СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяются следующие определения и сокращения.

Программное обеспечение – совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

Планирование маршрута – планирование маршрута относится к процессу поиска оптимального пути между несколькими точками. Планирование маршрута обычно характеризуется как проблема обхода графа, а алгоритмы, такие как A*, D* и RRT, являются обычными вариантами для реализации.

Планирование движения – планирование движения относится к процессу определения движения робота во времени, чтобы следовать определенной траектории.

Фреймворк – программное обеспечение, облегчающее разработку и объединение различных компонентов большого программного проекта.

Сериализация – процесс перевода структуры данных в битовую последовательность.

Десериализация – процесс создания структуры данных из битовой последовательности.

DDS (Data distribution system) – служба распространения данных для систем реального времени является стандартом межмашинного взаимодействия Object Managment Group, целью которого является обеспечение масштабируемых, оперативных, надежных, высокопроизводительных и совместимых обменов данными с применением шаблона «издатель — подписчик»

SLAM (Simultaneous localization and mapping) – одновременная локализация и построение карты.

IMU (Inertial measurement unit) – электронное устройство, которое измеряет и сообщает об удельной силе тела, угловой скорости и иногда ориентации тела, используя комбинацию акселерометров, гироскопов и иногда магнитометров.

GPS (Global positioning system) – система глобального позиционирования.

ROS – Robot Operating System

ROS2 – Robot Operating System 2

ВВЕДЕНИЕ

В современном мире развитие технологий автономных систем занимает одно из ключевых мест в научно-техническом прогрессе. Автономная навигация мобильных платформ представляет собой перспективное направление, которое находит применение в различных областях: от робототехники и логистики до сельского хозяйства.

Создание надежного и эффективного программного обеспечения для обеспечения самостоятельного перемещения таких платформ является актуальной задачей, требующей комплексного подхода к решению вопросов планирования маршрутов, обработки данных с датчиков и адаптации к изменяющимся условиям окружающей среды.

Задача автономной навигации мобильной системы состоит из следующего: система принимает данные с сенсоров и отправляет управляющие команды на шасси. Для её реализации необходимо решить большое количество объёмных задач: оценка текущей позиции, построение карты, построение машрута, получение данных с сенсоров, обработка аварийных ситуаций, и т. д.

На данный момент стандартом индустрии является TODO: фреймворк для разработки роботизированных систем ROS, который включает в себя пакеты для навигации и пакеты для решения задач связанных с навигацией (SLAM, локализация робота). На основе данных TODO: фреймворков разрабатывается ПО для различных нужд робототехники, в том числе и для навигации мобильных платформ.

TODO: Фреймворк предлагет использование DDS (Data Distribution System) в качестве медиатора между модулями системы, который потребляет аппаратные ресурсы, что позволяет экономить ресурсы при осуществлении всех коммуникаций между модулями внутри одного исполняемого процесса.

TODO: Исходя из вышесказанного, целью данной работы является анализ сущест- вующих решений в этой области, а также проектирование и разработка прило- жения для формирования форм опросников.

1 АНАЛИТИЧЕСКИЙ ОБЗОР ПРОГРАММНЫХ ПРОДУКТОВ, ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

1.1 Основные понятия и определения

Навигация мобильных систем представляет собой процесс определения положения устройства в пространстве и его перемещения в соответствии с заранее заданными целями. Эта область охватывает множество технологий и методов, включая системы позиционирования, карты и алгоритмы планирования маршрутов. Мобильные системы могут быть использованы в самых разных сферах — от автономных транспортных средств до мобильных роботов в промышленных и исследовательских приложениях.

Для навигации используются различные сенсоры для сбора информации о своем окружении. Это могут быть камеры, лазерные дальномеры, ультразвуковые датчики, IMU. Собранные данные обрабатываются с помощью специализированных алгоритмов, что позволяет системе точно определять свое положение и вносить изменения в маршрут в реальном времени. Успешная навигация зависит от способности системы адаптироваться к изменениям в окружающей среде, таким как перемещения других объектов, препятствия или изменения в маршруте.

1.2 SLAM

Задача SLAM возникает когда мобильный робот, помещённый в неизвестное месте с неизвестной окружающей средой, одновременно строит строит карту местности и определяет своё местоположение в пределах построенной карты. Информация об окружающей среде полученная с датчиков комбинируется алгоритмами и на её основе создаётся карта местности вместе с текущей оценкой позиции на ней.

Задача SLAM заключается в вычислении оценки метоположения x_t агента и карты окрущающей среды m_t из ряда наблюдений o_t над дискретным временем с шагом дискретизации t. Все перечисленные величины являются вероятностными. Цель задачи состоит в том, чтобы вычислить $P(m_t, x_t|o_{1:t})$. Применение правила Байеса является основой для последовательного обновления апостериорного местоположения, учитывая карту и функции перехода $P(x_t, x_{t-1})$:

$$P(x_t|o_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t|x_t, m_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) P(x_{t-1}|m_t, o_{1:t-1}) \qquad (1.1)$$

Точно так же карта может обновляться последовательно:

$$P(m_t|x_t, o_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t|x_t, m_{t-1}, o_t) P(m_{t-1}, x_t|o_{1:t-1}, m_{t-1}) \qquad (1.2)$$

Метод SLAM можно разделить на несколько этапов.

– система начинает с полной неопределённостью о своей позиции и об окружающей среде;

– с помощью сенсоров система собирает данные об окружающей среде, которые используются для построения карты, используя текущую оценку позиции;

– используя новые полученные данные система улучшая свою оценку позиции.

1.3 Планирование маршрута

Для перемещения в указанную позицию система должна уметь находить маршрут между текущей и целевой позицией. Построение маршрута должно учитывать габариты робота, строить оптимальный маршрут, минимизируя длину маршрута и расстояние от препятствий.

В качестве карты на которой строится маршрут используют сетки заполнения (Occupancy grid). Сетки заполненности представляют рабочую область робота как дискретную сетку, в каждой клетке которой лежит значение вероятности о том что эта клетка заполнена. Пример сетки заполненности представлен на рисунке 1.2.

1.4 Исполнение маршрута

После составления маршрута необходимо сформировать команды управления для шасси для его исполнения. Система должна следовать маршруту корректируя своё движение, избегая динамический препятствий. Для этого используются данные с сенсоров об окружающей среде и препятствиях. В процессе выполнения маршрута система может столкнуться с непредсказуемыми препятствиями, при невозможности продолжать текущий маршрут система должна перестроить изначальный маршрут с учётом возникшего препятствия.

1.5 Анализ существующих программных решений по теме дипломного проектирования

Программные фреймворки играют ключевую роль в разработке программного обеспечения, предоставляя инфраструктуру для создания, тестирования и внедрения, решая типовые задачи и позволяют сфокусироваться на разработке функционала продукта. Однако, в области автономной навигации роботизированных платформ многие разработки остаются закрытыми, что связано со спецификой определённых проектов и их проприетарным характером. Несмотря на это, в индустрии широко используется программное обеспечение с открытым исходным кодом.

В программировании роботов активно используются фреймворки для межпроцесного взаимодействия между отдельными модулями[1)]. Примером таких фреймворков служат ROS и YARP.

Это позволяет разрабатывать ПО с использованием разных языков программирования, осуществлять переиспользование отдельных модулей, анализировать и записывать потоки сообщений, настраивать маршрутизацию сообщений.

ROS является де-факто стандартным фреймворком для программного обеспечения роботизированных систем [1]. Основополагающая статья

”Software engineering research on the Robot Operating System: A systematic mapping study” [2] процитирована более 13 000 раз.

Yet Another Robot Platform (YARP) [3] – это фреймворк который преследует цели, очень схожие с ROS. YARP поддерживает построение системы управления роботом как набор программ общающимся в одноранговой сети используя различные каналы связи, что по своей сути не отличается от целей ros. YARP менее популярен и используется для более специализированных систем и не имеет отличительных преимуществ, поэтому далее его не рассматриваем.

ROS это распределённый фреймворк из процессов (также известных как *ноды*), который позволяет разрабатывать исполняемые файлы индивидуально, и свободно сочетать их во время исполнения. Эти процессы могут быть объединены в *пакеты* и *стэки*, которыми можно легко делится и распространять. ROS поддерживает единую систему кодовых *репозиториев* которые позволяют сотрудничеству быть распределённым.

Философские цели ROS можно кратко сформулировать следующим образом [2]:

[1)]Под модулями подразумеваются отдельные программы, являющиеся компонентами системы, исполняющиеся в отдельных процессах операционной системы, или даже на отдельных компьютерах.

– P2P;
– Основанный на инструментах;
– Многоязычный;
– Тонкий;
– Свободный и открытый исходный код.

На данный момент существует две версии ROS: ROS 1 и ROS2. Первый официальный релиз ROS (под кодовым названием ROS Box Turtler) состоялся 2 марта 2010 года. Первый официальный релиз ROS2 состоялся 8 декабря 2017 года. ROS2 это более расширенная версия ROS, спроектированная чтобы устранить недостатки ROS 1, такие как: масштабируемость, производительность и кросс-платформенная совместимость, используя Data Distribution Service (DDS) для общения и вводя новые понятия, такие как жизненный цикл ноды и качество обслуживания (QoS). Далее в дипломной записке при упоминании ROS идёт речь о ROS2.

В экосистеме ROS есть готовый фреймворк для навигации – Nav2 [4]. Nav2 - это профессионально поддерживаемый преемник навигационного стека ROS, в котором используются те же технологии, что и в автономных транспортных средствах, уменьшенные, оптимизированные и переработанные для мобильной и наземной робототехники. Этот проект позволяет мобильным роботам перемещаться по сложным средам для выполнения заданных пользователем прикладных задач практически с любым классом кинематики робота. Он может не только перемещаться из точки А в точку Б, но и принимать промежуточные позы, а также выполнять другие типы задач, такие как следование за объектом, навигация по всему покрытию и т. д. Nav2 - это высококачественный навигационный фреймворк промышленного уровня, которому доверяют более 100 компаний по всему миру.

В Nav2 есть инструменты:

– загрузки, обслуживания и хранения карт;
– локализации робота по предоставленной карте (SLAM предоставляет начальную карту);
– планирования полного пути через окружающую среду;
– управления роботом, чтобы он следовал по маршруту и динамически корректировался, чтобы избежать столкновений;
– сглаживания маршрутов, чтобы сделать их более непрерывными, плавными и/или выполнимыми.
– преобразование данных датчиков в модель окружающего мира;
– построение сложных и настраиваемых моделей поведения роботов с помощью деревьев поведения;
– выполнение заранее определенных действий в случае сбоя, вмеша-

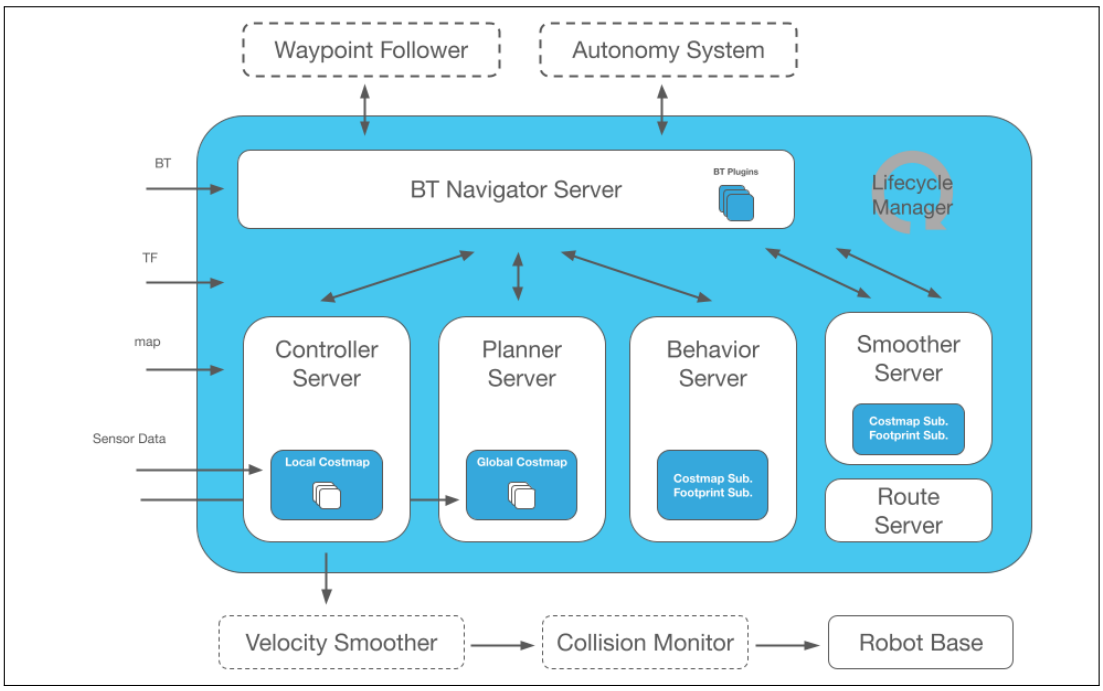

Рисунок 1.1 – Архитектура стэка Nav2

тельства человека или других ситуаций;

– выполнение последовательных маршрутных точек, составляющих миссию;

– управление жизненным циклом программы и сторожевым таймером для серверов;

– простые динамически загружаемые модули для создания индивидуальных алгоритмов, поведений и т. д.

– мониторинг необработанных данных датчиков на предмет неминуемого столкновения или опасной ситуации;

1.6 Анализ пакетов решающих задачу навигации, локализации и построения карты

Для навигации мобильной системы необходима карта, для построения которой используют SLAM (Одновременную локализацию и построение карты).

Алгоритмы SLAM можно разделить на две группы: более ранние алгоритмы, использующие подходы, основанные на фильтрах Байеса, и более новые методы, основанные на графах. Значимые реализации на основе фильтров, доступные в виде пакетов ROS: GMapping и HectorSLAM. Cartographer и KartoSLAM являются основными доступными реализациями на основе графов [5].

Рассмотрим пакеты ros, такие как: SLAM Toolbox и GMapping:

– SLAM Toolbox – использует подход оптимизации графов.

– GMapping [6] – использует Rao–Blackwellized Particle Filter (Фильтр частиц с использование теоремы Рао – Блэквелла – Колмогорова)

В SLAM Toolbox есть возможность делать почти всё, что есть в любой другой платной и бесплатной библиотеке SLAM. Это включает в себя:

– обычный точечный 2D SLAM для мобильных роботов (карта, сохранение pgm-файла) с утилитами, такими как сохранение карт;

– продолжение уточнения, перестройки карты или продолжения построения карты сохраненного (сериализованного) графа позиций в любое время;

– пожизненное картирование: загрузите сохраненный граф позиций и продолжайте строить карту, одновременно удаляя лишнюю информацию из новых сканов;

– режим локализации на основе оптимизации, построенный на основе pose-графа. Возможность запуска режима локализации без предварительной карты для режима «лидарной одометрии» с локальным замыканием контуров;

– синхронный и асинхронный режимы отображения;

– объединение кинематических карт (в разработке находится техника объединения манипуляций с эластичным графом);

– оптимизационные решатели на основе плагинов с новым оптимизированным плагином на основе Google Ceres;

– плагин RVIZ для взаимодействия с инструментами;

– инструменты манипулирования графами в RVIZ для манипулирования узлами и связями во время отображения;

– сериализация карт и хранение данных без потерь.

В то время как пакет GMapping предлагает обёртку над алгоритмом, описанным в статье [6], не включая дополнительный функционал который предоставляется SLAM Toolbox, предоставляя лишь возможность настройки параметров алгоритма и получения построенной карты.

1.7 Постановка целей и задач дипломного проектирования

Фреймворки для разработки ПО для робототехники используют сервис для обмена сообщения между модулями, но у этого архитектурного подхода есть ряд недостатков: дополнительные затраты на сериализацию и десериализацию данных, затраты на маршрутизацию сообщений, а также при использовании нескольких программных модулей конечный программный продукт по своей сути является распределённой системой, что вносит следующие недо-

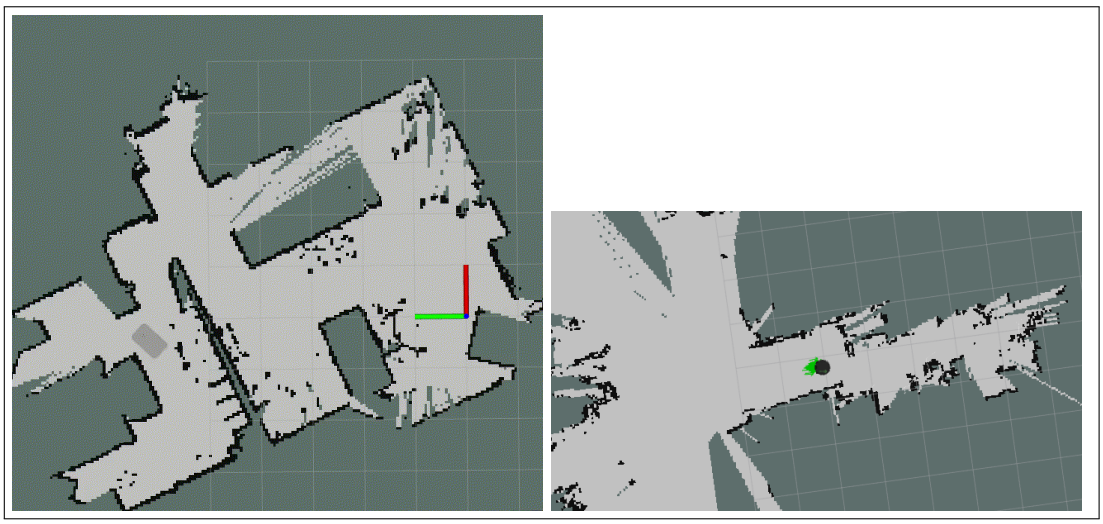

Рисунок 1.2 – Пример построения карты используя SLAM Toolbox (слева) и GMapping (справа).

статки:

– проблемы с синхронизацией состояния, неконсистентность состояния;

– потеря сообщений;

– каскадный отказ системы;

– невозможность использования отладчика подключённого к одному исполняемому файлу для отладки всей системы навигации.

Исходя из этого, целью дипломного проектирования является разработка программного средства осуществив вышеперечисленные оптимизации и устранив вышеперечисленные недостатки, а также реализовать необходимый набор функций, характерный для программных средств в данной предметной области.

Для достижения поставленных целей следует решить следующие задачи:

– определить требования к разрабатываемому программному средству и составление спецификации, включающей их;

– осуществить выбор технологии и языка программирования для реализации программного средства;

– провести проектирование архитектуры программного средства;

– разработка алгоритмов для метода SLAM;

– разработка алгоритмов для оценки местоположения;

– разработка алгоритмов для поиска маршрута;

– разработка алгоритмов для выполнения маршрута;

– программирование и тестирование отдельных программных модулей;

– тестирование готового программного средств.

