СОДЕРЖАНИЕ

ВВЕДЕНИЕ

По мере развития технологий все большее распространение получает беспилотная техника. Эти устройства способны выполнять различные задачи без прямого участия человека. Преимущества беспилотной техники включают снижение риска для жизни и здоровья людей, увеличение точности и эффективности выполнения задач, а также возможность работы в опасных или недоступных для человека местах. Эти устройства и системы могут выполнять различные функции как в промышленности, так и в других сферах деятельности. Примерами такой техники могут быть беспилотные летательные, подводные и надводные аппараты, роботы, выполняющие сложные задачи на производстве, или автоматизированные машины, выполняющие задачи доставки грузов.

Беспилотная техника часто оснащается различными системами навигации, которые позволяют ей перемещаться автономно по окружающей среде. Это может быть реализовано с помощью компьютерного зрения, лидаров, инерциальных измерительных блоков, GPS и других технологий. Какие конкретно средства используются для навигации зависит от среды, в которой работает беспилотная техника, и других факторов. Для работы с данными, полученными с датчиков, существуют различные методы и алгоритмы, отличающиеся по сложности реализации. Примером может служить SLAM (Simultaneous Localization and Mapping), использующий лидары или камеры. Датчики и другие системы, применяющиеся с беспилотной техникой для навигации, также могут выполнять и другие задачи, например, различные исследования окружающей среды.

Одним из важных критериев, по которым различается беспилотная техника, является способ осуществления движения. Например:

1. Сухопутная техника. Она может быть колесной, гусеничной, или иметь другие способы передвижения.
2. Летательная техника. Это могут быть дроны мультироторного типа, летательные аппараты с фиксированным крылом или даже бионические роботы, имитирующие полет животных.
3. Плавающая техника. Включает в себя подводные аппараты или роботов, способных перемещаться по поверхности воды. Большой интерес представляют сухопутные роботы, использующие бионические конечности для перемещения. Подобным системам необходимы алгоритмы, реализующие эффективное движение с использование сложной системы движущихся частей робота.

Темой данного дипломного проекта является разработка программной реализации комплекса сканирования пространства и навигации на основе роботизированных платформ и лидаров.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор аналогов

В данном подразделе будут рассмотрены несколько существующих реализаций SLAM.

1.1.1 Gmapping

Gmapping (Grid-based FastSLAM Mapping) – это алгоритм картографирования и локализации одновременно, разработанный для роботов с использованием сетки [1]. Он является одним из наиболее популярных и широко используемых алгоритмов SLAM для создания точных карт среды и определения положения робота в реальном времени.

Gmapping основывается на алгоритме FastSLAM, который комбинирует методы фильтрации частиц и построения карты на основе сетки. Он позволяет роботу одновременно определять свое местоположение и строить карту окружающей среды, используя данные из датчиков, таких как лазерный дальномер или сканирующий лазерный датчик.

Основные шаги, выполняемые алгоритмом Gmapping:

1. Инициализация. Алгоритм начинает с некоторой инициализации, устанавливая начальное положение робота и создавая пустую сетку карты.

2. Обработка данных с датчиков. Алгоритм получает данные с датчиков, таких как лазерный дальномер. Эти данные представляют собой измерения расстояния и углов сканирования вокруг робота.

3. Обновление частиц. Gmapping использует метод фильтрации частиц для оценки положения робота. Он создает набор частиц, представляющих возможные положения робота, и обновляет их в соответствии с полученными данными датчиков.

4. Построение карты. Для каждой частицы алгоритм строит локальную карту окружающей среды. Он использует данные сканирования с лазерного дальномера, чтобы определить препятствия и свободные области вокруг робота. Затем эти локальные карты объединяются в глобальную карту с использованием модели на основе сетки. Пример карты, построенной при помощи Gmapping, представлен на рисунке 1.1.

5. Обновление весов частиц. Алгоритм вычисляет веса каждой частицы на основе соответствия между сканированием с лазерного дальномера и картой. Частицы, которые лучше соответствуют данным, получают более высокие веса, тогда как менее соответствующие частицы получают более низкие веса.

6. Перевыборка частиц. Частицы с более высокими весами имеют большую вероятность быть выбранными для перевыборки. При перевыборке генерируются новые наборы частиц, основанные на весах предыдущих частиц. Это позволяет сосредоточиться на наиболее вероятных положениях робота.

7. Обновление оценки положения робота. Используя взвешенные частицы, алгоритм вычисляет оценку положения робота и карту окружающей среды. Эта оценка обновляется с каждым новым набором данных с датчиков.

Алгоритм Gmapping имеет несколько преимуществ:

1. Работа в реальном времени. Gmapping способен работать в реальном времени, обновляя карту и оценку положения робота по мере получения новых данных с датчиков.

2. Точность картографирования. Gmapping создает точные карты окружающей среды, используя модель на основе сетки. Он может обнаруживать препятствия и строить подробные и согласованные карты, которые могут быть использованы для планирования пути и навигации.

3. Устойчивость к шуму и неопределенности. Gmapping использует метод фильтрации частиц, который позволяет учитывать шумные данные с датчиков и уменьшать влияние неопределенности на оценку положения робота.

4. Масштабируемость. Gmapping может быть применен к различным типам роботов и средам. Он может работать как на малых, так и на больших площадях, а также адаптироваться к изменениям в окружающей среде.

5. Открытое программное обеспечение. Gmapping является частью пакета программного обеспечения ROS (Robot Operating System) и доступен в открытом исходном коде. Это позволяет разработчикам и исследователям легко использовать и настраивать алгоритм для своих специфических потребностей.

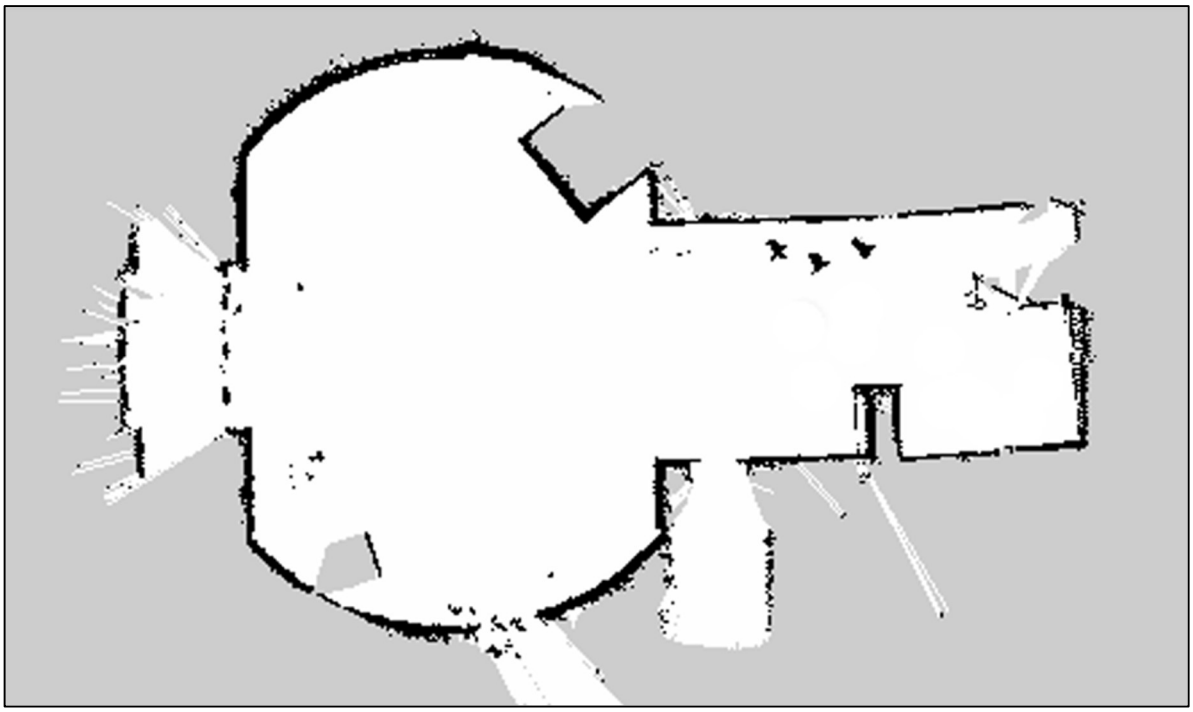

Рисунок 1.1 – Пример карты

Далее представлены некоторые ограничения, имеющиеся у Gmapping:

1. Требования к вычислительным ресурсам. Алгоритм Gmapping требует значительных вычислительных ресурсов для работы в реальном времени, особенно при обработке больших объемов данных с датчиков или при работе на сложных средах.

2. Чувствительность к движению. Gmapping может иметь трудности с точным картографированием и локализацией в случае быстрого движения робота или при наличии динамических объектов в окружающей среде. Это связано с ограничениями обработки данных с датчиков и возможностями модели на основе сетки.

3. Требования к калибровке датчиков. Для достижения наилучших результатов с Gmapping необходимо правильно настроить и откалибровать датчики, особенно лазерный дальномер. Неправильная калибровка может привести к неточностям в картах и оценке положения.

В целом, Gmapping является мощным алгоритмом картографирования и локализации SLAM, который обеспечивает точные карты окружающей среды и оценку положения робота в реальном времени. Он широко используется в робототехнике и автономных системах для выполнения задач навигации, планирования пути и взаимодействия с окружающей средой.

1.1.2 Google Cartographer

Google Cartographer – это библиотека и инструмент для создания 2D и 3D карт среды и выполнения одновременной локализации. Разработанная командой Google, Cartographer предоставляет мощные инструменты для робототехники и автономных систем, позволяя им строить детальные и точные карты окружающей среды и определять свое местоположение в реальном времени.

Вот основные компоненты и функции, предоставляемые Google Cartographer:

1. Картирование. Cartographer использует данные с датчиков, таких как лазерные дальномеры или камеры, для построения карты окружающей среды. Он использует методы на основе сетки для представления карты и может работать как в 2D, так и в 3D пространствах. Cartographer позволяет создавать детальные и точные карты, обнаруживать препятствия и другие объекты в окружающей среде.

2. Локализация. Кроме картографирования, Cartographer также выполняет одновременную локализацию, определяя местоположение робота в реальном времени. Он использует данные с датчиков и сопоставляет их с картой окружающей среды для определения положения робота с высокой точностью. Cartographer поддерживает как локализацию на основе признаков, так и локализацию на основе сканирования.

3. Интеграция с датчиками. Cartographer может работать с различными типами датчиков, включая лазерные дальномеры, камеры и инерциальные измерительные блоки (IMU). Он предоставляет гибкие возможности

интеграции с различными моделями и производителями датчиков, позволяя получать высококачественные данные для картографирования и локализации.

4. Поддержка различных платформ. Cartographer разработан для работы на различных платформах, включая настольные компьютеры, мобильные роботы и автономные автомобили. Он имеет модульную структуру, что позволяет легко настраивать его для конкретных потребностей и аппаратных сред.

5. Работа в реальном времени. Cartographer способен работать в режиме реального времени, обновляя карты и локализацию по мере получения новых данных с датчиков. Это делает его подходящим для задач навигации в реальном времени и планирования пути.

6. Синхронизация между роботами. Cartographer поддерживает многоагентную систему, что означает, что несколько роботов могут работать совместно для построения согласованных карт. Роботы могут обмениваться данными о картах и положении, что позволяет им эффективно координировать свои действия.

7. Автоматическое выравнивание карт. Одной из особенностей Cartographer является его способность автоматически выравнивать карты разных фрагментов одной общей карты. Это позволяет роботам создавать единое и последовательное представление окружающей среды, даже если они перемещаются по разным областям.

8. Гибкая настройка и расширение. Cartographer предоставляет гибкую архитектуру, которая позволяет настраивать систему и добавлять собственные модули и алгоритмы. Разработчики могут адаптировать Cartographer под свои конкретные потребности и интегрировать его с другими компонентами своей робототехнической системы.

9. Инструменты визуализации. Cartographer предоставляет инструменты визуализации, которые позволяют визуально представить текущую карту и оценку положения робота. Он поддерживает различные форматы визуализации, включая 2D и 3D визуализацию, что упрощает анализ и отладку результатов SLAM.

10. Открытое программное обеспечение. Cartographer является проектом с открытым исходным кодом, доступным в рамках инициативы открытого ПО Google. Это позволяет разработчикам и исследователям свободно использовать, модифицировать и распространять библиотеку, а также вносить свои вклады в ее развитие.

Основная особенность Google Cartographer заключается в том, что он состоит из локального SLAM и глобального SLAM. Локальный SLAM создает отдельные небольшие части карты. Затем, глобальный SLAM алгоритм ищет совпадения между частями карт, а также между данными сенсоров и всеми картами, созданными при помощи локального SLAM в параллельных процессах. В результате, глобальный SLAM формирует единую карту окружающего пространства [2]. Схема работы алгоритма Google Cartographer представлена рисунке 1.2.

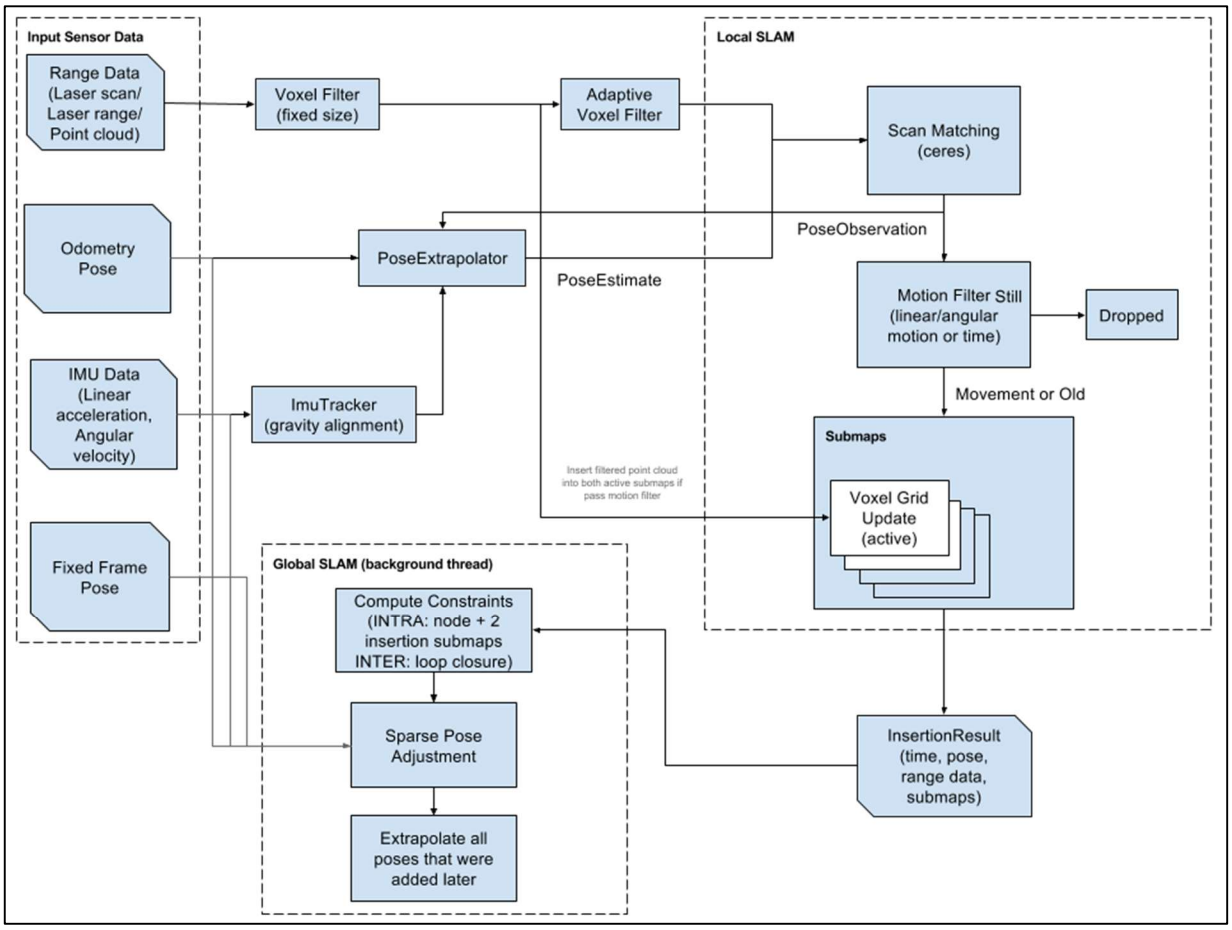

Рисунок 1.2 – Схема работы алгоритма Google Cartographer

Google Cartographer представляет собой мощный инструмент для робототехники и автономных систем, предоставляющий возможности построения детальных карт окружающей среды и определения местоположения в реальном времени. Его гибкость, масштабируемость и поддержка различных платформ делают его привлекательным выбором для разработчиков, которые стремятся реализовать SLAM в своих проектах.

1.1.3 RTAB-Map

Rtabmap (Real-Time Appearance-Based Mapping) – это библиотека и инструмент для одновременной локализации и построения карты в режиме реального времени. Разработанная и поддерживаемая командой IntRoLab (Laboratory of Intensive Robotics) в Монреальском университете, Rtabmap предлагает мощные возможности по картографированию и навигации для робототехники и автономных систем. Используемый алгоритм основан на поиске и сопоставлении соответствий визуальных данных сенсоров с использованием механизма памяти, где хранится база данных визуальных образов в соответствии с данными о местоположении робота [3]. Пример базы данных изображения представлен на рисунке 1.3.

