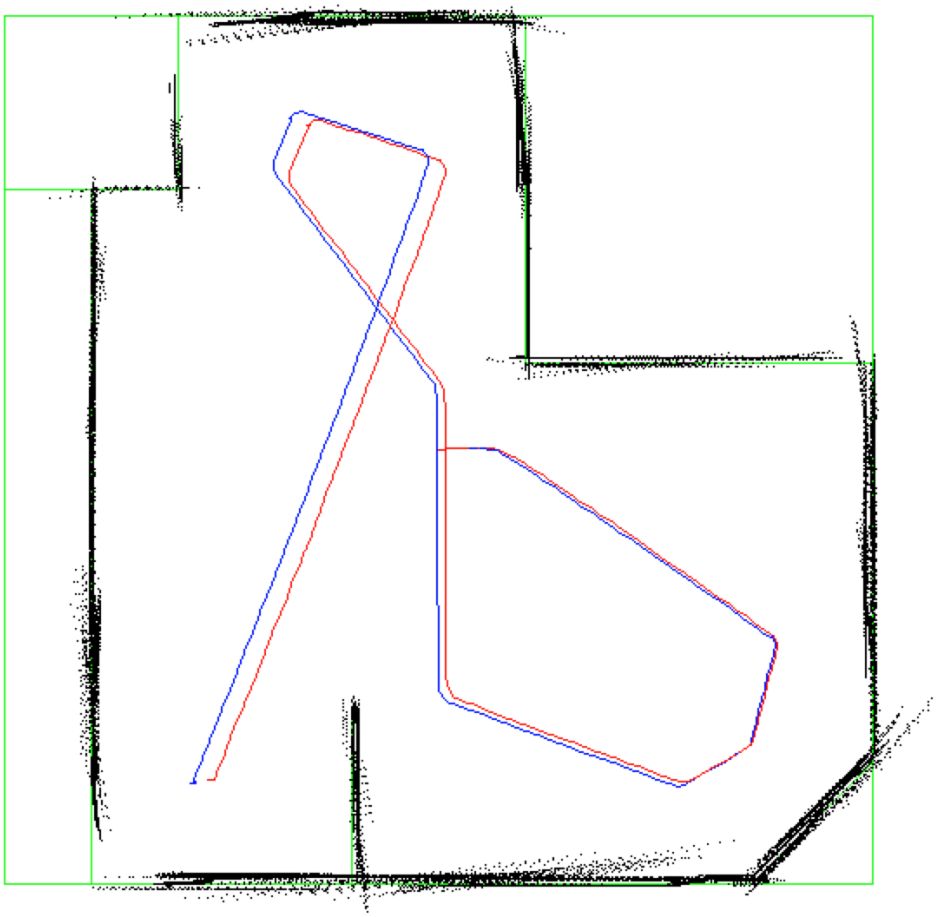

Рисунок 14 - Результат построение карты с помощью данных, поступающих с одометров робота и лазерных датчиков

Как видно из рисунка 14 результат работы программы в целом соответствует реальным положениям препятствий, что подтверждает правильность работы алгоритма. Смещения и отклонения, которые видны на построенной карте объясняются в первую очередь, не достаточным количеством ориентиров для расчета веса частиц в некоторые моменты перемещения робота.

3.2 Выводы по третей главе

В данной главе было проведено моделирование работы алгоритма SLAM в среде моделирования V-REP, который основан на использование фильтра частиц, а также произведено сравнение результатов работы данного алгоритма по сравнению с построением карты на основе данных только с одометров. А также было проведено сравнение результатов на основе данных полученных с помощью ультразвуковых и инфракрасных датчиков.

Полученные результаты подтверждают эффективность разработанного алгоритма SLAM для построения карты мобильным роботом с среде моделирования V-REP.

Но стоит отметить, что в результате анализа полученных результатов, можно утверждать, что при использовании в качестве дальномером инфракрасных датчиков, разработанный алгоритм SLAM максимально точно строит карту пространства, в то время как использование ультразвуковых датчиков вносит определенные погрешности, из-за чего определение положения робота производиться со серьёзными ошибками, а следователь и строящаяся карта также содержит серьезные погрешности. Но даже в случае с ультразвуковыми датчиками, получаемая карта больше соответствует действительности, чем при использовании только одометров.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был проведен обзор и анализ существующих алгоритмов SLAM, в двухмерном пространстве. Был рассмотрены основные преимущества и недостатки различных метод и подходом к решению задачи SLAM для мобильного робота.

Был предложен алгоритм решения задачи одновременной локализации и построения карты для мобильного робота на основе данных, получаемых с одометров, а также ультразвуковых или инфракрасных датчиков.

Разработанный подход позволяет с достаточной точность определить положение мобильного робота в двухмерном пространстве, а также построить карту окружающей его местности.

В ходе проведения моделирования работы разработанного алгоритма SLAM на основе фильтра частиц, тот подтвердил свою работоспособность для двухколесного мобильного робота, но также была выявлена большая зависимость, точности определения положения робота, от точности используемых датчиков.

В дальнейшем разработанный алгоритм может быть использован в качестве основы для решения задачи SLAM в трехмерном пространстве, а также на основе более совершенных датчиков для определения положения робота и препятствий.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. A new approach to linear filtering and prediction problems / R. E. Kalman // Journal of Basic Engineering. – 1960. – №2. – С. 35 – 45.
2. Welch, G. An Introduction to the Kalman Filter : учеб. пособие / G. Welch, G. Bishop // Department of Computer Science University of North Carolina. – M. : 2006. – 35.
3. Enhanced SLAM for a Mobile Robot using Extended Kalman Filter / K.S. Choi, and S.J. Lee // International Journal of Precision Engineering and Manufacturing. – 2010. – №2. – С. 255–264.
4. О некоторых особенностях применения не доопределённых моделей в робототехнике / В.Э. Карпов // V Международная научно-практическая конференция «Интегрированные модели и мягкие вычисления в искусственном интеллекте». – М., – 2009. – Физматлит. – С. 520-532.
5. An EKF SLAM Algorithm for Mobile Robot with Sensor Bias Estimation / X. Xie, Y. Yu, X. Lin, C. Sun // 2016 IEEE International Conference. – M., – 2016. – C. 281 – 285.
6. Адаптация фильтра Калмана для использования с локальной и глобальной системами навигации / А.Н. Забегаев, В.Е. Павловский // XII национальная конференция по искусственному интеллекту с международным участием. – М., – 2010. – Физматлит. – С. 399-404.
7. Zunino, G. Simultaneous Localization and Mapping for Navigation in Realistic Environments : учеб. пособие / G. Zunino – M. : KTH, 2002. – 187 с.
8. Riisgaard, S. SLAM for Dummies : учеб. пособие / S. Riisgaard, M.R. Blas. – M. : 2002. – 127 p.
9. Mobile Robot: SLAM Implementation for Unknown Indoor Environment Exploration, / M. Emharraf, M. Bourhaleb, M. Saber, M. Rahmoun // Journal of Computer Science. – 2016. – № 2. – С. 106 –112.

