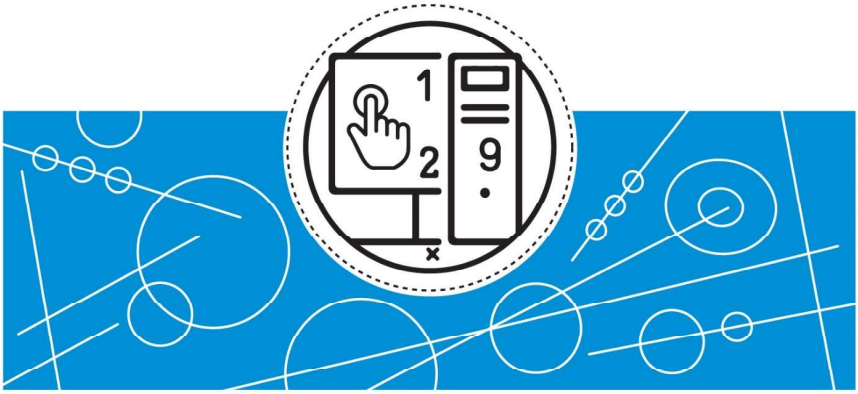
ИТМО

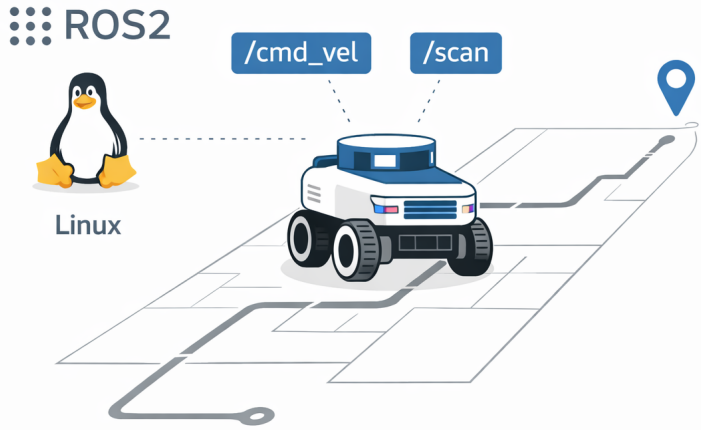
ОПЕРАЦИОННАЯ СИСТЕМА ДЛЯ РОБОТОВ ROS 2



Санкт-Петербург
2026

ИТМО

ОПЕРАЦИОННАЯ СИСТЕМА ДЛЯ РОБОТОВ ROS 2



Санкт-Петербург

2026

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

ОПЕРАЦИОННАЯ СИСТЕМА ДЛЯ РОБОТОВ ROS 2

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

Санкт-Петербург
2026

Операционная система для роботов ROS 2 / Бжихатлов И.А., Ткачев И.Ю., К.А. Зименко, [и др.] — Санкт-Петербург: Университет ИТМО, 2026. — 138 с.
Рецензент: Артемов Кирилл, к.т.н., ведущий инженер факультета систем управления и робототехники, Университета ИТМО.

В данном пособии изложены методические рекомендации по изучению современных инструментов для разработки и тестирования программного обеспечения для роботов. Практические занятия и лабораторные работы способствуют освоению технологий моделирования роботов, разработке моделей сенсоров и управляющих алгоритмов. Использование ROS 2 позволяет существенно увеличить скорость разработки прикладного программного обеспечения роботов.

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 разработки, сопровождения и масштабирования программных решений в робототехнике на базе ROS 2 (Robot Operating System 2). Пособие предназначено для использования в курсе, посвящённом робототехническим программным платформам.

В результате освоения материала студенты получат практические навыки разработки прикладного программного обеспечения с использованием ROS 2, научатся интегрировать робототехническое оборудование в реальные и симуляционные среды, а также приобретут компетенции в области отладки, тестирования и визуализации поведения робототехнических систем. Дополнительно будут рассмотрены архитектурные принципы ROS 2 и его инструменты, что позволит существенно повысить эффективность разработки за счёт повторного использования компонентов и применения моделирования перед внедрением в реальные системы.

Терминология, используемая в данном пособии, в значительной степени заимствована из английского языка, что обусловлено международным характером разработки ROS 2. ROS 2 представляет собой распределённую middleware-платформу с открытым исходным кодом, предназначенную для создания масштабируемых, надёжных и отказоустойчивых робототехнических приложений. В отличие от предыдущих версий, ROS 2 изначально проектировался с учётом требований промышленности, включая поддержку систем реального времени, мультиплатформенность и использование стандарта DDS (Data Distribution Service) [1] для обмена данными.

Несмотря на возможность разработки робототехнического программного обеспечения без использования ROS 2, такой подход приводит к значительному увеличению трудозатрат на реализацию базовой инфраструктуры: межпроцессного взаимодействия, управления конфигурацией, логирования и тестирования. ROS 2 предоставляет готовый набор инструментов и библиотек,

существенно сокращающих время разработки и повышающих надёжность решений.

Одним из ключевых аспектов современного подхода к разработке является использование имитационного моделирования. ROS 2 обеспечивает тесную интеграцию с симуляторами, такими как Gazebo, что позволяет тестировать алгоритмы в виртуальной среде до их применения на реальных роботах. Несмотря на дополнительные затраты времени на создание модели, данный подход значительно снижает риски и ускоряет процесс разработки на последующих этапах.

В пособии рассматриваются базовые принципы организации программного обеспечения в ROS 2 как на примере симуляционных моделей, так и при работе с физическими роботами. Особое внимание уделяется вопросам взаимодействия компонентов системы, построению распределённых приложений и особенностям работы в средах моделирования.

Перед началом изучения ROS 2 важно учитывать ряд требований и ограничений, накладываемых на разработчика:
Ограничения:
1. Наличие вычислительной платформы с достаточной производительностью (одноплатный компьютер или ПК), способной выполнять функции высокоуровневого контроллера.
2. Базовые знания операционной системы Linux и командной строки (bash).
3. Владение языками программирования C++ и/или Python (при этом многие низкоуровневые и высокопроизводительные компоненты реализуются на C++).
4. Умение работать с технической документацией (инструкции, руководства).

Преимущества использования ROS 2:
1. Богатая экосистема готовых модулей и библиотек с открытым исходным кодом.
2. Снижение объёма разрабатываемого кода за счёт использования стандартных решений.
3. Поддержка распределённых систем и масштабируемых архитектур.
4. Возможность интеграции с современными средствами моделирования и тестирования.

Учебное пособие предназначено для использования в курсе «Операционная система ROS» программ бакалавриата. В рамках дисциплины студенты получают навыки разработки прикладного программного обеспечения для роботов, знания о современных методах моделирования робототехнических систем, а также умения отладки управляющих программ и переноса их из виртуальной среды разработки на физически реализованные робототехнические платформы.

1 ROS 2 – теоретические сведения

1.1 Назначение ROS 2 и основной дистрибутив

ROS 2 (Robot Operating System 2) представляет собой программный фреймворк, предназначенный для разработки робототехнических систем. Несмотря на название, ROS 2 не является операционной системой в классическом смысле. Он выступает как промежуточный программный слой между операционной системой и прикладными программами робота, обеспечивая единые механизмы обмена данными, запуска компонентов, описания робота, визуализации и интеграции с симуляторами и аппаратурой.

Современные робототехнические системы включают в себя большое количество программных модулей: драйверы датчиков, алгоритмы обработки данных, системы локализации, навигации, управления приводами, визуализации и взаимодействия с оператором. Реализовывать такие компоненты независимо друг от друга неудобно. ROS 2 предоставляет единый вычислительный каркас, в рамках которого все эти модули могут взаимодействовать стандартным способом.

В данном учебном пособии рассматривается дистрибутив **ROS 2 Jazzy**, работающий под управлением Ubuntu 24.04 [2]. Выбор именно этого дистрибутива обусловлен его актуальностью, поддержкой современной экосистемы ROS 2 и совместимостью с используемыми в пособии пакетами. ROS 2 непрерывно эволюционирует, поэтому авторы создали интернет-ресурс [3] для размещения дополнительных материалов, которые не вошли в материалы данного учебного пособия, где вы найдете как базовый, так и более продвинутый материал по ROS 2.

По сравнению с ROS первой версии архитектура ROS 2 была существенно переработана. Ключевым отличием стало использование стандарта DDS (Data Distribution Service) [1] в качестве основы транспортного уровня. Благодаря этому ROS 2 не использует централизованный узел управления, а работает как распределённая система, в которой обнаружение участников сети и доставка сообщений выполняются средствами middleware. Такой

подход улучшает масштабируемость, устойчивость и пригодность системы для распределённых робототехнических приложений.

Стандарт DDS реализуется несколькими поставщиками. В ROS 2 Jazzy реализацией по умолчанию является **eProsima Fast DDS** (`rmw_fastrtps_cpp`) [4, 5]. При необходимости middleware может быть заменён на Eclipse Cyclone DDS или другую реализацию путём задания переменной окружения `RMW_IMPLEMENTATION`. Проверить текущую реализацию можно командой `ros2 doctor -report`. Знание используемого middleware важно при отладке проблем обнаружения узлов в сети.

1.2 Рабочее пространство, пакеты и сборка

Разработка в ROS 2 ведётся в рамках **рабочего пространства** (workspace). Рабочее пространство представляет собой каталог файловой системы, содержащий исходные тексты пакетов и результаты их сборки. В ROS 2 используются три каталога рабочего пространства: `build`, `install` и `src`. Они представляют собой каталоги файловой системы (папки). Каталог `src` необходимо создать самостоятельно; в нём размещаются разрабатываемые ROS-пакеты. Каталог `src` — единственное место, где разрешено редактировать исходный код. Каталоги `build` и `install` являются служебными: их содержимое автоматически обновляется при сборке пакетов.

Пакет (package) является основной единицей организации программного обеспечения в ROS 2. Обычно пакет содержит:

- исходный код программы;
- файл `package.xml` с описанием зависимостей;
- файлы сборки;
- launch-файлы;
- конфигурационные файлы;
- при необходимости каталоги `msg`, `srv`, `action`;
- ресурсы, модели и другие вспомогательные данные.

В ROS 2 стандартным инструментом сборки является `colcon` [6], а базовой инфраструктурой сборки — `ament`. Для пакетов на C++ обычно используется `ament_cmake`, для пакетов на Python — `ament_python`.

Типичная последовательность работы с проектом ROS 2 включает:

1. создание рабочего пространства;
2. добавление пакетов в каталог `src`;
3. установку зависимостей;
4. сборку при помощи `colcon`;
5. настройку окружения командой `source install/setup.bash`;
6. запуск узлов и launch-сценариев.

Для автоматической установки системных зависимостей используется инструмент `rosdep` [7]. Он анализирует зависимости пакетов и устанавливает недостающие компоненты операционной системы. В учебной практике использование `rosdep` является обязательным этапом перед первой сборкой загруженного проекта.

1.3 Вычислительный граф ROS 2

Основой программной архитектуры ROS 2 является **вычислительный граф**. Он образуется из взаимодействующих программных компонентов и их коммуникационных интерфейсов.

Главным вычислительным элементом является **узел** (node). Узел представляет собой отдельный процесс, выполняющий некоторую часть общей задачи. Например, один узел может считывать показания лидарного датчика, другой — обрабатывать одометрию, третий — формировать карту окружающей среды, а четвёртый — вычислять управляющее воздействие для робота.

Взаимодействие узлов может строиться несколькими способами.

Топики (topics) используются для асинхронного обмена данными по модели publish/subscribe. Узел-издатель публикует сообщения в топик, а узлы-подписчики получают эти сообщения. Такой подход удобен для потоковых данных: изображений,

измерений датчиков, состояний системы, координат и управляющих сигналов.

Сервисы (services) используются в схеме request/response, когда один узел формирует запрос, а другой возвращает ответ. Сервисы удобны для разовых операций, например для вызова вычислительной процедуры, смены режима работы или получения текущего состояния системы.

Actions применяются для длительных операций. Этот механизм позволяет отправить цель, получать промежуточную обратную связь и при необходимости отменять выполнение. В робототехнике actions часто используются в задачах навигации и управления манипулятором.

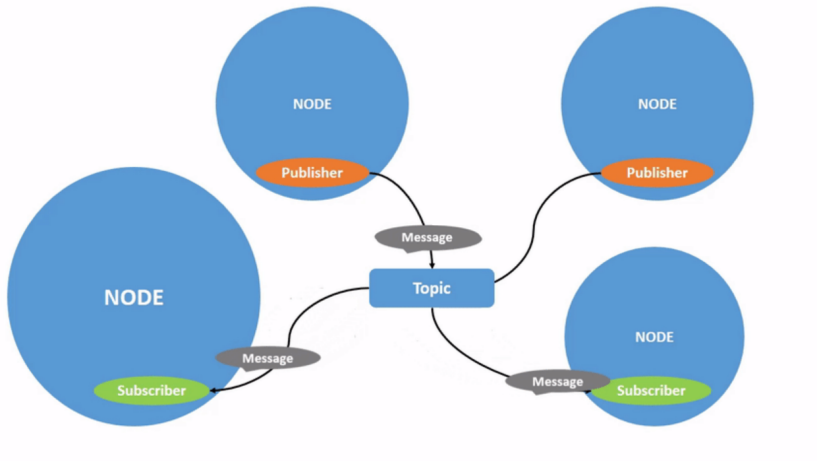


Рисунок 1.1. Иллюстрация вычислительного графа

Важной особенностью ROS 2 является отсутствие централизованного узла наподобие ROS Master. Обнаружение участников сети выполняется автоматически средствами DDS. Благодаря этому вычислительный граф является распределённым и не зависит от единой точки отказа.