Вот основные компоненты и функции, предоставляемые Rtabmap:

1. Картирование. Rtabmap использует данные с различных датчиков, таких как лазерные дальномеры, камеры и инерциальные измерительные блоки (IMU), для построения карты окружающей среды. Он использует методы на основе признаков для представления карты и может работать как в 2D, так и в 3D пространствах. Rtabmap позволяет создавать плотные карты с высокой детализацией, обнаруживать препятствия, сохранять информацию о среде и определять местоположение робота.

2. Локализация. Rtabmap выполняет одновременную локализацию, определяя местоположение робота в реальном времени. Он использует данные с датчиков, такие как видеопотоки и сканирование окружающей среды, и сопоставляет их с предыдущими данными для оценки положения. Rtabmap поддерживает как локализацию на основе признаков, так и локализацию на основе геометрии.

3. Обнаружение и сопоставление признаков. Rtabmap имеет встроенные алгоритмы для обнаружения и сопоставления признаков на основе изображений. Он может использовать различные дескрипторы, такие как SIFT, SURF или ORB, для идентификации ключевых точек и создания описания сцены. Это позволяет Rtabmap работать с различными типами датчиков и обрабатывать данные с камер и других источников.

4. Граф оптимизации. Rtabmap использует граф оптимизации для улучшения оценки положения и формирования карты. Он строит граф, где узлы представляют собой местоположения робота, а ребра – сопоставления и измерения между ними. Затем Rtabmap выполняет оптимизацию графа, чтобы улучшить точность оценки положения и форму карты.

5. Поддержка различных датчиков и платформ. Rtabmap разработан для работы с различными типами датчиков и платформ, включая мобильные роботы, автономные автомобили и дроны. Он имеет модульную архитектуру, что позволяет легко интегрировать новые датчики и настраивать его для конкретных потребностей.

6. Обнаружение и учет петель. Rtabmap способен обнаруживать петли в окружающей среде и эффективно их учитывать при построении карты. Это позволяет обеспечить более точную локализацию и улучшить качество карты путем устранения ошибок, связанных с повторным посещением областей.

7. Гибридный подход к SLAM. Rtabmap сочетает в себе особенности визуального и геометрического SLAM. Он использует информацию изображений и геометрических данных с датчиков для определения положения робота и построения карты. Это позволяет достичь лучшей точности и надежности в различных сценариях.

8. Встроенная система распознавания объектов. Rtabmap имеет встроенную систему распознавания объектов, позволяющую роботу определять и отслеживать объекты в окружающей среде. Это полезно для задач навигации и взаимодействия с реальными объектами.

9. Визуализация и анализ результатов. Rtabmap предоставляет инструменты визуализации, которые позволяют визуально представить

текущую карту и оценку положения робота. Он также предлагает средства анализа качества карты, такие как графики ошибок локализации и картографии, а также статистику по качеству сопоставления признаков.

10. Расширяемость и открытость. Rtabmap предоставляет API и набор инструментов для расширения его функциональности и интеграции с другими системами. Он является проектом с открытым исходным кодом, что позволяет разработчикам и исследователям свободно использовать и модифицировать его, а также вносить свои вклады в его развитие.

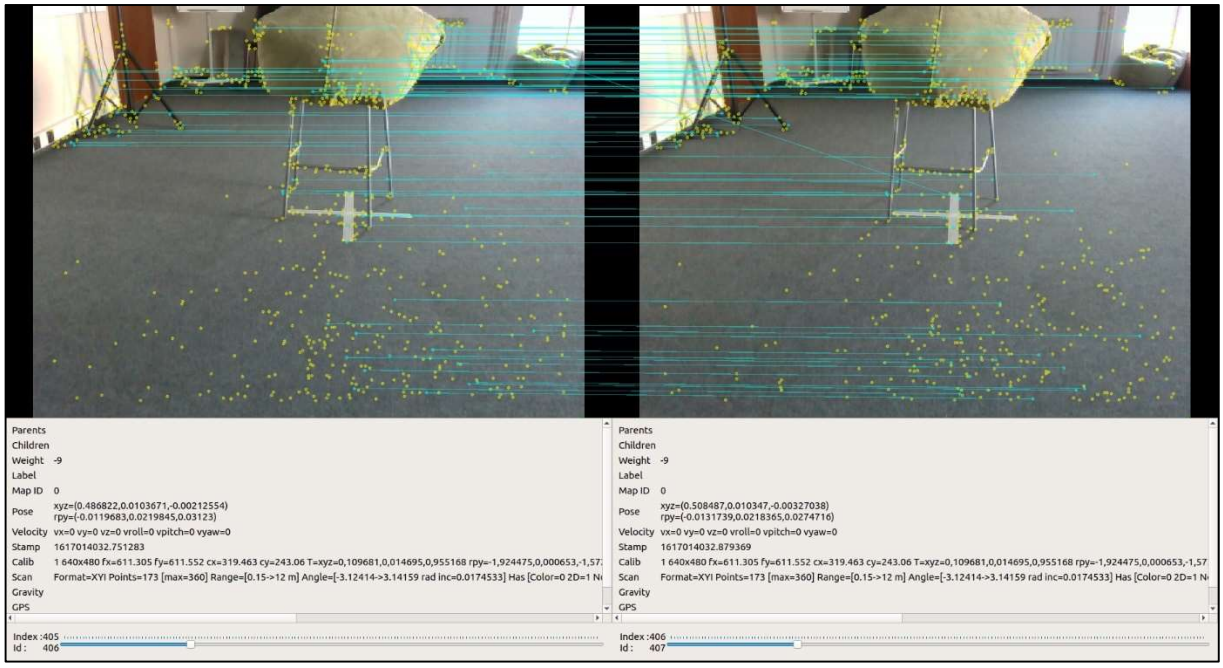

Рисунок 1.3 – База данных изображения Rtabmap

Rtabmap представляет собой мощный инструмент для робототехники и автономных систем, предоставляющий возможности построения детальных карт окружающей среды и определения местоположения в режиме реального времени. С его помощью можно реализовать различные задачи, связанные с картографированием и навигацией, включая автономную навигацию, планирование пути и взаимодействие с окружающей средой.

1.2 Обзор технологий и методов

В этом разделе будут рассмотрены и описаны технологии и методы, которые будут применяться при разработке дипломного проекта.

1.2.1 SLAM

SLAM, или одновременное локализация и построение карты (Simultaneous Localization and Mapping), является одним из ключевых методов в области робототехники и компьютерного зрения. Он позволяет роботу

одновременно определять свое местоположение в неизвестной среде и строить карту этой среды.

Принцип работы SLAM основывается на использовании датчиков и алгоритмов для сбора информации о среде и последующего анализа этой информации. Обычно в SLAM используются два основных типа датчиков: датчики измерения расстояния (например, лазерные сканеры или стереокамеры) и датчики измерения ориентации (например, инерциальные измерители).

С математической точки зрения, SLAM пытается оценить карту и весь путь, пройденный роботом. Таким образом, поза робота рассчитывается только в конце траектории, проделанной роботом [4]. Графическая модель SLAM подхода представлена на рисунке 1.4.

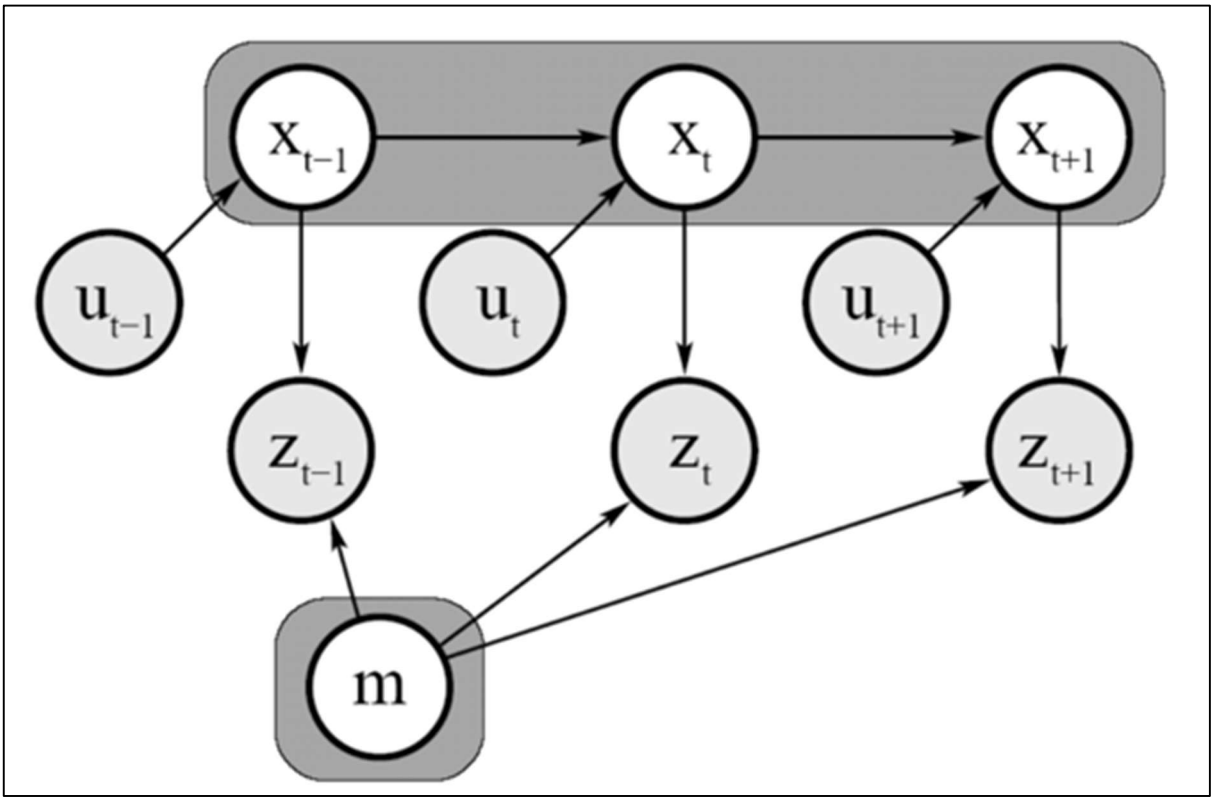

Рисунок 1.4 – Графическая модель SLAM подхода

Процесс SLAM обычно состоит из следующих шагов:

1. Захват данных. Робот собирает данные из своих датчиков, таких как лазерные сканеры, камеры и инерциальные измерители. Лазерные сканеры измеряют расстояния до окружающих объектов, а камеры могут использоваться для извлечения признаков и ориентации. Инерциальные измерители измеряют ускорение и угловую скорость робота.

2. Оценка движения. Используя данные с инерциальных измерителей, робот оценивает свое движение и изменение местоположения с течением времени. Это может включать оценку скорости, ускорения и изменения угла поворота робота.

3. Извлечение признаков. С помощью данных с лазерных сканеров или камер робот извлекает признаки из окружающей среды. Признаками могут быть контуры объектов, особые точки или текстуры. Они используются для сопоставления и определения положения робота относительно них.

4. Ассоциация данных. Следующий шаг состоит в ассоциации данных, то есть соотнесении данных с датчиков с предыдущими измерениями и признаками на карте. Это позволяет определить, какие измерения соответствуют одним и тем же объектам или признакам.

5. Обновление карты. После ассоциации данных робот обновляет карту окружающей среды, добавляя новые признаки и объекты. Карта может быть представлена в виде сетки, графа или другой структуры данных, которая описывает пространственное расположение объектов.

6. Обновление местоположения. С использованием ассоциированных данных и обновленной карты робот обновляет свое текущее местоположение в среде. Этот шаг позволяет роботу уточнить свое местоположение на основе новых данных.

7. Повторение. Весь процесс повторяется снова, начиная с захвата данных. Робот непрерывно собирает данные, обновляет карту и определяет свое местоположение в реальном времени.

SLAM является активной областью исследований, и существует много различных алгоритмов и подходов, позволяющих реализовать этот метод. Некоторые из наиболее распространенных алгоритмов SLAM включают в себя:

1. EKF-SLAM (Extended Kalman Filter SLAM). Этот алгоритм использует расширенный фильтр Калмана для оценки состояния робота и построения карты. Он предполагает линейность моделей движения робота и измерений признаков.

2. FastSLAM. Этот алгоритм использует алгоритм частицы для оценки состояния робота и построения карты. Он разбивает задачу SLAM на набор независимых задач локализации и построения карты для каждого признака на карте.

3. GraphSLAM. Этот алгоритм представляет карту и траекторию робота в виде графа и использует методы оптимизации графов для оценки состояния робота и построения карты.

4. ORB-SLAM. Этот алгоритм комбинирует методы извлечения признаков ORB (Oriented FAST and Rotated BRIEF) с алгоритмом RANSAC (Random Sample Consensus) для решения задачи SLAM в режиме реального времени.

1.2.2 Инверсная кинематика

Инверсная кинематика – это обратная задача к прямой кинематике, которая позволяет определить значения углов, положения и скорости звеньев манипуляционной системы на основе известного положения ее рабочего

органа в пространстве. В данном контексте мы рассмотрим инверсную кинематику роботов.

Инверсная кинематика имеет важное значение в робототехнике, поскольку позволяет управлять движением робота, задавая желаемое положение его рабочего органа. Вместо определения значений углов каждого звена вручную, инверсная кинематика автоматически рассчитывает эти значения с учетом геометрии и ограничений конкретной манипуляционной системы.

Для понимания работы инверсной кинематики с точки зрения геометрии, предположим, что у нас есть манипуляционная система с несколькими звеньями, объединенными в цепь. Каждое звено представляет собой сегмент, связанный с соседними звеньями при помощи сочленений, таких как вращательные или поступательные сочленения.

Главная задача заключается в определении значений углов или длин каждого звена, необходимых для достижения заданного положения рабочего органа (например, конца робота) в трехмерном пространстве.

Процесс решения инверсной кинематики может быть довольно сложным и зависит от конкретной геометрии манипулятора. В общем случае, для решения задачи инверсной кинематики требуется выполнение следующих шагов:

1. Определение координат и ориентации конца робота. Это заданные значения положения и ориентации, которые мы хотим достичь с помощью робота.

2. Анализ геометрии манипулятора. Необходимо изучить структуру и геометрию манипуляционной системы, включая длины звеньев, типы сочленений и ограничения на перемещение каждого звена.

3. Расчет углов или длин звеньев. На основе известных значений положения и ориентации конца робота, а также геометрии манипулятора, нужно вычислить углы или длины звеньев, обеспечивающие требуемое положение.

4. Проверка решения. После расчета значений углов или длин звеньев необходимо проверить, насколько близко полученное решение приближается к желаемому положению и ориентации конца робота. Если требуемая точность достигнута, можно передать вычисленные значения в соответствующие актуаторы робота для выполнения заданного движения.

Важно отметить, что инверсная кинематика может иметь несколько решений или быть неразрешимой в некоторых случаях, особенно когда манипуляционная система имеет ограничения или находится в сингулярных точках. Также могут возникать проблемы совмещения требуемого положения конца робота с физическими ограничениями системы, такими как препятствия или ограничения на движение звеньев.

В реальных системах решение инверсной кинематики обычно выполняется с использованием численных методов, таких как метод Ньютона-Рафсона или методы оптимизации. Эти методы позволяют находить

численное решение для требуемого положения конца робота, учитывая геометрию и ограничения манипулятора.

1.2.3 Пакетный менеджер Cargo

Cargo – это сборочный и управляющий пакетами инструмент в языке программирования Rust. Он является официальным инструментом для управления зависимостями и сборки проектов на Rust. Cargo обеспечивает простой и эффективный способ создания, сборки и управления проектами [5].

Cargo позволяет легко управлять зависимостями проекта. Он использует файл `Cargo.toml` для указания зависимостей и их версий. Cargo может автоматически загружать и устанавливать зависимости из центрального репозитория пакетов Cargo. Это упрощает добавление сторонних библиотек в проект и обновление зависимостей.

Cargo обеспечивает простой способ сборки проекта. Он автоматически обнаруживает и собирает все файлы исходного кода в проекте. Cargo предоставляет различные команды сборки, такие как `cargo build`, `cargo run` и `cargo test`, которые позволяют собирать, запускать и тестировать проект соответственно. Он автоматически управляет компиляцией и линковкой зависимостей проекта.

Cargo предлагает простой способ создания и управления проектами на Rust. Команда `cargo new` создает новый проект со структурой каталогов, включающей файл `Cargo.toml` и каталог `src` для исходного кода. Cargo также обеспечивает интеграцию с системой контроля версий Git.

Cargo позволяет упаковывать и распространять проекты на Rust в виде пакетов. С помощью команды `cargo package` можно создать архив проекта, содержащий все необходимые файлы для его сборки.

Cargo предлагает встроенную поддержку для модульного и интеграционного тестирования в проектах Rust. С помощью команды `cargo test` можно запустить все тесты в проекте или только определенные тесты.

Cargo также предоставляет команду, называемую `cargo check`. Эта команда быстро проверяет код, чтобы убедиться, что он компилируется, без создания исполняемого файла. `Cargo check` выполняется намного быстрее, чем `cargo build`, поскольку пропускает этап создания исполняемого файла. Таким образом, можно удобно проверять код на наличие ошибок в процессе разработки.

Cargo является мощным и удобным инструментом для разработки проектов на Rust. Он упрощает управление зависимостями, сборку проекта и распространение пакетов. Богатый функционал и простота использования делают Cargo неотъемлемой частью экосистемы Rust.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Структурная схема, иллюстрирующая блоки программного средства, а также межблочные связи, приведена на чертеже ГУИР.400201.106 Э1.