1.8 Алгоритмы фильтрации

Оценка состояния динамических систем в условиях неопределённости является ключевой задачей в задачах навигации, робототехники и обработки сигналов. Существует большое количество алгоритмов, позволяющие решать задачу оценки состояния с учётом заданных ограничений (например, ограничений по доступным вычислительным мощностям):

- линейный фильтр Калмана (см. пункт 1.8.1);
- расширенный фильтр Калмана (см. пункт 1.8.2);
- нелинейный фильтр Калмана (см. пункт 1.8.3);
- фильтр частиц (см. пункт 1.8.4).

Каждый метод анализируется с точки зрения его математической основы и применимости в задачах навигации.

1.8.1 Фильтр Калмана

Фильтр Калмана – рекурсивный алгоритм, обеспечивающий оптимальную оценку состояния линейной динамической системы с нормально распределёнными шумами. Предположение, что шум системы нормально распределён является ключевым. Динамику системы в момент времени k можно описывается уравнением момент времени:

$$x_k = F_k x_{k-1} + B_k u_k + w_k, \tag{1.3}$$

$$z_k = H_k x_k + v_k. \tag{1.4}$$

где x_k – вектор переменных состояния, z_k – вектор переменных измерений, u_k – управляющие переменные, w_k шум процесса, v_k – шум измерений, F_k – матрица перехода состояния, B_k – матрица управления, H_k – матрица измерений.

Переменные состояния (x_k) описывают характеристики системы на временном шаге k. Переменные состояния полностью определяют её динамическое поведение. Переменные измерения (z_k) представляют наблюдаемые данные, получаемые от датчиков на шаге k. Они описываются моделью измерений (см. уравнение 1.4) Управляющие переменные (u_k) описывают внешние воздействия на систему, влияющие на её динамику. Они входят в уравнение состояния и считаются известными, предоставляемыми системой управления. Например, команды управления для перемещающегося средства.

Матрица перехода состояния F_k описывает эволюцию состояния системы x_k во времени без учёта управления и шума. Матрица управления B_k описывает влияние управляющего сигнала u_k на состояние системы. Она также входит в уравнение состояния и имеет размер $n \times m$, где m – размерность u_k.

В идеальных условиях, шум процесса w_k и шум измерений $_k$ полагаются равны 0. Пользователь может самостоятельно изменять значения шума в зависимости от степени доверия системе или измерениям.

Все переменные, в зависимости от возможности произвести наблюдение за значением, можно разделить на скрытые и явные зависимости. К скрытым переменным относят:

– переменные состояния x_k;
– шум процесса w_k.

К явным переменным относят:

– переменные измерений z_k;
– управляющие переменные u_k;
– шум измерений v_k.

Работа фильтра Калмана основана на последовательном выполнении этапов предсказания (predict) и коррекции (update).

На этапе предсказания выполняется расчёт априорной оценки переменных состояния (см. уравнение 1.5) и априорной оценки ковариация ошибка системы (см. уравнение 1.6).

$$\hat{x}_{k|k-1} = F_k\hat{x}_{k-1|k-1} + B_k u_k, \tag{1.5}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k.. \tag{1.6}$$

Этап коррекции обновляет априорную оценку с учётом измерений, полученных с использованием датчиков. Для этого вычисляются коэффициент усиления Калмана $(K)_k$ (см. уравнение 1.7), апостериорная оценка состояния $x_{k|k}$ (см. уравнение 1.8), и апостериорная ковариация ошибки $P_{k|k}$ (см. уравнение 1.9).

$$K_k = P_{k|k-1}H_k^T(H_k P_{k|k-1} H_k^T + R_k)^{-1}, \tag{1.7}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H_k\hat{x}_{k|k-1}), \tag{1.8}$$

$$P_{k|k} = (E - K_k H_k)P_{k|k-1}. \tag{1.9}$$

Коэффициент K_k балансирует доверие к модели и измерениям, снижая неопределённость.

1.8.2 Расширенный фильтр Калмана (Extended Kalman Filter)

Расширенный фильтр Калмана (EKF) адаптирует фильтр Калмана (KF) для нелинейных систем:

$$x_k = f(x_{k-1}, u_k) + w_k, \tag{1.10}$$

$$z_k = h(x_k) + v_k. \tag{1.11}$$

Линеаризация выполняется с помощью матриц Якоби $F_k = \frac{\partial f}{\partial x}\big|_{\hat{x}_{k-1|k-1}}$ и $H_k = \frac{\partial h}{\partial x}\big|_{\hat{x}_{k|k-1}}$. Этапы предсказания и коррекции аналогичны фильтру Калмана, но ошибки линеаризации снижают точность при сильной нелинейности.

1.8.3 Нелинейный фильтр Калмана (Unscented Kalman Filter)

UKF использует сигма-точки для обработки нелинейностей без линеаризации. Сигма-точки генерируются на основе $\hat{x}_{k-1|k-1}$ и $P_{k-1|k-1}$, затем распространяются через f и h:

$$\hat{x}_{k|k-1} = \sum w_i f(x_i), \tag{1.12}$$

$$P_{k|k-1} = \sum w_i (f(x_i) - \hat{x}_{k|k-1})(f(x_i) - \hat{x}_{k|k-1})^T + Q_k. \tag{1.13}$$

Для состояния $x_{k-1|k-1}$ с оценкой $\hat{x}_{k-1|k-1}$ и ковариацией $P_{k-1|k-1}$ генерируется $2n + 1$ сигма-точек, где n – это размерность вектора состояний x_k. Для генерации точек используется следующий алгоритм:

– вычисление масштабирующего параметра:

$$\lambda = \alpha^2(n + \kappa) - n,$$

где α ($10^{-3} \leq \alpha \leq 1$) контролирует разброс, κ (обычно $3 - n$) – параметр настройки.

– генерация сигма-точек:

$$x_{k-1}^{(0)} = \hat{x}_{k-1|k-1},$$

$$x_{k-1}^{(i)} = \hat{x}_{k-1|k-1} + (\sqrt{(n+\lambda)P_{k-1|k-1}})_i, \quad i = 1, \ldots, n,$$

$$x_{k-1}^{(i)} = \hat{x}_{k-1|k-1} - (\sqrt{(n+\lambda)P_{k-1|k-1}})_{i-n}, \quad i = n+1, \ldots, 2n,$$

где $(\sqrt{(n+\lambda)P_{k-1|k-1}})_i$ – i-й столбец разложения Холецкого.

– назначение весов:

$$w_m^{(0)} = \frac{\lambda}{n+\lambda}, \quad w_m^{(i)} = \frac{1}{2(n+\lambda)}, \quad i = 1, \ldots, 2n,$$

$$w_c^{(0)} = \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta), \quad w_c^{(i)} = \frac{1}{2(n+\lambda)}, \quad i = 1, \ldots, 2n,$$

где $\beta \approx 2$ для нормального распределения.

В общем случае, UKF работает более точно, чем EKF:

– высокая точность при сильной нелинейности, так как сигма-точки лучше аппроксимируют распределение;

– отсутствие необходимости вычислять производные, что упрощает реализацию для сложных функций.

1.8.4 Фильтр частиц (Particle Filter)

Фильтр частиц (PF) представляет распределение состояния множеством частиц $\{x_k^{(i)}, w_k^{(i)}\}$. Этот метод оценки состояния динамической системы, основанный на методе Монте-Карло. PF особенно эффективен для нелинейных систем с шумами, которые распределены не нормально. То есть в тех системах, где фильтр Калмана и его модификации (EKF, UKF) могут быть недостаточно точны.

Система описывается уравнениями:

$$x_k = f(x_{k-1}, u_k, w_k), \tag{1.14}$$

$$z_k = h(x_k, v_k), \tag{1.15}$$

где x_k – переменные состояния, u_k – управляющие переменные, z_k – измерения, w_k – шум процесса, v_k — шум измерения, f, h — нелинейные функции.

Апостериорное распределение аппроксимируется:

$$p(x_k|z_{1:k}) \approx \sum_{i=1}^{N} w_k^{(i)} \delta(x_k - x_k^{(i)}), \tag{1.16}$$

где δ — дельта-функция Дирака, $\sum_{i=1}^{N} w_k^{(i)} = 1$.

Работа PF состоит из трёх этапов: предсказание, обновление весов и пересборку (ресэмплинг).

На этапе предсказания каждая частица обновляется по модели системы (см. уравнение 1.17). Все частицы формируют априорное распределение $\{x_k^{(i)}\}_{i=1}^{N}$.

$$x_k^{(i)} = f(x_{k-1}^{(i)}, u_k, w_k^{(i)}), \quad w_k^{(i)} \sim p(w_k). \tag{1.17}$$

На этапе обновления весов веса частиц обновляются по принципу правдоподобия:

$$w_k^{(i)} \propto w_{k-1}^{(i)} p(z_k | x_k^{(i)}). \tag{1.18}$$

где для нормально распределённого шума $v_k \sim \mathcal{N}(0, R_k)$:

$$p(z_k | x_k^{(i)}) = \frac{1}{\sqrt{(2\pi)^p |R_k|}} \exp\left(-\frac{1}{2} (z_k - h(x_k^{(i)}))^T R_k^{-1} (z_k - h(x_k^{(i)})) \right). \tag{1.19}$$

После веса всех частиц нормируются:

$$w_k^{(i)} = \frac{w_k^{(i)}}{\sum_{j=1}^{N} w_k^{(j)}}. \tag{1.20}$$

На этапе ресэмплинга устраняется вырождение частиц и осуществляется выбор нового набора частиц $\{x_k^{(i)}\}_{i=1}^N$ с вероятностями, пропорциональными $w_k^{(i)}$. После ресэмплинга $w_k^{(i)} = 1/N$ состояние системы определяется как:

$$\hat{x}_k = \sum_{i=1}^{N} w_k^{(i)} x_k^{(i)}. \tag{1.21}$$

1.9 AHRS

Система ориентации и курса (Attitude and Heading Reference System или AHRS) предназначена для оценки ориентации объекта: углов крена (roll), тангажа (pitch) и рысканья (yaw). В задачах навигации, подобных тем, где применяются фильтр частиц (PF) или фильтры Калмана, AHRS играет ключевую роль в определении ориентации. Основные источники данных, для которых используется AHRS:

– гироскопы ($\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$) для измерения угловой скорости;

– акселерометры ($a = [a_x, a_y, a_z]^T$) для оценки крена и тангажа через вектор силы тяжести;

– магнитометры ($m = [m_x, m_y, m_z]^T$) для определения рысканья относительно магнитного севера.

Ориентация представляется в системе кватернионом $q = [q_0, q_1, q_2, q_3]^T$,

что позволяет избежать блокировки кардана (gimbal lock).

Применение AHRS ограничивается рядом внешних условий:

1 Наличие магнитных помех искажают показания магнитометров.

2 Дрейф гироскопов требует внешней коррекции. Использование алгоритмов фильтрации с AHRS позволяет произвести корректировку дрейфа гироскопов.

3 AHRS не моделирует положение и линейную скорость.

1.10 Оценка измерений в AHRS

Для оценки ориентации в AHRS принято использовать один из двух алгоритмов: фильтр Маджвика или фильтр Махони.

1.10.1 Фильтр Маджвика

Фильтр Маджвика использует градиентный спуск для минимизации ошибки гироскопа с коррекцией от акселерометра и магнитометра. Он оценивает кватернион ориентации q_t путём численного интегрирования:

$$\dot{q}_t = \dot{q}_{\omega,t} - \beta \dot{q}_{\epsilon,t}, \tag{1.22}$$

где $\dot{q}_{\omega,t} = \frac{1}{2} q_{t-1} \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega}_t \end{bmatrix}$ – изменения ориентации от гироскопа $\boldsymbol{\omega}_t$; $\dot{q}_{\epsilon,t}$ – численное значение ошибки, вычисленное градиентным спуском из данных акселерометра (a_t) и магнитометра (m_t); β – коэффициент доверия к фильтру ($\beta \in [0.1, 1]$).

Основными преимуществами фильтра Маджвика являются:

– высокая точность вычисления ориентации;

– эффективная компенсация дрейфа гироскопа.

1.10.2 Фильтр Махони

Фильтр Махони основан на нелинейном комплементарном фильтре на группе $SO(3)$. Он минимизирует ошибку между измеренными и эталонными векторами с помощью пропорционально-интегрального (PI) компенсатора. Ориентация обновляется как:

$$\dot{q}_t = \frac{1}{2} q_t \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega}_t + e_t \end{bmatrix}, \tag{1.23}$$

где $e_t = k_P \boldsymbol{\omega}_{\text{err}} + k_I \int \boldsymbol{\omega}_{\text{err}}\, dt$ – коррекция, основанная на ошибке $\boldsymbol{\omega}_{\text{err}}$, вычисленной как векторное произведение измеренных и предсказанных векторов. $k_P \approx 1$, $k_I \approx 0.3$ – PI-компенсатора.

К основным преимуществам фильтра Махони относят:

– быстрая сходимость;

– низкая вычислительная сложность.

Фильтр требует тщательного выбора параметров k_P, k_I. По сравнению с фильтром Маджвика, фильтр Махони хуже оценивает ориентацию в пространстве.

Таким образом, оба фильтра имеют место быть для различных условий применения. Фильтр Маджвика предоставляет большую точность на низкой частоте отправке данных. Фильтр Махони используются на системах с ограниченной вычислительной мощностью.

1.11 Глобальные планировщики

Глобальные планировщики предназначены для построения маршрута от начальной точки до цели в известной или частично известной среде, обычно представленной в виде графа или сетки. В качестве глобальных планировщиков используют:

– алгоритм Дейкстры;

– алгоритм A*;

– алгоритм RRT.

1.11.1 Алгоритм Дейкстры

Алгоритм Дейкстры служит для поиска кратчайшего пути в графе с неотрицательными весами рёбер. Начиная с начальной вершины, алгоритм последовательно обновляет расстояния до всех остальных вершин, выбирая на каждом шаге вершину с минимальным текущим расстоянием. Для выбранной вершины проверяются её соседи, и расстояния до них обновляются, если найден более короткий путь. Процесс продолжается до обработки всех вершин или достижения цели.

Математически алгоритм минимизирует расстояние до вершины v по формуле:

$$d[v] = \min_{u \in V} \{d[u] + w(u, v)\}, \tag{1.24}$$

где $d[v]$ – кратчайшее расстояние от начальной вершины до v, а $w(u, v)$ – вес ребра между вершинами u и v.

Алгоритм гарантирует оптимальность пути, но для больших графов требует значительных вычислений, с временной сложностью $O((V+E) \log V)$ при использовании кучи.

1.11.2 Алгоритм A*

Алгоритм A* улучшает подход Дейкстры, добавляя эвристическую функцию для ускорения поиска. Каждая вершина оценивается по сумме стоимости пути от начальной точки ($g(v)$) и эвристической оценки расстояния до цели ($h(v)$). Выбирается вершина с минимальной суммой $f(v) = g(v) + h(v)$. Эвристика должна быть допустимой, то есть не переоценивать истинное расстояние: $h(v) \leq h^*(v)$, где $h^*(v)$ – реальное расстояние до цели.

Логику алгоритма можно описать формулой:

$$f(v) = g(v) + h(v). \tag{1.25}$$

A* эффективен для планирования в сетках или графах, особенно при использовании точной эвристики для сокращения количество проверяемых вершин.

1.11.3 RRT (Rapidly-exploring Random Tree)

RRT – алгоритм для планирования маршрутов в непрерывных высокомерных пространствах. Алгоритм строит дерево, начиная с начальной конфигурации, путём случайной выборки точек в конфигурационном пространстве. Для каждой случайной точки находится ближайший узел дерева, и к нему добавляется новая конфигурация на расстоянии δ в направлении случайной точки, если она свободна от препятствий. Процесс повторяется до достижения цели или превышения лимита итераций.

Упрощённо работу алгоритма можно описать формулой:

$$q_{\text{new}} = q_{\text{near}} + \delta \cdot \frac{q_{\text{rand}} - q_{\text{near}}}{\|q_{\text{rand}} - q_{\text{near}}\|}. \tag{1.26}$$

RRT вероятностно полный, но не обеспечивает оптимальность пути.

1.12 Локальные планировщики

Локальные планировщики корректируют движение робота в реальном времени, учитывая динамику и локальные препятствия.

1.12.1 DWA (Dynamic Window Approach)

DWA предназначен для выбора оптимальной пары скоростей (линейной и угловой) с учётом кинематических ограничений движущегося объекта. Алгоритм формирует множество допустимых скоростей (динамическое окно), ограниченных текущим состоянием и ускорениями:

$$W_d = \{(v, \omega) \mid v \in [v_{\min}, v_{\max}], \omega \in [\omega_{\min}, \omega_{\max}]\}. \tag{1.27}$$

Для каждой пары скоростей моделируется траектория, оцениваемая по ориентации к цели, расстоянию до препятствий и величине скорости. Выбирается пара скоростей, минимизирующая целевую функцию $G(v, \omega)$:

$$(v^*, \omega^*) = \min_{(v,\omega) \in W_d} G(v, \omega), \tag{1.28}$$

DWA активно используются для дифференциальных роботов. Но основным недостатком алгоритма является проблема застревания в локальных минимумах целевой функции.

1.12.2 TEB (Timed Elastic Band)

TEB оптимизирует траекторию, представляя её как эластичную ленту, которая деформируется для минимизации энергетический потерь на перемещение. Траектория состоит из набора конфигураций (состояний) $\{q_1, q_2, \ldots, q_n\}$, соединённых временными интервалами ΔT_i.

Алгоритм TEB оптимизирует эти конфигурации и интервалы, минимизируя целевую функцию:

$$J(B) = \sum_{i=1}^{n-1} \left(w_1 \|q_{i+1} - q_i\|^2 + w_2 \Delta T_i^2 + w_3 \sum_{\text{obj}} \text{obj}(q_i, \text{p})^{-2} \right), \tag{1.29}$$

где первый член обеспечивает компактность траектории, второй – минимизацию времени, а третий – избегание препятствий.

Конфигурации учитывают кинематические ограничения, такие как максимальная скорость:

$$v_i = \frac{q_{i+1} - q_i}{\Delta T_i}, \quad \|v_i\| \leq v_{\max}. \tag{1.30}$$

По сравнению с DWA, TEB требует большей вычислительной сложностью, но позволяет описать модель перемещения более точно.

