ИТМО

Бжихатлов И.А.

Операционная система для роботов ROS. Часть 1 (базовый уровень)

Санкт-Петербург
2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

И.А. Бжихатлов

ОПЕРАЦИОННАЯ СИСТЕМА ДЛЯ РОБОТОВ ROS. ЧАСТЬ 1 (БАЗОВЫЙ УРОВЕНЬ)

УЧЕБНОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ
В УНИВЕРСИТЕТЕ ИТМО

по направлениям подготовки 15.03.06 «Мехатроника и робототехника» и 27.03.04 «Управление в технических системах» в качестве учебного пособия для реализации основных профессиональных образовательных программ высшего образования бакалавриата

ИТМО

Санкт-Петербург
2025

И.А. Бжихатлов. Операционная система для роботов ROS. Часть 1 (базовый уровень) — Санкт-Петербург: Университет ИТМО, 2025. — 109 с.
Рецензент: Колюбин С.А, д.т.н, профессор, факультет Систем управления и робототехники, Университет ИТМО, Санкт-Петербург, Россия.

В данном пособии изложены методические рекомендации по изучению современными инструментов для разработки и тестирования программного обеспечения для роботов. Практические занятия и лабораторные работы способствуют освоению технологий моделирования роботов произвольной кинематики, построению виртуальных симуляционных сред и разработке управляющих алгоритмов для роботов на основе данных с датчиков.

ИТМО (Санкт-Петербург) — национальный исследовательский университет, научно-образовательная корпорация. Альма-матер победителей международных соревнований по программированию. Приоритетные направления: IT и искусственный интеллект, фотоника, робототехника, квантовые коммуникации, трансляционная медицина, Life Sciences, Art& Science, Science Communication. Лидер федеральной программы «Приоритет-2030», в рамках которой реализовывается программа «Университет открытого кода». С 2022 ИТМО работает в рамках новой модели развития — научно-образовательной корпорации. В ее основе академическая свобода, поддержка начинаний студентов и сотрудников, распределенная система управления, приверженность открытому коду, бизнес-подходы к организации работы. Образование в университете основано на выборе индивидуальной траектории для каждого студента.
ИТМО пять лет подряд — в сотне лучших в области Automation & Control (кибернетика) Шанхайского рейтинга. По версии SuperJob занимает первое место в Петербурге и второе в России по уровню зарплат выпускников в сфере IT. Университет в топе международных рейтингов среди российских вузов. Входит в топ-5 российских университетов по качеству приема на бюджетные места. Рекордсмен по поступлению олимпиадников в Петербурге. С 2019 года ИТМО самостоятельно присуждает ученые степени кандидата и доктора наук.

СОДЕРЖАНИЕ

Введение

Целью данного пособия является ознакомление студентов (изучающих дисциплины, связанные с программированием роботов) с современными инструментами для разработки, поддержания и развития программных решений для роботов. Пособие предназначено для использования в курсе "Операционная система ROS". В результате успешного освоения материалов пособия студенты получат навыки разработки прикладного программного обеспечения в среде ROS и интеграции робототехнического оборудования в реальных приложениях, а также умения отлаживать, тестировать и визуализировать поведение робота. Получат знания об архитектуре и инструментах ROS, что позволит повысить скорость разработки технических решений за счет моделирования и последующего внедрения в робототехнические приложения.

Название "Операционная система для роботов" и большинство терминов, приведённых в данном методическом пособии, заимствованы из английского языка, так как разработкой инструмента занимались инженеры из разных стран, также известного как "ROS сообщество". Robot Operation System (далее ROS) представляет собой набор инструментов для разработчиков робототехнических решений и предназначен для ускорения процесса разработки управляющих программ для различных роботов. Данное программное обеспечение имеет открытый исходный код и предусматривает возможность использования без нарушения авторских прав в любых приложениях, в том числе и коммерческих решениях.

Важно отметить, что разработка прикладного программного обеспечения может вестись и без использования ROS, но при таком подходе временные затраты на разработку, поддержку и развитие программных решений увеличивается в десятки раз. Основу ROS составляют большое количество уже разработанных программных модулей и набор инструментов, позволяющих ускорить настройку и отладку программных решений как с использованием физически воплощенных роботов, так и имитационных моделей. Использование имитационной модели при разработке прикладного программного обеспечения

для робототототехнических решений стало одним из ключевых факторов успеха, позволяющего избегать большого количества проблем. Разработка имитационной модели робота на начальной стадии требует существенных временных затрат, однако практика показывает, что в последующем это время окупается очень быстро.

Рассматриваемая в пособии первая версия ROS во многом легла в основу более новой, второй версии с рядом значительных улучшений во внутренних механизмах работы. Несмотря на то, что ROS2 является актуальной версией, изучение ROS рекомендуется начинать с первой версии, так как вторая версия является более сложной в изучении из-за расширенных возможностей и гибкости, заложенной в неё разработчиками.

В пособии представлены базовые возможности по организации работы программного обеспечения с использованием ROS как на примере модели в симуляторе, так и физического устройства, способного управлять роботом. Рассмотрены способы внедрения ROS в современные мобильные роботы и особенности работы в программе имитационного моделирования Gazebo.

Перед изучением ROS важно разобраться, какие преимущества предоставляет данный инструмент и какие ограничения накладывает на разработчиков. Начнем с накладываемых на разработчика ограничений:
1. Необходимо иметь в составе робота высокоуровневый вычислитель на базе микропроцессорных модулей, как правило, одноплатного компьютера или полноценного ПК.
2. Необходимо знание операционной системы Linux и основные bash команды.
3. Необходимо знание одного из двух языков программирования: C++ или Python, при этом некоторые возможности доступны только на C++.
4. Умение читать техническую документацию на английском языке.

Преимущества:
1. Наличие большого количества программных модулей с открытым исходным кодом.
2. Минимальное количество требуемого для написания

программного кода в типовых задачах.

1 ROS – общие сведения

1.1 Версии, дистрибутивы и система сборки

ROS - это фреймворк (middleware OS), его можно отнести к системному программному обеспечению, обеспечивающему взаимодействие базовой операционной системы и прикладного программного обеспечения, предназначен для ускорения разработки программного кода для роботов. Хоть официально и заявлена поддержка на различных операционных системах, наиболее стабильная работа ROS реализована с операционной системой на ядре Linux (Например, Ubuntu). Далее рассматривается ROS, работающая под операционной системой Ubuntu [1] версии 20.04, следовательно, необходимо знание основных команд для работы в терминале (bash-терминал).

Кроме того, что существует ROS первой и второй версии, также фреймворк разделяется на разные версии дистрибутива. Для ROS версии 1 последним выпущенным дистрибутивом является Noetic, дальнейшая разработка первой версии прекращена. Таким образом, при работе с ROS всегда необходимо учитывать версию дистрибутива, так как от этого будет зависеть, какие возможности доступны разработчику. Важно отметить, что вне зависимости от дистрибутива ROS в рамках своей версии имеет одинаковую структуру и компоненты, различие лишь в том, что некоторые возможности оптимизируются и дополняются с выходом новых версий дистрибутива.

Чтобы воспользоваться готовыми решениями, достаточно установить ROS на операционную систему с ядром Linux. Однако, для написания собственных программ необходимо разобраться в ряде вопросов, и первый вопрос заключается в том, где необходимо хранить свои программные решения и как их запустить. Разработка в ROS должна вестись в рабочем окружении. Для запуска своего решения, его необходимо собрать, поскольку ROS использует компилируемый язык программирования C++ (даже если программный код написан на Python, косвенно используется C++).

В данном курсе рассматривается ROS Noetic (ROS1), так как он остается наиболее простым для изучения ROS, в то время как

ROS2 является более сложной и улучшенной(оптимизированной и гибкой) версией, в которой возможности ROS1 дополняются. Далее ROS упоминается без уточнения версии, подразумевая версию 1.

Рабочее окружение в ROS содержит 3 рабочих пространства: src, build и devel. Создать рабочее окружение можно в любом месте, для этого достаточно создать папку, внутри которой будет содержаться подпапка с названием src. Название папки с рабочим окружением можно задать любое, однако принято его назвать `catkin_ws` или `workspace`, чтобы было удобно в дальнейшем его находить. В рабочем окружении создавать подпапки build и devel не нужно, они создаются автоматически. Также никогда не рекомендуется редактировать файлы, размещенные в эти папки (их обновление происходит автоматически).

Система сборки в ROS называется catkin, она устанавливается автоматически вместе с ROS. Catkin необходим для низкоуровневой сборки системных макросов и инфраструктуры для ROS. Чтобы ранее созданная папка стала рабочим окружением catkin, необходимо выполнить инициализацию. В дальнейшем, все программные пакеты, размещенные в `catkin_ws/src/*` будут использовать catkin для сборки. Чтобы выполнить сборку с использованием catkin, необходимо воспользоваться инструментом сборки. Инструментов сборки для catkin существует несколько, наиболее популярный catkin_make, catkin_make_isolated и catkin build. При этом catkin_make устанавливается автоматически, а catkin build относится к отдельному инструменту командной строки Catkin Tools [2], который необходимо установить дополнительно. Основные преимущества catkin build: изолированная сборка пакета, более структурированный и легко читаемый цветной вывод командной строки. Также пользователь получает много полезных команд, таких как catkin clean для очистки, build, devel, install, catkin list и т.д. Кроме того, catkin build работает в любом месте рабочего пространства, не только в верхнеуровневой папке рабочего окружения, имеется возможность собирать отдельные пакеты, а не только все пакеты в окружении одновременно (что может занимать много времени).

1.2 Компоненты и структура ROS

Каждое робототехническое решение имеет программные компоненты, которые являются одинаковыми и следовательно, необходимости писать программы каждый раз заново - нет. Это самый простой способ объяснить необходимость в ROS. ROS имеет три концептуальных уровня: уровень файловой системы, уровень вычислительного графа и уровень сообщества.

Уровень вычислительного графа.

Узлы (также известные как "ноды"), это процессы, выполняющие вычисления. ROS спроектирован так, чтобы быть модульным и мелкомасштабным; Обычно система состоит из большого количества узлов. Каждый узел выполняет определенную задачу, такую как получение данных от сенсоров, управление приводами робота или обработка изображений с камеры. Узлы обмениваются данными друг с другом через топики (topics) и сервисы (service).

Топики представляют собой каналы связи, по которым узлы публикуют и подписываются на данные. Такой механизм позволяет реализовать легко расширяемые каналы связи, при добавлении новых узлов в систему, отсутствует необходимость в изменении ранее созданных. Данные, передаваемые по топикам, называются сообщениями и имеют строго определенную структуру, заранее заданную разработчиком прикладного программного обеспечения.

Сообщение — это структура данных, содержащая типизированные поля. Поддерживаются стандартные типы (целые, с плавающей запятой, логические значения и т. д.), а также массивы примитивных типов.

При передаче данных через топики принято назвать предающий узел publisher, а принимающий subscriber. Важно отметить, что такой способ передачи данных через топики (как показано на рисунке 1.1) имеет важное преимущество в обеспечении гибкости, но подходит далеко не для всех задач из-за однонаправленности канала передачи данных. Поэтому, **сервисы** – важная альтернатива механизму topic/publisher/subscriber, когда нужно взаимодействовать в режиме запрос/ответ (также известный как Сервер/Клиент). Сервисы в ROS обеспечивают синхронный обмен

данными между узлами. Клиентский узел отправляет запрос на серверный узел, а серверный узел возвращает ответ по готовности.

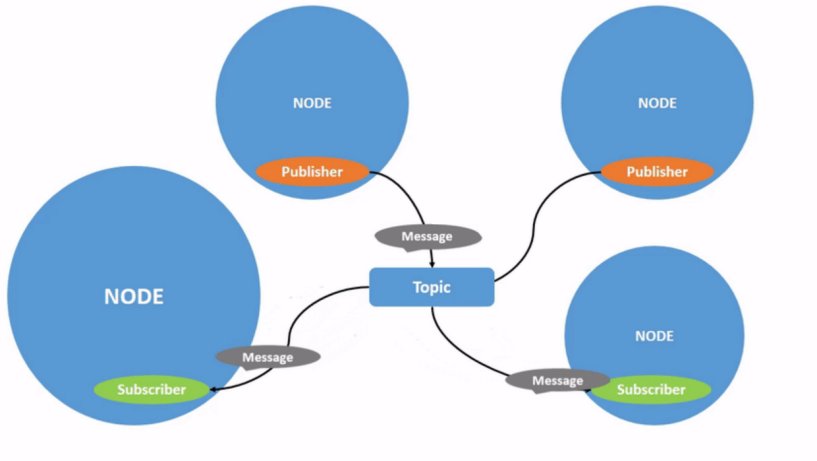

Рисунок 1.1. Иллюстрация передачи сообщения через вычислительный граф

Для обеспечения взаимодействия узлов необходим инструмент, выполняющий связующие функции. Таков в ROS является ROS мастер (ROS Master) - обеспечивает регистрацию имени и поиск по остальной части вычислительного графа. Без мастера узлы не смогут находить друг друга, обмениваться сообщениями или вызывать сервисы. Сервер параметров (часть мастера) – позволяет хранить данные централизованно, но в ограниченном количестве.

Уровень файловой системы.

Пакеты — это основная единица организации программного обеспечения в ROS. (Бывают еще метапакеты и стэки). Пакет содержит исходный код (ROS узлы), данные зависимости пакета от других пакетов и другие ресурсы для конкретной функциональности (наборы данных, файлы конфигурации или что-либо еще, относящееся к пакету). Пакеты — это самый атомарный элемент в решении с использованием ROS, который может распространяться отдельно и интегрироваться в другие робототехнические системы.

Содержание пакета в ROS:
- Package Manifests: Манифест (package.xml) – предоставить метаданные о пакете, включая его имя, версию, описание, зависимости и другие данные, например об экспортированных пакетах.
- Типы сообщений и сервисов: определяют структуры данных для сообщений, отправляемых в ROS.

1.3 Программный пакет в ROS

Как было ранее отмечено, программный пакет (также известный как ROS пакет) представляет собой основную единицу организации программного обеспечения в пардигме ROS. Он содержит все необходимые файлы для реализации определённой функциональности: исходный код программы (ноды), файлы конфигурации, описания сообщений, сервисов, а также скрипты для запуска и сборки. Разработка программ ведется в рамках ROS пакета, впоследствии из различных ROS пакетов собирается комплексное решение. Например, ROS пакет может реализовывать базовый алгоритм управления роботом определенного типа, при этом алгоритм планирования движения может быть реализован в другом пакете. Однако, эти пакеты могут легко взаимодействовать через программные интерфейсы и эффективно выполнять задачу движения робота к заданной точке.