Разделение проекта на блоки помогает лучше понять его структуру и организацию. Каждый часть схемы отвечает за определенную функциональность или задачу, и такая структурированность помогает ориентироваться в проекте. При разбиении проекта на блоки, каждый из них может быть разработан и протестирован независимо. Это облегчает интеграцию блоков между собой и обеспечивает более надежную работу.

В схеме представлено разбиение блоков на две группы в зависимости от того, к какой части программного комплекса относится блок: скрипт для роботизированной платформы или приложение для пользовательской станции.

В результате разработки были выделены следующие структурные единицы:

1. Блок сбора данных с лидара.
2. Блок передачи сигналов сервоприводам.
3. Блок общения с пользовательской станцией.
4. Блок общения с роботизированной платформой.
5. Блок управляющих элементов пользовательского интерфейса.
6. Блок вычисления углов поворотов сервоприводов.
7. Блок обработки данных с лидара.
8. Блок отображения карты.

Далее будут приведены краткие описания и функционал каждого из блоков.

2.1 Описание блоков

2.1.1 Блок сбора данных с лидара

Данный блок служит для сбора данных, которые в последствие будут использоваться для картографирования. Лидар представляет собой датчик, использующий лазер для измерения расстояний до окружающих объектов. Он осуществляет сканирование окружающей среды, отправляя лазерный луч и измеряя время, требуемое для его отражения от объектов и возвращения обратно к датчику. Лидар сканирует окружающую среду в горизонтальной, создавая точечное облако данных, которое представляет собой набор измерений расстояний и углов относительно робота. Подключается лидар к одноплатному компьютеру Raspberry Pi по интерфейсу UART (Universal Asynchronous Receiver/Transmitter). Через него осуществляется получение данных и управление лидаром.

Описываемый блок не обрабатывает данные, а только собирает их для дальнейшей передачи.

2.1.2 Блок передачи сигналов сервоприводам

Блок отправки сигналов сервоприводам получает данные об угле поворота нужного сервопривода от блока общения с пользовательской станцией, при этом в нем не реализован полный алгоритм движения. Для управления сервоприводами на Raspberry Pi используется 16-канальный контроллер PCA9685. Он позволяет управлять несколькими сервоприводами одновременно с помощью ШИМ-сигналов (Широтно-импульсная модуляция). Так как у роботизированной платформы 8 конечностей, в каждой из которых 3 сервопривода, одна PCA не может ими управлять. Поэтому необходимо задавать не только углы поворота сервоприводов, но и выбирать нужный контроллер PCA.

2.1.3 Блок общения с пользовательской станцией

Блок общения с пользовательской станцией обеспечивает взаимодействие между роботизированной платформой и персональным компьютером с работающим на нем приложением. Передача данных происходит в обе стороны. От блока сбора данных с лидаров непрерывно поступает поток отсканированных точек. Для дальнейшей обработки они отправляются на пользовательскую станцию.

Так как алгоритм движения роботизированной платформы не реализован в скрипте для одноплатного компьютера Raspberry Pi, в описываемый блок поступают данные, содержащие в себе информацию о том, на какой угол повернуть конкретный сервопривод.

Передача в обе стороны происходит по UDP (User Datagram Protocol). Это простой и легковесный протокол транспортного уровня в сетевой модели OSI (Open Systems Interconnection), который не гарантирует доставку данных.

2.1.4 Блок общения с роботизированной платформой

Блок общения с роботизированной платформой работает по аналогии с блоком общения с пользовательской станцией. Посредством локальной компьютерной сети, создается связь с одноплатным компьютером Raspberry Pi под управлением разработанного скрипта.

Передача пакетов данных осуществляется по протоколу UDP. С одной стороны возможны потери некоторых пакетов и, как следствие, небольшая неточность движения, с другой стороны легковесный протокол позволяется организовать быстрое соединения, что актуально в случае передачи большого объема данных с лидара.

Наборы точек, отсканированных лидаром, поступают непрерывно с роботизированной платформы и передаются в блок обработки данных с лидара для дальнейшего построения и актуализации карты. Данные о поворотах сервоприводов поступают с блока вычисления углов поворота в

зависимости от того, находится ли роботизированная платформа в режиме движения.

2.1.5 Блок управляющих элементов пользовательского интерфейса

Блок управляющих элементов позволяет через графический интерфейс задавать направление движения роботизированной платформы или точечно управлять отдельными элементами конечностей.

Так как сегменты ног связаны между собой шарнирами, движение одного сегмента приводит к изменения положения другого. Для вычисления углов поворотов, в блоке реализована инверсная кинематика. Так как изменение положения одного сегмента приводит к изменению положения другого, данные об углах поворота передаются в обе стороны между блоками.

В блоке управляющих элементов реализованы кнопки, нажатия на которые передают роботизированной платформе команды для движения. Перемещение может осуществляться в двух направлениях: вперед или назад. Также имеется возможность передать команду роботизированной платформе вращаться на месте вокруг своей оси в двух направлениях. Чтобы остановить движение робота-паука, реализована соответствующая кнопка. Присутствует возможность включать и выключать сервоприводы.

Отображаемые блоком управляющих элементов конечности роботизированной платформы являются интерактивными. Это означает, что можно управлять сервоприводами напрямую через данные элементы графического интерфейса.

2.1.6 Блок вычисления углов поворотов сервоприводов

Так как роботизированная платформа имеет конечности с 3 суставами, для достижения необходимого положения ноги робота паука, необходимо просчитать оптимальные углы поворота всех сервоприводов. Данную задачу выполняет блок вычисления углов поворотов. Для этого в нем реализованы алгоритмы инверсной кинематики, которые позволяют определить углы поворотов суставов, необходимые для достижения требуемой позиции и ориентации конечных точек конечностей робота. Данные о направлении движения каждой ноги приходят от блока управляющих элементов. После вычисления углов поворотов суставов на основе инверсной кинематики, полученные значения передаются далее для установки соответствующих позиций сервоприводов.

У каждой конечности роботизированной платформы есть 3 сустава, при этом 2 из них вращаются в одной плоскости. Таким образом в рамках инверсной кинематики рассматриваются два сегмента с двумя шарнирами. Углы, на которые поворачиваются рассматриваемые суставы, связаны между собой через теоремы синуса и косинуса.

Для определения конечного положения ноги роботизированной

платформы на основе данных о поворотах суставов используется прямая кинематика. Полученная информация идет в блок управляющих элементов для обновления графического отображения конечностей робота-паука.

2.1.7 Блок обработки данных с лидара

Рассматриваемый блок получает данные с лидара, которые представляют собой расстояния и углы от роботизированной платформы до окружающих объектов. Эти данные называются отметками сканирования. Они периодически поступают в сетку для построения карты. Она представляет собой множество ячеек, в которых хранится информация о заполненности пространства. Сетка карты обновляется на основе полученных отметок сканирования. Блок преобразует отметки в координаты точек в пространстве и затем помечает соответствующие ячейки сетки как занятые или свободные.

Для оценки и обновления положения робота в пространстве на основе полученных отметок сканирования и предыдущих оценок положения используется фильтр частиц. Он генерирует и обновляет набор гипотез о положении робота, используя статистические методы. Оценка положения робота корректируется на основе новых отметок сканирования. Что бы определить соответствие между отметками сканирования и ячейками сетки карты, выполняется ассоциация данных. Это помогает определить, какие отметки сканирования соответствуют занятым или свободным областям на карте.

Блок объединяет обновленные данные сетки, чтобы создать глобальную карту окружающей среды. Это делается путем суммирования информации от различных сканирований и обновления соответствующих ячеек сетки.

Описываемый блок также выполняет функцию задания режима работы лидара. Например, частоту сбора данных, так как данный параметр сильно влияет на качество построенной карты. Еще одним важным параметром является сектор, в котором лидар сканирует окружающее пространство.

2.1.8 Блок отображения карты

Основной функцией данного блока является отображение карты с помощью графического интерфейса. Данные с лидара поступают периодически, поэтому блок обработки, связанный с блоком отображения карты, также постоянно обновляет информацию об окружении роботизированной платформы. Сама карта представляет собой массив ячеек, в которых хранится информация о занятости пространства. Сетка карты периодически обновляется, по этой причине рассматриваемый блок постоянно актуализирует карту, отображаемую с помощью графического интерфейса. Кроме того, обновляется положение роботизированной платформы.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Функциональные блоки проекта продублированы из структурной схемы. В отдельный блок выделена реализация трейта eframe::App для приложения rust, так как остальные блоки, представленные через структуры данных, существуют внутри него. Таким образом, можно выделить следующие модули для описания:

1. Блок передачи сигналов сервоприводам.
2. Блок общения с пользовательской станцией.
3. Блок сбора данных с лидара.
4. Реализация eframe::App
5. Блок общения с роботизированной платформой.
6. Блок управляющих элементов пользовательского интерфейса.
7. Блок вычисления углов поворотов сервоприводов.
8. Блок обработки данных с лидара.
9. Блок отображения карты.

Взаимосвязь между основными компонентами представлена на диаграмме классов ГУИР.400201.106 РР.1 и диаграмме последовательности ГУИР.400201.106 РР.2.

3.1 Блок передачи сигналов сервоприводам

Класс `Servo` представляет сервопривод и содержит методы для управления его положением.

Описание полей класса `Servo`:

1. Поле `board`. Ссылка на контроллер, к которому подключен сервопривод.
2. Поле `servo_number`. Номер сервопривода для конкретного контроллера.
3. Поле `min_angle`. Минимальное допустимое положение сервопривода в градусах.
4. Поле `max_angle`. Максимальное допустимое положение сервопривода в градусах.
5. Поле `calibrated_angle`: Калибровочный угол, на который можно повернуть сервопривод для выравнивания.
6. Поле `midle_angle`. Угол, к которому нужно повернуть сервопривод, чтобы он находился в середине диапазона движения.

Описание методов класса `Servo`:

1. Метод `__init__(self, board, servo_number, min_angle, max_angle, calibrated_angle, midle_angle)`. Он инициализирует объект класса `Servo` и устанавливает значения атрибутов.
2. Метод `rotate_to_min(self)`. Он поворачивает сервопривод в положение минимального угла `min_angle`.

3. Метод `rotate_to_calibrated(self)`. Он поворачивает сервопривод на калибровочный угол `calibrated_angle`.

4. Метод `rotate_to_angle(self, angle)`. Он поворачивает сервопривод в заданный угол `angle`.

5. Метод `rotate_to_midle(self)`. Он поворачивает сервопривод в середину диапазона движения, заданный углом `midle_angle`.

6. Метод `rotate_to_max(self)`. Он поворачивает сервопривод в положение максимального угла `max_angle`.

3.2 Блок общения с пользовательской станцией

Данные передаются по протоколу UDP с использованием сокетов. Создаем сокет с помощью функции `socket.socket()`, указываются параметры `socket.AF_INET` для сетевого адреса и `socket.SOCK_DGRAM` для типа сокета UDP. Созданный сокет привязывается к конкретному адресу и порту с помощью метода `bind()`. Функция `divide_chunks(l, n)` используется для разделения последовательности `l` на фрагменты длиной `n`. Функция `get_message()` получает сообщения через сокет UDP. Она вызывает `socket.recvfrom(10024)` для получения данных из сокета. 10024 – максимальный размер получаемых данных. Затем функция разделяет полученные данные на фрагменты фиксированной длины с помощью `divide_chunks()`. После чего каждый фрагмент данных распаковывается с помощью `struct.unpack(">bbf", chunk)`, результат сохраняется в виде списка.

3.3 Блок сбора данных с лидара

Для сбора данных с лидара используется библиотека `adafruit_rplidar`. При помощи конструктора `adafruit_rplidar.RPLidar(None, '/dev/ttyUSB0')` создается объект `RPLidar`. Метод `iter_measurments()` получает последовательность измерений с лидара. В каждой итерации цикла измерений метод возвращает кортеж `(quality, angle, distance)`. Здесь `angle` – это угол, на который был совершен оборот лидара, а `distance` – это расстояние до объекта, обнаруженного лидаром.

3.4 Реализация eframe::App

Структура `MyEguiApp` является основой графического пользовательского интерфейса приложения. Она реализует трейт `App` из библиотеки `eframe`.

Этот трейт определяет методы, которые необходимо реализовать для

обработки событий и отрисовки графического интерфейса приложения.

Описание полей структуры `MyEguiApp`:

1. Поле `socket` типа `UdpSocket`. Представляет UDP-сокет для создания соединения и обмена данными с роботизированной платформой.
2. Поле `spider` типа `Spider`. Представляет объект `Spider` для конфигурации сервоприводов роботизированной платформы.
3. Поле `legs`, вектор типа. Представляющий массив объектов для управления конечностями роботизированной платформы.
4. Поле `last_legs`, вектор типа `Leg`. Массив для хранения предыдущего состоя конечностей робота-паука.
5. Поле `picked_leg` типа `usize`. Хранит индекс выбранной ноги.
6. Поле `motors_off` типа `bool`. Указывает, выключены ли сервоприводы.
7. Поле `i` типа `usize`. Используется внутри методов для итераций.
8. Поле `gait`, вектор типа `SingleLegGait`. Массив для управления движением роботизированной платформы.
9. Поле `remote_control` типа `DebugRemoteControl`. Представляет удаленное управление отладкой.
10. Поле `lidar_widget` типа `LidarWidget`. Представляет виджет для отображения данных лидара.
11. Поле `map_widget` типа `MapWidget`. Представляющее виджет для отображения карты.
12. Поле `correlation_widget` типа `MapWidget`. Представляет виджет для отображения корреляций на карте.
13. Поле `occupancy_grid` типа `OccupancyGrid`. Представляет сетку занятости для карты.
14. Поле `_points` типа `Vec<Point2<f32>>`. Представляет вектор точек двухмерного пространства.
15. Поле `_debug_lidar_data` типа `Receiver<DebugResponse>`. Представляет данные отладки для лидара.
16. Поле `navigation_handle` типа `NavigationModuleHandle`. Представляет обработчик навигационного модуля.
17. Поле `lines` типа `Lines<'static>`. Представляет коллекцию строк.
18. Поле `_scan_matcher` типа `ScanMatcher`. Представляет объект для сопоставления сканирования.
19. Поле `do_a_step` типа `bool`. Флаг, указывающий на необходимость выполнения одного шага.
20. Поле `probability_widget` типа `MapWidget`. Представляет виджет для отображения вероятности на карте.
21. Поле `view_angle` типа `u32`. Представляет угол обзора.
22. Поле `run` типа `bool`. Флаг, указывающий на необходимость запуска

или остановки работы.

23. Поле `_first_click` типа `Option<Point2<f32>>`. Представляет опциональную точку первого клика.

24. Поле `_last_shown_point` типа `usize`. Представляет последнюю показанную точку.

25. Поле `last_added_scan` типа `LocalizedRangeScan`. Представляет последнее добавленное сканирование с локализацией.

26. Поле `scans_to_process` типа `Arc<Notifier>`. Представляет синхронизированный оповещатель для обработки сканирований.

27. Поле `syncronize_visuals` типа `bool`. Флаг, указывающий, следует ли синхронизировать визуализацию.

28. Поле `points_of_points` типа `Vec<Vec<Point2<f32>>>`. Представляет вектор векторов точек двухмерного пространства с плавающей запятой.

Описание методов структуры `MyEguiApp`:

1. Метод `new(_cc: &eframe::CreationContext<'_>)`. Это конструктор структуры `MyEguiApp`, принимающий контекст создания и возвращающий новый экземпляр структуры. Внутри метода создается и инициализируется объект `udp_socket` типа `UdpSocket`, привязывается к адресу пользовательской станции и устанавливается соединение с адресом одноплатного компьютера Raspberry Pi, управляющего роботизированной платформой. Также создается массив, содержащий порядок ног в правильной последовательности. В конструкторе инициализируются векторы `legs` и `gait`. В итоге создается и возвращается новый экземпляр структуры `MyEguiApp` с инициализированными полями.

2. Метод `update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame)`. Он реализует обновление состояния приложения. В цикле обновляются каждый элемент в векторе `gait` и соответствующий элемент в векторе `legs`. Затем создается панель с использованием контекста `ctx`. Внутри панели отображаются кнопки для управления состоянием моторов и движениями роботизированной платформой. Затем отображаются виджеты для настройки параметров ног и элементов управления. Реализованы кнопки для запуска и остановки картографирования. Если состояние ног изменилось, то происходит отправка данных через UDP-сокет. Затем вызывается метод `request_repaint_after()` для запроса перерисовки интерфейса.

3.5 Блок общения с роботизированной платформой

Структура `DebugRemoteControl` представляет собой удаленный контроллер. Описываемая структура реализует трейт `Default`, который позволяет создать экземпляр `DebugRemoteControl` с значениями по

умолчанию. Внутри метода `default()` создается UDP-сокет и устанавливается соединение с удаленным контроллером. Затем создается и возвращается экземпляр `DebugRemoteControl` с установленными значениями полей `socket`, `manual_control`, `last_left_speed` и `last_right_speed`.

Описание полей структуры `DebugRemoteControl`:

1. Поле `socket` типа `UdpSocket`. Представляет сокет для отправки и приема данных по протоколу UDP. Этот сокет используется для установления соединения с удаленным контроллером.
2. Поле `last_left_speed` типа `f32`. Хранят последние значения скорости движение в левую сторону.
3. Поле `last_right_speed` типа `f32`. Хранят последние значения скорости движение в правую сторону.
4. Поле `manual_control` типа `bool`. Указывает, включено ли управление с помощью клавиатуры.