2 МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И РАЗРАБОТКА ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ

2.1 Общие сведения и требования к работе программного средства

TODO: ??? Функциональным назначением разрабатываемого программного решения является осуществление задачи навигации и локализации.

Пользователем программного средства выступают разработчики мобильных систем, требующих навигации.

Исходя из определения задачи о навигации, можно заключить что проектируемое программное решение должно реализовывать следующие функции: - сбор данных с датчиков; - отправка данных на шасси; - создание и сохранение карт, с возможностью последующей загрузки; - определение местоположения на карте; - построение маршрута между двумя точками на карте; - исполнение маршрута.

2.2 Формирование требований к проектируемому программному средству

Для успешной реализации системы мобильной навигации необходимо четко определить и описать функциональные требования, которые будут обеспечивать эффективность и точность работы системы. Эти требования являются основой для проектирования и разработки как аппаратной, так и программной части системы. В данном разделе мы рассмотрим ключевые аспекты, которые должны быть учтены при разработке функциональных требований для мобильной навигации, включая работу с картами, выполнение маршрутов и интеграцию различных сенсоров.

Первым и основным требованием является способность системы определять свое местоположение. Это должно включать в себя использование различных сенсоров, таких как GPS, IMU, Lidar, которые обеспечат точную локализацию устройства как в открытых, так и в закрытых помещениях. Для этого система должна использовать алгоритмы, обеспечивающие непрерывную и стабильную локализацию в реальном времени, минимизируя погрешности и ошибки.

При этом наличие всех сенсоров не является необходимостью на работы системы. Каждый сенсор имеет свои преимущества и недостатки, и при наличии необходимого подмножества сенсоров для заданной окружающей среды система должна обеспечить полную функциональность. Например, при

отсутствии GPS должна быть доступна навигация в помещении.

Важным аспектом является способность системы создавать карту окружающей среды на основе данных от сенсоров. Для этого применяется метод SLAM (Simultaneous Localization and Mapping), который позволяет одновременно и локализовать устройство, и строить карту его окружения. Эта карта должна быть динамической и изменяться в зависимости от новых данных, полученных от сенсоров.

Для обеспечения точности навигации система должна эффективно обрабатывать данные с различных сенсоров, таких как камеры, лидары, ультразвуковые датчики, и объединять их в единую модель пространства. Обработка этих данных должна происходить с минимальной задержкой, чтобы система могла адекватно реагировать на изменения в окружающей среде и корректировать маршрут в реальном времени.

На основе карты окружающей среды и информации о текущем местоположении, система должна быть способна планировать оптимальный маршрут до заданной цели. Планирование маршрута должно учитывать не только расстояние, но и такие факторы, как препятствия, зоны с ограничениями, а также предпочтения пользователя (например, избегать оживленных улиц или труднопроходимых территорий).

После того как маршрут спланирован, система должна быть способна проводить устройство по этому маршруту. Для этого требуется реализация алгоритмов, которые будут учитывать динамичные изменения в окружении и корректировать маршрут в случае появления новых препятствий или изменения дорожных условий. Система должна предоставлять пользователю понятные и своевременные подсказки о следующем шаге, а также информацию о текущем статусе маршрута.

Важно, чтобы система могла адаптироваться к изменениям окружающей среды, таким как перемещающиеся объекты или изменения в инфраструктуре. Для этого система должна использовать алгоритмы, способные перераспределять маршрут на лету, минимизируя влияние изменений на навигацию и обеспечивая бесперебойное выполнение маршрута.

2.3 Разработка технических требований к программному средству

Разрабатываемое программное решение должно обеспечивать корректное функционирование при развёртывании на компьютерном модуле BananaPi CM4, или на модуле со следующими техническими характеристиками:

– Оперативная память 4 Гбайт или более;

– Amlogic A311D шести ядерный процессов с четырьмя Arm Cortex-A73 ядрами, двумя Arm Cortex-A53 ядрами, или более быстродействующий процессор

– доступный объём дискового пространства 5 Гбайт.

3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

3.1 Датчики

В качестве поддерживаемых датчиков было выбрано три ключевых:
– 2D Lidar;
– IMU;
– GPS.

Эти датчики обеспечивают систему данными о пространстве, в котором находится робот, его ориентации и глобальном местоположении. 2D Lidar позволяет получать информацию о препятствиях вокруг устройства, IMU предоставляет данные о наклоне и угловых ускорениях, а GPS — о глобальной позиции робота. Все эти данные интегрируются в систему навигации, создавая основу для безопасного и эффективного перемещения устройства в различных условиях.

2D Lidar (Light Detection and Ranging) работает на основе принципа измерения расстояния до объектов с использованием лазерных импульсов. Ли дар излучает лазерные импульсы, которые отражаются от объектов, встречающих их на пути. Время, которое требуется импульсу для прохождения от лидара до объекта и обратно, используется для вычисления расстояния до объекта. Этот процесс повторяется многократно по всей области сканирования, создавая карту расстояний на основе измерений.

Рисунок 3.1 – 2D Lidar

2D лидары обычно работают в плоскости, что означает, что они измеряют расстояния только в одном направлении (по горизонтали или вертикали). Сканер вращается или перемещается по оси, чтобы покрыть широкую область, создавая двумерное изображение окружающего пространства. С помощью таких данных система может строить карту и распознавать объекты, определяя их положение и расстояние до них, что крайне важно для навигации роботов и беспилотных автомобилей.

IMU (Inertial Measurement Unit) — это датчик, который измеряет и сообщает информацию о движении и ориентации объекта в пространстве. Он состоит из трех основных компонентов: акселерометров, гироскопов и иногда магнитометров. Акселерометры измеряют ускорения по трем осям (X, Y, Z), что позволяет определить изменение скорости и положение объекта относительно земной гравитации. Гироскопы отслеживают угловые скорости вращения вокруг тех же осей, что помогает измерять ориентацию объекта и его вращения. Магнитометры, если они присутствуют, измеряют магнитное поле Земли, что позволяет дополнительно корректировать ориентацию.

Принцип работы IMU заключается в интеграции данных с этих сенсоров, чтобы получить полное представление о движении и положении объекта. Например, акселерометры могут обнаружить, если устройство наклоняется или ускоряется, а гироскопы отслеживают угловые изменения, такие как вращение вокруг своей оси. Это позволяет системе вычислить изменения ориентации и траекторию движения, что полезно в таких приложениях, как робототехника, авиация и навигация в условиях отсутствия GPS.

GPS – это навигационная система, основанная на использовании спутников для определения местоположения объектов на Земле. Система состоит из спутников, находящихся на орбите, наземных станций и приемников, которые используются для получения данных о местоположении. Спутники передают сигналы с точным временем, и приемник на Земле, получая эти сигналы от нескольких спутников, может вычислить свое местоположение.

Принцип работы GPS заключается в измерении времени, которое требуется сигналу, чтобы добраться от спутника до приемника. Поскольку спутники известны своей точной орбитой, приемник может определить расстояние до каждого спутника, используя это время. Получая сигналы от как минимум четырех спутников, приемник может точно вычислить свою абсолютную позицию в трехмерном пространстве — определяя широту, долготу и высоту, а также время. Эти данные обеспечивают высокую точность определения местоположения, что критически важно для навигации и локализации в реальном времени.

3.2 Проектирование архитектуры

После того, как были сформулированы функциональные требования к разрабатываемой системе, а также исходя из результатов анализа существующих программных решений, можно определить основные моменты организации системы, в которой будет функционировать разрабатываемое программное решение.

Процесс проектирования архитектуры программного обеспечения включает в себя сбор требований, их анализ и создание проекта для компонента программного обеспечения в соответствие с требованиями. Успешная разработка ПО должна обеспечивать баланс неизбежных компромиссов вследствие противоречащих требований; соответствовать принципам проектирования и рекомендованным методам, выработанным со временем; и дополнять современное оборудование, сети и системы управления.

Архитектуру программного обеспечения можно рассматривать как сопоставление между целью компонента ПО и сведениями о реализации в коде. Правильное понимание архитектуры обеспечит оптимальный баланс требований и результатов. Только программное обеспечение с хорошо продуманной архитектурой способно выполнять указанные задачи с параметрами исходных требований, одновременно обеспечивая максимально высокую производительность. Программное средство построено на основе модульной архитектуры.

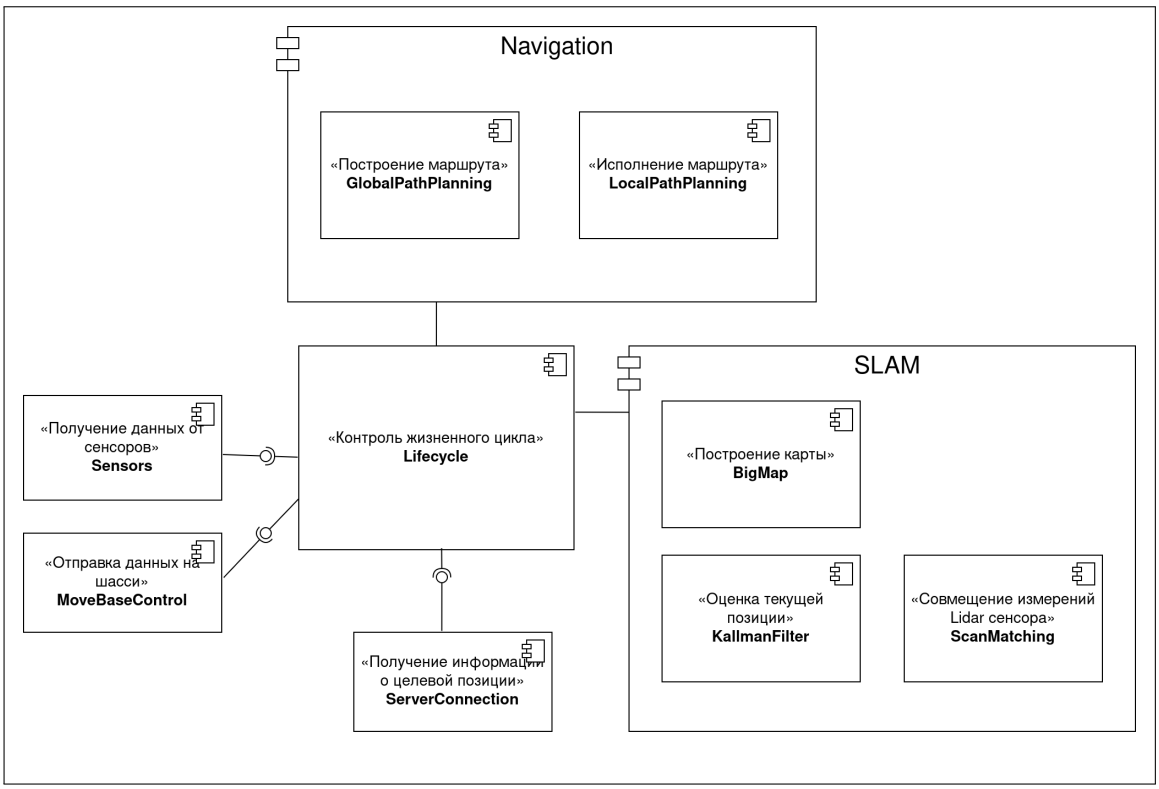

Рисунок 3.2 – Диаграмма компонентов проектируемого ПО

На рисунке 6.1 отображены модули системы:
- модуль жизненного цикла;
- модуль построения маршрута;
- модуль исполнения маршрута;
- модуль получения данных сенсоров;
- модуль отправки данных на шасси;
- модуль получения информации о целевой позиции;
- модуль построения карты;
- модуль оценки позиции;
- модуль совмещения измерений Lidar сенсора.

3.3 CSM

Correlative scan matching (CSM) — это метод регистрации сканов лидара, используемый в робототехнике для определения относительного положения робота на карте. Его ключевое преимущество — устойчивость к локальным минимумам и высокая точность, что делает его критически важным для задач одновременной локализации и построения карт (SLAM).

Принцип работы CSM заключается в поиске оптимального преобразования (сдвига и поворота) между двумя наборами точек (сканами), при ко-

тором достигается максимальное совпадение между ними. В отличие от итеративных методов, таких как ICP (Iterative Closest Point), которые зависят от начального приближения и могут застревать в локальных оптимумах, CSM осуществляет дискретный перебор возможных трансформаций в заданном диапазоне. Для каждой трансформации вычисляется функция качества совпадения, основанная на вероятностной модели окружающей среды или на карте стоимости.

Алгоритм строит карту стоимости, где каждой точке пространства соответствует значение, отражающее вероятность её принадлежности к объекту или свободному пространству. Затем, перебирая множество вариантов сдвигов и поворотов, CSM вычисляет суммарную оценку совпадения между текущим сканом и картой. Оптимальное преобразование выбирается как то, при котором эта оценка максимальна.

Преимущества CSM включают:

– глобальный поиск решения, минимизирующий риск сходимости к локальным минимумам;

– высокую устойчивость к шуму и ошибкам сенсорных данных;

– возможность работы при значительной начальной неопределённости положения.

CSM широко применяется в задачах одновременной локализации и построения карт (SLAM), особенно для коррекции ошибок одометрии и закрытия петель, что позволяет значительно повысить точность и надёжность навигационных систем мобильных роботов.

3.4 ICP

Iterative Closest Point (ICP) — это классический алгоритм регистрации облаков точек, широко используемый в компьютерном зрении и робототехнике для точного выравнивания двух наборов данных, полученных с помощью лидаров или других 3D-сканеров.

Основная цель ICP — минимизировать расстояние между двумя облаками точек: фиксированным эталонным (reference) и подвижным (source), который необходимо трансформировать (сдвинуть и повернуть) так, чтобы максимально приблизить к эталону. Алгоритм работает итеративно, последовательно уточняя параметры преобразования.

3.4.1 Принцип работы ICP

1 Алгоритм начинается с предварительной оценки преобразования, которое приблизительно совмещает исходное облако с эталонным. Качество на-

чального приближения существенно влияет на результат, поскольку ICP может сойтись к локальному минимуму.

2 Для каждой точки подвижного облака находится ближайшая точка в эталонном облаке по евклидову расстоянию. Для ускорения поиска обычно используется структура данных k-d дерево.

3 На основе найденных пар точек вычисляется оптимальное преобразование (смещение и поворот), минимизирующее среднеквадратичное расстояние между соответствующими точками. Часто применяется метод наименьших квадратов.

4 Подвижное облако точек трансформируется с использованием найденного преобразования.

5 Шаги поиска соответствий и оценки преобразования повторяются до тех пор, пока изменение ошибки не станет меньше заданного порога или не будет достигнуто максимальное число итераций.

3.4.2 Особенности и ограничения

– ICP чувствителен к качеству начального приближения и может застревать в локальных оптимумах;

– алгоритм хорошо работает при небольших смещениях и поворотах между сканами;

– существует множество вариантов ICP, включая point-to-point (точка к точке) и point-to-plane (точка к плоскости), последний из которых лучше подходит для структурированных поверхностей;

– ICP широко применяется для локализации роботов, построения карт, сшивки 3D-моделей и коррекции ошибок одометрии.

ICP является базовым инструментом для регистрации 2D и 3D данных в задачах SLAM, реконструкции объектов и навигации мобильных платформ, особенно когда требуется точное совмещение облаков точек, полученных с разных позиций или в разное время.

Таким образом, ICP — это эффективный и относительно простой алгоритм, обеспечивающий точное выравнивание облаков точек за счёт итеративного уточнения преобразования между ними.

В алгоритме Iterative Closest Point (ICP) задача сводится к поиску оптимального жёсткого преобразования (поворота и сдвига), которое минимизирует сумму квадратов расстояний между соответствующими точками двух облаков. Для решения этой задачи на каждом шаге, когда соответствия между точками уже известны, широко применяется метод сингулярного разложения матриц (SVD, Singular Value Decomposition).

3.5 Описание модулей системы

Коммуникация между модулями осуществляется через модуль жизненного цикла, все модули получают и отправляют информацию через него, не считая сильно-связных модулей в системе SLAM.

Модуль получения данных сенсоров осуществляет сбор информации: 2D Lidar предоставляет информацию о расстояниях до объектов в окружающем пространстве, IMU — данные об угловых ускорениях и наклоне устройства, а GPS — информацию о глобальном местоположении. Все эти данные передаются в модуль SLAM, который использует их для построения карты окружающей среды и вычисления текущего местоположения робота. Это позволяет системе иметь точную картину окружающего мира и следить за положением устройства.

Модуль совмещения измерений Lidar сенсора, является основой для построения карты и локализации робота. С помощью данных от лидаров и модуля оценки позиции он строит карту пространства, постоянно обновляя ее по мере движения робота, и вычисляет его местоположение относительно этой карты. Это позволяет системе динамично корректировать действия робота в зависимости от изменений в окружающей среде, таких как появление новых препятствий или изменение положения объектов.

Полученные от совмещения измерений данные о местоположении робота передаются в модуль оценки позиции. Этот модуль анализирует текущее положение устройства с использованием фильтрации и различных методов оценки, таких как фильтр Калмана. Оценка позиции робота имеет важное значение для корректного планирования маршрута, поскольку точность информации о местоположении напрямую влияет на точность движений устройства.

Модуль создания маршрута отвечает за вычисление оптимального пути от текущего местоположения робота до заданной цели. Этот модуль использует данные о местоположении, а также информацию о препятствиях, чтобы планировать наиболее эффективный и безопасный маршрут. Важно, чтобы система могла адаптироваться к изменениям окружающей среды, например, при возникновении новых препятствий, система должна пересчитать маршрут в реальном времени, обеспечивая продолжение движения робота без ошибок.

После того как маршрут спланирован, информация о нем передается в модуль управления моторами. Этот модуль отвечает за выполнение команд, таких как движение вперед, повороты и торможение. Модуль управления моторами должен обеспечить точное выполнение команд с минимальными

задержками, чтобы робот мог двигаться по маршруту с высокой точностью. Кроме того, он должен поддерживать оперативную реакцию на данные от сенсоров, такие как сигнал от лидаров, предупреждающий о близко расположенных препятствиях.

При обнаружении препятствий вблизи, например, если расстояние до объекта становится меньше заданного порога, система должна немедленно реагировать. Это может быть реализовано командой «стоп», которая отправляется в модуль управления моторами для немедленной остановки робота. Такие меры безопасности необходимы для предотвращения столкновений и обеспечения безопасной работы робота в различных условиях.

3.6 Карты

3.7 Модель состояния системы

Для описания динамики системы используется вектор состояния X:

$$X = (x, y, v_x, v_y, a_x, a_y, \theta, v_\theta, a_\theta)^T,$$

где x, y – координаты объекта в локальной карте, м; v_x, v_y – компоненты скорости по осям x и y, м/с; a_x, a_y – компоненты ускорения по осям x и y, м/с^2; θ – угол ориентации объекта, ; v_θ – угловая скорость, рад/с; a_θ – угловое ускорение, рад/с^2.

