

# Выпусная работа бакалавра (версия без ГОСТа)

**Выполнил:** Титов Алексей Константинович

**Группа:** ИВТ – 460

**Тема:** Система автономной навигации антропоморфного робота AR600E.

Подсистема планирования траектории движения антропоморфного робота AR600E.

**Факультет:** Электроники и Вычислительной техники

**Кафедра:** Электронно – вычислительные машины и системы

**Год:** 2017

## Аннотация

Данная работа посвящена исследованию и разработке системы автономной навигации для антропоморфного робота AR600E, а также модуля прокладки маршрута движения робота в частности на основе данных, получаемых с камеры глубины Kinect. В работе подробно рассмотрены основные подходы и библиотеки, применяемые при решении вышеупомянутых задач.

## Summary

This scientific work is devoted to research and developing of the autonomous navigation system for anthropomorphic robot AR600E and especially for subsystem of a route of movement, which are based on data obtained with RGB-D sensor named Kinect. In this work, the main approaches and libraries used in solving the above-mentioned problems are discussed in detail.

## Содержание

Введение.....	5
1. Обзор аналогов и анализ предметной области задачи.....	7
Решение от MIT DARPA Robotics Challenge Team.....	7
Планирование движений .....	7
Планирование маршрута.....	8
Решение от Humanoid Robots Lab .....	11
Решение для робота H7 .....	13
Решение от TEAM VIGIR .....	15
Решение для робота Walk-Man.....	17
2. Описание робота AR600, для которого реализуется система автономной навигации.....	18
Описание антропоморфного робота AR600E.....	18
Технические характеристики .....	19
Схема управления AR600.....	19
Описание комплекса управления антропоморфным роботом на основе ФРУНД.....	21
ФРУНД .....	21
Система управления роботом .....	22
3. Постановка задачи создания системы автономной навигации AR600E .....	24
Основные цели создания и требования, выдвигаемые при разработке системы технического зрения .....	24
Функционал системы автономной навигации.....	24
Функционал системы планирования маршрута робота .....	24
4. Определение путей и методов реализации системы автономной навигации для AR600E .....	25
Планирование системы автономной навигации.....	25
Выбор платформы для реализации.....	25
Выбор SLAM алгоритма и источников данных для него.....	29
Разработка архитектуры системы автономной навигации .....	36
Пути реализации подсистемы планирования маршрута движения AR600E .....	40
Планирование по карте препятствий (Occupancy Grid) или карте высот .....	40
Планирование по облакам точек (Point Cloud) и плотным облакам точек (OcTree).....	44
5. Разработка системы автономной навигации для AR600E .....	49
Принципиальная архитектура системы .....	49
Вариант 1. Шаги генерирует ФРУНД.....	49

Вариант 2. Шаги генерирует ФРУНД .....	50
Разработка системы взаимодействия модулей компьютерного зрения и ФРУНДа .....	51
Согласование систем координат ФРУНДа и карты от rtabmap .....	53
Системы координат в ROS .....	53
Системы координат в нашем случае .....	55
6. Разработка подсистемы планирования траектории движения антропоморфного робота AR600E .....	57
Разработка модуля передачи данных для планировщиков .....	57
Общие входные топики .....	57
Общие выходные топики.....	59
Передача данных планировщикам от SLAM алгоритма .....	60
Разработка планировщика маршрута на основе пакета footprint_planner .....	62
Разработка планировщика траектории.....	65
Описание идеи .....	65
Описание архитектуры .....	68
Описание алгоритмов .....	70
7. Анализ результатов и экспериментов .....	74
Качественная оценка точности SLAM алгоритма RTABMAP .....	74
Эксперименты с footprint_planner.....	75
Эксперименты с trajectory_planner.....	76
Дополнительные работы и эксперименты .....	78
Заключение.....	79
Приложение А. Характеристики Kinect ( <a href="https://msdn.microsoft.com/en-us/library/jj131033.aspx">https://msdn.microsoft.com/en-us/library/jj131033.aspx</a> ) ....	80
Приложение Б. Скрипты преобразования систем координат .....	81
Способ через добавление смещения между base_link и camera_link.....	81
Способ через задание initialpose .....	83
Приложение В. Код модуля передачи данных планировщикам.....	86
Приложение Г. Диаграмма классов модуля планирования траектории. ....	88
Приложение Д. Листинг программы планирования траектории .....	89

## Введение

Использование информации об окружении дает возможность учитывать окружение при планировании движений, перемещений или взаимодействий с окружающей средой. Благодаря этому возможно создание по-настоящему автономных роботов, полностью реализующих потенциал, заложенный в них физически, что делает возможным их применение в промышленности и области услуг. В данной работе рассмотрены все аспекты использования окружения роботом, в частности – построение маршрута движения для робота, что позволит ему перемещаться в сложном окружении без прямого контроля оператора.

Научная новизна данной работы заключается в создании комплексного решения для AR600E с учетом особенностей имеющейся системы управления роботом и системы генерации и расчета движений. Во время выполнения данной работы удалось собрать и проанализировать большое количество разрозненных работ, библиотек и подходов, чего я не видел в данных работах.

Практическая значимость работы, во-первых, заключается в том, что была успешна произведена разработка системы автономной навигации в нескольких вариантах, которая предоставляет ФРУНДу информацию о маршруте движения и его качестве.

Во-вторых, в том, что в процессе сбора информации было найдено много информации о смежных областях, таких как планирование движений с учетом окружения (так, чтобы не допускать коллизий с ним). Это позволит в дальнейшем заняться планированием сложных движений для робота с учетом, как и кинематической модели, так и модели окружения.

Разработчики многих из рассмотренных в работе готовых решений описали свои работы в соответствующих публикациях. И лишь несколько предоставили результаты в виде открытого исходного кода, скомпонованного в библиотеки, сообществу.

Но их использование ограничено по ряду причин, одной из которых является особенность системы управления роботом и системы генерации движений (ее роль выполняет ФРУНД).

В связи с этим целью данной работы являлась разработка системы на основе компьютерного зрения, позволявшая роботу автономно перемещаться в пространстве.

Для достижения поставленной цели решались следующие задачи:

- Анализ существующих решений
- Поиск путей реализации системы автономной навигации
- Поиск путей реализации подсистемы планирования маршрута робота
- Выбор библиотек и реализация вышеописанных систем
- Реализация коммуникационной части с ФРУНДом

В первой главе производится обзор аналогов. Мне удалось найти много примеров законченных решений, многие из которых так или иначе хорошо показали себя, например занимали призовые места в DARPA. Среди них есть одна работа датирующаяся 2003 годом, что меня весьма сильно поразило.

Во второй главе описывается антропоморфный робот AR600E, а также система управления роботом и система генерации движений для него.

В третьей главе осуществляется постановка задачи разработки системы и излагаются базовые требования к ней.

В четвертой главе производится обзор как общей архитектуры разрабатываемой системы, так и в частности обзор библиотек и подходов к задаче планирования маршрута и движений в зависимости от вида, в котором предоставляется карта окружения.

В пятой главе детально рассматриваются два варианта реализации системы автономной навигации в зависимости от того, кто является источником шагов – ФРУНД или планировщик маршрута. Также в этой главе уделено внимание задаче согласования систем координат ФРУНДа и планировщика.

В шестой главе подробно рассматривается разработка, принцип работы и архитектура двух вариантов подсистемы планирования маршрута исходя из формата траектории, выдаваемой каждым из планировщиков в результате планирования.

В седьмой главе кратко описаны результаты экспериментов, проводимых с планировщиками, а также представлены результаты нескольких дополнительных работ, связанных с роботом AR600E.

## 1. Обзор аналогов и анализ предметной области задачи

Далее представлен ряд готовых решений по управлению, навигации и планированию движений с учетом данных об окружении. В каждой из этих работ вкратце описаны принципы решения вышеупомянутых задач. В ряде работ авторы открыли код своего проекта, что в последствии помогло реализовать один из вариантов планирования маршрута. Самая ранняя из упомянутых здесь публикаций датируется 2003 годом. Причем в ней освещены многие моменты, которые попадались так или иначе в других работах, появившихся гораздо позже.

### Решение от MIT DARPA Robotics Challenge Team

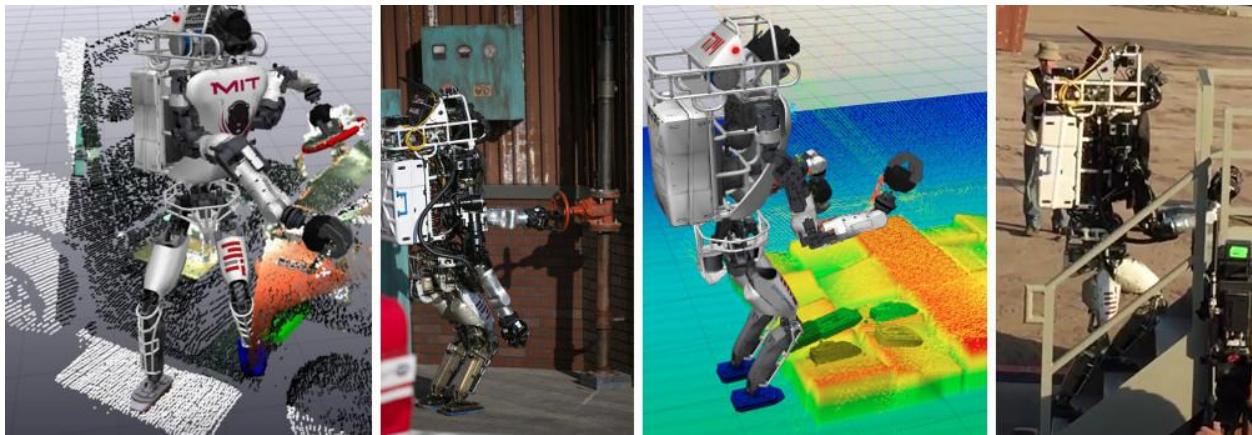


Рисунок 1 - Изображение с главной страницы сайта проекта

<http://drc.mit.edu/index.html>

### Планирование движений

Команда проекта разработала свои средства для планирования, управления и визуализации, работая с роботом Atlas. В их проекте планер совмещает в себе решение задач инверсной кинематики и избегания при планировании коллизий с окружением. Это позволяет роботу планировать и реализовывать сценарии, связанные с окружением, какие возникают при прохождении полосы в DARPA Robotic Challenge. Главный планировщик при помощи планировщика шагов позволяет получать гладкие движения, не задевающие окружения, которые затем рассчитываются движком инверсной кинематики с учетом сохранения равновесия и выполняются на роботе.

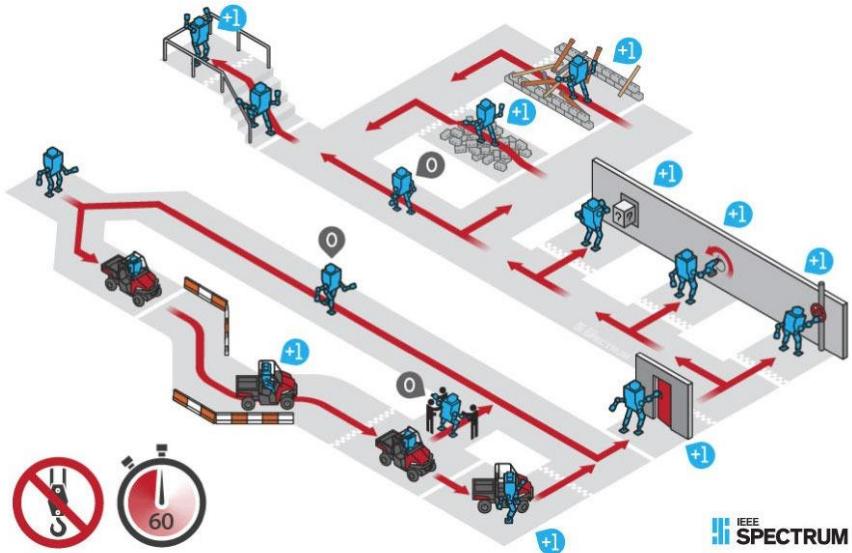


Рисунок 2 - Полоса препятствий, используемая на Darpa Robotic Challenge

### Планирование маршрута

Подробности по этому вопросу я почерпнул из следующих публикаций ([http://drc.mit.edu/docs/2014\\_jfr\\_fallon.pdf](http://drc.mit.edu/docs/2014_jfr_fallon.pdf)) ([http://groups.csail.mit.edu/robotics-center/public\\_papers/Deits14a.pdf](http://groups.csail.mit.edu/robotics-center/public_papers/Deits14a.pdf))

Планирование осуществляется не из набора конечного числа заранее заданных положений ступней друг относительно друга (шагов), а несколько по другому принципу.

Для начала был определен полигон кинематически выполнимых положений ступни при шаге. Зеленым изображена ступня, являющаяся опорной на данном шаге, а красным – возможные положения другой ступни на данном шаге.

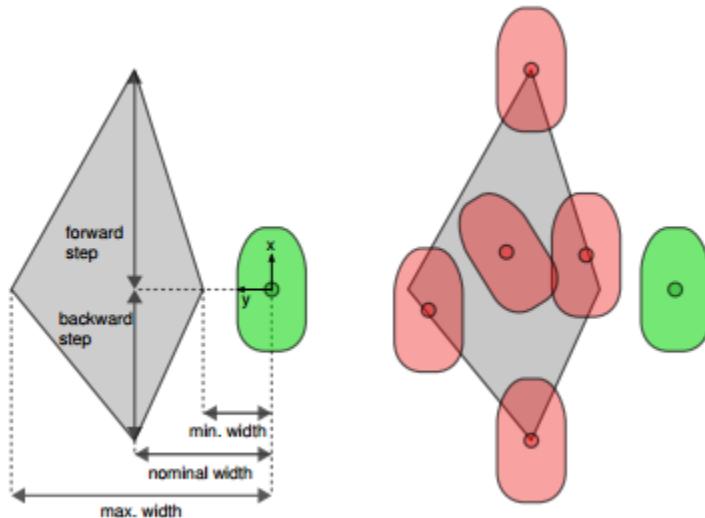


Рисунок 3 - Упрощенное представление кинематически выполнимых шагов

Поиск маршрута старается минимизировать число шагов, которые необходимо выполнить до поставленной цели.

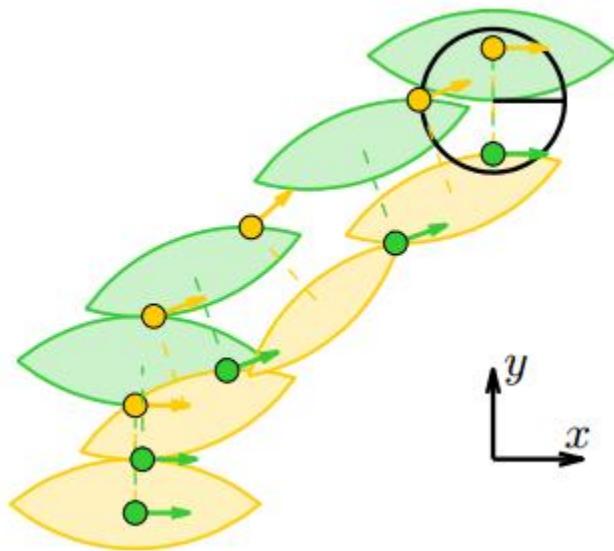


Рисунок 4 - Спланированный маршрут, на котором показаны области достижимости при шаге

Для того, чтобы планировать в 3D, используется алгоритм IRIS, который позволяет выделять плоскости, свободные от препятствий, по которым далее и осуществляется планирование шагов с параллельной проверкой коллизий для всего тела робота. Для того, чтобы задать регион, нужна помочь оператора (задание начальной точки для алгоритма). Таким образом алгоритм справляется с объектами разных высот, которые могут быть как препятствием, так и поверхностью, пригодная для совершения шага в нее.

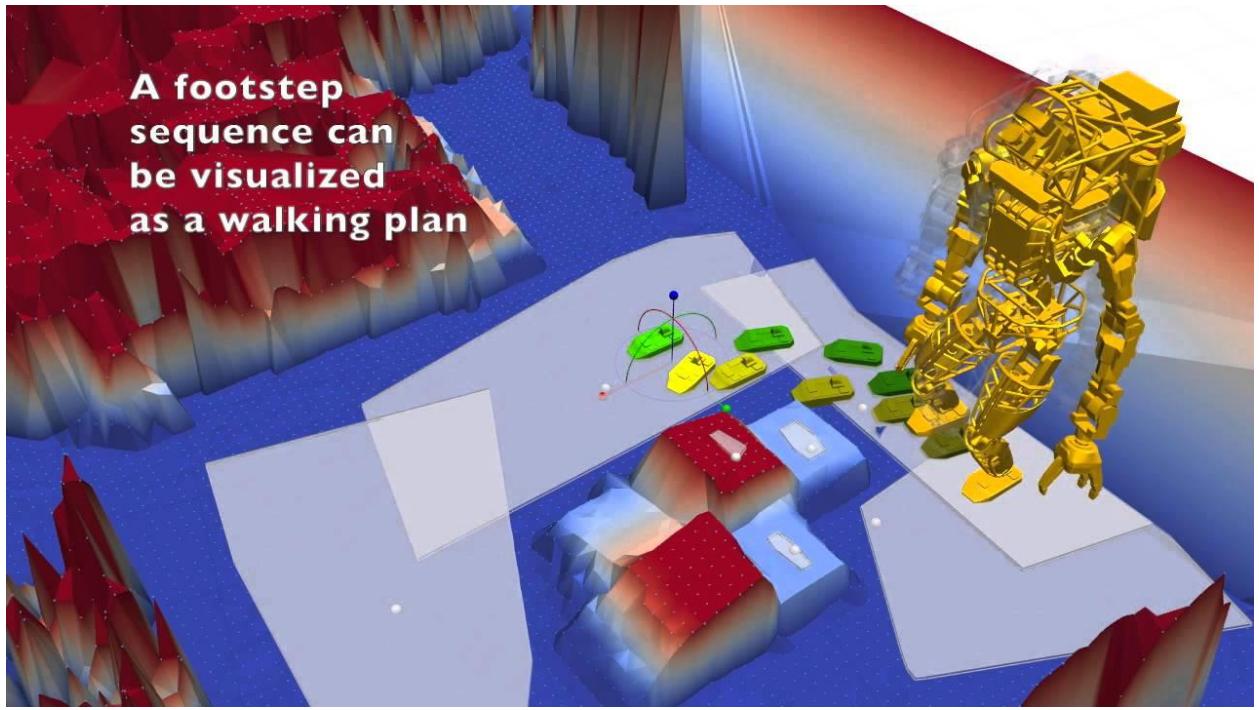


Рисунок 5 - Пример планирования маршрута в 3D с учетом безопасных регионов от алгоритма IRIS

## Решение от Humanoid Robots Lab

Эта лаборатория работает не только над задачами навигации и планирования, а также и над задачами локомоции робота и коррекции движений оператора (<http://hrl.informatik.uni-freiburg.de/> ).

Это комплексное решение для робота Nao по навигации и планированию маршрута.  
(<http://hrl.informatik.uni-freiburg.de/papers/hornung13icraws.pdf>)

Информация об окружении хранится в карте высот, которая строится на основе собственного SLAM алгоритма.

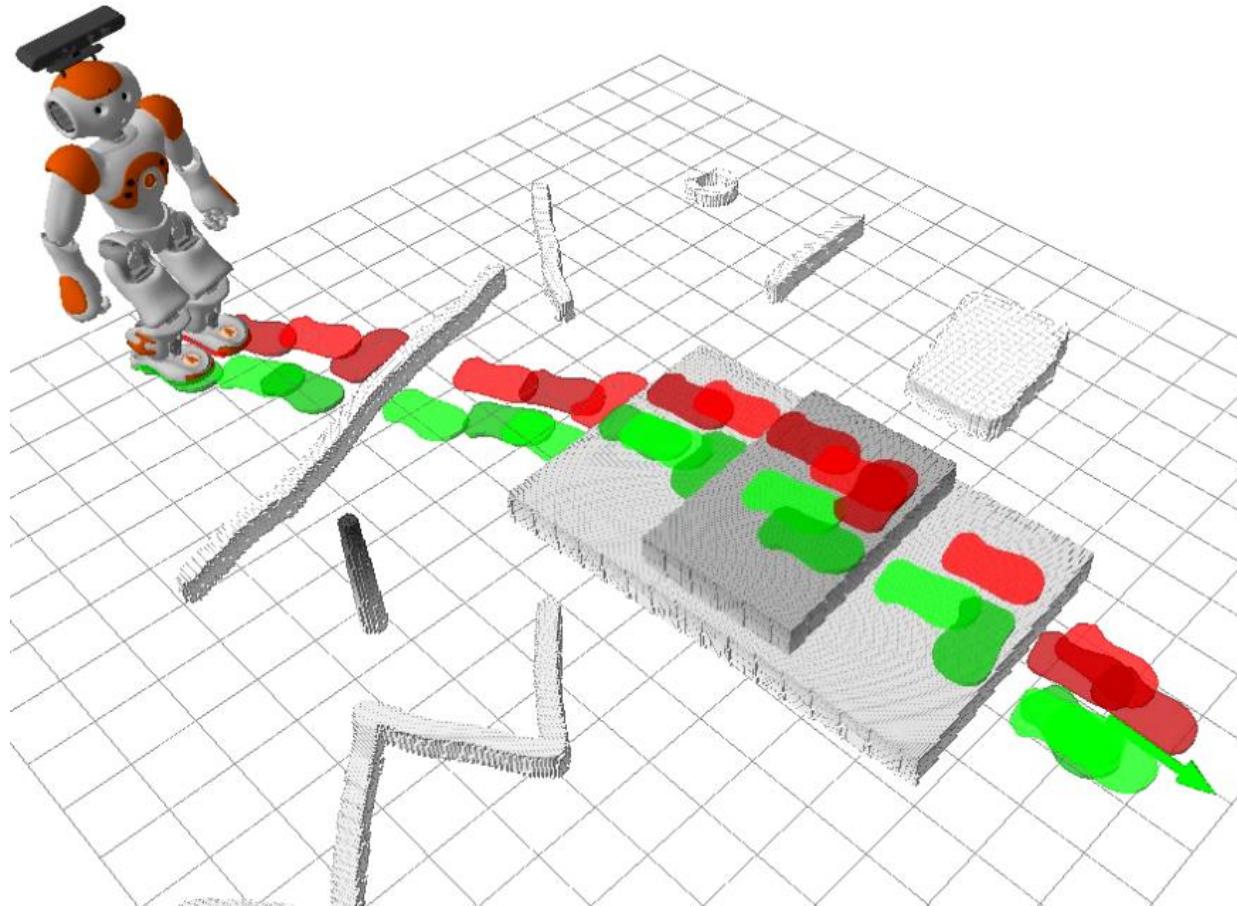


Рисунок 6 - Карта весов, а также построенные на ее основе маршрут

Планирование маршрута робота выполняется из набора заранее заданных параметризованных действий. Каждое положение проверяется на коллизии с окружением при помощи довольно хитрого подхода inverse height map.

Более ранние работы по локализации в заранее заданных окружениях в виде плотных облаков точек (<http://hrl.informatik.uni-freiburg.de/papers/maierd12humanoids.pdf> ) и планированию маршрута на плоской карте препятствий (<http://hrl.informatik.uni-freiburg.de/> ).

[freiburg.de/papers/hornung12humanoids.pdf](http://freiburg.de/papers/hornung12humanoids.pdf) ) были опубликованы в качестве пакетов для платформы ROS. ([http://wiki.ros.org/humanoid\\_navigation](http://wiki.ros.org/humanoid_navigation))

Также исходный код этих пакетов был выложен на GitHub командой проекта.

Отдельного внимания заслуживает пакет `footstep_planner`  
([http://wiki.ros.org/footstep\\_planner?distro=indigo](http://wiki.ros.org/footstep_planner?distro=indigo) )

Как описано в публикации ([http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12\\_Slides\\_Hornung.pdf](http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12_Slides_Hornung.pdf)), данный планер осуществляет планирование на плоской карте препятствий, исходя из списка шагов которые он может совершать. В планере реализовано несколько алгоритмов поиска, а также несколько эвристик для алгоритма A\*. Пакет, основанный на нем, позволяет планировать маршрут в виде набор шагов. Планер умеет переступать препятствия, но т.к. на карте препятствий у нас нет информации о высоте оных, то данная функция на мой взгляд, оказывается слабо применима.

Пакет, по словам авторов ([http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12\\_Slides\\_Hornung.pdf](http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12_Slides_Hornung.pdf)), умеет работать в динамически меняющихся окружениях, эффективно используя уже найденный ко времени изменения окружения маршрут.

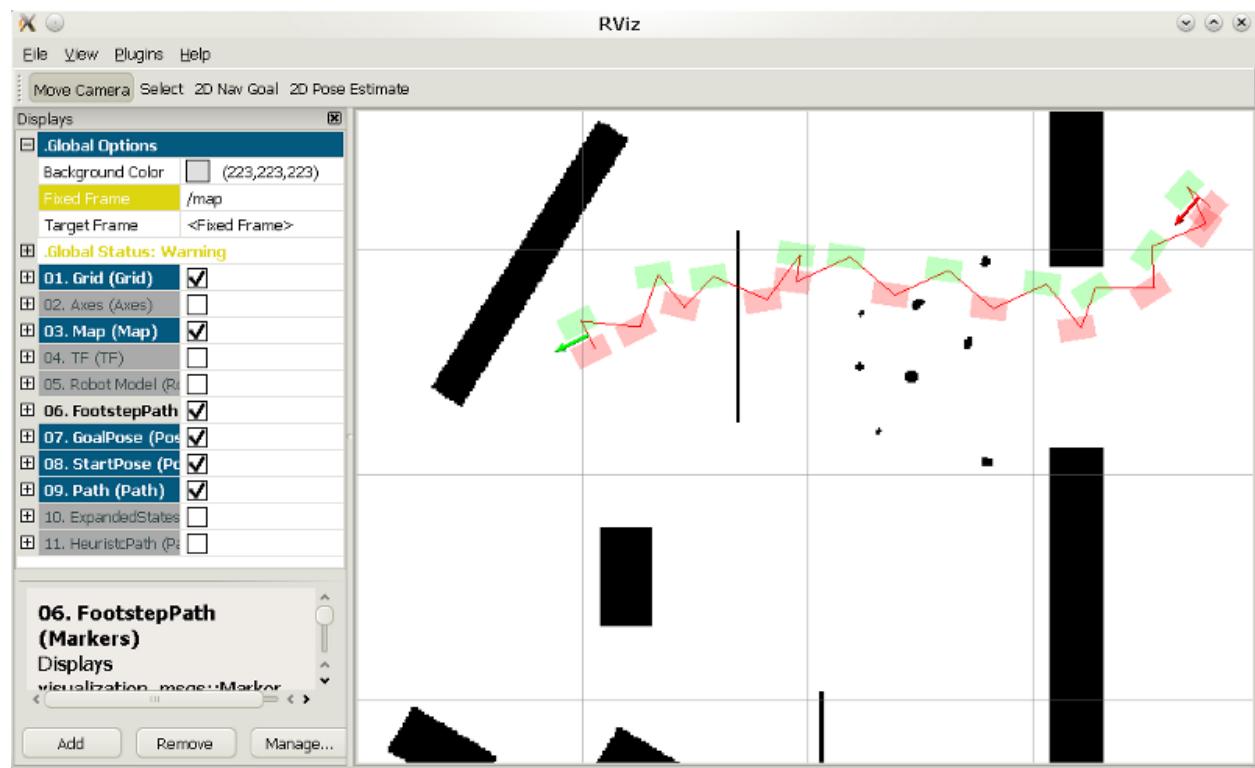


Рисунок 7 - Планирование маршрута в rviz на заранее созданной карте при помощи пакета `footstep_planner`

## Решение для робота Н7

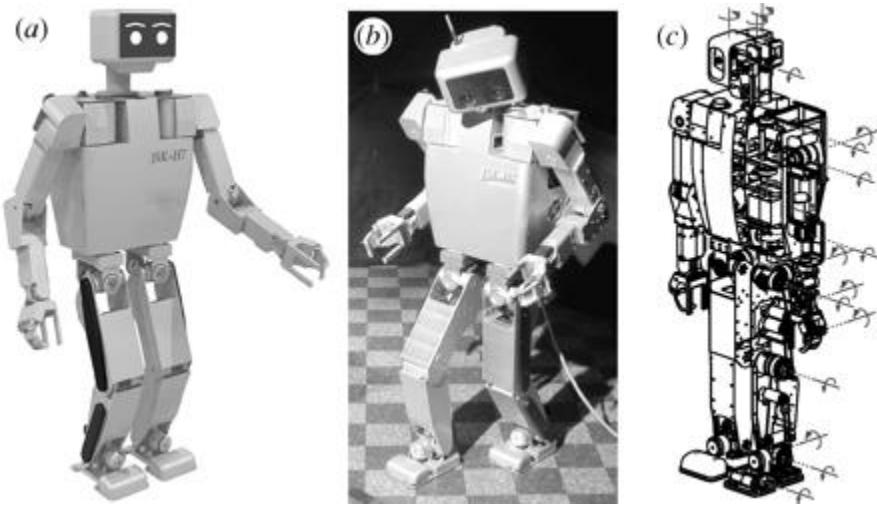


Рисунок 8 - Робот Н7, на основе которого велись исследования

Работа (<http://rsta.royalsocietypublishing.org/content/365/1850/79>), посвященная этому проекту, была опубликована в далеком 2006 году. В ней в общих чертах описывается создание системы автономной навигации для робота Н7.

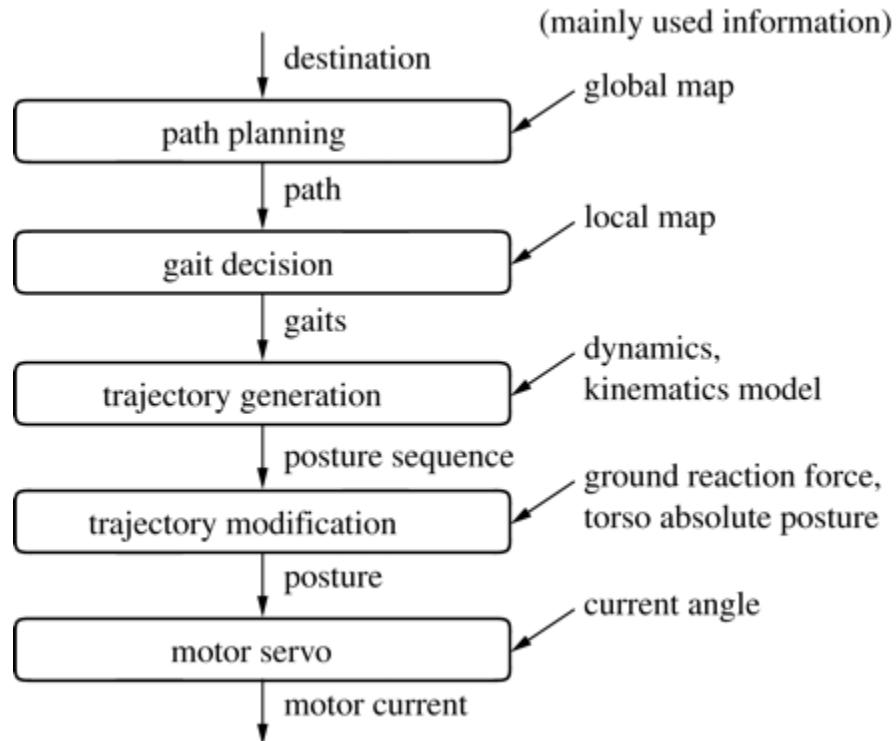


Рисунок 9 - Структура системы управления движением Н7

Довольно интересной мне показалась часть про генерацию движений робота. Генерация движений проводится на основании метода нулевого момента. Планировщику движений

подается на вход желаемая траектория движения точки нулевого момента, из которой он получает траекторию движения робота.

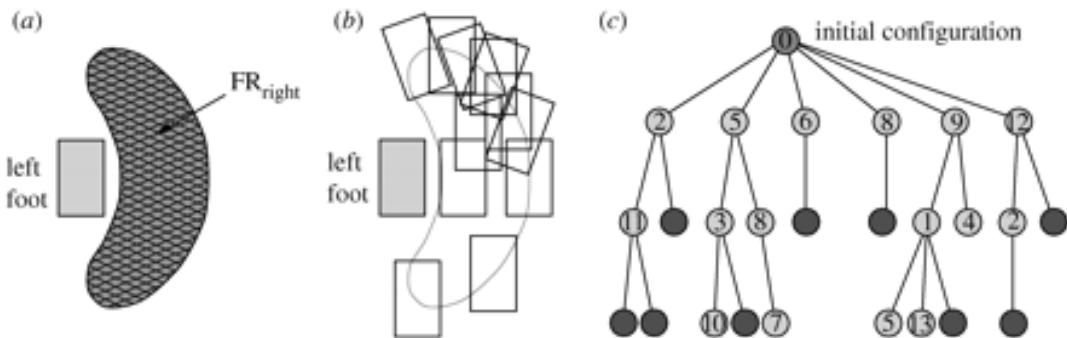


Рисунок 10 - Принцип поиска маршрута для робота

Для начального положения из множества дискретных выполнимых положений ступней, которые соответствуют возможным движениям, строится дерево возможных состояний. Далее в нем ищется оптимальная последовательность шагов, которая приводит как можно ближе к цели.