Описание методов структуры `DebugRemoteControl`:

1. Метод `ui(&mut self, ui: &mut Ui)`. Он отвечает за отображение пользовательского интерфейса контроллера. Метод принимает ссылки на `Ui`, представляющий интерфейс пользователя, и обновляет его содержимое. Внутри метода создается группа элементов интерфейса. Определяются значения скорости `go_speed` и `turn_speed` для движения вперед и поворота. Создаются переменные `right_speed` и `left_speed` и инициализируются нулями. В зависимости от нажатых клавиш, значения `left_speed` и `right_speed` изменяются соответственно. Затем вызывается метод `send_speed`, который отправляет скорости на удаленный контроллер. Наконец, выводится надпись с текущими значениями `left_speed` и `right_speed`.
2. Метод `send_speed(&mut self, left_motor: f32, right_motor: f32)`. Он отвечает за отправку скорости на удаленный контроллер. Метод принимает значения скорости для поворота влево или вправо и проверяет, отличаются ли они от последних отправленных скоростей. Если скорости совпадают, то метод завершается без отправки данных. В противном случае, метод сериализует скорости в формате `MessagePack` с помощью библиотеки `rmp_serde` и отправляет их через UDP-сокет по адресу `MAINBOARD_REMOTE_CONTROL_ADDR`. Затем метод обновляет значения `last_left_speed` и `last_right_speed`.

3.6 Блок управляющих элементов пользовательского интерфейса

3.6.1 Структура Spider

Структура `Spider` представляет собой модель паука. Внутри структуры

есть публичное поле configs, которое является массивом из восьми элементов типа LegConfig. Этот массив содержит в себе текущее состояние всех сервоприводов роботизированной платформы.

В методе new() структуры Spider происходит инициализация экземпляра паука. Создаются конфигурации для каждой из восьми конечностей роботизированной платформы и записываются в массив configs. Значения для каждой конфигурации берутся из заданных констант. В итоге, метод new() возвращает экземпляр структуры Spider с заполненным массивом configs.

3.6.2 Структура SingleLegGait

Функциональность структуры SingleLegGait связана с управлением ходом одной ноги и обновлением ее позиции и высоты.

Описание полей структуры SingleLegGait:

1. Поле current_pos типа Vec2. Представляет текущую позицию ноги робота-паука.
2. Поле current_height типа f32. Представляет текущую высоту ноги робота-паука.
3. Поле start_time типа Instant. Представляет момент начала движения роботизированной платформы.
4. Поле phase_shift типа f32. Представляющее сдвиг фазы движения роботизированной платформы.
5. Поле back_ground_pos типа Vec2. Представляет позицию ноги на соответствующем этапе движения роботизированной платформы.
6. Поле front_ground_pos типа Vec2. Представляет позицию ноги на соответствующем этапе движения роботизированной платформы.
7. Поле neutral_ground_pos типа Vec2. Представляет позицию ноги на соответствующем этапе движения роботизированной платформы.
8. Поле prestop_pos типа Vec2. Представляет позицию ноги на соответствующем этапе движения роботизированной платформы.
9. Поле cycle типа f32. Представляет текущую фазу движения роботизированной платформы.
10. Поле last_cycle типа f32. Представляет предыдущую фазу движения роботизированной платформы.
11. Поле stopping типа bool. Указывает, происходит ли остановка движения роботизированной платформы.
12. Поле stopped типа bool. Указывает, остановлено ли движение роботизированной платформы.
13. Поле pre_stop типа bool. указывает, находится ли роботизированная платформа в ожидании остановки остановки.

Описание методов структуры SingleLegGait:

1. Метод `new(leg: &Leg, phase_shift: f32)`. Это конструктор, инициализирующий поля структуры `SingleLegGait` на основе переданного объекта `leg` типа `Leg` и значения `phase_shift`. Возвращает новый экземпляр `SingleLegGait`.

2. Метод `start_cycle(&mut self)`. Он запускает новый цикл движения роботизированной платформы. Если движение не остановлено, обновляет время начала хода, сбрасывает текущую фазу и флаги остановки.

3. Метод `stop(&mut self, leg: &mut Leg)`. Он останавливает движение роботизированной платформы и обновляет позицию ноги на разных этапах остановки, основываясь на переданном объекте `leg` типа `Leg` и значениях не описанных в разделе констант.

4. Метод `update_cycle(&mut self)`. Обновляет текущую фазу движения роботизированной платформы на основе времени, прошедшего с начала хода.

5. Метод `update_leg(&mut self, leg: &mut Leg)`. Обновляет позицию ноги робота-паука в соответствии с текущей фазой движения. Использует различные значения не описанных в разделе констант для интерполяции позиции ноги.

3.7 Блок вычисления углов поворотов сервоприводов

3.7.1 Структура Leg

Структура `Leg` представляет модель ноги робота-паука и содержит поля и методы, связанные с кинематикой и позицией конечности.

Описание полей структуры `Leg`:

1. Поле `leg_name` типа `str`. Содержит обозначение ноги робота-паука.

2. Поле `femur_ang` типа `f32`. Представляет угол наклона бедра ноги робота-паука.

3. Поле `tibia_ang` типа `f32`. Представляет угол наклона голени ноги робота-паука.

4. Поле `coxa_ang` типа `f32`. Представляет угол наклона коксального сочленения ноги робота-паука.

5. Поле `len` типа `f32`. Представляет длина ноги робота-паука.

6. Поле `height` типа `f32`. Представляет высоту ноги робота-паука.

7. Поле `ground_position` типа `Vec2`. Представляет позицию ноги робота-паука относительно земли.

8. Поле `attach_pos` типа `Vec2`. Представляет позицию ноги робота-паука относительно точки крепления.

9. Поле `standing_position` типа `Vec2`. Представляет соответствующую позицию ноги робота-паука.

Описание методов структуры `Leg`:

1. Метод `forward_kinematics(&mut self, femur: f32, tibia: f32)`. Он реализует прямую кинематику ноги на основе переданных углов `femur` и `tibia`. Обновляет поля `femur_ang`, `tibia_ang`, `len` и `height` на основе расчетов позиции ноги.

2. Метод `inverse_kinematics(&mut self, x: f32, y: f32)`. Он реализует инверсную кинематику ноги на основе переданных координат `x` и `y`. Обновляет поля `femur_ang` и `tibia_ang` на основе расчетов обратной позиции ноги.

3. Метод `inverse_plane_kinematics(&mut self, x: f32, y: f32)`. Он реализует инверсную кинематику ноги в плоскости на основе переданных координат `x` и `y`. Обновляет поля `len`, `coxa_ang`, `ground_position` и вызывает метод `inverse_kinematics` для обновления положения ноги.

3.7.2 Структура MotorConfig

Структура `MotorConfig` определяет конфигурацию одного сервопривода роботизированной платформы.

Описание полей структуры `MotorConfig`:

1. Поле `board` типа `Board`. Представляет номер контроллера PCA, к которому подключен сервопривод.

2. Поле `pin` типа `u8`. Представляет номер контакта контроллера PCA, к которому подключен сервопривод.

3. Поле `min` типа `f32`. Представляет минимальное значение угла вращения сервопривода.

4. Поле `max` типа `f32`. Представляет максимальное значение угла вращения сервопривода.

5. Поле `r_min` типа `f32`. Представляет минимальное значение радиуса движения соответствующей части ноги робота-паука.

6. Поле `r_max` типа `f32`. Представляет максимальное значение радиуса движения соответствующей части ноги робота-паука.

7. Поле `mid` типа `f32`. Представляет среднее значение угла вращения сервопривода или среднее значение радиуса движения ноги робота-паука.

9. Поле `inversed` типа `bool`. Флаг, указывающий на инвертированное направление поворота сервопривода.

В структуре `MotorConfig` присутствует метод `calc_angle(&self, mut angle: f32)`. Он выполняет вычисления для определения угла вращения мотора на основе заданных конфигурационных параметров.

3.7.3 Структура LegConfig

Структура `LegConfig` представляет конфигурацию одной конечности

роботизированной платформы.

Описание полей структуры `LegConfig`:

1. Поле `coxa` типа `MotorConfig`. Представляет конфигурацию сервопривода коксального сочлинения одной из конечностей роботизированной платформы.
2. Поле `femur` типа `MotorConfig`. Представляет конфигурацию сервопривода бедра одной из конечностей роботизированной платформы.
3. Поле `tibia` типа `MotorConfig`. Представляет конфигурацию сервопривода голени одной из конечностей роботизированной платформы.

Структура LegConfig имеет один метод `calc_leg(&self, coxa: f32, femur: f32, tibia: f32)`, который принимает три аргумента типа `f32`, представляющих собой углы. Данный метод вызывает метод `calc_angle` для каждого из трех сервоприводов ноги робота-паука и передает соответствующие значения для вычисления угла вращения. Результатом метода `calc_leg` является кортеж из трех значений типа `f32`, представляющих вычисленные углы вращения для каждого сервопривода.

3.8 Блок обработки данных с лидара

3.8.1. Структура Notifier

Структура `Notifier` представляет собой простой механизм для синхронизации между потоками.

Описание полей структуры `Notifier`:

1. Поле `variable` типа `Mutex<usize>`. Обеспечивает многопоточный доступ к целочисленной переменной типа `usize`. `Mutex` позволяет только одному потоку получить доступ к переменной в определенный момент времени, блокируя остальные потоки, которые пытаются получить доступ. Это гарантирует, что переменная будет изменяться только одним потоком одновременно, предотвращая состояние гонки и другие проблемы синхронизации.
2. Поле `condvar` типа `Condvar`. Условная переменная используется для ожидания определенного условия и уведомления других потоков о его изменении. `Condvar` позволяет потокам ждать и получать уведомления, когда условие, связанное с `variable`, изменяется.

Описание методов структуры `Notifier`:

1. Метод `add_one(&self)`. Он создает новый экземпляр `Notifier`. Внутри метода инициализируются поля `variable` и `condvar` со значениями по умолчанию.
2. Метод `add_one(&self)`. Он увеличивает значение переменной на единицу и уведомляющий все ожидающие потоки о ее изменении с помощью `condvar.notify_all()`.

3. Метод `set_value(&self, value: usize)`. Он устанавливает значение переменной в заданное значение `value` и уведомляющий все ожидающие потоки о ее изменении с помощью `condvar.notify_all()`.

4. Метод `consume_one(&self)`. Он проверяет, когда значение переменной станет меньше нуля. Если значение равно нулю, метод блокирует поток и освобождает блокировку `variable`, позволяя другим потокам изменять ее. Когда другой поток увеличивает значение переменной и вызывает `condvar.notify_all()`, ожидающий поток пробуждается и продолжает выполнение, уменьшая значение переменной на единицу.

3.8.2 Структура Array2D

`Array2D` представляет собой структуру для работы с двумерными массивами, обеспечивающую методы для доступа к элементам, управления размерами, создания и заполнения массивов.

Описание полей структуры `Array2D`:

1. Поле `width` типа `u32`. Представляет ширину массива.

2. Поле `height` типа `u32`. Представляет высоту массива.

3. Поле `data` типа `Vec<T>`. Содержит фактические данные массива.

Описание методов структуры `Array2D`:

1. Метод `new_from_row_major(width: u32, rows: Vec<T>)`. Он позволяет создать новый экземпляр `Array2D<T>` на основе вектора `rows` типа `Vec<T>`, представляющего данные в порядке по строкам. Метод также принимает параметр `width` типа `u32`, определяющий ширину массива. Он проверяет, что количество элементов в `rows` соответствует ожидаемому размеру массива, и возвращает новый экземпляр `Array2D<T>` с установленными значениями полей.

2. Метод `size(&self)`. Он возвращает размер массива в виде `Vector2<i32>`, где `x` соответствует ширине, а `y` – высоте.

3. Метод `get_width(&self)`. Он возвращает ширину массива.

4. Метод `clamp_to_grid(&self, point: &Point2<i32>)`. Он ограничивает точку `point` типа `Point2<i32>` в пределах размеров массива. Если координаты `x` и `y` точки находятся за пределами массива, они будут сокращены до ближайших граничных значений.

5. Метод `index(&self, point: &Point2<i32>)`. Он возвращает линейный индекс внутри одномерного массива, соответствующий указанной точке `point` типа `Point2<i32>`. Индекс вычисляется на основе ширины массива и координаты x и y точки.

6. Метод `index_grid(&self, point: &Point2<i32>)`. Он возвращает ссылку на элемент массива, соответствующий указанной точке `point` типа `Point2<i32>`. Точка сначала ограничивается размерами массива, а затем используется для вычисления линейного индекса и получения

элемента из внутреннего вектора data.

7. Метод index_grid_mut(&mut self, point: &Point2<i32>). Он возвращает изменяемую ссылку на элемент массива, соответствующий указанной точке point типа Point2<i32>. Точка ограничивается размерами массива, вычисляется линейный индекс, и возвращается изменяемая ссылка на элемент из внутреннего вектора data.

8. Метод index_grid_mut_unclamped(&mut self, point: &Point2<i32>). Он возвращает изменяемую ссылку на элемент массива, соответствующий указанной точке point типа Point2<i32>. Точка не ограничивается размерами массива и используется для вычисления линейного индекса и получения элемента из внутреннего вектора data.

9. Метод from_index_to_xy(&self, index: usize). Он преобразует линейный индекс в координаты x и y внутри двумерного массива. Индекс делится на ширину массива, чтобы получить координату y, а остаток от деления используется для получения координаты x.

10. Метод new(width: u32, height: u32). Он создает новый экземпляр Array2D<T> с заданной шириной и высотой. Все элементы массива инициализируются значением по умолчанию для типа T.

11. Метод new_square(size: u32). Он создает новый квадратный экземпляр Array2D<T> с заданным размером. Размер используется для установки одинаковой ширины и высоты массива.

12. Метод fill(&mut self, element: T). Он заполняет все элементы массива указанным значением element. Значение element должно быть типа T и поддерживать копирование.

13. Метод create_blur_kernel(metres_per_px: f32, kernel_size_px: u32, smear_deviation: f32). Он создает экземпляр Array2D<u8>, представляющий размытое ядро. Этот метод используется для создания ядра размытия изображения. Метод принимает три параметра: metres_per_px, kernel_size_px и smear_deviation, которые влияют на форму и интенсивность размытия. Он вычисляет значения ядра и заполняет внутренний вектор data соответствующими значениями. Значения вектора data ограничиваются от 0 до 255, представляя интенсивность размытия.

3.8.3 Структура Metres

Структура Metres представляет собой обертку над значением типа f32, которое представляет длину или расстояние в метрах. Она реализует несколько трейтов, чтобы обеспечить удобную работу с этими значениями. Эти реализации и методы позволяют выполнять различные операции с значениями типа Metres, такие как умножение, деление, вычитание и присваивание. Например, можно умножать значение Metres на число, делить

значение Metres на число или другое значение Metres, вычитать одно значение Metres из другого и изменять значение Metres с помощью операции вычитания с присваиванием.

Описание методов структуры Metres:

1. Метод deref(&self). Реализация трейта Deref позволяет использовать оператор разыменования для получения ссылки на значение внутри Metres. Тип Target определен как f32, поэтому метод deref возвращает ссылку на значение f32 внутри Metres.

2. Метод mul(self, rhs: f32). Реализация трейта Mul позволяет умножать значения типа Metres на f32. Оператор умножения используется для выполнения умножения. Результатом является новый экземпляр Metres, содержащий умноженное значение.

3. Метод div(self, rhs: f32). Реализация трейта Div позволяет делить значения типа Metres на f32. Оператор деления используется для выполнения деления. Результатом является новый экземпляр Metres, содержащий результат деления.

4. Метод div(self, rhs: Metres). Реализация трейта Div позволяет делить значения типа Metres на другое значение типа Metres. Результатом является значение типа f32, которое представляет отношение между двумя значениями Metres.

5. Метод sub(self, rhs: Self). Реализация трейта Sub позволяет вычитать значения типа Metres. Оператор вычитания используется для выполнения вычитания. Результатом является новый экземпляр Metres, содержащий разницу между двумя значениями.

6. Метод sub_assign(&mut self, rhs: Self). Реализация трейта SubAssign позволяет выполнять операцию вычитания с присваиванием для значений типа Metres. Метод sub_assign изменяет текущий экземпляр Metres, вычитая из него значение Metres.

3.8.4 Структура MetresPerPixel

Структура MetresPerPixel представляет собой обертку над значением типа f32, которое представляет отношение метров к пикселям. Она предоставляет методы для конвертации значений между метрами и пикселями.

Описание методов структуры MetresPerPixel:

1. Метод new(size_in_metres: Metres, size_in_pixels: u32). Он создает новый экземпляр MetresPerPixel на основе размера в метрах и количества пикселей в этом метре. Метод вычисляет отношение метров к пикселям, разделяя размер в метрах на количество пикселей, и возвращает новый экземпляр MetresPerPixel.

2. Метод to_metres(&self, pixels: f32). Он преобразует значение из пикселей в метры. Метод умножает количество пикселей на

отношение метров к пикселям и возвращает результат в метрах.

3. Метод `to_pixels(&self, metres: Metres)`. Он преобразует значение из метров в пиксели. Метод делит количество метров на отношение метров к пикселям и возвращает результат в пикселях.

4. Метод `to_pixels_from_f32_metres(&self, metres: f32)`. Он преобразует значение из метров, переданное в виде `f32`, в пиксели. Метод делит количество метров на отношение метров к пикселям и возвращает результат в пикселях.