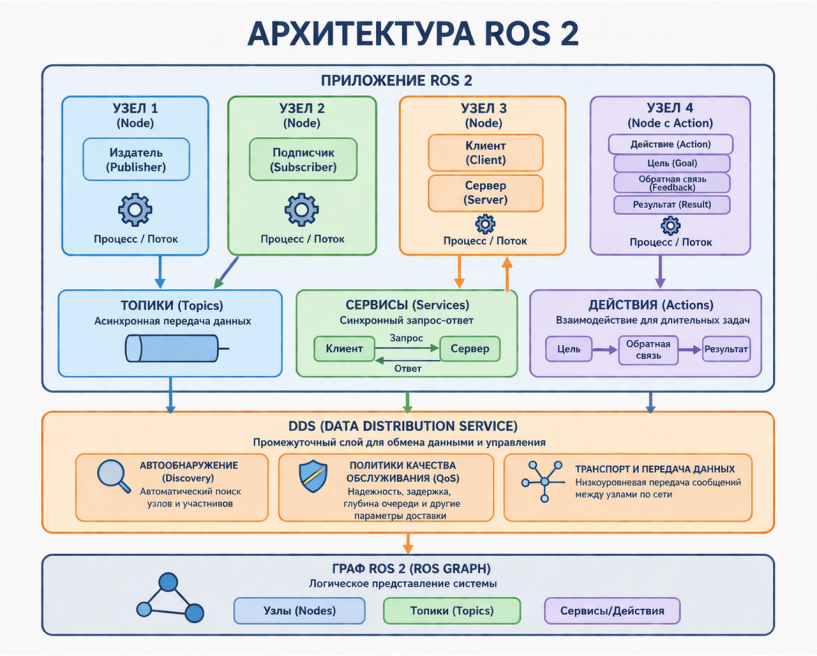


Рисунок 1.2. Упрощенное представление архитектуры ROS 2

1.4 Типы интерфейсов обмена данными

Обмен данными в ROS 2 основан на строго типизированных интерфейсах. Для их описания используются:

- файлы `.msg` — для сообщений;
- файлы `.srv` — для сервисов;
- файлы `.action` — для действий.

На основе этих описаний автоматически генерируется программный код для поддерживаемых языков. Поля сообщений могут иметь примитивные типы, массивы, вложенные сообщения, временные метки и длительности.

Использование строго определённых интерфейсов упрощает интеграцию компонентов и снижает вероятность ошибок при передаче данных. Именно поэтому в практических и лабораторных работах большое внимание уделяется созданию и использованию пользовательских типов `msg`, `srv` и `action`.

1.5 QoS и особенности обмена сообщениями

Одним из принципиальных отличий ROS 2 от ROS 1 является поддержка политик **QoS** (Quality of Service). QoS определяет правила доставки сообщений между узлами и позволяет адаптировать коммуникацию к требованиям конкретной задачи.

К основным политикам QoS относятся:

- надёжность доставки (*reliable*, *best effort*);
- история сообщений (*keep last*, *keep all*);
- глубина очереди;
- долговечность данных (*volatile*, *transient local*);
- время жизни сообщения;
- контроль активности publisher.

Например, поток изображений с камеры может передаваться в режиме *best effort*, если допустима потеря части кадров ради снижения задержек. Напротив, команды управления и критически важные данные обычно передаются в режиме *reliable*.

При проектировании приложений в ROS 2 необходимо учитывать, что publisher и subscriber должны иметь совместимые настройки QoS. В противном случае данные между ними передаваться не будут.

1.6 Launch-файлы, параметры и пространства имён

В реальной робототехнической системе обычно требуется одновременно запускать множество узлов, передавать им

параметры, задавать пространства имён, выполнять remapping имён и включать другие сценарии запуска. Для этого в ROS 2 используются **launch-файлы**.

В отличие от ROS 1, в ROS 2 основным форматом launch-файлов является Python. Это делает сценарии запуска более гибкими и позволяет использовать условия, переменные, вычисления и композицию сложных конфигураций.

Launch-файлы обычно размещаются в каталоге `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 котором запускается один узел

```
<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 котором запускаются узлы в различных пространствах имён

Другой важный механизм настройки приложений — это **параметры** (parameters). В ROS 2 параметры принадлежат конкретному узлу, а не хранятся в отдельном сервере параметров. Такой подход соответствует распределённой архитектуре системы.

Параметры используются для задания конфигурации узлов без изменения их исходного кода. Через них можно передавать частоты обновления, размеры модели робота, имена устройств, коэффициенты регуляторов и другие настройки.

Для предотвращения конфликтов имён в ROS 2 применяются **пространства имён** (namespaces). Они особенно полезны при работе с несколькими однотипными объектами, например несколькими роботами одной модели.

1.7 Интерфейс командной строки ROS 2

Для анализа и отладки вычислительного графа в ROS 2 широко используется интерфейс командной строки `ros2`. Он позволяет получать сведения о запущенных узлах, топиках, сервисах, параметрах и интерфейсах обмена без написания дополнительного кода.

К наиболее важным командам относятся:

- `ros2 node list`, `ros2 node info`;
- `ros2 topic list`, `ros2 topic echo`, `ros2 topic hz`;
- `ros2 service list`, `ros2 service call`;
- `ros2 action list`, `ros2 action send_goal`;
- `ros2 param list`, `ros2 param get`, `ros2 param set`;
- `ros2 interface list`, `ros2 interface show`;
- `ros2 bag record`, `ros2 bag play`, `ros2 bag info`;
- `ros2 pkg list`.

Отдельного внимания заслуживает инструмент `ros2 bag` [8], предназначенный для записи и воспроизведения сообщений. Команда `ros2 bag record` позволяет сохранить данные из указанных топиков в файл, а команда `ros2 bag play` — воспроизвести их. Это один из ключевых инструментов отладки робототехнических систем: он позволяет записать сеанс работы робота и затем многократно воспроизводить данные для анализа без повторного запуска эксперимента.

Именно эти инструменты активно применяются в практических и лабораторных работах, поэтому студенту необходимо уверенно владеть основными командами CLI.

1.8 Дискретность управления и ROS time

Стандартный подход к разработке программного обеспечения не предъявляет серьёзных требований к дискретизации выполнения программного кода; более того, зачастую архитектура программного обеспечения является последовательной. В ROS 2, и в частности в робототехнике, многие операции выполняются циклично с некоторой частотой дискретизации. Датчики выдают значения дискретно, и сразу после этого необходимо обновить управляющий сигнал.

В ROS 2 такой цикл обычно реализуется через timer, который вызывает callback с заданным периодом. Далее показан пример того, как реализовать дискретный и независимый контур управления на Python.

```
import rclpy
from rclpy.node import Node

class ControllerNode(Node):
    def __init__(self):
        super().__init__('controller_node')

        # Таймер с периодом 0.01 сек (100 Гц)
        self.timer = self.create_timer(0.01, self.control_step)
        self.last_time = self.get_clock().now()

    def control_step(self):
        current_time = self.get_clock().now()
        dt = (current_time - self.last_time).nanoseconds * 1e-9
        self.last_time = current_time

        # Основная логика управления может быть тут
        self.get_logger().info(f'dt = {dt:.3f} sec')

def main():
```

```
    rclpy.init()
    node = ControllerNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Частой ошибкой является реализация управления в функции callback, связанной с датчиками. Такое решение работает до тех пор, пока не появляются вычислительно дорогие операции.

Ещё одна важная особенность связана с хранением и обменом данными. Чтобы данные было легко хранить и иметь к ним доступ из любых функций обратного вызова (callback), рекомендуется не создавать локальные переменные, а хранить данные в экземпляре класса (self.variable_name) — тогда к ним будет доступ из любой функции обратного вызова.

В ROS 2 существует абстракция времени, которая отделяет «реальное время» от «времени системы»:

- System Time — системные часы (wall clock)
- Steady Time — монотонное время (без скачков)
- ROS Time — управляемое время (ключевое)

ROS Time может быть реальным (по умолчанию) или симулированным (например, в Gazebo). Рекомендуется использовать именно ROS Time практически всегда. Это важно, поскольку все данные передаются с привязкой ко времени (сенсоры, tf, фильтры используют timestamp). Связь с дискретностью заключается в том, что шаг дискретизации фактически определяется временем.

1.9 TF2 и описание геометрической структуры робота (URDF)

При разработке робототехнических систем необходимо задавать взаимное положение различных систем координат: корпуса робота,

датчиков, звеньев манипулятора, исполнительных органов и объектов внешней среды. В ROS 2 для этого используется система **TF2**.

TF2 предназначена для публикации, хранения и использования преобразований между системами координат с учётом времени. Она позволяет вычислять положение одного объекта относительно другого и поддерживает древовидную структуру связанных кадров.

Преобразования обычно передаются через топики `/tf` и `/tf_static`. Для визуального анализа дерева координатных систем и проверки корректности преобразований используется RViz2.

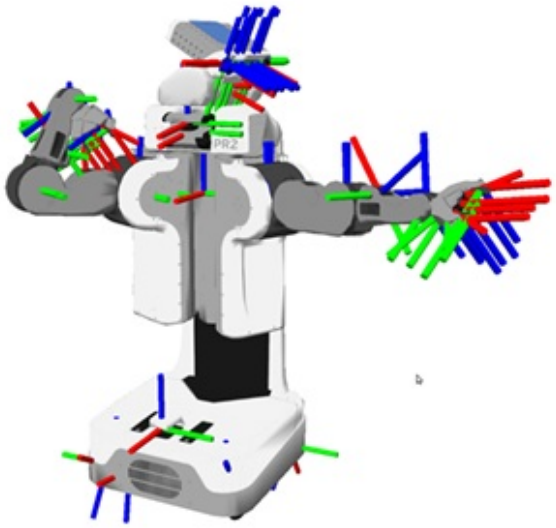


Рисунок 1.5. Пример визуализации систем координат в RViz2

Геометрическая и кинематическая структура робота задаётся через XML файл в формате **URDF** (Unified Robot Description Format), подробно описанной в официальной документации. В файле URDF описываются:

- звенья робота (размеры, расположение);

- соединения между звеньями (тип, направление оси вращения);
- визуальная геометрия (форма, расположение);
- геометрия столкновений (форма, расположение);
- инерционные параметры.

Описание URDF используется для визуализации модели, публикации TF-преобразований, а также в задачах симуляции и управления.

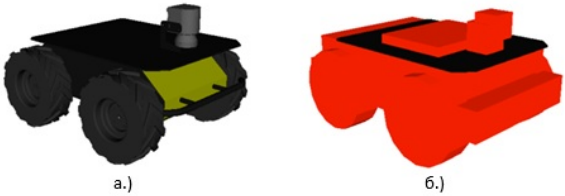


Рисунок 1.6. Пример модели робота: визуальная составляющая и модель столкновений

1.10 Xacro и визуализация робота

На практике описания роботов часто получаются большими и содержат повторяющиеся фрагменты. Для упрощения работы с URDF в ROS 2 широко используется **Xacro** — механизм XML-макросов, позволяющий параметризовать описание модели, выделять повторяющиеся элементы и разбивать модель на несколько файлов.

Использование Xacro делает описание робота более компактным, читаемым и удобным для сопровождения, что особенно важно в учебных проектах, где модель робота постепенно усложняется.

Для публикации состояния модели в ROS 2 часто используются:

- `robot_state_publisher`;

- `joint_state_publisher`;
- `joint_state_publisher_gui`.

Пакет `robot_state_publisher` публикует преобразования TF2 на основе модели робота и текущих состояний шарниров. Пакет `joint_state_publisher` формирует состояния шарниров, если они не поступают из симулятора или реального оборудования. Графическая версия этого пакета позволяет вручную изменять положения сочленений с помощью пользовательского интерфейса.

1.11 RViz2 как средство визуализации

RViz2 является стандартным инструментом визуализации данных в ROS 2. Он используется для отображения:

- систем координат;
- моделей роботов;
- облаков точек;
- лазерных сканов;
- изображений;
- маркеров;
- траекторий;
- карт и других типов данных.

RViz2 играет важную роль как при разработке программных компонентов, так и при отладке взаимодействия между ними. Именно через RViz2 студент может наглядно проверить корректность URDF-модели, TF-дерева, работы датчиков и результатов алгоритмов.

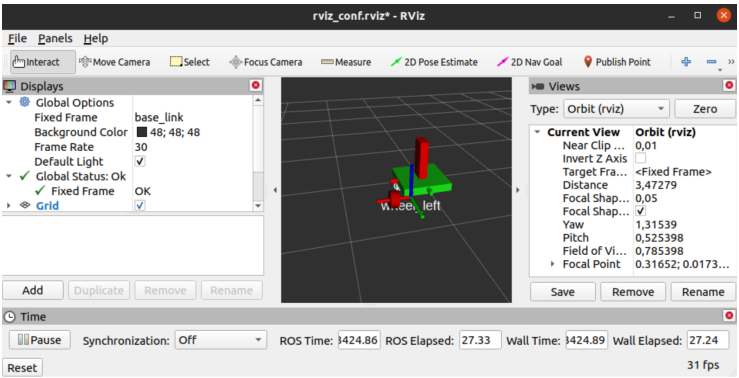


Рисунок 1.7. Окно программы RViz2

1.12 Имитационное моделирование в Gazebo

Имитационное моделирование является важным этапом разработки робототехнических систем, поскольку позволяет отлаживать алгоритмы без использования реального оборудования. В данном пособии для этих целей используется **Gazebo Harmonic** [9] — рекомендуемая версия симулятора для ROS 2 Jazzy.

Важно отличать современный Gazebo (ранее известный как Ignition, запускается командой `gz sim`) от устаревшего Gazebo Classic (команда `gazebo`, версии 9–11). В данном пособии используется исключительно новый Gazebo. Обмен данными между Gazebo и ROS 2 осуществляется через пакет `ros_gz_bridge` [10], который перенаправляет топики из транспорта Gazebo (`gz-transport`) в транспорт ROS 2 и наоборот. Конфигурация моста задаётся через YAML-файл, в котором указываются соответствия между топиками Gazebo и ROS 2, типы сообщений и направление передачи данных.

В связке с ROS 2 симулятор позволяет:

- загружать модели роботов;
- моделировать физику движения;
- рассчитывать столкновения;

- генерировать данные виртуальных датчиков;
- передавать в ROS 2 состояния модели и результаты измерений;
- принимать из ROS 2 управляющие воздействия.