Эвристики подобраны таким способом, чтобы минимизировать число шагов и их сложность.

Помимо набора возможных шагов, которые может выполнить робот планировщику движений

<http://rsta.royalsocietypublishing.org/content/365/1850/79>

Решение от TEAM VIGIR

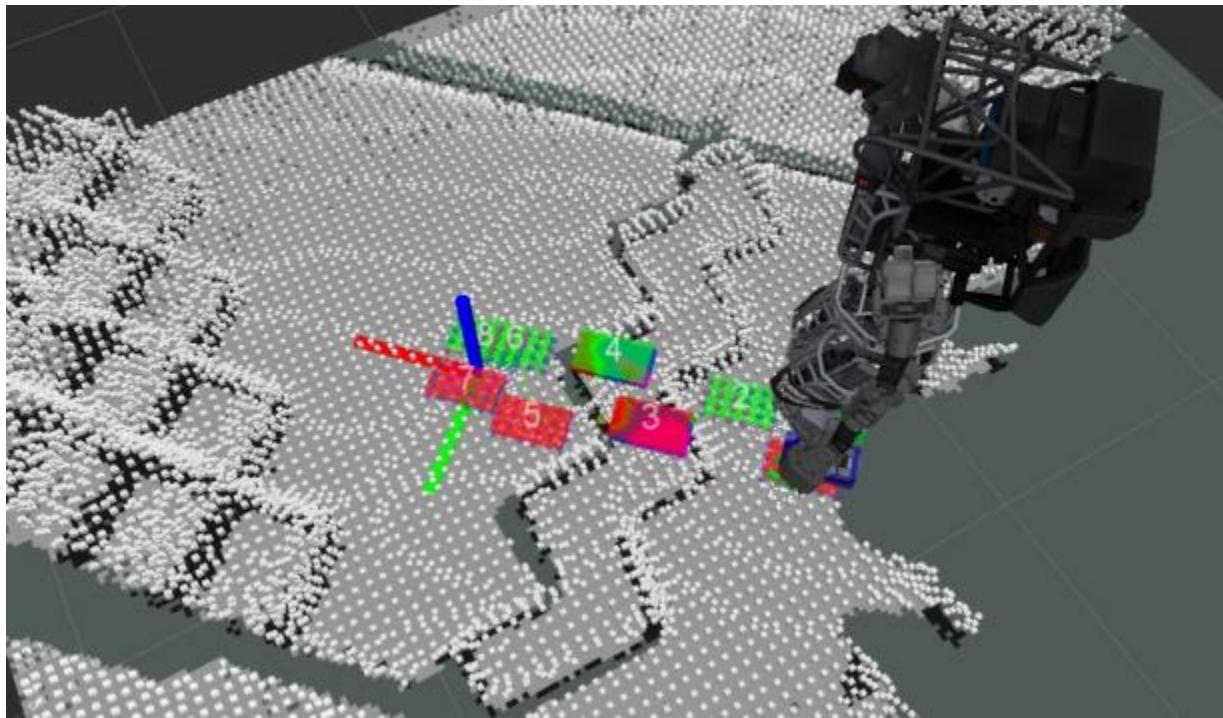


Рисунок 11 - Визуализация прохождения роботом спланированного маршрута

Данная команда создала решение по планированию на основе 3D данных об окружении. Переработав пакет `footstep_planner`, они реализовали `vigir_footstep_planner`, который позволяет планировать шаги на неровной поверхности. Шаги выбираются дискретно из полигона возможных шагов, как у команды MIT. ([http://www.sim.informatik.tu-darmstadt.de/publ/download/2014\\_stumpf\\_footstep\\_planning\\_Humanoids.pdf](http://www.sim.informatik.tu-darmstadt.de/publ/download/2014_stumpf_footstep_planning_Humanoids.pdf))

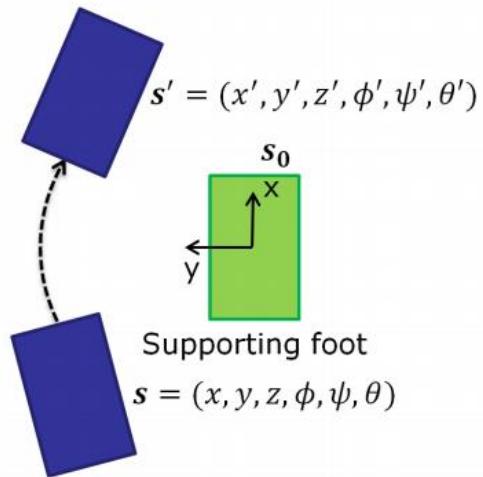


Рисунок 12 - Один из возможных шагов в данной позиции

При в качестве весов используется несколько функций, одна из которых – функция риска, которая улучшает качество получаемой траектории, при этом не исключая наличие небезопасных шагов.

Для управления роботом и планирования его движений используется библиотека Drake (A Planning, Control, and Analysis toolbox for Nonlinear Dynamical Systems), в которую вошли многие наработки команды из MIT, о которой упоминалось выше. (<http://drake.mit.edu/>)

<https://blog.acolyer.org/2015/11/03/human-robot-teaming-for-rescue-missions-team-vigirs-approach-to-the-2013-darpa-robotics-challenge-trials/>

[http://wiki.ros.org/vigir\\_footstep\\_planning](http://wiki.ros.org/vigir_footstep_planning)

## Решение для робота Walk-Man

<http://journal.frontiersin.org/article/10.3389/frobt.2016.00025/full>

Данная работа предстаетает несколько отличается от представленных ранее. По сути автономного здесь ничего и нет – авторы реализовали параметризованный набор действий, который в последствии запускается, управляетя и корректируется оператором из интерфейса GUI. Также у оператора была возможность самостоятельно планировать и выполнять движения.

В проекте была задействована библиотека YARP, которая по смыслу близка к концепции ROS, но является более низкоуровневым проектом, направленным на то, чтобы четко разграничить модели системы, такие как датчики, приводы и наладить управление над ними.

([http://www.yarp.it/what\\_is\\_yarp.html](http://www.yarp.it/what_is_yarp.html))

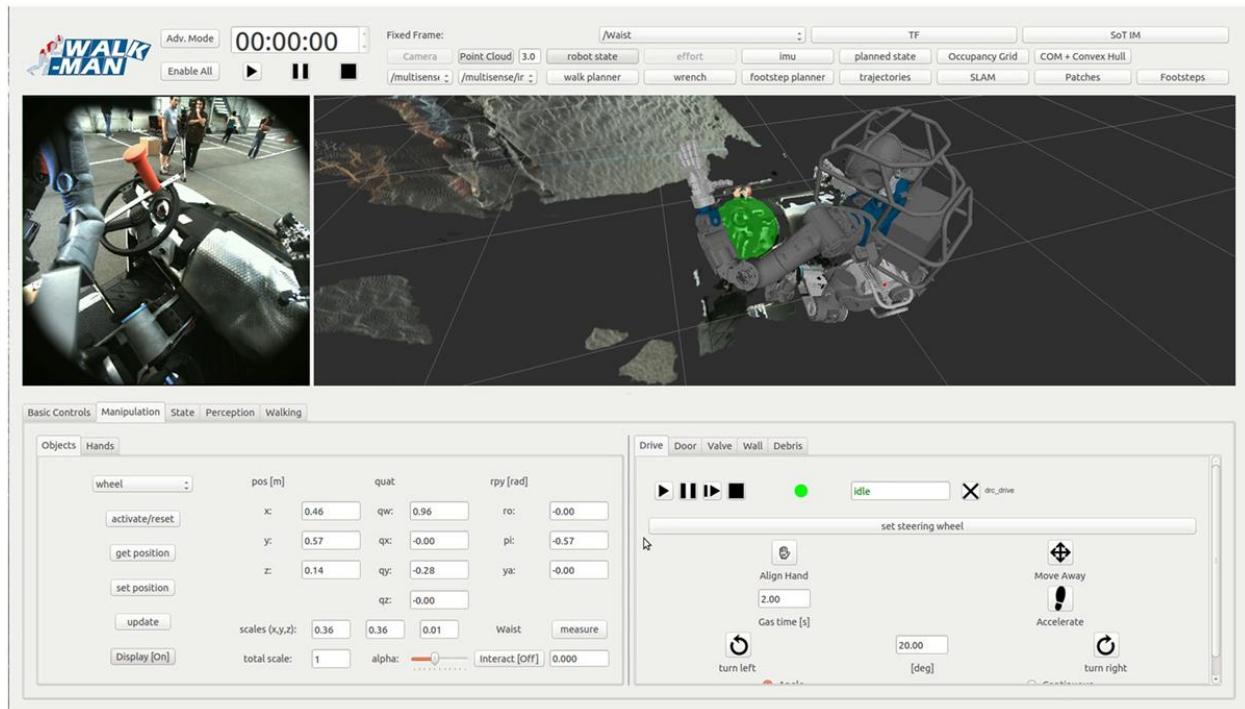


Рисунок 13 - Интерфейс пилота в проекте Walk - Man

## 2. Описание робота AR600, для которого реализуется система автономной навигации

Далее будет представлено краткое описание антропоморфного робота, часть информации о котором была взята из официальной документации, находящейся в закрытом доступе. (<http://npro-at.com/products/ar-600e/>). При рассмотрении был сделан упор на электронное оснащение робота, нежели на механические аспекты, так как именно этот аспект более важен при решении задач планирования и автономной навигации.

### Описание антропоморфного робота AR600E

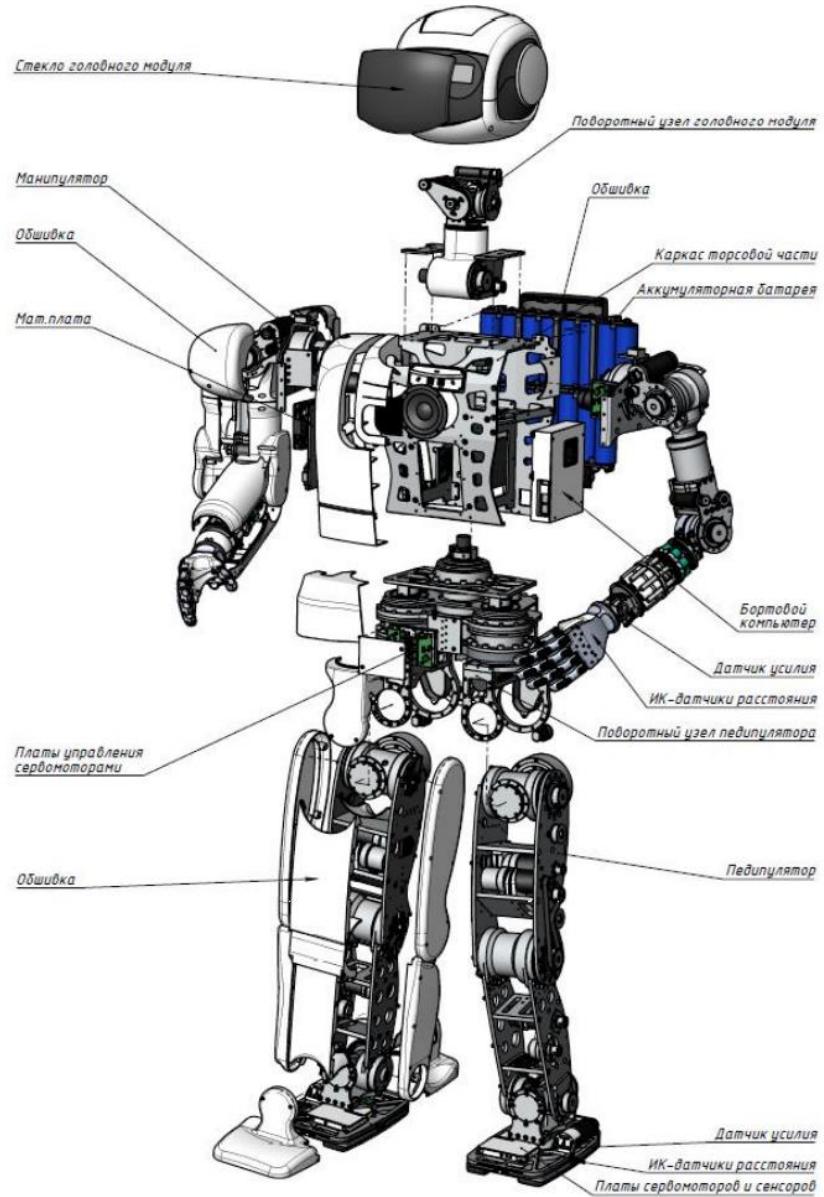


Рисунок 14 - Полная комплектация AR600

## Технические характеристики

### Габариты

- Рост: 1446 мм
- Ширина по плечам: 499 мм
- Ширина в грудной клетке: 259 мм

### Схема управления AR600

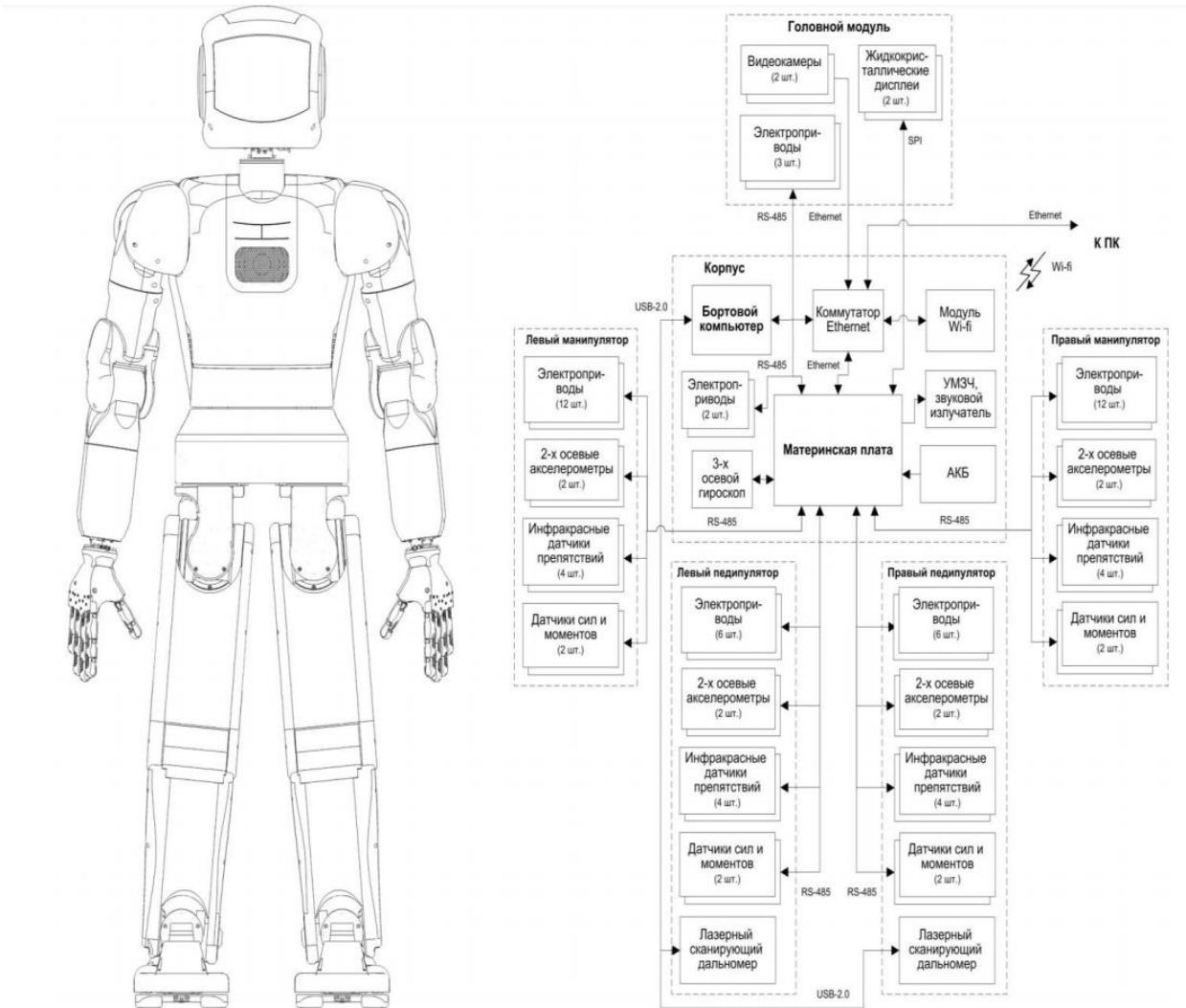


Рисунок 15 - Структурная схема системы управления AR600

### Материнская плата

Основным электронным модулем робота является материнская плата MBX-600.04, которая отвечает за все системы робота. Материнская плата взаимодействует со всеми платами управления, которые непосредственно управляют сервомоторами робота, информацию о требуемом положении того или иного сервомотора, а также принимает от них данные об углах сервомоторов и данные с датчиков (имеются датчики давления в стопах, а также инерциальная навигационная система). Эти данные также отправляются в сообщениях специального формата

через Ethernet соединение к управляющему стационарному ПК, где установлена программа управления роботом (собственная, как в нашем случае или же поставляемая вместе с роботом).

Иными словами, материнская плата осуществляет:

- - Связь системы управления с внешним персональным компьютером по интерфейсам Ethernet и Wi-Fi;
- - Получение пакетов данных из ПК и передача в контроллеры электроприводов и датчиков по интерфейсам RS485 или SPI;
- - Сбор данных с контроллеров электроприводов и датчиков по интерфейсам RS485 и SPI и формирование пакетов данных для передачи в ПК;
- - Преобразование напряжения питания 48 В в 5, 6, 8, 12В;
- - Управление электропитанием оборудования AR-600

#### *Бортовой компьютер AVALUE QM 87*

На борту установлен персональный компьютер AVALUE. Назначение компьютера: хранение и выполнение программ, связанных с автономным режимом робота. Также, персональный компьютер служит для обеспечения работы проектируемых систем технического зрения.

Одноплатный промышленный компьютер со следующими характеристиками:

- CPU - 4th Gen Intel® Core™ i7-4700EQ 4-Core 3.4GHz processor;
- System Chipset – Intel® QM87 Express Chipset;
- BIOS - AMI uEFI BIOS, 128Mbit SPI Flash ROM;
- I/O Chip - EC ITE IT8518E;
- System Memory - 1 x204-Pin DDR3L 1333MHz SO-DIMM up to 8 GB;
- SSD - 1 x mSATA Supported from Mini PCIe;
- SATA - 2 xSATA III;
- Expansion - 1 x PCIe Mini Card Slot (From The Daughter Board) I/O;
- MIO - 2 x Serial ATA Ports,
- 1 x RS-232, 1 x RS-232/ 422/485, LPC, 2 x SATA;
- USB - 2 x USB 2.0, 6 x USB 3.0; DIO 8-bit GPI, 8-bit GPO Display;

Были произведены некоторые исследования относительно производительности данного бортового ПК.

Они показали, что решение уравнения движения робота в системе ФРУНД при помощи бортового ПК приводит к загрузке ЦП на 90%-100%, что делает невозможным совместную с ней работу системы компьютерного зрения, которая является не менее сложной вычислительной задачей.

Несмотря на заявленные довольно впечатительные характеристики система показа себя не лучшим образом во время работы с ней. Как в плане производительности, так и в плане взаимодействия с ней (USB порты изначально были перекрыты корпусом, перестановка ОС возможна только с CD/DVD диска и т.д.).

## Описание комплекса управления антропоморфным роботом на основе ФРУНД

### ФРУНД

ФРУНД (<http://frund.vstu.ru/>) - программная система формирования решений уравнений нелинейной динамики. Предназначена для моделирования динамики систем твёрдых и упругих тел. Основные черты методов представления уравнений движения, применяемых в системе ФРУНД: возможность унификации задания расчетной схемы исследуемого объекта с помощью конечного множества типов составляющих элементов, простота автоматического формирования уравнений движения и программ для их интегрирования, гибкость при добавлении в систему новых видов специальных взаимодействий.

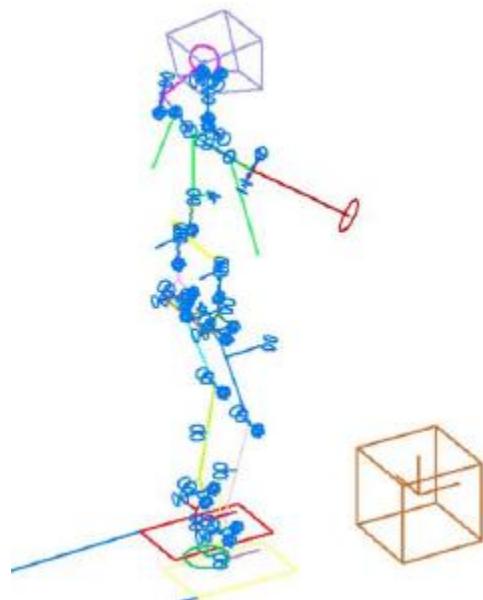


Рисунок 16 - Изображение компьютерной модели робота

Ранее во ФРУНДе была создана подробная компьютерная многотельная модель робота с учетом всех его конечностей, их степеней свободы и параметров движителей.

В дальнейшем внутри ФРУНДА были смоделированы некоторые типы движений: походка, переступание через ступень, поворот и т.д. Для стабилизации движения робота использовались уравнения связей, обеспечивающие поперечное перемещение центра масс. тела в зависимости от положения опорной ноги. Таким образом модель робота может сохранять статическое равновесие во время ходьбы.

Чтобы получить углы поворота сервомоторов решается задача инверсной кинематики. Полученные углы передаются системе управления роботом.

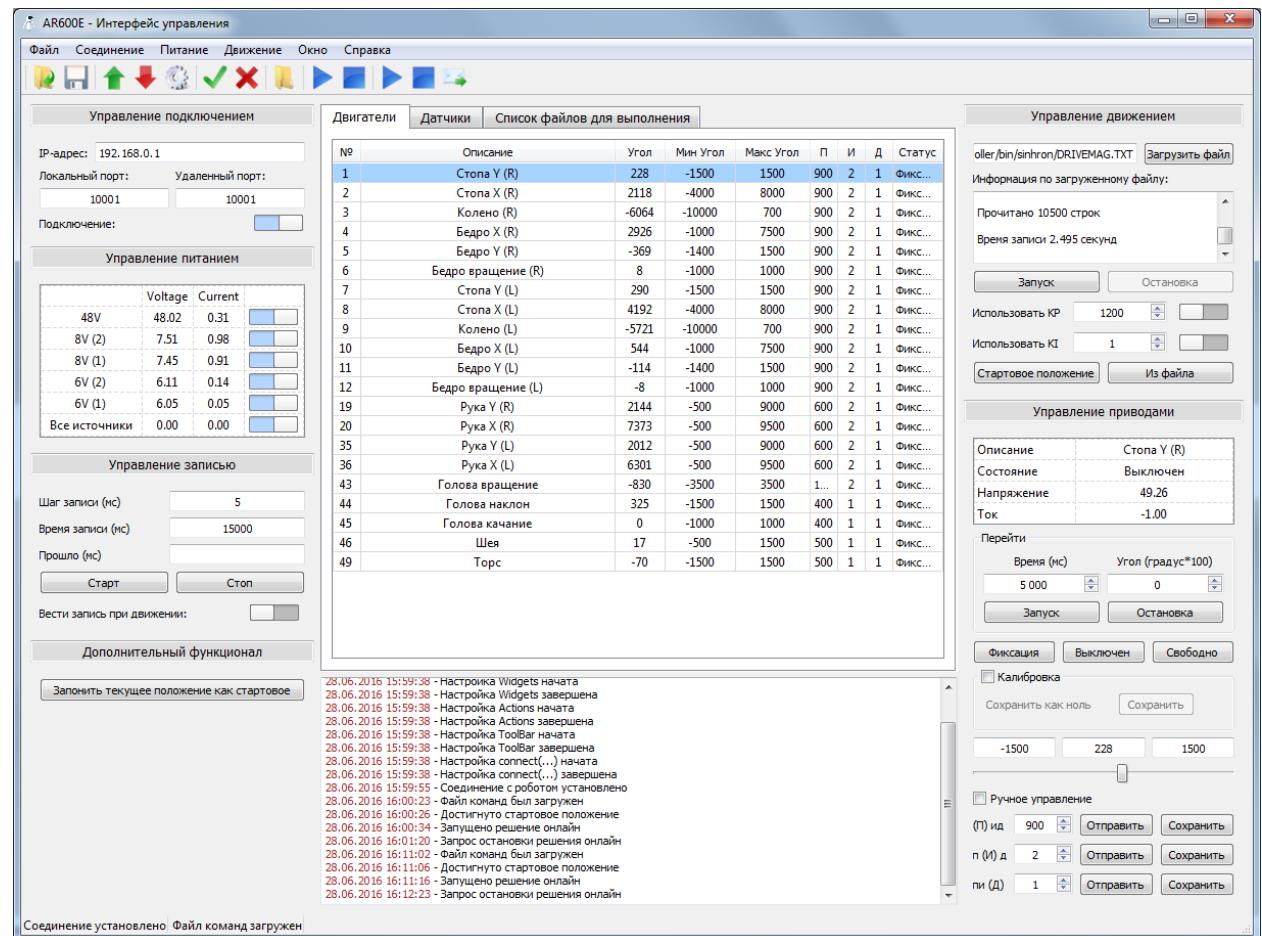
## Система управления роботом



Рисунок 17 - Структурная схема управления движением робота

На схеме можно выделить следующие блоки:

- - Оператор (пульт, интерфейс) который задает нужные движения и его параметры;
- - Устройство управления (система управления) которая формирует сигналы управления, используя данные полученные от датчиков и формирователя движений;
- - Формирователь движения, который на основе данных, полученных от оператора и устройства управления делает расчет движения для текущего момента времени и формирует последовательность, которая будет отправлена РТС через устройство управления.
- Формирователь движений является важной частью системы управления, так как результатом именно его работы являются программные углы, используемые для управления двигателями.



• Рисунок 18 - Интерфейс системы управления роботом

Схема управления получает от модели робота, рассчитываемой во ФРУНДе углы поворота сервомоторов и передает через Ethernet соединение на материнскую плату робота, которая обеспечивает их выполнение.

Таким образом, робот повторяет действия, которые совершает модель робота во ФРУНДе.

Более подробную информацию по вышеупомянутым вопросам можно найти в работах Павла Тарасова, Скорикова Андрея и Александра Сергеевича Горобцова.

### 3. Постановка задачи создания системы автономной навигации AR600E

Тема автономной навигации антропоморфных роботов крайне обширна, и возможных решений, и их конфигураций можно реализовать огромное количество. То же касается и подсистемы планирования маршрута для робота.

На протяжении исследований схемы многократно менялись и совершенствовались. Таким образом мне удалось получить два варианта решений, которые по – своему подходили под возможности движения робота.

Основные цели создания и требования, выдвигаемые при разработке системы технического зрения

Целю является создание программного комплекса, который позволит роботу прокладывать маршрут движения, а также корректировать информацию по мере прохождения данного маршрута. Система должна быть основана на технологиях компьютерного зрения.

Система должна быть надежной, т.е. гарантированно отвечать на запросы, формируемые ФРУНДом.

#### Функционал системы автономной навигации

Система автономной навигации должна предоставлять ФРУНДу по запросу информацию о маршруте движения робота, а также информацию о возможности совершения шага в данную точку и информацию о поверхности в точке наступления.

Первая из частей, заключающаяся в планировании маршрута является моей подзадачей.

#### Функционал системы планирования маршрута робота

Система планирования должна уметь рассчитывать маршрут для робота.

Она должна корректно обрабатывать некорректные ситуации, такие как запрос планирования до точки, находящейся в районе какого – либо препятствия и т.д.

Система должна предоставлять возможность задания цели планирования как из ФРУНДа, так и вручную, например при помощи визуализатора Rviz.

## 4. Определение путей и методов реализации системы автономной навигации для AR600E

### Планирование системы автономной навигации

#### Выбор платформы для реализации

Изначально планировалось разрабатывать систему, как программу, в которую напрямую подключались бы все нужные библиотеки. У многих найденных тогда библиотек (Point Cloud Library, Rtabmap, sbpl), которые были нужны для реализации системы, была возможность включения в основную программу.

У меня уже был негативный опыт разработки проектов с подключением громоздких библиотек. Основными проблемами являлось существенное время компиляции кода, что очень сильно усложняло отладку проекта, а сборка библиотек, библиотек, от которых зависят эти библиотеки, состыковка их версий и т.д.

Также в сыром виде у нас не было никаких инструментов для визуализации результатов. Их бы пришлось писать с нуля.

В это время я нашел вариант решения всех этих проблем, а именно платформу, на которой было большое количество нужных нам библиотек для навигации в формате пакетов, функционал которых позволял пользоваться большинством возможностей той или иной библиотеки, просто запустив соответствующий модуль из пакета. Далее будет рассмотрена эта и многие другие возможности, которые предоставляет ROS. Далее также будут рассмотрены основные концепции ROS, без которых понимание дальнейшего материала будет осложнено.

*O ROS (<http://www.ros.org/core-components/>)*



Рисунок 19 - Возможности, предоставляемые ROS

Robot Operating System –платформа и в каком – то смысле операционная система для разработки робототехники. Она имеет большую постоянно пополняющуюся коллекцию библиотек, в частности в следующих областях:

- Локализация
- Картография
- Навигация
- Планирование движений и маршрута
- И т.д.

Среди этих библиотек можно заметить следующие:

- OpenCV (обработка изображений)

- PointCloudLibrary (алгоритмы работы с облаками точек)
- MoveIt (планирование движений роботов в 3-мерном пространстве с учетом коллизий)
- Gazebo (инструмент для 3-мерной симуляции роботов)

Также на платформе представлено очень много решений для конкретных моделей роботов (<http://robots.ros.org/>), в числе которых есть также ряд человекоподобных роботов.

Также на платформе ROS имеется очень большое количество вспомогательных средств для отладки и разработки. Например собственный визуализатор rviz, средства записи и воспроизведения сообщений, публикуемых в системе, что позволяет работать с данными, поступающими с датчиков в систему при их физическом отсутствии, используя заранее заготовленную запись. И это далеко не исчерпывающий список возможностей, которые предоставляет ROS.

#### *Основные концепции ROS*

В ROS можно программировать на Python 2 или на C++. У обоих подходов есть свои плюсы и минусы.

В ROS программы существуют в виде Node. Это исполняемый файл который может использовать возможности и библиотеки ROS, а также любые другие сторонние библиотеки.

Node работают независимо друг от друга, за счет чего достигается асинхронность работы системы в целом. Взаимодействуют друг с другом они через Topics (в дальнейшем «топик»), которые представляют из себя своего рода именованные почтовые ящики, принимающие сообщение определенного типа.

В ROS существует много стандартных типов сообщений, от std\_msgs/Bool до sensor\_msgs/PointCloud или nav\_msgs/OccupancyGrid. Комбинируя их можно создавать собственные типы данных для своих нужд.

Из Topics можно читать сообщения. Для этого Node (в дальнейшем «нода») должна создать объект типа Subscriber, через который она сможет читать соответствующие сообщения в callback'е.

Аналогично с публикацией сообщений. Если Node хочет публиковать сообщения в какой – нибудь Topic, она должна создать объект типа Publisher и через него публиковать сообщения в соответствующий Topic.