10. FastSLAM: A factored solution to the simultaneous localization and mapping problem with unknown data association / M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit. // Proceedings of the AAAI National Conference on Artificial Intelligence. – M., – 2002. – C. 593-598.
11. FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges / M. Montemerlo, S. Thrun //
12. Simultaneous localization and mapping with unknown data association using FastSLAM / M. Montemerlo, S. Thrun. // In Robotics and Automation (ICRA), 2003 IEEE International Conference on. IEEE. – M., – 2003. – C. 234 – 238.
13. DP-SLAM: Fast, Robust Simultaneous Localization and Mapping Without Predetermined Landmark / A. Eliazar, R. Parr // Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence. – M., – 2003. – C. 345–353.
14. DP-SLAM 2.0 / A. Eliazar, R. Parr // 2004 IEEE International Conference. – M., – 2004. – C. 1314–1320.
15. A Flexible and Scalable SLAM System with Full 3D Motion Estimation / S. Kohlbrecher, O. von Stryk, J. Meyer, U. Klingauf. // Safety, Security, and RescueRobotics (SSRR), 2011 IEEE International Symposium. – M., – 2011. – P. 155-160.
16. 2D SLAM Quality Evaluation Methods / A. Filatov, A. Filatov, K. Krinkin // Proceeding of the 21st conference of fruct association. –2017. – C. 120-125.
17. Построение карты мобильным роботом, оснащенным лазерным дальномером, методом рекуррентной фильтрации / С.Л. Зенкевич, А.А. Минин // Мехатроника, автоматизация, управление. – М., – 2007. – № 8. – С. 5-12.
18. A comparison of slam algorithms based on a graph of relations. / W. Burgard, C. Stachniss, G. Grisetti, B. Steder, R. Kmmerle, C. Dornhege, M. Ruhnke, A. Kleiner, and J. D. Tards // 2009 IEEE International Conference on Intelligent Robots and Systems. – M., – 2009. – C. 2089–2095.

19. Tinyslam improvements for indoor navigation. In Multisensor Fusion and Integration for Intelligent Systems (MFI) / A. Huletski, D. Kartashov, K. Krinkin // 2016 IEEE International Conference. – M., – 2016. – C. 493–498.
20. Comparison of methods to efficient graph slam under general optimization framework. In Automation (YAC) / H. Li, Q. Zhang, and D. Zhao // 2017 32nd Youth Academic Annual Conference of Chinese Association. –2017. – C. 321–326.
21. An object-based semantic world model for long-term change detection and semantic querying. In Intelligent Robots and Systems (IROS) / J. Mason and B. Marthi // 2012 IEEE/RSJ International Conference on. – M., – 2012. – C. 3851–3858.
22. An evaluation of 2d slam techniques available in robot operating system. / J. M. Santos, D. Portugal, and R. P. Rocha // In 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR). – M., – 2013. – C. 1-6.
23. Real-Time Loop Closure in 2D LIDAR SLAM, in Robotics and Automation (ICRA) / W. Hess, D. Kohler, H. Rapp, D. Andor // 2016 IEEE International Conference. – M., – 2016. – C. 1271–1278.
24. A Visual Landmark Recognition System for Topological Navigation of Mobile Robot / M. Mata, J.M. Armingol, A. Escalera, and M.A. Salics // International Conference on Robotics and Automation. – M., – 2001. –C. 1124-1129.
25. Natural Corners Extraction Algorithm in 2D Unknown Indoor Environment with Laser Sensor / R.J. Yan, J. Wu, W.J. Wang // International Conference on Control, Automation and Systems. – M., – 2012. – C. 983-987.
26. Natural Corners-based SLAM in Unknown Indoor Environment / R.J. Yan, J. Wu, S.J. Lim, J.Y. Lee, and C.S. Han // International Conference on Ubiquitous Robots and Ambient Intelligence. – M., – 2012. – C. 259-261.
27. Алгоритм локальной навигации и картографии для бортовой системы управления автоматического мобильного робота / Р.В. Кучерский, С.В. Манько // In 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR). – M., –2013 – C. 13-22.

28. Development and Simulation on V-REP / Manuel F. Silva // 2014 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC). – M., – 2014. – C. 315-320.
29. Лутц, М. Изучаем Python: учебник / М. Лутц – 4-е изд. – М. : Символ–Плюс, 2011. – 1280 с.

СОДЕРЖАНИЕ

АННОТАЦИЯ ... 3
ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ ... 5
ВВЕДЕНИЕ ... 6
1 ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ ... 8
1.1 Актуальность и новизна ... 8
1.2 Классификация алгоритмов SLAM ... 9
1.2.1 Фильтр Калмана ... 9
1.2.2 Расширенный фильтр Калмана ... 14
1.2.3 Фильтр частиц ... 16
1.2.4 Методы основанные на графах ... 24
1.3 Выводы по первой главе ... 26
2 ОПИСАНИЕ РАЗРАБОТАННОГО АЛГОРИТМА SLAM ... 27
2.1 Модель объекта управления ... 27
2.2 Датчики робота ... 29
2.3 Карта пространства ... 30
2.4 Фильтр частиц ... 32
2.4.1 Движение ... 35
2.4.2 Измерение ... 35
2.4.3 Отсев ... 36
2.5 Выводы по второй главе ... 36
3 МОДЕЛИРОВАНИЕ РАБОТЫ АЛГОРИТМА ... 38
3.1 Требования к моделированию ... 38
3.1 Результаты моделирования ... 40
3.2 Выводы по третей главе ... 44
ЗАКЛЮЧЕНИЕ ... 45
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ... 46
ПРИЛОЖЕНИЕ А ... 51

ПРИЛОЖЕНИЕ А

Исходный код программы

В данном приложении приведен текст программы «SLAM», предназначенной для построения карты местности для мобильного робота с использование ультразвуковых и инфракрасных датчиков.