5. Метод `point_to_pixels(&self, point: &Point2<f32>)`. Он преобразует двумерную точку из метров в пиксели. Метод вызывает `to_pixels_from_f32_metres` для каждой координаты точки и возвращает новую двумерную точку с координатами в пикселях.

6. Метод `point_to_metres(&self, point: &Point2<i32>)`. Он преобразует двумерную точку из пикселей в метры. Метод вызывает `to_metres` для каждой координаты точки и возвращает новую двумерную точку с координатами в метрах.

7. Метод `vector_to_pixels(&self, vector: &Vector2<f32>)`. Он преобразует двумерный вектор из метров в пиксели. Метод вызывает `to_pixels_from_f32_metres` для каждой компоненты вектора и возвращает новый двумерный вектор с компонентами в пикселях.

8. Метод `vector_to_metres(&self, vector: &Vector2<i32>)`. Он преобразует двумерный вектор из пикселей в метры. Метод вызывает `to_metres` для каждой компоненты вектора и возвращает новый двумерный вектор с компонентами в метрах.

9. Метод `from(value: f32)`. Реализация трейта `From` позволяет конвертировать значение типа `f32` в `MetresPerPixel`. В данной реализации метод `from` просто создает новый экземпляр `MetresPerPixel` на основе переданного значения.

3.8.5 Структура OccupancyGrid

`OccupancyGrid` представляет собой структуру данных, которая хранит информацию о занятости ячеек сетки и предоставляет методы для преобразования координат, доступа к ячейкам и обновления информации о занятости на основе различных операций, таких как сканирование линии и обработка лучей датчика.

Описание полей структуры `OccupancyGrid`:

1. Поле `resolution` типа `MetresPerPixel`. Определяет разрешение сетки в метрах на пиксель.

2. Поле `width` типа `u32`. Ширина сетки в пикселях.

3. Поле `height` типа `u32`. Высота сетки в пикселях.

4. Поле `data` типа `Vec<u8>`. Вектор, содержащий информацию о

состоянии каждой ячейки сетки. Каждая ячейка представлена значением типа u8.

5. Поле hit_count типа Vec<f32>. Вектор, содержащий информацию о количестве попаданий в каждую ячейку. Используется для алгоритма оценки занятости.

6. Поле pass_count типа Vec<f32>. Вектор, содержащий информацию о количестве прохождений через каждую ячейку. Используется для алгоритма оценки занятости.

Описание методов структуры OccupancyGrid:

1. Метод new_empty(width: u32, height: u32, resolution: MetresPerPixel). Конструктор, создающий новый экземпляр OccupancyGrid с заданными параметрами ширины, высоты и разрешения. Инициализирует все ячейки сетки значением UNKNOWN_STATUS.

2. Метод index(&self, x: i32, y: i32). Он возвращает индекс вектора data для заданных координат x и y ячейки.

3. Метод size(&self). Он возвращает размер сетки в виде вектора Vector2<i32>, содержащего ширину и высоту сетки.

4. Метод iter_data(&self). Он возвращает итератор, позволяющий перебирать значения всех ячеек сетки.

5. Метод world_to_grid(&self, point: &Point2<f32>). Он преобразует координаты точки из мировой системы координат в координаты ячейки сетки.

6. Метод grid_to_world(&self, point: &Point2<i32>). Он преобразует координаты ячейки сетки в координаты точки в мировой системе координат.

7. Метод clamp_to_grid(&self, point: &Point2<i32>). Он ограничивает координаты точки, чтобы они не выходили за границы сетки.

8. Метод iter_line<F: FnMut(&mut u8)>(&mut self, start: &Point2<i32>, end: &Point2<i32>, mut func: F). Он итерирует через все ячейки линии, соединяющей две заданные точки, и применяет указанную функцию func к каждой ячейке.

9. Метод iter_line_world<F: FnMut(&mut u8)>(&mut self, start: &Point2<f32>, end: &Point2<f32>, func: F). Он итерирует через все ячейки линии, соединяющей две заданные точки в мировых координатах, и применяет указанную функцию func к каждой ячейке.

10. Метод sensor_ray_world(&mut self, start: &Point2<f32>, end: &Point2<f32>). Он обновляет информацию о занятости ячеек сетки по лучу, идущему от начальной точки до конечной точки в мировых координатах.

11. Метод update_occupancy_grid(&mut self, index: usize). Он обновляет состояние ячейки сетки по заданному индексу.

Вычисляет плотность ячейки на основе количества прохождений и попаданий, и обновляет значение состояния ячейки `data[index]` в соответствии с плотностью.

12. Метод `scan_line<F: FnMut(&u8) -> bool>(&mut self, start: &Point2<i32>, end: &Point2<i32>, mut stop_condition: F)`. Он проходит по линии, соединяющей две заданные точки, и применяет указанное условие `stop_condition` к каждой ячейке по пути. Если условие выполняется для какой-либо ячейки, метод возвращает координаты этой ячейки. В противном случае, возвращается `None`.

13. Метод `index_world_mut(&mut self, point: &Point2<f32>)`. Он возвращает изменяемую ссылку на состояние ячейки сетки по заданным координатам точки в мировых координатах.

14. Метод `index_grid_mut(&mut self, point: &Point2<i32>)`. Он возвращает изменяемую ссылку на состояние ячейки сетки по заданным координатам ячейки.

15. Метод `index_grid(&self, point: &Point2<i32>)`. Он возвращает ссылку на состояние ячейки сетки по заданным координатам ячейки.

16. Метод `index_world(&self, point: &Point2<f32>)`. Он возвращает ссылку на состояние ячейки сетки по заданным координатам точки в мировых координатах.

3.8.6 Структура CorrelationGrid

`CorrelationGrid` представляет собой структуру данных, которая хранит сетку корреляции и предоставляет методы для инициализации, очистки, обновления и добавления данных в сетку. Она используется для оценки степени корреляции между сканированиями и оценкой положения робота в окружающей среде.

Описание полей структуры `CorrelationGrid`:

1. Поле `grid` типа `Array2D<u8>`. Двумерный массив, представляющий сетку корреляции. Каждая ячейка сетки содержит значение типа `u8`, которое указывает на степень корреляции.

2. Поле `resolution` типа `MetresPerPixel`. Разрешение сетки в метрах на пиксель.

3. Поле `size_metres` типа `f32`. Размер сетки в метрах.

4. Поле `top_left_corner` типа `Vector2<f32>`. Вектор, представляющий левый верхний угол сетки корреляции в мировых координатах.

6. Поле `smear_kernel` типа `Array2D<u8>`. Двумерный массив, представляющий ядро размазывания `smear kernel`. Используется для вычисления корреляции.

Описание методов структуры `CorrelationGrid`:

1. Метод new(size_metres: f32, resolution: MetresPerPixel, pos: Vector2<f32>, smear_deviation: f32). Это конструктор, создающий новый экземпляр CorrelationGrid с заданными параметрами размера сетки, разрешения, позиции и стандартного отклонения размазывания. Инициализирует сетку корреляции, устанавливает левый верхний угол и создает ядро размазывания.

2. Метод clear(&mut self). Он очищает сетку корреляции, устанавливая все значения ячеек в 0.

3. Метод set_center(&mut self, center: Vector2<f32>). Он устанавливает центр сетки корреляции в заданной позиции, пересчитывая левый верхний угол на основе размера сетки и нового центра.

4. Метод add_scan(&mut self, scan: &LocalizedRangeScan). ОН добавляет сканирование окружающей среды в сетку корреляции. Проходит по каждой точке в сканировании и вызывает метод add_point для добавления каждой точки в сетку.

5. Метод add_point(&mut self, point: &Point2<f32>). Он добавляет точку в сетку корреляции. Вычисляет позицию точки на сетке, определяет область, в которой будет применено ядро размазывания, и обновляет значения ячеек в этой области на основе значения ядра и текущего значения ячеек.

3.8.7 Структура LocalizedRangeScan

LocalizedRangeScan представляет собой структуру данных, которая хранит локализованное сканирование дальностей. Она содержит информацию о точках сканирования и позиции робота. Структура содержит один метод cloud_at_position(point_cloud: Vec<Point2<f32>>, robot_pose: Pose2D). Это конструктор, создающий новый экземпляр LocalizedRangeScan. Метод принимает вектор точек point_cloud и позицию робота robot_pose. Затем он применяет преобразование к каждой точке в point_cloud, чтобы перевести их в координаты робота в момент сканирования. Затем преобразованные точки сохраняются в поле points нового экземпляра LocalizedRangeScan, а позиция робота сохраняется в поле robot_pose.

Описание полей структуры LocalizedRangeScan:

1. Поле points типа Vec<Point2<f32>>. Вектор точек, представляющих сканирование расстояния до объектов.

2. Поле robot_pose типа Pose2D. Позиция и ориентация робота в мировых координатах в момент выполнения сканирования.

3.8.8 Структура Pose2D

Pose2D представляет собой структуру данных, содержащую позицию и

ориентацию в двумерном пространстве. Она предоставляет методы для создания, преобразования и вычисления разности между позициями. Операторы позволяют выполнять арифметические операции с позициями и скоростями.

Описание полей структуры `Pose2D`:

1. Поле `coords` типа `Point2<f32>`. Координаты позиции в двумерном пространстве. `Point2` – это структура из библиотеки `nalgebra`, представляющая двумерную точку с координатами `f32`.

2. Поле `heading_rad` типа `f32`. Угол ориентации в радианах. Значение 0.0 соответствует направлению вперед.

Описание методов структуры `Pose2D`:

1. Метод `new(coords: Point2<f32>, heading_rad: f32)`. Это конструктор, создающий новый экземпляр `Pose2D` на основе заданных координат и угла ориентации. Метод принимает координаты `coords` и угол ориентации `heading_rad` и создает новый экземпляр `Pose2D` с этими значениями.

2. Метод `from_isometry(isometry: IsometryMatrix2<f32>)`. Он создает новый `Pose2D` на основе матрицы изометрии `IsometryMatrix2` из библиотеки `nalgebra`. Метод извлекает координаты перевода и угол поворота из матрицы изометрии и создает новый экземпляр `Pose2D` с этими значениями.

3. Метод `as_isometry()`. Он преобразует `Pose2D` в матрицу изометрии `IsometryMatrix2`. Метод создает новую матрицу изометрии на основе координат позиции и угла ориентации текущего `Pose2D` и возвращает ее.

4. Метод `sub(self, rhs: Self)`. Это реализация оператора вычитания для `Pose2D`. Позволяет вычислять разность между двумя позициями. Возвращает новый экземпляр `VelocityPose2D`, который представляет собой скорость изменения позиции и ориентации между двумя позициями.

5. Метод `add_assign(&mut self, rhs: VelocityPose2D)`. Это реализация оператора сложения с присваиванием для `Pose2D`. Позволяет изменять текущую позицию и ориентацию, добавляя к ним заданное значение скорости `VelocityPose2D`. Обновляет координаты позиции и угол ориентации текущего `Pose2D`.

3.8.9 Структура VelocityPose2D

`VelocityPose2D` представляет собой структуру данных, которая содержит информацию о скорости изменения позиции и ориентации. Она имеет поля для вектора скорости изменения позиции и значения скорости изменения угла ориентации.

Описание полей структуры VelocityPose2D:

1. Поле translation типа Vector2<f32>. Вектор скорости изменения позиции в двумерном пространстве. Vector2 – это структура из библиотеки nalgebra, представляющая двумерный вектор с компонентами типа f32.

2. Поле angular_speed типа f32. Скорость изменения угла ориентации в радианах в секунду.

Описание методов структуры VelocityPose2D:

1. Метод mul_duration(duration: Duration) . Он умножает текущую скорость на длительность duration. Метод принимает значение времени duration в формате Duration и возвращает новый экземпляр VelocityPose2D, в котором каждая компонента скорости умножена на соответствующее количество секунд из duration.

2. Метод mul_duration(&self, duration: Duration). Реализация оператора сложения для VelocityPose2D и Pose2D. Позволяет складывать скорость VelocityPose2D с позицией Pose2D. Возвращает новый экземпляр Pose2D, в котором координаты позиции увеличены на компоненты скорости изменения позиции, а угол ориентации увеличен на скорость изменения угла.

3.8.10 Структура ScanMatcher

Структура ScanMatcher предоставляет функциональность для сопоставления сканирования с набором сканирований и определения позиции и ориентации робота. Методы используют сетку корреляции и сетки вероятностей для выполнения вычислений.

Описание полей структуры ScanMatcher:

1. Поле correlation_grid типа CorrelationGrid. Сетка корреляции, используемая для хранения информации о сканированиях и их сопоставлении.

2. Поле search_space_probabilities типа Vec<Array2D<f32>>. Вектор сеток вероятностей для пространства поиска. Каждый элемент вектора представляет собой двумерный массив Array2D вероятностей типа f32, связанный с определенным углом вращения.

3. Поле search_space_resolution типа MetresPerPixel. Разрешение пространства поиска в метрах на пиксель.

4. Поле search_space_size типа f32. Размер пространства поиска в метрах.

5. Поле scan_circle_buffer типа Vec<LocalizedRangeScan>. Буфер для хранения сканирований типа LocalizedRangeScan.

Описание методов структуры ScanMatcher:

1. Метод new() . Он создает новый экземпляр ScanMatcher с

начальными значениями полей. Внутри метода происходит инициализация сетки корреляции, вычисление размера пространства поиска и создание соответствующих сеток вероятностей.

2. Метод `match_scan_against_set_of_scans(scan: &LocalizedRangeScan, set_of_scans: &Vec<LocalizedRangeScan>)`. Это основной метод для сопоставления сканирования с набором сканирований и определения позиции и ориентации робота. Он принимает текущее сканирование `scan` и набор сканирований `set_of_scans` в виде ссылок на `LocalizedRangeScan`. Внутри метода происходит инициализация сетки корреляции, вычисление вероятностей сопоставления, а затем вычисление позиции и ориентации робота на основе полученных вероятностей.

3. Метод `default()`. Он возвращает стандартное значение `ScanMatcher`, которое эквивалентно вызову `new()`.

3.8.11 Структура GlobalConfig

`GlobalConfig` представляет собой структуру, содержащую конфигурации для инициализации и работы алгоритма `SLAM`.

Описание полей структуры `GlobalConfig`:

1. Поле `icp_slam_init_config` типа `IcpSlamInitConfig`. Конфигурация для инициализации алгоритма SLAM. Этот объект содержит параметры, необходимые для начальной настройки SLAM, такие как размеры сетки, точность сопоставления и другие параметры, которые используются в начальном этапе SLAM.

2. Поле `icp_slam_config` типа `IcpSlamConfig`. Конфигурация для работы алгоритма SLAM. Этот объект содержит параметры, используемые во время работы SLAM, такие как максимальное количество итераций, пороговые значения для сопоставления и другие параметры, которые влияют на точность и производительность алгоритма.

3.8.12 Структура IcpSlamInitConfig

Структура `IcpSlamInitConfig` представляет собой конфигурацию для инициализации алгоритма ICP (Iterative Closest Point) SLAM. Она имеет один метод `default()`, реализующий стандартное значение для `IcpSlamInitConfig`. Он возвращает новый экземпляр `IcpSlamInitConfig` с предустановленными значениями. Внутри метода используются значения по умолчанию для `Pose2D`, `map_size_pixels` и `map_size_metres`.

Описание полей структуры `IcpSlamInitConfig`:

1. Поле `start_position` типа `Pose2D`. Начальная позиция робота в двухмерном пространстве. Это объект типа `Pose2D`, который содержит

координаты и угол поворота робота.

2. Поле `map_size_pixels` типа `u32`. Размер карты в пикселях. Это значение указывает количество пикселей, используемых для представления карты окружающей среды.

3. Поле `map_size_metres` типа `Metres`. Размер карты в метрах. Это объект типа `Metres`, который представляет размещение карты в физических единицах.

3.8.13 Структура IcpSlamConfig

Структура `IcpSlamConfig` представляет собой конфигурацию для работы алгоритма ICP SLAM. Она имеет один метод `default()`, реализующий стандартное значение для `IcpSlamConfig`. Он возвращает новый экземпляр `IcpSlamConfig` с предустановленными значениями. Внутри метода используются значения по умолчанию для каждого поля.

Описание полей структуры `IcpSlamConfig`:

1. Поле `move_threshold_for_adding_points` типа `Metres`. Пороговое значение перемещения, необходимое для добавления новых точек в карту. Если перемещение робота превышает это значение, новые точки будут добавлены в карту. Здесь `Metres` представляет расстояние в метрах.

2. Поле `max_iterations_icp` типа `usize`. Максимальное количество итераций, выполняемых алгоритмом ICP, в процессе сопоставления сканирования. Это значение определяет, сколько раз алгоритм будет повторно вычислять сопоставление, чтобы достичь наилучшего результата.

3. Поле `add_points_maximum_distance` типа `Metres`. Максимальное расстояние между точками, при котором они будут добавлены в карту. Если расстояние между двумя точками сканирования меньше этого значения, то эти точки будут объединены в одну точку в карте.

4. Поле `filter_scan_points` типа `Metres`. Пороговое значение для фильтрации точек сканирования. Точки, находящиеся на расстоянии меньше этого значения от робота, будут отфильтрованы и не будут использоваться в алгоритме.

3.8.14 Структура IcpSlamMap

`IcpSlamMap` представляет собой карту, используемую в алгоритме ICP SLAM. Она содержит информацию о занятости ячеек сетки и предоставляет функциональность для добавления сканирований в карту, а также для поиска ближайших точек в пределах заданного радиуса.