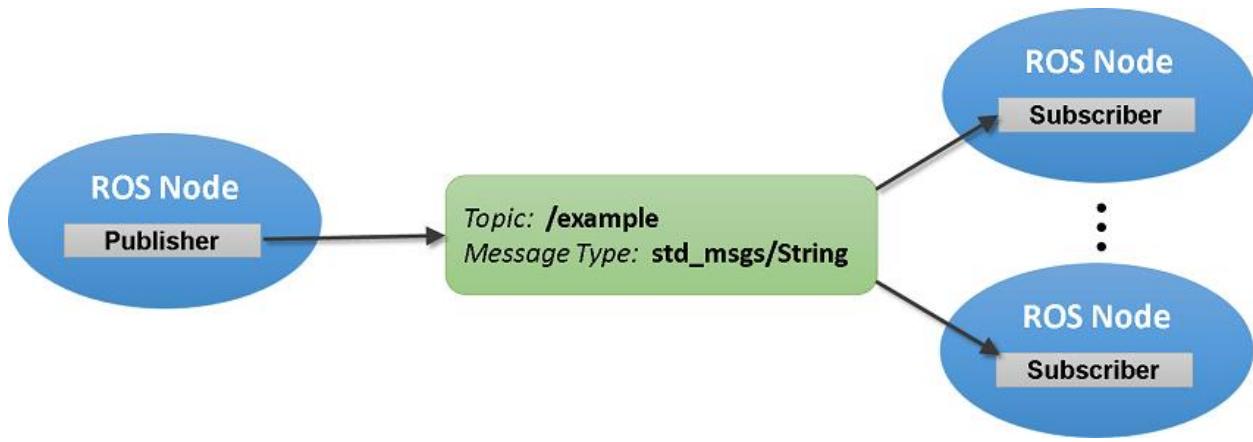


Рисунок 20 - Вариант взаимодействия между несколькими Nodes

Таким образом достигается асинхронность работы всей системы. Эта архитектура хорошо ложится на нужды робототехнических систем, в которых много датчиков, ноды которых публикуют информацию, получаемую с них.

Программы же, обрабатывающие эту информацию в этом случае не должны все время проверять доступность данных. Когда в топике появится сообщение, произойдет вызов callback'a, который запустит соответствующие вычисления.

Также имеются способы синхронного взаимодействия между нодами, которые называются сервисами (Services). Они работают сродни блокирующему удаленному вызову процедуры.

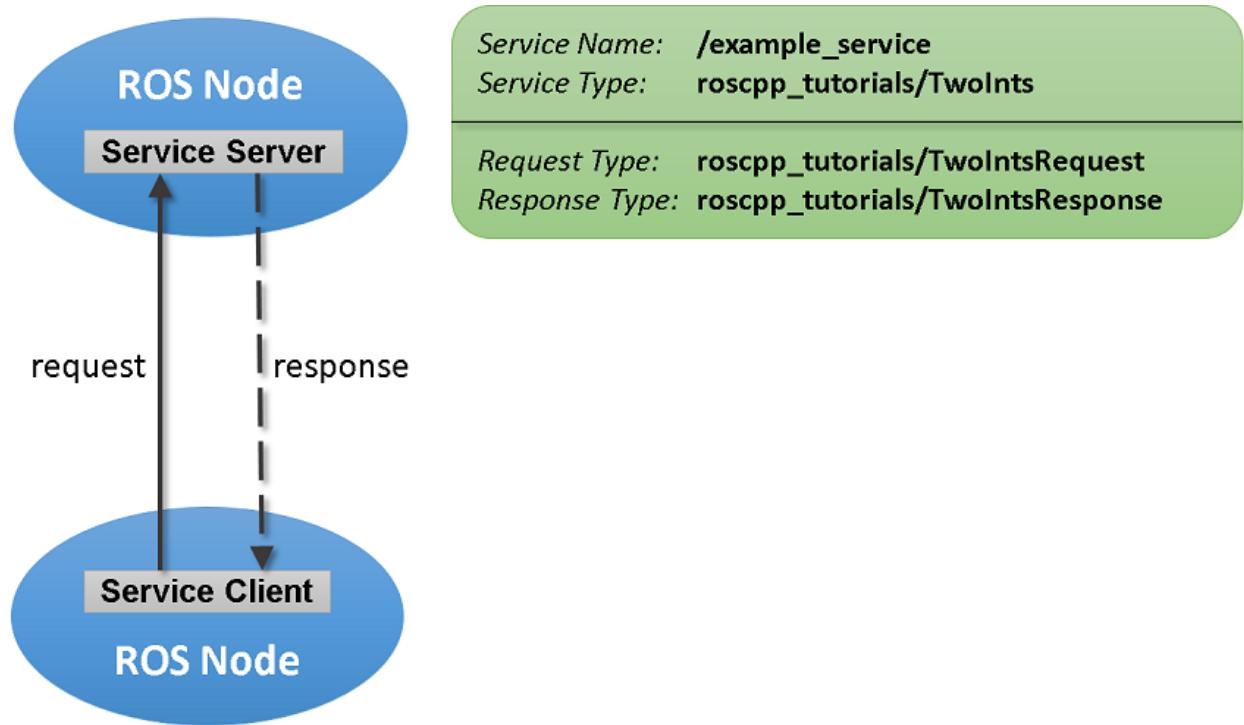


Рисунок 21 - Взаимодействие клиента и сервера сервиса

Итак, несколько нод, скрипты, необходимые для их запуска, конфигурационные файлы обычно объединяются в пакеты и, сопровождаемые документацией и информацией о требуемых и публикуемых топиках, нодах и сервисах, публикуются на Wiki ROS'a.

Выбор SLAM алгоритма и источников данных для него

#### *O SLAM алгоритмах*



Рисунок 22 - Облако точек, полученное реализацией SLAM алгоритма от библиотеки rtabmap

Для планирования, навигации и ориентации в окружении необходимо уметь строить карту окружения и определять свое местоположение в ней на каждый момент времени по данным с датчиков.

Для решения этих задач хорошо подходят SLAM алгоритмы (Simultaneous localization and mapping). Существует много различных подходов и вариантов реализации SLAM алгоритмов, различающихся как принципом работы, так и набором датчиков, с данными которых они работают. (<https://www.openslam.org/>)

Vision-based SLAM алгоритмы позволяют строить карту окружения и приблизительно оценивать местоположение в ней. Несмотря на вероятностный характер оценки, нам удалось получать вполне удовлетворяющие наши потребности результаты.

На изображении выше приведены результаты работы выбранной нами реализации SLAM алгоритма из библиотеки rtabmap, которые были получены одной из команд участниц IROS 2014 Kinect Challenge. Синим цветом отмечена траектория движения робота.

#### *Обзор SLAM алгоритмов*

Существует большое количество реализаций SLAM алгоритма как на ROS платформе, так и в сторонних библиотеках. На сайте openSLAM (<https://www.openslam.org/>) представлен внушающий набор подобных библиотек.

К сожалению, далеко не все из них продолжают поддерживаться их авторами и по поводу многих из них есть сомнения в надежности их работы .

К счастью, на платформе ROS имеется ряд отличных реализаций SLAM алгоритмов.

`rtabmap_ros` (<http://introlab.github.io/rtabmap/> )

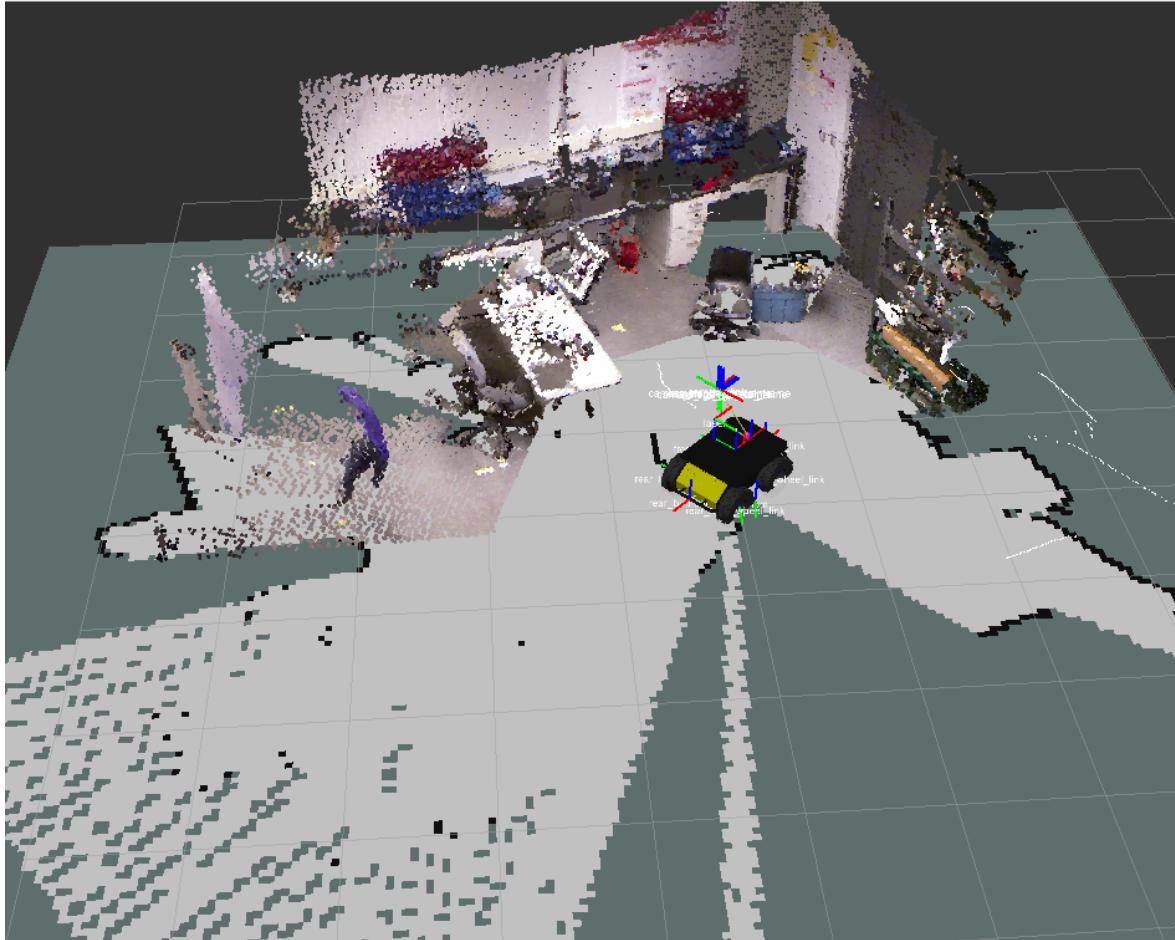


Рисунок 23 - Пример работы алгоритма. Черно - белая плоскость на «полу» - карта препятствий

ROS пакет, предоставляющий обвертку над библиотекой `rtabmap`. Наверное, самый популярный, удобный в использовании и многосторонний пакет. Из того, что он предоставляет пользователю следует особо отметить следующие особенности пакета:

- Качественная документация
- Поддерживает многие датчики в качестве источников данных (ряд RGB-D камер, стерео камеры, лидары)
- В меру требовательна к ресурсам даже при создании больших карт (хватает ноутбука с Intel Core i3 + 4 Гб RAM)
- Имеет множество интересных настроек, влияющих на скорость работы / качество
- Предоставляет много дополнительных функций.

Например:

- строит карту препятствий путем проецирования групп точек, начиная с некой высоты (можно заметить на изображении выше)
- имеет интеграцию с move\_base (стандартное средство ROS для планирования маршрута для колесных роботов)
- имеет ноду сегментации облака точек на 2 других облака (препятствия и земля)
- Интеграция с библиотекой Octomap (спец. структуры для создания, хранения и обработки плотных облаков точек)

[hector\\_slam \(\[http://wiki.ros.org/hector\\\_slam\]\(http://wiki.ros.org/hector\_slam\)\)](http://wiki.ros.org/hector_slam)

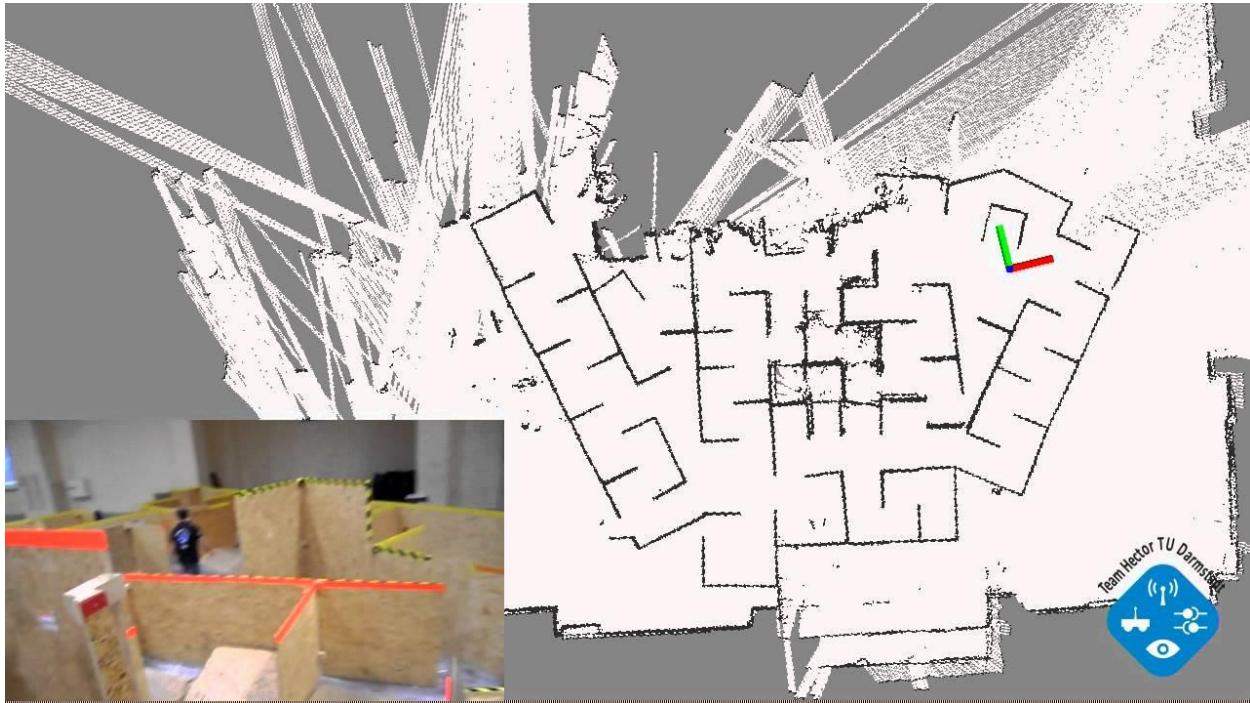


Рисунок 24 - Пример работы алгоритма на одном из соревнований по автономной навигации

SLAM алгоритм, строящий 2D карту препятствий.

По умолчанию использует данные от лидара, но с тем же успехом ему на вход можно подавать данные со стереокамеры, RGB-D камеры.

Близкий родственник: `slam_gmapping`.

Наверняка это самые легковесные реализации SLAM алгоритмов.

ElasticFusion (<https://github.com/mp3guy/ElasticFusion> )

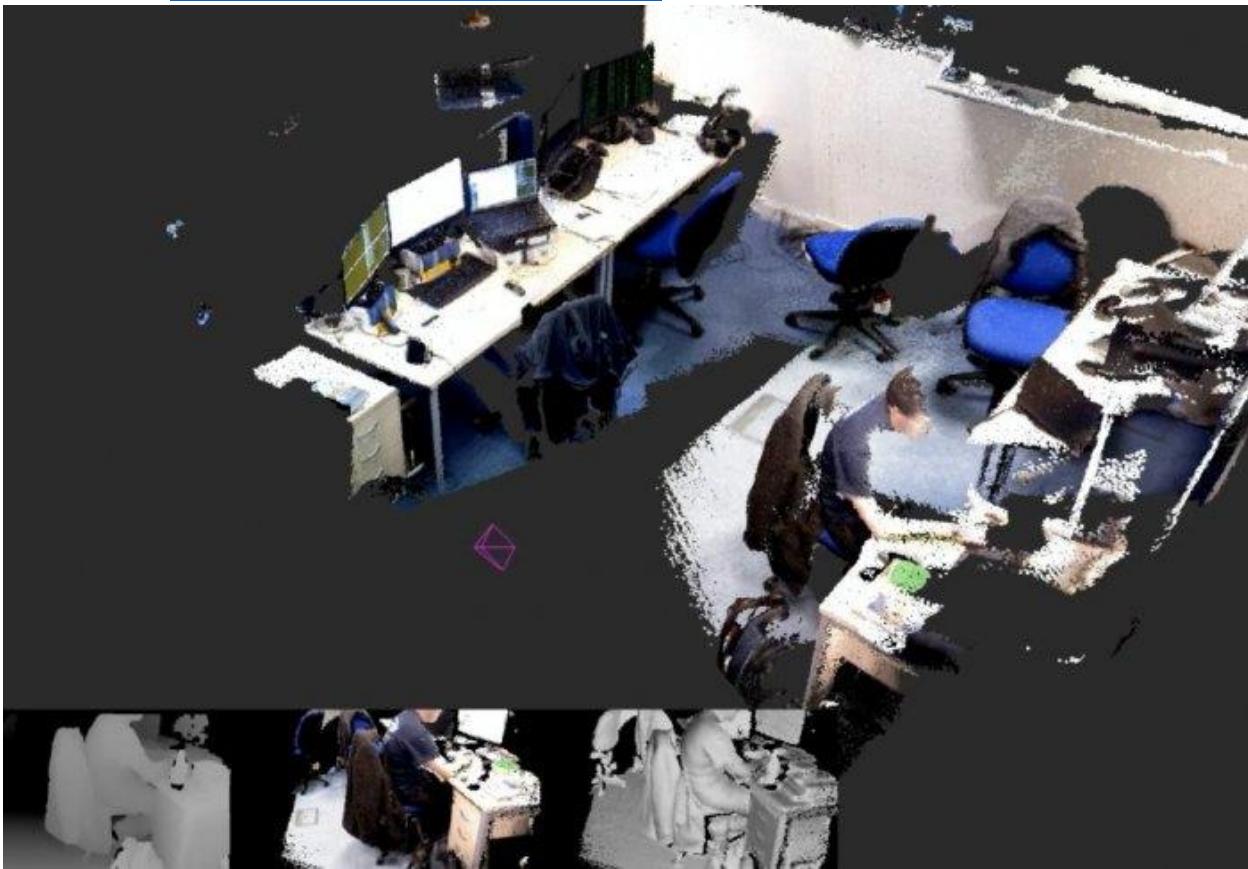


Рисунок 25 - Результатом работы алгоритма - мешь, а не облако точек

- строит мешь
- высокие требования к ресурсам (GPU)
- Не имеет пакета в ROS

#### *Выбор SLAM алгоритма*

Мы выбрали реализацию SLAM алгоритма из пакета rtabmap\_ros, который относится к библиотеке rtabmap. К тем преимуществам, которые были описаны ранее можно добавить также простоту надежность в работе и богатство API, которое позволило в дальнейшем совместить системы координат карты от алгоритма и системы координат ФРУНДа.

Были рассмотрены также и некоторые другие реализации SLAM алгоритма, такие как Kinect Fusion (<https://msdn.microsoft.com/en-us/library/dn188670.aspx>) , KinFu от PCL ([http://pointclouds.org/documentation/tutorials/using\\_kinfu\\_large\\_scale.php](http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php) ), но они подходят больше для получения 3D моделей небольших предметов или помещений и весьма ресурсоемки.

## *Выбор источников данных для SLAM алгоритма и разработка схемы коммуникации*

ROS – распределенная система. Таким образом 1 ПК, взаимодействующий с датчиками может находиться на 1 машине с запущенным ROS. Другой же ПК (например более мощный) может быть связан с ним по сети. ROS предоставляет возможность им работать в одном пространстве.

Таким образом, данные с датчиков будут появляться на другом ПК, который будет заниматься их обработкой и например построением маршрута, тут же отсылая команды по его выполнению другому ПК, находящемуся на роботе.

Данная возможность пока не используется, но между тем есть идеи по ее применению.

Rtabmap предоставляет возможность использовать информацию от нескольких датчиков (данные одометрии, Kinect'ы от 1 до 2 штук, данные от лидаров или стереокамер).

Один из вариантов такой системы продемонстрирован на рисунке ниже.

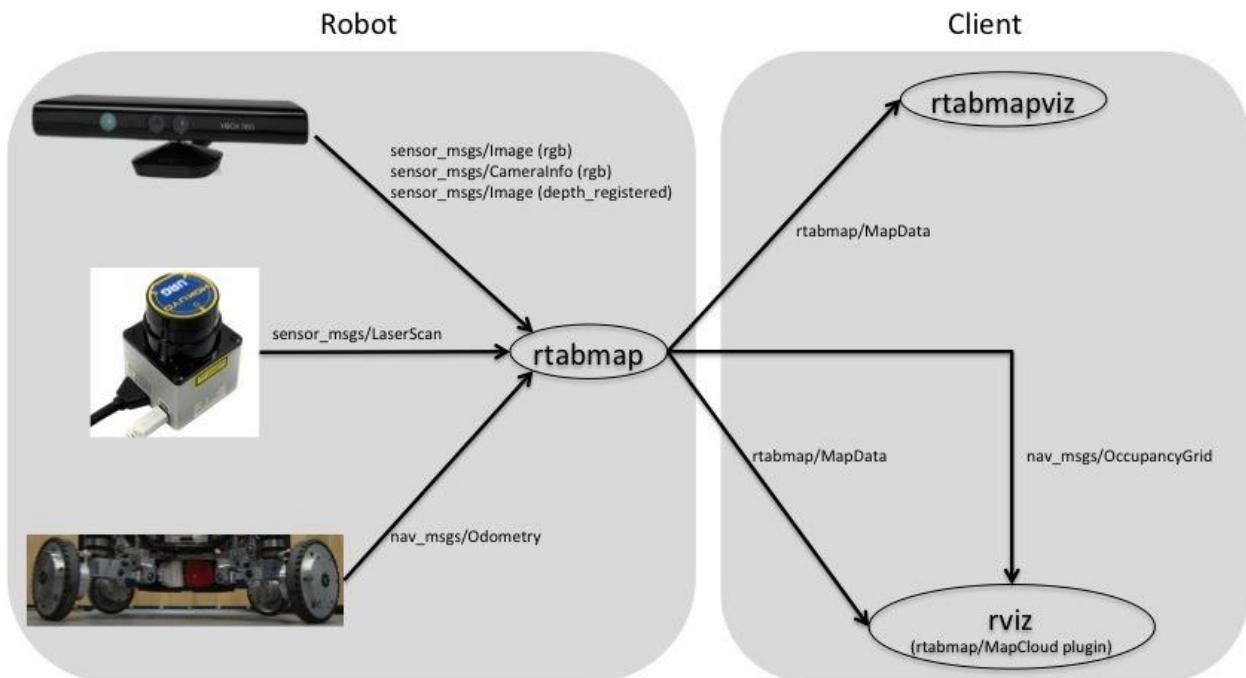


Рисунок 26 - Пример конфигурации робота для rtabmap

В нашем случае схема на данный момент состоит из одного Kinect'a. Также у робота есть инерциальная навигационная система, данные с которой также могут пригодится SLAM алгоритму для увеличения точности аппроксимации местоположения и построения карты.

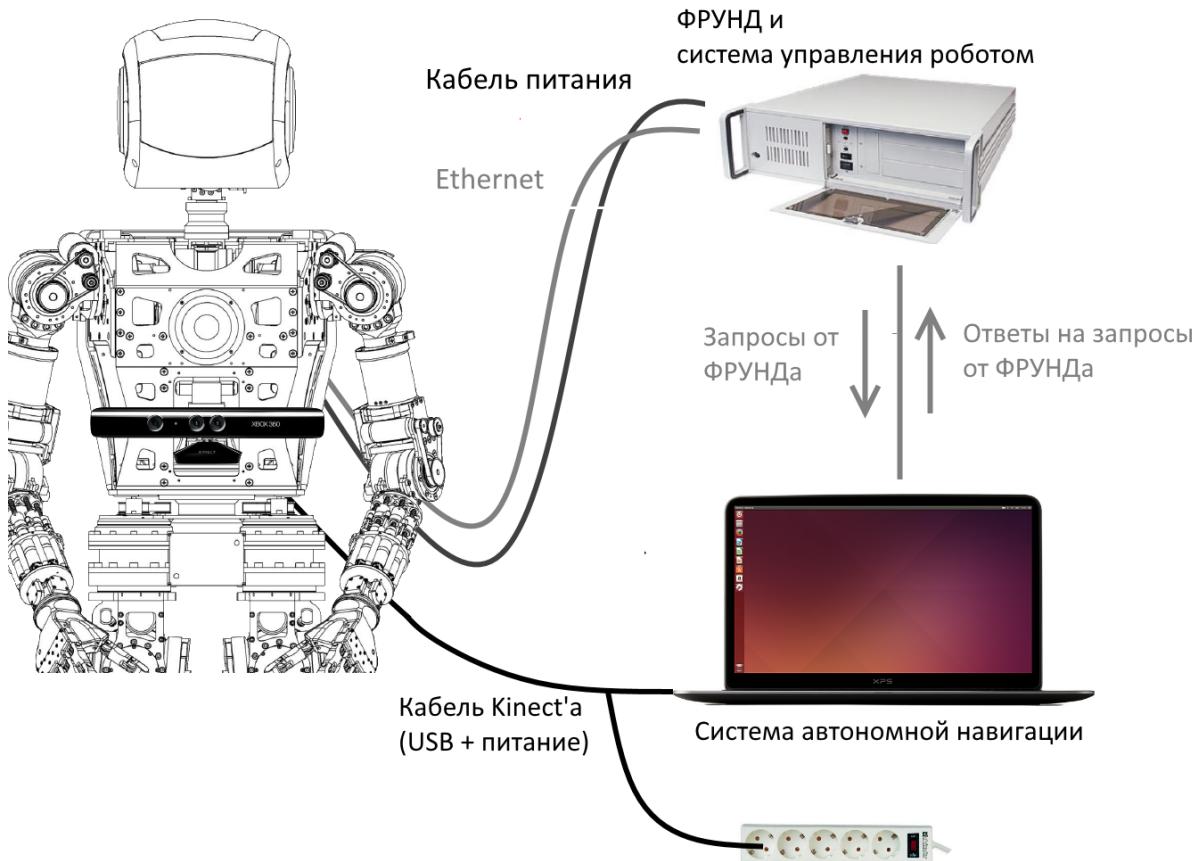


Рисунок 27 - Примерная схема подключения робота и Kinect на данный момент.

У робота есть бортовой компьютер, который в будущем можно задействовать для приема данных с датчиков и отправки их на ноутбук, на котором запущена система автономной навигации.

Однако совместить ноутбук и ПК с системой управления роботом и ФРУНДом не выйдет, т.к. ROS привязан к Linux – based операционным системам, а ФРУНД и система управления – к Windows. Портирование чего либо из этого списка – крайне нерациональная задача.

Так как кабель Kinect гораздо короче кабеля питания и Ethernet кабеля от робота к Пк с системой управления, то возможен также и такой вариант соединения. Правда придется разобрать Kinect и

подключить его к системе питания робота. Таким образом можно избавиться хотя бы от ограничения длины кабеля Kinect'a.

Работа же SLAM алгоритма на борту робота сильно затруднена, т.к. бортовой ПК робота не обладает достаточной производительностью для работы SLAM алгоритма в реальном времени (см 2 раздел про описание технических характеристик бортового ПК). Однако это не мешает выступать ему в качестве посредника при передаче данных с Kinect'a.

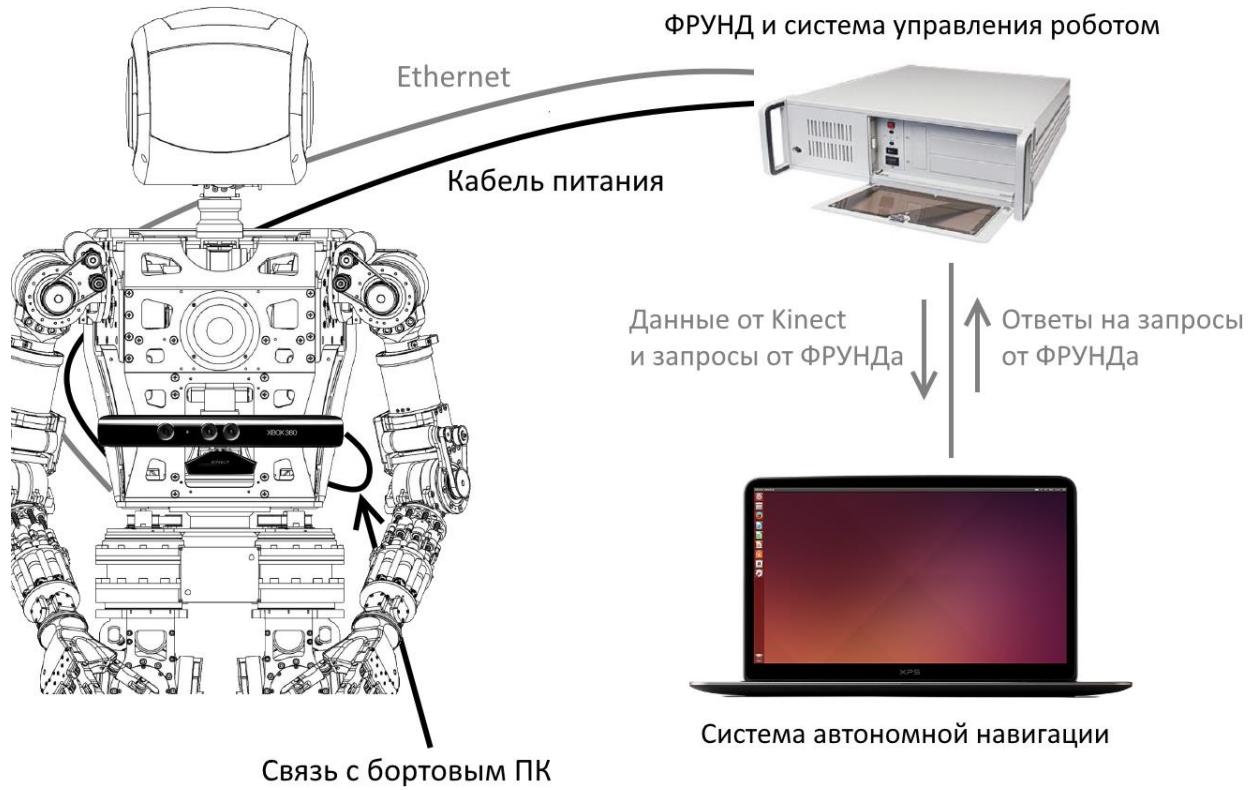


Рисунок 28 - Второй вариант соединения. Kinect передает данные через бортовой ПК

Разработка архитектуры системы автономной навигации

*Важные заметки по поводу устройства системы управления роботом*

На данный момент ФРУНД не предоставляет никакого API системе реализованной компьютерного зрения. Таким образом, система компьютерного зрения не может воздействовать на робота напрямую, например, «попросив» робота пригнуться неким нестандартным образом по причине наличия специфического препятствия по курсу движения робота.

Это накладывает некоторые ограничения на систему компьютерного зрения. В данной ситуации она может лишь информировать ФРУНД о наличии препятствий, возможной траектории безопасного движения и т.д. (т.е. взаимодействовать по заранее заданному жесткому протоколу)

В системах, описанных ранее система компьютерного зрения совмещена с планером, что позволяет ему планировать движения, исходя не только из уравнений равновесия и движения, но и из наличия препятствий на пути движения.

Такой подход позволяет роботу проводить сложные, комплексные движения, учитывая при этом положение его равновесия.

В последующих главах будет рассмотрен пакет MoveIt, который позволяет планировать движения для модели робота с учетом коллизий с окружением. Если неким образом совместить его с планированием движения во ФРУНДе (условия устойчивости и т.д.) и решением задачи инверсной кинематики, то вполне возможно получится нечто комплексное, что сможет полностью использовать данные об окружении при ходьбе и каких – либо сложных действий и манипуляций.

К сожалению, у меня нет идей об осуществлении данной состыковки, т.к. MoveIt и ФРУНД – вещи, грубо говоря, из разных миров.

В дальнейшем следует учитывать вышесказанное. И хотя в данной реализации робот не использует все информацию об окружении, как было сказано выше, то, что вышло в итоге вполне позволит ему в дальнейшем автономно перемещаться в простых окружениях (не по пересеченной местности).

## Архитектура системы автономной навигации

Параграф выше приводит нас к представленной ниже системе.

В данном случае система компьютерного зрения выступает в качестве вспомогательной для системы генерации движений (модуль во ФРУНДе).

Рассмотрим основные модули:

### Модуль планирования траектории

Предоставляет ФРУНДу информацию о маршруте движения, свободном от препятствий, по которому может пройти робот.

### Модуль корректировки параметров шага

В процессе прохождения траектории и генерации шагов ФРУНД запрашивает информацию о сгенерированных им шагах у данного модуля и корректирует параметры шага так, чтобы избежать столкновения с мелкими предметами или наступить на более подходящий для шага участок поверхности. Более подробную информацию об этом модуле можно почерпнуть в работе Маркова Алексея.

### ФРУНД (модуль управления роботом и генерации движений)

Генерирует устойчивые движения для робота на основе полученных от двух вышеописанных модулей.

В данном случае есть два варианта реализации, в зависимости от того, кто генерирует шаги для робота.

1) ФРУНД генерирует шаги, стараясь придерживаться глобальной траектории.