Текст программы реализован в виде символической записи в формате py-code в среде программирования PyCharm. Основной задачей программы является построение карты местности для мобильного робота, а также связь среды PyCharm с робототехническим симулятором V-REP для проведения моделирования работы алгоритма. Для этого используется интерфейс программирование удаленного приложения Remote API для Python3.

Исходный код программы состоит из 6 классов. Главным классом программы является класс SLAM. Краткое описание составных частей программы представлено ниже:

а) Класс SLAM – основной класс программы, в нём реализована основная логика программы;

б) Класс Mapping – этот класс отвечает за отрисовку определяемой карты пространства;

в) Класс ParticleFilter – класс, отвечающий за реализацию фильтра частиц;

г) Класс Particle -класс, реализующий частицу;

д) Класс Odometry – в данном классе реализуется расчет координат робота на основе изменения положения колес робота;

е) Класс Pioneer – класс, реализующий связь программы с роботом в среде V-REP.

Ниже приведен листинг всех классов программы:

Листинг класса SLAM

```
import Odometry
import Mapping
import Pioneer
import ParticleFilter
import math
import numpy as np
class SLAM:
    # Инициализация объекта класса
    def __init__(self, numParticle=15, width=600, height=600, coordinatesRobotX=0, coordinatesRobotY=0,
                 orientationRobot=0):
        # Параметры карты
        self.widthMap = width                                                   # Ширина карты
        self.heightMap = height                                                 # Высота карты
        self.Map = np.zeros(self.widthMap * self.heightMap).reshape(self.widthMap,
                                                self.heightMap)                 # инициализация Массива (Нулями)

        # Количество частицы
        self.numParticle = numParticle                                          # Кол.частиц
        # Инициализация объектов
        self.odometer = Odometry.Odometry(0.095, 0.1655, 0, 0, 0)               # Инициализация Одометра
        self.mapping = Mapping.Mapping(self.widthMap, self.heightMap)           # Инициализация Карты
        self.robot = Pioneer.Pioneer()                                          # Инициализация Робота
        self.particleFilter = ParticleFilter.ParticleFilter()                   # Инициализация Фильтра
        # Координаты робота
        self.coordinatesRobotX = coordinatesRobotX                              # Начальная координата X
        self.coordinatesRobotY = coordinatesRobotY                              # Начальная координата Y
        self.orientationRobot = orientationRobot                                # Начальный угол робота
        # Массив препятсвий
        self.massObstacle = []                                                  # Массив препятствий
        self.massCoordinate = []                                                # Массив координат
        # Создание файла для записи
        self.fileResults = open('Results.csv', 'w')                             # Создание файла
        self.fileResults.close()                                                # Закрытие файла
        # Счетчик
        self.j = 0

    # Обновление карты
    def update(self, N):
        # Считывание данных с робота (Положение колес, координаты и ореинтация робота и т.д.)
        coorX, coorY, orientation, distance1, sensorOrientation1, distance2, sensorOrientation2, orientation1,
orientation2 = self.robot.motion()
        coorRobotX, coorRobotY, orientationRobot = self.odometer.calculationOdometers(orientation1,
orientation2)
        orientationRobot = orientationRobot % (2 * math.pi)
        orientation = orientation % (2 * math.pi)
        # Обновление показаний датчика
        if self.j == 5:
            self.j = 0
            self.robot.stop()                                                   # Остановка робота
            self.scanning(N)                                                    # Сканирование карты

            self.coordinatesRobotX, self.coordinatesRobotY, self.orientationRobot =
self.particleFilter.localization(
                coorRobotX,
                coorRobotY,
                orientation,
                self.coordinatesRobotX,
                self.coordinatesRobotY,
                self.orientationRobot,
                self.massCoordinate)
```

```
            self.coordinatesRobotX = self.rsch(coorX, self.coordinatesRobotX)
            self.coordinatesRobotY = self.rsch(coorY, self.coordinatesRobotY)
            self.mapping.rendering(coorX, coorY, orientation, self.coordinatesRobotX,
                                   self.coordinatesRobotY, self.orientationRobot, self.massObstacle)

            self.odometer.updateOfCoordinates(self.coordinatesRobotX, self.coordinatesRobotY,
self.orientationRobot)

            # Работа с файлом
            self.fileResults = open('Results.csv', 'a')  # открытие файла для записи результатов
            self.fileResults.write(str(coorX) + ";" + str(coorY) + ";" + str(orientation) + ";;" + str(
                self.coordinatesRobotX) + ";" + str(self.coordinatesRobotY) + ";" + str(self.orientationRobot) +
"\n")
            self.fileResults.close()
        self.j = self.j + 1
    # Сканирование пространства
    def scanning(self, N):
        self.massObstacle = []                                          # Обнуление массива
        self.massCoordinate = []
        for _ in range(N):
            dist1, orien1, dist2, orien2 = self.robot.scanning()
            if dist1 != None:
                self.massObstacle.append(True)
                self.massObstacle.append(dist1)
                self.massObstacle.append(orien1)

                a, b = self.calculation(True, dist1, orien1)
                self.massCoordinate.append(0)
                self.massCoordinate.append(0)
                self.massCoordinate.append(a)
                self.massCoordinate.append(b)

            if dist2 != None:
                self.massObstacle.append(False)
                self.massObstacle.append(dist2)
                self.massObstacle.append(orien2)

                a, b = self.calculation(False, dist2, orien2)
                self.massCoordinate.append(0)
                self.massCoordinate.append(0)
                self.massCoordinate.append(a)
                self.massCoordinate.append(b)

        # Отрисовка в начальный момент
        if self.triger == True:
            self.triger = False
            self.mapping.rendering(0, 0, 0, self.massCoordinate)

    # Расчет данных
    def calculation(self, bol, dist, orien):
        if bol:
            return self.coordinatesRobotX + 9.9 * math.cos(
                math.pi / 4 + self.orientationRobot) + dist * math.cos(
                self.orientationRobot + orien), self.coordinatesRobotY + 9.9 * math.sin(
                -math.pi / 4 - self.orientationRobot) + dist * math.sin(
                -self.orientationRobot - orien)
        else:
            return self.coordinatesRobotX + 9.9 * math.cos(
                -math.pi / 4 + self.orientationRobot) + dist * math.cos(
                self.orientationRobot - orien), self.coordinatesRobotY - 9.9 * math.sin(
                -math.pi / 4 + self.orientationRobot) - dist * math.sin(
                self.orientationRobot - orien)
```