Выбор данной модели состояния обусловлен следующими факторами:

1 Вектор состояния включает положение, скорость и ускорение объекта как в поступательном, так и в угловом движении. Это позволяет точно описать сложные траектории, включая вращение.

2 Включение ускорений (a_x, a_y, a_θ), то есть использование модели константное ускорения, позволяет моделировать изменения скорости и ориентации. Таким образом, повышается точность предсказания в условиях неравномерного движения.

В данной системе с вектором состояния $X = (x, y, v_x, v_y, a_x, a_y, \theta, v_\theta, a_\theta)^T$ используются линейная модель преобразования состояния (F) и линейная модель преобразования к измерениям (H). Эти модели описывают динамику системы и связь состояния с измерениями от датчиков (GPS и LIDAR), соответственно.

3.8 Модель преобразования состояния

Модель преобразования состояния описывает эволюцию вектора состояния X_t во времени и задаётся линейным уравнением:

$$X_{t+1} = FX_t + w_t.$$

где F – матрица перехода состояния, $w_t \sim \mathcal{N}(0, Q)$ – гауссов шум процесса с ковариацией Q.

Для вектора состояния $X = (x, y, v_x, v_y, a_x, a_y, \theta, v_\theta, a_\theta)^T$, матрица F имеет блочно-диагональную структуру, отражая кинематические зависимости для поступательного и углового движения. Пример структуры матрицы F (для времени шага Δt) выглядит следующим образом:

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.1}$$

Матрица F отражает линейную модель поступательного движения и модель углового движения:

$$\begin{cases} x_{t+1} = x_t + v_{x,t}\Delta t + \dfrac{1}{2}a_{x,t}\Delta t^2, \\ v_{x,t+1} = v_{x,t} + a_{x,t}\Delta t, \\ a_{x,t+1} = a_{x,t} + w_{a_x,t}, \\ y_{t+1} = y_t + v_{y,t}\Delta t + \dfrac{1}{2}a_{y,t}\Delta t^2, \\ v_{y,t+1} = v_{y,t} + a_{y,t}\Delta t, \\ a_{y,t+1} = a_{y,t} + w_{a_y,t}, \\ \theta_{t+1} = \theta_t + v_{\theta,t}\Delta t + \dfrac{1}{2}a_{\theta,t}\Delta t^2, \\ v_{\theta,t+1} = v_{\theta,t} + a_{\theta,t}\Delta t, \\ a_{\theta,t+1} = a_{\theta,t} + w_{a_\theta,t}. \end{cases} \tag{3.2}$$

3.9 Модель измерений

Модель измерений связывает вектор состояния X_t с вектором измерений z_t:

$$z_t = HX_t + v_t, \tag{3.3}$$

где H – матрица измерений, $v_t \sim \mathcal{N}(0, R)$ – гауссов шум измерений с ковариацией R.

3.9.1 Вектор измерений

Вектор измерений z_t включает компоненты от GPS и LIDAR. Предполагается, что GPS предоставляет координаты (x, y), а LIDAR – координаты (x, y) и угол ориентации θ. Таким образом, вектор измерений имеет вид:

$$z_t = [x_{\text{GPS}}, y_{\text{GPS}}, x_{\text{LIDAR}}, y_{\text{LIDAR}}, \theta_{\text{LIDAR}}]^T.$$

Размерность z_t равна 5 (две координаты от GPS, две координаты и угол от LIDAR), хотя в случае асинхронного поступления данных некоторые компоненты могут быть недоступны в определённые моменты времени.

3.9.2 Матрица измерений H

Матрица H (см. уравнение 3.4) размером 5×9 заполняется единицами только в позициях, соответствующих измеряемым компонентам состояния.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \tag{3.4}$$

3.9.3 Обработка асинхронных измерений

В случае асинхронного поступления данных от GPS и LIDAR, если в момент времени t доступны не все компоненты z_t, матрица H и вектор z_t усекаются до соответствующих строк и элементов. Например, если доступны только данные GPS ($z_t = [x_{\text{GPS}}, y_{\text{GPS}}]^T$), используется подматрица:

$$H_{\text{GPS}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Аналогично, для данных от LIDAR ($z_t = [x_{\text{LIDAR}}, y_{\text{LIDAR}}, \theta_{\text{LIDAR}}]^T$) используется:

$$H_{\text{LIDAR}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Ковариационная матрица шума R также корректируется в зависимости от доступных измерений.

3.9.4 Нелинейность модели

Хотя модели преобразования состояния F и измерений H в данной системе линейны, для оценки состояния был выбран UKF вместо линейного фильтра Калмана. Основная причина выбора UKF заключается в:

1 Необходимости корректной обработки угловой компоненты θ, которая, несмотря на линейность модели, требует специального подхода к усреднению в циклическом пространстве углов. Например, через тригонометрические функции sin и cos для учёта переходов через $0/2\pi$).

2 UKF обеспечивает повышенную устойчивость к выбросам в данных GPS и LIDAR, эффективно обрабатывает асинхронные измерения благодаря очередям событий и обладает гибкостью для адаптации к возможным нелинейным расширениям модели в будущем.

Алгоритм работы (UKF состоит из двух последовательных алгоритмов: алгоритма шага предсказания (см. рисунок 3.3) и алгоритм шага обновления (см. рисунок 3.4).

Рисунок 3.3 – Алгоритм шага предсказания UKF

На этапе предсказания UKF генерирует набор сигма-точек на основе текущей оценки состояния и ковариации, пропускает их через модель системы для прогнозирования следующего состояния и ковариации. На этапе обновления, при поступлении измерений, сигма-точки пропускаются через модель измерений, после чего вычисляются среднее измерение, ковариация и коэффициент усиления Калмана, используемые для корректировки состояния и ковариации с учётом новых данных. Этот подход позволяет UKF эффективно обрабатывать нелинейности и обеспечивать точную оценку состояния.

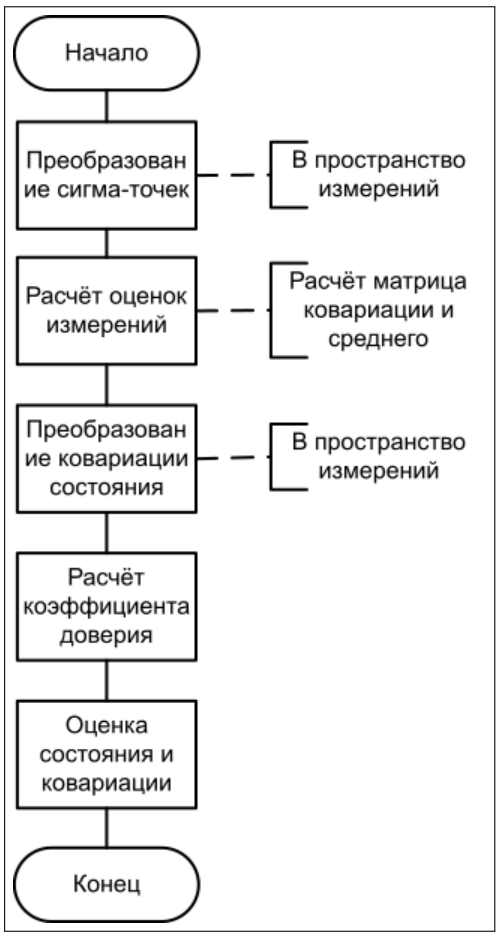

Рисунок 3.4 – Алгоритм шага обновления UKF

3.9.5 События обновления состояния

Данные от датчиков формируют события изменения состояния. Все события изменяют состояние системы в соответствии с алгоритмом обработки событий (см. рисунок 3.5)

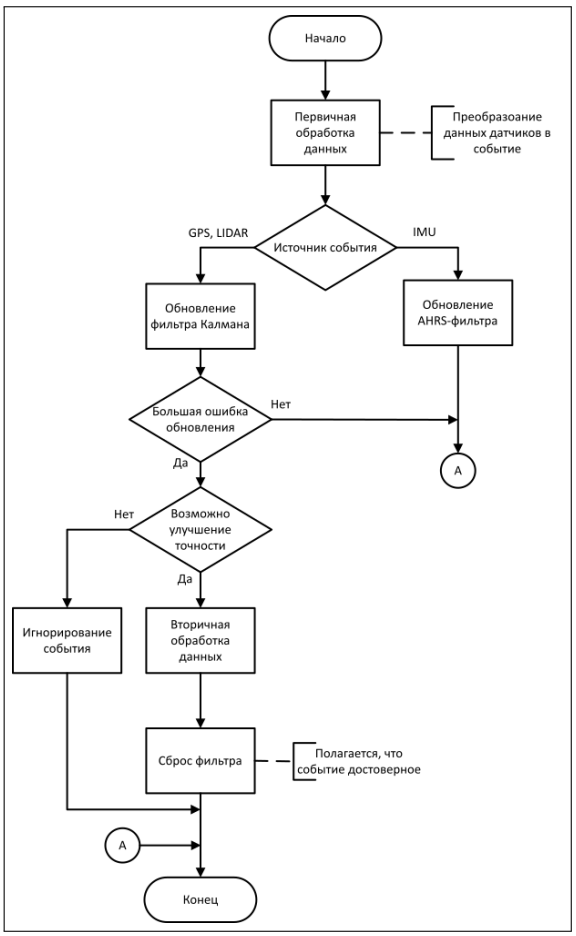

Рисунок 3.5 – Алгоритм обработки событий

Обновление состояния в UKF выполняется только по позиционным событиям, которые происходят относительно редко, но обладают высокой точностью. Позиционными событими являются содержат оценку координат робот в системе координат локальной карты робота (x, y) и необязательную оценку ориентации робота в пространстве (θ). К таким событиям относятся результаты обработки данных от GPS и LIDAR.

Для обработки данных IMU используется фильтр Маджвика (см. подраздел). Было принято решение использовать отдельный фильтр для обработки данных IMU, а не обновлять вектор системы X, по следующим причинам:

– данные IMU измеряют только ориентацию, потому фильтр не сможет произвести коррекцию всего вектора состояния во времени;

– данные от IMU приходят с гораздо большей частотой (1000 Hz), чем данные от LIDAR (10 Hz) или GPS (5 Hz).

Оценка ориентации с помощью фильтра Маджвика используется в процессе обработки данных LIDAR или GPS для формирования позиционных событий.

Таким образом, использованием фильтра Маджвика обеспечивает на-

дежную и частую оценку ориентации, в то время как UKF использует редкие, но точные позиционные данные для коррекции полного вектора состояния. Это повышает общую точность и устойчивость системы в условиях сложной динамики и возможных выбросов в измерениях.

3.10 Очереди событий и очереди состояний

Каждое состояние системы X_k и событие E_k закреплено к моменту времени t_k. Каждое новое событие обновляет модель системы в сооветствие с алгоритмом применения события (см. рисунок 3.6)

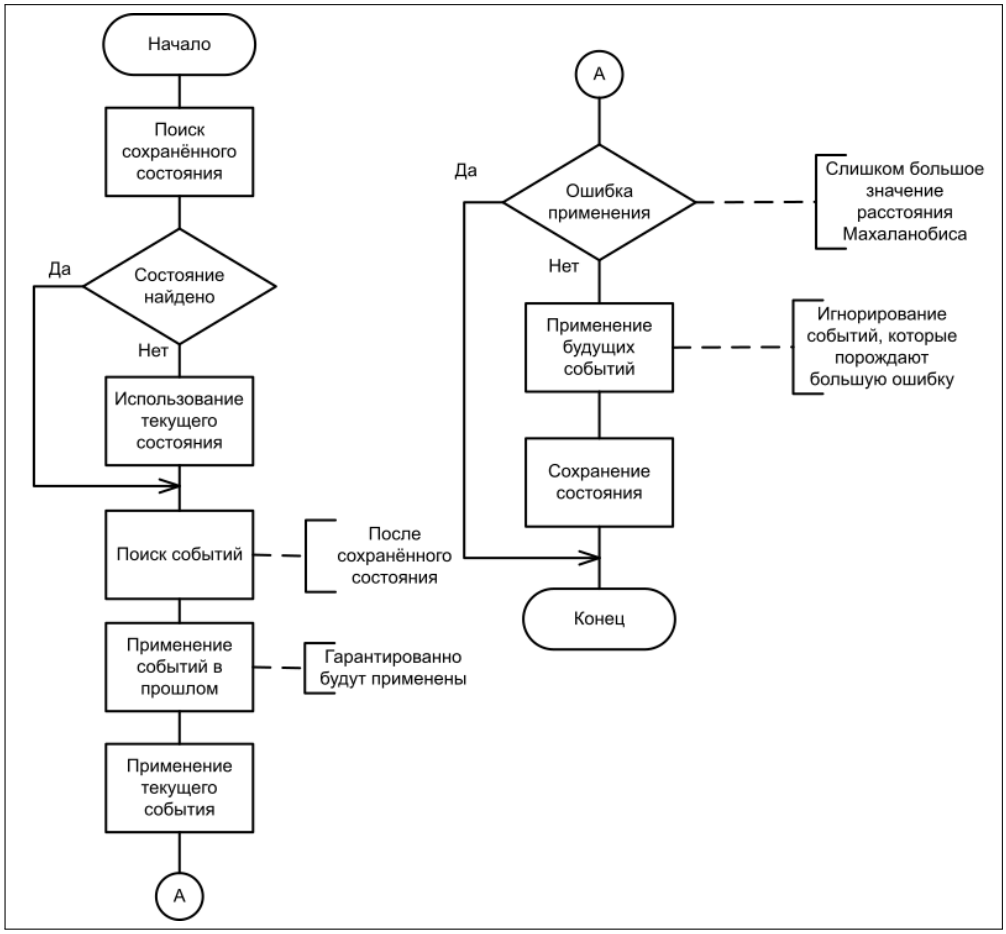

Рисунок 3.6 – Алгоритм применения события

Для эффективной обработки данных и управления временной динамикой системы реализованы очереди состояний и событий. Это позволяет синхронизировать данные от различных источников и корректно обновлять состояние системы в UKF.

3.10.1 Очередь состояний

Очередь состояний представляет собой упорядоченный набор векторов состояния системы. Основные характеристики очереди состояний:

– каждое состояние в очереди соответствует определённому моменту времени, что позволяет отслеживать эволюцию системы и выполнять ретроспективные корректировки при получении новых данных;

– временные метки состояний используются для сопоставления с событиями, обеспечивая согласованность между предсказаниями и измерениями;

– при поступлении новых данных очередь состояний обновляется, добавляя новое состояние.

3.10.2 Очередь событий

Очередь событий содержит результаты обработки данных от датчиков. Каждое из событие связано с моментом времени t. События представляют собой позиционные данные, которые используются для коррекции состояния в UKF. Основные характеристики очереди событий:

– каждое событие имеет метку времени для определения положения относительно текущего момента времени;

– относительно события с временем t_{current} события разделяются на события прошлого и события будущего;

– события прошлого используются для ретроспективной коррекции состояний, если данные поступили с задержкой или требуется уточнение предыдущих оценок;

– события будущего хранятся в очереди для обработки по мере продвижения текущего времени системы. Это позволяет системе быть готовой к асинхронным данных.

– очередь событий поддерживает асинхронное поступление данных от датчиков с разной частотой.

3.11 Планирование движения

3.11.1 Локальный планировщик

В качестве локального планировщика используется DWA. Целевая функция:

$$G(v, \omega) = \sum_{i=1}^{k} w_i \cdot g_{i,\text{norm}}(v, \omega), \tag{3.5}$$

где $g_{i,\text{norm}}(v, \omega)$ – значение i-й подцелевой функции, w_i – весовой коэффициент, настраиваемый для задания приоритетов, а k – количество метрик.

Для обеспечения сравнимости метрик, имеющих разные единицы изме-

рения и диапазоны, значения подцелевых функций $g_i(v, \omega)$ нормализуются по всем возможным парам скоростей в динамическом окне V_d. Нормализованное значение вычисляется как:

$$g_{i,\text{norm}}(v, \omega) = \frac{g_i(v, \omega) - \min_{(v',\omega')\in V_d} g_i(v', \omega')}{\max_{(v',\omega')\in V_d} g_i(v', \omega') - \min_{(v',\omega')\in V_d} g_i(v', \omega')}, \tag{3.6}$$

где $\min_{(v',\omega')\in V_d} g_i(v', \omega')$ и $\max_{(v',\omega')\in V_d} g_i(v', \omega')$ – минимальное и максимальное значения i-й подцелевой функции среди всех пар (v', ω') в V_d.

Нормализация приводит значения к диапазону $[0, 1]$, что позволяет весам w_i эффективно управлять приоритетами метрик и обеспечивает сбалансированную оценку траекторий.

Для каждой пары скоростей предлагается построение уникальной траектории. Оценка происходит не столько по значениями скорости, сколько по траекториям, которые скорости генерируют. В качестве метрик предлагается использовать:

а) Метрика близости к цели. Чем ближе конечная точка траектория к цели, тем траектория приоритетнее.

б) Метрика близости к глобальной траектории. Чем ближе форма траектории к прямолинейной форме траектории, тем траектория приоритетнее.

в) Метрика избегания препятствий. Приоритет отдаётся тем траектории, которые проходят дальше от препятствий.

г) Метрика колебательных движений. Предотвращает резкие (осциллирующие) изменения скоростей для обеспечения плавности и зацикливания.

д) Выравнивание по целевой ориентации. Минимизирует отклонение ориентации робота от желаемого угла в целевой точке.

е) Метрика коллизий. Отклоняет все траектории, в которых обнаружены столкновения с объектами препятствий.

3.12 Взаимодействие с периферией

В процессе разработки программного обеспечения для автономной навигации мобильных платформ одной из ключевых задач стало обеспечение гибкого и надёжного взаимодействия с периферийными устройствами, такими как датчики, камеры и лидары. После анализа различных подходов было принято решение реализовать это взаимодействие с использованием стека протоколов TCP/IP. Такой выбор обусловлен универсальностью и стандартизацией данного протокола, который широко применяется в сетевых технологиях и позволяет организовать стабильное соединение между компонентами системы. Это решение обеспечивает возможность передачи данных в

реальном времени, что критически важно для задач управления и обработки информации в динамичной среде.

Использование TCP/IP стека предоставляет значительное преимущество в виде модульности и расширяемости системы. Благодаря этому подходу стало возможным подключение различных датчиков к программе непосредственно во время её работы, без необходимости останавливать или перезапускать систему. Например, если в процессе эксплуатации мобильной платформы потребуется добавить новый лидар или ультразвуковой датчик, это можно сделать ”на лету что существенно повышает адаптивность системы к изменяющимся условиям или требованиям задачи. Такая гибкость особенно ценна в экспериментальных или полевых условиях, где заранее предусмотреть все сценарии использования невозможно.