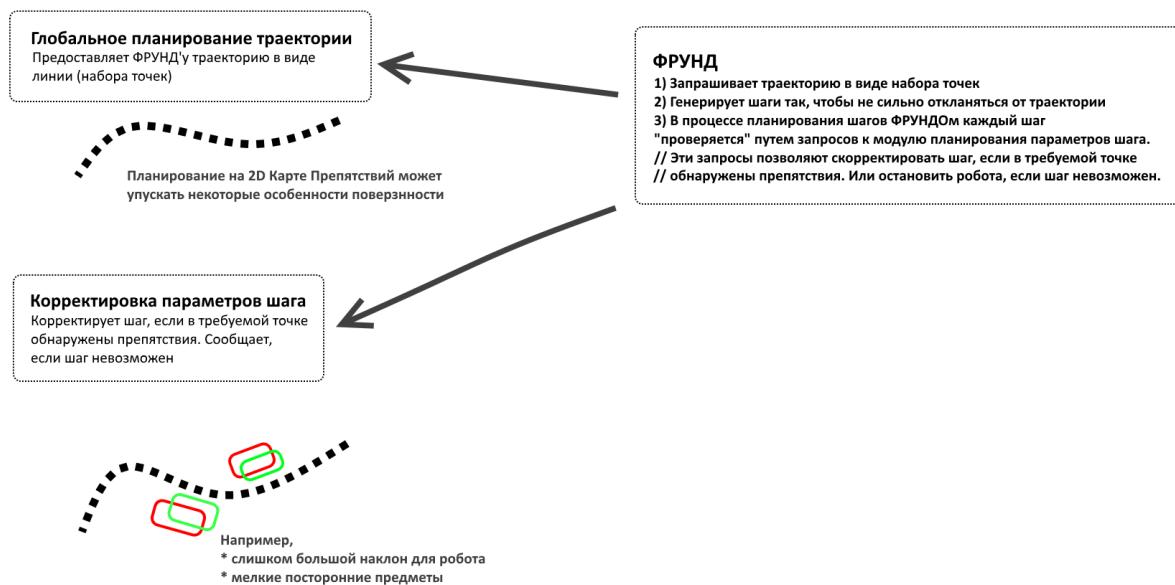


Рисунок 29 – Схема системы автономной навигации в первом случае

Генерация определенных положений для стоп на стороне ФРУНДа используется в модели для удержания равновесия (метод нулевого момента), а траектория – как желаемый ориентир для робота.

В таком случае необходимо спланировать траекторию в виде набора точек (линии), которые ФРУНД использует при ходьбе, решая уравнения локомоции так, чтобы не сильно отклоняться от спланированной системой компьютерного зрения траектории и таким образом следовать за ней.

Помимо генерации шагов, он запрашивает на каждый шаг у модуля корректировки параметров шага информацию о том, есть ли в точке наступления какие – нибудь мелкие препятствия и куда лучше наступить в этом случае. Получив информацию об этом он корректирует параметры данного шага в соответствии с полученной информацией и цикл повторяется снова.

2) ФРУНД выполняет шаги, которые генерирует система компьютерного зрения (используя планер, выдающий траекторию в виде набора положений ступней).

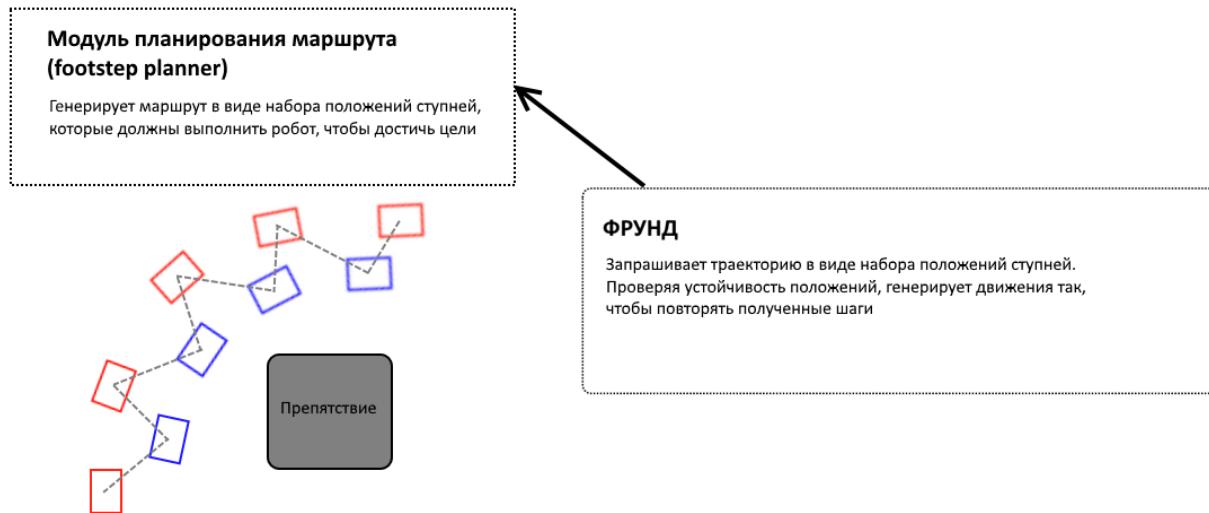


Рисунок 30 - Схема системы автономной навигации во втором случае

Этот способ возможен, но неизвестно как поведет он себя в реальной ситуации. Причина этого кроется в том, что в данном случае ФРУНД лишается возможности, удерживать равновесие при помощи совершения необходимых для выполнения метода нулевого момента шагов.

В то же время, система планирования шагов на основе компьютерного зрения не знает о физических параметрах робота. Она планирует шаги исходя из геометрических примитивов, не более. Это может привести к генерации такой траектории, которая будет потенциально опасной для робота в плане удержания равновесия (у робота есть момент, масса и импульс).

Это может и скорее всего приведет к тому, что робот будет вынужден выполнять ненадежную траекторию, не имея возможности поддерживать свое равновесие при помощи правильной постановки ступней (для удовлетворения метода нулевого момента). Хотя возможно, это можно будет компенсировать смещением торса робота и верхних конечностей.

Этот способ очень часто упоминается в других работах. Например, во всех решениях, разобранных в вводной статье. К сожалению, мне не удалось найти никаких упоминаний об устойчивости этого подхода. Однако, как раз он позволяет в полной мере использовать способность антропоморфного робота переступания предметов при планировании маршрута.

Пути реализации подсистемы планирования маршрута движения AR600E

Моя задача заключается в реализации подсистемы планирования маршрута (или траектории) для антропоморфного робота. В дальнейшем большее внимание будет уделяться именно этой задаче.

Планирование по карте препятствий (Occupancy Grid) или карте высот

*Основные определения*

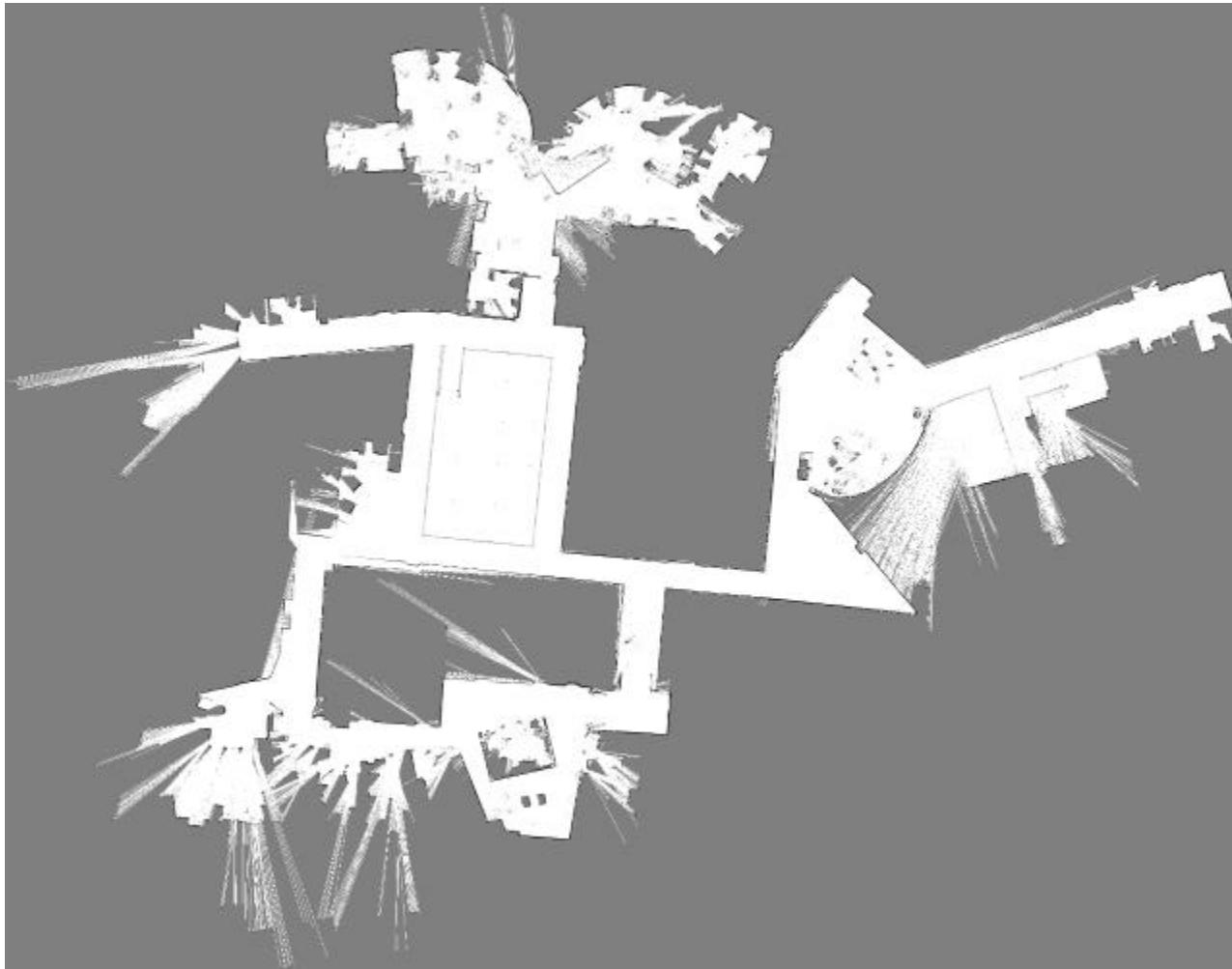


Рисунок 31 - Пример карты препятствий, построенной при сканировании некоего здания

**OccupancyGrid (или карта препятствий)** – дискретная двумерная карта, на которой каждая из ячеек может быть в одном из трех состояний:

Препятствие

Свободно

Неизвестно

Ниже изображен простейший из способов построения карты препятствий.

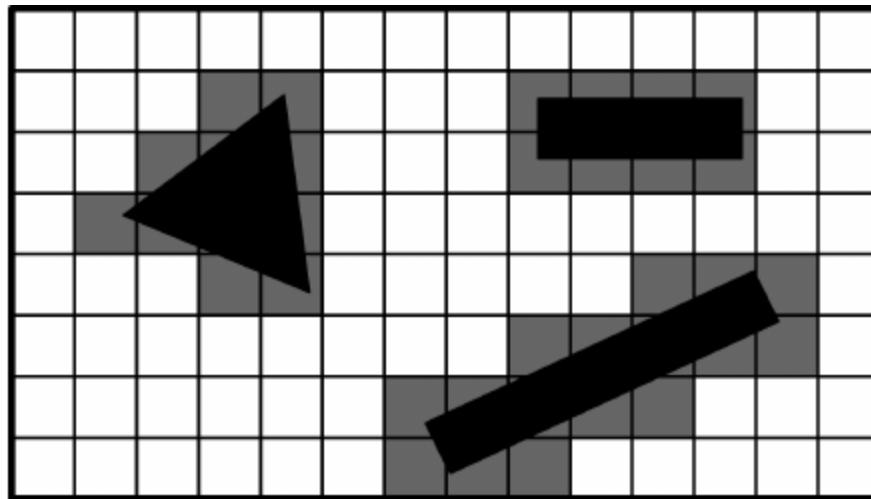
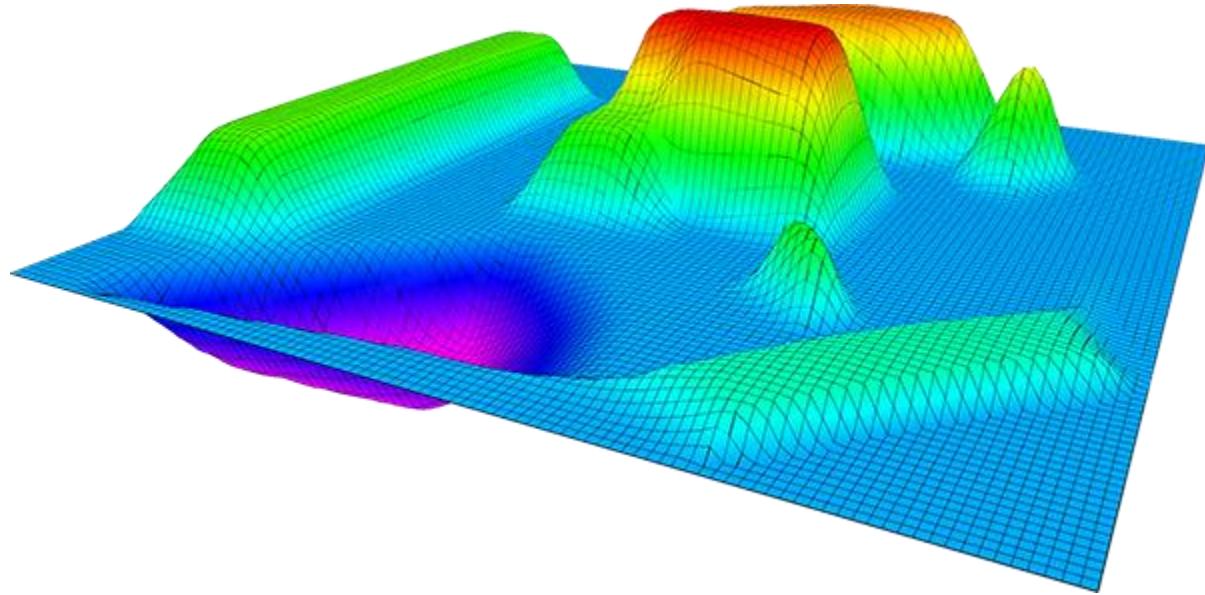


Рисунок 32 - Простейший из способов построения карты препятствий

Планирование по картам препятствий гораздо проще, нежели навигация в 3-мерном пространстве. Благодаря переходу на плоскость, мы получаем возможность эффективно использовать классические алгоритмы методы поиска маршрутов в графах состояний, которых в данном случае оказывается на порядок меньше, чем в трехмерном пространстве.

**Height map (или карта высот)** – 2.5 D карта препятствий. С ее помощью можно учитывать при 2D планировании какие – либо особенности поверхности, не теряя при этом плюсов, получаемых от перехода в 2D пространство.



#### *Способы получения карты препятствий*

Карту препятствий можно получить из облака точек несколькими способами:

- На основе среза на определенной высоте из облака точек

- Как проекцию всех точек на плоскость  $z = 0$  начиная с какой — нибудь заданной высоты  $z$ . Такой вариант предоставляет `rtabmap`. У него имеется много параметров, как то:
  - допустимый угол наклона того, что можно считать плоскостью (а не препятствием)
  - количество точек вблизи, которые стоит считать за препятствие (что позволяет не включать при построении мусор из облака точек)
  - величина дискретизации карты (что впрочем есть и в других библиотеках)
- Некоторые реализации SLAM алгоритмов изначально строят лишь карту препятствий (как например `hector_slam`, `gmapping`)

*Способы и подходы к планированию на карте препятствий*

ROS концепция `move_base` ([http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base))

Пакет представляет из себя сложную систему нод, которые вместе позволяют получить глобальный маршрут и линейные и угловые скорости для каждого момента времени, рассчитанные таким образом, чтобы робот не врезался в препятствия под действием импульса.

В центре этого планирования находятся два типа планеров, которые взаимодействуют друг с другом.

**Global Planner** находит глобальную траекторию. Изначально я хотел использовать его при планировании, но он выполняет планирование, исходя из предположения, что робот – колесная платформа, которая может двигаться в любом направлении, что в случае с антропоморфным роботом совсем не так. ([http://wiki.ros.org/global\\_planner?distro=kinetic](http://wiki.ros.org/global_planner?distro=kinetic))

Для антропоморфного робота больше подходит планирование из дискретного набора движений.

**Local Planner** по траектории, полученной от Global Planner'а, управляет мобильной платформой, рассчитывая локальную траекторию движения, а также угловые и линейные скорости по всем осям для робота. ([http://wiki.ros.org/base\\_local\\_planner?distro=kinetic](http://wiki.ros.org/base_local_planner?distro=kinetic))

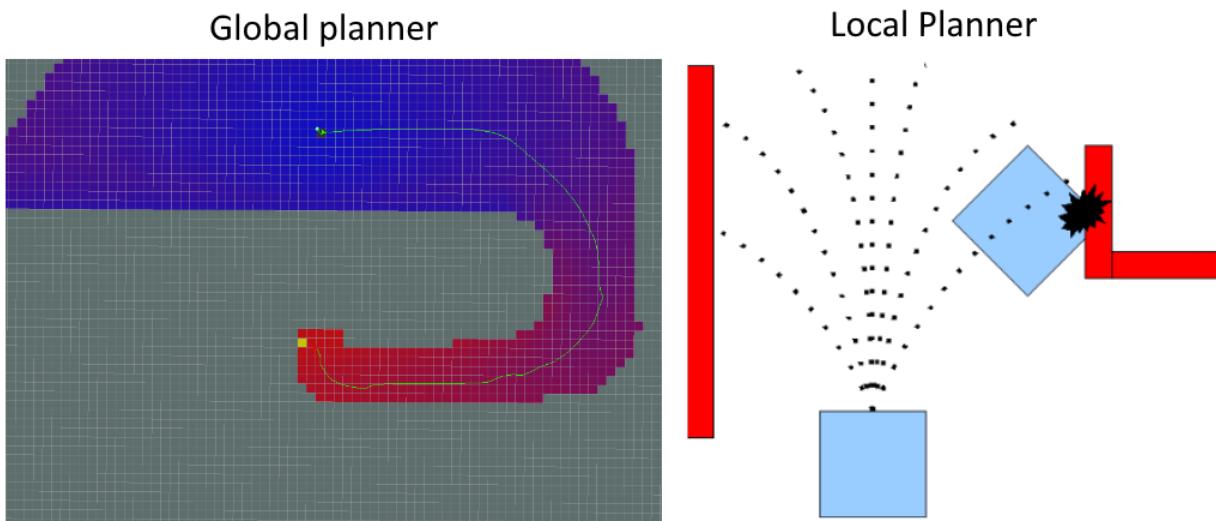


Рисунок 33 - Глобальные и локальные планеры - сердце концепции `move_base`

Из описания можно понять, что этот пакет предназначен в основном для колесных платформ, что исключает возможность его применения в нашем случае.

#### `footstep_planner`

Этот пакет – часть работы по навигации антропоморфного робота Nao в известном окружении ([http://wiki.ros.org/humanoid\\_navigation](http://wiki.ros.org/humanoid_navigation)). Этот проект также был упомянут в обзоре аналогов этой работы.

Пакет позволяет планировать маршрут для антропоморфного робота в виде набора позиций ступней, ведущей из стартовой в конечную позицию. Планирование осуществляется в пространстве состояний, где каждый следующий шаг выбирается из дискретного множества возможных шагов.

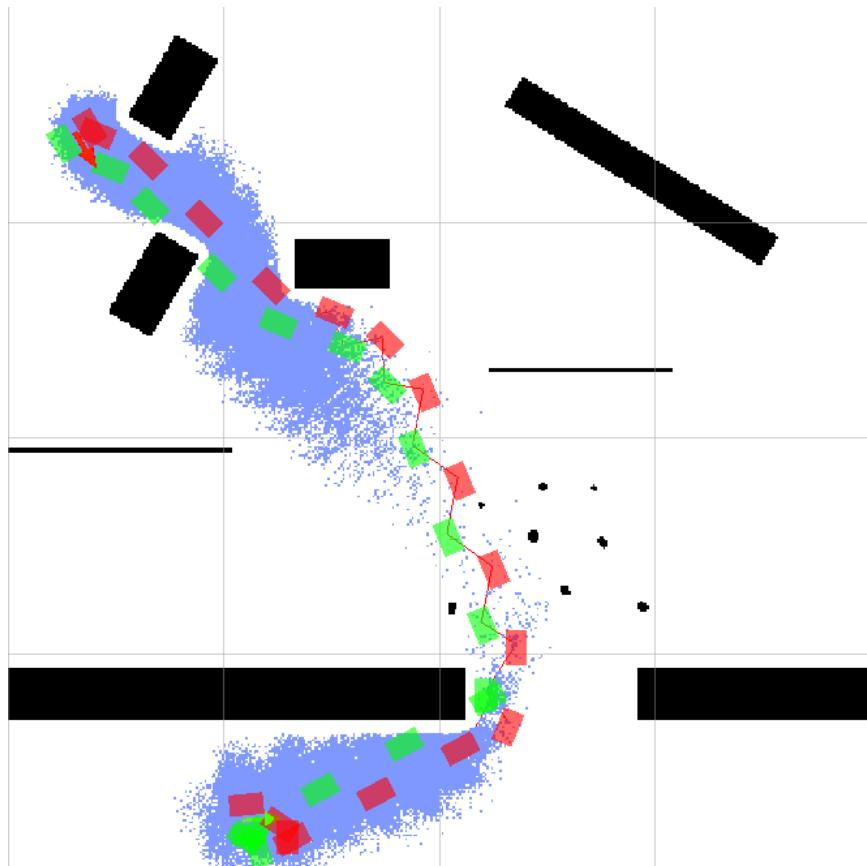


Рисунок 34 - Результат работы планера из пакета `footstep_planner`

В планере есть возможность «переступания» препятствий, в случае если ширина шага это позволяет, но по причине того, что в карте препятствий мы не знаем конкретной высоты препятствия, а знаем лишь о его наличии в данной точке, использование данной функции без применения карты высот не представляется разумным.

Пакет хорошо подстраивается под конкретного робота. Практически все параметры можно изменить (геометрические параметры ступней, максимальное расстояние между ступнями,

множество возможных шагов и т.д.), за счет чего можно настроить его для использования с любым другим антропоморфным роботом.

В пакете реализовано несколько алгоритмов планирования, среди которых A\*, ARA\*, R\*, а также различные эвристики для них (<http://hrl.informatik.uni-freiburg.de/papers/hornung12humanoids.pdf>).

Этот пакет я и использовал в дальнейшем с некоторыми изменениями и доработками.

Планирование по облакам точек (Point Cloud) и плотным облакам точек (OcTree)

#### Основные определения

**PointCloud (Облако точек)** – просто набор точек, каждая из которых имеет координаты (X;Y;Z). В библиотеке PCL (<http://pointclouds.org/>) , которая адаптирована к платформе ROS, есть много алгоритмов для обработки данных структур, как например (разные виды фильтрации, сжатие, нахождение нормалей в точках к поверхности, сегментация облаков, реконструкция мешей по ним и т.д.).

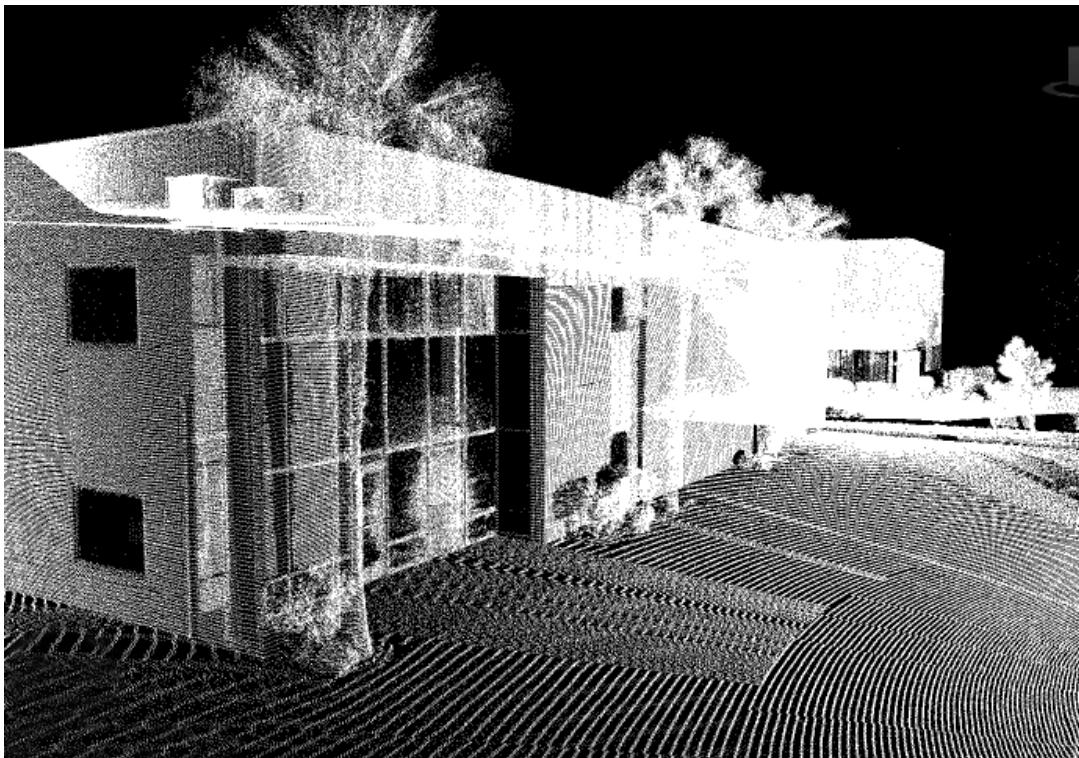
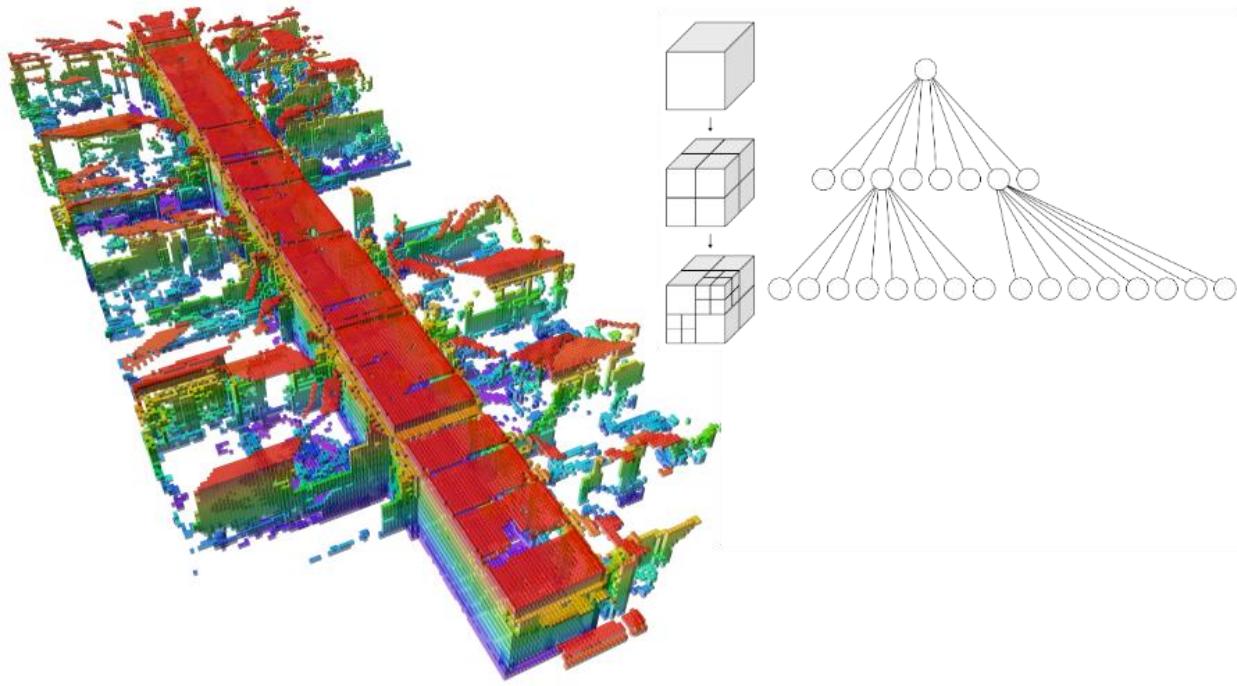


Рисунок 35 - Пример облака точек

#### Плотное облако точек



*Рисунок 36 - Пример плотного облака помещения и принципиальная схема данной структуры*

Известная мне реализация – OcTree из библиотеки Octomap (<http://wiki.ros.org/octomap>) . Грубо говоря, это аналог **OccupancyGrid**, рассмотренной выше, но в 3D. Если же посмотреть повнимательнее – это не только очень экономичный способ хранения информации об облаке точек, это также довольно мощная структура, которая заметно упрощает планирование в 3-мерном пространстве (<http://www2.informatik.uni-freiburg.de/~hornunga/pub/hornung13roscon.pdf> ).

*Способы и подходы к планированию на основе обычных и «плотных» облаков точек Vigir Footstep Planner ([http://wiki.ros.org/vigir\\_footstep\\_planning](http://wiki.ros.org/vigir_footstep_planning) )*

Это проект, который основывается на идеях пакета `footstep_planner`, но вместе с тем является его сильно изменённой версией. ([http://www.sim.informatik.tu-darmstadt.de/publ/download/2014\\_stumpf\\_footstep\\_planning\\_Humanoids.pdf](http://www.sim.informatik.tu-darmstadt.de/publ/download/2014_stumpf_footstep_planning_Humanoids.pdf))

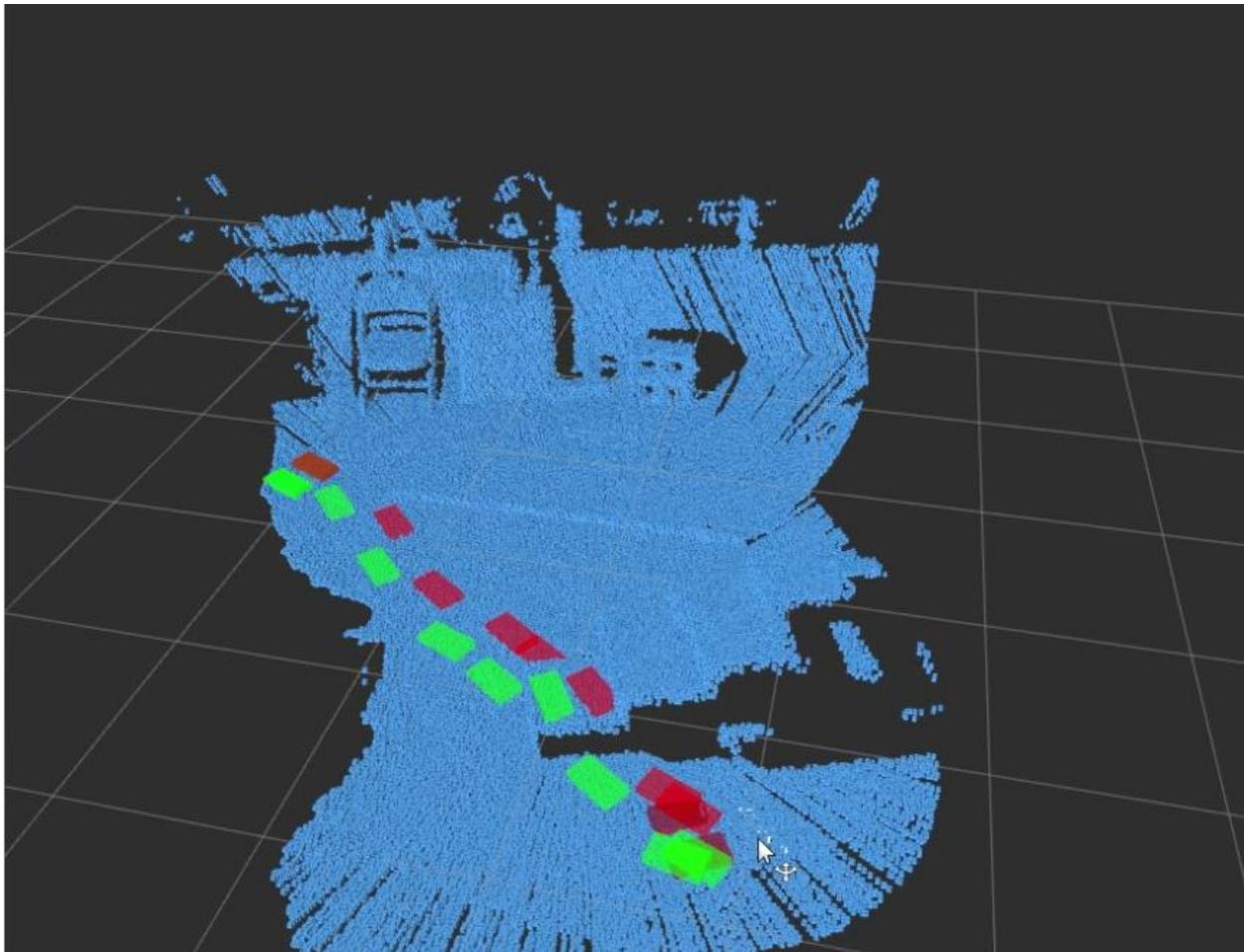


Рисунок 37 - Результат работы планира

В принципе, данный пакет также, как и `footstep_planner` может использоваться для планирования маршрута. Во время его тестирования я заметил довольно высокую скорость планирования, но малую вариативность при построении маршрута, что вероятно было связано с конкретной конфигурацией параметров алгоритма. Также алгоритм продемонстрировал возможность наступления на невысокие препятствия (на изображении выше «робот» прошелся бы по ступени и продолжил маршрут, сойдя с нее).