Листинг класса Pioneer:

```
import math
import vrep
import sys
import Mapping
import Odometry

class Pioneer:

    def __init__(self, ID=True):
        # Задание основных параметров и переменных роботов
        self.ID = ID
        self.noDetectionDist = 0.25
        self.maxDetectionDist = 0.1
        vrep.simxFinish(-1)
        self.clientID = vrep.simxStart('127.0.0.1', 19999, True, True, 5000, 5)
        self.pioneerUltrasonicSensor = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        self.usensors = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        self.braitenbergL = [-0.2, -0.4, -0.6, -0.8, -1, -1.2, -1.4, -1.6, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        self.braitenbergR = [-1.6, -1.4, -1.2, -1, -0.8, -0.6, -0.4, -0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        self.v0 = 2
        self.orientationSensor1 = 0
        self.orientationSensor2 = 0
        self.leftWheelOrientation = 0
        self.rightWheelOrientation = 0
        self.angleOfRotation = 5  # угол поворота датчика
        self.angleValue = 0  # текущий угол датчика
        self.lazerSensor = 0
        self.odometer = Odometry.Odometry(0.095, 0.1655, 0, 0, 0)
        if self.clientID != -1:
            print("Connected to remote server")
        else:
            # self.mapp.plottingPlane()
            print('Connection not successful')
            sys.exit('Could not connect')
        # Задание дискрипторов
        errorCode, self.pioneer_p3dx = vrep.simxGetObjectHandle(self.clientID, 'Pioneer_p3dx',
                                                vrep.simx_opmode_oneshot_wait)
        errorOrientation, self.orientationPieneer = vrep.simxGetObjectOrientation(self.clientID, self.pioneer_p3dx, -1,
                                                      vrep.simx_opmode_streaming)
        errorCode, self.left_motor_handle = vrep.simxGetObjectHandle(self.clientID, 'Pioneer_p3dx_leftMotor',
                                                    vrep.simx_opmode_oneshot_wait)
        errorCode, self.right_motor_handle = vrep.simxGetObjectHandle(self.clientID, 'Pioneer_p3dx_rightMotor',
                                                     vrep.simx_opmode_oneshot_wait)
        errorCode, self.motorSensor1 = vrep.simxGetObjectHandle(self.clientID, 'Pioneer_p3dx_leftMotor0',
                                                vrep.simx_opmode_oneshot_wait)
        errorCode, self.motorSensor2 = vrep.simxGetObjectHandle(self.clientID, 'Pioneer_p3dx_rightMotor0',
                                                vrep.simx_opmode_oneshot_wait)
        for _ in range(16):
            errorCode, self.pioneerUltrasonicSensor[_] = vrep.simxGetObjectHandle(self.clientID,
                                                         'Pioneer_p3dx_ultrasonicSensor' + str(
                                                             _ + 1),
                                                         vrep.simx_opmode_oneshot_wait)
        errorCode, self.Sensor1 = vrep.simxGetObjectHandle(self.clientID, 'Proximity_sensor',
                                           vrep.simx_opmode_oneshot_wait)
        errorCode, self.Sensor2 = vrep.simxGetObjectHandle(self.clientID, 'Proximity_sensor0',
                                           vrep.simx_opmode_oneshot_wait)
        # Ининциализация ультрозвуковых датчиков
        for _ in range(16):
            self.usensors[_] = vrep.simxReadProximitySensor(self.clientID, self.pioneerUltrasonicSensor[_],
                                            vrep.simx_opmode_blocking)
```

```
        self.distance1 = vrep.simxReadProximitySensor(self.clientID, self.Sensor1, vrep.simx_opmode_blocking)
        self.distance2 = vrep.simxReadProximitySensor(self.clientID, self.Sensor2, vrep.simx_opmode_blocking)

        # Вывод ошибки
        if errorCode == -1:
            print('Can not find Pioneer')
            sys.exit()

    def motion(self):
        # Определение позиции робота
        returnCode, self.positionPioneer = vrep.simxGetObjectPosition(self.clientID, self.pioneer_p3dx, -1,
                                                                      vrep.simx_opmode_oneshot_wait)

        self.detect = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        # Задание движения
        self.ultrasonicSensor()
        for _ in range(16):
            if self.usensors[_][0] == 0 and self.usensors[_][1] != False:
                dist = self.distanceRobot(self.usensors[_][2][0], self.usensors[_][2][1])
                if (dist < self.noDetectionDist):
                    if dist < self.maxDetectionDist:
                        dist = self.maxDetectionDist
                    self.detect[_] = 1 - (
                            (dist - self.maxDetectionDist) / (self.noDetectionDist - self.maxDetectionDist))
            else:
                self.detect[_] = 0

        b, self.orientationPieneer = vrep.simxGetObjectOrientation(self.clientID, self.pioneer_p3dx, -1,
vrep.simx_opmode_buffer)
        self.vLeft = self.v0
        self.vRight = self.v0

        for _ in range(16):
            self.vLeft = self.vLeft + self.braitenbergL[_] * self.detect[_]
            self.vRight = self.vRight + self.braitenbergR[_] * self.detect[_]

        if self.angleValue <= -45:
            self.angleOfRotation = 1
        elif self.angleValue >= 135:
            self.angleOfRotation = -1

        self.angleValue = self.angleValue + self.angleOfRotation
        error = vrep.simxSetJointTargetPosition(self.clientID, self.motorSensor1, self.angleValue * math.pi / 180,
                                        vrep.simx_opmode_streaming)
        error = vrep.simxSetJointTargetPosition(self.clientID, self.motorSensor2, -self.angleValue * math.pi / 180,
                                        vrep.simx_opmode_streaming)

        b, self.orientationSensor1 = vrep.simxGetJointPosition(self.clientID, self.motorSensor1,
                                                   vrep.simx_opmode_streaming)
        b, self.orientationSensor2 = vrep.simxGetJointPosition(self.clientID, self.motorSensor1,
                                                   vrep.simx_opmode_streaming)
        error, self.leftWheelOrientation = vrep.simxGetJointPosition(self.clientID, self.left_motor_handle,
vrep.simx_opmode_streaming)
        error, self.rightWheelOrientation = vrep.simxGetJointPosition(self.clientID, self.right_motor_handle,
vrep.simx_opmode_streaming)

        errorCode = vrep.simxSetJointTargetVelocity(self.clientID, self.left_motor_handle, self.vLeft,
vrep.simx_opmode_streaming)
        errorCode = vrep.simxSetJointTargetVelocity(self.clientID, self.right_motor_handle, self.vRight,
vrep.simx_opmode_streaming)

        if error == -1 or returnCode != 0:
            print('Can not find left or right motor')
```

```
            sys.exit()

        self.distance1 = vrep.simxReadProximitySensor(self.clientID, self.Sensor1, vrep.simx_opmode_blocking)
        self.distance2 = vrep.simxReadProximitySensor(self.clientID, self.Sensor2, vrep.simx_opmode_blocking)

        return [self.positionPioneer[0] * 50, self.positionPioneer[1] * 50, self.orientationPieneer[2],
                self.distance(self.distance1[2][2] * 50, self.distance1[2][0] * 50, self.distance1[1]),
                self.orientationSensor1,
                self.distance(self.distance2[2][2] * 50, self.distance2[2][0] * 50, self.distance2[1]),
                self.orientationSensor2,
                self.leftWheelOrientation, self.rightWheelOrientation]

    def ultrasonicSensor(self):
        for _ in range(16):
            self.usensors[_] = vrep.simxReadProximitySensor(self.clientID, self.pioneerUltrasonicSensor[_],
vrep.simx_opmode_streaming)

    def stop(self):
        errorCode = vrep.simxSetJointTargetVelocity(self.clientID, self.left_motor_handle, 0,
                                        vrep.simx_opmode_streaming)
        errorCode = vrep.simxSetJointTargetVelocity(self.clientID, self.right_motor_handle, 0,
                                        vrep.simx_opmode_streaming)

    def scanning(self):
        if self.angleValue <= -45:
            self.angleOfRotation = 1
        elif self.angleValue >= 135:
            self.angleOfRotation = -1
        self.angleValue = self.angleValue + self.angleOfRotation
        error = vrep.simxSetJointTargetPosition(self.clientID, self.motorSensor1, self.angleValue * math.pi / 180,
                                    vrep.simx_opmode_streaming)
        error = vrep.simxSetJointTargetPosition(self.clientID, self.motorSensor2, -self.angleValue * math.pi / 180,
                                    vrep.simx_opmode_streaming)
        b, self.orientationSensor1 = vrep.simxGetJointPosition(self.clientID, self.motorSensor1,
                                               vrep.simx_opmode_streaming)
        b, self.orientationSensor2 = vrep.simxGetJointPosition(self.clientID, self.motorSensor1,
                                               vrep.simx_opmode_streaming)
        self.distance1 = vrep.simxReadProximitySensor(self.clientID, self.Sensor1, vrep.simx_opmode_blocking)
        self.distance2 = vrep.simxReadProximitySensor(self.clientID, self.Sensor2, vrep.simx_opmode_blocking)

        return [self.distance(self.distance1[2][2] * 50, self.distance1[2][0] * 50, self.distance1[1]),
                self.orientationSensor1,
                self.distance(self.distance2[2][2] * 50, self.distance2[2][0] * 50, self.distance2[1]),
                self.orientationSensor2]

    def distance(self, a, b, bo):
        if bo == True:
            c = (a ** 2 + b ** 2) ** 0.5
            if c > 1000:
                return None
            elif c <= 30:
                return None
            else:
                return c
        else:
            return None

    def distanceRobot(self, a, b):
        c = (a ** 2 + b ** 2) ** 0.5
        if c >= 1000:
            return 0
        elif c <= 30:
            return 0
```

```
        else:
            return c
```