Для моделирования в Gazebo, помимо самой модели робота, требуется описание сцены, параметров физического движка и объектов окружающей среды. В современных сценариях для этого широко используется формат SDF [11].

```
<?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 1.8. Пример простейшего файла мира 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 1.9. Пример задания параметров физического движка в Gazebo

Имитационное моделирование подготавливает основу для следующих разделов пособия, в которых рассматриваются работа датчиков, управление роботом и интеграция алгоритмов с моделью в виртуальной среде.

1.13 Управление приводами: ros2_control

Для организации управления сочленениями и приводами в ROS 2 используется фреймворк **ros2_control** [12]. Он предоставляет единый подход к работе как с симуляторами, так и с реальным оборудованием.

Основная идея `ros2_control` состоит в разделении системы на:

- описание аппаратного интерфейса;
- контроллеры;
- менеджер контроллеров;
- интерфейсы передачи команд и получения состояний.

Такой подход делает архитектуру управления модульной и переносимой. Один и тот же алгоритм высокого уровня может использоваться как в симуляции, так и на физическом роботе, если аппаратный слой реализует совместимые интерфейсы.

1.14 Датчики в симуляции и в реальной системе

Современный робот взаимодействует с окружающей средой через датчики. В учебных задачах обычно рассматриваются:

- лидары;
- камеры;
- датчики глубины;
- одометрия;
- данные состояния шарниров;
- инерциальные измерительные модули.

В симуляции датчики реализуются средствами Gazebo и соответствующих плагинов. При этом данные публикуются в топики ROS 2 в тех же форматах, которые используются и в реальной системе. Это позволяет разрабатывать и отлаживать алгоритмы обработки данных сначала в виртуальной среде, а затем переносить их на физическое оборудование с минимальными изменениями.

1.15 Связь с физическими роботами

Заключительным этапом учебного курса является работа не только с виртуальными моделями, но и с физическими робототехническими системами. В этом случае ROS 2 используется как единая программная среда для связи между высокоуровневыми алгоритмами и аппаратной частью робота.

В зависимости от архитектуры конкретной платформы взаимодействие с оборудованием может осуществляться через драйверы устройств, аппаратные интерфейсы `ros2_control`, последовательные интерфейсы, сетевые протоколы и специальные мосты между вычислительным модулем и микроконтроллером.

Таким образом, логика изучения ROS 2 в рамках данного пособия строится последовательно:

1. освоение среды Linux и базовых инструментов;
2. создание и сборка пакетов;
3. изучение узлов и средств обмена данными;
4. работа с launch-файлами, параметрами и интерфейсом командной строки;
5. описание и визуализация робота;
6. имитационное моделирование;
7. управление сочленениями и датчиками;
8. перенос решений на физическую робототехническую платформу.

Именно такая последовательность положена в основу практических и лабораторных работ, представленных в следующих разделах учебного пособия.

---

2 Вводная практическая работа. Основы работы в Linux и установка ROS 2

Цель работы

- Установка и базовая настройка ROS 2 Jazzy
- Знакомство с базовыми возможностями ROS2 по обмену данными и методами исследования готового пакета
- Изучение основных bash-команд для работы с ROS2 на базе операционной системы Ubuntu
- Знакомство с процессом создания рабочего окружения ROS2

В данной работе необходимо пользоваться теоретическими материалами и инструкциями по установке программного обеспечения ROS2 Jazzy.

2.1 Подготовка операционной системы

Необходимо установить операционную систему Ubuntu 24.04 (либо установить в качестве второй операционной системы, либо установить WSL2 (также допускается использование WSL1) на Windows 11 с образом Ubuntu 24.04 [13]), следуя инструкциям, приведенным в приложении А (15).

2.2 Установка редактора исходного кода (IDE)

Чтобы установить редактор VS Code [14], 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/или Docker контейнера. В VS code имеются полезные расширения, которые облегчат работу (их желательно установить): Python, Cmake, ROS2.

2.3 Первый запуск ROS2

В данном разделе необходимо научиться работать с базовыми компонентами ROS2 и начать работу с уже установленным ROS-пакетом, в который невозможно вносить правки, но его легко установить и запустить.

ROS Master В отличие от ROS 1, в ROS 2 отсутствует необходимость в запуске какого-либо обособленного процесса (ROS Master), система является полностью распределенной.

Перед первым запуском полезно будет удостовериться в том, что установка ROS2 прошла успешно. Для этого можно воспользоваться следующей командой, которая выводит значение переменной ROS_DISTRO из окружения Linux, в результате вы должны увидеть в терминале "jazzy".

```
printenv $ROS_DISTRO
```

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/jazzy/setup.bash
```

Установка готового ROS-пакета
Откройте новый терминал и выполните команду для установки ROS-пакета под названием ros-jazzy-turtlesim.

```
sudo apt update && sudo apt install ros-jazzy-turtlesim
```

Дождитесь появления сообщения об успешной установке нового пакета в окне терминала. Установленный только что ROS-пакет называется "Turtlesim" (Симулятор черепахи) - простой пакет, позволяющий выполнять имитационное моделирование черепахи, разработанное для изучения ROS 2. В данной работе при помощи данного пакета мы будем иллюстрировать то, что делает ROS 2 на самом базовом уровне, чтобы получить представление о том, что вы будете делать с реальным роботом или его симуляцией позже. Начнем изучение установленного пакета, воспользовавшись следующей командой:

```
ros2 pkg executables turtlesim
```

Данная команда позволяет проверить наличие каких-либо исполняемых файлов внутри ROS2-пакета turtlesim и выводит названия этих исполняемых файлов.

Среди результатов вывода предыдущей команды вы можете увидеть turtlesim_node, согласно документации к данному ROS-пакету [15], этот файл отвечает за имитационное моделирование черепахи.

Запуск симуляции черепахи

Запустите симуляцию черепахи из установленного по умолчанию пакета turtlesim. Для этого выполните команду:

```
ros2 run turtlesim turtlesim_node
```

В результате должно отобразиться новое окно с черепахой, как показано на рисунке 2.1.

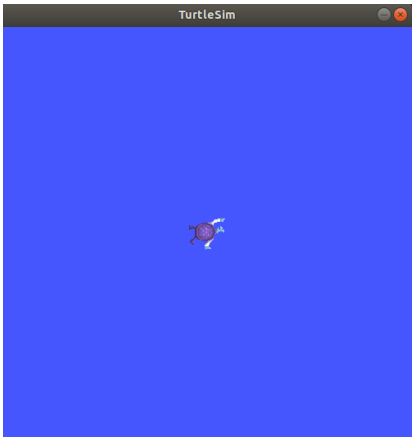


Рисунок 2.1. Симулятор черепахи turtlesim

Отправка управляющих команд Чтобы черепашка двигалась, необходимо отправить “правильное” сообщение в

“правильный” топик. Найдите названия “правильных” топиков, отвечающих за управление черепахой, с помощью ROS команды, которая выводит все активные топики:

```
ros2 topic list
```

Чтобы отправить сообщение в какой-либо топик, нам необходимо выяснить тип сообщения, который ожидает конкретный топик. Тип сообщения в топике "topic_name" (это название топика не существует, замените его на интересующий вас) выводится командой:

```
ros2 topic type /topic_name
```

Проверьте, за что отвечает каждый топик, подставляя вместо topic_name названия топиков из списка топиков, который выдала команда `ros2 topic list`. Для просмотра структуры сообщения с названием типа 'msg_type' можно воспользоваться следующей командой:

```
ros2 interface show msg_type
```

Публикация сообщений Сообщение для управления можно отправить из терминала (командной строки):

```
ros2 topic 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}}'
```

Такой способ публикации удобен для изучения пакета или во время отладки, чаще всего сообщения отправляются из программного пакета ROS какой-либо нодой.

Управление черепахой с клавиатуры Для управления черепахой с клавиатуры можно воспользоваться узлом `turtle_teleop_key`, который имеется в ROS пакете `turtlesim` и выполняет роль посредника. Когда вы нажимаете клавиши, данный узел отлавливает это событие и отправляет соответствующие сообщения в turtlesim через соответствующие топики. Запустите:

```
ros2 run turtlesim turtle_teleop_key
```

Теперь можно управлять черепахой с клавиатуры. Подробная информация о том, какие клавиши отвечают за управление, вы увидите в терминале.

Просмотр данных

После запуска turtlesim регулярно публикует сообщения о своем текущем положении в топик `/turtle1/pose`. Выведите в терминале данные о положении черепахи, которые приходят в топик `/turtle1/pose`, воспользовавшись командой:

```
ros2 topic echo /turtle1/pose
```

Проверьте частоту публикации сообщений:

```
ros2 topic hz /turtle1/pose
```

2.4 Визуализация вычислительного графа

Используйте утилиту `rqt_graph`, чтобы отобразить вычислительный граф (показывает потоки данных между узлами):

```
rqt_graph
```

Обратите внимание, команду `rqt_graph` необходимо выполнить в новом терминале, при этом обмен данными между узлами должен быть уже запущен, как было показано ранее (иначе будет нечего визуализировать). В верхней панели окна `rqt_graph` можно настраивать, что именно визуализировать, например, полезно выбирать не только Nodes (выбран по умолчанию), а выбрать "Nodes and Topics".

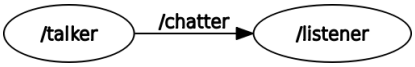


Рисунок 2.2. Иллюстрация передачи сообщения между процессами (узлами) через вычислительный граф (rqt_graph)

Ответьте на следующие вопросы:

- Через какой топик происходит обмен данными?
- Посмотрите на вычислительный граф и определите название узлов, которые обмениваются данными.
- Какие топики активны, но не получают никакой информации?
- Какие узлы не публикуют никаких сообщений?

2.5 Создание рабочего окружения и 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 colcon

Для работы с пользовательскими пакетами, которые разработчик может редактировать, необходимо установить систему сборки colcon (если она не установлена уже):

```
sudo apt install python3-colcon-common-extensions
```

2.7 Создание ROS 2 workspace

Следующие команды создают папку workspace, внутри которой еще одна папка src, запускают сборку всех имеющихся в окружении пакетов. Рекомендуется выполнять эти команды из домашнего каталога вашего linux пользователя.

```
mkdir -p workspace/src
cd workspace
colcon build
```

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

```
source install/setup.bash
```

2.8 Создание ROS пакета

Создадим свой собственный ROS 2 пакет, выполнив следующие команды:

```
cd src
ros2 pkg create --build-type ament_python demo_package
```

После создания пакета можно увидеть, что в нём уже имеются некоторые файлы. Чтобы понять их назначение, необходимо обратиться к лекционным материалам или к ранее рассмотренной структуре пакета в данном учебном пособии.

2.9 Исследование пакета

Предположим, что мы нашли новый пакет в интернете, в первую очередь необходимо его клонировать на свой компьютер.

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

Внутри склонированного репозитория вы найдете папку task1, внутри которой уже будет папка с названием "example_package" - это ROS2 пакет. Его необходимо скопировать в рабочее окружение ros2_ws, в папку src. В дальнейшем вся работа с ROS-пакетами будет выполняться в рабочем окружении ros2_ws. Данный пакет содержит необходимый минимум:

1. `example_package` - папка с Python файлами
2. `package.xml` - свойства пакета, такие как название, версия, авторы и зависимости от других пакетов.
3. `psetup.py` - установочный скрипт Python-пакета, используется системой сборки colcon.

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

2.10 Пример Паблишера

Далее найдем исполняемый файл в папке `example_package`, файл с названием `publisher.py`, который будет содержать программный код, изучите его, попробуйте разобраться, что именно делает данный пакет?

2.11 Пример Сабскрайбера

Найдите файл `subscriber.py`, изучите его содержимое. Определите, что именно делает этот файл и как он взаимосвязан с предыдущим файлом.

Запустите паблишер и сабскрайбер в разных терминалах с помощью команды `ros2 run`. С помощью `rqt_graph` изучите, как эти два исполняемых файла (процесса) взаимосвязаны.

2.12 Запись и воспроизведение данных (ros2 bag)

Запустите turtlesim и управляйте черепахой с клавиатуры (turtle_teleop_key). Одновременно в отдельном терминале запишите данные из топика с положением черепахи:

```
ros2 bag record /turtle1/pose -o my_bag
```

Остановите запись (Ctrl+C) через 10–15 секунд. Просмотрите информацию о записи:

```
ros2 bag info my_bag
```

Ответьте на вопросы: сколько сообщений было записано? Какова длительность записи?

Воспроизведите записанные данные:

```
ros2 bag play my_bag
```

В другом терминале убедитесь, что данные поступают:

```
ros2 topic echo /turtle1/pose
```

Вопросы для самопроверки

- Какой элемент архитектуры ядра ROS отвечает за отслеживание существующих узлов вычислительного графа?
- Как называется утилита для визуализации вычислительного графа в ROS?
- Возможно ли редактирование ROS пакета, если его установка выполнена через apt?
- В какой каталог файловой системы разрешается размещать программный код на языке `Python` внутри ROS-пакета?
- Для чего используется ros2 bag record?

3 Практическая работа №1. 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
- Получение навыков создания файлов запуска

3.1 Создание ROS пакета

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

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

```
cd src
ros2 pkg create --build-type ament_python turtle_controller
colcon build
```

После создания пакета необходимо проверить, что папка ROS пакета находится в `src`, а также его содержимое соответствует нашим ожиданиям, например, можно проверить `package.xml`.

Примечание: Папка `turtle_controller` внутри ROS пакета (с аналогичным названием, как и ROS пакет) создается автоматически, именно во внутренней папке мы будем дальше создавать исполняемые файлы.

3.2 Создание ноды с паблишером и сабскрайбером

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

Обратите внимание, в этих командах используются относительные пути, они указаны с учетом того, что вы находитесь внутри папки ROS-пакета!

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

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

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist

class TurtleSuperNode(Node):
    def __init__(self):
        super().__init__('turtle_super_node')
        self.get_logger().info("Node started...")

        # Publisher на /turtle1/cmd_vel
        self.publisher = self.create_publisher(Twist,
        ↪  '/turtle1/cmd_vel', 10)

        # Subscriber на /turtle1/pose
        self.subscription = self.create_subscription(
            Pose,
            '/turtle1/pose',
            self.pose_callback,
            10
        )

    def pose_callback(self, pose: 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
```

```
        self.publisher.publish(cmd)
        self.get_logger().info("Super node sending new msg...")

def main(args=None):
    rclpy.init(args=args)
    node = TurtleSuperNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

Чтобы мы могли запустить этот узел, нам необходимо указать `entry_points` в файле setup.py. Откройте этот файл и укажите параметры, как показано ниже.

```
entry_points={
    'console_scripts': [
        'super_node = my_turtle_pkg.super_node:main',
    ],
},
```

Обратите внимание! Ваш пакет называется turtle_controller и название пакета в entry_points нужно указывать правильно (в фрагмент выше нужно внести правки).