В ROS программные пакеты можно устанавливать из отрытых репозиториев ROS либо собирать(компилировать) из исходного кода в окружении catkin. В случае установки из открытых репозиториев возможность изменения исходного кода отсутствует, но для интеграции в свое решение может быть достаточно воспользоваться программными интерфейсами. Однако гораздо больше возможностей доступно в случае работы с исходным кодом. Далее рассмотрим структуру пакета для ROS:

В ROS к пакету предъявляются три основных требования:

- Каждый пакет должен располагаться в отдельной папке;
- Пакет должен содержать файл package.xml, совместимый с catkin;

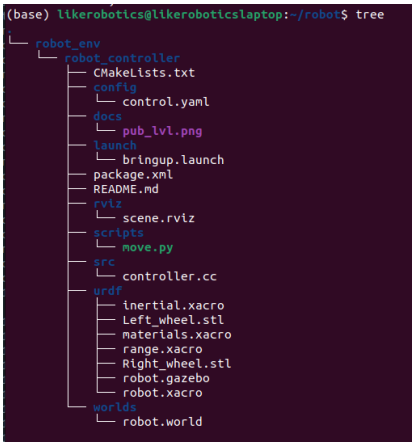

Рисунок 1.2. Пример содержания ROS пакета (ROS)

- Пакет должен содержать файл CMakeLists.txt, который использует Catkin.

Можно выделить следующие способы создания пакета:

1. Скопировать существующий пакет, переименовать название папки и название пакета в файлах package.xml и CMakeLists.txt;

2. Воспользоваться командой `catkin_create_pkg`.

Второй способ является наиболее рекомендуемым, так как позволяет явно указывать параметры пакета и снижает возможности допустить опечатку в названиях и потенциально возможном переносе лишнего программного кода.

Пример вызова команды:

```
1  catkin_create_pkg my_robot_super_controller
     dependency_package_name
```

где *my_robot_super_controller* - название пакета, *dependency_package_name* - список названий пакетов, разделенных пробелами, от которых имеется зависимость

(позволяет автоматически добавить сведения о зависимостях в служебные файлы пакета).

1.4 Система сборки catkin

Как ранее было упомянуто, catkin command line tools имеет ряд преимуществ, следовательно, в дальнейшем вы будете ей пользоваться. Для сборки необходимо воспользоваться catkin build вместо catkin_make, несмотря на то что во всех инструкциях на официальном сайте ROS Wiki [3] предлагается catkin_make.

Основное отличие — изолированная среда, которую вы получаете при сборке Catkin. Это делает всю конфигурацию сборки более разделенной и устойчивой к изменениям в конфигурации (добавление/удаление пакета, изменение переменной cmake и т. д.). Кроме того, вы получаете более структурированный и легко читаемый цветной вывод командной строки, что намного приятнее. Также вместе с catkin вы получаете много полезных команд, таких как catkin clean для очистки служебных рабочих пространств build, devel, install, catkin list и т.д. Также catkin build работает в любом месте рабочего пространства, не только в верхнеуровневой директории, также можно собирать отдельные пакеты, а не только все сразу (что существенно ускоряет процесс).

Создание рабочего окружения catkin

Рабочим окружением называется каталог (папка) файловой системы, в которой будут храниться рабочие пространства src, build, devel и т.д., необходимые для работы catkin. Рабочее окружение может иметь любое имя, но чаще всего используются названия "catkin_ws" или "workspace" и располагаются в домашнем каталоге пользователя. Чтобы обычный каталог стал рабочим окружением catkin, достаточно внутри него создать подкаталог(папку) с названием src, в которой будут хранится ROS пакеты. Кроме требований к структуре файловой системы, необходимо выполнить инициализация при помощи команды `catkin init`. Чтобы проверить, является ли папка рабочим окружением Catkin, можно воспользоваться командой вывода параметров окружения `catkin config`.

1.5 Система контроля версий git

Система контроля версий git будет использоваться как инструмент для отправки выполненной работы на проверку и не является темой данного пособия, поэтому мы коснемся только практических аспектов, позволяющих воспользоваться данным инструментом. За подробностями по git обращайтесь к официальной документации [4]. Программа Git установлена по умолчанию в операционной системе Ubuntu 18 и выше. Проверить это можно, выполнив следующую команду, которая должна вывести версию установленной программы.

```
1  git --version
```

Программа git работает с директориями git, также часто называемыми репозиториями. Чтобы узнать, является ли та или иная папка git-репозиторией, необходимо проверить наличие в ней скрытой папки с названием .git. В терминале можно отобразить все содержащиеся файлы, включая скрытые, воспользовавшись командой.

```
1  ll
```

При выполнении "клонирования" репозитория из интернета через команду "git clone" будет создана папка, которая является git-репозиторием.

После изменения содержимого репозитория в него можно зайти в терминале командой "cd" и проиндексировать все изменения.

```
1  git add .
```

После этого можно сделать "коммит". Подробнее о том, что это такое, можно прочитать в документации. Важно: отправка изменений на сервер невозможна без выполнения коммита. Далее приведён пример выполнения коммита с комментарием "some comment" (комментарий обязательно нужно указать, при этом его содержание может быть любым):

```
1  git commit -m "some comment"
```

После этого останется только отправить изменения на сервер git (это может быть gitlab, github или любой другой сервис с поддержкой git) воспользовавшись командой.

```
1  git push
```

1.6 Зависимости в ROS

ROS разрабатывался таким образом, чтобы максимально избегать повторного написания программного кода и использовать уже написанные программные пакеты. В связи с этим ROS пакеты имеют зависимости от других ROS пакетов, которые необходимо указывать в package.xml. В самом простом случае используются зависимости от пакетов std_msgs, rospy, roscpp, необходимых для работы простейшего ROS-пакета..

Зависимости, которые мы указываем при создании пакета являются зависимостями первого порядка, их можно просмотреть с помощью инструмента rospack.

```
1  rospack depends1 my_robot_super_controller
```

В большинстве случаев зависимость также будет иметь свои собственные зависимости (косвенные зависимости). Например, у rospy есть и другие зависимости:

```
1  rospack depends1 rospy
```

ROS пакет может иметь довольно много косвенных зависимостей. Но rospack может рекурсивно определять все вложенные зависимости при помощи команды depends.

```
1  rospack depends my_robot_super_controller
```

При использовании загруженных из интернета ROS пакетов можно легко проверить имеющиеся зависимости и установить недостающие пакеты при помощи rosdep.

Rosdep является автономным инструментом командной строки для установки системных зависимостей, который вы можете загрузить и использовать отдельно от ROS. Установка rosdep выполняется следующим образом:

```
sudo apt-get install python3-rosdep
```

Инициализация выполняется один раз после установки:

```
sudo rosdep init
```

Обновление пакета рекомендуется выполнять после установки:

```
rosdep update
```

Установка зависимостей отдельного пакета с названием AMAZING_PACKAGE_FROM_INTERNET:

```
rosdep install AMAZING_PACKAGE_FROM_INTERNET
```

Установка зависимостей всех пакетов (выполняем в корневой папке пакета):

```
rosdep install --from-paths src --ignore-src
  -r -y
```

1.7 Файлы запуска roslaunch в ROS

Файлы типа .launch предназначены для одновременного запуска нескольких узлов (процессов) и располагаются в папке launch внутри пакета.

```
<launch>
    <!--<node pkg="turtlesim" name="sim" type="turtlesim_node"/>-->
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>

</launch>
```

Рисунок 1.3. Пример launch файла, в которой запускается один узел

Рассмотрим на примерах 1.3 и 1.4 основные элементы launch файла:

<launch> - тип разметки (сопровождается закрывающим тегом </launch>) <group ns="turtlesim1"> - группа ресурсов (с

```
<launch>
  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
</launch>
```

Рисунок 1.4. Пример launch файла, в которой запускаются узлы внутри различных пространств имен.

указанием пространства имен) <node pkg="turtlesim" name="sim" type="turtlesim_node"/> - запуск ноды из пакета (одной строкой). <node pkg="turtlesim" name="mimic" type="mimic"> - второй способ запуска ноды из пакета (в две строки) <remap from="input" to="turtlesim1/turtle1"/> - вложенный тег переадресации (внутри ноды) </node>- закрывающий тэг запуска ноды

Пример запуска файла из терминала:

```
roslaunch package_name robot_setup.launch
```

При запуске узла через файл запуска вывод информации через loginfo по умолчанию не работает. Чтобы включить вывод логов, необходимо указать параметр output="screen", как показано далее.

1.8 Имена и пространства имен

Имена ресурсов графа — это важный механизм в ROS, обеспечивающий инкапсуляцию. Каждый ресурс определяется в пространстве имен, которое он может использовать совместно со многими другими ресурсами. Соединения могут устанавливаться между ресурсами в разных пространствах имен, но обычно это делается с помощью кода интеграции над обоими пространствами имен. Эта инкапсуляция изолирует различные части системы от случайного захвата ресурса с неверным именем или глобального перехвата имен.

К названиям ресурсов предъявляются следующие требования:

- Первый символ – это буква ([a-z|A-Z]), тильда ( ) или косая черта (/).
- Последующие символы могут быть буквенно-цифровыми ([0-9|a-z|A-Z]), подчеркиванием (_) или косой чертой (/).
- Исключение: базовые имена (описанные ниже) не могут содержать косую черту (/) или тильды ( ).

В ROS существует четыре типа имен ресурсов графа: базовые, относительные, глобальные и локальные(приватные), которые имеют следующий синтаксис:

1. base - базовые имена
2. relative/name - относительные имена
3. /global/name - глобальные имена
4. private/name - локальные имена

Имена, начинающиеся с «/», являются глобальными. Глобальных имен следует избегать, насколько это возможно, поскольку они ограничивают переносимость программного кода.

Внутри программного кода также можно получить данные о самих ресурсах, например:

rospy.get_name() - Получить полное имя этого узла.
rospy.get_namespace() – Получить текущее пространство имен.

1.9 Типы сообщений

Файлы с расширением .msg представляют собой простые текстовые файлы, описывающие поля сообщений ROS (каждое поле включает тип и имя). Они используются для генерации исходного кода сообщений на разных языках. Сами файлы msg хранятся в каталоге пакета, в подпапке msg.

Типы полей, которые вы можете использовать (примитивы):

- int8, int16, int32, int64 (plus uint*)

- float32, float64
- string
- time, duration
- прочие файлы msg
- переменной длины массивы array[] и фиксированной длины массивы array[C]

1.10 Система преобразований TF

Чтобы описывать движение объектов в пространстве, бывает удобно ввести прикреплённые к ним относительные системы координат (СК), а переход из одной СК в другую задавать преобразованием системы координат (TF).

Система преобразований TF — это модуль для отслеживания изменений между системами координат во времени. Он поддерживает древовидную структуру взаимосвязей между системами координат с учётом временной буферизации. Кроме того, модуль позволяет пользователю преобразовывать точки, векторы и другие данные между системами координат в нужный момент времени.

TF реализован через механизм publisher/subscriber в топиках /tf и /tf_static. Используется специальный тип сообщений, содержащий header с временной меткой (временем отправки сообщения).

Удобнее всего отслеживать преобразование систем координат при помощи визуализатора RViz, как показано на рисунке 1.5.

Также бывает полезно визуализировать цепочку преобразований систем координат.

rosrun tf view_frames – вывод дерева преобразований evince frames.pdf – открыть файл в режиме просмотра

1.11 URDF - моделирование робота

URDF (Unified Robot Description Format) - задает в формате XML описание модели робота, включающее:

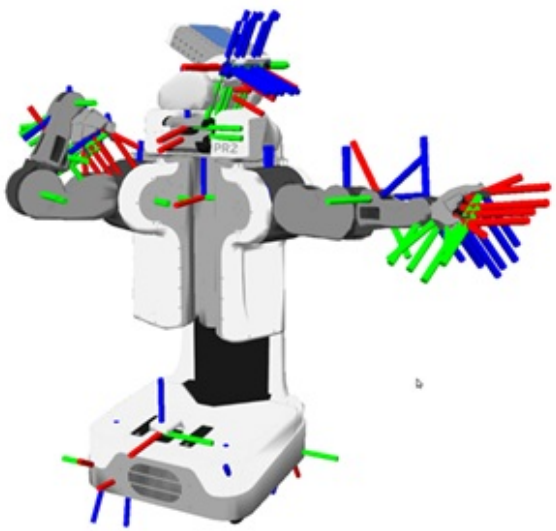

Рисунок 1.5. Пример визуализации систем координат в Rviz.

- Кинематику и динамику
- Визуальное представление
- Модель столкновения

Моделирование контакта (столкновения) является одной из самых вычислительно требовательных операций, поэтому принято всегда использовать апроксимацию модели контакта при помощи простых фигур. На рисунке 1.6 показаны визуальная составляющая модели робота и как моделируется контакт робота.

Далее, рассмотрим структуру URDF на примере только визуальной составляющей, построенной из простых фигур. Описание состоит из набора элементов (звеньев) и набора соединительных элементов (шарниров). Звенья могут соединяться только через шарниры (подвижные или фиксированные). Также важно отметить, что каждый шарнир или звено имеет жестко соединенную с ним систему координат (<origin>).

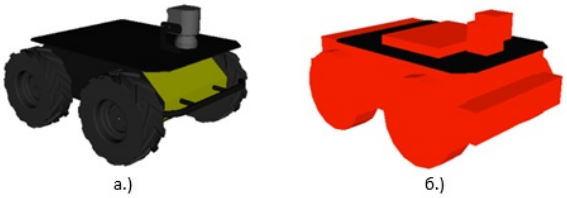

Рисунок 1.6. Пример модели робота: а. - визуальной составляющей модели, б. - модель контакта.

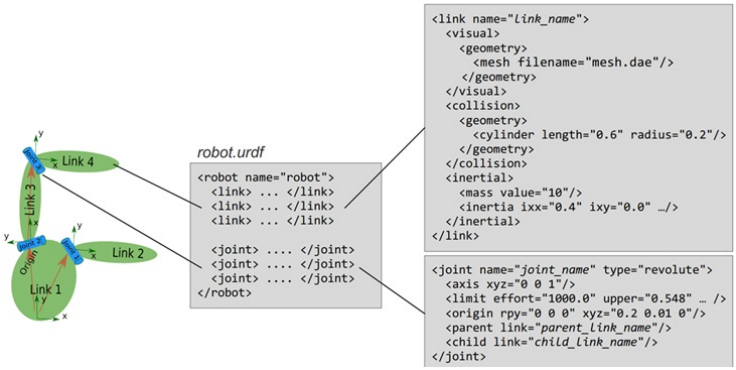

Рисунок 1.7. Структура URDF разметки

В URDF доступны следующие типы шарниров:

- revolute — шарнирное соединение, которое вращается вдоль оси и имеет ограниченный диапазон, определяемый верхним и нижним пределами;
- continuous — непрерывное шарнирное соединение, вращающееся вокруг оси и не имеющее верхних и нижних пределов;
- prismatic — скользящее соединение, которое скользит вдоль оси и имеет ограниченный диапазон, определяемый верхним и нижним пределами;

- fixed — На самом деле это не шарнир, потому что он не может двигаться. Все степени свободы заблокированы. Этот тип соединения не требует <axis>, <calibration>, <dynamics>, <limits> или <safety_controller>;
- floating — допускает движение по всем 6 степеням свободы;
- planar — допускает движение в плоскости, перпендикулярной оси.