Однако, если вопросы об устойчивости этого дела возникали при исследовании пакета `footstep_planner`, то тут их еще больше, т.к. шаги далеко не всегда корректно размещаются на перепадах высот.

И хотя создатели предоставляют интерфейс для корректировки маршрута, этот подход вызывает некоторые опасения.

#### [3D navigation \(\[http://wiki.ros.org/3d\\\_navigation\]\(http://wiki.ros.org/3d\_navigation\)\)](http://wiki.ros.org/3d_navigation)

Это пакет для планирования в 3D, который вероятнее всего разрабатывался для колесного антропоморфного робота PR2, что нивелирует использование возможности антропоморфного робота переступать препятствия, а также без учета того, каким перемещения может выполнять робот.

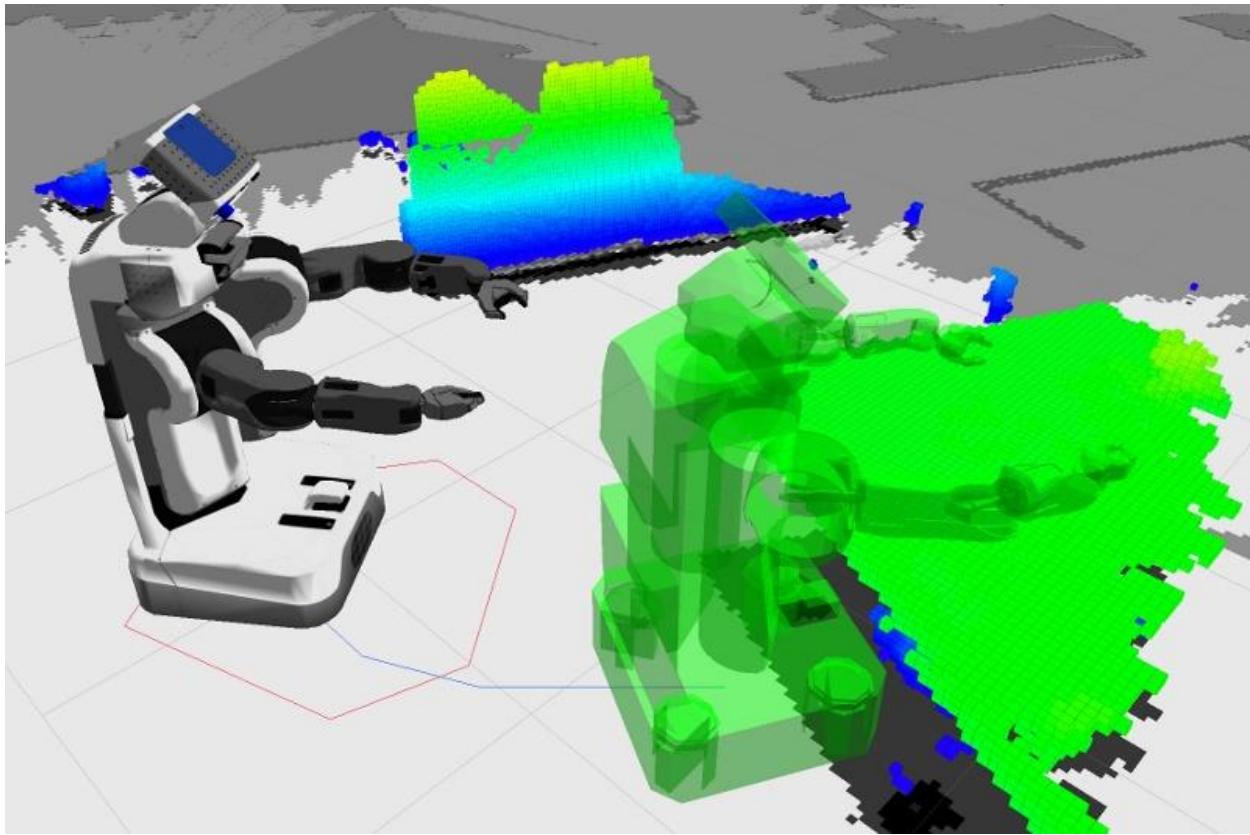


Рисунок 38 - Пример маршрута, спланированного для робота PR2 при помощи данного пакета

Планирование осуществляется в плотном облаке точек исходя из 3D статической модельки робота ([http://hrl.informatik.uni-freiburg.de/papers/hornung12icra\\_3Dnav.pdf](http://hrl.informatik.uni-freiburg.de/papers/hornung12icra_3Dnav.pdf)) . В результате получаем траекторию движения робота в виде линии. В принципе, это пригодный вариант для решения поставленной нами задачи.

Стоит заметить, что пакет на мой взгляд обладает крайне скучной документацией.

#### Movelt (<http://moveit.ros.org/>)

Это многосторонняя библиотека, которая предоставляет много возможностей в таких направлениях, как планирование движения, манипуляция объектами, расчёт кинематики, управление и навигация.

Для того, чтобы использовать ее вместе с роботом создается модель робота с учетом всех его подвижных конечностей и сочленений. Затем планер позволяет найти такой переход из начального положения модели робота в требуемое, который не вызовет коллизий с окружением, которое принимается в виде плотных облаков точек (Octree).

([http://docs.ros.org/kinetic/api/moveit\\_tutorials/html/index.html](http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html) )

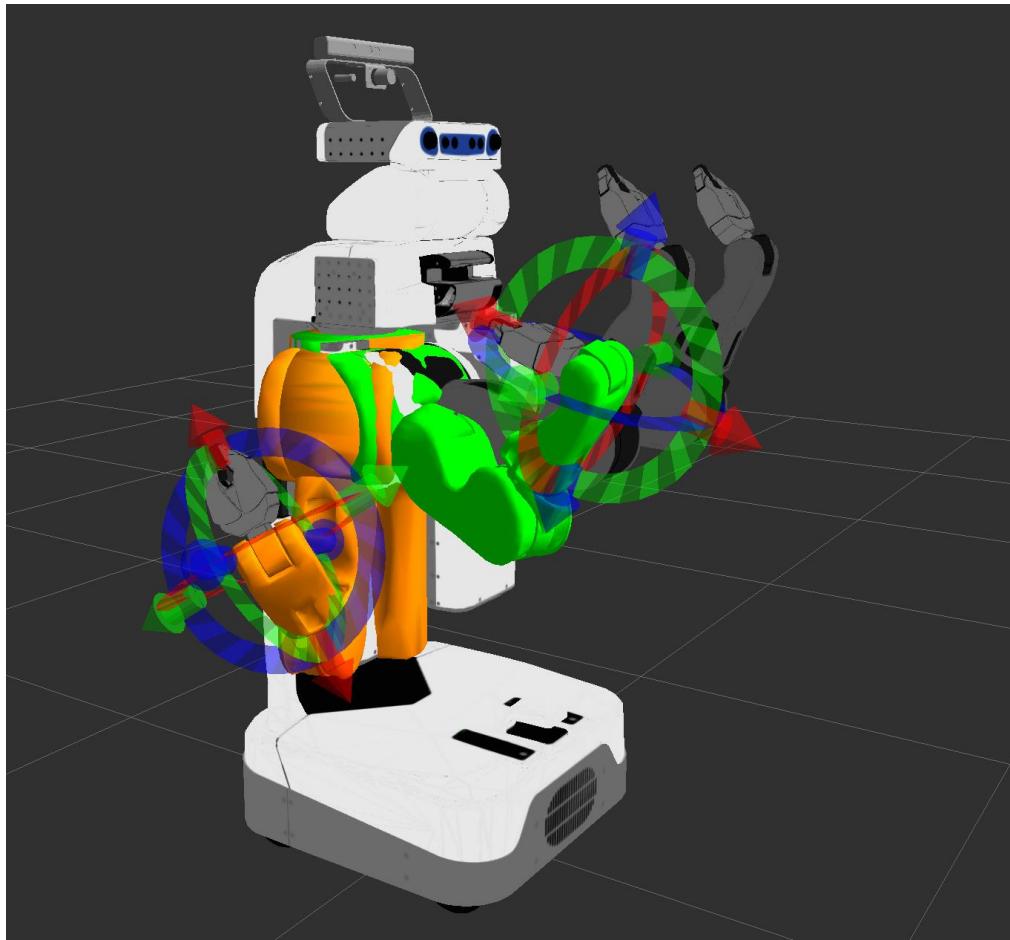


Рисунок 39 - Визуализация движений, спланированных MoveIt для робота PR2

В библиотеке есть модуль кинематики, который может решать задачи прямой и инверсной кинематики, что позволяет использовать спланированные траектории для управления роботами.

Однако, я не нашел ничего, что прямо указывало бы на наличие учета физики в расчётах. Т.е. ничего, что намекало бы на сохранение равновесия, учета физических параметров робота, таких, как, например, масса.

Это довольно интересный пакет, но, чтобы применить его на антропоморфном роботе, необходим некий сплав между ним и ФРУНДом. Подробнее об этой идее можно почитать ранее в этой работе в разделе 4.

## 5. Разработка системы автономной навигации для AR600E

### Принципиальная архитектура системы

В 4 главе данной работы описывалось 2 варианта реализации системы, в зависимости от того, как генерирует положения ступней для робота, а также что возвращает в качестве результата глобальный планер. Схемы работы в обоих случаях отличаются лишь набором используемых модулей в системе автономной навигации.

#### Вариант 1. Шаги генерирует ФРУНД

Все процессы, описанные на схеме ниже, работают асинхронно.

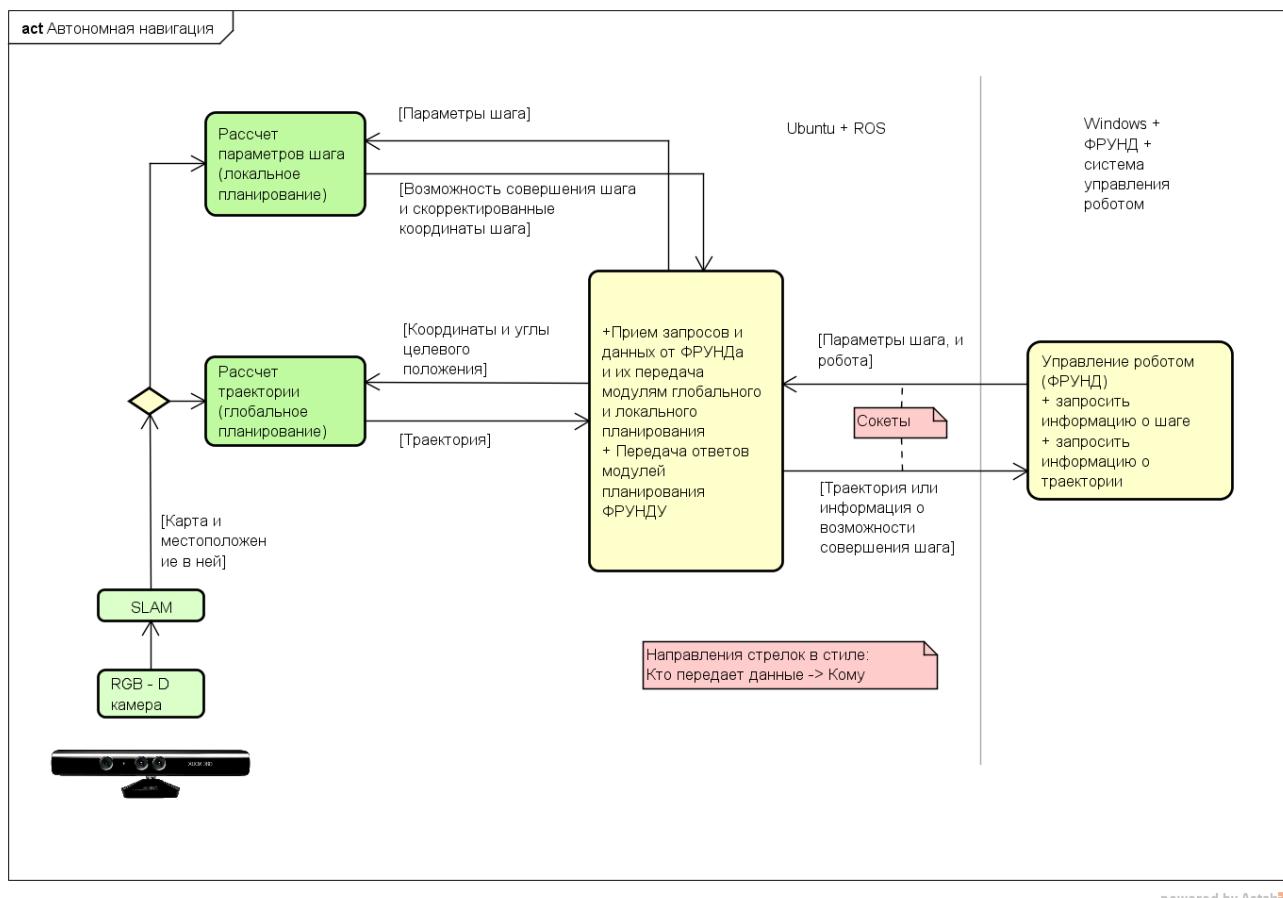


Рисунок 40 - Траектория - набор точек. Шаги генерирует ФРУНД

Рассмотрим процесс слева направо. Модули системы автономной навигации получают нужную им для работы информацию от SLAM алгоритма.

В случае с планировщиком траектории это карта препятствий, которую создает на основе проекций SLAM алгоритм из библиотеки rtabmap и местоположение камеры (а соответственно и робота) в ней.

В случае модуля планирования параметров шага, это облако точек построенной на данный момент карты окружения.

Для планирования обоим модулям осталось лишь дать задание, указав координаты, в которых нужно рассчитать параметры шага или местоположение точки, куда «желает» попасть робот.

Эта информация приходит от ФРУНДа по сокетам с довольно высокой частотой (см следующие параграфы этой главы).

Чтобы не задерживать запросы ФРУНДа на время расчета со стороны системы автономной навигации, был создан дополнительный модуль, выполняющий роль буфера.

Этот модуль (посредник) передает новые данные модулям расчета, если они завершили предыдущий расчет, по завершении расчета записывает результат в свой буфер. А также на каждый запрос ФРУНДа возвращает данные из буфера в ответ.

### Вариант 2. Шаги генерирует ФРУНД

Данная схема несколько отличается внутренним устройством системы автономной навигации.

Модуль глобального планирования здесь планирует маршрут в виде набора позиций ступней. Некоторые пример такого планирования можно увидеть ранее. Также далее будет подробно рассмотрена архитектура, особенности реализации данных модулей и их подключение в общую систему.

Модуль локального планирования в данном случае может проверяться как для проверки устойчивости шагов, генерируемых в маршруте, так и для создания карты высот (см. ранее), что позволит более адекватно учитывать при подобном планировании качество поверхности.

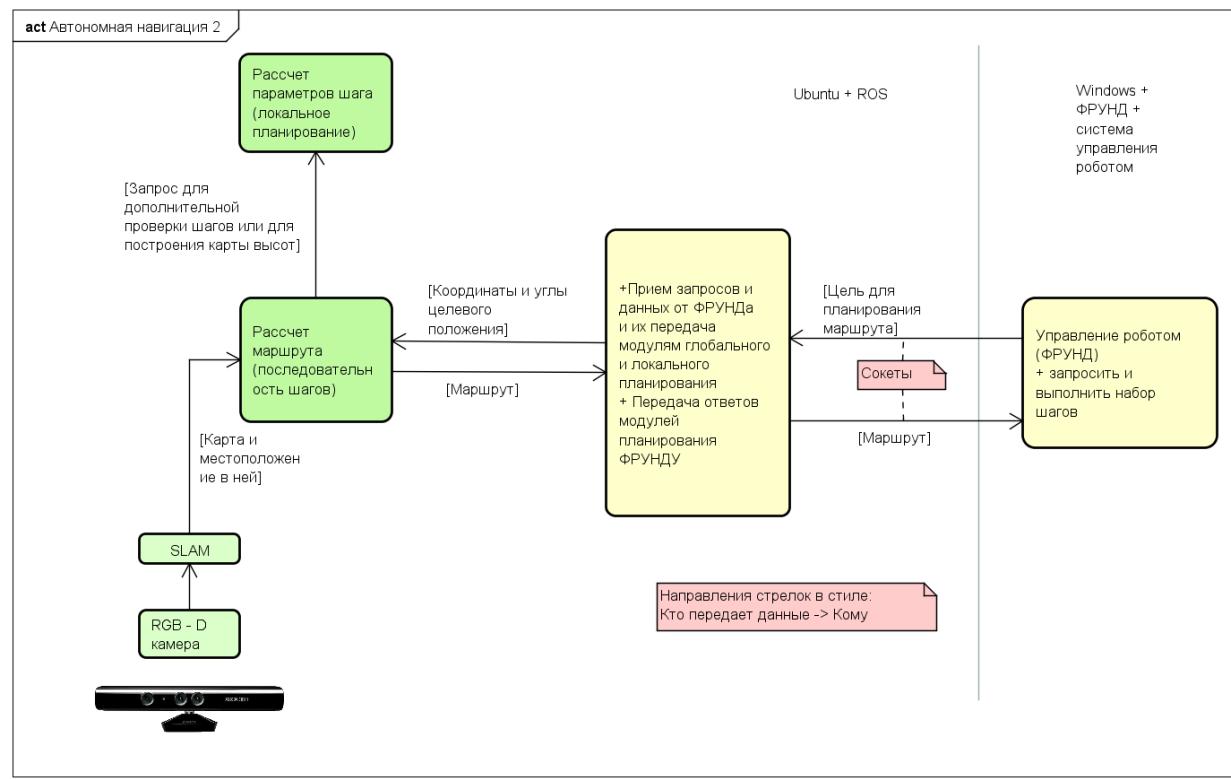


Рисунок 41 - Положения ступней (маршрут) генерирует планер

## Разработка системы взаимодействия модулей компьютерного зрения и ФРУНДа

В нашей системе используются два разных ПК, т.к. система управления роботом и ФРУНД работают только под ОС Windows, в то время, как платформа ROS работает лишь под рядом Linux систем (в нашем случае – Ubuntu 14.04 LTS).

Следует отметить, что ФРУНД запрашивает информацию в микротактах расчёта, что приводит к тому, что обращение к системе компьютерного зрения выполняется на довольно высокой частоте (~200 Гц). Каждое такое обращение от ФРУНДа является блокирующим. Таким образом, если расчет на стороне системы компьютерного зрения будет идти слишком долго или возникнут неполадки с сетью между этими двумя ПК, то это приведет к тому, что генерация движений для робота будет блокирована этим не выполнившимся вызовом, что приведет к прекращению генерации движений для робота.

Необходимо заметить, что оба модуля системы автономной навигации, к которым обращается ФРУНД затрачивают гораздо большее время, чем 1 / 200 секунды на получение результатов расчета. К этому также следует добавить незначительные задержки по сети.

На данный момент было принято решение реализовать следующую архитектуру системы, которая позволяет избежать длительного ожидания результатов вычислений ФРУНДом.

Ниже приведена диаграмма коммуникаций для взаимодействия одного модуля системы автономной навигации и ФРУНДа, а также даны пояснения относительно нее.

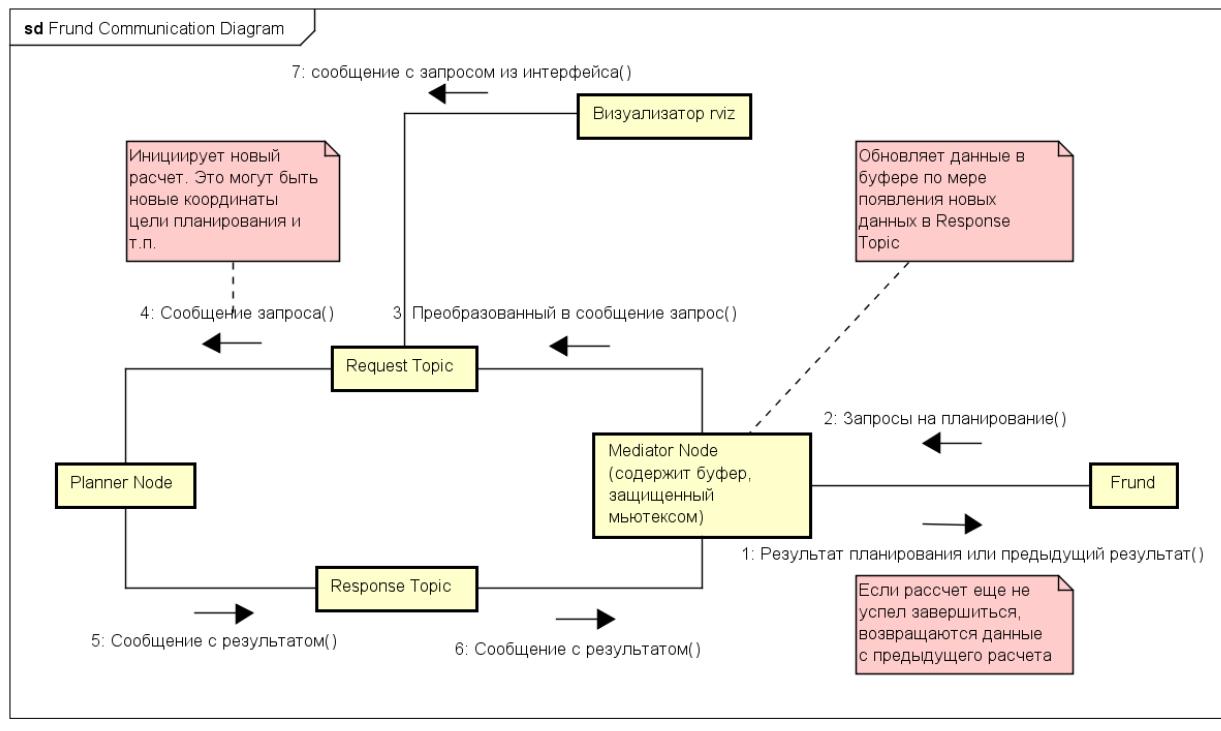


Рисунок 42 - Система коммуникаций ФРУНДа и одного из модулей компьютерного зрения

Модуль компьютерного зрения (Planner Node) читает сообщения из топика Request Topic, в который публикуются сообщения с необходимой для него информацией (координаты цели

планирования и т.п.). Приход сообщения в этот топик вызывает callback в Planner Node, в котором производятся некие вычисления и результат публикуется в Response Topic.

MediatorNode, выполняющий роль посредника между ФРУНДом и соответствующим модулем компьютерного зрения читает сообщения из Response Topic, куда публикуются сообщения с результатами расчета. По приходу нового сообщения с результатом он пытается перехватить мьютекс, который синхронизирует доступ к буферу, и записать в буфер данные.

Параллельно Mediator Node слушает сокеты, которые связаны с ФРУНДом. По приходе сообщения из сокетов, Mediator Node пытается захватить мьютекс буфера и отправить данные из него ФРУНДу.

Таким образом, ФРУНД не обязан ждать завершения вычислений. Он гарантированно получает данные лишь с задержкой на их транспортировку и получение из буфера.

В качестве улучшения можно перенести часть блокировки на сторону ФРУНДа, чтобы исключить задержки на передачу по сети.

Помимо этого, такая синхронная структура позволяет модулям системы автономной навигации получать данные как от ФРУНДа, так и от любых других источников (в топики request Topic может публиковать какая угодно программа). Это используется, чтобы, например, задавать цель при планировании из визуализатора rviz или из консольных утилит ROS'a (например **rostopic pub**).

Результат планирования при этом также будет доступен ФРУНДу.

Согласование систем координат ФРУНДа и карты от rtabmap

Системы координат ФРУНДа и rtabmap никак не связаны изначально.

Системы координат в ROS

В ROS системы координат реализованы в пакете tf <http://wiki.ros.org/tf>.

Этот пакет предоставляет средства для работы со многими связанными между собой системами координат. Связи между системами координат представляются в виде древовидной структуры.

У каждой из систем есть предок, на которого она ссылается, а также собственные позиция и ориентация относительно родительской системы координат.

Общепринятой практикой является публикация дерева трансформаций для робота, состоящего из систем координат всех его конечностей и датчиков относительно его опорной системы координат (`base_link`), что используется в дальнейшем при планировании комплексных движений роботов с учетом траекторий всех его конечностей и плотных карт окружения (<http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF>).

На рисунке ниже указан пример дерева трансформаций на основе робота PR2 относительно некой глобальной системы координат (вероятно `odom` или `base_link`).

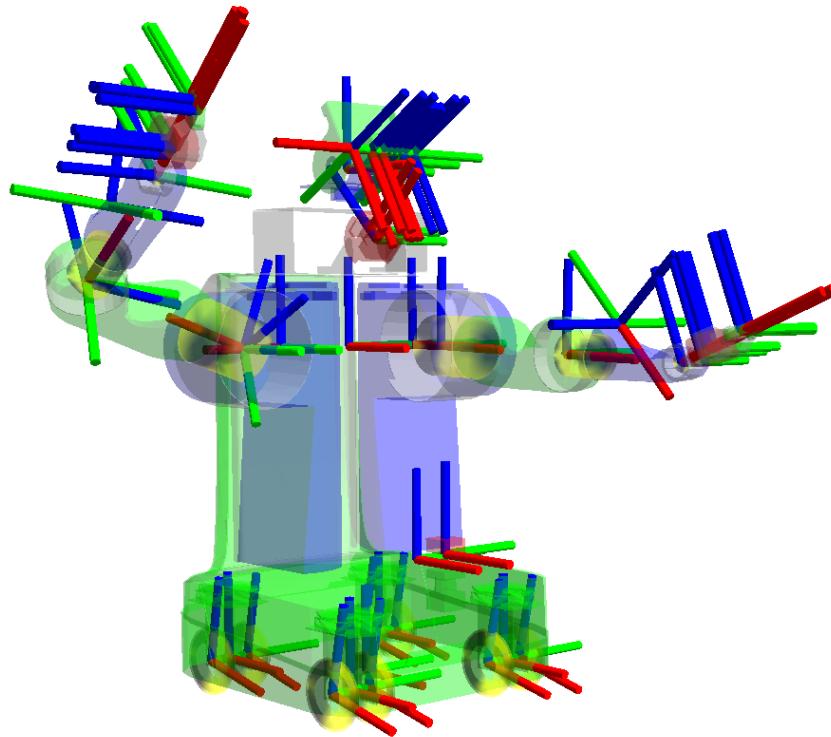


Рисунок 43 - Дерево трансформаций робота PR2

В ROS для мобильных платформ приняты следующие соглашения для систем координат

(<http://www.ros.org/reps/rep-0105.html> ):

### **base\_link**

Эта система координат жестко связана с роботом / мобильной платформой. Относительно нее задаются системы координат конечностей робота, его датчиков (например Kinect'a) и т.д.

### **odom**

Это глобальная система координат. Позиция робота в ней гарантированно изменяется плавно, а не скачками. Однако, позиция робота в этой системе координат может смещаться со временем от «корректной» позиции, причем без каких – ибо ограничений.

Этот момент делает данную систему координат полезной в качестве кратковременного ориентира, но мало полезной для долговременного использования в качестве ориентира.

### **map**

Это еще одна система координат. Позиция робота в ней не смещается значительно с течением времени, как при использовании в качестве опорной с.к. **odom**. Но в то же время позиция робота в этой системе координат может меняться дискретными скачками в любое время.

Такое часто случается, когда происходит замыкание круга в некоторых SLAM алгоритмах. Происходит корректировка положения робота, что приводит к скачкообразному изменению позиции робота в этой системе координат. Если быть точнее, при таких коррекциях дискретно меняется связь между системами координат **map** и **odom** и, как следствие, положение **base\_link**, которая находится «внутри» с.к. **odom**

На изображении ниже для наглядности приведено дерево трансформаций для импровизированного колесного робота с Lidar'ом.

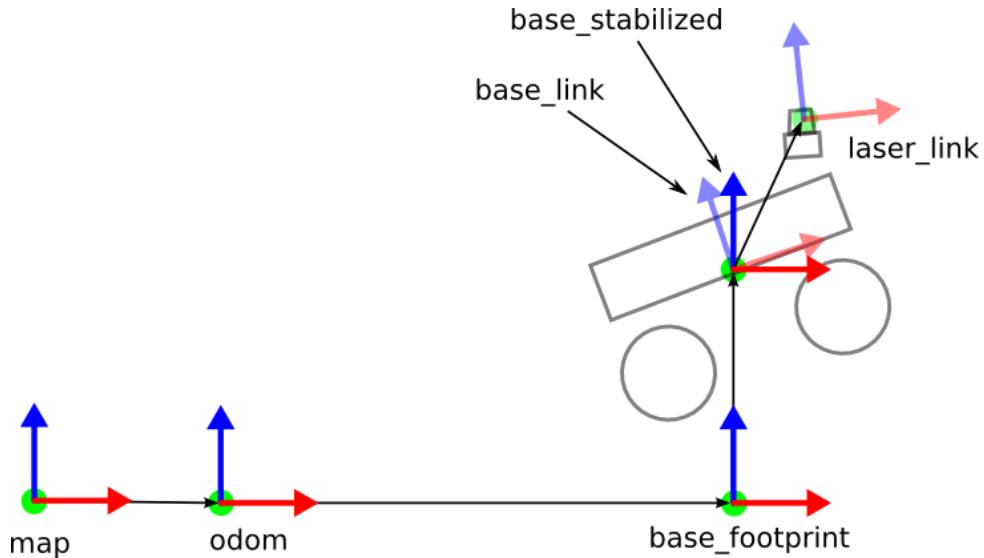


Рисунок 44 - Взаимосвязь систем координат в пакете *hector\_slam*

Системы координат в нашем случае

Так как AR600E не взаимодействует напрямую с платформой ROS и ФРУНД не посылает ей никакой информации о роботе, положении его конечностей и его самого, то системы координат камеры (**camera\_link**, аналог которой на изображении выше – **laser\_link**) и робота (**base\_link**) совмещены между собой.

При этом появляется проблема совмещения систем координат ФРУНД'а, который посылает запросы исходя из своей собственной системы координат и системы координат, в которой работает SLAM алгоритм, предоставляя облако точек и местоположение.

Я нашел 2 способа состыковать систему координат SLAM алгоритма из библиотеки rtabmap с системой координат ФРУНДа.