Данный пример реализует простейший релейный регулятор для управления моделью черепахи (моделирование черепахи должно быть запущено командой `ros2 run turtlesim turtlesim_node`).

Запустите только что созданный файл с помощью команды ros2 run, как показано ниже.

```
ros2 run turtle_controller super_node
```

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.

Задание для самостоятельного выполнения (опционально): Измените регулятор, чтобы черепашка перемещалась только вдоль края рабочего поля окна.

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

В ROS 2 появилась такая особенность, которая не позволяет создавать пользовательские типы сообщений внутри пакета, использующего ament_python, поэтому необходимо создать отдельный пакет использующий ament_cmake, при этом писать программы на C++ внутри этого пакета нас это не обязывает.

Создайте новый пакет в рабочем окружении:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0
↪  custom_msgs_pkg
```

Внутри ROS пакета создайте папку msg для хранения структуры новых сообщений:

```
mkdir msg
```

Создайте файл структуры сообщения с названием Person.msg (воспользуйтесь редактором VS Code или nano).

Отредактируйте файл, изменив поля в соответствии с примером ниже:

```
string name
string secondname
int32 age
```

На этом этапе только что созданный пользовательский тип сообщений будет недоступен для использования.

3.3.2 Настройка сборки сообщений

Чтобы сборка новых типов сообщений выполнялась каждый раз при вызове команды build, необходимо указать зависимости от следующих стандартных пакетов `rosidl_default_generators`, `rosidl_default_runtime` и `rosidl_interface_packages`.

Отредактируйте `package.xml`, добавляя строки (Убедитесь, что у вас нет дубликатов этих тегов.):

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Также необходимо отредактировать `CMakeLists.txt` путем добавления зависимости от `rosidl_default_generators` и `rosidl_generate_interfaces`, как показано ниже, с указанием ранее созданного нами файла, определяющего пользовательский тип сообщений `"msg/Person.msg"`

```
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

# Генерация интерфейсов
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Person.msg"
)

ament_export_dependencies(rosidl_default_runtime)
ament_package()
```

Вернитесь в рабочее пространство (workspace) и выполните повторную сборку пакета:

```
colcon build
```

Не забудьте, что после каждой сборки необходимо выполнить source, как было показано в прошлой практической работе (в дальнейшем это действие будет само собой подразумеваться).

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

```
ros2 interface show <package_name>/msg/<MessageName>
```

3.4 Создание простейшего регулятора по положению

Далее разберем пример реализации управляющей программы, которая часто встречается в робототехнике. В симуляторе черепахи turtlesim у нас имеется объект управления - черепаха. Его состояние характеризуется координатами x, y и углом поворота θ, а управляющим воздействием являются его линейная и угловая скорость. Таким образом, для управления положением черепахи необходимо считывать ее положение и ориентацию с заданной дискретностью, вычислять ошибку по углу поворота и расстоянию до желаемого положения, затем рассчитывать скорость таким образом, чтобы она уменьшала ошибку по углу и положению.

```
import math

import rclpy
from rclpy.node import Node
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist


class ControllerNode(Node):
    def __init__(self):
        super().__init__('controller_node')
        # Целевая точка
```

```
        self.target_x = 6.0
        self.target_y = 6.0
        # Коэффициенты регулятора
        self.k_linear = 1.5
        self.k_angular = 6.0
        # Точность достижения цели
        self.distance_tolerance = 0.05
        self.pose = None
        # Подписка на pose черепахи
        self.pose_subscriber = self.create_subscription(
            Pose,
            '/turtle1/pose',
            self.pose_callback,
            10
        )
        # Публикация команд скорости
        self.cmd_publisher = self.create_publisher(
            Twist,
            '/turtle1/cmd_vel',
            10
        )
        # Таймер с периодом 0.01 сек, 100 Гц
        self.timer = self.create_timer(0.05, self.control_step)

    def pose_callback(self, msg):
        self.pose = msg

    def normalize_angle(self, angle):
        while angle > math.pi:
            angle -= 2.0 * math.pi
        while angle < -math.pi:
            angle += 2.0 * math.pi
        return angle

    def control_step(self):

        if self.pose is None:
            self.get_logger().info('Waiting for turtle
            ↪  pose...')
            return
```

```
        dx = self.target_x - self.pose.x
        dy = self.target_y - self.pose.y

        distance = math.sqrt(dx ** 2 + dy ** 2)
        target_angle = math.atan2(dy, dx)
        angle_error = self.normalize_angle(target_angle -
        ↪  self.pose.theta)

        cmd = Twist()

        if distance < self.distance_tolerance:
            cmd.linear.x = 0.0
            cmd.angular.z = 0.0
            self.cmd_publisher.publish(cmd)
            self.get_logger().info('Target reached')
            return

        # Если черепаха сильно не направлена на цель - сначала
        ↪  поворачиваем
        if abs(angle_error) > 0.2:
            cmd.linear.x = 0.0
        else:
            cmd.linear.x = self.k_linear * distance

        cmd.angular.z = self.k_angular * angle_error

        self.cmd_publisher.publish(cmd)

        self.get_logger().info(
            f'x={self.pose.x:.2f}, y={self.pose.y:.2f}, '
            f'target=({self.target_x:.2f},
            ↪  {self.target_y:.2f}), '
            f'distance={distance:.2f},
            ↪  angle_error={angle_error:.2f}'
        )


def main():
    rclpy.init()
    node = ControllerNode()
    rclpy.spin(node)
```

```
    node.destroy_node()
    rclpy.shutdown()


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

3.4.1 Задание для самостоятельного выполнения

1. Создать новый тип сообщений, чтобы он содержал поля с координатами `x, y`.

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

3. Проверить результат с помощью команды: `ros2 topic echo /position2d`.

3.5 Запуск нескольких узлов одновременно

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

Вернитесь к редактированию пакета turtle_controller и внутри ROS пакета создайте папку для хранения файлов запуска:

```
mkdir launch
```

Создайте файл запуска:

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

Отредактируйте файл, добавив фрагмент кода для создания двух узлов, первая будет запускать имитационное моделирование черепахи, а вторым узлом будет выступать регулятор, который вы уже написали.

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
            output='screen'
        ),
        Node(
            package='turtle_controller',
            executable='super_node',
            name='super_node',
            output='screen'
        )
    ])
```

Поскольку ваш ROS-пакет использует ament_python, необходимо в setup.py указать путь к файлу запуска:

```
data_files=[
('share/' + package_name + '/launch',
↪  ['launch/my_setup.launch.py']),
],
```

Если бы вы использовали ament_cmake, потребовалось бы указать в CMakeLists.txt путь:

```
  install(DIRECTORY launch/
     DESTINATION share/${PROJECT_NAME}/launch
   )
```

Вызвать файл запуска мы можем следующей командой:

```
   ros2 launch turtle_controller my_setup.launch.py
```

В результате вы должны увидеть запуск модели черепахи и узла, который создается в super_node.py. Но этого не случится, потому что вы, вероятно, забыли сделать build и source. Пересоберите данный пакет и подгрузите переменные окружения (source), как было ранее показано.

В файлах запуска (launch-файлах) ROS 2 можно задать пространство имён (namespace), в котором будет запускаться узел. Это удобно, например, для запуска нескольких экземпляров одних и тех же узлов, но в изолированных контекстах.

В следующем примере super_node.py запускается внутри пространства имен с названием ns1, и таким образом все относительные имена топиков и узлов будут начинаться с /ns1/*.

```
Node(
    package='turtle_controller',
    executable='super_node',
    namespace='ns1',  # ← вот здесь задаётся namespace
    name='super_node',
    output='screen'
)
```

Задание для самостоятельного выполнения

Отредактируйте `my_setup.launch`, чтобы одновременно с управляющей программой запускалась имитационная модель черепахи из пакета `turtlesim` и при этом у них все имена ресурсов (топиков, ноды, и т.д.) начинались с ns3.

Вопросы для самопроверки

- Достаточно ли создать файл с расширением **.msg**, чтобы новый тип сообщений стал доступен в ROS 2?
- Какой командой необходимо воспользоваться для вызова файла запуска?

4 Практическая работа №2. Создание пользовательских интерфейсов (service и action)

Цель работы:

- Ознакомиться с механизмами сервисов и действий (services и actions) в ROS 2.
- Научиться создавать пользовательские интерфейсы: собственные `.srv` и `.action` файлы.
- Освоить особенности использования сервера и клиента для интерфейсов service и action.

4.1 Создание service

Перед началом работы рекомендуется вспомнить, что **Сервисы** (services) обеспечивают взаимодействие в режиме «запрос — ответ»: один узел формирует запрос (запрос может быть даже пустым), а другой возвращает ответ. Сервисы удобны для разовых операций, например для вызова вычислительной процедуры, смены режима работы или получения текущего состояния системы (или режима работы).

Важной особенностью сервиса является то, что он блокирует процесс (узел), из которого идёт вызов (то есть функции обратного вызова callback в клиенте не будут обрабатываться до тех пор, пока ответ не будет получен от сервера).

Практическая реализация состоит из 3 этапов:

- создание формата данных — файла `.srv`;
- создание сервера, который будет обрабатывать запросы;
- создание клиента, который будет отправлять запросы.

В общем случае клиентом может быть любой вычислительный узел (нода), также можно вызывать сервис (отправлять запрос серверу сервиса) из терминала.

В качестве учебной задачи в рамках текущей лабораторной работы вам предстоит реализовать сервис, принимающий 3 числа

и возвращающий результат суммирования. Поскольку нам необходимо создать новый тип сервиса, нам нужен дополнительный ROS-пакет, который будет содержать исключительно форматы данных .srv и использовать систему сборки ament_cmake. Это необходимо, поскольку мы используем в качестве основного языка разработки Python, а ROS-пакеты с Python не поддерживают компиляцию новых типов данных.

Создайте новый пакет в рабочем окружении с именем `custom_interfaces_pkg`, воспользовавшись командой:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0
↪  custom_interfaces_pkg
```

4.1.1 Создание структуры service

Внутри ROS пакета создайте папку `srv` для хранения структуры новых сервисов, для этого можно перейти внутрь папки и в терминале выполнить:

```
mkdir srv
```

Перейдите внутрь папки командой cd:

```
cd srv
```

Создайте файл структуры сообщения с названием AddThreeInts.srv внутри папки srv:

```
code AddThreeInts.srv
```

Отредактируйте файл, изменив поля в соответствии с примером ниже (a,b,c - названия полей в запросе, sum - название единственного поля в ответе):

```
int64 a
int64 b
int64 c
---
int64 sum
```

4.1.2 Настройка сборки нового формата данных

Чтобы сборка новых типов сообщений выполнялась каждый раз при вызове команды `colcon build`, необходимо подключить зависимости от следующих стандартных пакетов `rosidl_default_generators`, `rosidl_default_runtime` и `rosidl_interface_packages`.

Отредактируйте `package.xml`, добавив следующие строки. Убедитесь, что у вас нет дубликатов этих тегов.

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Также необходимо отредактировать `CMakeLists.txt` путем добавления зависимости от `rosidl_default_generators` и `rosidl_generate_interfaces`, как показано ниже, с указанием ранее созданного нами файла, определяющего пользовательский тип сервиса `"srv/AddThreeInts.srv"`. Всегда при добавлении новых строк проверяйте, что вы не создаете дублирования (дублирование приведет к ошибке сборки).

```
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

# Генерация интерфейсов
rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/AddThreeInts.srv"
)

ament_export_dependencies(rosidl_default_runtime)
ament_package()
```

Вернитесь в рабочее пространство (workspace) и выполните повторную сборку пакета:

```
colcon build
```

Не забудьте, что после каждой сборки необходимо выполнить команду `source`, как это было показано в прошлой

практической работе (в дальнейшем это действие будет само собой подразумеваться).

Проверьте доступность созданного типа сервиса в ROS и отобразите содержимое созданной структуры следующей командой, заменив <package_name> на название вашего ROS-пакета:

```
ros2 interface show <package_name>/srv/<ServiceName>
```

Замечание. Не забывайте в команде для терминала указывать ваши актуальные названия пакета и название сервиса. Всегда читайте полностью команды, которые вы копируете из внешних источников, включая данное учебное пособие.

Также сервис можно вызвать напрямую из терминала без запуска клиентского узла. Для этого используется команда `ros2 service call`, в которой указываются имя сервиса, тип сервиса и значения полей запроса. Например, для сервиса `/add_three_ints` с типом `custom_interfaces_pkg/srv/AddThreeInts` команда будет выглядеть следующим образом:

```
ros2 service call /add_three_ints
↪  custom_interfaces_pkg/srv/AddThreeInts "{a: 1, b: 2, c: 3}"
```

В результате сервер получит три числа, выполнит их сложение и вернёт ответ в поле `sum`.

4.1.3 Создание клиента и сервера

После создания пользовательского типа сервиса (формата данных) необходимо реализовать программный код сервера и клиента, которые будут использовать этот интерфейс.

1. Создадим новый пакет, использующий `ament_python`, в котором будет храниться прикладной программный код:

   ```
   ros2 pkg create --build-type ament_python --license
   ↪  Apache-2.0 client_server_interfaces_pkg
   ```

2. Перейдите в только что созданный ROS-пакет и в подпапке `client_server_interfaces_pkg` создайте файл сервера `srv_server.py`. В этом файле реализуется сервер, который принимает запрос с тремя числами и возвращает их сумму.

3. Создайте файл клиента `srv_client.py`. Клиент отправляет три числа на сервер и выводит полученный ответ.