Описание полей структуры `IcpSlamMap`:

1. Поле `map` типа `OccupancyGrid`. Сетка занятости, представляющая карту окружающей среды. Это объект типа `OccupancyGrid`, который хранит

информацию о занятости каждой ячейки сетки.

2. Поле `kd_tree` типа `MyKdTree`. Дерево, используемое для эффективного поиска ближайших точек в картографической структуре. Это объект типа `MyKdTree`, который содержит точки, представленные в карте, и предоставляет функциональность для поиска ближайших точек.

Описание методов структуры `IcpSlamMap`:

1. Метод `new(map: OccupancyGrid, kd_tree: MyKdTree)`. Конструктор, создающий новый экземпляр `IcpSlamMap` с заданной картой и деревом.

2. Метод `add_scan(&mut self)`. Он позволяет добавить набор точек в карту. Этот метод обновляет карту и дерево на основе новых данных сканирования.

3. Метод `get_nearby_points(&self, origin: Point2<f32>, radius: f32)`. Он возвращает список ближайших точек к заданной точке `origin` внутри заданного радиуса `radius`. Он использует дерево для эффективного поиска ближайших точек и возвращает результат в виде вектора точек.

3.8.15 Структура RobotState

Структура `RobotState` представляет состояние робота.

Описание полей структуры `RobotState`:

1. Поле `robot_position` типа `Pose2D`. Положение робота в двухмерном пространстве. Это объект типа `Pose2D`, который содержит координаты и угол поворота робота. Поле `robot_position` указывает текущее положение робота.

2. Поле `robot_composite_velocity` типа `VelocityPose2D`. Скорость робота, которая включает линейную скорость и скорость поворота. Это объект типа `VelocityPose2D`, который содержит значения линейной скорости и скорости поворота робота. Поле `robot_composite_velocity` указывает текущую скорость робота.

3.8.16 Структура IcpSlamData

`IcpSlamData` представляет собой структуру данных для алгоритма ICP SLAM. Он содержит информацию о положении, скорости робота, последней добавленной позиции в карту, карту окружающей среды, точки сканирования и их сопоставление, а также конфигурационные параметры для алгоритма. Эта структура используется для хранения и обмена данными между различными компонентами алгоритма.

Описание полей структуры `IcpSlamData`:

1. Поле `robot_position` типа `Pose2D`. Положение робота в двухмерном пространстве. Это объект типа `Pose2D`, который содержит

координаты и угол поворота робота.

2. Поле `robot_composite_velocity` типа `VelocityPose2D`. Скорость робота, которая включает линейную скорость и скорость поворота. Это объект типа `VelocityPose2D`, который содержит значения линейной скорости и скорости поворота робота.

3. Поле `last_added_pose` типа `Pose2D`. Последняя добавленная позиция в карту. Это объект типа `Pose2D`, который содержит координаты и угол поворота последней добавленной позиции.

4. Поле `my_kd_tree` типа `Arc<Mutex<MyKdTree>>`. Общий доступ к дереву, используемому для эффективного поиска ближайших точек в картографической структуре. `Arc<Mutex<MyKdTree>>` обеспечивает многопоточный доступ к дереву.

5. Поле `map` типа `OccupancyGrid`. Сетка занятости, представляющая карту окружающей среды. Это объект типа `OccupancyGrid`, который хранит информацию о занятости каждой ячейки сетки.

6. Поле `recent_matched_scan` типа `Vec<Point2<f32>>`. Недавно сопоставленные точки сканирования. Это вектор точек типа `Point2<f32>`, которые были недавно сопоставлены с точками на карте.

Поле `points_pre_match` типа `Vec<Point2<f32>>`. Точки сканирования до сопоставления. Это вектор точек типа `Point2<f32>`, которые были получены в процессе сканирования до выполнения сопоставления с картой.

7. Поле `config` типа `IcpSlamConfig`. Конфигурация для работы алгоритма ICP SLAM. Это объект типа `IcpSlamConfig`, который содержит различные параметры и настройки для алгоритма.

3.8.17 Структура IcpSlam

Структура `IcpSlam` представляет собой реализацию алгоритма ICP SLAM.

Описание полей структуры `IcpSlam`:

1. Поле `map` типа `OccupancyGrid`. Сетка занятости, представляющая карту окружающей среды. Это объект типа `OccupancyGrid`, который хранит информацию о занятости каждой ячейки сетки.

2. Поле `robot_position` типа `Pose2D`. Положение робота в двухмерном пространстве. Это объект типа `Pose2D`, который содержит координаты и угол поворота робота.

3. Поле `robot_composite_velocity` типа `VelocityPose2D`. Скорость робота, которая включает линейную скорость и скорость поворота. Это объект типа `VelocityPose2D`, который содержит значения линейной скорости и скорости поворота робота.

4. Поле `last_added_pose` типа `Pose2D`. Последняя добавленная

позиция в карту. Это объект типа Pose2D, который содержит координаты и угол поворота последней добавленной позиции.

5. Поле my_kd_tree типа MyKdTree. Дерево, используемое для эффективного поиска ближайших точек в картографической структуре.

6. Поле recent_matched_scan типа Vec<Point2<f32>>. Недавно сопоставленные точки сканирования. Это вектор точек типа Point2<f32>, которые были недавно сопоставлены с точками на карте.

7. Поле config типа IcpSlamConfig. Конфигурация для работы алгоритма ICP SLAM. Это объект типа IcpSlamConfig, который содержит различные параметры и настройки для алгоритма.

8. Поле points_pre_match типа Vec<Point2<f32>>. Точки сканирования до сопоставления. Это вектор точек типа Point2<f32>, который используется для хранения точек сканирования перед их сопоставлением.

Описание методов структуры IcpSlam:

1. Метод new(init_config: &IcpSlamInitConfig, config: IcpSlamConfig). Конструктор, который создает новый экземпляр структуры IcpSlam на основе исходной конфигурации init_config и настроек config.

2. Метод rebuild_tree(&mut self). Перестраивает дерево my_kd_tree.

3. Метод update_map(&mut self). Обновляет карту map на основе недавно сопоставленных точек recent_matched_scan.

4. Метод clear_points(&mut self). Удаляет точки из дерева my_kd_tree, которые находятся в ячейках сетки, имеющих низкую степень занятости, определенную пороговым значением MAP_KEEP_POINT_THRESHOLD.

5. Метод update_lidar(&mut self, lidar_data: LidarResponse). Обновляет данные лидара lidar_data и выполняет шаг алгоритма ICP SLAM. Возвращает результат в виде структуры LocalizedRangeScan, содержащей точки сканирования и текущую позицию робота.

6. Метод algorithm_step(&mut self, lidar_data: LidarResponse). Выполняет один шаг алгоритма ICP SLAM на основе данных лидара lidar_data. Внутри этого метода происходит сопоставление точек сканирования с точками на карте, обновление позиции робота и добавление новых точек в карту, если превышен порог перемещения move_threshold_for_adding_points.

7. Метод first_scan(&mut self, lidar_data: LidarResponse). Обрабатывает первое сканирование lidar_data и инициализирует карту и позицию робота на основе полученных данных. Возвращает результат в виде структуры LocalizedRangeScan,

содержащей точки сканирования и текущую позицию робота.

8. Метод `get_map_points(&self)`. Возвращает точки, хранящиеся в дереве `my_kd_tree` в виде облака точек `PointCloud`.

9. Метод `get_lidar_matched(&self)`. Возвращает недавно сопоставленные точки сканирования в виде облака точек `PointCloud`.

10. Метод `get_position(&self)`. Возвращает текущую позицию робота в виде кортежа, содержащего координаты и угол поворота.

11. Метод `get_map(&self)`. Возвращает текущую карту окружающей среды в виде объекта `OccupancyGrid`.

3.8.18 Структура MyKdTree

Структура `MyKdTree` представляет собой реализацию дерева.

Описание полей структуры `MyKdTree`:

1. Поле `kd_tree` типа `Option<KdTree>`. Опциональное поле, содержащее дерево. Если дерево не создано, значение равно `None`.

2. Поле `points` типа `Vec<Point2<f32>>`. Вектор точек типа `Point2<f32>`, которые хранятся в дереве.

Описание методов `MyKdTree`:

1. Метод `new()`. Конструктор, который создает новый экземпляр структуры `MyKdTree` с пустыми полями `kd_tree` и `points`.

2. Метод `is_empty()`. Возвращает `true`, если дерево пустое, иначе возвращает `false`.

3. Метод `add_points_within_range(points: &Vec<Point2<f32>>, range_limit: f32)`. Добавляет точки из вектора `points` в дерево, если расстояние до ближайшей точки в дереве больше `range_limit`.

4. Метод `retain_points(retain_condition: F)`. Оставляет только те точки в дереве, для которых выполнено условие, заданное через замыкание `retain_condition`. Метод возвращает `true` для точек, которые должны быть оставлены, и `false` для точек, которые должны быть удалены.

5. Метод `rebuild_tree()`. Перестраивает дерево, собирая все точки из вектора `points`.

6. Метод `set_vector_points_into_tree()`. Создает новое дерево или перестраивает существующее на основе точек из вектора `points`.

7. Метод `get_kdtree()`. Возвращает ссылку на дерево.

8. Метод `get_points()`. Возвращает ссылку на вектор точек.

3.8.19 Структура IcpConfig

Структура `IcpConfig` представляет собой конфигурацию для

алгоритма ICP.

Описание полей структуры `IcpConfig`:

1. Поле `max_iterations` типа `usize`. Максимальное количество итераций алгоритма ICP.

2. Поле `max_distance_for_nn_matching` типа `f32`. Максимальное расстояние между точками при поиске ближайшего соседа `nearest neighbor matching`. Если расстояние между двумя точками превышает это значение, они не будут считаться соответствующими.

3. Поле `delta_error_stop` типа `f32`. Пороговое значение изменения ошибки между последовательными итерациями, при котором алгоритм ICP останавливается.

4. Поле `min_error_stop` типа `f32`. Минимальное значение ошибки, при достижении которого алгоритм ICP останавливается.

Описание методов структуры `IcpConfig`:

1. Метод `max_distance_for_nn_matching(max_distance_for_nn_matching: f32)`. Устанавливает значение поля `max_distance_for_nn_matching` и возвращает измененную структуру `IcpConfig`. Этот метод используется для установки значения максимального расстояния для ближайшего соседа.

2. Метод `default()`. Реализация `default()` из трейта `Default`, который создает и возвращает новый экземпляр структуры `IcpConfig` с значениями по умолчанию для всех полей.

3.8.20 Структура SharedData

`SharedData` представляет собой структуру для общих данных, используемых при навигации. `SharedData` имеет метод `from_config(config: &NavigationConfig)`, который создает новый экземпляр структуры на основе переданной конфигурации `NavigationConfig`. Внутри метода создаются и инициализируются различные данные, такие как `BigMap`, `IcpSlamMap`, `Pose2D`, `SlamDebugData`, `MotionEstimation`, `ScanMatcher` и `ScanRingBuffer`. Затем эти данные помещаются в соответствующие мьютексы и возвращаются в виде экземпляра `SharedData`.

Описание полей структуры `SharedData`:

1. Поле `big_map` типа `RwLock<BigMap>`. Защищенный мьютексом контейнер `BigMap`, который представляет большую карту.

2. Поле `slam_map` типа `RwLock<IcpSlamMap>`. Защищенный мьютексом контейнер `IcpSlamMap`, который представляет карту для алгоритма ICP SLAM.

3. Поле `last_added_pose` типа `RwLock<Pose2D>`. Защищенный

мьютексом контейнер Pose2D, который хранит последнюю добавленную позицию.

4. Поле debug_data типа RwLock<SlamDebugData>. Защищенный мьютексом контейнер SlamDebugData, который содержит отладочные данные для алгоритма SLAM.

5. Поле motion_estimation типа RwLock<MotionEstimation>. Защищенный мьютексом контейнер MotionEstimation, который представляет оценку движения.

6. Поле correlative_scan_matching типа RwLock<ScanMatcher>. Защищенный мьютексом контейнер ScanMatcher, который используется для корреляционного сопоставления сканирований.

7. Поле scan_ring_buffer типа RwLock<ScanRingBuffer>. Защищенный мьютексом контейнер ScanRingBuffer, представляющий кольцевой буфер для сканирований.

3.8.21 Структура NavigationModuleHandle

Структура NavigationModuleHandle предоставляет интерфейс для взаимодействия с навигационным модулем.

Описание полей структуры NavigationModuleHandle:

1. Поле input_data типа Sender<InputData>. Является отправителем и представляет канал для отправки входных данных типа InputData. Он используется для передачи данных внутрь навигационного модуля.

2. Поле output_data типа Receiver<OutputData>. Является приемником и представляет канал для получения выходных данных типа OutputData. Оно используется для получения результатов работы навигационного модуля.

3. Поле shared_data типа Arc<SharedData>. Является общим указателем на данные типа SharedData. Он используется для совместного доступа к общим данным между различными модулями или компонентами навигационной системы. Arc обеспечивает потокобезопасность и разделение владения данными.

3.8.22 Структура NavigationConfig

Структура NavigationConfig представляет собой конфигурацию для модуля навигации.

Описание полей структуры NavigationConfig:

1. Поле icp_slam_init типа IcpSlamInitConfig. Представляет собой конфигурацию для инициализации алгоритма ICP SLAM. Оно содержит

параметры, такие как размер карты и начальная позиция для алгоритма.

2. Поле `icp_slam_config` типа `IcpSlamConfig`. Представляет собой конфигурацию для алгоритма ICP SLAM. Оно содержит параметры, такие как количество итераций, пороговые значения ошибки и другие настройки, влияющие на работу алгоритма.

3.8.23 Структура NavigationModule

Структура `NavigationModule` представляет собой модуль навигации.

Описание полей структуры `NavigationModule`:

1. Поле `input_data` типа `Receiver<InputData>`. Является приемником и представляет канал для получения входных данных типа `InputData`. Он используется для получения данных от других компонентов системы и передачи их внутрь модуля навигации.

2. Поле `output_data` типа `Sender<OutputData>`. Является отправителем и представляет канал для отправки выходных данных типа `OutputData`. Он используется для передачи результатов работы модуля навигации другим компонентам системы.

3. Поле `shared_data` типа `Arc<SharedData>`. Является общим указателем на данные типа `SharedData`. Он используется для совместного доступа к общим данным между различными модулями или компонентами навигационной системы. `Arc` обеспечивает потокобезопасность и разделение владения данными.

4. Поле `config` типа `NavigationConfig`. Представляет собой конфигурацию модуля навигации типа `NavigationConfig`. Оно содержит настройки и параметры, которые влияют на работу модуля.

Описание методов структуры `NavigationModule`:

1. Метод `new(config: NavigationConfig)`. Конструктор создает новый экземпляр модуля навигации на основе заданной конфигурации. Он инициализирует каналы `input_data` и `output_data` с ограниченной емкостью 10 элементов. Затем создается общий указатель `shared_data` на основе конфигурации. Запускается новый поток выполнения, где вызывается метод `run` для обработки входных данных. Наконец, возвращается экземпляр `NavigationModuleHandle`, который представляет управление модулем навигации.

2. Метод `run(self)`. Является основной функцией модуля навигации. Он работает в бесконечном цикле, ожидая получение входных данных через `input_data`. После получения данных происходит их обработка в зависимости от типа сообщения. Например, если получено сообщение типа `InputData::LidarScan`, вызывается метод `process_lidar_scan` для обработки сканирования лидара. Если получено сообщение типа

InputData::DebugGps, выполняется обновление данных GPS (Global Positioning System). Обработка данных зависит от текущей конфигурации модуля и использует общие данные из shared_data.

3. Метод detect_obstacles(&mut self, interpreted_lidar_data: &LocalizedRangeScan). Предназначен для обнаружения препятствий на основе интерпретированных данных сканирования лидара.

4. Метод process_lidar_scan(&mut self, interpreted_lidar_data: &LocalizedRangeScan, time: SystemTime). Выполняет обработку сканирования лидара. Он принимает интерпретированные данные сканирования interpreted_lidar_data и время time. В методе происходит соответствующая обработка сканирования, такая как сопоставление с картой, обновление позиции робота и т. д. Результаты обработки сохраняются в общих данных shared_data.

3.8.24 Структура ScanRingBuffer

Общая структура ScanRingBuffer предоставляет функциональность для хранения сканирований лидара в кольцевом буфере. Когда новое сканирование добавляется методом add_scan, оно помещается в конец буфера. Если количество элементов в буфере превышает заданную длину, старые элементы удаляются, чтобы поддерживать заданное ограничение. Это позволяет сохранять только наиболее актуальные сканирования и обеспечивает ограниченный размер буфера.

Описание полей структуры ScanRingBuffer:

1. Поле buf типа Vec<LocalizedRangeScan>. Является вектором и представляет буфер сканирования типа LocalizedRangeScan. Он используется для хранения сканирований лидара.

2. Поле len типа usize. Представляет собой длину буфера, т.е. максимальное количество элементов, которые могут быть хранены в буфере.

Описание методов структуры ScanRingBuffer:

1. Метод new(len: usize). Конструктор создает новый экземпляр ScanRingBuffer с заданной длиной len. Он инициализирует пустой вектор buf и сохраняет заданную длину.

2. Метод add_scan(&mut self, scan: LocalizedRangeScan). Добавляет сканирование scan в буфер. Он помещает сканирование в конец вектора buf с помощью метода push. Затем проверяет, превышает ли текущая длина буфера заданную длину.