Реализация взаимодействия через TCP/IP также упрощает интеграцию с современными технологиями и стандартами, используемыми в робототехнике. Например, многие устройства уже имеют встроенную поддержку сетевых протоколов, что позволяет избежать разработки сложных проприетарных интерфейсов для каждого типа периферии. Кроме того, TCP/IP обеспечивает надёжную передачу данных с механизмом проверки ошибок, что снижает риск потери критически важной информации от датчиков. Это особенно актуально для автономных систем, где точность и своевременность получения данных напрямую влияют на качество навигации и принятия решений.

Наконец, выбор TCP/IP стека открывает перспективы для дальнейшего развития проекта в сторону распределённых систем. В будущем это позволит не только подключать датчики локально, но и организовывать взаимодействие между несколькими мобильными платформами или центральным сервером через сеть. Такой подход может быть полезен, например, для координации группы роботов или передачи данных в облако для анализа. Таким образом, использование TCP/IP не только решает текущие задачи взаимодействия с периферией, но и закладывает фундамент для масштабирования системы, делая её более универсальной и готовой к новым вызовам в области автономной навигации.

3.13 Язык программирования

Robot Operating System (ROS) представляет собой широко используемую программную платформу для разработки робототехнических систем, и одной из её ключевых особенностей является то, что она написана на языке программирования C++. Этот выбор не случаен: C++ считается стандартом индустрии благодаря своей высокой производительности, гибкости и возмож-

ности работы на низком уровне с аппаратным обеспечением. В контексте робототехники, где требуется быстрая обработка данных с датчиков и управление механизмами в реальном времени, такие качества C++ становятся незаменимыми. Использование C++ в ROS позволяет разработчикам создавать эффективные и масштабируемые решения для сложных задач, таких как автономная навигация, обработка сигналов или взаимодействие с физическими устройствами. Этот язык обеспечивает тонкий контроль над ресурсами системы, что особенно важно для мобильных платформ с ограниченными вычислительными мощностями. Кроме того, C++ обладает богатым набором библиотек и инструментов, которые упрощают интеграцию ROS с другими технологиями, укрепляя его как стандарта в индустрии робототехники.

Несмотря на все преимущества C++ как стандарта индустрии и основы для ROS, в последние годы всё большее внимание в разработке программного обеспечения, включая робототехнику, привлекает язык программирования Rust. В контексте ROS уже появляются инициативы по интеграции Rust, что может дополнить или даже со временем частично заменить C++, предлагая разработчикам более надёжный и удобный инструмент для создания автономных систем, сохраняя при этом совместимость с существующей экосистемой ROS.

Одним из ключевых преимуществ Rust является его способность обеспечивать безопасность многозадачности. В отличие от C++, который требует дополнительных усилий для безопасного выполнения параллельных операций, Rust изначально предусматривает механизмы предотвращения гонок данных, что делает код более надежным. Это особенно важно для системы навигации, где необходимо параллельно обрабатывать данные с различных сенсоров и вычислять управляющие команды без риска возникновения ошибок синхронизации.

Rust также предоставляет встроенные инструменты для работы с асинхронным программированием, что позволяет эффективно организовать обработку данных в реальном времени. Асинхронные операции позволяют системе собирать данные с сенсоров, планировать маршрут и управлять моторами без блокировки основного потока выполнения, что способствует повышению производительности и снижению задержек.

Программная экосистема Rust активно развивается, и существует множество библиотек, которые могут быть использованы для решения задач, связанных с обработкой сенсорных данных, математическими расчетами и оптимизацией маршрутов. Это позволяет разработчикам легко интегрировать необходимые инструменты и сокращать время на разработку и тестирование системы. Также, благодаря хорошей поддержке со стороны сообщества,

Rust предоставляет разработчикам множество ресурсов для быстрого решения возникающих вопросов.

Ключевым преимуществом Rust является его кроссплатформенность. Код, написанный на этом языке, может быть скомпилирован для различных платформ, что делает Rust отличным выбором для мобильных роботов, которые могут работать на разных типах оборудования. Это позволяет без значительных усилий адаптировать систему под разные архитектуры и аппаратные платформы.

Будущие улучшения системы могут включать в себя добавление новых сенсоров, улучшение алгоритмов SLAM и маршрутизации, а также интеграцию с внешними системами, такими как онлайн-карты или системы для прогнозирования дорожной ситуации. Rust, благодаря своей гибкости и безопасному управлению памятью, идеально подходит для такой работы, обеспечивая долгосрочную устойчивость и развитие проекта.

Таким образом, проектирование программного обеспечения для системы мобильной навигации с использованием сенсоров и алгоритмов SLAM требует тщательной проработки архитектуры, выбора эффективных технологий и инструментов. Язык Rust является отличным выбором для разработки таких систем, благодаря своим преимуществам в безопасности, производительности и поддержке многозадачности, что делает его идеальным для создания высоконадежных и высокопроизводительных приложений для робототехники.

Robot Operating System (ROS) представляет собой широко используемую программную платформу для разработки робототехнических систем, и одной из её ключевых особенностей является то, что она написана на языке программирования C++. Этот выбор не случаен: C++ считается стандартом индустрии благодаря своей высокой производительности, гибкости и возможности работы на низком уровне с аппаратным обеспечением. В контексте робототехники, где требуется быстрая обработка данных с датчиков и управление механизмами в реальном времени, такие качества C++ становятся незаменимыми. Использование C++ в ROS позволяет разработчикам создавать эффективные и масштабируемые решения для сложных задач, таких как автономная навигация, обработка сигналов или взаимодействие с физическими устройствами. Этот язык обеспечивает тонкий контроль над ресурсами системы, что особенно важно для мобильных платформ с ограниченными вычислительными мощностями. Кроме того, C++ обладает богатым набором библиотек и инструментов, которые упрощают интеграцию ROS с другими технологиями, укрепляя его как стандарта в индустрии робототехники.

Несмотря на все преимущества C++ как стандарта индустрии и осно-

вы для ROS, в последние годы всё большее внимание в разработке программного обеспечения, включая робототехнику, привлекает язык программирования Rust. В контексте ROS уже появляются инициативы по интеграции Rust, что может дополнить или даже со временем частично заменить C++, предлагая разработчикам более надёжный и удобный инструмент для создания автономных систем, сохраняя при этом совместимость с существующей экосистемой ROS.

Одним из ключевых преимуществ Rust является его способность обеспечивать безопасность многозадачности. В отличие от C++, который требует дополнительных усилий для безопасного выполнения параллельных операций, Rust изначально предусматривает механизмы предотвращения гонок данных, что делает код более надежным. Это особенно важно для системы навигации, где необходимо параллельно обрабатывать данные с различных сенсоров и вычислять управляющие команды без риска возникновения ошибок синхронизации.

Rust также предоставляет встроенные инструменты для работы с асинхронным программированием, что позволяет эффективно организовать обработку данных в реальном времени. Асинхронные операции позволяют системе собирать данные с сенсоров, планировать маршрут и управлять моторами без блокировки основного потока выполнения, что способствует повышению производительности и снижению задержек.

Программная экосистема Rust активно развивается, и существует множество библиотек, которые могут быть использованы для решения задач, связанных с обработкой сенсорных данных, математическими расчетами и оптимизацией маршрутов. Это позволяет разработчикам легко интегрировать необходимые инструменты и сокращать время на разработку и тестирование системы. Также, благодаря хорошей поддержке со стороны сообщества, Rust предоставляет разработчикам множество ресурсов для быстрого решения возникающих вопросов.

Ключевым преимуществом Rust является его кроссплатформенность. Код, написанный на этом языке, может быть скомпилирован для различных платформ, что делает Rust отличным выбором для мобильных роботов, которые могут работать на разных типах оборудования. Это позволяет без значительных усилий адаптировать систему под разные архитектуры и аппаратные платформы.

Будущие улучшения системы могут включать в себя добавление новых сенсоров, улучшение алгоритмов SLAM и маршрутизации, а также интеграцию с внешними системами, такими как онлайн-карты или системы для прогнозирования дорожной ситуации. Rust, благодаря своей гибкости и безопас-

ному управлению памятью, идеально подходит для такой работы, обеспечивая долгосрочную устойчивость и развитие проекта.

Таким образом, проектирование программного обеспечения для системы мобильной навигации с использованием сенсоров и алгоритмов SLAM требует тщательной проработки архитектуры, выбора эффективных технологий и инструментов. Язык Rust является отличным выбором для разработки таких систем, благодаря своим преимуществам в безопасности, производительности и поддержке многозадачности, что делает его идеальным для создания высоконадежных и высокопроизводительных приложений для робототехники.

4 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

4.1 Модуль жизненного цикла

Программа спроектирована по принципу цикла событий, где мы обрабатываем получаемые сообщения от системы.

В цикле прописаны обработчики каждого сообщения. Виды сообщений:

– сообщения от устройств периферии;

– сообщения от модуля навигации.

Каждый обработчик предполагает чтобы в нём не было трудоёмких вычислений, что позволяет циклу событий непрерывно выполнятся вне зависимости от того какие входные данные были получены.

Схема обработки выглядит вот так:

4.2 SLAM

Задача построения карты заключается в добавлении новых данных поступающих с датчиков на уже построенную карту. В качестве входных данных у нас данные с лидара с позицией в которых они были произведены, а так-же карта на которую происходит наложение скана.

В качестве алгоритма наложения был выбран алгоритм ICP (Iterative closest point), который был разработан с использованием наработок в KISS ICP. Алгоритм заключается в итеративном приближении наложения скана к карте.

4.3 Разработка UKF

Фильтр описывается структурой:

```
pub struct KallmanFilter<const N: usize, const K: usize, const
   S: usize> {
    pub state: SMatrix<f32, 1, N>,
    prior_state: Option<SMatrix<f32, 1, N>>,
    pub state_sigmas: Option<SMatrix<f32, S, N>>,
    pub state_noise: SMatrix<f32, N, N>,
    pub meas_sigmas: SMatrix<f32, S, K>,
    pub meas_noise: SMatrix<f32, K, K>,
    pub meas_covariance: Option<DMatrix<f32>>,
    pub inv_meas_covariance: Option<DMatrix<f32>>,
    pub covariance: SMatrix<f32, N, N>,
    prior_covariance: Option<SMatrix<f32, N, N>>,
    pub gain: Option<OMatrix<f32, Const<N>, Dyn>>,
    pub weights: SigmaWeights<S>,
}
```

Конструктор:

```
pub fn new() -> Self {
     assert_eq!(sigma_order(N), S);

     let weights =
        sigma_points::estimate_merwe_weights::<N, S>(
            sigma_points::SigmaMetadata {
                alpha: 0.1,
                beta: 2.0,
                kappa: (N as f32 - 3.0),
            },
     );
     Self::with_weights(weights)
}
```

Расчёт весов сигма-точек выполняется в момент инициализации фильтра Калмана следующим образом:

```
pub fn estimate_merwe_weights<const N: usize, const S: usize>(
    metadata: SigmaMetadata,
) -> SigmaWeights<S> {
    assert!(S == sigma_order(N));
    let SigmaMetadata { alpha, beta, kappa } = metadata;
    let n = N as f32;
    let lambda = (alpha * alpha) * (n + kappa) - n;
    let c = 0.5 / (n + lambda);
    let mut w_m = vec![c; S];
    let mut w_c = vec![c; S];
    w_c[0] = lambda / (n + lambda) + (1.0 - alpha * alpha +
       beta);
    w_m[0] = lambda / (n + lambda);
    SigmaWeights {
        covariance: SVector::<f32, S>::from_vec(w_c).transpose(),
        mean: SVector::<f32, S>::from_vec(w_m).transpose(),
        sigma_order: S,
        metadata,
    }
}
```

Оценка значений сигма-точек выполняется по алгоритму:

```
pub trait FnDistance<const N: usize>:
    Fn(SMatrixView<f32, 1, N>, SMatrixView<f32, 1, N>) ->
       SMatrix<f32, 1, N>
{/
}

pub fn estimate_merwe_sigmas<const S: usize, const N: usize, D>(
    state: SMatrixView<f32, 1, N>,
    covariance: SMatrixView<f32, N, N>,
    metadata: SigmaMetadata,
    distance_of: &D,
) -> Result<SMatrix<f32, S, N>, MathError>
where
    D: FnDistance<N>,
{
```

```
    let SigmaMetadata { alpha, kappa, .. } = metadata;
    let mean_estimations = {
        let state_size = N as f32;
        let lambda = alpha.powi(2) * (state_size + kappa) -
           state_size;
        let matrix = covariance.scale(lambda + state_size);
        let matrix = matrix.symmetric_part() +
           SMatrix::identity() * 0.0001;
        let Some(cholesky) = Cholesky::new(matrix) else {
            let dmatrix = DMatrix::from_iterator(
                matrix.nrows(),
                matrix.ncols(),
                matrix.iter().copied(),
            );
            return Err(MathError::CholeskyFailed(dmatrix));
        };
        cholesky.l()
    };
    let mut sigmas = SMatrix::<f32, S, N>::zeros();
    sigmas.row_mut(0).copy_from(&state);
    for i in 0..state.len() {
        let mean_estimation =
           mean_estimations.column(i).transpose();
        sigmas
            .row_mut(i + 1)
            .copy_from(&distance_of(state, (-1.0 *
               mean_estimation).as_view()));
        sigmas
            .row_mut(N + i + 1)
            .copy_from(&distance_of(state, (1.0 *
               mean_estimation).as_view()));
    }
    Ok(sigmas)
}
```

Для извлечения квадратного корня из матрицы ковариации P используется декомпозиция Холецкого. Операция декомпозиции определена только над положительно-определёнными матрицами. По определению, матрица ковариации всегда положительно определена (элементами матрицы являются ковариации между переменными). Но в процессе работы фильтра возможна ситуация, что матрица P оказывается отрицательно-определённой. Подобная ситуация возникает из-за:

– накопления аппаратной вычислительной ошибки;

– особенностей способа обновления значений ковариаций в матриц;

– высокой сложности модели системы (более трёх переменных в векторе X).

В случае, если невозможно произвести декомпозицию по Холецкому, предлагается сброс фильтра (см. подраздел 4.4).

Ключевым этапом в работе UKF является реализация алгоритма преоб-

разования

```
pub fn transform<T, F, const N: usize, const S: usize>(
    sigmas: &SMatrix<f32, S, N>,
    weights: &SigmaWeights<S>,
    noise_opt: Option<SMatrixView<f32, N, N>>,
    residual: &T,
    mean: &F,
) -> (SMatrix<f32, 1, N>, SMatrix<f32, N, N>)
where
    T: FnDistance<N>,
    F: FnMean<N, S>,
{
    let state = mean(weights.mean.as_view(), sigmas.as_view());

    let mut covariance = SMatrix::<f32, N, N>::zeros();

    for i in 0..S {
        let row = sigmas.row(i).clone_owned();

        let y = residual(row.as_view(), state.as_view());

        let mut outer = y.transpose() * y;
        outer.scale_mut(weights.covariance[i]);
        covariance += outer;
    }

    if let Some(noise) = noise_opt {
        //assert noise is diagonal matrix
        covariance += noise;
    }

    (state, covariance)
}
```

4.4 Разработка модуля оценки состояния

```
pub type FilterState = ukf::FilterState<
    ACCEL_MODEL_STATE_SIZE,
    MEASUREMENT_SIZE,
    ACCELERATION_MODEL_SIGMA_ORDER
>;

pub type KallmanFilter = ukf::KallmanFilter<
    ACCEL_MODEL_STATE_SIZE,
    MEASUREMENT_SIZE,
    ACCELERATION_MODEL_SIGMA_ORDER,
>;

pub struct MotionEstimation {
    pub config: MotionEstimationConfig,
    pub states: Vec<FilterState>,
    pub events: Vec<SensorEvent>,
    pub last_control_input: Option<ControlInput>,
    pub ukf_filter: KallmanFilter,
```

```
    pub imu_filter: Option<ImuFilter>,
}

fn apply_event(
    &mut self,
    event: SensorEvent,
    observation_time: SystemTime,
) -> Result<(), LocalizationError> {
    let dt_secs = event
        .arrival_time()
        .duration_since(observation_time)
        .expect("Time␣goes␣in␣reverse␣order")
        .as_secs_f32()
        .max(0.001);

    let mut indices = Vec::<usize>::new();
    let mut meas_matrix = SVector::<f32,
       MEASUREMENT_SIZE>::zeros();

    match event {
        SensorEvent::MatchedPose(data) => {
            meas_matrix[SCAN_X] = data.coords.x;
            meas_matrix[SCAN_Y] = data.coords.y;
            meas_matrix[SCAN_THETA] = data.heading_rad;

            indices.push(SCAN_X);
            indices.push(SCAN_Y);
            indices.push(SCAN_THETA);

            self.imu_filter = None;
        }

        SensorEvent::Imu(data) => {
            let mut filter: ImuFilter =
                self.imu_filter.take().unwrap_or_else(|| {
                    ImuFilter::new(
                        self.config.epoch_duration_ms as f32 /
                           1000.0,
                        &self.get_estimated_state_unchecked(),
                        &self.config.noise,
                    )
                })
            filter.predict(dt_secs)?;
            filter.update(
                    dt_secs,
                    self.config.params.imu_filter,
                    &self.config.noise,
                    data
            )?;
            return Ok(());
        }

        SensorEvent::GpsCoords(data) => {
            meas_matrix[GPS_X] = data.coords.x;
            meas_matrix[GPS_Y] = data.coords.y;
```

```
            indices.push(GPS_X);
            indices.push(GPS_Y);
        }
    }

    let transform = build_dyn_transform(&indices);

    let _ = self.predict(dt_secs)?;

    self.ukf_filter.update(
        &meas_matrix.transpose(),
        Some(transform),
        UpdateConfig {
            state_to_meas,
            mean_transform: meas_mean,
            meas_sub: meas_residual,
            state_sub: state_residual,
        },
    )?;

    Ok(())
}
```