Создайте файл с названием `srv_server.py` и следующим программным кодом сервера сервиса:

```
from custom_interfaces_pkg.srv import AddThreeInts

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddThreeInts,
        ↪  'add_three_ints', self.add_three_ints_callback)

    def add_three_ints_callback(self, request, response):
        response.sum = request.a + request.b + request.c
        self.get_logger().info('Incoming request\na: %d b: %d
        ↪  c: %d' % (request.a, request.b, request.c))

        return response

def main(args=None):
    rclpy.init(args=args)
    minimal_service = MinimalService()
    rclpy.spin(minimal_service)
    rclpy.shutdown()

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

Создайте файл с названием srv_client.py и следующим программным кодом клиента сервиса:

```
from custom_interfaces_pkg.srv import AddThreeInts
import sys
import rclpy
from rclpy.node import Node
class MinimalClientAsync(Node):
    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddThreeInts,
        ↪  'add_three_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available,
            ↪  waiting again...')
        self.req = AddThreeInts.Request()

    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.req.c = int(sys.argv[3])
        self.future = self.cli.call_async(self.req)

def main(args=None):
    rclpy.init(args=args)
    minimal_client = MinimalClientAsync()
    minimal_client.send_request()
    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
                minimal_client.get_logger().info('Service
                ↪  Responce recienved! ')
                minimal_client.get_logger().info('Result of
                ↪  add_three_ints: for %d + %d + %d = %d' %
            (minimal_client.req.a, minimal_client.req.b,
            ↪  minimal_client.req.c, response.sum))
            except Exception as e:
                minimal_client.get_logger().info('Service call
                ↪  failed %r' % (e,))
        else:
            # тут можно что-то делать, пока не пришел ответ от
            ↪  сервиса
```

```
            pass
        break
    minimal_client.destroy_node()
    rclpy.shutdown()

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

Важно! Чтобы иметь возможность запускать эти исполняемые файлы, необходимо вспомнить материалы предыдущей лабораторной работы, касающиеся добавления `entry_points`. После указания `entry_points` запустите в первом терминале сервер сервиса, а во втором — клиент сервиса (который будет отправлять запрос серверу и ожидать ответа).

При запуске клиента необходимо дополнительно указать в конце команды через пробел три числа, которые внутри программного пакета будут доступны как `sys.argv[1]`, `sys.argv[2]` и `sys.argv[3]`. Эти три числа клиент должен отправить серверу, получить результат их сложения и вывести его в терминал.

4.1.4 Задание для самостоятельного выполнения

1. Исправьте программный код сервера, чтобы Сервер обрабатывал запросы с задержкой N секунд, где N - случайное число от 2 до 5.

2. Создайте новый тип service, чтобы в запросе он содержал поля линейной и угловой скорости `linear, angular`, и возвращал флаг `success` (который может иметь значение True или False).

3. Напишите программный код для сервера и клиента сервиса `set_turtle_velocity`, который отправляет указанные вами линейную и угловую скорости в топик `/turtle1/cmd_vel`

4.2 Создание action

Перед началом практической работы будет полезно вспомнить, что **action** применяются для длительных операций. Этот механизм позволяет отправить цель, получать промежуточную обратную связь и при необходимости отменять выполнение. В робототехнике

action часто используются в комплексных задачах навигации и управления манипулятором, поэтому на начальном этапе изучения бывает сложно понять, зачем такой механизм коммуникации нужен.

Важной особенностью механизма action является то, что на его серверной части имеется бесконечный цикл, который запускается при получении цели (цель может содержать любые данные) и с заданной регулярностью отправляет обратную связь. Как мы знаем из Практической работы 1, для реализации управляющего контура нам также необходим бесконечный цикл, который реализуется через механизм rclpy.spin(), следовательно, возникает конфликт, поскольку в одном процессе не могут одновременно работать два бесконечных цикла, где один вложен в другой. В таком случае необходимо воспользоваться возможностью распараллеливания процессов (multi threads) [16].

В данной практической работе серверу необходимо принять в качестве цели количество секунд, которые нужно отсчитывать и в качестве обратной связи отправлять процент уже пройденного времени, а когда обратный отсчет дойдет до 0, то action вернет результат. В данной задаче процесс изменения состояния объекта (которым мы обычно управляем, но гораздо важнее, что мы его можем наблюдать и получать актуальные данные о состоянии) будет имитироваться при помощи ожидания в 1 секунду. Это очень упрощенный пример, однако он позволяет сфокусироваться на action без использования Executors.

4.2.1 Создание структуры action

Внутри ROS пакета custom_interfaces_pkg создайте папку action для хранения структуры данных:

```
mkdir action
```

Создайте файл структуры данных с названием MyTask.action и отредактируйте файл, изменив поля в соответствии с примером ниже:

```
# Goal
int32 duration_seconds
---
# Result
bool success
---
# Feedback
float32 progress_percent
```

4.2.2 Настройка сборки формата данных

Так как мы уже зависимости настроили перед сборкой сервисов, добавлять новые зависимости в package.xml не потребуется.

В `CMakeLists.txt` необходимо указать в разделе `rosidl_generate_interfaces` созданную нами структуру action `"action/MyTask.action"` как это показано ниже:

```
# Генерация интерфейсов
rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/AddThreeInts.srv"
  "action/MyTask.action"
)
```

Вернитесь в рабочее пространство и выполните повторную сборку пакета:

```
colcon build
```

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

```
ros2 interface show <package_name>/action/<ActionName>
```

4.2.3 Создание клиента и сервера

Аналогично сервису, после создания пользовательского типа action необходимо реализовать сервер и клиент.

1. Используем уже созданный пакет (переходим в этот ROS-пакет) `client_server_interfaces_pkg`. Создайте файл `action_server.py`. В этом файле реализуется сервер, который принимает цель (в данном случае, это длительность работы), выполняет вычисления с отсчетом времени и отправляет клиенту промежуточные значения прогресса (feedback) и итоговый результат.

2. В той же папке создайте файл `action_client.py`. Клиент отправляет запрос серверу, получает промежуточные результаты и выводит их в терминал до завершения задачи.

Создайте файл с программным кодом сервера action_server.py и изучите комментарии, которые добавлены в исходный код программы:

```
import time
import rclpy
from rclpy.node import Node
from rclpy.action import ActionServer
from custom_interfaces_pkg.action import MyTask

class TaskActionServer(Node):
    def __init__(self):
        super().__init__('task_action_server')
        self._server = ActionServer(
            self,
            MyTask,
            'do_task',
            self.execute_callback
        )

    def execute_callback(self, goal_handle):
        self.get_logger().info(f'[NEW GOAL] Работать на
        ↪  протяжении {goal_handle.request.duration_seconds}
        ↪  секунд')

        feedback = MyTask.Feedback()
        total = goal_handle.request.duration_seconds

        for i in range(total):
```

```
            if goal_handle.is_cancel_requested:
                goal_handle.canceled()
                self.get_logger().warn('[CANCEL] Клиент отменил
                ↪ цель')
                return MyTask.Result(success=False)

            progress = float(i + 1) / total * 100.0
            feedback.progress_percent = progress
            goal_handle.publish_feedback(feedback)
            self.get_logger().info(f'[FEEDBACK] Прогресс:
            ↪ {progress:.1f}%')

            time.sleep(1)

        goal_handle.succeed()
        self.get_logger().info('[RESULT] Задача завершена
        ↪ успешно')

        result = MyTask.Result()
        result.success = True
        return result

def main(args=None):
    rclpy.init(args=args)
    node = TaskActionServer()
    rclpy.spin(node)
    rclpy.shutdown()

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

Создайте файл с программным кодом клиента сервиса action_client.py:

```
import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient
from custom_interfaces_pkg.action import MyTask

class TaskActionClient(Node):
    def __init__(self):
```

```
        super().__init__('task_action_client')
        self._client = ActionClient(self, MyTask, 'do_task')

    def send_goal(self, duration_sec=5):

        self._client.wait_for_server() # Ждём, пока сервер
        ↪ будет доступен
        goal_msg = MyTask.Goal()
        goal_msg.duration_seconds = duration_sec

        # Отправляем цель асинхронно
        self._client.send_goal_async(
            goal_msg,
            feedback_callback=self.feedback_callback
        ).add_done_callback(self.goal_response_callback)

    def goal_response_callback(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().warn('Цель отклонена сервером')
            return

        self.get_logger().info('Цель принята. Ожидаем
        ↪ результат...')
        self._get_result_ = goal_handle.get_result_async()
        self._get_result_.add_done_callback(self.res_callback)

    def feedback_callback(self, feedback_msg):
        progress = feedback_msg.feedback.progress_percent
        self.get_logger().info(f'Feedback: {progress:.1f}%')

    def res_callback(self, future):
        result = future.result().result
        self.get_logger().info(f'Результат:
        ↪ success={result.success}')

        rclpy.shutdown()

def main(args=None):
    rclpy.init(args=args)
    client = TaskActionClient()
```

```
    client.send_goal(duration_sec=5)
    rclpy.spin(client) # упрощенный спин для клиента

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

Для запуска вновь созданных исполняемых файлов необходимо добавить новые entry_points, затем необходимо выполнить повторную сборку пакета и обновление переменных окружения (build и source).

При разработке action бывает полезно протестировать работоспособность action сервера. В первую очередь необходимо запустить отдельно сервер и проверить, стал ли он доступен через терминал:

```
ros2 action list
```

Если сервис доступен в терминале, его можно вызвать без использования action клиента (исключая возможные проблемы на стороне клиента), следующей командой в терминале:

```
ros2 action send_goal /do_task
↪  custom_interfaces_pkg/action/MyTask "{duration_seconds:
↪  10.0}" --feedback
```

Если в терминале начинают приходить данные feedback от сервера action, значит все работает корректно на стороне сервера.

Задание для самостоятельного выполнения

1. В предыдущем примере клиент задает всегда одно и то же значение в качестве цели, отредактируйте его программный код таким образом, чтобы при его вызове можно было передавать число в терминале (по аналогии с тем, как это было сделано в примерах с сервисами)

2. Написать программный код для сервера и клиента экшена `go_to_pose`, клиент должен отправлять координаты target_x,target_y, а сервер должен их принимать и управлять

черепашкой в `turtlesim` путем отправки необходимого управления в топик `/turtle1/cmd_vel`, выводя в консоль оставшееся расстояние до цели. При достижении цели, сервер должен вернуть результат (True).

Вопросы для самопроверки

- В чём принципиальное отличие сервиса и экшена от обычных топиков в ROS 2?
- Из каких частей состоит описание .srv и .action файлов и зачем нужны разделители —?
- Какие зависимости необходимо добавить в package.xml и CMakeLists.txt, чтобы сгенерировать пользовательские интерфейсы?
- Как проверить, что новый сервис или экшен успешно собран и доступен в ROS 2?
- В каких случаях удобнее использовать сервис, а в каких — экшен?

5 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 2 и подготовка к лабораторной работе №3.

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

5.1 Создание первого urdf файла

В данной практической работе используются материалы из официальной документации ROS Jazzy [17].

1. Создайте новый пакет в рабочем пространстве

```
cd <your_ros2_ws>/src/
ros2 pkg create --build-type ament_python --license
↪  Apache-2.0 my_robot_model
```

2. Внутри ROS-пакета создайте папки config, launch и urdf используя следующие команды:

```
cd my_robot_model
mkdir launch urdf config
```

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

.urdf. Сделать это можно как через графический интерфейс операционной системы, так и воспользовавшись командной строкой и следующими командами:

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

Обратите внимание! Новые файлы ros-пакета обязательно должны быть указаны в setup.py по аналогии с launch файлами, как это было показано ранее.

Важно! Убедитесь, что у вас в Visual Studio Code установлено расширение **Robotics Developer Environment**. Также вам необходимо установить расширение ROS Snippets(Liews Wuttipat), данные расширения позволяют облегчить работу с URDF.

После установки расширений запустить визуализатор URDF можно при помощи CTRL+Shift+P, затем нажав на поле ввода в верхней панели окна, введя "URDF" для поиска и выбрав из найденных вариантов "Preview URDF". Обратите внимание, что данный визуализатор отображает URDF файл, который вы открыли для редактирования. Соответственно, вам необходимо открыть файл с валидным (т.е. файл не должен содержать синтаксических ошибок и не быть пустым) URDF. Сейчас вы создали только пустой файл, следовательно, у вас ничего не отобразится. На следующем этапе в описании лабораторной работы вы найдете пример валидной разметки URDF.

Обратите внимание! Редактор исходного кода VS Code способен определять ваш проект как ROS (внизу окна слева должна отображаться версия ROS), это может быть необходимо для корректной работы расширений в VS Code (как правило, работает и без этого), например, если вы хотите использовать автодополнение при написании отдельных блоков разметки URDF. Чтобы проект распознавался как ROS, VS Code необходимо запускать из рабочего окружения, например:

```
cd <your_ros2_workspace>
code .
```

4. Создание модели

   В качестве тренировки предлагается создать модель мобильного робота с дифференциальным приводом, показанную на рисунке 5.1.

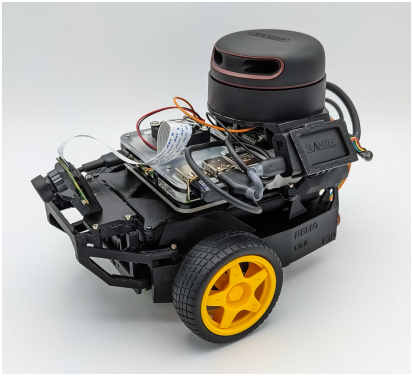


Рисунок 5.1. Мобильный робот с дифференциальным приводом

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 (но это не значит, что модели с замкнутой кинематикой не могут быть описаны с 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>
```

В приведенной разметке вы можете заметить, что весь код описания модели лежит внутри тега robot, в каждом файле urdf обязательно должен быть этот тег. Внутри тега robot нашего примера вы найдете 3 вложенных блока: два блока link (два звена, представляющих твердые тела) и один joint (шарнир, соединяющий два звена).

- Теперь внесите необходимые изменения в описание модели. У шарнира, который соединяет колесо с базой необходимо установить type="continuous", поскольку мы не хотим задавать какие-либо ограничения на допустимые углы поворота. Также у шарниров типа "continuous" необходимо задать ось вращения, для этого добавьте свойство axis (ось вращения в системе координат, связанной с шарниром).

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

- Необходимо скопировать и переименовать звено wheel_left в звено wheel_right с соответствующим шарниром для его соединения с базой робота.
- После этого необходимо настроить положение и ориентацию колес в соответствии с изображением робота (вспомните из материалов лекции или обратитесь к справочной литературе, если забыли как именно преобразования систем координат задаются в описании модели робота).

Обратите внимание: в ROS 2 часто возникает проблема, когда наличие синтаксических ошибок не позволяет отобразить модель, а система не сообщает об ошибках в самой XML-разметке. Чтобы решить эту проблему, необходимо

воспользоваться программой для проверки корректности URDF под названием check_urdf, как показано ниже:

```
check_urdf my_robot.urdf
```

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**.

   Установите пакет, который не является стандартным:

```
sudo apt install ros-jazzy-joint-state-publisher
```

   Далее необходимо создать launch файл, как это было показано в предыдущей практической работе. Добавьте запуск следующих дополнительных углов (пояснения к ним будут далее).