3.8.25 Структура MapCell

Структура MapCell предоставляет функциональность для хранения

информации о ячейке карты. Она содержит верхний левый угол ячейки, точки, относящиеся к этой ячейке, и сетку занятости, которая отображает информацию о занятости внутри ячейки.

Описание полей структуры `MapCell`:

1. Поле `top_left` типа `Point2<f32>`. Представляет верхний левый угол ячейки типа `Point2<f32>`. Он содержит координаты точки верхнего левого угла ячейки в формате с плавающей запятой.

2. Поле points типа `Vec<Point2<f32>>`. Является вектором и представляет точки типа `Point2<f32>`. Он используется для хранения точек, относящихся к данной ячейке карты.

3. Поле `occupancy_grid` типа `OccupancyGrid`. Представляет сетку занятости типа `OccupancyGrid`. Она используется для представления информации о занятости внутри ячейки карты.

Описание методов структуры `MapCell`:

1. Метод `new(top_left: Point2<f32>, cell_size: Metres, occupancy_grid_resolution: MetresPerPixel)`. Конструктор создает новый экземпляр `MapCell` на основе заданных параметров. Он инициализирует верхний левый угол ячейки `top_left`, создает пустой вектор `points` и инициализирует сетку занятости `occupancy_grid`. Размер сетки занятости `occupancy_grid_pixel_size` вычисляется на основе разрешения и размера ячейки.

2. Метод `get_points_and_grid(&mut self)`. Возвращает изменяемые ссылки на вектор точек и сетку занятости. Он используется для получения доступа к данным ячейки карты и их последующего изменения.

3.8.26 Структура MapChunk

Структура `MapChunk` предоставляет функциональность для хранения и обработки информации о части карты. Она содержит верхний левый угол части карты, размер ячейки, и массив ячеек, которые могут быть одновременно изменены или прочитаны. Методы предоставляют функции для получения точек из ячеек, добавления точек.

Описание полей структуры `MapChunk`:

1. Поле `top_left_corner` типа `Point2<f32>`. Представляет верхний левый угол части карты типа `Point2<f32>`. Он содержит координаты точки верхнего левого угла части карты в формате с плавающей запятой.

2. Поле `cell_size` типа `Metres`. Представляет размер ячейки типа `Metres`. Поле определяет размер каждой ячейки в части карты.

3. Поле `chunk` типа `Array2D<RwLockWriteGuard<'cell, MapCell>>`. Представляет двумерный массив `Array2D` ячеек типа `MapCell`. Поле хранит ячейки карты с возможностью одновременной записи

и чтения RwLockWriteGuard.

Описание методов структуры MapChunk:

1. Метод new(top_left_corner: Point2<f32>, cell_size: Metres, chunk: Array2D<RwLockWriteGuard<'cell, MapCell>>). Конструктор создает новый экземпляр MapChunk на основе заданных параметров. Он инициализирует поле chunk, cell_size и top_left_corner.

2. Метод get_points(&self). Возвращает все точки, содержащиеся в ячейках части карты. Метод перебирает все ячейки в chunk и собирает все точки из каждой ячейки в вектор points, который затем возвращается.

3. Метод get_cell_points(&self). Возвращает точки для каждой ячейки в виде вектора векторов. Метод перебирает все ячейки в chunk и клонирует точки из каждой ячейки в вектор points, который затем добавляется в результирующий вектор points.

4. Метод add_points(&mut self, points: &Vec<Point2<f32>>). Добавляет точки в часть карты. Метод перебирает все переданные точки и определяет соответствующую ячейку, основываясь на top_left_corner, cell_size и координатах точки. Затем точка добавляется в соответствующую ячейку.

5. Метод filter_points(&mut self). Удаляет точки с области карты. Метод перебирает все ячейки в chunk и удаляет точки, которые имеют значение занятости chunk_grid больше 128.

6. Метод filter_temp_map(&mut self, grid: &OccupancyGrid). Удаляет точки, основываясь на переданной сетке занятости grid. Метод перебирает все ячейки в chunk и удаляет точки, которые имеют значение занятости grid.index_world больше 128.

3.9 Блок отображения карты

3.9.1 Структура LidarWidget

Структура LidarWidget в Rust представляет собой виджет для визуализации данных с лидара. В ней реализован трейт Default, который позволяет создать экземпляр LidarWidget со значениями по умолчанию.

Описание полей структуры LidarWidget:

1. Поле lidar_data типа roboq_messages::lidar::LidarResponse. Хранит данные, полученные с лидара.

2. Поле metres_per_px типа f32. Представляет соотношение между метрами и пикселями при визуализации данных. Это значение используется для преобразования измерений дистанции из метров в пиксели при отображении точек на экране.

Описание методов структуры LidarWidget:

1. Метод new(). Он создает новый экземпляр LidarWidget. Внутри метода инициализируются поля lidar_data и metres_per_px, используя значения по умолчанию.

2. Метод ui(&mut self, ui: &mut Ui). Он отвечает за отображение пользовательского интерфейса виджета. Метод принимает ссылки на Ui, который представляет интерфейс пользователя, и обновляет его содержимое. Внутри метода создается вертикальная группа элементов интерфейса. Внутри группы отображается надпись, а затем вызывается метод allocate_painter, который выделяет пространство для рисования, используя размеры LIDAR_WIDGET_SIZE. После этого вызывается метод draw_lidar_data, который отрисовывает данные лидара.

3. Метод draw_lidar_data(&mut self, painter: &Painter). Он отвечает за отображение данных лидара. Метод принимает ссылку на Painter, который используется для рисования графики. Внутри метода сначала рисуется перекрестие, а затем для каждой точки из lidar_data.points вычисляются координаты и отрисовывается круговой маркер с помощью метода circle на переданном painter.

4. Метод draw_crosshair(&mut self, painter: &Painter). Он отвечает за отображение перекрестия на экране. Метод принимает ссылку на Painter и использует его для рисования двух линий, образующих перекрестие. Длина и цвет линий определены константами внутри метода.

3.9.2 Структура MapMarker

Структура MapMarker предоставляет простой способ создания маркеров на карте с указанием их позиции, цвета, радиуса и поворота.

Описание полей структуры MapMarker:

1. Поле pos_px типа Pos2. Представляет позицию маркера на карте относительно верхнего левого угла виджета. Pos2 содержит координаты x и y в пикселях.

2. Поле color типа Color32. Определяет цвет маркера. Color32 представляет цвет в формате RGBA (red green blue alpha), где каждый канал: красный, зеленый, синий и альфа, представлен 8-битным значением.

3. Поле radius типа f32. Определяет радиус маркера. Значение радиуса является фиксированным и неизменным для данного маркера.

4. Поле rotation типа Option<f32>. Представляет поворот маркера в радианах. Поворот является необязательным полем и может быть Some для указания конкретного угла поворота, либо None, если маркер не поворачивается.

Описание методов структуры MapMarker:

1. Метод new(pos: Point2<f32>, color: Color32, radius:

f32). Он позволяет создать новый экземпляр MapMarker. Метод принимает позицию pos типа Point2<f32>, цвет color типа Color32 и радиус radius типа f32. Внутри метода создается экземпляр MapMarker с установленными значениями полей pos_px, color, radius и rotation, для которого задано знаение Some.

2. Метод with_rotation(pos: Point2<f32>, color: Color32, radius: f32, rotation: f32). Он позволяет создать новый экземпляр MapMarker с указанным поворотом. Метод принимает позицию pos типа Point2<f32>, цвет color типа Color32, радиус radius типа f32 и угол поворота rotation типа f32. Внутри метода создается экземпляр MapMarker с установленными значениями полей pos_px, color, radius и rotation, для которого задано значение Some.

3.9.3 Структура MapWidget

Структура MapWidget представляет собой виджет для визуализации карты.

Описание полей структуры MapWidget:

1. Поле widget_size_px типа Vec2. Представляет размер виджета в пикселях. Vec2 содержит координаты x и y в пикселях.

2. Поле texture типа TextureId. Идентификатор текстуры, используемой для визуализации карты.

3. Поле texture_pixel_data типа Option<ColorImage>. Содержит пиксельные данные текстуры. Может быть Some, если данные доступны, или None, если данные еще не были обновлены.

4. Поле map_size_pixels типа Vector2<i32>. Представляет размер карты в пикселях.

5. Поле resolution типа MetresPerPixel. Определяет разрешение карты в метрах на пиксель.

6. Поле map_markers типа Vec<MapMarker>. Содержит список маркеров на карте.

7. Поле show_markers типа bool. Указывает, следует ли отображать маркеры на карте.

8. Поле zoom типа f32. Определяет масштаб карты.

9. Поле offset типа Vec2. Определяет смещение карты.

10. Поле is_map_clicked типа bool. Указывает, было ли выполнено нажатие на карте.

11. Поле hover_position типа Option<Pos2>. Содержит позицию указателя мыши над картой. Может быть Some, если указатель мыши находится над картой, или None, если указатель мыши не над картой.

Описание методов структуры MapWidget:

1. Метод new(cc: &egui::Context, widget_size_px: Vec2).

Он позволяет создать новый экземпляр MapWidget. Метод принимает ссылку на контекст cc типа egui::Context и размер виджета widget_size_px типа Vec2. Внутри метода создается экземпляр MapWidget с установленными значениями полей, включая инициализацию текстуры и других полей.

2. Метод update_texture(&mut self, grid: &OccupancyGrid). он позволяет обновить текстуру карты на основе сетки занятости grid типа OccupancyGrid. Метод принимает ссылку на сетку занятости grid и обновляет поле texture_pixel_data с пиксельными данными, а также обновляет поля resolution и map_size_pixels.

3. Метод update_correlation_grid(&mut self, grid: &CorrelationGrid). Он позволяет обновить текстуру карты на основе сетки корреляции grid типа CorrelationGrid. Метод принимает ссылку на сетку корреляции grid и обновляет поле texture_pixel_data с пиксельными данными, а также обновляет поля resolution и map_size_pixels.

4. Метод update_probability_grid(&mut self, grid: &Array2D<f32>). Он позволяет обновить текстуру карты на основе сетки вероятности grid типа Array2D<f32>. Метод принимает ссылку на сетку вероятности grid и обновляет поле texture_pixel_data с пиксельными данными, а также устанавливает фиксированное разрешение resolution и обновляет поле map_size_pixels.

5. Метод add_markers(&mut self, points: &[Point2<f32>], color: Color32, radius: f32). Он позволяет добавить маркеры на карту. Метод принимает список точек points типа [Point2<f32>], которые представляют позиции маркеров, цвет color типа Color32, определяющий цвет маркеров, и радиус radius типа f32, определяющий размер маркеров. Метод добавляет маркеры в поле map_markers структуры MapWidget.

6. Метод clear_markers(&mut self). Он позволяет удалить все маркеры с карты. Метод очищает поле map_markers структуры MapWidget.

7. Метод draw(&mut self, ui: &mut egui::Ui). Он выполняет отрисовку виджета на пользовательском интерфейсе. Метод принимает ссылку на интерфейс ui типа egui::Ui и использует функции интерфейса для отображения карты, маркеров и других элементов.

8. Метод handle_event(&mut self, event: &egui::Event, ui: &mut egui::Ui). Он обрабатывает события, связанные с виджетом. Метод принимает ссылки на событие event типа egui::Event и интерфейс ui типа egui::Ui, и возвращает ответ Response типа egui::Response, который указывает, как виджет обработал событие.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе описана работа ключевых алгоритмов разработанного программного средства. В схеме программы ГУИР.400201.106 ПД.1 и схеме программы ГУИР.400201.106 ПД.2 показаны алгоритмы, реализующие вычисление углов и нахождение координат конечностей роботизированной платформы.

4.1 Алгоритм прямой кинематики

Алгоритм представлен методом `forward_kinematics(&mut self, femur: f32, tibia: f32)`. Его задача состоит в вычислении прямой кинематики для конечностей робота на основе заданных углов.

Описание алгоритма:

Шаг 1. Задать угол поворота бедра `self.femur_ang`.

Шаг 2. Задать угол поворота голени `self.tibia_ang`.

Шаг 3. Преобразовать угол поворота бедра `self.femur_ang` в радианы. Для этого используем метод `to_radians()`.

Шаг 4. Преобразовать угол поворота голени `self.tibia_ang` в радианы. Для этого используем метод `to_radians()`.

Шаг 5. Вычесть 90 градусов из угла поворота бедра `self.femur_ang`. Градусы преобразуются в радианы с помощью метода `to_radians()`.

Шаг 6. К углу поворота голени `self.tibia_ang` прибавляется угол поворота бедра `self.femur_ang`.

Шаг 7. Задать длину бедра `femur_len`.

Шаг 8. Задать длину голени `tibia_len`.

Шаг 9. Умножить длину бедра `femur_len` на косинус угла `self.femur_ang`.

Шаг 10. Умножить длину голени `tibia_len` на синус угла `self.tibia_ang`.

Шаг 11. Получить координаты точки крепления бедра `femur_joint`. Для этого используем метод `pos2()`. Он возвращает вектор из двух значений, полученных на шаге 9 и шаге 10.

Шаг 12. Умножить длину голени `tibia_len` на косинус угла `self.tibia_ang`.

Шаг 13. Умножить длину голени `tibia_len` на синус угла `self.tibia_ang`.

Шаг 14. Получить координаты точки крепления голени `tibia_joint`. Для этого вектор `femur_joint` нужно сложить с вектором, полученным из значений, вычисленных на шаге 12 и шаге 13.

Шаг 15. Сохранить длину вектора перемещения `self.len` как составляющую вектора точки крепления голени `tibia_joint`.

Шаг 15. Сохранить высоту вектора перемещения `self.height` как составляющую вектора точки крепления голени `tibia_joint`.

Шаг 16. Получить углы поворота. Для этого используем функцию `inverse_kinematics()`. Она принимает вектор со значениями `self.len` и `self.height` положения и возвращает кортеж с углами поворота `q1` и `q2`.

Шаг 17. К углу поворота бедра `q1` нужно добавить 90 градусов. Градусы преобразуются в радианы с помощью метода `to_radians()`.

Шаг 18. Если угол поворота бедра `q1` больше максимального угла поворота `TAU`, то перейти к шагу 19, иначе перейти к шагу 20.

Шаг 19. Из угла поворота бедра `q1` вычесть максимальный угол поворота `TAU`.

Шаг 20. Если угол поворота бедра `q1` больше чем полтора числа пи, то перейти к шагу 22, иначе перейти к шагу 21.

Шаг 21. Если угол поворота бедра `q1` меньше чем 0, то перейти к шагу 22, иначе перейти к шагу 23.

Шаг 22. Присвоить углу поворота бедра `q1` значение 0. Перейти к шагу 24.

Шаг 23. Присвоить углу поворота бедра `q1` минимальное значение из двух: `q1` и `PI`.

Шаг 24. Сохранить угол поворота бедра `q1` как `self.femur_ang` в градусах. Для этого используем метод `to_degrees()`.

Шаг 25. Сохранить угол поворота голени `q2` как `self.tibia_ang` в градусах. Для этого используем метод `to_degrees()`.

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Текст раздела

5.1 Подраздел

Текст раздела

5.2 Подраздел

Текст раздела

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Текст раздела

6.1 Подраздел

Текст раздела

6.2 Подраздел

Текст раздела

6.3 Подраздел

Текст раздела

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ КОМПЛЕКСА СКАНИРОВАНИЯ ПРОСТРАНСТВА И НАВИГАЦИИ НА ОСНОВЕ РОБОТИЗИРОВАННЫХ ПЛАТФОРМ И ЛИДАРОВ

7.1 Характеристика программного средства, разрабатываемого для реализации на рынке

Разработанный дипломный проект представляет собой скрипт, загружаемый на робота-паука под управлением одноплатного компьютера Raspberry Pi. Данный скрипт получает сигналы через локальную компьютерную сеть и на их основе осуществляет непосредственное управление конечностями роботизированной платформы, а также собирает данные с лидара, установленного на робота, и отправляет на сервер. Еще одной частью разработанного дипломного проекта является приложение на Windows, предоставляющее удобный пользовательский интерфейс для управления роботизированной платформы через сигналы, принимаемые и обрабатываемые скриптом, и выступающее в роли сервера, интерпретирующего данные лидара для построения на их основе карты и локализации робота-паука.

Программное обеспечение для дистанционного управления роботом-пауком с лидаром и функциональностью, предоставляемой методом SLAM, имеет широкий спектр применений и может быть полезно в различных областях. Вот некоторые из них:

1. Исследование и картография неизвестных или опасных территорий. Робот-паук с лидаром и возможностью построения карты в реальном времени может использоваться для исследования территорий, до которых человеку трудно или опасно добраться. Например, в случае поиска и проведения спасательных работ при стихийных бедствиях, исследовании неизвестных пещер или других опасных зон.

2. Промышленность и инспекция. Робот-паук с лидаром может выполнять задачи инспекции и мониторинга в сложных промышленных средах, где доступ человеку ограничен или рискован. Например, он может использоваться для проверки состояния инфраструктуры, такой как трубопроводы, нефтяные платформы, электростанции, или для инспекции в местах проведения строительных работ.

3. Робототехнические исследования. Проект может быть полезным для исследований и разработок в области робототехники. Робот-паук с лидаром и SLAM-картированием может служить платформой для проведения экспериментов и разработки новых алгоритмов в области автономной навигации и мобильной робототехники.

4. Образование и наука. Проект может быть использован в учебных заведениях для обучения студентов основам робототехники, компьютерного зрения и автономной навигации. Робот-паук с лидаром и SLAM-

картированием предоставляет практическую платформу для изучения данных технологий.