Листинг класса ParticleFilter

```
import Particle

from math import *
import numpy as np
from PIL import Image, ImageDraw, ImageTk

# Class ParticleFilter
class ParticleFilter:
    # Инизиализация
    def __init__(self, numParticles=50, x=0, y=0, phi=0, X = 600, Y = 600):

        self.particlesObj = Particle.Particle(numParticles, 0, 0, 0)

        self.img = np.zeros((X, Y, 4), np.uint8)
        self.pilImage = Image.fromarray(self.img, 'RGBA')

        self.sizeMap = X
        self.sizeCell = 1

        self.Grid = [[0.0 for row in range(int(self.sizeMap / self.sizeCell))] for col in range(int(self.sizeMap /
self.sizeCell))]

    # Локализация
    def localization(self, x0, y0, phi0, x1, y1, phi1, sensorReadings):
        delx = -(x1 - x0)
        dely = -(y1 - y0)
        delphi = -phi1 + phi0
        delphi = self.boundAngle(delphi)

        deltaS = pow(delx * delx + dely * dely, 0.5)
        alpha = 0.0
        if (dely == delx == delphi == 0.0):
            alpha = 0.0
        else:
            alpha = -atan2(dely, delx) + phi0

        self.particlesObj.updateParticles(deltaS, alpha, delphi)
        for k in range(self.particlesObj.numParticles):
            weight = self.computeWeight(self.particlesObj, k, sensorReadings)
            self.particlesObj.updateWeights(k, weight)
        self.particlesObj.normalizeWeights()

        [x, y, theta, bestParticle] = self.particlesObj.selectChampion()
        self.particlesObj.resampleParticles()

        for k in range(0, 600):
            for l in range(0, 600):
                belief = self.Grid[l][k]
                red = 0
                blue = 0
                green = 0

                if (belief < - 666666):
                    red = 0
                    green = 0
                    blue = 255
```

```
            if (belief > -666666 and belief < 0):
                red = 255
                blue = 255
                green = 255
            if (belief == 0.0):
                red = 170
                blue = 170
                green = 170
            if (belief > 30.0):
                red = 0
                blue = 0
                green = 0

            self.img[k][l] = [red, blue, green, 255]

        pilImage = Image.fromarray(self.img, 'RGBA')
        pilImage.save('mapping' + '.png')

        return (x, y, theta)

    #
    def computeWeight(self, particlesObj, k, sensorReadings):
        c_x, c_y, theta = particlesObj.X[k], particlesObj.Y[k], particlesObj.Theta[k]

        weight = 0.0

        for k in range(0, len(sensorReadings), 2):

            x1 = float(sensorReadings[k])
            y1 = float(sensorReadings[k+1])
            x0, y0 = 0.0, 0.0
            d = sqrt(pow((x0 - x1), 2) + pow((y0 - y1), 2))
            alpha1 = atan2(y1, x1)
            G_x1 = d * cos(theta + alpha1) + c_x
            G_y1 = d * sin(theta + alpha1) + c_y

            x0 = c_x
            y0 = c_y
            d0 = sqrt(pow((x0 - c_x), 2) + pow((y0 - c_y), 2))
            alpha0 = atan2(y0, x0)

            G_x0 = d0 * cos(theta + alpha0) + c_x
            G_y0 = d0 * sin(theta + alpha0) + c_y

            t = atan2((G_y1 - G_y0), (G_x1 - G_x0))
            d = sqrt(pow(G_x1 - G_x0, 2) + pow(G_y1 - G_y0, 2))
            x = int(round(G_x1 / 1))
            y = int(round(G_y1 / 1))
            x += int(self.sizeMap / (self.sizeCell * 6))
            y += int(self.sizeMap / (self.sizeCell * 2))
            ON = 0
            if (x >= 0 and x < int(self.sizeMap / self.sizeCell) and y >= 0 and y < int(self.sizeMap / self.sizeCell)):
                for q in [-1, 0, 1]:
                    for r in [-1, 0, 1]:
                        if (x + r > 0 and x + r < len(self.Grid) and y + q > 0 and y + q < len(self.Grid)):
                            if (self.Grid[x + r][y + q] > 600.0):
                                weight += (12.0)
                                ON = 1
            OUTSIDE = 0
            if (ON == 0):
                for K in range(int(d * 1.0)):
                    k = K / 1.0
                    x = int((G_x0 + ((G_x1 - G_x0) / fabs(G_x1 - G_x0)) * fabs(k * cos(t))) / self.sizeCell)
```

```
                y = int((G_y0 + ((G_y1 - G_y0) / fabs(G_y1 - G_y0)) * fabs(k * sin(t))) / self.sizeCell)
                x += int(self.sizeMap / (self.sizeCell * 6))
                y += int(self.sizeMap / (self.sizeCell * 2))
                if (x >= 0 and x < int(self.sizeMap / self.sizeCell) and y >= 0 and y < int(self.sizeMap / self.sizeCell)
and k > 0):
                    if (self.Grid[x][y] > 600.0):
                        weight += (4.0)
                        OUTSIDE = 1
                        break
            if (OUTSIDE == 0):
                weight += 8.0
        return weight

    def boundAngle(self, phi):
        from math import fmod, pi
        # Bound angle to [-pi,pi]
        if (phi >= 0):
            phi = fmod(phi, 2 * pi)
        else:
            phi = fmod(phi, -2 * pi)
        if (phi > pi):
            phi -= 2 * pi
        if (phi < -pi):
            phi += 2 * pi
        return phi
```