```
# получить путь к расположению пакета
pkg_lab =
↪  get_package_share_directory('my_robot_model')
xacro_file = os.path.join(pkg_lab, 'urdf',
↪  'my_first_model.urdf')

robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    output='both',
    parameters=[
        {'robot_description': Command(['xacro ',
        LaunchConfiguration('urdf_model')])},
    ]
)
joint_state_publisher = Node(
    package='joint_state_publisher',
    executable='joint_state_publisher',
    name='joint_state_publisher',
```

```
    output='both',
)

return LaunchDescription([
    DeclareLaunchArgument(
        'urdf_model',
        default_value=xacro_file,
        description='Full path to the Xacro file'
    ),
    robot_state_publisher,
    joint_state_publisher,
])
```

Дополнительно необходимо указать import os в начале launch файла, это стандартная библиотека python, которая используется для работы с файловой системой.

В вышеуказанном листинге представлен запуск robot_state_publisher и joint_state_publisher с одновременной загрузкой модели робота в сервер параметров ROS.

6. Теперь необходимо открыть модель робота в rviz

   Rviz можно открыть из терминала, воспользовавшись командой:

```
rviz2
```

   либо указать автоматический запуск в файле запуска.

   Чтобы rviz автоматически запускался, необходимо добавить в launch файл запуск еще одной ноды с указанием пути к файлу конфигурации:

```
rviz = Node(
       package='rviz2',
       executable='rviz2',
       arguments=['-d', os.path.join(pkg_lab,
       'config', 'rviz.rviz')],
)
```

Обратите внимание, файла конфигурации у вас нет, поэтому при запуске у вас будет использоваться конфигурация по умолчанию (в которой ничего не визуализируется), в дальнейшем в текущей практике вы создадите этот файл конфигурации.

Добавьте в файл запуска вышеуказанную разметку с запуском rviz и запустите launch файл.

```
ros2 launch my_robot_model robot.launch.py
```

В результате у вас ничего не запустится скорее всего. Причинами могут быть:

- забыли build или source
- забыли в setup.py указать launch файл (было в предыдущих практиках)
- забыли указать путь к urdf файлу в setup.py (любые файлы проекта должны быть там указаны)

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 "RobotModel". Далее в левой панели настроек RViz найти настройки этого плагина и напротив поля Description topic нажать на невидимое поле, там появится выпадающий список, из которого нужно выбрать топик, где хранится модель робота.

   После успешной настройки визуализации сохраните конфигурационный файл в свой ROS пакет, в подпапку config

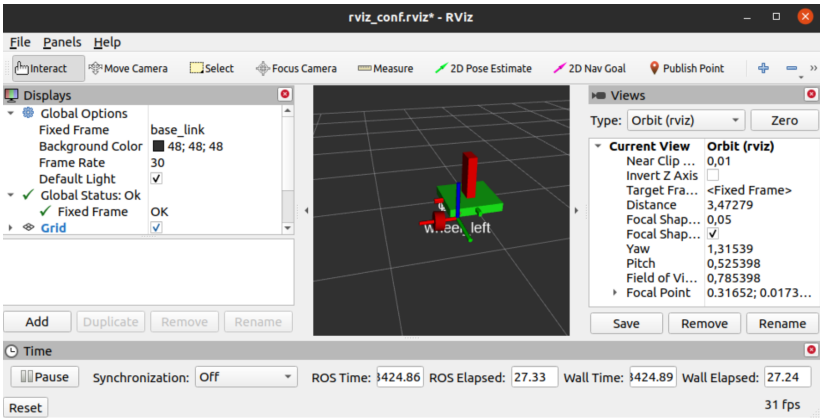


Рисунок 5.2. Окно Rviz.

с именем файла rviz.rviz. При следующем запуске launch файла эти настройки визуализации будут автоматически загружены.

8. Управление шарниром

Запустите графический интерфейс 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-jazzy-joint-state-publisher-gui
```

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

```
joint_state_publisher_gui = Node(
    package='joint_state_publisher_gui',
    executable='joint_state_publisher_gui',
    name='joint_state_publisher_gui',
    output='both',
```

```
)
```

Также необходимо закомментировать запуск 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, показанному на изображении выше.

5.2 Создание модели в XACRO

Чтобы ваша модель была более гибкой и универсальной, необходимо воспользоваться возможностями XACRO. Часто использование XACRO позволяет избавиться от дублирования XML-разметки, позволяя динамически собирать модель из разных частей. Например, при наличии симметрии в модели XACRO позволяет сократить объём разметки почти в 2 раза.

Обратите внимание: XACRO всегда преобразуется в URDF во время сборки ROS-пакета, поэтому бывает сложно проверить наличие синтаксических ошибок именно в XACRO. Чтобы решить эту проблему, мы можем перевести модель в URDF и воспользоваться программой check_urdf, как это показано ниже:

```
xacro my_robot.xacro | check_urdf
```

Начните с установки пакета командой:

```
sudo apt install ros-jazzy-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. ROS должен как-то различать чистый URDF и XACRO, для этого используется параметр xmlns:xacro="http://www.ros.org/wiki/xacro" внутри тега robot.

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro"
↪  name="my_robot">
```

2. Для работы с `xacro` необходимо также изменить launch-файл. В частности, требуется исправить переменную, задающую путь к xacro-модели, как показано ниже.

```
xacro_file = os.path.join(pkg_lab, 'urdf',
↪  'diff_robot_model.urdf.xacro')
```

3. Создайте также свойство в файле с разметкой модели робота:

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

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

   Пример использования:

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

> Задание на 5 минут: Создайте свойство с названием **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 вы хотите использовать в модели. Пример файла показан ниже.

```
<?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 (одно из преимуществ xacro) следующим образом:

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

Использовать доступное свойство цвета для визуального элемента модели робота, например, добавив его в визуальные свойства базы робота.

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

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 (этот XML-код также можно взять из ранее созданной URDF-модели):

```
<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 (параметры 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" />
```

5.3 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://urdf/models/base.stl"
↪  scale="1.0 1.0 1.0"/>
```

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

Задание для самостоятельного выполнения

1. Добавить двухзвенный манипулятор к подвижной платформе.
2. Добавить ограничения к шарнирам манипулятора, чтобы первое звено не имело возможности сталкиваться со вторым звеном.

Вопросы для самопроверки

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

6 Практическая работа №4.1 Создание имитационной модели в Gazebo с поддержкой ROS 2

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.

6.1 Создание симуляции робота средствами ROS

1. Установка зависимостей

```
sudo apt install ros-jazzy-ros-gz \
ros-jazzy-ros-gz-sim \
ros-jazzy-ros-gz-bridge -y
```

2. Создание пакета

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

```
ros2 pkg create --build-type ament_python --license
↪  Apache-2.0 my_robot_simulation
```

3. Создание файла запуска

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

```
import os

from ament_index_python.packages import
↪  get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.actions import IncludeLaunchDescription
from launch.conditions import IfCondition
from launch.launch_description_sources import
↪  PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration,
↪  PathJoinSubstitution, Command

from launch_ros.actions import Node

def generate_launch_description():
    pkg_ros_gz_sim =
    ↪  get_package_share_directory('ros_gz_sim')
    pkg_lab =
    ↪  get_package_share_directory('my_robot_simulation')
    pkg_model =
    ↪  get_package_share_directory('my_robot_model')

    xacro_file = os.path.join(pkg_model, 'urdf',
            'diff_robot_model.urdf.xacro')

    gz_sim = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_ros_gz_sim, 'launch',
            ↪  'gz_sim.launch.py')),
        launch_arguments={'gz_args':
        ↪  PathJoinSubstitution([
            pkg_lab,
            'worlds',
```

```
            'empty.world'
        ])}.items(),
    )

    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='both',
        parameters=[
            {'use_sim_time': True},
            {'robot_description': Command(['xacro ',
                LaunchConfiguration('urdf_model')])},
        ]
    )

    rviz = Node(
       package='rviz2',
       executable='rviz2',
       parameters=[{'use_sim_time': True}],
       arguments=['-d', os.path.join(pkg_model, 'config',
       ↪  'rviz.rviz')],
    )

    spawn = Node(
        package='ros_gz_sim',
        executable='create',
        arguments=[ '-name', 'diff_drive', '-topic',
                    'robot_description', '-x', '1', '-y',
                    ↪  '1'],
        output='screen'
    )

    bridge = Node(
        package='ros_gz_bridge',
        executable='parameter_bridge',
        parameters=[{
        'config_file': os.path.join(pkg_lab, 'config',
        ↪  'ros_gz_bridge.yaml'),
        'qos_overrides./tf_static.publisher.durability':
        ↪  'transient_local',
```

```
        }],
        output='screen'
    )

    return LaunchDescription([
        gz_sim,
        DeclareLaunchArgument(
            'urdf_model',
            default_value=xacro_file,
            description='Full path to the Xacro file'
        ),
        bridge,
        spawn,
        robot_state_publisher,
        rviz,
    ])
```

4. Проверка разметки модели робота и параметров collision и inertial

   Для работы симулятора обязательно должны быть теги collision 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>
```

5. Настройка параметров 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 характеристики зависят только от формы и плотности материала (однородная масса), поэтому вы можете написать макросы, которыми можно пользоваться для автоматического расчета инерциальных характеристик в зависимости от формы объекта с предположением, что масса по объему распределена равномерно.

   Файл с макросами можно 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>
```

Далее в самом файле wheel.xacro необходимо добавить прием тега inertial и его вставку в разметку. Найдите строчку <xacro:macro name="wheel" и добавьте еще один параметр *inertial, чтобы получилось:

```
<xacro:macro name="wheel" params="side x y z *inertial">
```

Далее добавьте внутри блока wheel следующую строчку в месте добавления inertial (вложенность должна быть такая же как у тегов visual и collision):

```
<xacro:insert_block name="inertial"/>
```

6. Настройка свойств для Gazebo

   Теперь необходимо указать важные свойства для работы симулятора. Создайте файл robot_diff.gazebo в папке urdf.

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

```
<?xml version="1.0"?>
<robot>
<gazebo>
<plugin filename="gz-sim-diff-drive-system"
    name="gz::sim::systems::DiffDrive">
```

```
<left_joint>wheel_left_joint</left_joint>
<right_joint>wheel_right_joint</right_joint>
<wheel_separation>0.65</wheel_separation>
<wheel_radius>0.1</wheel_radius>
<topic>model/diff_drive/cmd_vel</topic>
<tf_topic>model/diff_drive/pose</tf_topic>
<topic>model/diff_drive/odometry</topic>
<frame_id>diff_drive/odom</frame_id>
<child_frame_id>diff_drive</child_frame_id>
<odom_publish_frequency>50</odom_publish_frequency>
<max_linear_acceleration>1</max_linear_acceleration>
<min_linear_acceleration>-1</min_linear_acceleration>
<max_angular_acceleration>2</max_angular_acceleration>
<min_angular_acceleration>-2</min_angular_acceleration>
<max_linear_velocity>0.5</max_linear_velocity>
<min_linear_velocity>-0.5</min_linear_velocity>
<max_angular_velocity>1</max_angular_velocity>
<min_angular_velocity>-1</min_angular_velocity>
</plugin>

<plugin
filename="gz-sim-joint-state-publisher-system"
name="gz::sim::systems::JointStatePublisher">
</plugin>

<plugin
filename="gz-sim-pose-publisher-system"
name="gz::sim::systems::PosePublisher">
<publish_link_pose>true</publish_link_pose>
<use_pose_vector_msg>true</use_pose_vector_msg>
<static_publisher>true</static_publisher>
<static_update_frequency>1</static_update_frequency>
</plugin>

</gazebo>

</robot>
```

Выше мы добавили 3 разных плагина: gz-sim-joint-state-publisher-system (нужен для публикации состояния шарниров в симуляции), gz-sim-pose-publisher-system (нужен для публикации положения элементов модели робота), gz-sim-

diff-drive-system (нужен для управления дифференциальной платформой робота).

Убедитесь, что параметры плагина gz-sim-diff-drive-system были настроены корректно, их необходимо исправить в соответствии с названиями шарниров вашей модели робота (иначе симулятор не узнает, какими шарнирами управлять).

7. 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:

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

<sdf version="1.8">
  <world name="default">
    <plugin
      filename="gz-sim-physics-system"
      name="gz::sim::systems::Physics">
    </plugin>

    <plugin
      filename="gz-sim-scene-broadcaster-system"
      name="gz::sim::systems::SceneBroadcaster">
    </plugin>

    <plugin
      filename="gz-sim-user-commands-system"
      name="gz::sim::systems::UserCommands">
    </plugin>

    <plugin
      filename="gz-sim-sensors-system"
      name="gz::sim::systems::Sensors">
      <render_engine>ogre2</render_engine>
    </plugin>
```

```
<light name="sun" type="directional">
  <cast_shadows>true</cast_shadows>
  <pose>0 0 10 0 0 0</pose>
  <diffuse>0.8 0.8 0.8 1</diffuse>
  <specular>0.2 0.2 0.2 1</specular>
  <attenuation>
    <range>1000</range>
    <constant>0.9</constant>
    <linear>0.01</linear>
    <quadratic>0.001</quadratic>
  </attenuation>
  <direction>-0.5 0.1 -0.9</direction>
</light>

<model name="ground_plane">
  <static>true</static>
  <link name="link">
    <collision name="collision">
      <geometry>
        <plane>
          <normal>0 0 1</normal>
          <size>100 100</size>
        </plane>
      </geometry>
    </collision>
    <visual name="visual">
      <geometry>
        <plane>
          <normal>0 0 1</normal>
          <size>100 100</size>
        </plane>
      </geometry>
      <material>
    <ambient>0.8 0.8 0.8 1</ambient>
    <diffuse>0.8 0.8 0.8 1</diffuse>
    <specular>0.8 0.8 0.8 1</specular>
      </material>
    </visual>
  </link>
</model>
```

```
  </world>
</sdf>
```

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

8. Настройка ros_gz_bridge

   Gazebo и ROS 2 используют разные транспортные протоколы: Gazebo работает через `gz-transport`, а ROS 2 — через DDS. Пакет `ros_gz_bridge` выступает посредником, перенаправляя данные между этими двумя системами. Конфигурация моста задаётся в YAML-файле, где для каждого топика указывается: название топика в ROS, название топика в Gazebo, типы сообщений и направление передачи (`GZ_TO_ROS`, `ROS_TO_GZ` или `BIDIRECTIONAL`).

   Для просмотра списка топиков, доступных внутри Gazebo (они не видны через `ros2 topic list`), используется команда:

```
gz topic -l
```

   В папке `config` создайте файл `ros_gz_bridge.yaml`. Необходимо корректно задать содержимое конфигурационного файла. Данный файл определяет, какие топики в ROS будут связаны с топиками симулятора Gazebo с учётом типов передаваемых данных и направления передачи.

   Важно! Нельзя задавать в этом файле отображения (mapping) с типами данных, которые недоступны в текущем рабочем пространстве (это приведёт к нарушению работы механизма перенаправления).

   Далее приведено содержимое конфигурационного файла `ros_gz_bridge.yaml`. Проверьте, действительно ли необходимо каждое из указанных перенаправлений.