Разработанное программное обеспечение в отличие от аналогов, таких как Gmapping, является не инструментарием, а готовым решением для работы с конкретной роботизированной платформой. Кроме оптимизированной и точной обработки данных лидара по методу SLAM, приложение реализует эффективный алгоритм движения для робота-паука.

7.2 Расчёт инвестиций в разработку программного средства

7.2.1 Расчёт зарплат на основную заработную плату разработчиков

Расчёт затрат на основную заработную плату разработчиков производится исходя из количества людей, которые занимаются разработкой программного продукта, месячной зарплаты каждого участника процесса разработки и сложности выполняемой ими работы. Затраты на основную заработную плату рассчитаны по формуле:

$$З_о = К_{пр} \sum_{i=1}^{n} З_{чi} \cdot t_i, \quad (7.1)$$

где $К_{пр}$ – коэффициент премий и иных стимулирующих выплат;

n – категории исполнителей, разрабатывающих программного средства;

$З_{чi}$ – часовая заработная плата исполнителя i-й категории, р.;

t_i – трудоемкость работ, выполняемых исполнителем i-й категории, ч.

Разработкой всего приложения занимается инженер-программист, Обязанности тестирования приложения лежат на инженере-тестировщике. Задачами инженера-программиста, являются разработка приложения, реализующего обработку данных лидара и алгоритма движения, и скрипта для роботизированной платформы. Инженер-тестировщик занимается выявлением неработоспособных частей приложения и скрипта для робота паука, а также оценивает пользовательский опыт, получаемый от использования приложения.

Месячная заработная плата основана на медианных показателях для Junior инженера-программиста за 2024 год по Республике Беларусь, которая составляет 925 Долларов США в месяц, а для Junior инженера-тестировщика – 700 Долларов США [6]. По состоянию на 15 апреля 2024 года, 1 Доллар США по курсу Национального Банка Республики Беларусь составляет 3,2640 белорусских рублей [7].

В перерасчёте на Белорусские рубли месячные оклады для инженера-программиста и инженера-тестировщика соответственно составляют составляет 3 019,2 и 2 284,8 белорусских рублей соответственно.

Часовой оклад исполнителей высчитывается путём деления их

месячного оклада на количество рабочих часов в месяце, то есть 160 часов.

Коэффициент премии приравнивается к единице, так как она входит в сумму заработной платы. Затраты на основную заработную плату приведены в таблице:

Таблица 7.1 – Затраты на основную заработную плату

Категория исполнителя	Месячный оклад, р	Часовой оклад, р	Трудоёмкость работ, ч	Итого, р
Инженер-программист	3 019,2	18,87	208	3 924,96
Инженер-тестировщик	2 284,8	14,28	24	342,72
Итого				4 267,68
Премия и иные стимулирующие выплаты (0%)				0
Всего затраты на основную заработную плату разработчиков				4 267,68

7.2.2 Расчет затрат на дополнительную заработную плату разработчиков

Расчёт затрат на дополнительную заработную плату команды разработчиков рассчитывается по формуле:

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

где $Н_д$ – норматив дополнительной заработной платы.

Значение норматива дополнительной заработной платы принимает за 10 %.

7.2.3 Расчет отчислений на социальные нужды

Размер отчислений на социальные нужды определяется согласно ставке отчислений, которая на апрель 2024 г. равняется 35%. Из этой общей ставки 29% отчисляется на пенсионное страхование, а 6% – на социальное страхование. Расчёт отчислений на социальные нужды вычисляется по формуле:

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100}, \quad (7.3)$$

где $Н_{соц}$ – норматив отчислений в ФСЗН.

7.2.4 Расчет прочих расходов

Расчёт затрат на прочие расходы определяется при помощи норматива

прочих расчётов. Эта величина имеет значение 30%. Расчёт прочих расходов вычисляется по формуле:

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

где $Н_{пр}$ – норматив прочих расходов.

7.2.5 Расчет расходов на реализацию

Для того, чтобы рассчитать расходы на реализацию, необходимо знать норматив расходов на неё. Принимаем значение норматива равным 3%. Формула, которая использована для расчёта расходов на реализацию:

$$Р_р = \frac{З_о \cdot Н_р}{100}, \quad (7.5)$$

где $Н_р$ – норматив расходов на реализацию.

7.2.6 Расчет общей суммы затрат на разработку и реализацию

Определяем общую сумму затрат на разработку и реализацию как сумму ранее вычисленных расходов: на основную заработную плату разработчиков, дополнительную заработную плату разработчиков, отчислений на социальные нужды, расходы на реализацию и прочие расходы. Значение определяется по формуле:

$$З_р = З_о + З_д + Р_{соц} + Р_{пр} + Р_р \quad (7.6)$$

Таким образом, величина затрат на разработку программного средства высчитывается по указанной выше формуле и указана в таблице:

Таблица 7.2 – Затраты на разработку

Название статьи затрат	Формула/таблица для расчёта	Значение, р.
1	2	3
1. Основная заработная плата разработчиков	См. таблицу 7.1	4 267,68
2. Дополнительная заработная плата разработчиков	$З_д = \frac{4\,267{,}68 \cdot 10}{100}$	426,77
3. Отчисление на социальные нужды	$Р_{соц} = \frac{(4\,267{,}68 + 426{,}77) \cdot 35}{100}$	1 643,06

Продолжение таблицы 7.2

1	2	3
4. Прочие расходы	$\text{Р}_{\text{пр}} = \frac{4\,267{,}68 \cdot 30}{100}$	1 280,31
5. Расходы на реализацию	$\text{Р}_{\text{р}} = \frac{4\,267{,}68 \cdot 3}{100}$	128,03
6. Общая сумма затрат на разработку и реализацию	$\text{З}_{\text{р}} = 4\,267{,}68 + 426{,}77 + +1\,643{,}06 + 1\,280{,}31 + 128{,}03$	7 745,85

7.3 Расчет экономического эффекта от реализации программного средства на рынке

Для расчёта экономического эффекта организации-разработчика программного средства, а именно чистой прибыли, необходимо знать такие параметры как объем продаж, цену реализации и затраты на разработку. В качестве количества проданных копий программного обеспечения за год будет взято 50. Стоимость одной копии составляет 400 рублей.

Для расчёта прироста чистой прибыли необходимо учесть налог на добавленную стоимость, который высчитывается по следующей формуле:

$$\text{НДС} = \frac{\text{Ц}_{\text{отп}} \cdot N \cdot \text{Н}_{\text{д.с}}}{100\% + \text{Н}_{\text{д.с}}}, \tag{7.7}$$

где N – количество копий(лицензий) программного продукта, реализуемое за год, шт.;

$\text{Ц}_{\text{отп}}$ – отпускная цена копии программного средства, р.;

N – количество приобретённых лицензий;

$\text{Н}_{\text{д.с}}$ – ставка налога на добавленную стоимость, %.

Ставка налога на добавленную стоимость по состоянию на 15 апреля 2024 года, в соответствии с действующим законодательством Республики Беларусь, составляет 20%. Используя данное значение, посчитаем НДС:

$$\text{НДС} = \frac{400 \cdot 50 \cdot 20\%}{100\% + 20\%} = 3\,333{,}33 \text{ р.}$$

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли от продажи программного продукта. Для этого используется формула:

$$\Delta\text{П}_{\text{ч}}^{\text{р}} = (\text{Ц}_{\text{отп}} \cdot N - \text{НДС}) \cdot \text{Р}_{\text{пр}} \cdot \left(1 - \frac{\text{Н}_{\text{п}}}{100}\right), \tag{7.8}$$

где *N* – количество копий программного продукта, реализуемое за год, шт.;

$Ц_{отп}$ – отпускная цена копии программного средства, р.;

НДС – сумма налога на добавленную стоимость, р.;

$Н_{п}$ – ставка налога на прибыль, %;

$Р_{пр}$ – рентабельность продаж копий;

$Р_{пр}$ – рентабельность продаж копий.

Ставка налога на прибыль, согласно действующему законодательству, по состоянию на 14.04.2024 равна 20%. Рентабельность продаж копий взята в размере 40%. Зная ставку налога и рентабельность продаж копий (лицензий), рассчитывается прирост чистой прибыли:

$$\Delta П_{ч}^{р} = (400 \cdot 50 - 3\ 333{,}33) \cdot 40\% \cdot \left(1 - \frac{20}{100}\right) = 5\ 333{,}33$$

7.4 Расчет показателей экономической эффективности разработки для реализации программного средства на рынке

Оценим экономическую эффективность разработки и реализации программного средства на рынке, рассчитав простую норму прибыли по формуле:

$$Р_{и} = \frac{\Delta П_{ч}^{р}}{З_{р}} \cdot 100\%, \quad (7.9)$$

где $\Delta П_{ч}^{р}$ – прирост чистой прибыли, полученной от реализации программного средства на рынке, р.;

$З_{р}$ – затраты на разработку программного средства, р.

Исходя из того, что прирост чистой прибыли равен 5 333,33 рублей и затраты на разработку равны 7 745,85 рублей, получим:

$$Р_{и} = \frac{5\ 333{,}33}{7\ 745{,}85} \cdot 100\% = 68{,}9\%$$

Поскольку рентабельность инвестиций меньше 100%, затраты на разработку и реализацию программного средства на рынке окупятся не ранее, чем через год. В этом случае оценку экономической эффективности необходимо проводить на основе расчета и оценки показателей эффективности для расчетного периода продолжительностью 3 года.

Приведение доходов и затрат к настоящему моменту времени осуществляется путем их умножения на коэффициент дисконтирования, который определяется по следующей формуле:

$$\alpha = \frac{1}{(1 + Е_{н})^{t - t_{р}}}, \quad (7.10)$$

где t – порядковый номер года, доходы и затраты которого приводятся к расчетному периоду;

$Е_н$ – требуемая норма дисконта, которая по своей природе соответствует норме прибыли, устанавливаемой инвестором в качестве критерия рентабельности инвестиций, доли единицы;

t_p – расчетный год, к которому приводятся доходы и инвестиционные затраты ($t_p = 1$).

Требуемая норма дисконта может быть:

1. Не ниже ставки рефинансирования Национального банка Республики Беларусь на момент проведения расчетов.

2. На уровне или выше ставки по долгосрочным банковским депозитам, если проект финансируется за счет собственных средств.

3. На уровне банковской процентной ставки по кредитам, если проект финансируется за счет заемных средств.

Поскольку проект финансируется за счет собственных средств, установим норму дисконта равной 9,5% годовых.

Результаты расчета коэффициентов дисконтирования по годам расчетного периода по формуле 7.10 представлены в таблице 7.3.

Расчет чистого дисконтирования дохода за расчетный период вычисляется по формуле:

$$ЧДД(NVP) = \sum_{t=1}^{n} Р_t \cdot \alpha_t - \sum_{t=1}^{n} З_t \cdot \alpha_t, \qquad (7.11)$$

где α_t – коэффициент дисконтирования, рассчитанный для года t.

Результаты расчета чистого дисконтирования дохода нарастающим итогом представлены в таблице 7.3.

Расчет дисконтированного срока окупаемости инвестиций (DPP) осуществляется по формуле:

$$\sum_{t=1}^{DPP} Р_t \cdot \frac{1}{(1 + Е_н)^{t-t_p}} > \sum_{t=1}^{DPP} З_t \cdot \frac{1}{(1 + Е_н)^{t-t_p}}. \qquad (7.12)$$

Решая неравенство, получаем, что дисконтированный срок окупаемости равен 2 годам.

Расчет рентабельности инвестиций без учета фактора времени определяется по формуле:

$$Р_и = \frac{П_{чср}}{\sum_{t=1}^{n} З_t} \cdot 100\%, \qquad (7.13)$$

где $П_{чср}$ – среднегодовая чистая прибыль, р.

Таблица 7.3 – Расчет эффективности инвестиций

Показатель	Значение по годам расчетного периода		
	1	2	3
Результат			
1. Прирост чистой прибыли от реализации, р.	5 333,33	5 333,33	5 333,33
2. Дисконтированный результат, р.		2 456,81	6 904,81
Затраты			
3. Инвестиции в разработку программного средства, р.	7 745,85	0	0
4. Дисконтированные инвестиции, р.	7 745,85	7 745,85	7 745,85
5. Чистый дисконтированный доход нарастающим итогом, р.	-2 412,52	4 869,33	4 447,99
6. Чистый дисконтированный доход нарастающим итогом, р.	-2 412,52	2 456,81	6 904,81
Коэффициент дисконтирования, доли единицы	1	0,913	0,834

Расчет рентабельности инвестиций:

$$\mathrm{Р_и} = \frac{5\ 333{,}33}{7\ 745{,}85 + 0 + 0} \cdot 100\% = 68{,}9\%.$$

Расчет срока окупаемости без учета фактора времени производится по формуле:

$$\mathrm{Т_{ок}} = \frac{\sum_{t=1}^{\mathrm{n}} З_t}{\mathrm{Р_{ср}}}, \qquad (7.14)$$

где n – расчетный период, количество лет;

$З_t$ – затраты в году t, р.;

$\mathrm{Р_{ср}}$ – среднегодовая сумма экономического эффекта, р.

Расчет срока окупаемости:

$$\mathrm{Т_{ок}} = \frac{7\ 745{,}85 + 0 + 0}{5\ 333{,}33} = 1{,}45 \text{ года}.$$

7.5 Вывод об экономической эффективности

Проведённые расчёты технико-экономического обоснования позволяют сделать предварительный вывод о целесообразности разработки данного программного продукта. Общая сумма затрат на разработку и реализацию

составила 7 745,85 Белорусских рублей, а отпускная цена была установлена на уровне 400 Белорусских рублей.

Чистая прибыль в год от реализации на программного средства на рынке равна 5 333,33 рублей, а срок окупаемости составляет 1,45 года. Из этого можно сделать вывод, что проект имеет потенциал для возврата инвестиций в короткий период времени. Однако при фактическом выходе на рынок надо учитывать множество внешних факторов. Вот некоторые из них:

1. Конкуренция. Уровень конкуренции на рынке может существенно влиять на экономическую эффективность проекта. Если рынок насыщен, может быть сложно проникнуть на него и достичь высокой прибыли. Также важно отслеживать, не отстает ли программное обеспечение от конкурентов в технологическом плане. В наше время технологии развиваются очень быстро, и данная проблема является актуальной, как никогда прежде. Необходимо провести анализ конкурентной среды и определить уникальные преимущества разработанного программного средства.

2. Размер целевой аудитории. Важно оценить размер и потенциал целевой аудитории. Чем больше спрос, тем выше вероятность достижения экономической эффективности. Фактический спрос может отличаться от используемого при расчетах.

3. Маркетинговая стратегия. Правильная маркетинговая стратегия играет важную роль в достижении экономической эффективности. Необходимо разработать эффективный план маркетинга, включающий определение целевой аудитории, выбор каналов продвижения, позиционирование продукта и установление конкурентоспособной цены. Есть вероятность, что при выбранной цене, разработанное программное средство не выдержит конкуренции и ее придется снижать.

4. Обратная связь от клиентов. Важно активно собирать информацию от клиентов, чтобы понимать их потребности, выявлять проблемы и внедрять улучшения. Регулярное взаимодействие с пользователями поможет выстроить долгосрочные отношения, повысить удовлетворенность клиентов и улучшить продукт. Разработанное программное средство служит для выполнения прикладных задач, таким образом при тестировании сложно смоделировать все условия, в которых может работать роботизированная платформа. Например, если реализованный алгоритм движение не будет эффективен в условиях эксплуатации робота-паука, придется внедрять изменения в программное обеспечение. Для решения подобных проблем нужна обратная связь с клиентами.

Все эти факторы необходимо тщательно исследовать и анализировать для окончательного решения о выходе на рынок. Комбинированный подход, учитывающий конкурентные преимущества, спрос, затраты и маркетинг, поможет более успешно продвинуть проект на рынок.

ЗАКЛЮЧЕНИЕ

Текст раздела

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

[1] Gmapping [Электронный ресурс] – Режим доступа: http://wiki.ros.org/gmapping/. – Дата доступа: 09.04.2023.

[2] Google Cartographer ROS [Электронный ресурс] – Режим доступа: https://google-cartographer-ros.readthedocs.io/en/latest/. – Дата доступа: 09.04.2023.

[3] RTAB-Map, Real-Time Appearance-Based Mapping [Электронный ресурс] – Режим доступа: https://introlab.github.io/rtabmap/. – Дата доступа: 09.04.2023.

[4] Autonomous Navigation with Simultaneous Localization and Mapping in/outdoor / E. Pedrosa [и др.]. – Aveiro, 2020.

[5] The Cargo Book [Электронный ресурс] – Режим доступа: https://doc.rust-lang.org/cargo/. – Дата доступа: 10.04.2023.

[6] Интернет-издание «Dev.by» [Электронный ресурс]. – Зарплата в ИТ – Режим доступа: https://salaries.devby.io – Дата доступа: 15.04.2024.

[7] Национальный банк Республики Беларусь [Электронный ресурс]. – Официальные курсы белорусского рубля по отношению к иностранным валютам, устанавливаемые Национальным банком Республики Беларусь ежедневно, на 15.04.2023 – Режим доступа: https://www.nbrb.by/statistics/rates/ratesdaily.asp – Дата доступа: 15.04.2024.

ПРИЛОЖЕНИЕ А
(*обязательное*)
Название приложения

ПРИЛОЖЕНИЕ Б
(*обязательное*)
Название приложения

ПРИЛОЖЕНИЕ В
(*обязательное*)
Спецификация

ПРИЛОЖЕНИЕ Г
(*обязательное*)
Ведомость документов