*Добавление дополнительного смещения в цепочку трансформаций base\_link -> camera\_link*

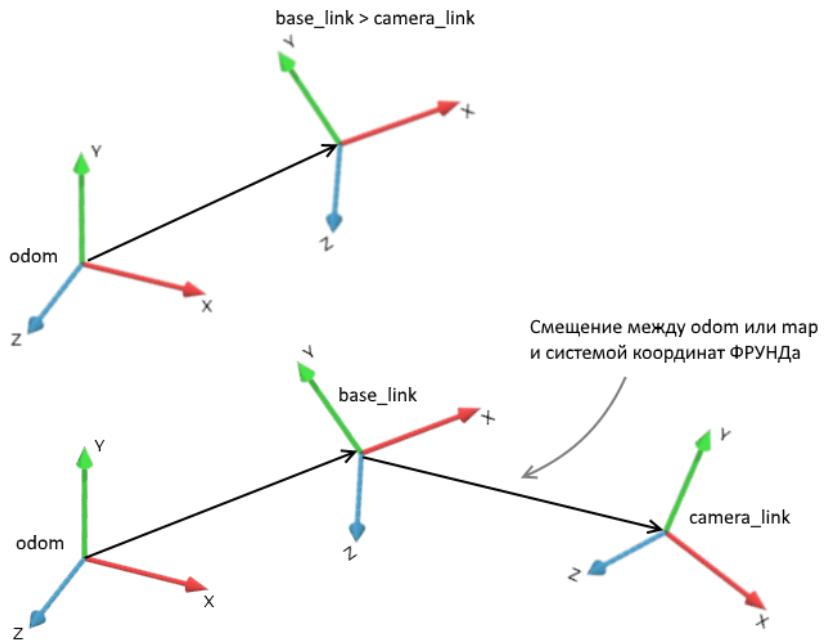


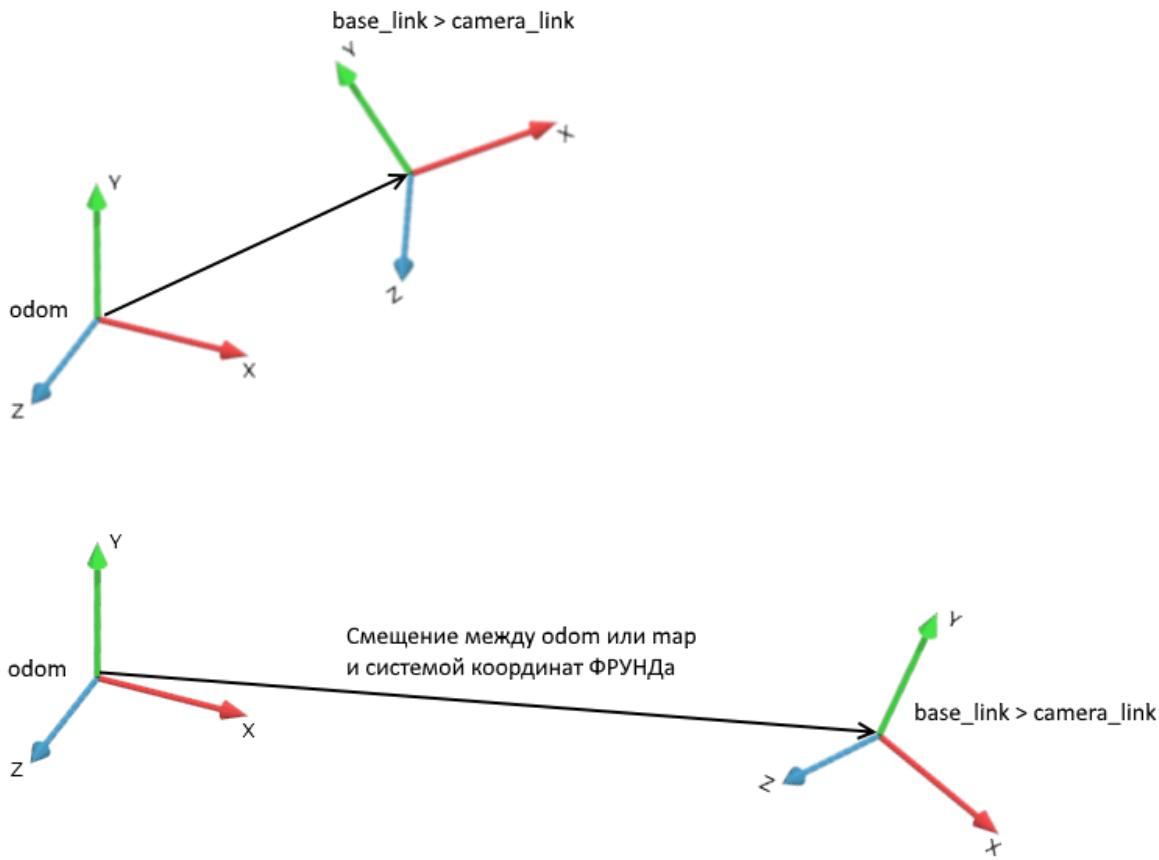
Рисунок 45 - Иллюстрация дерева трансформаций без с со смещением

Для начала найдем разницу между системами координат ФРУНДа и SLAM алгоритма.

Затем, добавим в дерево трансформаций между **base\_link** и **camera\_link** дополнительную трансформацию, равную той самой разнице, найденной выше.

Таким образом, камере будет казаться, что она находится в системе координат ФРУНДа.

*Задание начального положения камеры для Node Visual Odometry*



*Рисунок 46 - Задание начального положения через ~initial\_pose*

Роль определения местоположения в пакете rtabmap выполняет Node Visual Odometry и ее частные реализации (Stereo Odometry, RGBD Odometry).

В параметрах Node семейства Visual Odometry есть скрытый параметр `~initial_pose`.

Если мы возьмем координаты и ориентацию системы координат камеры внутри системы координат ФРУНДа, то сможем использовать их в качестве начального положения камеры.

Т.е. камера в самом начале будет думать, что находится в С.К. ФРУНДа в том положении, в котором она находится во ФРУНДе.

В приложении Б приведены скрипты, реализующие оба способа

## 6. Разработка подсистемы планирования траектории движения антропоморфного робота AR600E

Одной из моих главных задач была разработка подсистемы планирования траектории движения робота.

Основываясь на работах, описанных ранее и приняв в учет 2 возможных реализации архитектуры системы, я разработал два планера, один из которых рассчитывает маршрут в виде набора положений левых и правых ступней для робота, а другой – в виде набора точек геометрического центра робота.

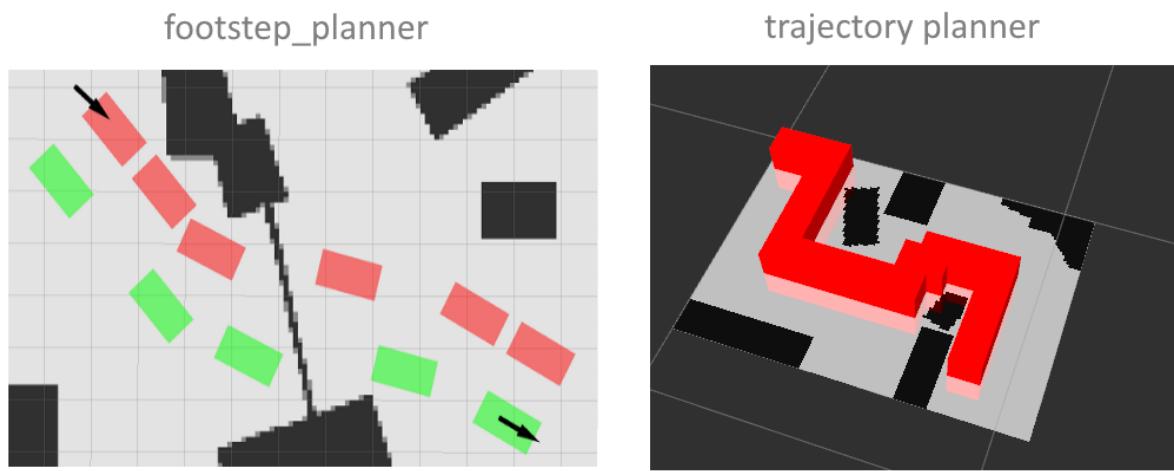


Рисунок 47 - Маршруты, спланированные каждым из планировщиков

### Разработка модуля передачи данных для планировщиков

Интерфейс обоих планировщиков выглядит практически одинаково. Различаются лишь возвращаемые типы данных и некоторые из параметров.

Опишем общие входные данные в терминах топиков, типов их сообщений и способ их получения и передачи планировщикам, который я разработал.

Общие входные топики

*geometry\_msgs/PoseStamped goal*

Это топик, в который публикуется положение и ориентация позиции, к которой требуется проложить маршрут.

**geometry\_msgs/PoseStamped** – тип топика (т.е. тип сообщений, которые он принимает), а **goal** – его наименование, зная которое можно считывать и публиковать в него сообщения соответствующего типа.

На изображении ниже приведена структура родственного к **geometry\_msgs/PoseStamped** типа сообщения – **geometry\_msgs/Pose**. Различие между ними в наличии дополнительной временной информации в заголовке сообщения.

## [geometry\\_msgs/Pose Message](#)

---

File: [geometry\\_msgs/Pose.msg](#)

### Raw Message Definition

```
# A representation of pose in free space, composed of position and orientation.  
Point position  
Quaternion orientation
```

### Compact Message Definition

```
geometry_msgs/Point position  
geometry_msgs/Quaternion orientation
```

Рисунок 48 - Формат сообщения *geometry\_msgs/Pose*

### [nav\\_msgs/OccupancyGrid\\_map](#)

Данное сообщение содержит карту препятствий, на которой и производится планирование. Сообщение содержит информацию о начале системы координат, величину собственной дискретизации (т.к. карта препятствий – дискретная карта), геометрические размеры, а также данные в виде одномерного массива.

## [nav\\_msgs/OccupancyGrid Message](#)

File: [nav\\_msgs/OccupancyGrid.msg](#)

### Raw Message Definition

```
# This represents a 2-D grid map, in which each cell represents the probability of
# occupancy.

Header header

#MetaData for the map
MapMetaData info

# The map data, in row-major order, starting with (0,0). Occupancy
# probabilities are in the range [0,100]. Unknown is -1.
int8[] data
```

### Compact Message Definition

```
std_msgs/Header header
nav_msgs/MapMetaData info
int8[] data
```

Рисунок 49 - Формат сообщения nav\_msgs/OccupancyGrid

[geometry\\_msgs/PoseWithCovarianceStamped initial\\_pose](#)

тип сообщения близок к рассмотренному ранее **geometry\_msgs/PoseStamped**, но с дополнительной информацией.

В этот топик публикуется стартовое положение для планировщика.

Общие выходные топики

[nav\\_msgs/Path path](#)

В данный топик публикуются сообщения, содержащие информацию о рассчитанном планировщиком маршруте.

Маршрут – набор позиций, которые представляют из себя вектор координат положения и ориентацию, заданную кватернионом (см. выше в описании входных топиков).

## nav\_msgs/Path Message

---

File: nav\_msgs/Path.msg

### Raw Message Definition

```
#An array of poses that represents a Path for a robot to follow
Header header
geometry_msgs/PoseStamped[] poses
```

### Compact Message Definition

```
std_msgs/Header header
geometry_msgs/PoseStamped[] poses
```

Рисунок 50 - Формат сообщения nav\_msgs/Path

Передача данных планировщикам от SLAM алгоритма

**goal**, т.е. цель, к которой будет планироваться маршрут, может задаваться на основании запроса, полученного от ФРУНДа или от визуализатора rviz'а, в котором можно задавать goal и initial\_pose вручную при помощи удобного инструмента.

**Initial\_pose**, т.е. начальную позицию для планирования можно получить из текущего местоположения камеры, которое известно SLAM алгоритму.

**map**, т.е. карту препятствий можно также получить у SLAM алгоритма. Но здесь встает вот какая дилемма. Каждый раз, когда в топик с картой, на который подписан планер, приходит карта, вызывается callback, в котором происходит перезапуск планировщика. Частота, с которой осуществляется процесс публикации новых сообщений с картой, довольно высока.

Это можно обойти, например, ограничив величину очереди приема сообщений на планировщике. Но по каким – то причинам, это вызывало отказ и аварийное завершение нод из пакета `footstep_planner`, которые я использовал в одном из вариантов реализации модуля планирования маршрута.

В дальнейшем я нашел и устранил ошибку, но это было уже гораздо позже написания данной программы.

Рассмотрим принцип работы вспомогательного модуля, которые занимается передачей недостающих данных планировщику.

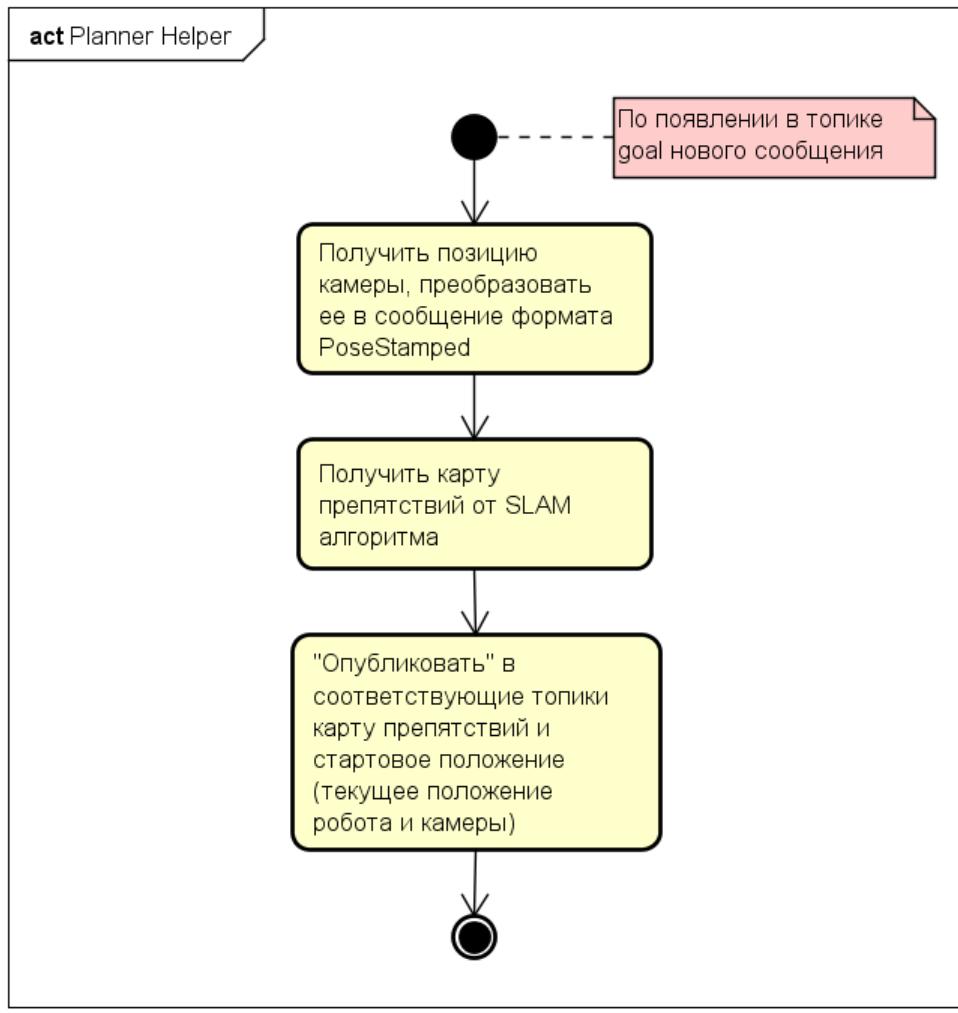


Рисунок 51 - Activity Diagram для вспомогательного модуля

Итак, эта программа «слушает» топик **goal**. Как только в нем появляется сообщение, т.е. откуда – то приходит новая цель, программа запрашивает координаты и ориентацию камеры от SLAM алгоритма. Таким образом она получает **pose**, который затем публикуется в соответствующий топик. Также она запрашивает карту препятствий на текущий момент времени, которую также публикует в топик.

Таким образом, эта программа по появлении нового **goal**, собирает и переправляет недостающую планеру информацию.

Код программы доступен в приложении В.

## Разработка планировщика маршрута на основе пакета `footstep_planner`

Об этом пакете уже было упомянуто ранее. Он позволяет планировать траекторию, исходя из набора положений ступней, которые должен повторить робот, чтобы попасть в желаемое место на карте. В данном случае применяется поиск в пространстве состояний. Где состояние – положение одной из ступней. Каждое состояние может породить ряд других, используя дискретное множество возможных шагов для другой ступни. ([http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12\\_Slides\\_Hornung.pdf](http://www.ais.uni-bonn.de/humanoidsoccer/ws12/slides/HSR12_Slides_Hornung.pdf))

Данная задача очень напоминает задачу поиска маршрута на графах. В данном случае конкретное состояние – вершина графа. А ребро – совершающее действие.

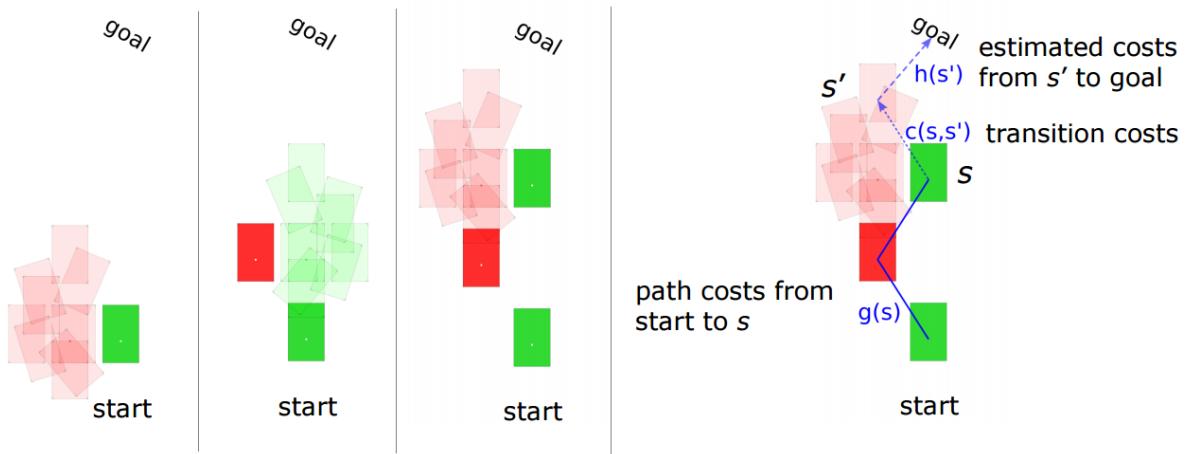


Рисунок 52 - Способ планирования, который применяется в `footstep_planner`

Возможные шаги для робота задаются довольно просто.

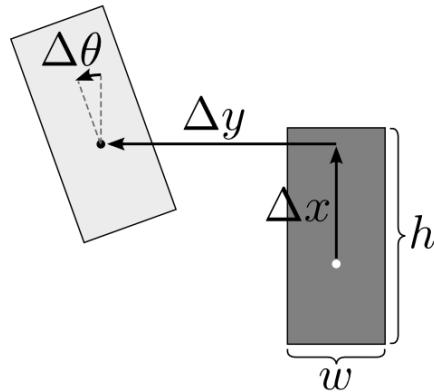


Рисунок 53 - Иллюстрация способа задания шага в пакете `footstep_planner`

Вместе со вспомогательной программой передачи данных, о которой я упомянул выше, мне удалось заставить работать этот пакет в реальном времени.

Правда перед этим пришлось отладить эту ноду и исправить в ней ошибку, появлявшуюся в момент смены карты. Ошибка заключалась в некорректном высвобождении памяти (была

проблема с индексами). В результате при смене карты с больше на меньшую нода завершалась с ошибкой.

The screenshot shows a GitHub pull request page for the repository 'ahornung / humanoid\_navigation'. The pull request is titled 'Fix footstep\_planner node crashes when map changes #17' and is marked as 'Merged'. It was merged by 'ahornung' from the branch 'AR600Vision:fix\_changing\_map\_fails' on March 7. The commit message indicates that memory deallocation was incorrect, leading to crashes. Some screenshots of debugging are included. The pull request has 7 additions and 4 deletions. On the right side, there are sections for Reviewers (No reviews), Assignees (No one assigned), Labels (None yet), Projects (None yet), and Milestone (No milestone).

Рисунок 54 - Принятые автором пакета правки

Таким образом этот способ при условии получения данных от вспомогательной программы работает более – менее стабильно.

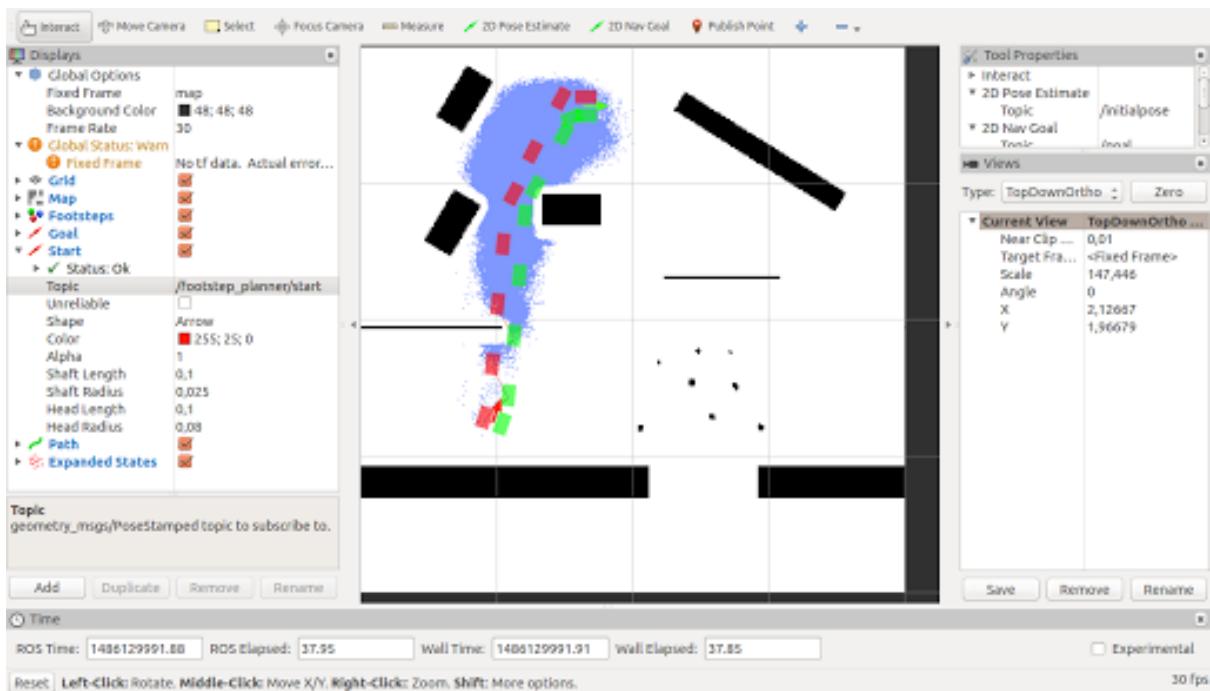


Рисунок 55 - Визуализация результатов планирования в rviz

Я проводил эксперимент после исправления ошибки, в котором я подписал ноду планировщика напрямую к карте, публикуемой SLAM алгоритмом. Это привело к аварийному завершению через несколько циклов планирования.

Также было заметно, что планировщик не успевает за частотой обновления карты.

## Разработка планировщика траектории

### Описание идеи

Для реализации я выбрал один из двух поддерживаемых ROS языков – Python 2.7. Причина этого лежит в более простом учете зависимостей, а также в более простой работе со ссылками.

Перед этим я потратил некоторое время на попытку данного модуля на языке C++, что частично удалось. Но последующая реализация на Python заняла в разы меньше времени и позволила получить более качественный и компактный код.

Данный планировщик я реализовал, основываясь на идее планирования в пространстве состояний исходя из дискретного количества возможных движений, которые может совершать робот, которые в свою очередь порождают новые состояния.

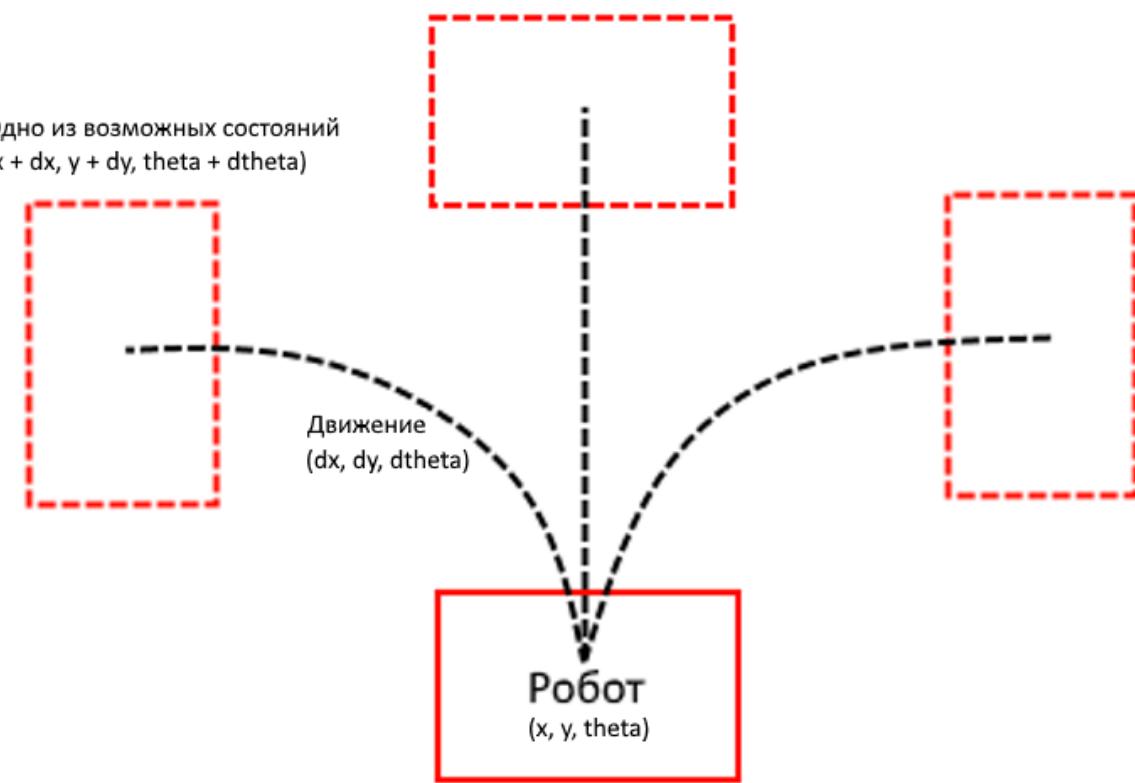


Рисунок 56 - Выбор нового состояния путем совершения одного из возможных движений

Робот в данном случае представляется в виде геометрического примитива (прямоугольника).

Результатом планирования является набор точек, по которым должен перемещаться центр этого прямоугольника, чтобы достичь требуемого положения (**goal**).

Движения, из учета которых планируется маршрут робота представляют из себя дельты координат и ориентации робота, которые показывают, как изменится положение робота при совершении данного движения.

Дельты могут быть большими, за счет чего можно проводить планирование из таких движений, как «пройти 5 метров», «пройти 3 метра», «повернуть на 90 градусов, пройдя 1 метр» и т.д.

Весь маршрут, который проходит робот при выполнении действия проверяется на коллизии по карте препятствий.

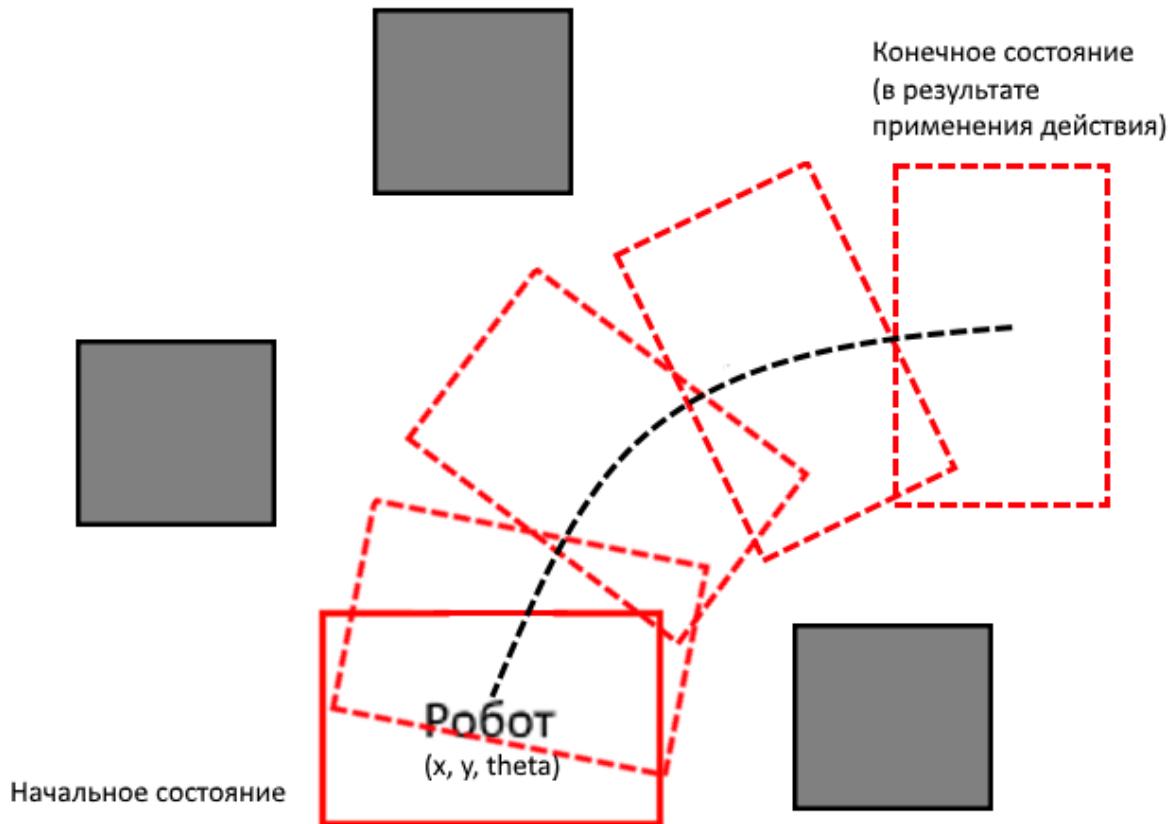


Рисунок 57 - Иллюстрация расчета перемещений между состояниями

Если немного модифицировать алгоритм и в результате сохранять информацию не только о том, в каких состояниях должен побывать робот, чтобы прийти к цели, но и какие движения он делал между ними, можно реализовать следующий вариант управления роботом.

У планировщика есть набор движений, которые может сделать робот. У робота же заготовлен этот набор движений, заранее сгенерированный ФРУНДом и записанный в файлы.

Планер возвращает набор движений, которые должен пройти робот. А робот последовательно выполняет движения, рассчитанные заранее в порядке, полученном от планировщика.