```
---
- ros_topic_name: "/diff_drive/cmd_vel" # /cmd_vel чтобы
↪  управлять через teleop
  gz_topic_name: "/model/diff_drive/cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "gz.msgs.Twist"
  direction: ROS_TO_GZ
- ros_topic_name: "/clock"
  gz_topic_name: "/clock"
  ros_type_name: "rosgraph_msgs/msg/Clock"
  gz_type_name: "gz.msgs.Clock"
  direction: GZ_TO_ROS
- ros_topic_name: "/diff_drive/odometry"
  gz_topic_name: "/model/diff_drive/odometry"
  ros_type_name: "nav_msgs/msg/Odometry"
  gz_type_name: "gz.msgs.Odometry"
  direction: GZ_TO_ROS
- ros_topic_name: "/joint_states"
  gz_topic_name:
  ↪  "/world/demo/model/diff_drive/joint_state"
  ros_type_name: "sensor_msgs/msg/JointState"
  gz_type_name: "gz.msgs.Model"
  direction: GZ_TO_ROS
- ros_topic_name: "/tf"
  gz_topic_name: "/model/diff_drive/pose"
  ros_type_name: "tf2_msgs/msg/TFMessage"
  gz_type_name: "gz.msgs.Pose_V"
  direction: GZ_TO_ROS
- ros_topic_name: "/tf_static"
  gz_topic_name: "/model/diff_drive/pose_static"
  ros_type_name: "tf2_msgs/msg/TFMessage"
  gz_type_name: "gz.msgs.Pose_V"
  direction: GZ_TO_ROS
```

9. Запуск симулятора Обратите внимание, что joint_state_publisher больше не нужен, его запуск мы исключаем из launch, и вместо него теперь положения шарниров будут приходить из симулятора. Теперь вы можете запустить симулятор и проверить корректность работы, для этого воспользуйтесь командой:

```
ros2 launch my_robot_simulation sim.launch.py
```

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 /diff_drive/cmd_vel, в который вы и можете отправлять команды для управления.

10. 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>
```

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

```
<limit effort="100" velocity="100"/>
<joint_properties damping="0.02" friction="0.02"/>
```

12. 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

13. Управление с учетом кинематической модели робота Установите пакет `teleop_twist_keyboard`. После этого появится возможность управлять роботом в симуляторе с помощью клавиатуры, задавая направление движения посредством нажатия соответствующих клавиш. Данный пакет обрабатывает нажатия клавиш и отправляет соответствующие управляющие команды средствами ROS.

    Команда для установки:

```
sudo apt-get install ros-jazzy-teleop-twist-keyboard
```

    Команда для запуска:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Данный пакет по умолчанию публикует управляющие команды в топик `/cmd_vel`, а модель робота в симуляторе ожидает управляющие команды через топик `/diff_drive/cmd_vel`. Поэтому необходимо устранить данное несоответствие. Либо добавить переадресацию между топиками (remap `/cmd_vel`), либо в конфигурационном файле `ros_gz_bridge` изменить название топика, в который поступают управляющие команды, с `/diff_drive/cmd_vel` на `/cmd_vel`.

Альтернативным способом управления является отправка соответствующих сообщений в топик /diff_drive/cmd_vel стандартными средствами ROS через собственную ноду или из терминала, как показано далее.

```
ros2 topic pub /diff_drive/cmd_vel ***
```

Бонус. Чтобы не редактировать setup.py при каждом добавлении новых файлов в пакет, можно воспользоваться библиотекой glob. В начале файле setup.py добавьте строчку

```
from glob import glob
```

а вместо подключения конкретных файлов, например, из папки config укажите:

```
('share/' + package_name + '/config',
↪  glob('config/*')),
```

таким образом, автоматически будут прописаны адреса ко всем файлам, которые вы уже создали или будете в дальнейшем создавать в этих папках.

Задание для самостоятельного выполнения

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

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

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)?

7 Практическая работа №4.2. Создание регулятора для роботов произвольной кинематики на ros2_control

Цель работы

- Установка дополнительных пакетов и добавление зависимостей ros2_control.
- Создание регулятора для модели робота с произвольной кинематикой.

Создание пакета Создайте новый пакет с названием `my_robot_simulation_control`:

```
ros2 pkg create --build-type ament_python --license Apache-2.0
↪  my_robot_simulation_control
```

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

```
mkdir launch config
```

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 Для работы ros_control нам необходимо установить 2 ROS-пакета: первый с основным функционалом и второй с регуляторами.

```
sudo apt install ros-jazzy-ros2-control
↪  ros-jazzy-ros2-controllers
```

Также необходимо установить плагины для Gazebo.

```
sudo apt install ros-jazzy-gz-ros2-control
↪  ros-jazzy-gz-ros2-control-demos
```

7.1 Добавление необходимых компонентов

Создание конфигурационного файла `config/diff_control.yaml`

Нам необходимо задать параметры для работы ros_control. Здесь мы указываем параметры controller_manager и самого регулятора (в нашем случае это регулятор по скорости) с указанием названия шарниров, которыми нам предстоит управлять.

```
controller_manager:
  ros__parameters:
    update_rate: 60  # Hz
    use_sim_time: true

    velocity_controller:
      type: velocity_controllers/JointGroupVelocityController

velocity_controller:
  ros__parameters:
    joints:
      - joint1
      - joint2
```

Важно! Убедитесь, что параметры плагина были настроены корректно, в соответствии с вашим urdf.

Добавление плагина `gz_ros2_control` **[18]** Ранее мы использовали специализированный плагин gazebo::systems::DiffDrive, который может управлять только роботами с определенной кинематикой(2 шарнира, дифференциальная платформа). Однако такие ограничения нам не нужны, нам необходимо иметь возможность управлять роботами с любой кинематикой, что позволяет ros2_control. Удалите из файла с расширением .gazebo блок с плагином gazebo::systems::DiffDrive. Добавьте следующий программный код в файл с расширением `.gazebo` внутри тега gazebo. Здесь мы подключаем плагин ros2_control для gazebo и указываем путь к конфигурационному файлу (где указаны контроллеры).

```
<plugin filename="gz_ros2_control-system"
name="gz_ros2_control::GazeboSimROS2ControlPlugin">
<parameters>
$(find my_robot_simulation_control)/config/diff_control.yaml
</parameters>
</plugin>
```

Добавьте `Transmission` в `diff_drive_robot.xacro`

Transmission описывает связь между приводом (joint, который двигает контроллер) и его физическим исполнительным механизмом (hardware interface). Transmission необходимо указать в URDF-файле робота (или .xacro) рядом с определениями joint'ов (Обычно transmission идёт сразу после определения всех joint'ов, чтобы было удобно редактировать параметры шарнира и трансмиссии согласованно). Убедитесь, что названия шарниров указаны правильно!

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

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

```
        <hardwareInterface>
    hardware_interface/EffortJointInterface
    </hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
```

```
<ros2_control name="DiffBotSystem" type="system">
<hardware>
    <plugin>
                gz_ros2_control/GazeboSimSystem
    </plugin>
</hardware>

<joint name="wheel_left_joint">
    <command_interface name="velocity"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
</joint>

<joint name="wheel_right_joint">
    <command_interface name="velocity"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
</joint>
```

Настройка файла `launch`

Используйте файл запуска `.launch.py` и добавьте в него запуск нового узла, отвечающего за запуск регуляторов.

```
 spawn_controller = Node(
        package="controller_manager",
        executable="spawner",
        name=f"spawn_controller_wheel_controller",
        arguments=["velocity_controller"],
        output="screen"
)
```

7.2 Запуск ros2_control

```
ros2 launch my_robot_simulation_control
↪  robot_sim_control.launch.py
```

7.3 Отправка управляющих команд

Через консольное окно (в терминале)

```
ros2 topic pub -1 /velocity_controller/commands
↪  std_msgs/msg/Float64MultiArray 'data: [1.0, 1.0]'
```

Через `rqt` также можно отправлять управляющие команды

```
ros2 run rqt_gui rqt_gui
```

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

Задание для самостоятельного выполнения

1. Разработать ноду, которая парсит команды из топика /cmd_vel в топик /velocity_controller/commands (учитывая кинематику мобильного робота)

2. Настроить в `rqt` вывод графика скорости каждого из колес робота.

Вопросы для самопроверки

- Каким образом можно управлять роботом с дифференциальным приводом?
- Что такое ros2_control?
- Что такое controller_manager?
- Что такое gz_ros2_control?

- Как подавать управляющий сигнал в ros2_control?
- Какие дополнительные параметры необходимо добавить в URDF для управления через ros2_control?

8 Практическая работа №5. Настройка датчиков и обработка данных

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

8.1 Настройка датчика типа Camera

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

```
ros2 pkg create my_robot_sim_sensors --build-type ament_python
```

Далее необходимо создать файл запуска в папке launch по аналогии с предыдущими занятиями. Рекомедуется максимально использовать наработки из практики 4.1 и 4.2 (модель робота, конфигурационные файлы, файлы .world для симулятора).

При копировании файлов или фрагментов исходного кода (или разметки) проверяйте корректность указанных путей (включая название файлов). Не забывайте указывать все нестандартные файлы в setup.py (рекомендуется использовать glob).

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_link">
       <visual name="camera_link_visual">
         <origin xyz="0 0 0" rpy=" 0 0 0"/>
           <geometry>
   ```

```
        <box size="0.04 0.04 0.04"/>
        </geometry>
    </visual>

    <collision name="camera_link_colision">
      <origin xyz="0 0 0" rpy=" 0 0 0"/>
        <geometry>
            <box size="0.04 0.04 0.04"/>
        </geometry>
    </collision>

    <inertial name="camera_link_inertial">
      <origin xyz="0 0 0" rpy=" 0 0 0"/>
        <mass value="0.1"/>
        <inertia ixx="1e-3" ixy="0" ixz="0"
                 iyy="1e-3" iyz="0"
                 izz="1e-3"/>
    </inertial>
</link>
<joint name="camera_joint" type="continuous">
    <origin xyz="-0.25 0 0.35" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="camera_link"/>
    <axis xyz="0 0 1"/>
</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_link">
<material>Gazebo/Black</material>
<sensor name="camera" type="camera">
    <pose> 0 0 0 0 0 0 </pose>
    <visualize>true</visualize>
    <update_rate>10</update_rate>
    <camera>
        <camera_info_topic>
            camera/camera_info
```

```
        </camera_info_topic>
        <horizontal_fov>1.089</horizontal_fov>
        <image>
            <format>R8G8B8</format>
            <width>640</width>
            <height>480</height>
        </image>
        <clip>
            <near>0.05</near>
            <far>8.0</far>
        </clip>
    </camera>
    <topic>camera/image_raw</topic>
    <gz_frame_id>diff_drive/camera_link</gz_frame_id>
</sensor>
</gazebo>
```

3. проверка работоспособности камеры после настройки

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

```
import os
from ament_index_python.packages import
↪  get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
↪  PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration,
↪  PathJoinSubstitution, Command
from launch_ros.actions import Node

def generate_launch_description():
    pkg_ros_gz_sim =
    ↪  get_package_share_directory('ros_gz_sim')
    pkg_lab =
    ↪  get_package_share_directory('my_robot_sim_sensors')
```

```
    xacro_file = os.path.join(pkg_lab, 'urdf',
    ↪  'robot_model.xacro')

    gz_sim = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_ros_gz_sim, 'launch',
            ↪  'gz_sim.launch.py')),
        launch_arguments={'gz_args':
        ↪  PathJoinSubstitution([
            pkg_lab,
            'worlds',
            'empty.world'
        ])}.items(),
    )
    robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='both',
        parameters=[
            {'use_sim_time': True},
            {'robot_description': Command(['xacro ',
                LaunchConfiguration('urdf_model')])},
        ]
    )
    rviz = Node(
        package='rviz2',
        executable='rviz2',
        parameters=[{'use_sim_time': True}],
        arguments=['-d', os.path.join(pkg_lab, 'config',
        ↪  'rviz.rviz')],
    )
    spawn = Node(
        package='ros_gz_sim',
        executable='create',
        arguments=[ '-name', 'diff_drive', '-topic',
                'robot_description', '-x', '1', '-y', '1',
                ↪  '-z', '0.2'],
        output='screen'
    )
    bridge = Node(
```

```
        package='ros_gz_bridge',
        executable='parameter_bridge',
        parameters=[{
        'config_file': os.path.join(pkg_lab, 'config',
        ↪  'ros_gz_bridge.yaml'),
        'qos_overrides./tf_static.publisher.durability':
        ↪  'transient_local',
        }],
        output='screen'
    )
    spawn_controller = Node(
        package="controller_manager",
        executable="spawner",
        name=f"spawn_controller_left_wheel_controller",
        arguments=["velocity_controller"],
        output="screen"
    )
    return LaunchDescription([
        gz_sim,
        DeclareLaunchArgument(
            'urdf_model',
            default_value=xacro_file,
            description='Full path to the Xacro file'
        ),
        bridge,
        spawn,
        robot_state_publisher,
        rviz,
        spawn_controller,
    ])
```

Мы только что настроили камеру внутри Gazebo, теперь необходимо настроить перенаправление данных из gazebo в ROS, воспользовавшись gz_ros_bridge.yaml, добавьте следующие строки в конец конфигурационного файла.