Модель робота обычно хранится в robot_description сервера параметров и его можно визуализировать в Rviz при помощи плагина Robot Model.

Список основных тегов URDF разметки:

- robot - Описывает все свойства робота.
- link - Описывает кинематические и динамические свойства звена;
- transmission - Трансмиссии связывают приводы с шарнирами и представляют собой их механическое соединение;
- joint - Описывает кинематические и динамические свойства шарнира;
- gazebo - Описывает параметры моделирования, такие как демпфирование, трение и т. д.;
- sensor - Описывает датчик, например камеру, лучевой датчик и т. д.;
- model_state - Описывает состояние модели в определенный момент времени;
- model - Описывает кинематические и динамические свойства конструкции робота.

После внесения изменений в URDF разметку рекомендуется проверить наличие ошибок в файле при помощи инструмента check_urdf.

1.12 XACRO - оптимизация URDF

URDF может формироваться с помощью макросов. XML Macros (macro language) также известен как **XACRO**. С помощью XACRO можно создавать более короткие и легко читаемые XML-файлы, что расширяет возможности XML-разметки.

Среди преимуществ XACRO можно выделить:

- разделение разметки модели робота на несколько файлов и сборка в один файл через <xacro:include>, <xacro:macro>;
- использование математических выражений (автоматически рассчитывается и подставляется во время компиляции);
- параметры <xacro:property> и <xacro:arg>, значение которых подставляется при компиляции;
- использование условий <xacro:if>, позволяющих включать или исключать целые блоки разметки.

Поиск и исправление ошибок в Xacro.

1. сгенерируйте urdf из xacro в файл my_robot.xacro

```
rosrun xacro xacro my_robot.xacro >
my_robot.urdf

```

2. затем воспользуйтесь инструментом check_urdf и выполните:

```
check_urdf my_robot.urdf

```

При возникновении ошибок в xacro ошибки не отображаются. Чтобы это исправить, укажите путь к xacro в файле запуска следующим способом, тогда ошибки компиляции xacro будут выводиться в терминале.

```
<param name="robot_description" command="$(
 find xacro)/xacro '$(find
 my_robot_description)/urdf/my_robot.xacro'"
 />
```

1.13 Rviz – инструмент для визуализации данных

RViz является универсальным инструментом для визуализации данных различного рода. В частности, его удобно использовать для отображения измерений датчиков различных типов. Для каждого типа данных в Rviz имеется плагин, который необходимо включить и указать требуемые параметры, чтобы визуализация отобразилась. Одной из самых часто используемых плагинов является TF. После добавления данного плагина необходимо указать базовую систему координат (выбор фиксированной глобальной системы координат 1.8). Важно отметить, что система TF использует структуру модели робота, поэтому необходимо выполнить загрузку модели робота в сервер параметров(robot_description) через файл запуска.

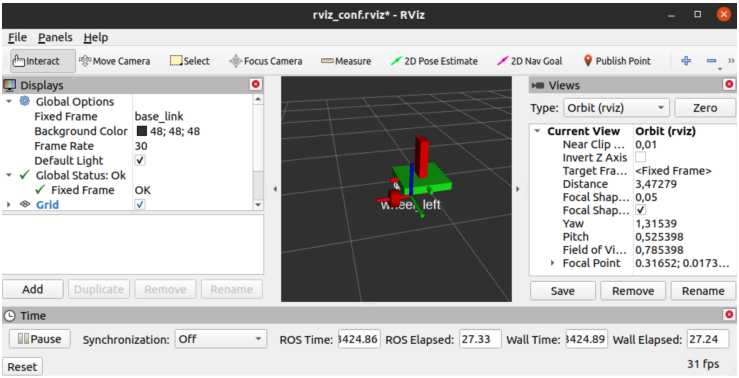

Рисунок 1.8. Окно программы Rviz

Перед запуском визуализации робота в Rviz необходимо запустить две ключевые компоненты ROS: robot_state_publisher и joint_state_publisher.

ROS пакет robot_state_publisher позволяет публиковать состояние робота в tf. Как только состояние будет опубликовано, оно станет доступно всем компонентам системы, которые также используют tf. Пакет принимает значения углов шарниров робота в качестве входных данных и публикует трехмерные положения

звеньев робота (относительно предыдущего звена), используя кинематическую древовидную модель робота из robot_description. Необходимо его использовать в качестве узла ROS (не как библиотеку).

ROS пакет joint_state_publisher является инструментом для публикации значений состояния шарниров, указанных в URDF. Считывает параметр robot_description с сервера параметров, находит все незафиксированные шарниры и публикует сообщение Sensor_msgs/JointState для всех шарниров.

Аналогичный ROS пакет joint_state_publisher_gui позволяет выполнять ровно такие же функции и дополнительно предоставляет графическое окно 1.9 с ползунками для управления шарнирами.

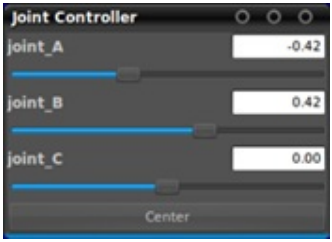

Рисунок 1.9. Графическое окно joint_state_publisher_gui

На рисунке 1.10 представлены компоненты, взаимодействие которых необходимо для корректной визуализации модели робота средствами ROS.

1.14 Gazebo – имитационное моделирование

Gazebo является основным инструментом имитационного моделирования (симулятор) для ROS. Gazebo поставляется вместе с ROS и не требует отдельной установки. Однако вместе с ROS могут работать многие другие симуляторы, такие как CoppeliaSim (V-REP), Unity, Unreal, Godot, Isaac Sim, Webots. Далее рассматривается взаимодействие ROS и Gazebo для разработки и тестирования программного обеспечения для роботов.

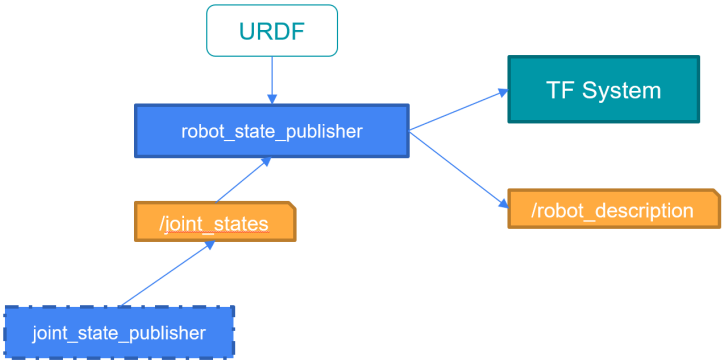

Рисунок 1.10. Компоненты ROS для визуализации модели и структура взаимодействия

Наличие модели робота недостаточно для выполнения симуляции. Симулятору необходимо предоставить описание среды моделирования в формате SDF (Simulation Description Format). Несмотря на название формата, файлы с описанием модели имеют расширение .world.

Основной файл настройки мира (.world) – содержит основные элементы сцены. Например, ссылки на модели. На рисунке 1.11 показан пример добавления трех моделей из локальной базы данных моделей Gazebo.

В Gazebo могут использоваться различные системы расчета моделирования (также известные как "физические движки"), однако по умолчанию доступен только ODE (Open Dynamics Engine). Настройки физического движка также указываются в файле .world. На рисунке 1.12 показан пример с настройками физического движка.

При использовании симулятора исчезает необходимость в пакете joint_state_publisher, так как положения шарниров будут учитываться в имитационной модели робота на стороне симулятора. Однако симулятор не может напрямую взаимодействовать с ROS, все взаимодействие может проходить только посредством специальных плагинов для Gazebo [5].

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://gas_station</uri>
      <name>gas_station</name>
      <pose>-2.0 7.0 0 0 0 0</pose>
    </include>
  </world>
</sdf>
```

Рисунок 1.11. Пример простейшего .world файла для Gazebo

```
<physics type="ode">
  <real_time_update_rate>1000.0</real_time_update_rate>
  <max_step_size>0.001</max_step_size>
  <real_time_factor>1</real_time_factor>
  <ode>
    <solver>
      <type>quick</type>
      <iters>150</iters>
      <precon_iters>0</precon_iters>
      <sor>1.400000</sor>
      <use_dynamic_moi_rescaling>1</use_dynamic_moi_rescaling>
    </solver>
    <constraints>
      <cfm>0.00001</cfm>
      <erp>0.2</erp>
      <contact_max_correcting_vel>2000.000000</contact_max_correcting_vel>
      <contact_surface_layer>0.01000</contact_surface_layer>
    </constraints>
  </ode>
```

Рисунок 1.12. Пример простейшего .world файла для Gazebo

На рисунке 1.13 представлены компоненты и их взаимодействие при использовании симулятора, которые необходимо для корректной работы имитационного моделирования в Gazebo при полной интеграции с возможностями ROS.

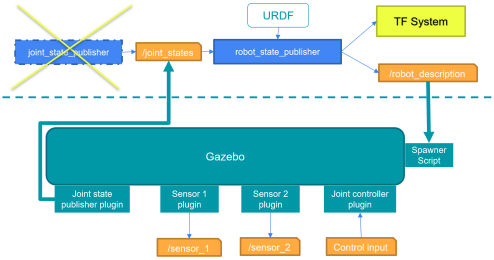

Рисунок 1.13. Компоненты ROS и Gazebo для имитационного моделирования

2 Практическая работа №1. Особенности работы в Linux и установка ROS

Цель работы

- Изучение основ Linux, необходимых для работы с ROS
- Установка и базовая настройка ROS
- Знакомство с базовыми возможностями ROS по обмену данными

В данной работе необходимо пользоваться теоретическими материалами и инструкциями по установке необходимого программного обеспечения.

2.1 Подготовка операционной системы

Необходимо установить операционную систему Ubuntu (либо как вторую операционную систему, либо установить WSL [6]), следуя инструкциям, приведенным в приложении А (14).

2.2 Установка редактора исходного кода (IDE)

Чтобы установить редактор VS Code [7], выполните на Ubuntu:

```
sudo snap install code --classic
```

Если вы работаете на операционной системе Windows, необходимо скачать с официального сайта программу Visual Studio Code и установить, убедившись, что выполняете операцию вне WSL и Docker контейнера.

В VS code имеются полезные расширения, которые облегчат работу (их желательно установить): Python, Cmake, ROS.

2.3 Первый запуск ROS

В данном разделе необходимо научиться работать с базовыми компонентами ROS и попробовать работу с уже установленным ROS-пакетом, в который невозможно вносить правки.

ROS Master

Для работы ROS необходим ROS Master, который запускается утилитой roscore. Если roscore не установлен, установить можно следующей командой (в версии desktop-full уже установлен roscore):

```
sudo apt install python3-roslaunch
```

Перед первым запуском полезно будет удостовериться в том, что установка ROS прошла успешно. Для этого можно воспользоваться следующей командой, которая выводит все важные параметры из окружения линукс по заданному ключевому слову "ROS":

```
printenv | grep ROS
```

Следующим шагом откройте файл /.bashrc в текстовом редакторе и проверьте его содержимое. В файле должен быть прописан путь к установленной версии ROS, как показано ниже.

```
source /opt/ros/noetic/setup.bash
```

Запуск ROS Master

Запустить ROS Master можно при помощи команды roscore в отдельном окне терминала. Запустите **roscore**, выполнив команду:

```
roscore
```

Запуск "publisher"

Откройте новый терминал (не закрываем предыдущий, в котором запущен `roscore`) и попробуйте запустить узел, публикующий сообщения (publisher), используя уже установленный пакет `rospy_tutorials`:

```
rosrun rospy_tutorials talker
```

Вы должны увидеть, как `talker` публикует сообщения с содержанием “Hello world”.

Запуск "listener"

Откройте еще один терминал и запустите ноду `listener` из аналогичного пакета, воспользовавшись следующей командой:

```
rosrun rospy_tutorials listener
```

В результате у вас должно быть открыто три окна (три терминала), и вы должны увидеть, как `listener` считывает сообщения, публикуемые `talker`. Если вы остановите `talker`, нажав курсором на окно, и затем клавиши CTRL + C, то `listener` перестанет получать сообщения.

2.4 Визуализация вычислительного графа

В данном разделе необходимо научиться исследовать готовый (написанный другими разработчиками) ROS пакет.

Используйте утилиту `rqt_graph`, чтобы отобразить вычислительный граф:

```
rqt_graph
```

Ответьте на следующие вопросы:

- Через какой топик происходит обмен данными?
- Посмотрите на вычислительный граф и определите название узлов, которые обмениваются данными.

Запуск симуляции черепахи

Остановите все ранее запущенные файлы (кроме `roscore`) и запустите симуляцию черепахи из установленного по умолчанию пакета turtlesim. Для этого выполните команду:

```
rosrun turtlesim turtlesim_node
```

Снова запустите `rqt_graph` и проверьте, какие новые узлы и топики появились в вычислительном графе.

Отправка управляющих команд

Чтобы черепашка двигалась, необходимо отправить “правильное” сообщение в “правильный” топик. Найдите названия “правильных” топиков с помощью ROS команд:

```
rostopic list
```

Тип сообщения в топике "topic_name" определяется командой:

```
rostopic type /topic_name
```

Проверьте, за что отвечает каждый топик, подставляя вместо topic_name названия топиков из списка топиков, который выдала команда `rostopic list`.

Для просмотра структуры сообщения с названием типа 'msg_type':

```
1    rosmsg show msg_type
2
```

Публикация сообщений

Сообщение для управления можно отправить из терминала (консоли):

```
1    rostopic pub /turtle1/cmd_vel
     geometry_msgs/Twist '{linear: {x: 2.0, y:
     0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z:
     1.8}}'
2
```

Управление черепахой с клавиатуры

Для управления черепахой с клавиатуры можно воспользоваться узлом

`turtle_teleop_key`, который имеется в ROS пакете `turtlesim`

Запустите:

```
1    rosrun turtlesim turtle_teleop_key
2
```

Теперь можно управлять черепахой с клавиатуры.

Просмотр данных

Выведите в терминале данные по положении черепахи, которые приходят в топик `/turtle1/pose`:

```
rostopic echo /turtle1/pose
```

Проверьте частоту публикации сообщений:

```
rostopic hz /turtle1/pose
```

Выведите данные из топика в виде графика (пример построения графика):

```
rosrun rqt_plot rqt_plot /turtle1/pose
```

2.5 Создание рабочего окружения и запуск ROS пакета

1. Знакомство с процедурой написания программного кода и его компиляции.
2. Изучение базовых возможностей ROS по обмену данными на примере управления моделью черепахи.

2.6 Установка catkin

Для работы с пользовательскими пакетами, которые разработчик может редактировать, необходимо установить утилиту catkin:

```
sudo apt-get install python3-catkin-tools
```

При возникновении ошибок в результате выполнения выше указанной команды, следуйте инструкции на официальном сайте catkin [2].

В дальнейшем будет важно помнить: Наличие папки `src` внутри рабочего окружения catkin является обязательной для инициализации, в дальнейшем работа должна вестись именно внутри этой папки!

2.7 Создание catkin workspace

Создайте пользовательское окруженеие catkin в любом месте и запомните расположение, в дальнейшем вся работа с ROS пакетами будет осуществляться внутри этого окружения.

Следующие команды создают папку workspace, внутри которой еще одна папка src, затем инициализируют папку workspace в качестве catkin workspace и запускают пересборку всех имеющихся в окружении пакетов.