Функции для преобразования сигма-точек

```
fn meas_residual(
    m1: SMatrixView<f32, 1, MEASUREMENT_SIZE>,
    m2: SMatrixView<f32, 1, MEASUREMENT_SIZE>,
) -> SMatrix<f32, 1, MEASUREMENT_SIZE> {
    let mut m = m1 - m2;
    m[SCAN_THETA] = normalize_angle(m[SCAN_THETA]);
    m[IMU_THETA] = normalize_angle(m[IMU_THETA]);
    m
}

fn meas_mean(
    weights: SMatrixView<f32, 1, ACCELERATION_MODEL_SIGMA_ORDER>,
    sigmas: SMatrixView<f32, ACCELERATION_MODEL_SIGMA_ORDER,
       MEASUREMENT_SIZE>,
) -> SMatrix<f32, 1, MEASUREMENT_SIZE> {
    let column = sigmas.column(SCAN_THETA).transpose();

    let sum_sin = weights.dot(&column.map(f32::sin));
    let sum_cos = weights.dot(&column.map(f32::cos));

    let lidar_theta = f32::atan2(sum_sin, sum_cos);

    let column = sigmas.column(IMU_THETA).transpose();

    let sum_sin = weights.dot(&column.map(f32::sin));
    let sum_cos = weights.dot(&column.map(f32::cos));

    let imu_theta = f32::atan2(sum_sin, sum_cos);

    let x = weights.dot(&sigmas.column(SCAN_X).transpose());
```

```
    let y = weights.dot(&sigmas.column(SCAN_Y).transpose());

    let gps_x = weights.dot(&sigmas.column(GPS_X).transpose());
    let gps_y = weights.dot(&sigmas.column(GPS_Y).transpose());

    SMatrix::from_row_slice(&[
        x,
        y,
        lidar_theta,
        imu_theta,
        imu_v_theta,
        imu_a_x,
        imu_a_y,
        gps_x,
        gps_y,
        compass_theta,
    ])
}
```

4.5 Локальный планировщик

Формирование минимальной скорости основывается на тенденции к застреванию:

```
let mut estimate_min_velocity = || -> Result<f32,
   LocalPlanError> {
    if rotation.abs() > config.stuck.rotation_tolerance
        || dist > *config.stuck.distance_tolerance
    {
        world.tracker.reset_stuck();
        return Ok(default_min_velocity);
    }
    let stuck_duration = world.tracker.detect_stuck();
    if stuck_duration < config.stuck.stuck_time_tolerance {
        return Ok(default_min_velocity);
    }
    if stuck_duration > config.stuck.max_stuck_time {
        return Err(LocalPlanError::StuckDetected);
    }

    let elapsed = stuck_duration;
    let max_elapsed = config.stuck.max_stuck_time;
    let v = config.robot.min_velocity
        + (config.robot.max_velocity - config.robot.min_velocity)
            * elapsed.as_secs_f32()
            / max_elapsed.as_secs_f32();

    if stuck_duration > config.stuck.stuck_time_force_backward {
        force_back_movement = true;
        Ok(v)
    } else {
        Ok(config.robot.max_velocity)
    }
```

```
};
```

Оценка максимальной скорости происходит по:

```
let estimate_max_velocity = |state: &RobotState| -> f32 {
    let dist_to_goal =
        (world.target_pose.coords -
           state.position.coords).norm();
    let stop_time = dist_to_goal / config.robot.max_velocity;
    if stop_time <= limits.deceleration_time.as_secs_f32() {
        //slow deceleration
        let y = limits.deceleration_time.as_secs_f32();
        let base_pow = -y.log10() / y;
        let pow =
            base_pow * (limits.deceleration_time.as_secs_f32() -
               stop_time);
        let output = f32::exp(pow) * config.robot.max_velocity;
        output.clamp(config.robot.min_velocity,
           config.robot.max_velocity)
    } else if motion_time < limits.acceleration_time {
        //slow acceleration
        let x = motion_time.as_secs_f32();
        let a = x / limits.acceleration_time.as_secs_f32();
        f32::max(v_min, a.powi(3) * config.robot.max_velocity)
    } else {
        config.robot.max_velocity
    }
};
```

Оценка максимальной угловой скорости происходит (с задаваемой табличной функции $\omega_{max} = F(v)$):

```
impl MovementOptimization {
    pub fn estimate_angular(&self, velocity: f32) -> f32 {
        let model = self.model.as_ref().unwrap();

        model
            .predict(&DenseMatrix::from_2d_vec(&vec![vec![velocity]]).u
            .unwrap()[0]
    }
}

let estimate_max_angular = move |state: &RobotState| -> f32 {
    let v_local =
       state.velocity.into_local(state.position.heading_rad);
    optimization
        .estimate_angular(v_local.translation.x)
        .clamp(0.0, max_angular)
};
```

Метрика выравнивания по углу в конечной точке:

```
pub fn target_alignment(&self, target_heading: f32) ->
   TrajectoryMetric {
    Box::new(move |trajectory| {
```

```
        let rotation = normalize_angle(
            trajectory.finish_state().position.heading_rad -
               target_heading,
        );
        let cost = rotation.abs() / f32::consts::FRAC_PI_2;
        Ok((MetricKind::TargetAlignment, MetricCost::new(cost)))
    })
}
```

Метрика кратчайшего пути

```
pub fn shortest_path_metric(&self, goal: Point2<f32>) ->
   TrajectoryMetric {
    Box::new(move |trajectory| {
        let dist2 = (goal -
           trajectory.finish_state().position.coords)
            .norm_squared();

        Ok((MetricKind::ShortestPath, MetricCost::new(dist2)))
    })
}
```

Метрика коллизии:

```
pub fn collision_metrics(
    &self,
    obstacles: &'a DynamicObstacles,
    map_chunk: &'a MapChunk,
) -> TrajectoryMetric {
    let robot = self.robot;
    let config = self.planner_config;
    Box::new(move |trajectory| {
        for state in trajectory.states.iter() {
            let state_shape = robot.build_shape(
                map_chunk.resolution,
                config.collision_distance,
                &(*state).into(),
            );
            if has_dynamic_collision(&state_shape, obstacles)
                || has_static_collision(&state_shape, map_chunk)
            {
                //collisions are not acceptable
                return Err(MetricError::Unreacheable);
            }
        }
        Ok((MetricKind::Collision, MetricCost::ZERO))
    })
}
```

Для расчёта метрики удалённости от коллизий в позиции можно заранее просчитать расстояние до ближайшего препятствия в каждой точке, что превратит вычислительную задачу в поисковую.

Метрика удалённости от коллизий:

```
fn collision_map_metric(
    &'a self,
```

```
    map: &'a DistanceGrid,
    maybe_obstacles: Option<&'a DynamicObstacles>,
) -> TrajectoryMetric<'a> {
    Box::new(move |trajectory| {
        let state = trajectory.finish_state();
        let mut min_dist = f32::MAX;
        let coords = state.position.coords;
        let dist_cost = map.index_distance(&coords);
        if dist_cost.is_reacheable() {
            let dist =
               map.resolution.to_metres(dist_cost.as_f32());
            min_dist = f32::min(min_dist, dist);
        }
        if let Some(obstacles) = maybe_obstacles {
            let obstacle = obstacles.closest(&coords);
            let dist_to_dynamic_obstacle = (obstacle -
               coords).norm();
            min_dist = f32::min(min_dist,
               dist_to_dynamic_obstacle);
        }
        Ok((
            MetricKind::CollisionDistance,
            MetricCost::new(1.0 / min_dist),
        ))
    })
}
```

Метрика близости траектории к глобальному маршруту:

```
fn distance_map_metric(
    &'a self,
    distance_grid: &'a DistanceGrid,
) -> TrajectoryMetric<'a> {
    Box::new(move |trajectory| {
        let states = &trajectory.states;
        let mut distance_cost = 0.0;
        for state in states.iter() {
            let world_point = state.position.coords;
            let distance =
               distance_grid.index_distance(&world_point);
            if distance.is_unreacheable() {
                return Err(MetricError::Unreacheable);
            }
            distance_cost += distance.as_f32();
        }
        Ok((MetricsKind::GlobalPath,
           MetricCost::new(distance_cost)))
    })
}
```

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Инструкция по установке RUST

5.1.1 Установка на Windows

Перейдите на страницу `https://www.rust-lang.org/tools/install` и следуйте инструкциям по установке. Установщик предложит выбрать одну из опций. Выберите опцию 1, чтобы установить Rust со стандартными настройками.

5.1.2 Установка на macOS/Linux

Откройте терминал и выполните следующую команду:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

В процессе установки rustup предложит выбрать один из вариантов. Для стандартной установки выберите опцию 1.

5.2 Проверка установки

После установки Rust, закройте и снова откройте терминал, чтобы обновить переменную среды PATH. Чтобы проверить, правильно ли установлен Rust, введите следующую команду: `ustc --vesion`

Вы должны увидеть версию Rust, хэш коммита, дату коммита и дату сборки.

5.3 Управляющие команды

Конфигурация определяет несколько переменных окружения для управления поведением задач и настройками проекта. Эти переменные используются в различных задачах для настройки путей, режимов и параметров развертывания.

TODO: Почистить всё это

– compress: Создает архивы драйверов для выбранного ROBO_MODE. Требует установки ROBO_MODE и наличия скрипта compress.sh в директории scripts. Выполняет скрипт compress.sh.

– cross: Собирает бинарный файл проекта для указанной целевой архитектуры. Зависит от задачи setup_features. Использует команду cross с режимом release, именем бинарного файла проекта, без стандартных функций и с дополнительными флагами функций на основе FEATURE_NAME.

– compress_driver: Сжимает файлы, связанные с драйверами, для вы-

бранного ROBO_MODE. Требует установки ROBO_MODE и наличия скрипта compress_driver.sh. Выполняет скрипт compress_driver.sh.

– copy_binary: Копирует скомпилированный бинарный файл в указанный путь. Требует установки COPY_TO_PATH. Копирует бинарный файл из директории release целевой архитектуры в указанный путь.

– copy_config: Копирует конфигурационные файлы в указанный путь. Требует установки CONFIG_FILES и COPY_TO_PATH. Перебирает CONFIG_FILES и копирует каждый в место назначения.

– copy_scripts: Копирует скриптовые файлы в указанный путь. Требует установки SCRIPT_FILES и COPY_TO_PATH. Копирует каждый скриптовый файл и выводит BUILD_FLAGS для отладки.

– rename_zip: Переименовывает zip-файлы, добавляя суффикс. Требует установки SUFFIX. Переименовывает все .zip файлы в текущей директории, добавляя значение SUFFIX.

– setup_features: Настраивает флаги функций для задач, которые их требуют. Требует установки FEATURE_NAME. Устанавливает переменную окружения FEATURE_FLAG в –features.

– sync_config: Получает конфигурационные файлы с удаленного робота. Требует установки CONFIG_FILES. Использует scp для копирования IcpConfig.toml из директории ROBO_HOME робота в локальную директорию config/ROBO_MODE через указанный DEPLOY_PORT.

– copy_config_deploy: Развертывает конфигурационные файлы на удаленный робот. Требует установки CONFIG_FILES. Использует scp для копирования CONFIG_FILES в директорию ROBO_HOME робота.

– run_oneshot_script: Запускает одноразовый скрипт на удаленном роботе после развертывания. Требует установки ONESHOT_SCRIPT. Копирует скрипт на робот, выполняет его через ssh и удаляет после выполнения.

– copy_binary_deploy: Развертывает скомпилированный бинарный файл на удаленный робот. Зависит от copy_config_deploy и run_oneshot_-script. Останавливает соответствующую systemd-службу, копирует бинарный файл в ROBO_HOME и использует указанный DEPLOY_PORT и SSH-ключ.

– scripts_deploy: Развертывает файл службы systemd для робота. Требует наличия файла службы в директории robot_deployment/ROBO_MODE. Копирует файл службы в /etc/systemd/system на роботе, перезагружает демон systemd, включает и перезапускает службу.

– deploy: Оркестрирует процесс развертывания. Выполняет задачи cross, copy_binary_deploy и scripts_deploy последовательно для сборки и развертывания бинарного файла на удаленный робот.

– test_single_default: Запускает одиночный тестовый сценарий парал-

лельно. Зависит от setup_features и запускает задачу solo_no_move в параллельном режиме с форком.

– solo_no_move: Запускает тестовую симуляцию для сценария solo_no_move. Устанавливает переменные окружения SCENARIO_PATH и WORK_DIR и выполняет cargo run –bin test_sim в директории test_sim.

– gui: Запускает тестовый GUI в режиме release. Выполняет cargo run –release –bin test_gui в директории tools/test_gui.

– gui_debug: Запускает тестовый GUI в режиме отладки. Выполняет cargo run –bin test_gui в директории tools/test_gui.

– run_webots_app: Запускает приложение Webots. Выполняет команду webots из директории WEBOTS_HOME.

– run_mock_driver: Запускает имитацию аппаратного контроллера Webots в режиме release. Выполняет cargo run –bin hardware_mock_webots_controller –release в директории hardware_mock_webots_controller.

– run_mock_robot: Запускает приложение Webots и имитацию драйвера последовательно. Запускает webots, ждет 5 секунд, затем запускает имитацию драйвера.

– stop_mock_robot: Останавливает приложение Webots и имитацию драйвера. Завершает все запущенные процессы webots и hardware_mock_webots_controller.

– control: Запускает бинарный файл roboq_service в режиме release с функцией eureka. Выполняет cargo run –release –bin roboq_service –no-default-features –features eureka.

– brains: Запускает бинарный файл roboporter в режиме release с функцией simulation. Устанавливает переменные окружения ICP_CONFIG и DEVICE_CONFIG и выполняет cargo run –bin roboporter –release –no-default-features –features simulation в директории roboporter.

6 ТЕСТИРОВАНИЕ РАБОТОСПОСОБНОСТИ ПРОГРАММНОГО СРЕДСТВА

6.1 Тестирование функции построения карты

Тестирование функции построения карты: Изначально тестирование ПО производилось в симуляции на ПК. В качестве симулятора была выбрана среда WeBots из-за лёгкости интеграции языка программирования Rust.

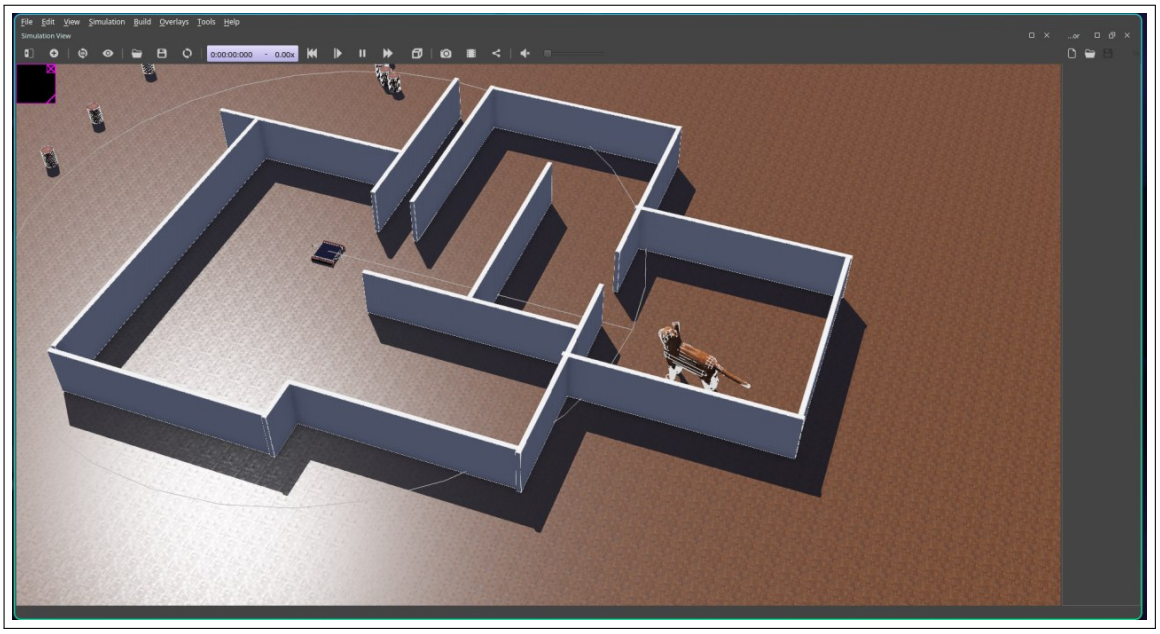

Рисунок 6.1 – Среда симуляции WeBots

6.2 Тестирование базовой навигации

Цель: убедиться, что робот может перемещаться из точки A в точку B по прямой линии без препятствий. Метод: В симуляторе Webots была создана простая среда без объектов, и роботу была задана задача достичь целевой точки. Результат: Робот успешно достиг целевой точки, продемонстрировав корректную работу датчиков и базового алгоритма движения.

6.3 Тестирование при наличии статических препятствий

Цель: Проверить, как алгоритм строит маршрут, обходя неподвижные объекты. Метод: В среде были размещены стены и коробки, и роботу была задана задача их обойти. Результат: Алгоритм точно обнаружил препятствия и спланировал безопасный путь, избегая столкновений.

6.4 Тестирование при наличии динамических препятствий

Цель: Оценить адаптацию алгоритма к движущимся объектам. Метод: В симуляцию были добавлены движущиеся объекты, и была проверена реакция робота. Результат: Робот успешно избегал столкновений с движущимися объектами, демонстрируя быструю реакцию на изменения.

6.5 Тестирование в сложных средах

Цель: Проверить работу алгоритма в запутанных пространствах. Метод: Была создана карта с множеством поворотов и тупиков. Результат: Алгоритм нашел оптимальный путь и избежал зацикливания, что подтверждает его эффективность в сложных условиях.

6.6 Тестирование при сбоях датчиков

Цель: Оценить устойчивость алгоритма к неточным данным. Метод: В симуляции были смоделированы отказы датчиков и добавлен шум к их показаниям. Результат: Алгоритм продолжил выполнение задачи, справившись с ошибками и неточностями данных.

6.7 Тестирование локализации

Цель: Проверить точность определения местоположения робота на карте. Метод: Были использованы алгоритмы SLAM для построения карты и локализации. Результат: Карта была построена с высокой точностью, и робот успешно корректировал свою позицию при ошибках.

Все проведенные тесты были успешно пройдены, что подтверждает высокую эффективность, надежность и адаптивность разработанного алгоритма построения карты и нахождения пути. Результаты тестирования позволяют рекомендовать данный алгоритм для использования в реальных условиях.

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И ИСПОЛЬЗОВАНИЯ ПРОГРАММНОГО СРЕДСТВА НАВИГАЦИИ МОБИЛЬНЫХ СИСТЕМ

7.1 Характеристика программного средства

Программное средство навигации мобильных систем осуществляет задачу перемещения и определения местоположения мобильной системы, построение и исполнение маршрута с использованием сенсоров и приводов. ПС оптимизировано для навигации голономных колёсных роботов. Предполагается что мобильная система управляется через отправку команды установки угловой и линейной скорости. Также необходима конфигурация под размеры и движение каждого определённого робота.