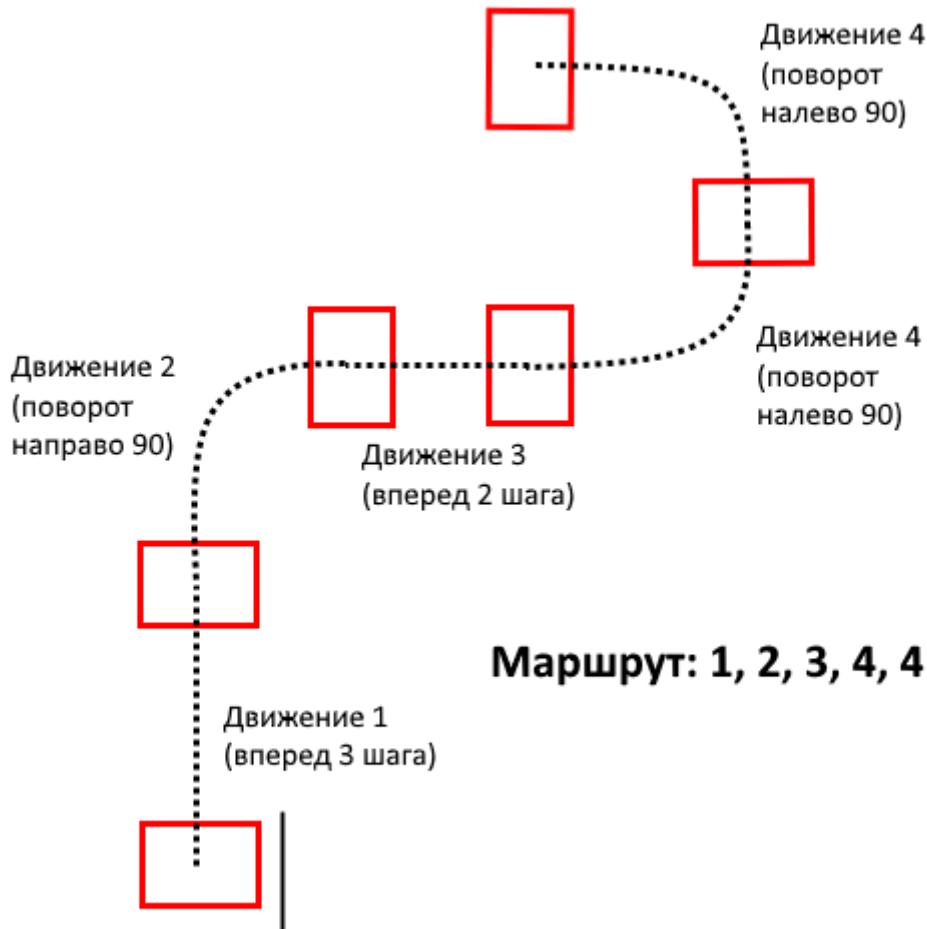
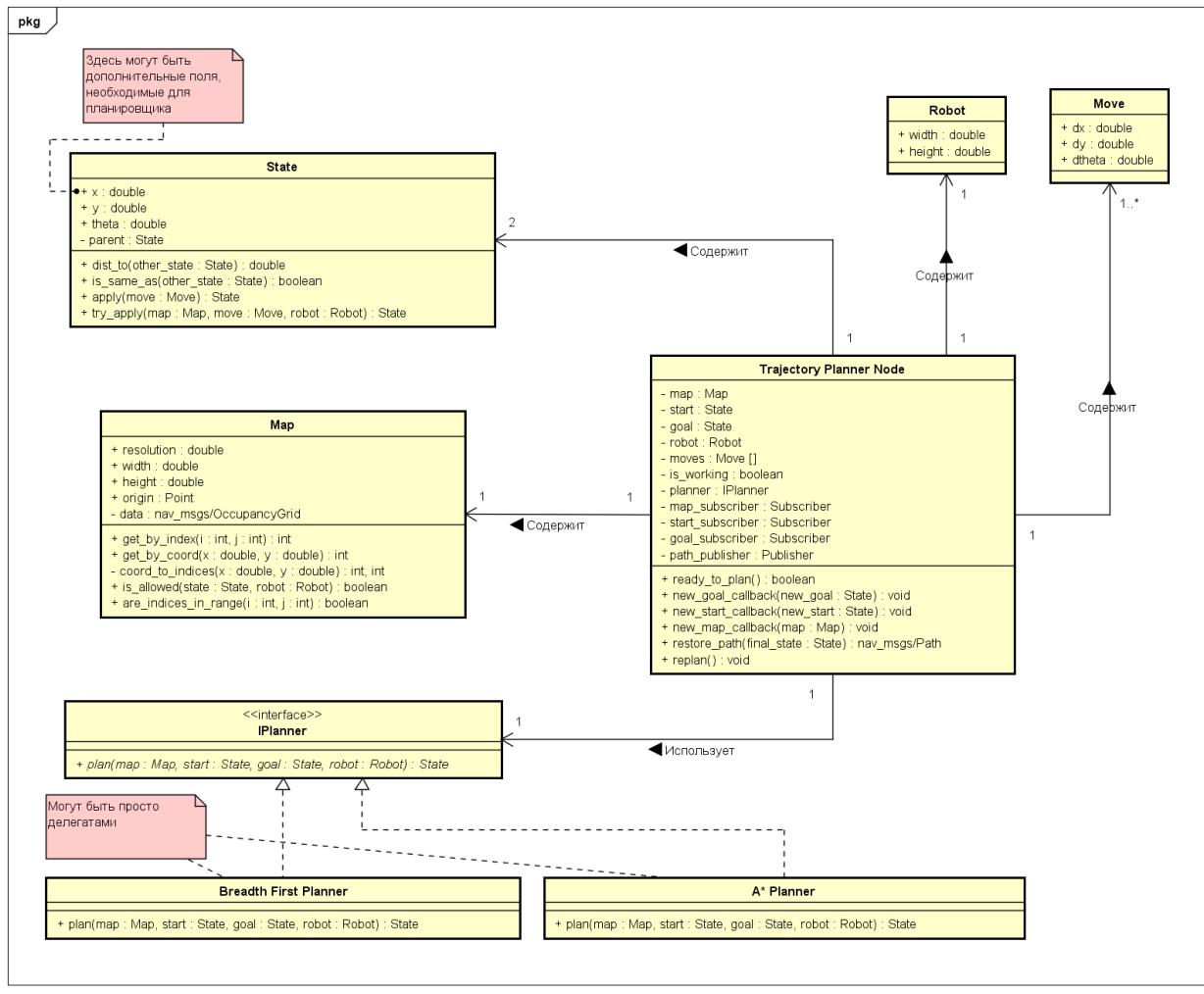


Рисунок 58 - Один из вариантов применения планировщика

Таким образом, мы можем получить довольно примитивный вариант, который тем не менее позволит роботу автономно ориентироваться и перемещаться в пространстве.

## Описание архитектуры



powered by Astah

Рисунок 59 - Диаграмма классов системы

Примечание: диаграмма продублирована в альбомной ориентации в приложении Г.

### State

Состояние робота. Так как робот – прямоугольник, то для того, чтобы охарактеризовать его состояние, т.е. его положение, достаточно координат его центра и угла поворота относительно положительного направления оси X.

Помимо этого, каждое состояние хранит ссылку на родительское состояние, из которого оно было получено применением одного из возможных движений. Начальное состояние не ссылается ни на какое другое.

Это необходимо для того, чтобы в конце планирования была возможность восстановить маршрут, имея непосредственно лишь конечное состояние.

**dist\_to(other\_state)** – расстояние до другого состояния.

**is\_same\_as(other\_state)** – являются ли два расстояния одинаковыми с точки зрения расстояния друг между другом и разницей между углами их поворотов. Используется при планировании, чтобы проверить, были ли мы в подобном состоянии или нет.

**apply(move)** – применить действие и получить новое состояние, ссылающееся на текущее

**try\_apply(map, robot, move)** – попробовать применить действие на данное состояние. Сюда входит проверка всех промежуточных состояний, которые могут возникнуть при выполнении некоего «большого» действия.

Например при действии «пройти 6 метров» будут проверены все 6 метров пути (с использованием вспомогательных позиций).

#### *Robot*

Дабы не хранить в каждом из состояний информацию о роботе, которая задается лишь на старте программы, была создана данная структура.

Она содержит информацию о геометрических размерах робота.

#### *Move*

Движение, которое позволяет перейти из одного состояния в другое.

Характеризуется дельтами следующих компонент (**x, y, theta**). В связи с этим может описывать такие движения, как поворот и прямолинейное движение вдоль какого – либо направления.

#### *Map*

Надстройка над сообщением **nav\_msgs/OccupancyGrid**, которое не очень удобно при использовании в сыром виде.

Часть значений, характеризующих карту, таких как ширина, высота, разрешение и координаты центра начальной точки (0, 0) вынесены в поля Мар.

Также написаны некоторые вспомогательные методы, позволяющие обращаться к значению ячеек как по индексам, так и по координатам.

**get\_by\_index(i,j)** – получить значение ячейки по индексам.

**get\_by\_coord(x, y)** – получить значение ячейки по координатам. Ниже приведен способ преобразования координат в индексы.

```
i = int((y - origin.y) / resolution)
j = int((x - origin.x) / resolution)
```

**is\_allowed(state, robot)** – определить, возможно ли данное состояние для указанных геометрических размеров робота. При проверку учитывается возможность выхода за границы карты и наличие препятствий на площади, занимаемой роботом.

#### *Planner*

Осуществляет планирование траектории. Выделен в отдельный объект для использования возможностей полиморфизма в Trajectory Planner Node.

Т.е. при любом количестве планеров достаточно будет заполнить объектом соответствующего класса поле `planner`. Также здесь можно использовать делегатов.

На данный момент я реализовал два планировщика – на основе алгоритмов A\* и поиска в ширину.

#### *Trajectory Planner Node*

Объект этого можно отождествлять с нодой. Он инкапсулирует в себе все данные., которые нужны для расчёта, такие как:

- Map (карта)
- Start (начальное состояние. Объект класса State)
- Goal (конечное состояние. Объект класса State)
- Moves [] (набор возможных движений)
- Robot (объект робота)

Помимо этого, он занимается приемом и проверкой этих данных на валидность. А также запускает планирование, когда пришла новая стартовая или конечная позиция, или карта в случае наличия всех их.

`ready_to_plan()` – возвращает информацию о наличии валидных карты, начального и конечного положений.

```
new_goal_callback(new_goal),  
new_start_callback(new_start),  
new_map_callback(new_map)
```

Проверка и установка новых значений в случае валидности. При получении положительного результата от `ready_to_plan()` вызывает перепланирование траектории.

`restore_path(final_state)` – восстанавливает полную траекторию по последнему состояние в нем.

`replan()` – перепланирование траектории. Вызывает выбранный планировщик, а затем используя метод `restore_path(final_state)` получает траекторию и публикует ее в топик `path`.

Описание алгоритмов

#### *Служебные алгоритмы*

#### *Проверка валидности и установка новых значений*

Для того, чтобы «отфильтровать» некорректный или преждевременный ввод (если расчеты еще не закончились) применяется следующий нехитрый подход.

Перед установкой новой цели проверяется завершены ли расчеты, есть ли карта препятствий и допускается ли на ней установка данной цели. Только в этом случае новая цель будет установлена.

Далее проверяется, все ли готово для планирования. Если стартовая позиция и карта «на месте», то осуществляется перепланирование маршрута.

Аналогичные сценарии осуществляются для карты и начальной позиции.

Ниже представлен код callback'а устанавливающего новое значение цели (goal).

```
def new_goal_callback(self, goal_pose):
    if not self.is_working:
        self.is_working = True
    new_goal = State.from_pose(goal_pose.pose)
    if self.map is not None and self.map.is_allowed(new_goal, self.robot):
        self.goal = new_goal
        rospy.loginfo("New goal was set")
    if self.ready_to_plan():
        self.replan()
    self.is_working = False
```

Для синхронизации callback'ов на данный момент используется переменная self.is\_working.

Проверка осуществимости движения

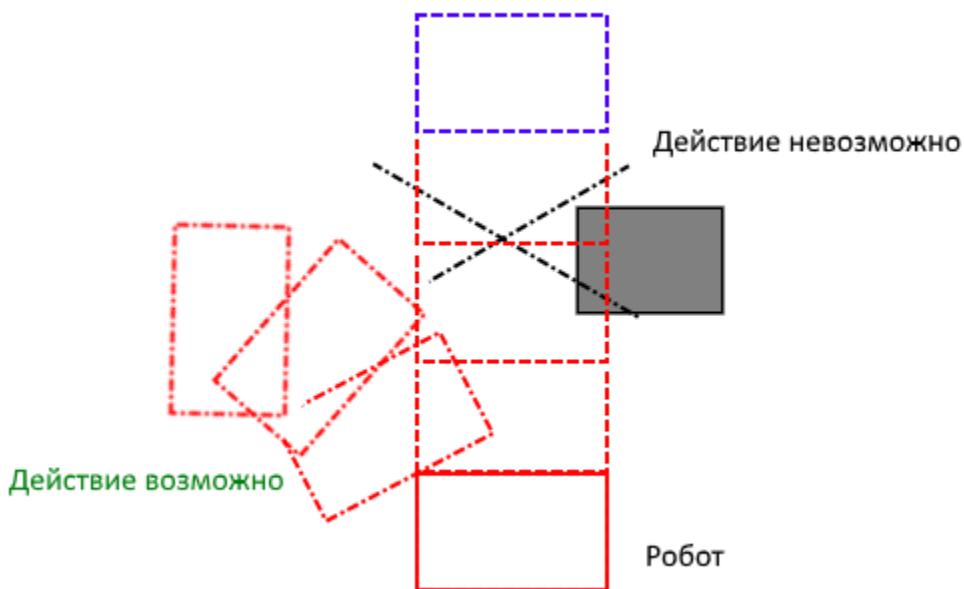


Рисунок 60 - Пример проверки осуществимости двух движений

Суть алгоритма в следующем: определяется количество шагов «симуляции» движения. Т.е. все движение разбивается на равномерные «отрезки» так, чтобы между подсостояниями, образующихся в процессе моделирования не оставалось «слепых пятен».

Ниже приведен код метода try\_apply, который осуществляет проверку осуществимости движения и порождает новое состояние в случае успеха оной проверки.

```
def try_apply(self, map, move, robot):
    model_state = copy.copy(self)
    model_state.parent = self

    goal = self.apply(move)
    goal.parent = self
```

```

steps_count = max(int(self.dist_to(goal) / min(robot.height, robot.width)), 1)
step = 1.0 / steps_count

for i in range(steps_count):
    model_state.x += move.dx * step
    model_state.y += move.dy * step
    model_state.theta += move.dtheta * step

    if not map.is_allowed(model_state, robot):
        return None
return goal

```

Проверка коллизий с окружением

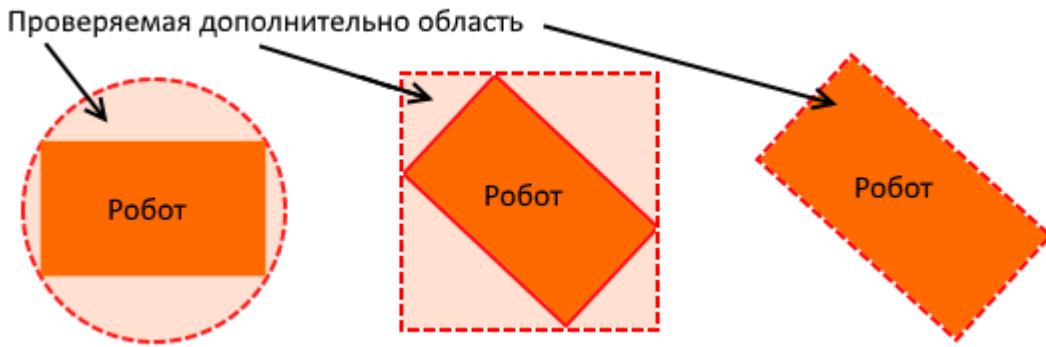


Рисунок 61 - Варианты проверки состояния на коллизии с препятствиями в окружении

Я выделил три способа осуществлять подобную проверку.

Первый – это проверять окружность с центром, совпадающим с центром геометрического примитива робота и диаметром, равным большей из его сторон. Это довольно простой в реализации способ, который в качестве бонуса позволяет исключать препятствия неподалеку от состояния.

Второй заключается в проверке квадрата со стороной, равной наибольшей из сторон примитива робота и с центром, равным центру примитива робота. Является наиболее простым из трех и реализован на данный момент.

Третий, заключается в проверке пространстве, занимаемого самим роботом и является более сложным, нежели оба варианта рассмотренных выше. Для реализации этого подхода можно узнать координаты углов прямоугольника робота через пару операций с векторами. А затем применить алгоритм «закрашивания» двух треугольников, хорошо разобранный в данной статье (<https://habrahabr.ru/post/248159/> ).

Третий подход использовался при написании небольшой программы по обнаружению препятствий по курсу движения робота.

*Алгоритмы расчёта*

Непосредственно планированием занимается объект класса **Planner**.

**Trajectory Planner Node** передает ему всю необходимую информацию, а в конце выполнения алгоритма восстанавливает траекторию из последнего состояния в спланированной траектории и, отправляет ее в топик **path**, сконвертировав ее в сообщение **visualization\_msgs/MarkerArray** или **nav\_msgs/Path**.

Так как каждый объект **State** указывает на своего предка, это не представляет особой сложности. Этот процесс заключается в том, что мы проходим по указателям до тех пора, пока не наткнемся на начальную позицию (у нее указатель будет равен **null**).

На данный момент мне удалось реализовать алгоритмы **A\*** с простой эвристикой и поиск в ширину. Оба алгоритма показывают отличную скорость работы.

#### A\*

Мне понадобилось некоторое время, чтобы найти достаточно понятные пояснения к алгоритму. В реализации мне очень помогла одна из публикаций Rajiv Eranki (<http://web.mit.edu/eranki/www/tutorials/search/> ).

Ниже представлен псевдокод A\*, предложенный автором.

```
struct node {
    node *parent;
    int x, y;
    float f, g, h;
}

// A*
initialize the open list
1: initialize the closed list
2: put the starting node on the open list (you can leave its f at zero)
3:
-   while the open list is not empty
4:       find the node with the least f on the open list, call it "q"
5:       pop q off the open list
6:       generate q's 8 successors and set their parents to q
7:       for each successor
8:           if successor is the goal, stop the search
9:           successor.g = q.g + distance between successor and q
10:          successor.h = distance from goal to successor
11:          successor.f = successor.g + successor.h
12:
-           if a node with the same position as successor is in the OPEN list \
13:               which has a lower f than successor, skip this successor
-           if a node with the same position as successor is in the CLOSED list \
14:               which has a lower f than successor, skip this successor
-           otherwise, add the node to the open list
15:       end
16:       push q on the closed list
17:   end
18:
```

## 7. Анализ результатов и экспериментов

### Качественная оценка точности SLAM алгоритма RTABMAP

У SLAM алгоритма из пакета rtabmap\_ros есть очень много параметров, регулируя которые можно повысить точность / скорость выполнения алгоритма и приспособить его к какому – то конкретному типу окружения. ([http://wiki.ros.org/rtabmap\\_ros/Tutorials/Advanced%20Parameter%20Tuning](http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning) ).

Используя стандартные настройки для работы Kinect'a (из launch файла rgbd\_mapping.launch) удавалось получать вполне качественные результаты. Помимо этого местоположение камеры нам удалось определять довольно точно (с точностью до 2-5 см при переносе на 50 см в одну и другую стороны и обратно). Качество результатов сильно восприимчиво к резким и большим поворотам. Алгоритм в таком случае может потерять одометрию и с малой вероятностью сможет восстановиться, что может быть фатально при планировании маршрута в реальном времени.

Ниже приведен пример облака точек, полученного от SLAM алгоритма. Можно заметить несколько слоев точек, что мешает прямой обработке облака. Перед этим его следует отфильтровать или преобразовать в плотное облако точек, что является более надежным по сравнению с первым вариантом.

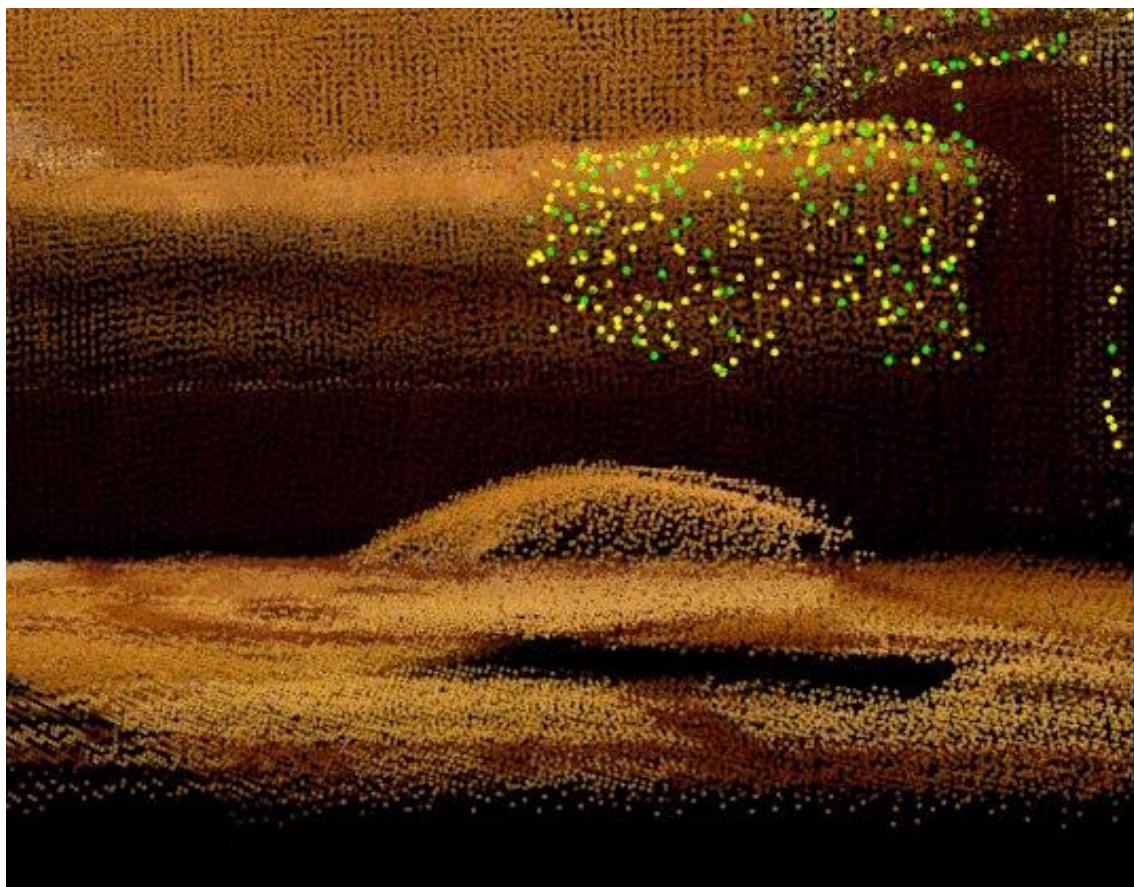


Рисунок 62 - Часть облака точек, создаваемого rtabmap при ближайшем рассмотрении

## Эксперименты с footstep\_planner

Время планирования в среднем составляет 7 секунд. Его можно существенно ускорить, изменив некоторые параметры (например указать алгоритму не улучшать первый полученный успешный маршрут, а возвращать ее в сыром виде или установить более простой способ проверки на коллизии с препятствиями и т.д.)

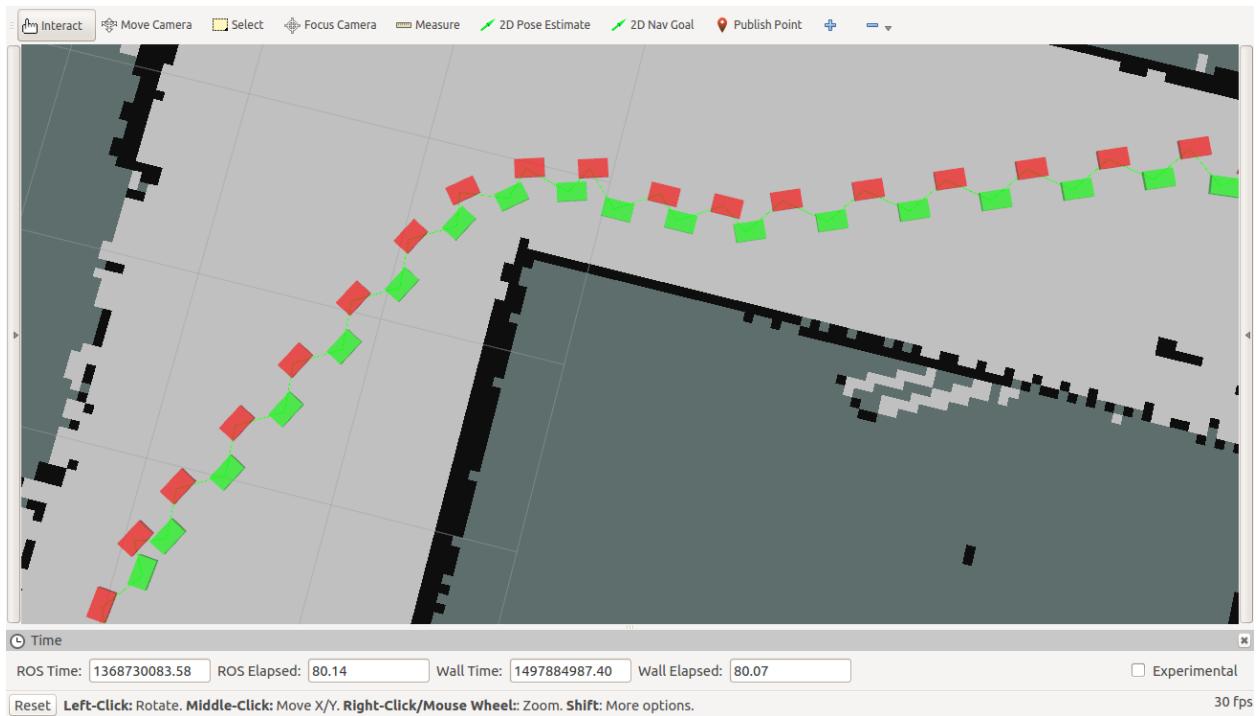


Рисунок 63 - Участок спланированной траектории вблизи

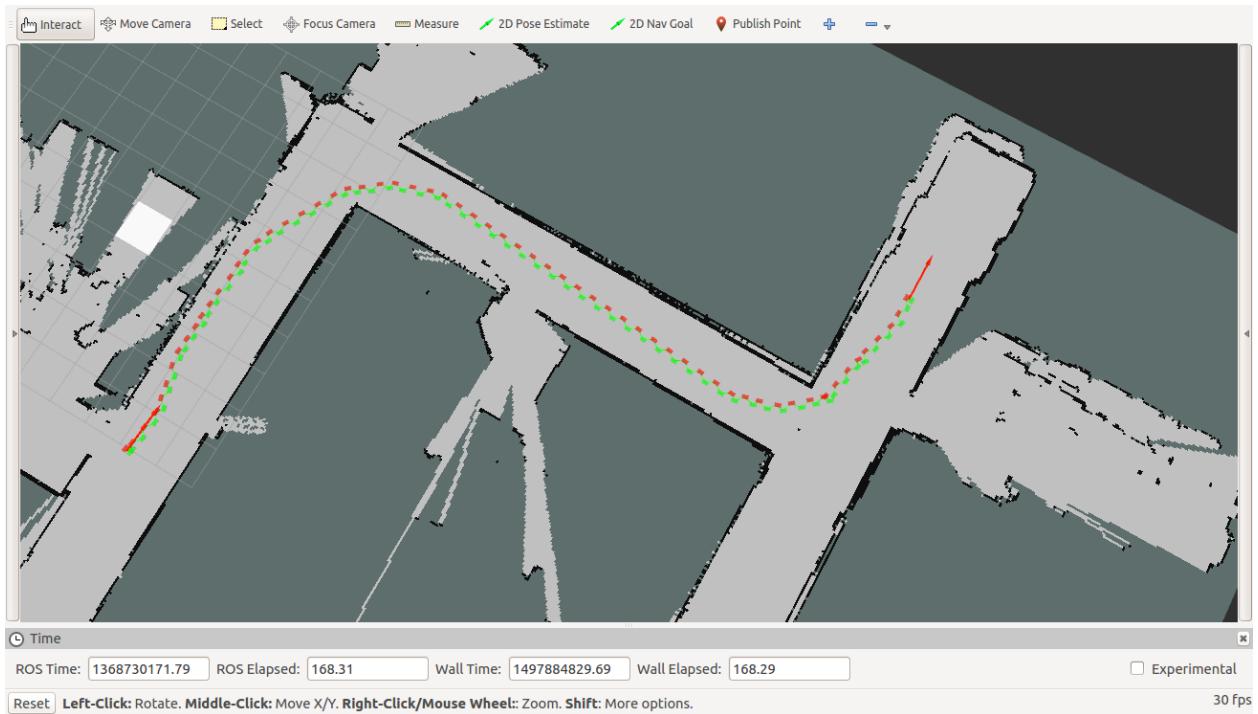


Рисунок 64 - Пример планирования на большой карте

### Эксперименты с trajectory\_planner

Во время экспериментов удалось выявить некоторые недочеты в алгоритме (например планировщик не хотел уходить от начального состояния слишком далеко). Изменив эвристику, удалось получить добиться более устойчивого построения траектории.

В алгоритме еще остается много мест, которые можно оптимизировать.

А именно: поиск ближайшего состояния на данный момент решается перебором. Хотя эта задача куда более эффективно решается при помощи методом разбиения пространства. Например при помощи 2-мерных или 3-мерных деревьев.

К возможным улучшениям относится также выбор более подходящих эвристик для алгоритма A\*.

Несмотря на свои недостатки метод работает довольно быстро. Например, планирование траектории, изображенной на рисунке 64 заняло меньше 1 секунды.

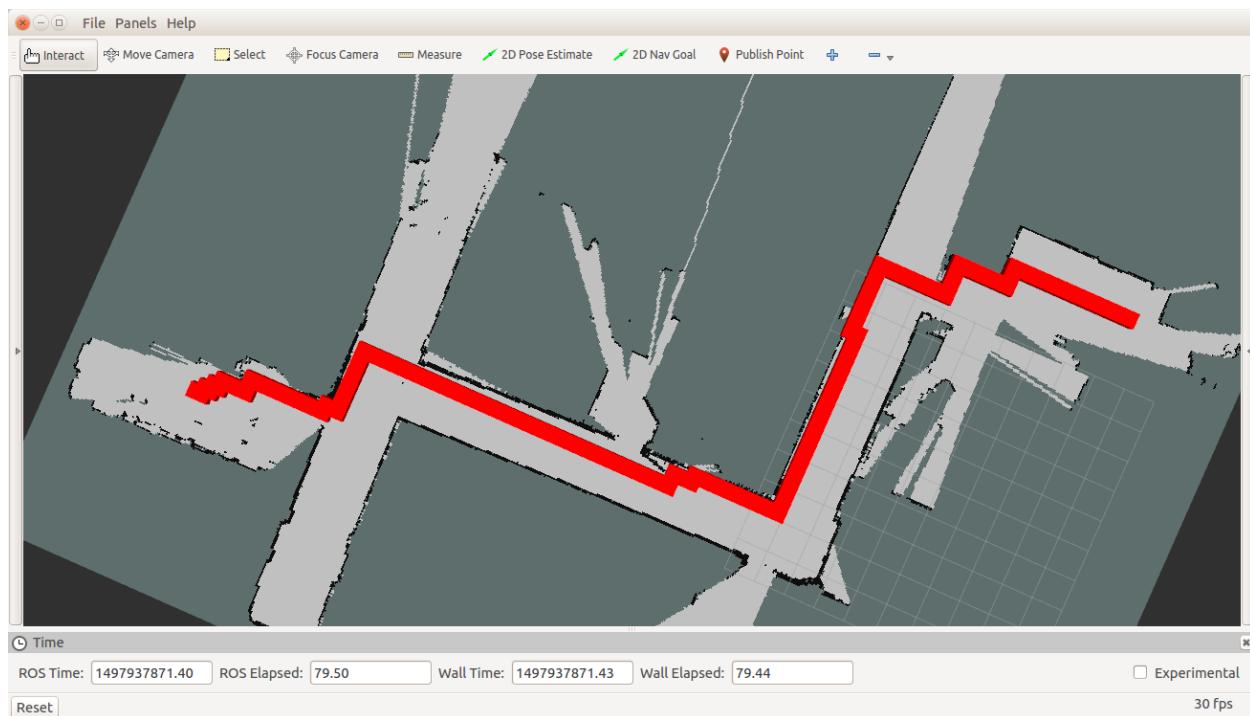


Рисунок 65 - Планирование trajectory\_planner на большое карте

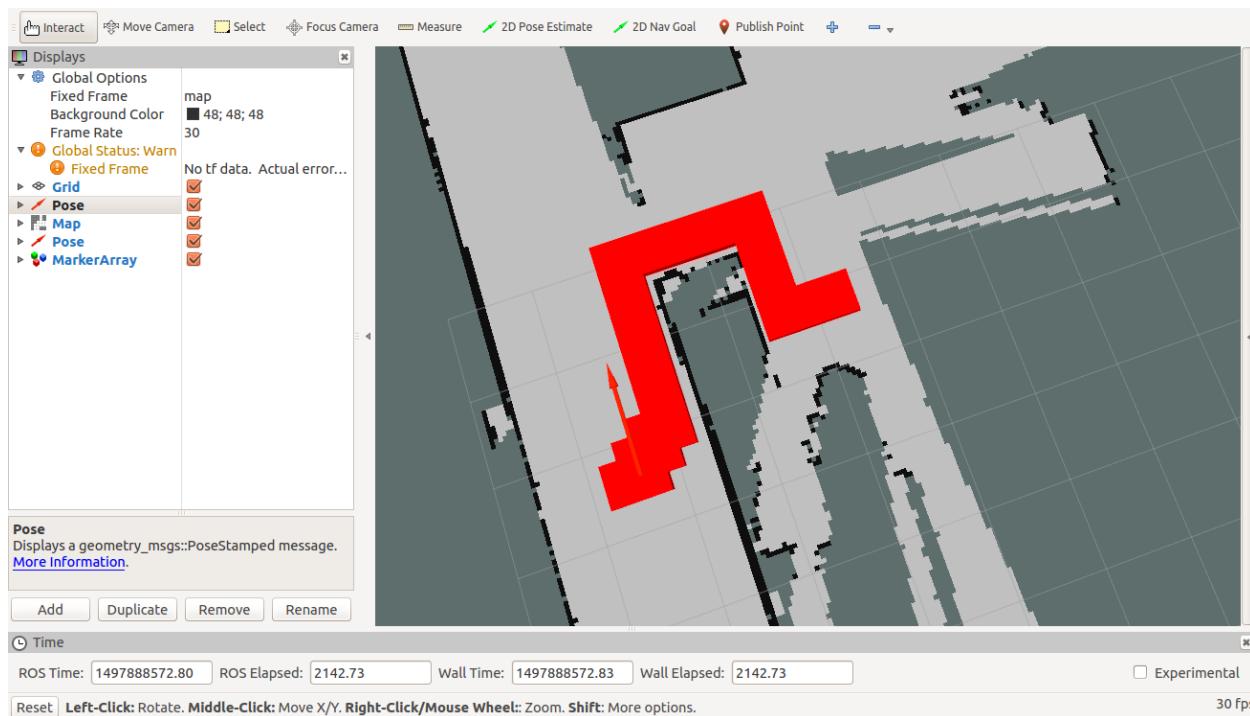


Рисунок 66 - Планирование на небольшом участке карты

## Дополнительные работы и эксперименты

В процессе работы, как побочный продукт была разработана программа определения наличия препятствий на пути движения робота. Информация о препятствиях передавалась системе управления роботом, которая предпринимала решение об остановке и переходе в стартовое состояние, о чем впоследствии сигнализировал робот при помощи системы речи.