```
mkdir workspace
cd workspace
mkdir src
catkin init
catkin build
```

После каждой сборки пакетов catkin необходимо выполнить (обновление переменных окружения):

```
source devel/setup.bash
```

2.8 Создание ROS пакета

В рамках данной практической работы предлагается скачать уже готовый ROS пакет с минимальным исходным кодом из

репозитория [8], для этого необходимо выйти из папки workspace и склонировать весь репозиторий [8] воспользовавшись командой (в дальнейшем мы будем использовать много готовых фрагментов и пакетов из этой папки):

```
cd ../
git clone https://gitlab.com/likerobotics/
 ros-course-itmo.git
```

В скачанных файлах необходимо найти и скопировать папку `my_robot_controller` из `task_1` в свое окружение catkin в папку src (пример команды для копирования на linux: `cp -r /roscourseitmo/task_1/my_robot_controller /workspace/src/my_robot_controller`).

2.9 Исследование пакета

Данный пакет содержит:

1. `scripts` - папка с Python файлами
2. `CMakeLists.txt` - инструкции для системы сборки программного кода
3. `package.xml` - свойства пакета, такие как название, версия, авторы и зависимости от других пакетов.

Перед началом разработки программного кода с использованием ROS необходимо вспомнить два фундаментальных понятиях, часто называемых `Publisher` и `Subscriber`. Publisher - структура данных, имеющаяся в библиотеке ROS для публикации сообщений в соответствии с правилами ROS. Часто называют Publisher сам узел, который содержит вышеупомянутую структуру данных, если она публикует сообщения. Узел в общем случае может содержать как Publisher, так и Subscriber одновременно. Однако для простоты далее мы рассмотрим два узла, один из них Publisher, а другой Subscriber.

2.10 Создание Publisher

Далее создадим первый свой исполняемый файл в папке `scripts`, для этого перейдите внутрь папки и создайте файл с

названием my_publisher.py, который будет содержать следующий программный код:

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Данный пример публикует сообщения типа String в топик с названием chatter.

Дополните его, чтобы одновременно с этим публиковалось второе сообщение типа Twist в топик cmd_vel.

Добавьте импорт Twist по аналогии со String и в основном цикле добавьте следующее:

```
  msg = Twist()
  msg.linear.x = 1
  pub.publish(msg)
```

2.11 Создание Subscriber

Необходимо создать файл my_subscriber.py со следующим содержанием:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I
    heard %s", data.data)

def listener():

    rospy.init_node('listener', anonymous=True
    )
    rospy.Subscriber("chatter", String,
    callback)

    rospy.spin()

if __name__ == '__main__':
    listener()
```

Выше указанный пример только принимает сообщение из топика chatter и выводит результат в терминал.

Не забудьте сделать файлы исполняемыми перед запуском. Запустить Publisher и Subscriber в разных терминалах при помощи команды `rosrun`.

2.11.1 Вопросы для самопроверки

- Какую функцию выполняет ROS Master?
- Как называется утилита для визуализации вычислительного графа в ROS?
- Возможно ли редактирование ROS пакета, если его установка выполнена через apt ?
- В какой папке необходимо располагать программный код на языке `Python`?

3 Практическая работа №2. Создание ROS-пакета и пользовательских сообщений

Цель работы:

- Изучение процедуры создания пользовательского программного пакета
- Изучение процедуры создания пользовательского типа сообщений с использованием базовых типов данных
- Научиться создавать файлы запуска

3.1 Создание ROS пакета

Чтобы создать свой пакет, вам понадобится ранее созданное рабочее окружение `catkin` располагающееся в папке workspace. **Важно!** Всегда работаем в папке `src`, и пакеты создаем внутри неё.

Создайте новый пакет с названием `my_super_robot_controller`, выполнив следующие команды:

```
1  cd src
2  catkin_create_pkg my_super_robot_controller
    rospy std_msgs
3  catkin build
```

После создания пакета необходимо проверить, что внутри ROS пакета находится папка `src`, а также файлы `CMakeLists.txt` и `package.xml`.

Примечание: Папка `scripts` внутри ROS пакета не создается автоматически, её нужно создать вручную:

```
1  mkdir scripts
```

3.2 Создание ноды с паблишером и сабскрайбером

Создайте скрипт в папке `scripts` и сделайте его исполняемым файлом (новые файлы в Linux по умолчанию не являются исполняемыми):

```
touch scripts/super_node.py
chmod +x scripts/super_node.py
```

Откройте файл super_node.py для редактирования и добавьте следующий программный код:

```
#!/usr/bin/env python3

import rospy
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist

def pose_callback(pose):
    cmd = Twist()
    if pose.x > 9.0 or pose.x < 2.0:
        cmd.linear.x = 1.0
        cmd.angular.z = 1.4
    else:
        cmd.linear.x = 2.0
        cmd.angular.z = 0.0

    publisher.publish(cmd)
    rospy.loginfo("Super node sending new msg
    ...")

if __name__ == '__main__':
    rospy.init_node("turtle_super_node")
    rospy.loginfo("Node started...")
    subscriber = rospy.Subscriber("/turtle1/
    pose", Pose, callback=pose_callback)
    publisher = rospy.Publisher("/turtle1/
    cmd_vel", Twist, queue_size=10)
    rospy.spin()
```

Данный пример реализовывает простейший релейный регулятор для управления моделью черепахи (моделирование черепахи должно быть запущего командой `rosrun turtlesim turtlesim_node`).

Запустите только что созданный файл с помощью команды rosrun, как показано ниже.

```
rosrun my_super_robot_controller super_node.
 py
```

В результате вы должны увидеть сообщение "Super node sending new msg...". А если моделирование черепахи запущено, то должны увидеть движение модели черепахи.

Чтобы остановить запущенную программу, воспользуйтесь сочетанием клавиш CTRL+C.

3.3 Пользовательский тип сообщений

Ранее вы использовали только уже созданные типы сообщений, однако ROS предоставляет возможность создавать сообщения, имеющие произвольную структуру и любые поля. Такие типы сообщений называются пользовательскими. Любые типы сообщений в ROS существуют внутри определённого ROS-пакета, и название типа сообщения включает название пакета. Например, самые часто используемые типы сообщений относятся к пакету `std_msgs`.

3.3.1 Создание структуры сообщений

Внутри ROS пакета создайте папку msg для хранения структуры сообщений:

```
mkdir msg
```

Создайте файл структуры сообщения с названием Num.msg и содержанием "int64 num", выполнив следующую команду:

```
echo "int64 num" > msg/Num.msg
```

Отредактируйте файл, изменив поля в соответствии с примером ниже:

```
string first_name
string last_name
uint8 age
uint32 score
```

На этом этапе только что созданный пользовательский тип сообщений будет недоступен для использования в вашем ROS пакете.

3.3.2 Настройка сборки сообщений

Чтобы сборка новых типов сообщений выполнялась каждый раз при вызове команды catkin build, необходимо подключить два стандартных пакета `message_generation` и `message_runtime`.

Отредактируйте `package.xml`, добавляя строки:

```
<build_depend>message_generation</
 build_depend>
<exec_depend>message_runtime</exec_depend>
```

Также отредактируйте `CMakeLists.txt` добавив зависимости от `message_generation` и `message_runtime`, как показано ниже:

```
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
message_generation
)

catkin_package(
CATKIN_DEPENDS message_runtime
)
```

Добавьте название файла со структурой сообщения в add_message_files:

```
add_message_files(
FILES
Num.msg
)
```

Разрешите генерацию сообщений в generate_messages:

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

Выполните повторную сборку пакета:

```
catkin build
```

Не забудьте, что после каждой сборки необходимо выполнить source (2.7), как было показано в Практической работе №1 (в дальнейшем это действие будет само собой подразумеваться).

Проверьте доступность созданного типа сообщений в ROS и отобразите содержимое созданной структуры сообщения:

```
rosmsg show my_super_robot_controller/Num
```

3.3.3 Задание для самостоятельного выполнения

1. Изменить созданный тип сообщений, чтобы он содержал только координаты `x, y`.

2. Написать программный код для узла, подписывающегося на топик `Pose` из `turtlesim` и публикующее `position2d` сообщения с координатами `x, y`.

3. Проверить результат с помощью команды: `rostopic echo /position2d`.

3.4 Запуск нескольких узлов одновременно

Для одновременного запуска нескольких узлов используются файлы запуска `.launch`.

Внутри ROS пакета создайте папку для хранения файлов запуска:

```
mkdir launch
```

Создайте файл запуска:

```
touch launch/my_setup.launch
```

Отредактируйте файл, добавив следующее содержание:

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <node name="my_super_robot_controller" pkg
   ="my_super_robot_controller" type="
   super_node.py"/>
</launch>
```

Выполните файл запуска:

```
roslaunch my_super_robot_controller my_setup
  .launch
```

В результате вы должны увидеть узлы, которые создает super_node.py.

В файлах запуска также можно указать пространство имен, внутри которого будет запускаться узел. В следующем примере super_node.py запускается внутри пространства имен с названием ns1, и таким образов все относительные имена топиков и узлов будут начинаться с /ns1/*.

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <group ns="ns1">
    <node name="controller" pkg="
   my_super_robot_controller" type="controller
   .py" output="screen"></node>
  </group>

  <group ns="ns2">
    <node name="controller" pkg="
   my_super_robot_controller" type="controller
   .py" output="screen"></node>
  </group>
</launch>
```

3.4.1 Задание для самостоятельного выполнения

Отредактируйте `my_setup.launch`, чтобы одновременно с управляющей программой запускалась имитационная модель черепахи из пакета `turtlesim`.

3.4.2 Вопросы для самопроверки

- Достаточно ли создать файл с расширением **.msg**, чтобы новый тип сообщений стал доступен в ROS?
- Какой командой необходимо воспользоваться для выполнения файла запуска?

4 Практическая работа №3. Создание моделей роботов и объектов окружения

Цель работы

- Знакомство с основами построения модели робота
- Создание первой urdf модели робота
- Изучение инструментов для визуализации модели робота

Целью данной работы является закрепление теоретического материала по созданию моделей роботов с использованием ROS и подготовка к лабораторной работе №3.

В данной работе необходимо пользоваться теоретическими материалами. Чтобы создать первую модель, нам понадобится пакет в системе catkin. В дальнейшем вы будете пользоваться некоторыми вспомогательными инструментами Visual Studio Code (VS Code), поэтому рекомендуется установка программы.

4.1 Создание первого urdf файла

В данной практичческой работе используются материалы из официальной документации ROS [9].

1. Создайте новый пакет в рабочем пространстве catkin

```
cd catkin_ws/src
catkin_create_pkg my_robot_model rospy
 std_msgs roscpp
catkin build
```

2. Внутри ROS пакета создайте подпапки scripts, launch и urdf.

```
cd my_robot_model
mkdir scripts launch urdf
```

3. Создайте свою первую модель в формате urdf, для этого перейдите в папку urdf и создайте xml файл с расширением .urdf

```
cd urdf
code my_first_model.urdf
```

Замечание! Убедитесь, что у вас в Visual Studio Code установлено расширение ROS от **Microsoft. Также вам необходимо установить расширение ROS Snippets(Liews Wuttipat)** . Запустить визуализатор можно при помощи CTRL+Shift+P и выбрать из списка URDF Viewer. Также рекомендуется запускать VS Code из рабочего окружения catkin, например:

```
cd catkin_ws
code .
```

Затем необходимо убедиться, что VS Code определил ваш проект как ROS (внизу окна слева должна отображаться версия ROS), это может быть необходимо для корректной работы расширений в VS Code.

4. Создание модели

   В качестве тренировки предлагается создать модель мобильного робота с дифференциальным приводом, показанную на рисунке 4.1.

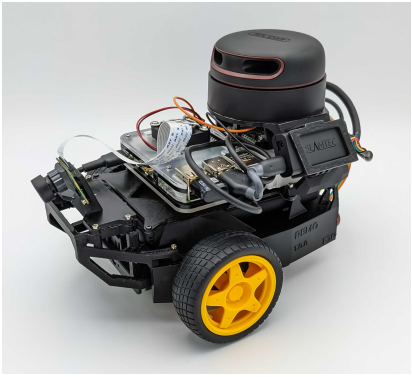

Рисунок 4.1. Мобильный робот с дифференциальным приводом

В первую очередь вы должны определить, какие элементы робота имеют относительную подвижность. Как можно видеть из изображения, два колеса являются приводными и имеют вращательную степень подвижности вокруг одной оси. Третье колесо (его не видно на изображении) имеет пассивно-сферический тип, т.е. может свободно вращаться. Также необходимо определить, какое звено является базовым, базовое звено - тот элемент, к которому все остальные звенья присоединяются при помощи шарниров, образуя при этом древовидную структуру. Важно отметить, что замкнутая кинематика в явном виде не поддерживается в разметке URDF.

- Создайте первый элемент - базовое звено, это будет корпус платформы для данного примера.

```
<?xml version='1.0'?>
<robot name="robot" xmlns:xacro="http://www.ros.org/wiki/xacro">
 <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
 <link name='base_link'>
     <visual name='base_visual'>
         <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
         <geometry>
             <box size="0.2 0.6 0.6"/>
         </geometry>
     <!-- <material name="">
     <color rgb="1.0 0.0 0.0">
     </material> -->
     </visual>
 <!-- <inertial>
 <origin xyz="0 0 0.5" rpy="0 0 0" />
 <mass value="1"/>
 <inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100" />
 </inertial> -->
 <!-- <collision>
```

```
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
    <cylinder radius="1" length="0.5"/
    >
    </geometry>
    </collision> -->
    </link>
<link name='wheel_left'>
    <visual name='wheel_left_visual'>
        <origin xyz="0.2 0 0" rpy=" 0
    0 0"/>
        <geometry>
            <cylinder length="0.7"
    radius="0.1"/>
        </geometry>
        <material name="gray"/>
    </visual>
</link>
<joint type="fixed" name="wheel_joint"
    >
    <origin xyz="0.15 0.0 0" rpy="0 0
    0"/>
    <child link="wheel_left"/>
    <parent link="base_link"/>
</joint>
</robot>
```

- Теперь внесите необходимые изменения.

  - У шарнира, который соединяет колесо с базой необходимо установить type=“continuous”, поскольку мы не хотим задавать какие либо огрничения на его вращение. Также у шарниров типа “continuous” необходимо задать ось вращения, для этого добавьте свойство axis.

```
    <axis xyz="1.0 0.0 0.0"/>

```

- Необходимо скопировать и переименовать звено wheel_left в звено wheel_right

- После чего необходимо настроить положение и ориентацию колес в соответствии с изображением робота.