ПС выполняет следующие функции:

– сбор данных с датчиков;
– расчёт текущей позиции;
– построение карты;
– сохранение и загрузка карты;
– планирование маршрута;
– планирование движения;
– исполнение маршрута, учитывая динамические препятствия.

В сравнении с ROS, который является наиболее популярным аналогом, ПС упрощает развёртывание, требует меньше вычислительных ресурсов за счёт минимизации затрат на общении модулей путём расположения их в одном процессе операционной системы, что позволяет использовать менее мощное аппаратное обеспечение.

ПС получает данные с датчиков, информацию о цели которой ей необходимо достигнуть и отправляет управляющие сигналы на ходовую часть. Решается задача локализации, построения маршрута и выполнения маршрута к заданной точке.

7.2 Расчёты затрат на разработку программного средства

Расчет затрат на разработку ПО производится в разрезе следующих статей затрат:

– затраты на основную заработную плату разработчиков;
– затраты на дополнительную заработную плату разработчиков;
– отчисления на социальные службы;
– прочие затраты (амортизационные отчисления, расходы на электро-

энергию, командировочные расходы, арендная плата за офисные помещения и оборудование, расходы на управление и реализацию и т. п.).

Расчёт основной заработной платы осуществляется по формуле

$$З_o = К_{пр} \sum_{i=0}^{n} З_{чi} \cdot t_i \, , \tag{7.1}$$

где n – категории исполнителей, занятых разработкой программного средства;

$К_{пр}$ - коэффициент премий и иных стимулирующих выплат (1,3);

$З_ч$ – Часовой оклад исполнителя i-й категории, р.;

t – трудоёмкость работ, выполняемых исполнителем i-й категории, ч.

Затраты на основную заработную плату команды разработчиков делятся исходя из численности, состава команды (категорий исполнителей), размеров месячной заработной платы каждого из участников команды, а также общей трудоёмкости разработки ПО.

Согласно постановлению Министерства труда и социальной защиты Республики Беларусь от 15 ноября 2024 г. №67 «Об установлении расчетной нормы рабочего времени на 2024 год» при полной норме продолжительности рабочего времени на 2025 год для пятидневной рабочей недели с выходными днями в субботу и воскресенье расчетная норма рабочего времени составит 2007 ч. На основании этих данных среднее количество рабочих ч. в месяце принято равным 167 ч.

Трудоёмкость определялась на основе сложности разработки программного средства, объема функций. За основу в том числе брались фактические значения трудоёмкости работ при разработке ПО со схожим функционалом в месте прохождения преддипломной практики.

Для расчёта возьмём размер премии 20%.

На основании плановых данных был выполнен расчет основной заработной платы команды разработчиков, результаты которого приведены в таблице 7.1.

Таблица 7.1 – Расчёт основной заработной платы команды разработчиков

Наименование должности разработчика	Вид выполненной работы	Месячная заработная плата, р.	Часовая заработная плата, р.	Трудоёмкость работ, ч	Сумма, р.
Руководитель проекта	Координация работы, контроль сроков и этапов разработки	2500	14,97	120	1796,40
Инженер-программист	Разработка программного средства	2700	16,17	458	7405,86
Специалист по тестированию программного обеспечения	Тестирование программного средства	2100	12,57	200	2514,00
Итого					11 716,26
Премия (20%)					2343,25
Общая сумма затрат на разработку					14 059,51

Расчёт затрат на дополнительную заработную плату команды разработчиков.

Затраты на дополнительную заработную плату команды разработчиков включают выплаты, предусмотренные законодательство о труде (оплата трудовых отпусков, льготных ч., времени выполнения государственных обязанностей и других выплат, не связанных с основной деятельностью исполнителей), и определяются по формуле

$$З_д = \frac{З_о \cdot Н_д}{100}, \quad (7.2)$$

где $З_о$ – затраты на основную заработную плату;

$Н_д$ – норматив дополнительной заработной платы (15%).

Дополнительная заработная плата составит

$$З_о = \frac{14\,059{,}51 \cdot 15}{100} = 2108{,}93 \text{ р.} \quad (7.3)$$

Отчисления на социальные нужды определяются по формуле

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100}, \quad (7.4)$$

где $Н_{соц}$ – норматив отчислений от фонда оплаты труда (35%).

Отчисления на социальные нужды составят

$$Р_{соц} = \frac{(14\,059{,}51 + 2108{,}93) \cdot 35}{100} = 5658{,}95 \text{ р.} \quad (7.5)$$

Прочие затраты рассчитываются по формуле

$$Р_{пз} = \frac{З_о \cdot Н_{пз}}{100}, \quad (7.6)$$

где $Н_{пз}$ – норматив прочих затрат, 35%.

Прочие затраты составят

$$Р_{пз} = \frac{14\,059{,}51 \cdot 35}{100} = 4920{,}83 \text{ р.} \quad (7.7)$$

Общая сумма затрат на разработку рассчитывается по формуле

$$З_{общ} = З_о + З_д + Р_{соц} + Р_{пз}. \quad (7.8)$$

Расчёт затрат на разработку программного продукта предоставлен в таблице 7.2

Таблица 7.2 – Затраты на разработку программного обеспечения

Наименование статьи затрат	Значение, р.
1. Основная заработная плата разработчиков	14 059,51
2. Дополнительная заработная плата разработчиков	2108,93
3. Отчисления на социальные нужды	5658,95
4. Прочие затраты	4920,83
Общая сумма инвестиций в разработку	26 748,22

7.3 Экономический эффект от разработки программного обеспечения и применения программного обеспечения для собственных нужд

В общем виде экономический эффект при использовании ПО рассчитывается по формуле по формуле

$$\Delta П_ч = (Э_з - И_{разр} - \Delta З_{тек}) \cdot (1 - \frac{Н_п}{100}), \quad (7.9)$$

где $Э_з$ – экономия текущих затрат, полученная в результате применения ПО, р.;

$И_{разр}$ – затраты на разработку программного обеспечения, р.

$\Delta З_{тек}$ – прирост текущих затрат, связанных с поддержкой и сопровождением ПО, р.;

$Н_п$ – ставка налога на прибыль согласно действующему законодательству (20%).

Прирост текущих затрат, связанных с сопровождением и поддержкой ПО, примем за 10% от затрат на разработку ПО, что составит

$$З_{тек} = 26\,748{,}22 \cdot \frac{10}{100} = 2674{,}82 \text{ р.} \quad (7.10)$$

Использование данного программного средства позволяет использовать более дешёвое аппаратное обеспечение. Так как навигация и SLAM являются ресурсоёмкими операциями, обычно используют компьютер NVIDIA Jetson Nano, стоимостью 1421,83 р., в то время как ПС позволяет использовать Banana Pi CM4, стоимостью 300,12 р.

Это позволяет экономить 1121,71 р. на единицу продукции. Если взять в расчёт что в год производится 40 мобильных систем, получаем экономию текущих затрат в 44 868,40 р.

Экономический эффект для организации-заказчика при использовании ПО и выпуске партии в 40 единиц составляет

$$\Delta П_ч = (44\,868{,}40 - 26\,748{,}22 - 2674{,}82) \cdot (1 - \frac{20}{100}) = 12\,356{,}29 \text{ р.} \quad (7.11)$$

Уровень рентабельность затрат рассчитывается по формуле

$$У_р = \frac{\Delta П_ч}{И_{разр}} \cdot 100, \quad (7.12)$$

уровень рентабельности составляет

$$У_р = \frac{12\,356{,}29}{26\,748{,}22} \cdot 100 = 46{,}19\% . \quad (7.13)$$

В результате расчёта были получены следующие показатели (см. табл. 7.3)

Таблица 7.3 – Экономические показатели

Наименование показателя	Значение
Прогнозируемая сумма затрат на разработку программного продукта	26 748,22 р.
Прирост чистой прибыли	12 356,29 р.
Рентабельность инвестиций	46,19%

Средняя процентная ставка по банковским депозитным вкладам на январь 2025-го г. не превышает 13,76% [7], рентабельность инвестиций в проект составляет 46,19%. Инвестиции в разработку проекта окупятся за первый год реализации проекта. Это означает, что данный проект программного средства навигации мобильных систем является экономический эффективным, разработка и последующая продажа программного продукта являются экономически целесообразными.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта был разработан программный инструмент на языке программирования Rust для навигации мобильных систем. Основной задачей разработки было создание эффективного и минималистичного инструмента для построения карт и обеспечения навигации при минимизации зависимостей от сторонних библиотек и операционных систем, что повышает переносимость и надежность решения.

В рамках реализации программного обеспечения были успешно реализованы следующие ключевые функциональные модули:

– Алгоритмы совмещения сканов, включающие в себя методы Iterative Closest Point (ICP), Corellative Scan Matcher (CSM) и Multi-Scale Corellative Scan Matcher (MSCM), обеспечивающие точное и надежное создание локальных карт на основе данных сенсоров;

– алгоритмы глобальной навигации, позволяющие планировать маршруты и обеспечивать ориентацию мобильной платформы в пространстве с учетом глобальной карты;

– алгоритмы локальной навигации, направленные на обеспечение безопасного и эффективного движения робота в непосредственной близости от препятствий и динамических объектов.

В процессе работы был проведен комплексный анализ предметной области, включающий обзор и критический анализ существующих решений и технологий в области картографии и навигации мобильных роботов. Это позволило выявить основные недостатки и ограничения существующих библиотек и определить направления для оптимизации и инноваций.

Результаты работы подтвердили достижение всех поставленных целей и задач дипломного проекта. Полученное программное средство демонстрирует высокую производительность, надежность и гибкость, что делает его перспективным инструментом для применения в различных робототехнических системах.

В качестве направлений дальнейшего развития проекта рассматривается оптимизация программного кода с целью повышения производительности и снижения ресурсопотребления, а также расширение функциональности за счет внедрения дополнительных алгоритмов совмещения сканов и навигации.

Таким образом, завершенный проект вносит значительный вклад в область разработки программного обеспечения для робототехники и открывает новые возможности для создания высокоэффективных и надежных автономных навигационных систем на базе языка Rust.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Software engineering research on the Robot Operating System: A systematic mapping study / Michel Albonico, Milica Đorđević, Engel Hamer, Ivano Malavolta // Journal of Systems and Software. – 2023. – Vol. 197. – 111574 P.

[2] ROS: an open-source Robot Operating System / Morgan Quigley [et al.] // ICRA workshop on open source software / Kobe. – Vol. 3. – 2009. – 5 P.

[3] Metta, Giorgio. YARP: yet another robot platform / Giorgio Metta, Paul Fitzpatrick, Lorenzo Natale // International Journal of Advanced Robotic Systems. – 2006. – Vol. 3, no. 1. – 8 P.

[4] The Marathon 2: A Navigation System / Steven Macenski, Francisco Martin, Ruffin White, Jonatan Ginés Clavero // 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). – 2020.

[5] Macenski, Steve. SLAM Toolbox: SLAM for the dynamic world / Steve Macenski, Ivona Jambrecic // Journal of Open Source Software. – 2021. – Vol. 6, no. 61. – 2783 P.

[6] Grisetti, Giorgio. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling / Giorgio Grisetti, Cyrill Stachniss, Wolfram Burgard // Proceedings of the 2005 IEEE international conference on robotics and automation / IEEE. – 2005. – Pp. 2432–2437.

[7] Динамика ставок кредитно-депозитного рынка [Электронный ресурс]. – 2025. – Режим доступа: https://www.nbrb.by/statistics/creditdepositmarketrates.