Листинг класса Particle:

```
from random import gauss, random
from math import *

class Particle:
    # Инициализация частицы
    def __init__(self, numParticles=50, x=0, y=0, theta=0):

        self.random = random
        self.X = [x for k in range(numParticles)]                                    # Координаты частиц X
        self.Y = [y for k in range(numParticles)]                                    # Координаты частиц Y
        self.Theta = [theta for k in range(numParticles)]                            # Ориентиры частиц
        self.Weight = [(1.0 / numParticles) for k in range(numParticles)]            # Веса частиц
        self.numParticles = numParticles                                             # Количество частиц

        self.gauss = gauss                                                           # Функция Гаусса
        self.random = random
        self.cos = cos
        self.sin = sin

    # образец
    def resampleParticles(self):
        weightArr = [0 for k in range(self.numParticles)]                            # Создание массива весов
        for k in range(self.numParticles):
            weightArr[k] = sum(self.Weight[:(k + 1)])
        tempX = [0 for k in range(self.numParticles)]
        tempY = [0 for k in range(self.numParticles)]
        tempTheta = [0 for k in range(self.numParticles)]
        tempWeight = [1.0 / self.numParticles for k in range(self.numParticles)]
        for j in range(self.numParticles):
            sample = self.random()
            for k in range(self.numParticles):
                if (sample < weightArr[k]):
                    tempX[j], tempY[j], tempTheta[j], tempWeight[j] = self.X[k], self.Y[k], self.Theta[k], self.Weight[
```

```
                k]
        self.X = tempX                                  # Координата X
        self.Y = tempY                                  # Координата Y
        self.Theta = tempTheta                          # Ориентация
        self.Weight = tempWeight                        # Обновление весов
        self.normalizeWeights()                          # Нормализация весов

    # Обновление частиц
    def updateParticles(self, deltaS, alpha, deltaPhi):
        k1 = 0.03
        k2 = 0.3
        for k in range(self.numParticles):
            g = self.gauss(0, 1.0)
            e1 = g * k1 * alpha
            e2 = g * k2 * deltaS
            e3 = g * k1 * (deltaPhi - alpha)
            self.X[k] += ((deltaS + e2) * self.cos(self.Theta[k] + alpha + e1))
            self.Y[k] += ((deltaS + e2) * self.sin(self.Theta[k] + alpha + e1))
            self.Theta[k] += (deltaPhi + e1 + e3)
            self.Theta[k] = self.boundAngle(self.Theta[k])
    # Связанный угол
    def boundAngle(self, phi):
        from math import fmod, pi
        # Bound angle to [-pi,pi]
        if (phi >= 0):
            phi = fmod(phi, 2 * pi)
        else:
            phi = fmod(phi, -2 * pi)
        if (phi > pi):
            phi -= 2 * pi
        if (phi < -pi):
            phi += 2 * pi
        return phi

    # Обновление весов
    def updateWeights(self, particle, weight):
        self.Weight[particle] = weight

    # Нормализация весов
    def normalizeWeights(self):
        sumOfWeights = sum(self.Weight)
        for k in range(self.numParticles):
            self.Weight[k] /= sumOfWeights

    # Выборка Чемпиона
    def selectChampion(self):
        champion = self.Weight.index(max(self.Weight))
        return [self.X[champion], self.Y[champion], self.Theta[champion], champion]
```