5. Поскольку в модели имеются шарниры, информация о состоянии этих элементов должны быть доступна в ROS, для этого необходимы два пакета: **joint_state_publisher** и **robot_state_publisher**, сделайте это в launch файле путем добавления следующих строк.

```
<node name="joint_state_publisher" pkg="
 joint_state_publisher" type="
 joint_state_publisher" />
<node name="robot_state_publisher" pkg="
 robot_state_publisher" type="
 robot_state_publisher" />
```

6. Теперь необходимо открыть модель робота в rviz.

   Rviz можно открыть из терминала, воспользовавшись командой

```
  rviz

```

   либо указать автоматический запуск в файле запуска. Однако, чтобы модель робота была доступна в Rviz, необходимо в файле запуска выполнить загрузку модели робота в robot_description сервера парметров ROS.

   Далее показан пример с загрузкой модели робота в сервер парметров и запуск rviz:

```
<launch>
<arg name="gui" default="False" />
<param name="robot_description"textfile=
 "$(find my_robot_model)/urdf/
 my_first_model.urdf" />
<param name="use_gui" value="$(arg gui)"
 />
<node name="rviz" pkg="rviz" type="rviz"
  />
```

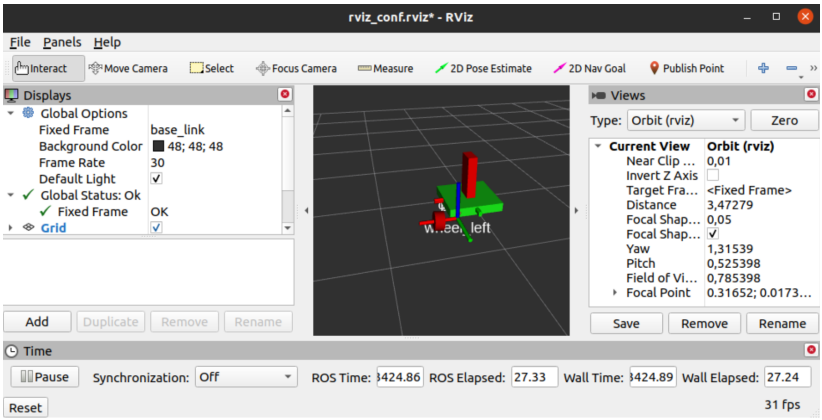

Рисунок 4.2. Окно Rviz.

```
6   </launch>
```

Добавьте в файл запуска выше указанную разметку и запустите launch файл.

```
1   roslaunch my_robot_model robot.launch
```

В результате должен запуститься Rviz, но модель робота не отобразится. Далее рассмотрим причины и способы, как это исправить.

7. Настройка визуализации в Rviz После подключения пакетов и запуска Rviz, вы можете визуализировать все системы координат, имеющиеся в модели, для этого нажмите в Rviz следующие кнопки: **Add** a new display и выберите **TF**. Также необходимо в меню **Global Options** в пункте **Fixed Frame** выбрать base_link (система отсчета). Чтобы отобразить саму модель, необходимо нажать кнопку "Add" и выбрать "Robot-Model".

8. Управление шарниром Запустите графический итерфейс для joint_state_publisher, чтобы попробовать повращать шарниры колеса, но сначала нужно установить данный пакет

под названием joint_state_publisher_gui, воспользовавшись командой:

```
sudo apt install ros-noetic-joint-state-publisher-gui
```

Подключение это можно сделать и через терминал, но вам рекомендуется сделать это через launch файл, указав дополнительную строку.

```
<node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />
```

Также необходимо закомментировать joint_state_publisher так как одновременно два пакета не должны публиковать в одни и те же топики.

9. Проверьте корректность расположения осей вращения и звеньев мобильного робота и внесите необходимые изменения, чтобы модель соответствовала роботу, изображенному на картинке выше.

4.2 Создание модели в xacro

Чтобы ваша модель была более гибкой и универсальной, необходимо воспользоваться возможностями XACRO.

1. Создайте новый файл под названием diff_robot_model.urdf.xacro, скопируйте туда описание модели и укажите дополнительный параметр тега robot, как показано далее:

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="my\_robot">

```

2. Также для работы xacro вам необходимо изменить launch файл, интересует строка загрузки robot_description в сервере параметров, его необходимо исправить как показано ниже

```
    <param name="robot_description"
    command="xacro '$(find my_robot_model)/
    urdf/diff_robot_model.urdf.xacro'" />

```

3. Создайте также свойство в файле

```
    <xacro:property name="side" value="
    right" />

```

Теперь вместо значения вы можете указать $(side), чтобы обеспечить быстрое редактирование (как у звена, так и у шарнира).

Например:

```
    <link name='wheel_${side}'>
    <visual name='wheel_\${side}_visual'>

```

Задание на 5 минут: Создайте свойство с наванием **wheel_diameter** и в описании модели замените диаметры цилиндров на значение свойства. Проверьте в RViz результат.

4. Создайте новый файл с названием materials.xacro и задайте внутри цвета, которые ыы хотите использовать в модели.

```
<?xml version="1.0"?>
<robot>

  <material name="black">
    <color rgba="0.0 0.0 0.0 1.0"/>
  </material>

  <material name="blue">
    <color rgba="0.0 0.0 0.8 1.0"/>
```

```
  </material>

  <material name="green">
    <color rgba="${30/255} ${255/255}
    ${30/255} 1.0"/>
  </material>

  <material name="grey">
    <color rgba="0.2 0.2 0.2 1.0"/>
  </material>

  <material name="orange">
    <color rgba="${255/255} ${108/255}
    ${10/255} 1.0"/>
  </material>

  <material name="brown">
    <color rgba="${222/255} ${207/255}
    ${195/255} 1.0"/>
  </material>

  <material name="red">
    <color rgba="0.8 0.0 0.0 1.0"/>
  </material>

  <material name="white">
    <color rgba="1.0 1.0 1.0 1.0"/>
  </material>

</robot>
```

Теперь в основном файле .xacro вы можете подключить этот файл следующим образом:

```
    <xacro:include filename="\$(find
    my_robot_model)/urdf/materials.xacro

```

И использовать данные цвета для модели робота.

```
    <material name="red"/> <!--inside
    visual tag -->
```

```
2
3
```

5. Использование **xacro:macro** позволяет избегать повторения фрагментов разметки, например, колесо повторяется для левой и правой стороны, при этом в фрагменте меняется только имя элемента и положение. Создадим отдельный файл, в котором будет содержаться описание колеса и будем его копии импортировать с автоматической подстройкой под нужную сторону.

6. Создайте новый файл wheel.xacro со следующим содержанием:

```
<xacro:macro name="wheel" params="side x y z">
    <link name='wheel_${side}'>
    <visual name='wheel_${side}_visual'>
    <origin xyz="0.2 0 0" rpy=" 1.5 0.0 1.5"/>
    <geometry>
    <cylinder length="0.1" radius="0.1"/>
    </geometry>
    <material name="red"/>
    </visual>
    </link>
    <joint name="wheel_${side}_joint" type="continuous">
    <origin xyz="${x} ${y} ${z}" rpy="0 0 0"/>
    <child link="wheel_${side}"/>
    <parent link="base_link"/>
    </joint>
</xacro:macro>
```

7. И в файле модели укажем вызов macro с необходимыми нам параметрами (парметры очень похожи на свойства, с единственным отличием - их значения нам надо передавать при вызове)

```
    <xacro:wheel side="left" x="0.25" y="
    0.4" z="0.0" />
```

4.3 Использование трехмерных моделей сложной формы

Чтобы ваши модели были более реалистичными, вы можете подключить любой stl файл внутри блока "geometry" следующим образом:

```
<mesh filename="package://my_robot_model/
 models/base_frame.stl" scale="1.0 1.0 1.0"/
 >
```

При этом важно отметить, что путь до файла указывается относительно ROS пакета.

4.4 Первый запуск симуляции

Чтобы смоделировать движение мобильного робота, необходимо, чтобы у каждого звена были блоки **collision** и **inertial**. Если они отсутствуют, то их необходимо добавить. Также массы объектов должны быть больше 0 (рекомендуется не моделировать элементы массой менее 0.01 кг).

Для зупуска Gazebo пропишите следующий исходный код в launch файле:

```
<include file="$(find gazebo_ros)/launch/
 empty_world.launch">
  <arg name="world_name" value="\$(find
 my_robot_model)/worlds/empty.world"/>
  <arg name="paused" value="false"/>
  <arg name="use_sim_time" value="true"/>
  <arg name="gui" value="true"/>
  <arg name="headless" value="false"/>
  <arg name="debug" value="false"/>
</include>
```

```
9   <node name="my_robot_spawn" pkg="gazebo_ros"
      type="spawn_model" output="screen" args="-
     urdf -param robot_description -model
     my_robot_model"/>
10
```

Цвета объектов из urdf не поддерживаются в Gazebo, поэтому вам необходимо указывать специальные теги для симулятора.

4.5 Задание для самостояятельного выполнения

1. Добавить двухзвенный манипулятор к подвижной платформе.

2. Добавить ограничения к шарнирам манипулятора, чтобы он неимел возможности сталкиваться со вторым звеном

4.5.1 Вопросы для самопроверки

- Что такое URDF ?
- Чем отличается XACRO от URDF ?
- Зачем нужен joint_state_publisher в Gazebo?
- Для чего необходим robot_state_publisher?
- Какой тег обязательно должен присутствовать в разметке любого XACRO файла?

5 Практическая работа №4. Создание имитационной модели в Gazebo ROS

Цель работы

- Изучение основ симулятора Gazebo
- Взаимодействие с моделью робота в Gazebo средствами ROS
- Изучение основных ограничений и способов оптимизации моделей для имитационного моделирования

Целью данной работы является закрепление теоретического материала по созданию сцены имитационного моделирования в программе Gazebo с использованием ROS для подготовки к лабораторной работе №4.

В данной работе необходимо пользоваться лекционными материалами. Необходимо настроить модель робота и окружение среды моделирования, затем выполнить запуск имитационного моделирования для управления роботом.

5.1 Создание симуляции робота средствами ROS

1. Создание пакета

   Создайте новый пакет и назовите **my_robot_simulation** выполнив команду:

   ```
   catkin_create_pkg my_robot_simulation
    rospy std_msgs
   ```

2. Создание файла запуска

   Далее необходимо создать файл запуска в папке launch, который будет загружать конфигурационный файл для Rviz, загружать модель робота в сервер парамтеров, включая необходимые пакеты для работы модели в ROS. Файл в результате должен иметь следующее содержание:

```
<launch>
    <arg name="rviz_conf_file" default="$(
   find my_robot_model)/urdf/rviz_conf.
   rviz" />
    <param name="robot_description"
        command="xacro '$(find
   my_robot_model)/urdf/
   diff_robot_model_done.urdf.xacro'" />

    <!-- <node name="joint_state_publisher
   " pkg="joint_state_publisher" type="
   joint_state_publisher"/> -->
    <node name="joint_state_publisher_gui"
    pkg="joint_state_publisher_gui"
        type="joint_state_publisher_gui" /
   >
    <node pkg="robot_state_publisher" type
   ="robot_state_publisher" name="
   robot_state_publisher"
        output="screen">
        <param name="publish_frequency"
   type="double" value="40.0" />
    </node>
    <include file="$(find gazebo_ros)/
   launch/empty_world.launch">
        <arg name="world_name" value="$(
   find my_robot_model)/worlds/empty.world
   " />
        <arg name="paused" value="false" /
   >
        <arg name="use_sim_time" value="
   true" />
        <arg name="gui" value="true" />
        <arg name="headless" value="false"
    />
        <arg name="debug" value="false" />
    </include>
    <node name="my_robot_model_spawn" pkg=
   "gazebo_ros" type="spawn_model" output=
   "screen"
```

```
        args="-urdf -param
    robot_description -model my_robot_model
    " />
     <node name="rviz" pkg="rviz" type="
    rviz" args="-d $(arg rviz_conf_file)" /
    >
</launch>
```

Если вы отредактировали файл запуска, может потребоваться перезапуск roscore (имейте это в виду, если что-то начнет работать не так, как ожидалось, и ошибок тоже не обнаружено).

3. Проверка разметки модели робота и параметров colission и inertial

   Для работы симулятора обязательно должны быть теги colission и inertial (не нулевые массы) у всех звеньев, поэтому в соответствующих файлах модели проверьте и исправьте это при необходимости.

   Далее приведен пример разметки для звена base_link:

```
<link name="base_link">
    <pose>0 0 0.2 0 0 0</pose>
    <visual>
        <origin xyz="0.0 0.0 0" rpy="0.0
   0.0 0.0" />
        <geometry>
            <box size="0.6 0.6 0.2" />
        </geometry>
        <material name="green" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0." rpy="0.0
   0.0 0.0" />
        <geometry>
            <box size="0.6 0.6 0.2" />
        </geometry>
    </collision>
    <inertial>
        <origin xyz="0 0 0." rpy="0 0 0" /
    >
```

```
        <mass value="0.1" />
        <inertia ixx="100" ixy="0" ixz="0"
    iyy="100" iyz="0" izz="100" />
    </inertial>
</link>
```

4. настройка парметров тега inertial

   Правильная настройка инерционных характеристик материала является важной задачей, от которой будет зависеть качество симуляции. Вам необходимо реализовать способ, который упростит работу. Наиболее часто используемые типы примитивов:

   - цилиндр
   - кубоид
   - сфера

   Как известно из физики твердого тела, инерционные характристики зависят только от формы и плотности материала (однородная масса), поэтому вы можете написать макросы, которыми можно пользоваться для автоматического расчета характеристик.

   Файл с макросами можно скачать из репозитория курса **task5/urdf/inertial.usdf.xacro**, либо скопировать из следующего листинга:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/
   wiki/xacro">
<!--
    EXAMPLE OF USE IN EXTERNAL FILE
<xacro:inertial_box mass="0.5" x="0.5" y="
   0.25" z="0.15">
            <origin xyz="0 0 0.075" rpy="0
    0 0"/>
        </xacro:inertial_box>
         -->
    <xacro:macro name="inertial_sphere"
   params="mass radius *origin">
```

```
        <inertial>
            <xacro:insert_block name="
origin"/>
            <mass value="${mass}" />
            <inertia ixx="${(2/5) * mass *
 (radius*radius)}" ixy="0.0" ixz="0.0"
                     iyy="${(2/5) * mass *
(radius*radius)}" iyz="0.0"
                     izz="${(2/5) * mass *
(radius*radius)}" />
        </inertial>
    </xacro:macro>


    <xacro:macro name="inertial_box"
params="mass x y z *origin">
        <inertial>
            <xacro:insert_block name="
origin"/>
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass
* (y*y+z*z)}" ixy="0.0" ixz="0.0"
                     iyy="${(1/12) * mass *
 (x*x+z*z)}" iyz="0.0"
                     izz="${(1/12) * mass *
 (x*x+y*y)}" />
        </inertial>
    </xacro:macro>


    <xacro:macro name="inertial_cylinder"
params="mass length radius *origin">
        <inertial>
            <xacro:insert_block name="
origin"/>
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass
* (3*radius*radius + length*length)}"
ixy="0.0" ixz="0.0"
```

```
                    iyy="${(1/12) * mass *
    (3*radius*radius + length*length)}"
    iyz="0.0"
                    izz="${(1/2) * mass *
    (radius*radius)}" />
        </inertial>
    </xacro:macro>
</robot>
```