ПРИЛОЖЕНИЕ А (ОБЯЗАТЕЛЬНОЕ) ТЕКСТ ПРОГРАММНОГО МОДУЛЯ TODO

```
==> ./config/mod.rs <==
mod world_map;

use std::time::Duration;

use roboq_types::{
    from_variant,
    model::{
        localization::{LocalizationParams, MatchingParams,
           NoiseParams},
        map::MapParams,
        plan::{MetricsParams, PlannerParams, StuckParams},
        robot_params::RobotParams2D,
        Metres,
    },
};
use serde::{Deserialize, Serialize};

use crate::{
    icp::icp_slam::{IcpSlamConfig, IcpSlamInitConfig},
    planner::PlannerSimulationConfig,
};

pub use roboq_types::model::localization::SensorParams;

#[cfg(feature = "localization")]
pub use crate::localization::MotionEstimationConfig;

pub use crate::planner::RobotConfig;
pub use world_map::WorldMapConfig;

#[derive(Debug, Clone)]
pub enum ConfigProperty {
    UpdateMap(bool),
    UseCollisions(bool),
    UseDebug(bool),
    /// the maximal period of time
    /// after which robot will be stopped
    /// if no more commands were supplied
    ReactionTimeout(Duration),
    CollisionDistance(Metres),

    MapParams(MapParams),
    RobotParams(RobotParams2D),
    NoiseParams(NoiseParams),
    LocalizationParams(LocalizationParams),
    MatchingParams(MatchingParams),
    SensorParams(SensorParams),

    //planner related params
    PlannerParams(PlannerParams),
    StuckParams(StuckParams),
```

```
    MetricsParams(MetricsParams),
}

from_variant!(ConfigProperty, SensorParams);
from_variant!(ConfigProperty, MapParams);
from_variant!(ConfigProperty, NoiseParams);
from_variant!(ConfigProperty, PlannerParams);
from_variant!(ConfigProperty, LocalizationParams);
from_variant!(ConfigProperty, MatchingParams);
from_variant!(ConfigProperty, StuckParams);
from_variant!(ConfigProperty, MetricsParams);

#[derive(Default, Clone, Serialize, Deserialize)]
pub struct DebugConfig {
    pub enabled: bool,
}

#[derive(Default, Clone, Serialize, Deserialize)]
pub struct NavigationConfig {
    pub icp_slam_init_config: IcpSlamInitConfig,
    //base icp_slam_config
    pub icp_slam_config: IcpSlamConfig,
    pub planner_config: PlannerSimulationConfig,

    #[cfg(all(feature = "localization", feature = "gps"))]
    pub world_map_config: WorldMapConfig,

    #[cfg(feature = "localization")]
    pub estimation_config: MotionEstimationConfig,

    pub debug_config: DebugConfig,
    #[cfg(feature = "localization")]
    pub devices_config: SensorParams,

    #[serde(default)]
    pub matching_config: MatchingParams,
}

impl NavigationConfig {
    pub fn apply_property(&mut self, property: ConfigProperty) {
        let config = self;

        match property {
            ConfigProperty::UpdateMap(updating_map) => {
                config.icp_slam_config.updating_map =
                   updating_map;
            }

            ConfigProperty::UseCollisions(use_collisions) => {
                config.planner_config.use_collisions =
                   use_collisions;
            }

            ConfigProperty::UseDebug(enabled) => {
                config.debug_config.enabled = enabled;
            }
```

```
            ConfigProperty::RobotParams(params) => {
                config.planner_config.robot.update(params);
            }

            ConfigProperty::ReactionTimeout(timeout) => {
                config.planner_config.idle_timeout_ms =
                    timeout.as_millis() as u64;
            }

            ConfigProperty::CollisionDistance(distance) => {
                config.planner_config.collision_distance =
                   distance;
            }

            ConfigProperty::PlannerParams(params) => {
                config.planner_config.update(params);
            }

            ConfigProperty::NoiseParams(params) => {
                config.estimation_config.update_noise(params);
            }

            ConfigProperty::MapParams(params) => {
                #[cfg(feature = "gps")]
                config.world_map_config.update(params);

                #[cfg(not(feature = "gps"))]
                drop(params);
            }

            ConfigProperty::LocalizationParams(params) => {
                config.estimation_config.update(params);
            }

            ConfigProperty::MatchingParams(params) => {
                config.matching_config = params;
            }

            ConfigProperty::SensorParams(params) => {
                config.devices_config = params;
            }

            ConfigProperty::StuckParams(params) => {
                config.planner_config.stuck = params;
            }

            ConfigProperty::MetricsParams(params) => {
                config.planner_config.metrics = params;
            }
        }
    }

    pub fn to_properties(&self) -> Vec<ConfigProperty> {
        vec![
```

```
                ConfigProperty::UpdateMap(
                    self.icp_slam_config.updating_map),
                ConfigProperty::UseCollisions(
                    self.planner_config.use_collisions),
                ConfigProperty::UseDebug(self.debug_config.enabled),
                ConfigProperty::ReactionTimeout(Duration::from_millis
                    (
                     self.planner_config.idle_timeout_ms,
                )),
                ConfigProperty::RobotParams(
                     self.planner_config.robot.clone().into(),
                ),
                ConfigProperty::CollisionDistance(
                     self.planner_config.collision_distance,
                ),
                ConfigProperty::PlannerParams((&self.planner_config)
                    .into()),
                ConfigProperty::StuckParams(
                    self.planner_config.stuck.clone()),
                ConfigProperty::MatchingParams(
                    self.matching_config.clone()),
                #[cfg(feature = "localization")]
                ConfigProperty::SensorParams(
                    self.devices_config.clone()),
                ConfigProperty::MetricsParams(
                    self.planner_config.metrics.clone()),
                #[cfg(feature = "localization")]
                ConfigProperty::NoiseParams(
                    self.estimation_config.noise_params()),
                #[cfg(not(feature = "localization"))]
                ConfigProperty::NoiseParams(Default::default()),
                #[cfg(feature = "gps")]
                ConfigProperty::MapParams((&self.world_map_config)
                    .into()),
                #[cfg(feature = "localization")]
                ConfigProperty::LocalizationParams(
                    self.estimation_config.params()),
                #[cfg(not(feature = "localization"))]
                ConfigProperty::LocalizationParams(Default::default()
                    ),
            ]
        }
}

==> ./config/world_map.rs <==
use std::time::Duration;

use roboq_types::model::map::MapParams;

#[derive(Clone, serde::Serialize, serde::Deserialize, o2o::o2o)]
#[from(MapParams)]
#[into(MapParams)]
pub struct WorldMapConfig {
    /// Should we use predefined zone or not
```

```
    #[serde(default)]
    pub utm_zone: Option<u8>,

    #[serde(default = "default_utm_threshold")]
    pub utm_threshold: f64,

    pub map_rotation: f64,

    #[serde(
        serialize_with = "roboq_types::serde::serialize_millis",
        deserialize_with = "roboq_types::serde::
           deserialize_millis",
        default = "default_coords_lifetime"
    )]
    pub coords_lifetime: Duration,
}

impl WorldMapConfig {
    pub fn update(&mut self, params: MapParams) {
        *self = params.into();
    }
}

fn default_utm_threshold() -> f64 {
    10000.0
}

fn default_coords_lifetime() -> Duration {
    Duration::from_millis(1_000)
}

impl Default for WorldMapConfig {
    fn default() -> Self {
        Self {
            coords_lifetime: default_coords_lifetime(),
            utm_threshold: default_utm_threshold(),
            map_rotation: 0.0,
            utm_zone: None,
        }
    }
}

==> ./correlative_scan_matching/correlation_grid.rs <==
use nalgebra::{Point2, Vector2};
use roboq_types::model::{map::MetresPerPixel, Metres};
use serde::{Deserialize, Serialize};

use super::localized_range_scan::LocalizedRangeScan;
use crate::model::array2d::Array2D;

#[derive(Clone, Serialize, Deserialize)]
pub struct CorrelationGrid {
    pub grid: Array2D<u8>,
    pub resolution: MetresPerPixel,
    pub size_metres: f32,
    pub top_left_corner: Vector2<f32>,
```

```
    pub gauss_blur: Array2D<u8>,
}

impl CorrelationGrid {
    pub fn new(
        size_metres: f32,
        resolution: MetresPerPixel,
        pos: Vector2<f32>,
        std_deviation: Metres,
    ) -> Self {
        let correlation_grid_size_pixels =
            resolution.to_pixels(Metres(size_metres)) as u32;

        let kernel_size = 2 * resolution.to_pixels(std_deviation
           * 2.0) + 1;

        let gauss_blur = new_gaussian_blur_kernel(
            kernel_size as u32,
            &resolution,
            std_deviation,
        );

        CorrelationGrid {
            resolution,
            size_metres,
            grid: Array2D::new_square(
               correlation_grid_size_pixels),
            top_left_corner: pos,
            gauss_blur,
        }
    }

    /// Reset probability lookup grid
    pub fn clear(&mut self) {
        self.grid.fill(0);
    }

    pub fn set_center(&mut self, center: Vector2<f32>) {
        let grid_size_metres =
            self.resolution.vector_to_metres(&self.grid.size());
        self.top_left_corner = center - grid_size_metres / 2.0;
    }

    pub fn add_scan(&mut self, scan: &LocalizedRangeScan) {
        for point in &scan.points {
            self.add_point(point);
        }
    }

    ///Add point to grid applying gaussian blur
    ///to estimate pose probability for this one
    pub fn add_point(&mut self, point: &Point2<f32>) {
        let point_on_grid = self
            .resolution
            .point_to_pixels(&(point - self.top_left_corner));
```

```
        let kernel_size = self.gauss_blur.get_width();
        let radius = (kernel_size - 1) / 2;

        let top_left_kernel_on_grid =
            point_on_grid - Vector2::new(radius, radius);

        if top_left_kernel_on_grid.x < 0
            || top_left_kernel_on_grid.y < 0
            || top_left_kernel_on_grid.x > (self.grid.get_width()
                - kernel_size)
            || top_left_kernel_on_grid.y > (self.grid.get_width()
                - kernel_size)
        {
            return;
        }

        for x in 0..kernel_size {
            for y in 0..kernel_size {
                let kernel_weight =
                    self.gauss_blur.index_grid(&Point2::new(x, y)
                        );

                let grid_weight = self.grid.index_grid_mut(
                    &(top_left_kernel_on_grid + Vector2::new(x, y
                        )),
                );

                *grid_weight = (*grid_weight).max(*kernel_weight)
                    ;
            }
        }
    }
}

fn new_gaussian_blur_kernel(
    size: u32,
    resolution: &MetresPerPixel,
    std_deviation: Metres,
) -> Array2D<u8> {
    let mut kernel = Array2D::new_square(size);

    let kernel_size = size as i32;
    let sigma = *std_deviation;

    for x in 0..kernel_size {
        for y in 0..kernel_size {
            let x_dist = (x - kernel_size / 2) as f32;
            let y_dist = (y - kernel_size / 2) as f32;

            let distance = resolution
                .to_metres((x_dist * x_dist + y_dist * y_dist)
                    .sqrt());

            let z = ((-0.5f32) * (distance / sigma).powi(2)).exp
                ();
```

```
                *kernel.index_grid_mut(&Point2::new(x, y)) = (z * 255
                    .0) as u8;
            }
        }

        kernel
    }

    ==> ./correlative_scan_matching/localized_range_scan.rs <==
    use nalgebra::Point2;
    use serde::{Deserialize, Serialize};

    use crate::{icp::icp_svd::PointCloud, model::pose::Pose2D};

    #[derive(Debug, Clone, Default, Serialize, Deserialize)]
    pub struct LocalizedRangeScan {
        /// All points in LocalizedRangeScan are already in the
            position of robots pose
        /// changing robot pose doesn't change the offset of the
            points, and points are not stored in
        /// respect to the origin.
        pub points: Vec<Point2<f32>>,
        /// Center of the scan pose with rotation.
        pub pose: Pose2D,
    }

    impl LocalizedRangeScan {
        pub fn cloud_at_position(
            point_cloud: PointCloud,
            robot_pose: Pose2D,
        ) -> Self {
            let transformation = robot_pose.as_isometry();

            let points: Vec<_> = point_cloud
                .into_iter()
                .map(|p| transformation * p)
                .collect();

            LocalizedRangeScan {
                pose: robot_pose,
                points,
            }
        }
    }

    ==> ./correlative_scan_matching/mod.rs <==
    pub mod correlation_grid;
    pub mod localized_range_scan;
    pub mod scan_matcher;
    pub mod scan_ring_buffer;

    /*
    pub fn find_corelation(scan: LidarResponse, map: &mut Map) -> (
       f32, f32, f32) {
        let mut most_good = (0.0, 0.0, 0.0);
```

```
    let mut most_good_score = 0;

    for angle in 0..360 {
        let points = point_cloud_from_lidar_response_angle(&scan,
            (angle as f32).to_radians());
        for x in -100..100 {
            for y in -100..100 {
                let mut current_score = 0;
                for point in &points {
                    let translated_point = point + Vector2::new(x
                        as f32, y as f32);
                    current_score += *map.access_clamped_metres(&
                       translated_point) as u64;
                }
                if current_score > most_good_score {
                    most_good_score = current_score;
                    most_good = (x as f32, y as f32, angle as f32
                       );
                }
            }
        }
    }

    most_good
}

pub fn match_two_scans(source: Vec<Point2<f32>>, target: Vec<
   Point2<f32>>) {
    let probability_map: Map = Map::new_empty(Vector2::new(500,
       500), 0.03);
}

/*struct MapperParemeters {
    use_scan_matching: bool,
    use_scan_barycetner: bool,
    minimum_time_interval: f64,
    minimum_travel_distance: f64,
    minimum_travel_heading: f64,
    scan_buffer_size: u32,
    scan_buffer_maximum_scan_distance: f64,
    link_match_minumum_response_fine: f64,
    linux_scan_maximum_distance: f64,
    do_loop_closing: bool,
    loop_search_maximum_distance: f64,
    loop_match_minimum_chain_size: u32,

    loop_match_maximum_variance_coarse: f64,

    loop_match_minimum_response_coarse: f64,
    loop_match_minimum_respose_fine: f64,

    correlation_search_space_dimension: f64,
    correlation_search_space_resolution: f64,
    correlation_search_space_smear_deviation: f64,

    loop_search_space_dimension: f64,
```

```
    loop_search_space_resolution: f64,
}

struct LocalizedRangeScan {

}

struct Mapper {

}

impl Mapper {
    fn process_scan(scan: LocalizedRangeScan) {

    }
}

pub fn correlate_scan() {

}*/*/

==> ./correlative_scan_matching/scan_matcher.rs <==
use nalgebra::{Rotation2, Vector2};
use roboq_types::model::{map::MetresPerPixel, Metres};
use serde::{Deserialize, Serialize};

use crate::model::{array2d::Array2D, pose::Pose2D};

use super::{
    correlation_grid::CorrelationGrid, localized_range_scan::
       LocalizedRangeScan,
};

#[derive(Serialize, Deserialize)]
pub struct ScanMatcher {
    correlation_grid: CorrelationGrid,

    search_space_probabilities: Vec<Array2D<f32>>,
    search_space_resolution: MetresPerPixel,
    search_space_size: f32,
    scan_circle_buffer: Vec<LocalizedRangeScan>,
}

impl ScanMatcher {
    pub fn new() -> Self {
        let correlation_grid_resolution = MetresPerPixel(0.01);

        let max_range_scan = 30.0;

        let search_space_size = 0.5;

        // all values <= 3 * sigma = 10cm
        let std_deviation = Metres(0.03);

        let correlation_grid_size_metres =
            max_range_scan * 2.0 + search_space_size;
```

```
        let correlation_grid_pos = Vector2::new(0.0, 0.0);

        let correlation_grid = CorrelationGrid::new(
            correlation_grid_size_metres,
            correlation_grid_resolution,
            correlation_grid_pos,
            std_deviation,
        );

        let search_space_resolution = MetresPerPixel(0.01);

        let mut search_space_grid_size =
            search_space_resolution.to_pixels(Metres(
               search_space_size)) as u32;

        // Ensuring that search space is odd number, so we can
           search -half_size and +half_size
        // without worrying about 0 (like 5 is -2 to +2, with 0)
        if (search_space_grid_size % 2) == 0 {
            search_space_grid_size += 1;
        }
        assert!(search_space_grid_size % 2 == 1);

        let revolution_resolution = 360;

        let space_at_resolution = Array2D::new_square(
           search_space_grid_size);

        let search_space_probabilities =
            vec![space_at_resolution; revolution_resolution];

        ScanMatcher {
            correlation_grid,
            search_space_probabilities,
            scan_circle_buffer: Vec::new(),
            search_space_resolution,
            search_space_size,
        }
    }

    pub fn match_scan_against_set_of_scans<'a>(
        &mut self,
        scan: &LocalizedRangeScan,
        set_of_scans: impl Iterator<Item = &'a LocalizedRangeScan
           >,
    ) -> Pose2D {
        let revolution_resolution = 360;
        let rad_per_degree =
            std::f32::consts::TAU / revolution_resolution as f32;

        // Initialization of correlation grid
        {
            self.correlation_grid.clear();
            self.correlation_grid.set_center(
               scan.pose.coords.coords);
```

```
        for scan in set_of_scans {
            self.correlation_grid.add_scan(scan);
        }
    }

    let (search_space_size, half_search_space) = {
        //each search_space has same width
        let width = self.search_space_probabilities[0]
           .get_width() as usize;

        let half_width = ((width - 1) / 2) as i32;

        (width, half_width)
    };

    for iangle in 0..revolution_resolution {
        let angle = iangle as f32 * rad_per_degree;

        let mut current_test_scan = Vec::with_capacity(
           scan.points.len());

        let rotation = Rotation2::new(angle);

        for point in scan.points.iter() {
            let rotated_point = rotation
                * (point - scan.pose.coords.coords)
                + scan.pose.coords.coords
                - self.correlation_grid.top_left_corner;

            let point_on_grid = self
                .correlation_grid
                .resolution
                .point_to_pixels(&rotated_point)
                - Vector2::new(half_search_space,
                   half_search_space);

            let point_weigth =
                self.correlation_grid.grid.index(&(
                   point_on_grid));

            current_test_scan.push(point_weigth);
        }

        let search_space_probabilities =
            &mut self.search_space_probabilities[iangle];

        let correlation_grid_size =
            self.correlation_grid.grid.get_width() as usize;

        for y in 0..search_space_size {
            let y_offset = y * correlation_grid_size;
            let y_offset_search_space = y * search_space_size
               ;

            for point in &current_test_scan {
```

```
            let point_offset = point + y_offset;

            update_search_space(
                &mut search_space_probabilities.data,
                &self.correlation_grid.grid.data,
                point_offset,
                y_offset_search_space,
                search_space_size,
            );
        }
    }
}

let divisor: f32 = 1.0 / scan.points.len() as f32;
let mut max_prob: f32 = f32::MIN;

for array in self.search_space_probabilities.iter_mut() {
    for prob in &mut array.data {
        *prob *= divisor;

        if *prob > max_prob {
            max_prob = *prob;
        }
    }
}

let mut avg_count: usize = 0;
let mut avg_x: f32 = 0.0;
let mut avg_y: f32 = 0.0;
let mut avg_sin: f32 = 0.0;
let mut avg_cos: f32 = 0.0;
let mut angle_old = 0.0;

assert_eq!(revolution_resolution, 360);

for (angle_index, array) in
    self.search_space_probabilities.iter().enumerate()
{
    let angle: f32 = angle_index as f32;

    for (index, prob) in array.data.iter().enumerate() {
        if *prob >= max_prob {
            angle_old = angle;
            let (sin, cos) = angle.to_radians().sin_cos()
               ;
            let (x, y) = array.from_index_to_xy(index);
            avg_sin += sin;
            avg_cos += cos;
            avg_x += x as f32;
            avg_y += y as f32;
            avg_count += 1;
        }
    }
}

log::trace!("Old␣angle:␣{angle_old}");
```

```
            log::trace!("Max␣response:␣{max_prob}");
            log::trace!("AVG␣X:␣{avg_x}.␣AVG␣Y:␣{avg_y}");
            log::trace!("AVG␣COUNT:␣{avg_count}");

            avg_x /= avg_count as f32;
            avg_y /= avg_count as f32;

            let angle = scan.pose.heading_rad + avg_sin.atan2(avg_cos
               );

            let coords = scan.pose.coords
                - Vector2::new(
                    self.search_space_size / 2.0,
                    self.search_space_size / 2.0,
                )
                + Vector2::new(
                    self.search_space_resolution.to_metres(avg_x),
                    self.search_space_resolution.to_metres(avg_y),
                );

            Pose2D {
                heading_rad: angle,
                coords,
            }
        }
    }

    /// Update values in given search space `probabilities`
    /// applying weight values from grid lookup slice
    #[inline]
    pub fn update_search_space(
        probabilities: &mut [f32],
        grid: &[u8],
        point_offset: usize,
        y_offset_search_space: usize,
        search_space_size: usize,
    ) {
        unsafe {
            let weight_ptr = grid.get_unchecked(point_offset) as *
               const u8;

            let prob_ptr =
                probabilities.get_unchecked_mut(y_offset_search_space
                   ) as *mut f32;

            for x in 0..search_space_size {
                *(prob_ptr.add(x)) += *weight_ptr.add(x) as f32;
            }
        }
    }

    // #[inline]
    // pub fn update_search_space_safe(
    //     probabilities: &mut [f32],
    //     grid: &[u8],
```

```
//     point_offset: usize,
//     y_offset_search_space: usize,
//     search_space_size: usize,
// ) {
//     for x in 0..search_space_size {
//         probabilities[y_offset_search_space + x] +=
//             grid[point_offset + x] as f32;
//     }
// }
//
// #[inline]
// pub fn update_search_space2(
//     probabilities: &mut [u32],
//     grid: &[u8],
//     point_offset: usize,
//     y_offset_search_space: usize,
//     search_space_size: usize,
// ) {
//     unsafe {
//         for x in 0..search_space_size {
//             let data = grid.get_unchecked(point_offset + x);
//             *(probabilities.get_unchecked_mut(
   y_offset_search_space + x)) +=
//                 *data as u32;
//         }
//     }
// }

impl Default for ScanMatcher {
    fn default() -> Self {
        Self::new()
    }
}

==> ./correlative_scan_matching/scan_ring_buffer.rs <==
use std::{collections::HashMap, time::SystemTime};

use roboq_types::devices::DeviceId;
use serde::{Deserialize, Serialize};

use super::localized_range_scan::LocalizedRangeScan;

#[derive(Clone, Serialize, Deserialize)]
pub struct ScanRingBuffer {
    #[serde(skip)]
    raw_scans: HashMap<DeviceId, (LocalizedRangeScan, SystemTime)
       >,
    #[serde(skip)]
    matched_scans: Vec<LocalizedRangeScan>,

    len: usize,
}

impl ScanRingBuffer {
    pub fn new(len: usize) -> Self {
```

```
        assert!(len >= 1, "Not␣empty␣buffer␣is␣allowed");

        ScanRingBuffer {
            raw_scans: HashMap::new(),
            matched_scans: vec![],
            len,
        }
    }

    pub fn iter(&self) -> impl Iterator<Item = &
       LocalizedRangeScan> {
        self.matched_scans.iter()
    }

    pub fn add_raw_scan(
        &mut self,
        device_id: DeviceId,
        scan: LocalizedRangeScan,
    ) {
        self.raw_scans.insert(device_id, (scan, SystemTime::now()
           ));
    }

    pub fn add_matched_scan(&mut self, scan: LocalizedRangeScan)
       {
        let scans = &mut self.matched_scans;

        scans.push(scan);

        if scans.len() > self.len {
            *scans = scans[(scans.len() - self.len)..scans.len()]
               .to_vec();
        }
    }

    pub fn latest(
        &self,
    ) -> impl Iterator<Item = (DeviceId, &LocalizedRangeScan,
       SystemTime)> {
        self.raw_scans
            .iter()
            .map(|(&id, (scan, time))| (id, scan, *time))
    }

    pub fn is_empty(&self) -> bool {
        self.matched_scans.is_empty()
    }

    pub fn len(&self) -> usize {
        self.len
    }
}

impl std::ops::Deref for ScanRingBuffer {
    type Target = [LocalizedRangeScan];
```

```
    fn deref(&self) -> &Self::Target {
        &self.matched_scans
    }
}

==> ./debug/mod.rs <==
pub use roboq_types::model::debug::{DebugData, DebugDataKind};
pub use tokio::sync::mpsc::error::TryRecvError;
pub use tokio::sync::mpsc::{channel, Receiver, Sender};

pub use roboq_types::model::debug::AlgoKind;

pub type DebugSender = Sender<DebugData>;

==> ./devices/compass.rs <==
use std::time::SystemTime;

use roboq_types::model::localization::{SensorData, SensorEvent};

use crate::shared_data::SharedData;

pub fn handle(shared_data: &SharedData, angle: f32) -> Option<
   SensorEvent> {
    let map_rotation = shared_data
        .config()
        .blocking_read()
        .world_map_config
        .map_rotation;

    log::trace!("Compass␣angle:␣{angle}");
    let angle_rad = angle as f64;

    let mut rotation_rad = (angle_rad - map_rotation).abs() as
       f32;

    if rotation_rad < 0.0 {
        rotation_rad += 2.0 * std::f32::consts::PI;
    }

    if rotation_rad >= 2.0 * std::f32::consts::PI {
        rotation_rad -= 2.0 * std::f32::consts::PI;
    }

    if rotation_rad > std::f32::consts::PI {
        rotation_rad -= 2.0 * std::f32::consts::PI;
    }

    let data = SensorData::new(rotation_rad, SystemTime::now());

    Some(SensorEvent::CompassRotation(data))
}
```