Подробную информацию о реализации модуля обнаружения препятствий можно найти в работе Маркова Алексея.

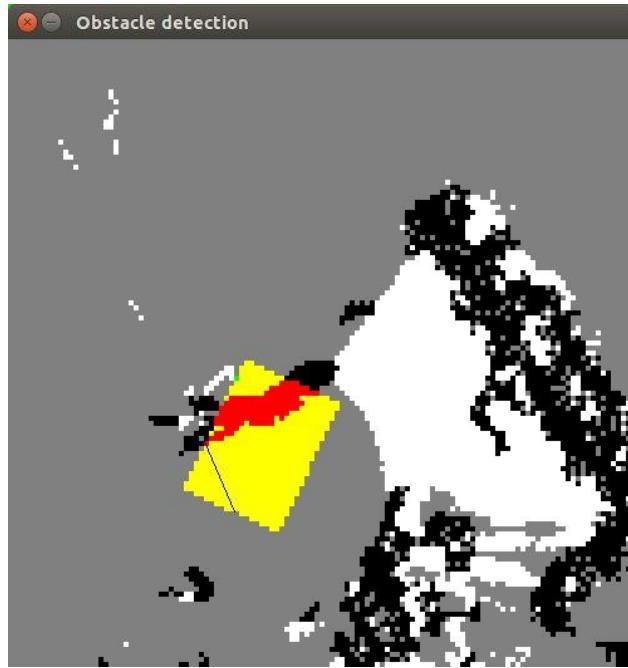


Рисунок 67 - Определение наличия препятствий по курсу движения робота

## Заключение

В данной работе был произведен исчерпывающий обзор подходов к реализации системы автономной навигации и планированию движений. Были рассмотрены основные библиотеки, которые помогают реализовать некоторые из этих подходов. Среди которых были найдены некоторые, связанные с планированием самих движений с учетом препятствий, что является одним из возможных дальнейших направлений работы.

Помимо этого, была разработана архитектура системы автономной навигации, которая при небольшой доработке коммуникации между системами, позволит роботу автономно перемещаться в различных окружениях.

Были реализованы все компоненты системы автономной навигации. А именно, подобран SLAM алгоритм, реализованы модули локального и глобального планирования а также реализована коммуникация между системой генерации движений робота и системой автономной навигации.

Также были разработаны 2 варианта подсистемы планирования маршрута для робота, которые позволяют получить маршрут в виде набора шагов или траекторию в виде набора точек. Оба варианта работают весьма стабильно и готовы к использованию.

В возможные дальнейшие направления работы входят нижеописанные пункты.

Доработка коммуникации между модулями системы и ФРУНДом, доработка модуля обработки полученной информации и учета ее при планировании на стороне ФРУНДа.

Доработка планировщика траектории и публикация его на Wiki ROS.

Эксперименты с планированием движений робота с использованием пакетов Move It и Drake. И работа над учетом окружения при планировании во ФРУНДе.

Приложение А. Характеристики Kinect  
(<https://msdn.microsoft.com/en-us/library/jj131033.aspx>)

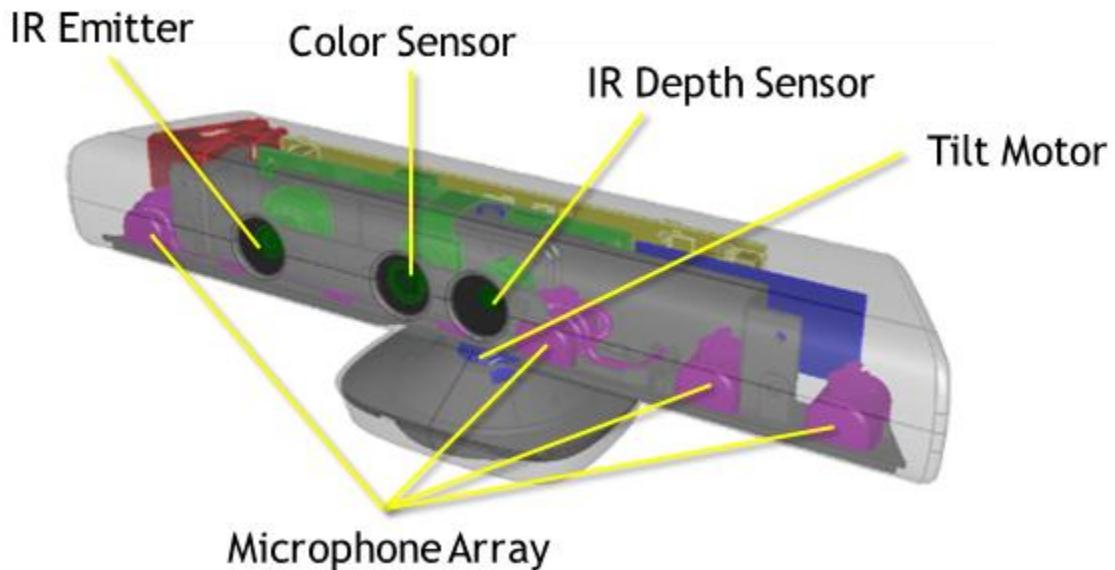


Рисунок 68 – Схема расположения компонентов Kinect

Таблица ?. Технические характеристики Kinect

Kinect	Array Specifications
Viewing angle	43° vertical by 57° horizontal field of view
Vertical tilt range	±27°
Frame rate (depth and color stream)	30 frames per second (FPS)
Audio format	16-kHz, 24-bit mono pulse code modulation (PCM)
Audio input characteristics	A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing including acoustic echo cancellation and noise suppression
Accelerometer characteristics	A 2G/4G/8G accelerometer configured for the 2G range, with a 1° accuracy upper limit.

## Приложение Б. Скрипты преобразования систем координат

Способ через добавление смещения между base\_link и camera\_link

Задаем дополнительное смещение

```
<launch>
    <!-- Simple transform publisher -->
    <node pkg="tf" type="static_transform_publisher"
name="base_link_broadcaster" args="1 0 0 0 0 1 base_link camera_link 100"
/>
</launch>
```

Далее запускаем SLAM алгоритм

```
</launch>
<!-- Backward compatibility launch file, use rtabmap.launch instead -->

    <!-- Your RGB-D sensor should be already started with
"depth_registration:=true".
        Examples:
            $ rosrun freenect_launch freenect.launch
depth_registration:=true
            $ rosrun openni2_launch openni2.launch depth_registration:=true
-->

    <!-- Choose visualization -->
<arg name="rviz"                                default="true" />
<arg name="rtabmapviz"                           default="false" />

    <!-- Localization-only mode -->
<arg name="localization"                        default="false"/>

    <!-- Corresponding config files -->
<arg name="rtabmapviz_cfg"                      default="~/.ros/rtabmap_gui.ini" />
<arg name="rviz_cfg"                            default="$(find
rtabmap_ros)/launch/config/rbd.rviz" />

    <arg name="frame_id"                           default="camera_link"/>    <!-- Fixed
frame id, you may set "base_link" or "base_footprint" if they are published --
->
<arg name="database_path"                       default="~/.ros/rtabmap.db"/>
<arg name="rtabmap_args"                         default="" />                <!--
delete_db_on_start, udebug -->
<arg name="launch_prefix"                       default="" />                <!-- for
debugging purpose, it fills launch-prefix tag of the nodes -->
<arg name="approx_sync"                          default="true"/>              <!-- if
timestamps of the input topics are not synchronized -->

<arg name="rgb_topic"                           default="/camera/rgb/image_rect_color"
/>
<arg name="depth_registered_topic"             default="/camera/depth_registered/image_raw" />
```

```

<arg name="camera_info_topic"           default="/camera/rgb/camera_info" />
<arg name="compressed"                 default="false"/>

<arg name="subscribe_scan"            default="false"/>          <!-- Assuming
2D scan if set, rtabmap will do 3DoF mapping instead of 6DoF -->
<arg name="scan_topic"               default="/scan"/>

<arg name="subscribe_scan_cloud"     default="false"/>          <!-- Assuming
3D scan if set -->
<arg name="scan_cloud_topic"         default="/scan_cloud"/>

<arg name="visual_odometry"          default="false"/>          <!--
Generate visual odometry -->
<arg name="odom_topic"              default="/odom"/>          <!-- Odometry
topic used if visual_odometry is false -->
<arg name="odom_frame_id"           default=""/>             <!-- If set,
TF is used to get odometry instead of the topic -->

<arg name="namespace"                default="rtabmap"/>
<arg name="wait_for_transform"        default="0.2"/>

<include file="$(find rtabmap_ros)/launch/rtabmap.launch">
  <arg name="rtabmapviz"             value="$(arg rtabmapviz)"/>
  <arg name="rviz"                  value="$(arg rviz)"/>
  <arg name="localization"          value="$(arg localization)"/>
  <arg name="gui_cfg"               value="$(arg rtabmapviz_cfg)"/>
  <arg name="rviz_cfg"              value="$(arg rviz_cfg)"/>

  <arg name="frame_id"              value="$(arg frame_id)"/>
  <arg name="namespace"             value="$(arg namespace)"/>
  <arg name="database_path"          value="$(arg database_path)"/>
  <arg name="wait_for_transform"     value="$(arg wait_for_transform)"/>
  <arg name="rtabmap_args"           value="$(arg rtabmap_args)"/>
  <arg name="launch_prefix"          value="$(arg launch_prefix)"/>
  <arg name="approx_sync"            value="$(arg approx_sync)"/>

  <arg name="rgb_topic"              value="$(arg rgb_topic)"/>
  <arg name="depth_topic"            value="$(arg depth_registered_topic)"/>

  <arg name="camera_info_topic"      value="$(arg camera_info_topic)"/>
  <arg name="compressed"             value="$(arg compressed)"/>

  <arg name="subscribe_scan"         value="$(arg subscribe_scan)"/>
  <arg name="scan_topic"              value="$(arg scan_topic)"/>

  <arg name="subscribe_scan_cloud"   value="$(arg subscribe_scan_cloud)"/>
  <arg name="scan_cloud_topic"        value="$(arg scan_cloud_topic)"/>

  <arg name="visual_odometry"        value="$(arg visual_odometry)"/>
</include>
</launch>

```

## Способ через задание initialpose

Скрипт для запуска rgbd\_odom, который рассчитывает местоположение камеры.

```
<launch>
  <node name="rgbd_odometry" pkg="rtabmap_ros" type="rgbd_odometry"
output="screen">
    <remap from="/rgb/image" to="/camera/rgb/image_rect_color" />
    <remap from="/depth/image" to="/camera/depth_registered/image_raw" />
    <remap from="/rgb/camera_info" to="/camera/rgb/camera_info" />
    <param name="initial_pose" value="3 3 3 0 0 1">
    <arg name="frame_id" default="camera_link"/>
  </node>
</launch>
```

Далее необходимо запустить SLAM алгоритм с учетом того, что нода rgbd\_odom уже запущена выше

```
<launch>
<!-- Backward compatibility launch file, use rtabmap.launch instead --&gt;

  &lt;!-- Your RGB-D sensor should be already started with
"depth_registration:=true".
  Examples:
    $ roslaunch freenect_launch freenect.launch
depth_registration:=true
    $ roslaunch openni2_launch openni2.launch depth_registration:=true
--&gt;

  &lt;!-- Choose visualization --&gt;
&lt;arg name="rviz"                      default="true" /&gt;
&lt;arg name="rtabmapviz"                 default="false" /&gt;

  &lt;!-- Localization-only mode --&gt;
&lt;arg name="localization"              default="false"/&gt;

  &lt;!-- Corresponding config files --&gt;
&lt;arg name="rtabmapviz_cfg"            default="~/.ros/rtabmap_gui.ini" /&gt;
&lt;arg name="rviz_cfg"                  default="$(find
rtabmap_ros)/launch/config/rgbd.rviz" /&gt;

  &lt;arg name="frame_id"                default="camera_link"/&gt;    &lt;!-- Fixed
frame id, you may set "base_link" or "base_footprint" if they are published -
--&gt;
&lt;arg name="database_path"             default="~/.ros/rtabmap.db"/&gt;
&lt;arg name="rtabmap_args"              default="" /&gt;                  &lt;!--
delete_db_on_start, udebug --&gt;
&lt;arg name="launch_prefix"             default="" /&gt;                  &lt;!-- for
debugging purpose, it fills launch-prefix tag of the nodes --&gt;
&lt;arg name="approx_sync"               default="true" /&gt;                &lt;!-- if
timestamps of the input topics are not synchronized --&gt;</pre>
```

```

<arg name="rgb_topic" default="/camera/rgb/image_rect_color"
/>
<arg name="depth_registered_topic" default="/camera/depth_registered/image_raw" />
<arg name="camera_info_topic" default="/camera/rgb/camera_info" />
<arg name="compressed" default="false"/>

<arg name="subscribe_scan" default="false" /> <!-- Assuming
2D scan if set, rtabmap will do 3DoF mapping instead of 6DoF -->
<arg name="scan_topic" default="/scan"/>

<arg name="subscribe_scan_cloud" default="false" /> <!-- Assuming
3D scan if set -->
<arg name="scan_cloud_topic" default="/scan_cloud"/>

<arg name="visual_odometry" default="false" /> <!--
Generate visual odometry -->
<arg name="odom_topic" default="/odom" /> <!-- Odometry
topic used if visual_odometry is false -->
<arg name="odom_frame_id" default="" /> <!-- If set,
TF is used to get odometry instead of the topic -->

<arg name="namespace" default="rtabmap"/>
<arg name="wait_for_transform" default="0.2"/>

<include file="$(find rtabmap_ros)/launch/rtabmap.launch">
<arg name="rtabmapviz" value="$(arg rtabmapviz)"/>
<arg name="rviz" value="$(arg rviz)"/>
<arg name="localization" value="$(arg localization)"/>
<arg name="gui_cfg" value="$(arg rtabmapviz_cfg)"/>
<arg name="rviz_cfg" value="$(arg rviz_cfg)"/>

<arg name="frame_id" value="$(arg frame_id)"/>
<arg name="namespace" value="$(arg namespace)"/>
<arg name="database_path" value="$(arg database_path)"/>
<arg name="wait_for_transform" value="$(arg wait_for_transform)"/>
<arg name="rtabmap_args" value="$(arg rtabmap_args)"/>
<arg name="launch_prefix" value="$(arg launch_prefix)"/>
<arg name="approx_sync" value="$(arg approx_sync)"/>
<arg name="rgb_topic" value="$(arg rgb_topic)"/>
<arg name="depth_topic" value="$(arg depth_registered_topic)"/>
/>
<arg name="camera_info_topic" value="$(arg camera_info_topic)"/>
<arg name="compressed" value="$(arg compressed)"/>

<arg name="subscribe_scan" value="$(arg subscribe_scan)"/>
<arg name="scan_topic" value="$(arg scan_topic)"/>

<arg name="subscribe_scan_cloud" value="$(arg subscribe_scan_cloud)"/>
<arg name="scan_cloud_topic" value="$(arg scan_cloud_topic)"/>

<arg name="visual_odometry" value="$(arg visual_odometry)"/>

```

```
<arg name="odom_topic"
    <arg name="odom_frame_id"
        <arg name="odom_args"
            </include>
</launch>
```

value="\$(arg odom\_topic)"/>  
value="\$(arg odom\_frame\_id)"/>  
value="\$(arg rtabmap\_args)"/>

## Приложение В. Код модуля передачи данных планировщикам

```
#include <ros/ros.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <tf/tf.h>
#include <tf/transform_listener.h>
#include <nav_msgs/GetMap.h>
#include <nav_msgs/OccupancyGrid.h>
#include <chrono>

// These things will be used in callback
// and initialized in main
ros::ServiceClient rtabmap_client;
ros::Publisher map_pub;
ros::Publisher map_walls_pub;
ros::Publisher pose_publisher;

// todo I think problem not in my node, but in planner node
// fixme now i saw callback time (4 seconds)
// todo play with rtabmap cells_grid_size and footstep_planner grid_size
// When goal is published
// Get tf of camera from rtabmap and make from it PoseWithCovarianceStamped
// Get proj_map from rtabmap
// Publish proj_map
// Publish pose as initialpose
void goal_callback(const geometry_msgs::PoseStamped goal)
{
    auto start = std::chrono::high_resolution_clock::now();
    // Getting pose from camera tf
    tf::StampedTransform camera_tf;
    tf::TransformListener tf_listener;

    geometry_msgs::PoseWithCovarianceStamped camera_pose;
    try
    {
        tf_listener.waitForTransform("/map", "/base_link",
                                    ros::Time(0), ros::Duration(1));
        // // todo I think odom - where object is now. rtabmap has params to
        // set the requencey of tf publishing
        // // in demo there are no visual odometry and so odom still at start
        // point
        // // maybe we should monitor odometry in other place
        tf_listener.lookupTransform("/map", "/base_link",
                                   ros::Time(0), camera_tf);
        const tf::Vector3 & origin = camera_tf.getOrigin();
        const tf::Quaternion rotation = camera_tf.getRotation();

        camera_pose.header.frame_id = "map";
        camera_pose.pose.pose.position.x = origin.x();
        camera_pose.pose.pose.position.y = origin.y();

        // check here for more information
    }
}
```

```

        // http://answers.ros.org/question/9772/quaternions-orientation-
representation/
        camera_pose.pose.orientation.z = rotation.getZ();
        camera_pose.pose.orientation.w = rotation.getW();
    }
    catch (tf::TransformException ex)
    {
        ROS_ERROR("%s", ex.what());
        return;
    }

    // Getting map from rtabmap service (get_proj_map)
    // todo try just subscribe map. You fixed error. Now it may work
    nav_msgs::GetMap proj_map_srv;
    bool success = rtabmap_client.call(proj_map_srv);
    if (!success)
    {
        ROS_ERROR("Failed to get grid_map from rtabmap");
        return;
    }

    // Publish map
    map_pub.publish(proj_map_srv.response.map);
    //map_walls_pub.publish(proj_map_srv.response.map);

    // publish pose as geometry_msgs/PoseWithCovarianceStamped
    pose_publisher.publish(camera_pose);

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    ROS_INFO("Callback time was %lf", elapsed.count());
}

int main(int argc, char ** argv)
{
    ros::init(argc, argv, "path_controller");
    ros::NodeHandle n;

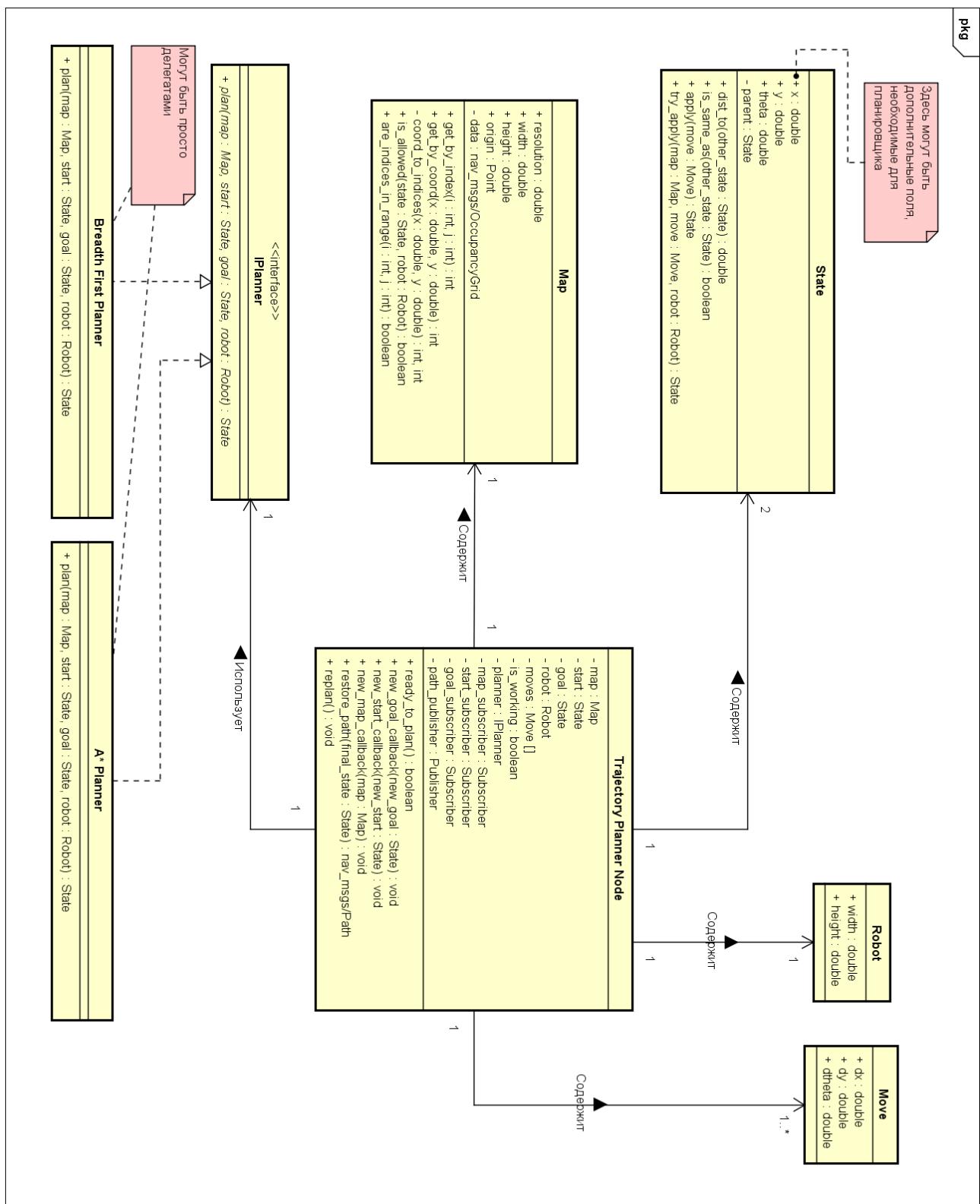
    rtabmap_client =
n.serviceClient<nav_msgs::GetMap>("/rtabmap/get_proj_map");
    map_pub = n.advertise<nav_msgs::OccupancyGrid>("map", 1, true);
    map_walls_pub = n.advertise<nav_msgs::OccupancyGrid>("map_walls", 1,
true);
    pose_publisher =
n.advertise<geometry_msgs::PoseWithCovarianceStamped>("initialpose", 1,
true);

    ros::Subscriber goal_sub = n.subscribe("goal", 1, goal_callback);
    ros::spin();

    return 0;
}

```

## Приложение Г. Диаграмма классов модуля планирования траектории.



## Приложение Д. Листинг программы планирования траектории

move.py

```
class Move:  
    def __init__(self, dx=0.0, dy=0.0, dtheta=0.0):  
        self.dx = dx  
        self.dy = dy  
        self.dtheta = dtheta
```

robot.py

```
class Robot:  
    def __init__(self, width = 2.0, height = 3.0):  
        self.width = width  
        self.height = height
```

state.py

```
import copy  
import rospy  
import tf.transformations  
from geometry_msgs.msg import Pose  
from geometry_msgs.msg import PoseStamped  
from visualization_msgs.msg import Marker
```

class State:

```
    def __init__(self, x=0.0, y=0.0, theta=0.0, parent=None):  
        self.x = x  
        self.y = y  
        self.theta = theta  
        self.parent = parent  
  
        # Variables for A* algorhytm  
        self.g = self.f = self.h = 0  
  
    @staticmethod  
    def from_pose(pose):  
        new_state = State()  
        new_state.x = pose.position.x  
        new_state.y = pose.position.y  
  
        quaternion = (pose.orientation.x, pose.orientation.y,  
pose.orientation.z, pose.orientation.w)  
        (roll, pitch, yaw) =  
tf.transformations.euler_from_quaternion(quaternion)  
        new_state.theta = yaw  
        return new_state  
  
    def dist_to(self, o_s):  
        return ((self.x - o_s.x)**2 + (self.y - o_s.y)**2)**0.5
```

```

def is_same_as(self, o_s):
    # Improoving this constant from 0.01 to 0.05 speed up my algorhytm
    # 1000 times
    return self.dist_to(o_s) < 0.05 # think a much better about this
constant

def apply(self, move):
    return State(self.x + move.dx, self.y + move.dy, self.theta +
move.dtheta)

def try_apply(self, _map, move, robot):
    model_state = copy.copy(self)
    model_state.parent = self

    # TODO fix this misunderstanding with goal
    goal = self.apply(move)
    goal.parent = self
    steps_count = max(int(self.dist_to(goal)) / min(robot.height,
robot.width)), 1)
    step = 1.0 / steps_count

    for i in range(steps_count):
        model_state.x += move.dx * step
        model_state.y += move.dy * step
        model_state.theta += move.dtheta * step

    if not _map.is_allowed(model_state, robot):
        return None
    return goal

def to_pose_stamped(self):
    pose = PoseStamped()
    pose.header.stamp = rospy.Time.now()
    pose.header.frame_id = "map"
    pose.pose.position.x = self.x
    pose.pose.position.y = self.y
    pose.pose.position.z = 0.25
    # TODO angle a little bit later
    return pose

def to_marker(self, robot):
    marker = Marker()
    marker.header.frame_id = "map"
    marker.header.stamp = rospy.Time.now()
    marker.pose.position.x = self.x
    marker.pose.position.y = self.y
    marker.pose.position.z = 0

    marker.scale.x = robot.width
    marker.scale.y = robot.height
    marker.scale.z = 0.2

```

```
marker.color.r = 1.0
marker.color.g = 0.0
marker.color.b = 0.0
marker.color.a = 1.0

marker.type = Marker.CUBE
marker.action = marker.ADD

return marker
```

```

map.py
from nav_msgs.msg import OccupancyGrid
from exceptions import IndexError
from geometry_msgs.msg import Point
import rospy

class Map:
    def __init__(self, grid_map):
        self.map = grid_map
        self.width = grid_map.info.width
        self.height = grid_map.info.height
        self.resolution = grid_map.info.resolution

        self.origin = Point()
        self.origin.x = grid_map.info.origin.position.x
        self.origin.y = grid_map.info.origin.position.y

    def get_by_index(self, i, j):
        if not self.are_indices_in_range(i, j):
            raise IndexError()
        return self.map.data[i*self.width + j]

    # i is for row (y), j is for col (x)
    def get_by_coord(self, x, y):
        return self.get_by_index(*self.coord_to_indices(x, y))

    def coord_to_indices(self, x, y):
        i = int((y - self.origin.y) / self.resolution)
        j = int((x - self.origin.x) / self.resolution)
        return (i, j)

    def are_indices_in_range(self, i, j):
        return 0 <= i < self.height and 0 <= j < self.width

    def is_allowed(self, state, robot):
        was_error = False
        i, j = self.coord_to_indices(state.x, state.y)
        side = int((max(robot.width, robot.height) / self.resolution) / 2)
        try:
            for s_i in range(i-side, i+side):
                for s_j in range(j-side, j+side):
                    if self.get_by_index(s_i, s_j) == 100:
                        return False
        except IndexError as e:
            # rospy.loginfo("Indices are out of range")
            was_error = True
        return True and not was_error

```

```

trajectory_planner.py
#!/usr/bin/python2.7
import rospy
from geometry_msgs.msg import PoseStamped, PoseWithCovarianceStamped
from nav_msgs.msg import OccupancyGrid, Path
from visualization_msgs.msg import MarkerArray, Marker
import visualization_msgs
import copy
import heapq

from move import Move
from state import State
from robot import Robot
from map import Map

class TrajectoryPlanner:
    def __init__(self):
        self.map = None
        self.start = None
        self.goal = None

        self.moves = [Move(0.1, 0, 0), Move(0, 0.1, 0),
                     Move(-0.1, 0, 0), Move(0, -0.1, 0)]
        self.robot = Robot(0.5, 0.5)
        self.is_working = False # Replace with mutex after all

        self.map_subscriber = rospy.Subscriber("grid_map", OccupancyGrid,
self.new_map_callback)
        self.start_subscriber = rospy.Subscriber("initialpose",
PoseWithCovarianceStamped, self.new_start_callback)
        self.goal_subscriber = rospy.Subscriber("goal", PoseStamped,
self.new_goal_callback)

        self.path_publisher = rospy.Publisher("trajectory", MarkerArray,
queue_size=1)
        self.pose_publisher = rospy.Publisher("debug_pose", PoseStamped,
queue_size=1)

    def ready_to_plan(self):
        return self.map is not None and self.start is not None and self.goal
is not None

    def new_goal_callback(self, goal_pose):
        if not self.is_working:
            self.is_working = True
            new_goal = State.from_pose(goal_pose.pose)
            if self.map is not None and self.map.is_allowed(new_goal,
self.robot):
                self.goal = new_goal
                rospy.loginfo("New goal was set")
                if self.ready_to_plan():
                    self.replan()

```

```

        self.is_working = False

    def new_start_callback(self, start_pose):
        if not self.is_working:
            self.is_working = True
            new_start = State.from_pose(start_pose.pose)
            if self.map is not None and self.map.is_allowed(new_start,
self.robot):
                self.start = new_start
                rospy.loginfo("New start was set")
                if self.ready_to_plan():
                    self.replan()
            self.is_working = False

    def new_map_callback(self, grid_map):
        if not self.is_working:
            self.is_working = True
            self.map = Map(grid_map)
            rospy.loginfo("New map was set")
            if self.ready_to_plan():
                self.replan()
            self.is_working = False

# In future we can create field planner, which will contain object of
algorhytm
# A* algorhytm based on http://web.mit.edu/eranki/www/tutorials/search/
def replan(self):
    # If we colud make result of is_same_as discrete we could use set
here and speed up this algo
    rospy.loginfo("Planning was started...")
    robot_dimension = min(self.robot.width, self.robot.height)
    final_state = None
    opened = []
    heapq.heappush(opened, (0.0, self.start))
    closed = []

    while opened and final_state is None: # or while we have enough time
        q = heapq.heappop(opened)[1]
        self.pose_publisher.publish(q.to_pose_stamped())
        for move in self.moves:
            successor = q.try_apply(self.map, move, self.robot)
            if successor is not None:
                if successor.dist_to(self.goal) < robot_dimension:
                    final_state = successor
                    break
                successor.g = q.g + successor.dist_to(q)
                successor.h = successor.dist_to(self.goal)
                successor.f = 0.2 * successor.g + successor.h
                successor.parent = q

                better_in_opened =
any(other_successor.is_same_as(successor) and other_f < successor.f for
other_f, other_successor in opened)

```

```

        if not better_in_opened:
            better_in_closed =
any(other_successor.is_same_as(successor) and other_successor.f < successor.f
for other_successor in closed)
        if not better_in_closed:
            heapq.heappush(opened, (successor.f, successor))
closed.append(q)

if final_state is None:
    rospy.loginfo("No path found")
else:
    # Restore and publish path
    rospy.loginfo("Restoring path from final state...")
    path = self.restore_path(final_state)
    self.path_publisher.publish(path)
    rospy.loginfo("Planning was finished...")

def replan_width(self):
    rospy.loginfo("Planning was started...")
    final_state = None
    opened = [self.start]
    closed = []

    while opened and final_state is None: # or while we have enough time
        item = opened.pop()
        self.pose_publisher.publish(item.to_poseStamped())
        for move in self.moves:
            new_item = item.try_apply(self.map, move, self.robot)
            if new_item is not None:
                if new_item.dist_to(self.goal) < 0.5:
                    final_state = new_item
                    break
            else:
                if not any(other_item.is_same_as(new_item) for
other_item in opened):
                    opened.insert(0, new_item)
                    closed.append(item)

    if final_state is None:
        rospy.loginfo("No path found")
    else:
        # Restore and publish path
        rospy.loginfo("Restoring path from final state...")
        path = self.restore_path(final_state)
        self.path_publisher.publish(path)
        rospy.loginfo("Planning was finished...")

def restore_path(self, final_state):
    current_state = copy.copy(final_state)
    path = MarkerArray()
    pose_id = 0
    while True:
        pose_marker = current_state.to_marker(self.robot)

```

```
pose_marker.id = pose_id
path.markers.append(pose_marker)

current_state = current_state.parent
pose_id += 1

if current_state is None:
    break
return path

def main():
    rospy.init_node("trajectory_planner")
    planner = TrajectoryPlanner()
    rospy.spin()

main()
```