Далее приведен пример использования макроса (не забудьте подключить макрос перед использованием).

```
<link name="left_wheel">
    <xacro:inertial_cylinder mass="0.1"
    length="0.05" radius="0.05">
        <origin xyz="0 0 0" rpy="0 0 0" />
    </xacro:inertial_cylinder>
</link>
```

5. настройка свойств для Gazebo

Теперь необходимо указать важные свойства для работы симулятора, начните с указания цветов для Gazebo. Создайте файл robot_diff.gazebo, содержащий следующий программный код:

```
<?xml version="1.0"?>
<robot>
    <!-- SET COLORS FOR GAZEBO -->
    <gazebo reference="base_link">
        <material>Gazebo/Green</material>
    </gazebo>
    <gazebo reference="demo_link">
        <material>Gazebo/Red</material>
    </gazebo>
</robot>
```

Чтобы вы могли управлять роботом в симуляторе, необходимо подключить плагин, сделайте это в том же файле robot_diff.gazebo путем добавления следующего программного кода:

```
<gazebo>
    <plugin name="
   differential_drive_controller" filename
   ="libgazebo_ros_diff_drive.so">
        <legacyMode>false</legacyMode>
        <alwaysOn>true</alwaysOn>
        <updateRate>30</updateRate>
        <leftJoint>wheel_left_joint</
   leftJoint>
        <rightJoint>wheel_right_joint</
   rightJoint>
        <wheelSeparation>0.3</
   wheelSeparation>
        <wheelDiameter>0.21</wheelDiameter
   >
        <wheelTorque>20</wheelTorque>
        <commandTopic>cmd_vel</
   commandTopic>
        <odometryTopic>odom</odometryTopic
   >
        <odometryFrame>odom</odometryFrame
   >
        <robotBaseFrame>base_link</
   robotBaseFrame>
    </plugin>
</gazebo>
```

Убедитесь, что параметры плагина были настроены корректно!

6. создание "world" файла

   Теперь необходимо создать ваш базовый файл мира .world. Сделать это можно 2 способами:

   (a) Запустить Gazebo и сохранить пустую сцену через графический интерфейс симулятора.

   (b) Создать файл внутри ROS-пакета (в папке worlds) со следующим содержанием:

```
<sdf version="1.4">
```

```
<world name="default">

  <scene>
    <ambient>0.4 0.4 0.4 1</ambient>
    <background>0.7 0.7 0.7 1</
 background>
    <shadows>true</shadows>
  </scene>

  <!-- A global light source -->
  <include>
    <uri>model://sun</uri>
  </include>

  <!-- A ground plane -->
  <include>
    <uri>model://ground_plane</uri>
  </include>

  <gravity>0 0 -9.8</gravity>

  <physics type="ode">
    <real_time_update_rate>1000.0</
 real_time_update_rate>
    <max_step_size>0.001</
 max_step_size>
    <real_time_factor>1</
 real_time_factor>
    <ode>
      <solver>
        <type>quick</type>
        <iters>150</iters>
        <precon_iters>0</
 precon_iters>
        <sor>1.400000</sor>
        <use_dynamic_moi_rescaling>1
 </use_dynamic_moi_rescaling>
      </solver>
      <constraints>
        <cfm>0.00001</cfm>
        <erp>0.2</erp>
```

```
            <contact_max_correcting_vel>
    2000.000000</
    contact_max_correcting_vel>
            <contact_surface_layer>
    0.01000</contact_surface_layer>
         </constraints>
       </ode>
     </physics>
   </world>

</sdf>
```

Если все сделано правильно, то после запуска launch файла у вас запустится симулятор Gazebo.

7. запуск симулятора

   Теперь вы можете запустить симулятор и проверить корректность работы, для этого воспользуйтесь командой:

```
1  roslaunch my_robot_simulation sim.launch
2
```

   Обратите внимание, что управлять роботом при помощи robot_joint_state_publisher у вас не получится. Интерфейс для управления роботом предоставляет плагин, который вы подключили. Соответственно, после запуска симуляции у вас появится топик /cmd_vel, в который вы и можете отправлять команды для управления.

8. доработка модели робота У вашего робота всего 2 колеса, таким неустойчивым роботом сложно управлять, поэтому вы можете сделать третье колесо, только в этот раз другого типа (используя широкие возможности симулятора). Третье колесо будет сферическим элементом, которое создает минимальное трение о поверхность пола. *Обратите внимание, здесь выполняется упрощенное моделирование пассивного сферического колеса.*

   Добавьте к описанию base_link следующую разметку:

```
<collision name='caster_collision'>
```

```
    <origin xyz="0.0 -0.25 -0.11" rpy="0.0
    0.0 0.0" />
    <geometry>
        <sphere radius="0.1" />
    </geometry>
    <surface>
        <friction>
            <ode>
                <mu>0</mu>
                <mu2>0</mu2>
                <slip1>1.0</slip1>
                <slip2>1.0</slip2>
            </ode>
        </friction>
    </surface>
</collision>
<visual name='caster_visual'>
    <origin xyz="0.0 -0.25 -0.11" rpy="0.0
    0.0 0.0" />
    <geometry>
        <sphere radius="0.1" />
    </geometry>
    <material name="blue" />
</visual>
```

9. Дополнительные элементы моделирования

   Одним из недостатков моделей является их "идеализация". Моделирование трения в шарнирах (жесткость и демпфирование) часто не учитывается в модели. Также очень важно установить ограничения на момент и скорость в шарнирах, как это показано в следующем примере.

```
1    <limit effort="100" velocity="100"/>
2    <joint_properties damping="0.0"
    friction="0.0"/>
3
```

10. Добавление объектов окружения из библиотеки объектов Gazebo Библиотеки объектов Gazebo содержит много полезных моделей, которые могут быть добавлены в

симуляцию достаточно легко. Примеры таких добавлений можно найти в файле empty.world

11. Управление с учетом кинематической модели робота

    Установите пакет для управления платформой в симуляторе teleop_twist_keyboard, и у вас появится возможность управлять роботом посредством нажатия соответствующих клавиш на клавиатуре для задания направления движения.

    Альтернативным способом управления является отправка соответствующих сообщений в топик /cmd_vel средствами ROS.

```
1    rostipic pub /cmd_vel ***
2
```

5.1.1 Задание для самостоятельного выполнения

1. Создать сцену с объектами окружения (стандартные из Газебо), сохранить в файл с новым названием и настроить подгрузку новой сцены в файле запуска.

2. Написать скрипт, который будет отправлять сообщения в /cmd_vel (любые сообщения)

3. При вызове метода "spawn" (добавление робота в симуляцию) необходимо задать координаты x y z как аргументы

4. Добавить в симуляцию несколько одинаковых роботов одновременно в разных координатах сцены

5.1.2 Вопросы для самопроверки

- Какой формат разметки используется в файлах с расширением .world
- От чего зависит качество симуляции вашего робота?
- Какие дополнительные требования предъявляет к URDF модели Gazebo?

- Совпадает ли симуляционное время в Gazebo с системным временем?
- Что такое плагины Gazebo?
- Что делает spawn (спавнер)?

6 Практическая работа №5. Создание регулятора для роботов произвольной кинематики на ros_control

Цель работы

- Установка дополнительных пакетов и добавление зависимостей ros_control.
- Создание PID-регулятора для модели робота с произвольной кинематикой.

Создание пакета Создайте новый пакет с названием `my_robot_simulation_control`:

```
catkin_create_pkg
 my_robot_simulation_control roscpp rospy
 std_msgs
```

Создание файлов и директорий Создадим внутри пакета необходимые папки и файл запуска:

```
mkdir launch scripts
```

Также скопируйте в новый пакет файлы модели робота с дифференциальным приводом. Обратите внимание, что внутри URDF файлов может использоваться название пакета, их необходимо исправить после копирования.

Установка необходимых пакетов

```
sudo apt-get install ros-noetic-ros-control
 ros-noetic-ros-controllers
```

Настройка зависимостей В файлах `CMakeLists.txt` и `package.xml` необходимо указать зависимости:

- `controller_manager`
- `joint_state_controller`

6.1 Добавление необходимых компонентов

Добавление плагина gazebo_ros_control **[10]** Добавьте следующий программный код в файл с расширением .gazebo:

```
<gazebo>
    <plugin name="gazebo_ros_control" filename
   ="libgazebo_ros_control.so">
        <robotNamespace>/diffrobot</
   robotNamespace>
    </plugin>
</gazebo>
```

Добавьте Transmission **в** diff_drive_robot.xacro

```
<transmission name="tran1">
    <type>transmission_interface/
   SimpleTransmission</type>
    <joint name="joint1">
        <hardwareInterface>
   EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
        <hardwareInterface>
   EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</
   mechanicalReduction>
    </actuator>
</transmission>

<transmission name="tran2">
    <type>transmission_interface/
   SimpleTransmission</type>
        <joint name="joint2">
            <hardwareInterface>
   EffortJointInterface</hardwareInterface>
        </joint>
    <actuator name="motor2">
        <hardwareInterface>
   EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</
   mechanicalReduction>
    </actuator>
```

```
</transmission>
```

Создание конфигурационного файла `config/diff_control.yaml`

```
diffrobot:
    # Publish all joint states
   ----------------------------------
    joint_state_controller:
    type: joint_state_controller/
   JointStateController
    publish_rate: 50

    # Position Controllers
   ---------------------------------------
    joint1_position_controller:
    type: effort_controllers/
   JointPositionController
    joint: joint1
    pid: {p: 100.0, i: 0.01, d: 10.0}

    joint2_position_controller:
    type: effort_controllers/
   JointPositionController
    joint: joint2
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

Важно! Убедитесь, что параметры плагина были настроены корректно.

Настройка файла `launch` Создайте конфигурационный файл `.yaml` со следующим содержанием.

```
    <!-- Load joint controller configurations
   from YAML file to parameter server -->
    <rosparam file="$(find
   my_robot_simulation_control)/config/
   diff_control.yaml" command="load"/>

    <!-- Load the controllers -->
    <node name="controller_spawner" pkg="
   controller_manager" type="spawner" respawn=
   "false"
```

```
    output="screen" ns="/diffrobot" args="
  left_wheel_controller
  right_wheel_controller
  joint_state_controller --timeout 60"/>
```

6.2 Запуск ros_control

```
roslaunch my_robot_simulation_control
 robot_sim_control.launch
```

6.3 Отправка управляющих команд

Через консольное окно (в терминале)

```
rostopic pub -1 /rrbot/joint1_controller/
 command std_msgs/Float64 "data: 1.5"
```

Через `rqt`

```
rosrun rqt_gui rqt_gui
```

Затем добавьте `'Topics->Message Publisher'` из раздела меню `Plugins`.

6.3.1 Задание для самостоятельного выполнения

1. Настроить управление колесами не по положению (как в примере), а по скорости.
2. Настроить в `rqt` вывод графика скорости каждого из колес робота.
3. Настроить коэффициенты ПИД-регулятора.

6.3.2 Вопросы для самопроверки

- Каким образом можно управлять роботом с дифференциальным приводом?
- Что такое ros_control?

- Что такое Controller manager?
- Что такое gazebo_ros_control ?
- Как подавать управляющий сигнал в ros_control?
- Какие дополнительные парметры необходимо добавить в URDF для управления через ros_control?

7 Практическая работа №6. Настройка и получение данных с датчиков

Цель работы

- Изучение особенностей настройки и работы в Gazebo ROS с датчиком типа Camera
- Изучение особенностей настройки и работы в Gazebo ROS с датчиком типа LIDAR

7.1 Настройка датчика типа Camera

Создайте новый пакет и назовите его **my_robot_simulation**, для этого необходимо выполнить команду:

```
catkin_create_pkg
 my_robot_simulation_sensors rospy std_msgs
```

Далее необходимо создать файл запуска в папке launch по аналогии с предыдущими занятиями.

При копировании файлов или фрагментов исходного кода (или разметки) проверяйте корректность указанных путей (включая название файлов).

1. добавление физической модели датчика

   Чтобы работать с датчиками, необходимо добавить физическую модель датчика и связать ее с системой координат в модели робота. Добавьте физический элемент в модель робота diff_drive_robot.urdf.xacro, как это показано ниже.

```
<link name='camera'>
    <visual name='camera_visual'>
        <origin xyz="0 0 0" rpy=" 0 0 0" /
   >
        <geometry>
            <box size=".05 .05 .05" />
        </geometry>
        <material name="gray" />
    </visual>
```

```
</link>
<joint type="fixed" name="camera_joint">
    <origin xyz="0.3 0 0" rpy="0 0 0" />
    <child link="camera" />
    <parent link="base" />
</joint>
```

2. настройка плагина, обеспечивающего работу датчика

   Теперь необходимо подключить и настроить плагин, который будет обеспечивать связь ROS и Gazebo для вашего сенсора. Для этого добавьте в diff_drive.gazebo следующую разметку:

```
<gazebo reference="camera">
    <material>Gazebo/Black</material>
    <sensor type="camera" name="camera1">
        <update_rate>30.0</update_rate>
        <camera name="head">
            <horizontal_fov>1.3962634</
   horizontal_fov>
            <image>
                <width>600</width>
                <height>400</height>
                <format>R8G8B8</format>
            </image>
            <clip>
                <near>0.02</near>
                <far>300</far>
            </clip>
        </camera>
        <plugin name="camera_controller"
   filename="libgazebo_ros_camera.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>0.0</updateRate>
            <cameraName>diff_drive_robot/
   camera1</cameraName>
            <imageTopicName>image_raw</
   imageTopicName>
            <cameraInfoTopicName>
   camera_info</cameraInfoTopicName>
            <frameName>camera</frameName>
```

```
            <hackBaseline>0.07</
    hackBaseline>
            <distortionK1>0.0</
    distortionK1>
            <distortionK2>0.0</
    distortionK2>
            <distortionK3>0.0</
    distortionK3>
            <distortionT1>0.0</
    distortionT1>
            <distortionT2>0.0</
    distortionT2>
         </plugin>
     </sensor>
</gazebo>
```

3. проверка работоспособности камеры после настройки

   Чтобы проверить корректность работы, необходимо запустить Gazebo и указать необходимые параметры launch файла.