Листинг класса Mapping:

```
import math
import numpy as np
import pygame
from PIL import Image


class Mapping:
    # Инициализация Карты
    def __init__(self, width=600, height=600):

        # Параметры карты
```

```
self.widthMap = width                                   # Ширина Карты
self.heightMap = height                                 # Высота Карты
self.size = (width, height)                             # Параметры Карты

self.Map = np.zeros(self.widthMap * self.heightMap).reshape(self.widthMap,
                                                        self.heightMap)     # инициализация Массива (Нулями)
self.img = np.zeros((self.widthMap, self.heightMap, 4), np.uint8)
self.white = 255, 255, 255
self.dlinX = int(width / 2)
self.dlinY = int(height / 2)
# Определяемое положение робота
self.startPositionX = int(width / 2)                    # Начальное положение X
self.startPositionY = int(height / 2)                   # Начальное положение Y
self.robotOrientation = 0                               # Начальное направение
self.realPositionX = int(width / 2)                     # Координата робота X (реальная)
self.realPositionY = int(height / 2)                    # Координата робота Y (реальная)
self.robotRealOrientation = 0                           # Ориентир робота (реальный)
self.odometerPositionX = int(width / 2)                 # Координата робота X (расчитанная)
self.odometerPositionY = int(height / 2)                # Координата робота Y (расчитанная)
self.robotOdometerOrientation = 0                       # Ориентир робота (расчитанный)

# Инициализация дисплея в PyGame
pygame.init()
pygame.display.set_caption("Результаты моделирования")
self.screen = pygame.display.set_mode(self.size)

self.image = pygame.image.load("mapp.png")
self.screen.fill(self.white)

# ----------------------------------------------------------------
# Отрисовка препятствий
pygame.draw.line(self.screen, (0, 255, 0), (50, 50), (550, 50))
pygame.draw.line(self.screen, (0, 255, 0), (50, 50), (50, 550))
pygame.draw.line(self.screen, (0, 255, 0), (50, 550), (550, 550))
pygame.draw.line(self.screen, (0, 255, 0), (550, 50), (550, 550))

pygame.draw.line(self.screen, (0, 255, 0), (350, 50), (350, 250))
pygame.draw.line(self.screen, (0, 255, 0), (350, 250), (550, 250))

pygame.draw.line(self.screen, (0, 255, 0), (250, 550), (250, 450))

pygame.draw.line(self.screen, (0, 255, 0), (100, 150), (100, 550))

pygame.draw.line(self.screen, (0, 255, 0), (50, 150), (150, 150))
pygame.draw.line(self.screen, (0, 255, 0), (150, 50), (150, 150))

pygame.draw.line(self.screen, (0, 255, 0), (480, 550), (550, 480))
pygame.image.save(self.screen, "mapp.png")

# Инициализация массивов
self.positionsRobot= []                                 # Позиция робота
# кодировка цветов на карте
self.white = [255, 255, 255, 255]                       # Кодировка отсутствие препятствия
self.black = [0, 0, 0, 255]                             # Кодировка препятсвий
self.red = [255, 0, 0, 255]                             # Кодировка траектории по SLAM
self.green = [0, 255, 0, 255]                           # Кодировка реальных объектов
self.blue = [0, 0, 255, 255]                            # Кодировка реальной траектории
self.x = int(width / 2)                                 #
self.y = int(height / 2)
self.x1 = int(width / 2)
self.y1 = int(height / 2)
self.pdatchik = 19.8/2
```

```
    # Обновление карты
    def update(self, coorRX, coorRY, orienR, coorOX, coorOY, orienO):

        # Положение робота (реальное)
        self.realPositionX = coorRX + self.dlinX                    # реальныое значение X
        self.realPositionY = -coorRY + self.dlinY                   # реальные значения Y
        self.robotRealOrientation = orienR                          # направление движения робота

        # Положение робота (по одометру (или одометр + SLAM) )
        self.odometerPositionX = coorOX + self.dlinX                 # Координата робота X (рассчитанная)
        self.odometerPositionY = -coorOY + self.dlinY                # Координата робота Y (рассчитанная)
        self.robotOdometerOrientation = orienO % (math.pi * 2)       # Ориентация робота (рассчитанная)

    # Сканирование пространства
    def scanning(self, distanse1, sensorOrientation1, distanse2, sensorOrientation2):
        # Показание первого датчика
        self.distanse1 = distanse1
        self.sensorOrientation1 = sensorOrientation1
        # Показания второго датчика
        self.distanse2 =distanse2
        self.sensorOrientation2 = sensorOrientation2
    # Отрисовка
    def rendering(self, x, y, phi, x1, y1, phi1, mass):
        massObstracle =[]
        x = x + 300
        y = -y + 300
        x1 = x1 + 300
        y1 = -y1 + 300
        self.image = pygame.image.load("mapp.png")
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
        self.screen.blit(self.image, (600, 600))
        pygame.draw.line(self.screen, (0, 0, 255), (self.x, self.y), (x, y))
        pygame.draw.line(self.screen, (255, 0, 0), (self.x1, self.y1), (x1, y1))
        for i in range(0, len(mass), 3):
            if mass[i] == True:
                pygame.draw.rect(self.screen, (0, 0, 0), (int(x1 + self.pdatchik * math.cos(math.pi / 4 + phi1) + mass[i +
1] * math.cos(
                    phi1 + mass[i + 2])), int(y1 + self.pdatchik * math.sin(-math.pi / 4 - phi1) + mass[i + 1] * math.sin(
                                -phi1 - mass[i + 2])), 1, 1))
            if mass[i] == False:
                pygame.draw.rect(self.screen, (0, 0, 0), (int(x1 + self.pdatchik * math.cos(-math.pi / 4 + phi1) + mass[i +
1] * math.cos(
                    phi1 - mass[i + 2])), int(
                                y1 - self.pdatchik * math.sin(-math.pi / 4 + phi1) - mass[i + 1] * math.sin(
                                    phi1 - mass[i + 2])), 1, 1))

        self.x = x
        self.y = y
        self.x1 = x1
        self.y1 = y1
        pygame.image.save(self.screen, "mapp.png")
        pygame.display.flip()
```

Листинг класса Odometry

```
import math

class Odometry:
    """ Класс отвечающий за реализацию расчета координат на основе одометров """
```

```
    def __init__(self, r=0.095,l=0.1655, startOdometerLeft=0, startOdometerRight=0, starOrientationRobotsPhi=0):

        self.radius = r                                         # радиус колес
        self.length = l                                         # длина половины расстояния между колесами
        self.diameter = r + r                                   # диаметр колеса робота
        self.width = l + l                                      # ширина робота

        self.coordinatesRobotX = 0                              # обновленная координата робота X
        self.coordinatesRobotY = 0                              # обновленная координата робота Y
        self.orientationRobotPhi = 0                            # обновленный угол

        self.oldOdometerLeft = startOdometerLeft                # начальные показания левого одометра
        self.oldOdometerRight = startOdometerRight              # начальные показания правого одометра

    # Определение координат робота на основе данных с одометров
    def calculationOdometers(self, odometerLeft, odometerRight):
        # расчет левого колеса
        if ((0 <= odometerLeft) and (0 <= self.oldOdometerLeft)) or ((0 >= odometerLeft) and (0 >=
self.oldOdometerLeft)):
            self.odometerLeft = odometerLeft - self.oldOdometerLeft
        elif (math.pi/2 <= odometerLeft) and (-math.pi/2 >= self.oldOdometerLeft):
            self.odometerLeft = -(math.pi - odometerLeft) + (math.pi + self.oldOdometerLeft)
        elif (-math.pi/2 >= odometerLeft) and (math.pi/2 <= self.oldOdometerLeft):
            self.odometerLeft = (math.pi+odometerLeft)+(math.pi-self.oldOdometerLeft)
        else:
            self.odometerLeft = odometerLeft - self.oldOdometerLeft
        # расчет правого колеса
        if ((0 <= odometerRight) and (0 <= self.oldOdometerRight)) or ((0 >= odometerRight) and (0 >=
self.oldOdometerRight)):
            self.odometerRight = odometerRight - self.oldOdometerRight
        elif (math.pi/2 <= odometerRight) and (-math.pi/2 >= self.oldOdometerRight):
            self.odometerRight = -(math.pi - odometerRight) + (math.pi + self.oldOdometerRight)
        elif (-math.pi/2 >= odometerRight) and (math.pi/2 <= self.oldOdometerRight):
            self.odometerRight = (math.pi+odometerRight)+(math.pi-self.oldOdometerRight)
        else:
            self.odometerRight = odometerRight - self.oldOdometerRight

        Dr = math.pi * self.diameter * self.odometerRight/(math.pi*2)     # пройденный путь правого колеса
        Dl = math.pi * self.diameter * self.odometerLeft/(math.pi*2)      # пройденный путь левого колеса
        D = (Dr+Dl)/2                                                     # прйденный путь

        self.coordinatesRobotX = self.coordinatesRobotX + D * math.cos(self.orientationRobotPhi)    # обновленная
координата робота X
        self.coordinatesRobotY = self.coordinatesRobotY + D * math.sin(self.orientationRobotPhi)    # обновленная
координата робота Y
        self.orientationRobotPhi = self.orientationRobotPhi + (Dr - Dl)/self.width                  # обновленная
ориентация робота phi

        self.oldOdometerLeft = odometerLeft
        self.oldOdometerRight = odometerRight

        coordinatesRobot = [self.coordinatesRobotX*50, self.coordinatesRobotY*50, self.orientationRobotPhi]

        return coordinatesRobot                                 # Возврат изменения координат и ориентации

    def updateOfCoordinates(self, X, Y, Phi):
        self.coordinatesRobotX = X / 50
        self.coordinatesRobotY = Y / 50
        self.orientationRobotPhi = Phi
```

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Дальневосточный федеральный университет»

Инженерная школа
Кафедра автоматизации и управления

ОТЗЫВ РУКОВОДИТЕЛЯ

На выпускную квалификационную работу студента(ки) Степаненко Владислава Юрьевича
(фамилия, имя, отчество)

Направление подготовки Мехатроника и робототехника

группа Б3421

Руководитель ВКР д.т.н. (ученая степень, ученое звание) Юхимец Д.А. (ФИО)

На тему Разработка алгоритма построения карты для мобильного робота на основе данных, поступающих от инфракрасных и ультразвуковых датчиков

Дата защиты ВКР «04» июля 2018 г.

Выпускная квалификационная работа Степаненко В.Ю. посвящена разработке алгоритма навигации мобильного робота на основе данных, поступающих от их одометров и датчиков дистанции. Актуальность представленной в работе задачи обусловлена расширением области использования мобильной робототехники для выполнения различных транспортных и сервисных операций в закрытых помещениях. Основной сложностью для управления мобильными роботами в таких условиях является неточность работы их навигационных систем, построенных на основе инерциальных датчиков. При этом существующие методы построения навигационных систем предполагают использование лидаров, что приводит к значительному увеличению стоимости мобильного робота.
Основной задачей, решаемой в работе, является разработка алгоритма построения карты и определения положения мобильного робота на основе данных, поступающих от дешевых инфракрасных и ультразвуковых датчиков. Решение этой задачи позволит увеличить точность навигационных систем мобильных роботов без существенного увеличения их стоимости.
В работе были рассмотрены различные методы решения SLAM задачи, выделены их преимущества и недостатки и выбран метод на основе фильтра частиц как наиболее

подходящий для решения задачи, определенной в работе. Была разработана модификация выбранного метода для случая использования дешевых ультразвуковых или инфракрасных датчиков. Разработанный алгоритм был реализован на языке программирования Python и его работа была апробирована в среде моделирования V-REP.

В результате, Степаненко В.Ю. самостоятельно была разработана модификация SLAM алгоритма на основе фильтра частиц, обеспечивающая построение карты и определение положения мобильного робота при использовании дешевых ультразвуковых или инфракрасных датчиков, было проведено моделирование работы разработанного алгоритма с использованием среды моделирования V-REP.

В целом ВКР Степаненко В.Ю. выполнена на достаточно высоком уровне. Результаты, полученные в ходе выполнения работы, развивают практические приложения в области разработки систем навигации автономных робототехнических систем.

Оригинальность текста составляет 81% (по данным системы «SafeAssign», bb.dvfu.ru).

Считаю, что выпускная квалификационная работа заслуживает оценки «отлично», а студентка Степаненко В.Ю. – присвоения квалификации «бакалавр».

Руководитель ВКР _____ Юхимец Д.А.
(подпись) (ФИО)

«22» июня 2019 г.