```
<launch>

<arg name="rviz_conf_file" default="$(find
     my_robot_simulation)/urdf/rviz_conf.
    rviz" />

  <node name="joint_state_publisher" pkg="
    joint_state_publisher" type="
    joint_state_publisher" ></node>

<param name="robot_description" command="
    xacro '$(find my_robot_simulation)/
    diff_drive_robot/urdf/diff_drive_robot.
    xacro'" />

  <node pkg="robot_state_publisher" type="
    robot_state_publisher" name="
    robot_state_publisher" output="screen"
    >
     <param name="publish_frequency" type="
    double" value="40.0" />
```

```
  </node>

  <include file="$(find gazebo_ros)/launch
   /empty_world.launch">
    <arg name="world_name" value="$(find
   my_robot_simulation)/worlds/empty.world
   "/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/
   >
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <node name="my_robot_spawn" pkg="
   gazebo_ros" type="spawn_model" output="
   screen"
   args="-urdf -param robot_description -
   model diff_drive_robot1" />

<node name="rviz" pkg="rviz" type="rviz"
   args="-d $(arg rviz_conf_file)" />

</launch>
```

Запустите launch файл.

```
1    roslaunch my_robot_simulation_sensors
   robot_sim_camera.launch
2
```

Откройте Rviz и выполните последовательность команд: Add $--$ $>$ Image, затем выберите топик, куда публикуется изображение (image_raw).

4. обработка изображения с камеры в программном коде

   Чтобы получить данные с камеры в управляющей программе, вам понадобится создать файл исходного кода и прописать в нем импорт библиотеки cv_bridge, который позволит

вам преобразовать формат данных в более удобный для дальнейшей обработки при помощи библиотеки opencv **??**.

Далее приведен пример, где изображение из камеры преобразовывается в другой формат и обрабатывается в sensor_data_proc.py :

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import Image

import cv2
from cv_bridge import CvBridge

class sensor_datahandler:

    def __init__(self):
        sub_topic_name ="/diff_drive_robot
    /camera1/image_raw"
        self.camera_subscriber = rospy.
    Subscriber(sub_topic_name, Image, self.
    camera_cb)
        # self.out = cv2.VideoWriter('/
    home/superuser/output.avi',cv2.
    VideoWriter_fourcc('M','J','P','G'),
    10, (640,480))
        self.bridge = CvBridge()
        self.curent_image = None

    def camera_cb(self, data):
        frame = self.bridge.imgmsg_to_cv2(
    data)
        print(frame.shape)
        # edge_frame = cv2.Canny(frame,
    100, 200)
        # self.out.write(frame)
        cv2.imshow("output", frame)
        cv2.waitKey(1)


if __name__ == '__main__':
```

```
    rospy.init_node("
    camera_data_processing")
    sensor_datahandler()
    rospy.spin()
```

Напишите скрипт обработки и добавьте запуск программного кода обработки изображения в файле запуска.

7.2 Добавление и настройка датчика лидара

Процесс добавления датчика лидара в имитационную модель состоит из аналогичных шагов. Необходимо добавить физическую модель датчика в модель робота и подключить плагин лидара с указанием необходимых параметров датчика.

1. Добавление физической модели датчика лидара.

   Добавьте физический элемент в модель робота diff_drive_robot.urdf.xacro, как это показано:

```
<link name="lidar">
    <visual name="lidar_visual">
        <geometry>
            <box size="0.01 0.01 0.01" />
        </geometry>
        <origin xyz="0 0 0" rpy="${pi/2} 0
    0" />
        <material name="yellow" />
    </visual>
</link>

<joint name="lidar_joint" type="fixed">
    <origin xyz="0.1 0.02 0.08" rpy="0 0 0
   " />
    <child link="lidar" />
    <parent link="base_link" />
</joint>
```

2. Настройка плагина, обеспечивающего работу датчика лидара.

Подключите плагин, обеспечивающий связь ROS и Gazebo для вашего лидара, путем добавления в diff_drive.gazebo следующей разметки:

```
<gazebo reference="lidar">
    <material>Gazebo/Black</material>
    <sensor type="ray" name="lidar-right-
   eye">
      <pose>0 0 0 0 0 0</pose>
      <visualize>true</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
            <resolution>1</resolution>
            <min_angle>-0.785</min_angle>
            <max_angle>0.785</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.03</min>
          <max>3.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>

          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="
   gazebo_ros_head_lidar_controller"
   filename="libgazebo_ros_laser.so">
        <topicName>/diff_drive_robot/laser
   </topicName>
        <frameName>lidar</frameName>
      </plugin>
    </sensor>
</gazebo>
```

3. Проверка работоспособности лидара.

   Чтобы проверить корректность работы, необходимо запустить Gazebo и указать необходимые параметры launch файла.

```
<launch>

<arg name="rviz_conf_file" default="$(find
    my_robot_simulation)/urdf/rviz_conf.
   rviz" />

  <node name="joint_state_publisher" pkg="
   joint_state_publisher" type="
   joint_state_publisher" ></node>

<param name="robot_description" command="
   xacro '$(find my_robot_simulation)/
   diff_drive_robot/urdf/diff_drive_robot.
   xacro'" />

  <node pkg="robot_state_publisher" type="
   robot_state_publisher" name="
   robot_state_publisher" output="screen"
   >
    <param name="publish_frequency" type="
   double" value="40.0" />
  </node>

  <include file="$(find gazebo_ros)/launch
   /empty_world.launch">
    <arg name="world_name" value="$(find
   my_robot_simulation)/worlds/empty.world
   "/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/
   >
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>
```

```
  <node name="my_robot_spawn" pkg="
   gazebo_ros" type="spawn_model" output="
   screen"
   args="-urdf -param robot_description -
   model diff_drive_robot1" />

<node name="rviz" pkg="rviz" type="rviz"
   args="-d $(arg rviz_conf_file)" />

</launch>
```

Запустите launch файл.

```
1    roslaunch my_robot_simulation_sensors
     robot_sim_camera.launch
2
```

Откройте Rviz и выполните последовательность команд: "Add", "Image", затем выберите топик, куда публикуется изображение (image_raw).

4. Обработка данных лидара в программном коде.

   Чтобы получить данные с лидара в управляющей программе, вам понадобится создать файл исходного кода и подписаться на соответсвующий топик.

   Далее приведен пример, где данные из лидара обрабатываются в sensor_data_proc.py :

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import Image

import cv2
from cv_bridge import CvBridge

class sensor_datahandler:

    def __init__(self):
        sub_topic_name ="/diff_drive_robot
   /camera1/image_raw"
```

```
        self.camera_subscriber = rospy.
    Subscriber(sub_topic_name, Image, self.
    camera_cb)
        # self.out = cv2.VideoWriter('/
    home/superuser/output.avi',cv2.
    VideoWriter_fourcc('M','J','P','G'),
    10, (640,480))
        self.bridge = CvBridge()
        self.curent_image = None

    def camera_cb(self, data):
        frame = self.bridge.imgmsg_to_cv2(
    data)
        print(frame.shape)
        # edge_frame = cv2.Canny(frame,
    100, 200)
        # self.out.write(frame)
        cv2.imshow("output", frame)
        cv2.waitKey(1)


if __name__ == '__main__':
    rospy.init_node("
    camera_data_processing")
    sensor_datahandler()
    rospy.spin()
```

Напишите скрипт обработки и добавьте запуск программного кода в файле запуска.

7.2.1 Задание для самостоятельного выполнения

1. Создать сцену с объектами окружения (стандартные из Gzebo), сохранить в файл с новым названием и настроить его подгрузку в файле запуска;

2. Написать скрипт, который будет отправлять сообщения в /cmd_vel (любые сообщения);

3. При выполении "spawn" (добавление робота в симуляцию) необходимо задать координаты x y z как аргументы;

4. Добавить в симуляцию несколько одинаковых роботов одновременно в разных координатах сцены/

7.2.2 Вопросы для самопроверки

- Какие действия необходимо выполнить для добавления датчика в имитационную модель робота?
- Что такое плагин датчика и зачем он нужен?
- Какие типы датчиков доступны в Gazebo?
- Какую информацию возвращает лидар?

8 Практическая работа №7. Работа с физическими роботами на ROS (ROS Serial)

Цель работы

- Подключение физически воплощенного робота вместо виртуального
- Обмен данными между высокоуровным ПО на ROS и низкоуровневым ПО на микроконтроллере

8.1 Установка Arduino на Ubuntu

1. Перейдите на страницу скачивания Arduino IDE и скачайте `AppImage`, сделайте его исполняемым (в свойствах установить флаг `executable`).

2. Создайте новую папку `Arduino` в корневой директории пользователя на Linux и скопируйте туда `AppImage`.

3. Добавьте вашего пользователя в группу `dialout` путем выполнения следующей команды:

```
sudo usermod -a -G dialout superuser

```

(где `superuser` – это имя вашего пользователя)

4. Предоставьте права на запись и чтение с порта:

```
sudo chmod a+rw /dev/ttyUSB0

```

(Путь к устройству "/dev/ttyUSB0" может отличаться, укажите актуальный)

5. Можно создать ярлык Arduino для удобного запуска программы (опционально).

8.2 Настройка Arduino IDE

1. В папке `Arduino` создайте подпапку `ArduinoSketches`.
2. Запускаем и в `Arduino IDE` заходите в **File** → **Preferences** и в поле **Sketchbook location** укажите путь до папки `ArduinoSketches`.

8.3 Установка ROSSerial

ROSSerial является библиотекой, которая будет обеспечивать связь между ROS и микроконтроллером через последовательный порт компьютера.

1. Устанавливаем необходимые пакеты:

```
sudo apt-get install ros-noetic-
 rosserial-arduino
sudo apt-get install ros-noetic-
 rosserial-python
sudo apt-get install ros-noetic-
 rosserial
```

2. В `Arduino IDE` выберите **Добавить библиотеку**, найдите `rosserial` и нажмите **установить**.
3. После установки исправьте ошибку в библиотеке:
   - Найти файл в папке установки `/ArduinoSketches/libraries /Rosserial_Arduino_Library/src/ros/msg.h`.
   - Замените строку `#include <cstring>` на `#include <string.h>`.
   - Замените `std::memcpy` на `memcpy` в двух местах файла.

8.3.1 Подключение Arduino и запуск ROSSerial

1. Подключите устройство Arduino к компьютеру и проверьте название порта (проще всего это сделать в `Arduino IDE`).
2. Установите права на чтение и запись для порта:

```
sudo chmod 666 /dev/ttyUSB0
```

(Название порта может отличаться, проверьте его перед выполнением команды)

3. Нажмите кноку начала загрузки прошивки на устройство `Arduino` с необходимым программным кодом (например, стандартный пример `HelloWorld`).

4. На компьютере запустите `roscore`:

```
roscore
```

5. Запустите коммуникационный узел:

```
rosrun rosserial_python serial_node.py
/dev/ttyUSB0
```

6. Проверьте обмен данными через соответствующие топики:

```
rostopic echo /chatter
```

8.3.2 Задание для самостоятельного выполнения

1. Открыть пример **Blink** и загрузить его на устройство Arduino.

2. Подключить сонар к Arduino и открыть пример **UltraSound**, настроить считывание ультразвукового дальномера в ROS.

8.3.3 Вопросы для самопроверки

- Можно ли использовать библиотеку ROSSerial с микроконтроллерами других производителей, кроме Arduino?

- Через какой аппаратный интерфейс работает взаимодействие микроконтроллера и комьютера?
- Имеются ли ограничения на количество ROS узлов (publisher/subscriber), которые можно создавать на микроконтроллере Arduino при использовании библиотеки ROSSerial?

9 Лабораторная работа №1. Исследование ROS пакета

Цель работы

Целью лабораторной работы является:

- Изучение основ работы с Linux
- Установка и базовая настройка ROS
- Знакомство с базовыми возможностями обмена данными в ROS

В данной работе необходимо пользоваться лекционными материалами и инструкциями по установке необходимого программного обеспечения, которые приведены в приложении А на странице 14.

Перед выполнением лабораторной работы убедитесь, что вы успешно освоили материалы, приведенные в практической работе №1, так как вышеуказанная практическая работа является подготовительной для выполнения лабораторной работы.

9.1 Задание к лабораторной работе № 1

1. Найдите папку lab1 в ранее скаченной папке [8] и положите в папку рабочего окружения workspace в подпапку src.

2. Вам необходимо исправить содержимое ROS пакета [11] таким образом, чтобы там оказалось только 3 файла/3 ноды (их названия указаны в таблице), два узла должны отправлять по одному числу в третий узел (в Таблице 9.1 последний столбик), который в свою очередь должен публиковать результат в топик, название которого состоит из слова “result_” + ваш идентификатор в ISU (например result_123456).

9.2 Как отправить свое решение на проверку?

Результат вашей работы должен находиться в пакете (подпапке) с названием my_robot_controller. Перед отправкой работы

Таблица 9.1. Номера вариантов для студентов соответствуют последней цифре ISU ID

ID	node1.py	node2.py	node3.py	операция	Результат вычисляет
0	+	+		+	node3.py
1		+	+	+	node1.py
2	+		+	+	node2.py
3	+	+		*	node3.py
4		+	+	*	node1.py
5	+		+	+	node2.py
6	+	+		−	node3.py
7		+	+	−	node1.py
8	+		+	−	node2.py
9		+	+	/	node1.py

воспользуйтесь инструкций по настройке и процедуре отправки работ на проверку (Приложение B).

10 Лабораторная работа №2. Создание пакета и сообщений, первый регулятор в ROS

Цель работы

- Создание ROS-пакета
- Создание пользовательского типа сообщений в ROS
- Разработка простого регулятора в ROS

10.1 Задание к лабораторной работе № 2

В рамках лабораторной работы необходимо написать регулятор, реализующий разные функции в зависимости от того, из какого пространства имен его запустили. А именно, регулятор должен управлять первой черепахой таким образом, чтобы она проехала через заданные точки (их координаты указаны в таблице), при этом необходимо зафиксировать время, за которое черепашка проезжает из начала до последней точки и отправить затраченное время в топик, название которого состоит из слова "result_" + ваш ID в ISU. Вторая черепаха должна подписываться на топик с положением первой черепаки и повторять движение за первой черепахой (не имея явного доступа к координатам целевых точек первой черепахи).

1. Скопируйте содержимое папки `lab2` из локальной копии в рабочее пространство catkin, то есть папку `my_best_controller` в папку `workspace/src/`.

2. Создайте launch-файл, внутри которого будут запускаться два экземпляра одного и того же регулятора в разных пространствах имен:

   (a) Откройте терминал и запустите ROS Master:

```
1       roscore
2
```

(b) Создайте новый файл запуска с названием `lab2_setup.launch` и определите два пространства имен с названиями `ns1_ISUID` и `ns2_ISUID`, где `ISUID` — это ваш номер в ИСУ.

(c) Внутри каждого пространства имен необходимо запустить `turtlesim`.

(d) В папке `scripts` создайте файл `lab2_controller.py` и напишите в нем программу движения для первой черепахи согласно разделу «Программа движения» (см. пункт 4).

Важно: обратите внимание на названия ресурсов, запускаемых внутри пространства имен.

(e) Вторая черепаха должна подписываться на топик `.../pose` первой черепашки и повторять его действия (запрещено использовать `remap`). Управляющая программа для второй черепашки также должна быть прописана в файле `lab2_controller.py`.

(f) Обратите внимание, что внутри контроллера необходимо использовать имена ресурсов либо относительные, либо локальные (`private`), иначе возникнет конфликт имен при запуске двух или более экземпляров одного и того же контроллера.

3. **Программа движения:** черепашка №1 должна пройти последовательность точек согласно варианту, указанному в Таблице 10.1. В Таблице 10.1 указаны 5 разных точек, но программа должна проходить только по тем точкам, у которых указаны координаты напротив вашего номера варианта.

   Формат координат: `(x, y)`. Порядок прохождения точек не имеет значения,

4. Готовый пакет должен запускаться командой:

```
1    roslaunch lab2\_setup.launch
2
```

5. **Ограничение по времени:** выполнение поставленной задачи для черепахи не должно превышать 5 минут. Хаотичное движение черепашки не допускается.

Таблица 10.1. Номера вариантов для студентов соответствуют последней цифре ISU ID

ID	Точка 1	Точка 2	Точка 3	Точка 4	Точка 5
1	(2,3)	(4,2)	(7,1)	(8,4)	
2	(2,3)	(4,2)		(5,4)	
3	(2,3)		(7,1)	(5,4)	(8,4)
4		(4,2)		(5,4)	(8,4)
5	(2,3)		(7,1)		(8,4)
6		(4,2)	(7,1)	(5,4)	
7	(2,3)		(5,4)	(4,2)	(5,4)
8	(7,1)	(5,4)		(8,4)	(5,4)
9	(2,3)	(4,2)	(7,1)	(5,4)	(4,2)
0	(2,3)	(4,2)	(7,1)	(8,4)	(4,2)

10.2 Отправка решения

Решение отправляется согласно инструкции курса по ROS через систему git.

11 Лабораторная работа №3. Создание моделей роботов и объектов окружения

Цель работы

- Создание модели робота с использованием универсального языка описания динамики и кинематики робота URDF.
- Оптимизация модели робота с использованием XACRO.
- Изучение инструментов визуализации модели робота в формате URDF.

11.1 Описание лабораторной работы

1. Скопируйте содержимое папки `lab3` из локальной копии в рабочее пространство catkin, т.е. папку `my_firts_robot_model` в папку `catkin_ws/src/`.
2. **Создание модели робота:**
   (a) Создать папку `launch` и `urdf` в пакете `my_firts_robot_model`.
   (b) В папке `urdf` создать файл `my_robots_super_model.xacro` и внутри составить XML-описание мобильного робота с закрепленным на нем манипулятором. Параметры модели (габариты, количество и типы звеньев) соответствуют варианту в таблице **??**.
   (c) Создать файл запуска `lab3_setup.launch` и прописать загрузку модели робота в сервер параметров.
   (d) В файле `lab3_setup.launch` также указать запуск необходимых пакетов для корректной визуалиазции модели робота: `robot_state_publisher` и `joint_state_publisher`.
   (e) Добавить запуск `rviz` с подгруженной конфигурацией модели и TF.
3. Готовый пакет должен запускаться с помощью команды:

```
roslaunch lab3_setup.launch
```

Варианты заданий. Номера вариантов для студентов соответствуют последней цифре ISU ID, значения парметров указаны в Таблице 11.1. R - вращательный шарнир, P

Таблица 11.1. Параметры системы

Номер варианта	Кол-во звеньев	Кол-во и тип шарниров	Габариты (мм) Д×Ш×В
1	5	RRRR CC	200x140x90
2	6	PRR CC	200x140x120
3	7	PRP CCCC	260x140x100
4	8	PRP CCS	260x160x100
5	5	RRP CC	190x160x130
6	6	RRRR CC	200x160x90
7	7	PRR CC	200x160x90
8	8	PRP CCCC	230x140x90
9	9	RPRC CC	220x160x90
0	10	RRPR CCCC	200x180x120

- призматический шарнир, C - вращательный шарнир без ограничений на угол поворота (постоянного вращения)

12 Лабораторная работа №4. Создание имитационной модели с роботом произвольной кинематики

Цель работы

- Создание имитационной модели робота в Gazebo.
- Реализация управления через ros_control и настройка регуляторов.
- Подключение и настройка дитчиков (лидар, камера).
- Разработка и тестирование автономной управляющей программы с использованием данных датчиков.

Для выполнения лабораторной работы №4 необходимо освоить материалы, представленные в практических работах №4, №5 и №6.

12.1 Создание нового пакета в рабочем пространстве catkin

Необходимо перейти в рабочую папку окружения catkin и создать пакет с названием **my_robot_simulation_control**, для этого выполним команды из листинга

```
cd catkin_ws/src
catkin_create_pkg
 my_robot_simulation_control roscpp rospy
 std_msgs
```

Затем необходимо выполнить сборку созданного ROS-пакета.

Внутри ROS пакета создайте подпапки scripts, launch и urdf. Затем, необходимо создать файл запуска с названием lab4.launch.

12.2 Подготовка сцены для симуляции

Сцена для имитационного моделирования создается в соответствии с назначением робота и должна содержать модели объектов, с которыми роботу предстоит взаимодействовать после физического воплощения.

В рамках данной лабораторной работы необходимо:

1. Создать файл сцены в формате sdf, содержащий минимальный набор элементов: источник освещения, плоскую поверхность и гравитацию, как показано в следующем примере.

```
<sdf version="1.4">
  <world name="default">

    <scene>
      <ambient>0.4 0.4 0.4 1</ambient>
      <background>0.7 0.7 0.7 1</
    background>
      <shadows>true</shadows>
    </scene>

    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <gravity>0 0 -9.8</gravity>

    <physics type="ode">
      <real_time_update_rate>1000.0</
    real_time_update_rate>
      <max_step_size>0.001</max_step_size>
      <real_time_factor>1</
    real_time_factor>
      <ode>
        <solver>
          <type>quick</type>
          <iters>150</iters>
          <precon_iters>0</precon_iters>
          <sor>1.400000</sor>
```

```
          <use_dynamic_moi_rescaling>1</
    use_dynamic_moi_rescaling>
        </solver>
        <constraints>
          <cfm>0.00001</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>
    2000.000000</contact_max_correcting_vel
    >
          <contact_surface_layer>0.01000</
    contact_surface_layer>
        </constraints>
      </ode>
    </physics>
  </world>

</sdf>
```

2. Также необходимо добавить минимум 3 разных по форме объекта из реального мира в сцену (это не могут быть примитивы: кубойд, сфера или цилиндр).

12.3 Подготовка модели робота

1. Скопируйте в новый пакет файлы модели робота с дифференциальным приводом, которая была создана на практических задания по управлению роботом. При необходимости внесите изменения в модель (согласно таблице с вариантами задания).

2. В рамках лабораторной работы необходимо создать симуляцию робота с дифференциальным приводом, который не использует плагин differential_drive, а использует ros_control, при этом упраление колесами должно выполняться по скорости, а не по положению. Следовательно, необходимо удалить differential_drive из файла .gazebo и вместо него подключить плагин ros_control.

3. Использование ros_control требует создания конфигурационных файлов регуляторов, а также их загрузку в сервер парметров.

4. Далее необходимо добавить в модель робота сенсоры тапа: лидар и камера.

5. После добавления датчиков проверьте их работоспособность при помощи RViz.

12.4 Настройка регуляторов

На практических заданиях был рассмотрен регулятор, позволяющий управлять по положению. На практике довольно часто возникает необходимость управления по скорости. Таким примером является колесо мобильного робота, управление которым необходимо выполнять именно по скорости.

При создании любых регуляторов стоит помнить, что фактически управление в большинстве приводов осуществляется по моменту, однако мы можем использовать такой регулятор, который на вход принимает заданную скорость, и имея обратную связь по положению от привода, может задавать необходимый момент для обеспечения заданной скорости вращения (имеет внутренний контур управления). Исходя из того, что фактическое управление выполняется моментом, мы можем догадаться, что для этого подходит пакет effort_controllers, из которого мы можем импортировать регулятор.

Процесс добавления и настройки регуляторов подробно описан в практической работе №5.

12.5 Написание управляющей программы

В данной работе также необходимо написать управляющую программу, которая будет считывать данные с сенсоров и формировать управляющий сигнал для регуляторов. Цель алгоритма управления может быть любой, как и поведение робота, при этом задача робота должна быть понятной внешнему наблюдателю. Например (даннный пример нельзя использовать в

лабораторной работе), алгоритм управления может по данным с датчиков найти дверь и проехать через дверной проем.

Варианты заданий.

Номера вариантов для студентов соответствуют последней цифре ISU ID:

Пакет с лабораторной работой должен быть загружен в папку с названием **lab4** в приватном каталоге на сайте gitlab. Имитационная модель должна запускаться из пакета **my_robot_simulation_control** при выполнении файла запуска с названием **lab4.launch**.

Таблица 12.1. Параметры системы

Последняя цифра ису ID	Диапазон видимости лидара(мин-макс)	Разрешение камеры	Габаритные размеры в мм, Д Ш В
1	30 мм 3 м	1024×960	$200 \times 140 \times 90$
2	40 мм 4 м	960×840	$200 \times 140 \times 120$
3	30 мм 3 м	960×960	$260 \times 140 \times 100$
4	30 мм 3 м	1024×960	$260 \times 160 \times 100$
5	30 мм 4 м	1024×1024	$190 \times 160 \times 130$
6	30 мм 5 м	800×600	$200 \times 160 \times 90$
7	50 мм 5 м	1024×960	$200 \times 160 \times 90$
8	30 мм 3 м	1024×600	$230 \times 140 \times 90$
9	100 мм 3 м	960×960	$220 \times 160 \times 90$
0	300 мм 3 м	960×840	$200 \times 180 \times 120$

Все параметры, значения которых явно не указаны в лабораторной работе, выбираются произвольно.

СПИСОК ЛИТЕРАТУРЫ

[1] Ubuntu Сообщество. Операционная система Ubuntu. — https://ubuntu.com/. — [Дата обращения: 13.02.2025].

[2] Документация по Catkin tools. — https://catkin-tools.readthedocs.io/en/latest/. — [Дата обращения: 13.02.2025].

[3] Официальная документация ROS Noetic. — https://wiki.ros.org/noetic. — [Дата обращения: 13.02.2025].

[4] Официальная документация Git. — https://git-scm.com/. — [Дата обращения: 04.03.2025].

[5] Gazebo Classic Plugins. — https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#Tutorial:UsingGazebopluginswithROS. — [Дата обращения: 28.02.2025].

[6] West Kenya. Документация по подсистеме Windows для Linux. — https://ru.wikipedia.org/wiki/Windows_Subsystem_for_Linux. — [Дата обращения: 13.02.2025].

[7] Официальный сайт Visual Studio Code. — https://code.visualstudio.com/. — [Дата обращения: 05.03.2025].

[8] Асланович Бжихатлов Ислам. Материалы курса "Операционная системв ROS". — https://gitlab.com/likerobotics/ros-course-itmo. — [Дата обращения: 13.02.2025].

[9] Building a Visual Robot Model with URDF from Scratch. — https://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch. — [Дата обращения: 22.02.2025].

[10] Gazebo Classic ROS Control. — https://classic.gazebosim.org/tutorials?tut=ros_control. — [Дата обращения: 28.02.2025].

[11] Бжихатлов Ислам Асланович. Исходный код ROS пакета для Лабораторной работы №1. — https://gitlab.com/likerobotics/ros-course-itmo/lab1/. — [Дата обращения: 28.02.2025].

[12] Официальный сайт инструмента для контейнеризации Docker. — https://www.docker.com/. — [Дата обращения: 05.03.2025].

14 Приложение А. Подготовка операционной системы

В общем случае можно выделить 3 основных варианта подготовки основной операционной системы для использования ROS, каждый из которых имеет свои преимущества и недостатки.

- Установка системы, подходящей под версию дистрибутива ROS (Для ROS noetic это Ubuntu 20.04).
- Установка WSL на операционную систему Windows
- Установка Docker и использование готовых инструкций для Docker

Далее авторы приводят практическое руководство, которое позволит выбрать вам наиболее подходящий вариант.

Если вы хотите работать на операционной системе Ubuntu 22, 24 или WSL, вам потребуется Docker [12] . Пользователям Ubuntu 20.04 не нужен Docker, достаточно сразу перейти к установке ROS.

14.1 Для пользователей Windows – настройка WSL

Для данного курса можно использовать WSL внутри ОС Windows 10 или 11. Для этого выполните следующие шаги:

1. Откройте строку поиска, введите `cmd` и нажмите **Enter**. Должно открыться командное окно Windows.
2. В командной строке введите:

   ```
   wsl --install
   ```

   и нажмите **Enter**. Во время установки или позже система может запросить ввод имени пользователя и/или пароля. Обязательно сохраните пароль, он понадобится позже.
3. После завершения установки вы окажетесь внутри системы Linux.

4. В любое время можно открыть `cmd` и использовать команду:

   ```
   wsl
   ```

   чтобы войти в Linux.

5. Перейдите к разделу установки Docker.

14.2 Для пользователей Ubuntu 22 и 24

Следуйте инструкции по установке Docker.

14.3 Установка Docker

Docker — это полезный инструмент, позволяющий запускать любую операционную систему в изолированной среде.

Инструкции по установке Docker доступны по ссылке: `https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository`

Не забудьте добавить пользователя в группу Docker: `https://docs.docker.com/engine/install/linux-postinstall/`

После установки переходите к разделу использования Docker.

14.4 Настройка работы в Docker

Разработчик курса подготовил bash-скрипты для установки Docker. Чтобы получить их на свой компьютер, выполните следующие шаги:

1. Склонируйте репозиторий:

   ```
   git clone https://gitlab.com/likerobotics/ros-course-itmo.git
   ```

2. Перейдите в каталог Docker:

   ```
   cd ros-course-itmo/docker
   ```

3. Проверьте, являются ли файлы исполняемыми:

   ```
   ls
   ```

   Если файлы не выделены зеленым цветом, сделайте их исполняемыми:

   ```
   chmod +x *
   ```

4. Выполните следующие команды:

   ```
   xhost +local:root
   ./docker_install.bash
   ./docker_build.bash
   ./docker_run.bash
   ```

   Первая команда позволяет GUI-интерфейсу работать внутри Docker. Остальные команды устанавливают и запускают контейнер.

Замечание.
При закрытии терминала или нажатия ctrl+C сессия Docker завершится, а все данные внутри контейнера будут удалены, за исключением данных в папке `workspace`. Создавайте рабочие окружения catkin только в этой папке, чтобы не потерять данные.

14.5 Запуск терминала внутри Docker

Если вам нужен дополнительный терминал внутри Docker, нельзя использовать предыдущий скрипт, так как контейнер уже запущен. Вместо этого:

1. Откройте новый терминал.
2. Перейдите в каталог Docker:

```
cd docker
```

3. Выполните команду:

```
./docker_new.bash
```

При установке Docker с использованием этих bash-скриптов, ROS также будет установлен внутри Docker.

Бжихатлов Ислам Асланович

Операционная система для роботов ROS. Часть 1 (базовый уровень)

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А