```
# gz topic published by Sensors plugin (Camera)
- topic_name: "/camera/image_raw"
  ros_type_name: "sensor_msgs/msg/Image"
  gz_type_name: "gz.msgs.Image"
  lazy: true
```

```
  direction: GZ_TO_ROS

- topic_name: "/camera/camera_info"
  ros_type_name: "sensor_msgs/msg/CameraInfo"
  gz_type_name: "gz.msgs.CameraInfo"
  lazy: true
  direction: GZ_TO_ROS
```

Убедитесь, что все параметры соответствуют имеющимся в вашей модели названиям и запустите launch файл.

```
ros2 launch my_robot_sim_sensors robot_sim_camera.launch
```

Откройте Rviz и выполните последовательность команд: Add → Image, затем выберите топик, куда публикуется изображение (image_raw). Название топика вы указывали при добавлении датчика в файле .gazebo.

8.2 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 сам датчик лидара с указанием необходимых параметров.

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.

   Добавьте разметку для датчика, обеспечивающего извлечение информации из Gazebo для вашего лидара, путем добавления в вашу URDF модель следующей разметки сразу после физической модели лидара (его link и joint):

```
<gazebo reference="lidar">
    <material>Gazebo/Black</material>
    <sensor type="ray" name="lidar-right-eye">
    <pose>0 0 0 0 0 0</pose>
    <topic>scan</topic>
    <gz_frame_id>diff_drive/lidar_link</gz_frame_id>
    <update_rate>10</update_rate>
    <lidar>
      <scan>
        <horizontal>
          <samples>640</samples>
          <resolution>1</resolution>
          <min_angle>-1.396263</min_angle>
          <max_angle>1.396263</max_angle>
        </horizontal>
        <vertical>
          <samples>1</samples>
          <resolution>1</resolution>
          <min_angle>0.0</min_angle>
          <max_angle>0.0</max_angle>
        </vertical>
      </scan>
      <range>
        <min>0.08</min>
        <max>10.0</max>
        <resolution>0.01</resolution>
      </range>
    </lidar>
    <visualize>true</visualize>
```

```
    </sensor>
</gazebo>
```

Теперь осталось только указать в gz_ros_bridge.yaml перенаправление потока данных из Gazebo в ROS.

```
- ros_topic_name: "/diff_drive/scan"
  gz_topic_name: "scan"
  ros_type_name: "sensor_msgs/msg/LaserScan"
  gz_type_name: "gz.msgs.LaserScan"
  direction: GZ_TO_ROS
```

3. Проверка работоспособности лидара

   Чтобы проверить корректность работы, необходимо запустить Gazebo с тем же launch файлом, что и раньше.

```
    ros2 launch my_robot_sim_sensors
    ↪  robot_sim_camera.launch.py
```

   В программе Rviz выполните последовательность команд: "Add", "Laser scan", затем выберите топик, куда публикуется изображение (у вас он был указан при перенаправлении данных в gz_ros_bridge.yaml).

   Обратите внимание, что лидар не будет возвращать какие-либо измерения, если в зоне его видимости нет никаких препятствий (пустая сцена в симуляции). Сообщения от лидара будут приходить с нулевыми значениями, и соответственно в RViz ничего не будет отображаться.

8.3 Обработка данных с датчиков в управляющей программе

Добавление датчиков в имитационную модель и проверка их работоспособности является лишь началом, в дальнейшем мы хотим эти данные использовать для формирования управляющих программ нашего робота.

Для обработки данных с датчиков нам будут необходимы сторонние библиотеки (не входящие в стандартную установку python).

В Ubuntu 24.04 нельзя ставить пакеты Python напрямую в системный (/usr/lib/python3...). Все новые библиотеки и проекты нужно устанавливать в своё отдельное виртуальное окружение — это безопасно, удобно и не сломает систему.

1. Создание виртуального окружения

   Переходим в cd /ros2_ws/ и создадим виртуальное окружение с названием rosvenv

   ```
   python3 -m venv rosvenv
   ```

   Чтобы в это окружение установить новые пакеты, необходимо его активировать через команду

   ```
   source rosvenv/bin/activate
   ```

   Далее устанавливаем пакеты, как обычно, через pip. Теперь нам необходимо, чтобы наши ROS-пакеты (точнее, исполняемые файлы) запускались из этого окружения. Давайте настроим launch файл, чтобы он запускал узел обработки сенсорной информации внутри нашего окружения, для этого добавим после import-ов строку с адресом нашего окружения:

   ```
   venv_python =
   ↪  os.path.expanduser("~/ros2_ws/rosvenv/bin/python3")
   ```

   И далее внутри ноды передадим его (по аналогии с параметрами), указываем его в prefix.

   ```
       prefix=[venv_python, " "],
   ```

2. обработка изображения с камеры в программном коде

   Нам нужен ROS-пакет, который позволяет преобразовывать файлы из ROS формата в удобный для цифровой обработки. Установите пакет:

```
sudo apt install ros-jazzy-cv-bridge
```

Активируйте свое окружение и установите пакет:

```
pip install opencv-python
pip uninstall numpy
pip install "numpy<2"
```

Чтобы получить данные с камеры в управляющей программе, вам понадобится создать файл исходного кода и прописать в нем импорт библиотеки cv_bridge [19], который позволит вам преобразовать формат данных в более удобный для дальнейшей обработки при помощи библиотеки opencv [20].

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

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image

import cv2
from cv_bridge import CvBridge

class SensorDataHandler(Node):
    def __init__(self):
        super().__init__('camera_data_processing')
        sub_topic_name =
        ↪  "/diff_drive_robot/camera1/image_raw"
        self.camera_subscriber =
        ↪  self.create_subscription(Image,
        ↪  sub_topic_name, self.camera_cb, 10)
        self.bridge = CvBridge()
        self.current_image = None
        self.get_logger().info(f"Subscribed:
        ↪  {sub_topic_name}")

    def camera_cb(self, msg):
```

```
        # Конвертация ROS2 Image в формат OpenCV
        frame = self.bridge.imgmsg_to_cv2(msg,
        ↪  desired_encoding='bgr8')

        print(f'Image size {frame.shape}')
        # Тут можно реализовать любой алгоритм обработки
        ↪  изображения

        cv2.imshow("output", frame) # чтобы посмотреть -
        ↪  выводим в окне программы
        cv2.waitKey(1)


def main(args=None):
    rclpy.init(args=args)
    node = SensorDataHandler()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    node.destroy_node()
    cv2.destroyAllWindows()
    rclpy.shutdown()

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

Теперь необходимо вспомнить, как мы делали python файлы исполняемыми, и запустить узел (ноду) в файле запуска (launch файле). Вы также можете прописать дополнительный программный код обработки изображения в этом файле.

3. Обработка данных лидара в программном коде.

   Чтобы получить данные с лидара в управляющей программе, вам понадобится создать файл исходного кода и подписаться на соответствующий топик. Далее приведен пример, где данные из лидара считываются и определяется минимальное измеренное расстояние (расстояние до ближайшего препятствия). Далее приведен lidar_data_proc.py:

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
import numpy as np

class LidarHandler(Node):

    def __init__(self):
        super().__init__('lidar_min_distance_node')
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.lidar_callback,
            10
        )
        self.get_logger().info("initialized. Subscribed
        ↪  /scan")

    def lidar_callback(self, msg: LaserScan):
        # Convert list → numpy array
        ranges = np.array(msg.ranges)

        # Remove invalid values (inf, nan, zeros)
        valid = ranges[np.isfinite(ranges)]
        # ignore tiny noise values
        valid = valid[valid > 0.01]

        if len(valid) == 0:
            self.get_logger().info("No valid LiDAR
            ↪  points")
            return

        min_dist = float(np.min(valid))

        self.get_logger().info(f"Min distance:
        ↪  {min_dist:.2f} m")


def main(args=None):
    rclpy.init(args=args)
    node = LidarHandler()
```

```
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    node.destroy_node()
    rclpy.shutdown()

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

Сделайте данный файл также исполняемым и пропишите его запуск в файле launch.

Задание для самостоятельного выполнения

1. Создайте сцену с объектами окружения (столб, конус или другой объект из Gazebo), сохранить в файл с новым названием и настроить его загрузку в файле запуска (вместо empty.world);
2. Напишите скрипт, который будет отправлять сообщения в /cmd_vel (любые сообщения);
3. При выполнении "spawn" (добавление робота в симуляцию) укажите еще и координату z;
4. Добавьте в симуляцию несколько одинаковых роботов, но в разных координатах сцены.

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?

9 Практическая работа №6. Интеграция аппаратной части в ROS2 через ros2_control

Цель работы

- Освоить создание пользовательского аппаратного интерфейса в ROS2, настройку контроллеров и организацию обмена данными с микроконтроллером.

9.1 Теоретический минимум

Некоторые модели роботов уже имеют возможность управления через ROS 2 [21], что значительно упрощает процесс управления и ускоряет внедрение в комплексные решения. Однако ROS 2 Control предоставляет незаивисимую от производителей роботов архитектуру для управления, которую можно внедрить в любое робототехническое решение, изначально не имевшее поддержку ROS 2. Архитектура ROS 2 Control включает:

- Hardware Interface — взаимодействие с оборудованием
- Controller Manager — управление контроллерами
- Controllers — алгоритмы управления

Контур управления включает три этапа:

1. Чтение состояния (read)
2. Обновление контроллеров (update)
3. Отправка команд (write)

Также используются:

- state_interface — получение данных
- command_interface — отправка команд

9.2 Создание Hardware Interface

Создайте пакет:

```
ros2 pkg create --build-type ament_cmake my_robot_hardware
```

Реализуйте класс, наследующийся от:

```
hardware_interface::SystemInterface
```

Реализуйте методы:

- read()
- write()
- export_state_interfaces()
- export_command_interfaces()

9.2.1 Создание plugin

Создайте файл **plugin_description.xml** и опишите плагин. Добавьте зависимости:

- hardware_interface
- rclcpp
- pluginlib

9.2.2 Настройка URDF

Добавьте блок ros2_control:

```
<ros2_control name="my_robot" type="system">
<hardware>
<plugin>my_robot_hardware/MyHardware</plugin>
</hardware>

<joint name="wheel_left">
<command_interface name="velocity"/>
<state_interface name="velocity"/>
```

```
</joint>

<joint name="wheel_right">
<command_interface name="velocity"/>
<state_interface name="velocity"/>
</joint>
</ros2_control>
```

Важно: URDF используется не только как модель робота, но и как система координат.

9.2.3 Настройка контроллеров

Создайте YAML файл:

```
controller_manager:
ros__parameters:
update_rate: 30

joint_state_broadcaster:
type: joint_state_broadcaster/JointStateBroadcaster

base_controller:
type: diff_drive_controller/DiffDriveController
```

9.3 Обмен данными с Arduino

Реализуйте протокол обмена:
Отправка:

```
SET v_left v_right
```

Получение:

```
ENC ticks_left ticks_right
```

9.4 Реализация control loop

В методах:

- read() — получение данных энкодеров
- write() — отправка скоростей

9.5 Запуск системы

Создайте launch-файл с параметрами:

```
use_sim_time: False
```

9.6 Проверка работоспособности

Используйте команды:

```
ros2 control list_hardware_interfaces
ros2 control list_controllers
```

9.7 Ожидаемый результат

- Робот принимает команды движения
- Контроллер преобразует команды
- Hardware Interface отправляет их на микроконтроллер
- Получается обратная связь через энкодеры

9.8 Задание для самостоятельного выполнения

1. Реализовать собственный контроллер для другой кинематики
2. Добавить QoS настройки

9.9 Вопросы для самопроверки

- Какие основные компоненты входят в архитектуру `ros2_control` и какую роль выполняет каждый из них?
- В чем различие между `state_interface` и `command_interface`? Приведите примеры их использования.

- Опишите цикл управления (control loop) в `ros2_control`. Какие этапы он включает?
- Как осуществляется обмен данными между ROS2 и микроконтроллером (например, Arduino) в рамках данной работы?

10 Лабораторная работа №1. Создание пакета и регулятора в ROS 2

Цель работы

- Создание ROS 2 пакета и файла запуска
- Разработка простого инкапсулированного регулятора

Задание к лабораторной работе № 1

В рамках лабораторной работы необходимо написать launch-файл, создающий два пространства имен (namespace) и запускающий в каждом пространстве имен имитационную модель черепахи и управляющий файл для каждой черепахи.

В управляющем файле должен быть реализован регулятор, выполняющий 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 черепахи и повторять движение за первой черепахой (не имея явного доступа к координатам целевых точек первой черепахи).

Обратите внимание. Непосредственно в самом программном коде можно получить текущее пространство имен, используя стандарный метод "rclpy.get_namespace()".

1. Создайте в локальной копии своего репозитория папку `lab1`, в нее необходимо будет скопировать результаты вашей работы (ROS2 пакеты) из папки `<your_ros2_ws>/src/`.

2. Создайте новый пакет, использующий ament_python с названием mybestcontroller

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

   (a) Создайте новый файл запуска с названием `lab1.launch.py` и определите два пространства имен с названиями `ns1_ISUID` и `ns2_ISUID`, где `ISUID` — это ваш ITMO ID.

   (b) Внутри каждого пространства имен необходимо указать запуск `turtlesim`.

   (c) В папке `mybestcontroller` создайте файл `lab1_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 (относительные и абсолютные), воспользуйтесь командой ros2 topic list при необходимости.

   (d) Управляющая программа для второй черепашки также должна быть прописана в файле `lab1_controller.py`. При этом, управляющая программа второй черепахи должна подписываться на топик `.../pose` первой черепашки и повторять его действия (запрещено использовать `remap` или подписываться на /cmd_vel первой черепахи).

   (e) 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.

4. **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,

5. Готовый пакет должен запускаться командой:

```
ros2 launch mybestcontroller lab1.launch.py
```

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

Таблица 10.1. Номера вариантов соответствуют последней цифре ID

Вариант	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)

Решение отправляется согласно инструкции курса по ROS через систему контроля версий git (gitlab).

11 Лабораторная работа №2. Создание пользовательских интерфейсов (service и action)

Цель работы

- Ознакомиться с механизмами сервисов и действий (services и actions) в ROS 2.
- Научиться создавать пользовательские интерфейсы: `.srv` и `.action` файлы.
- Научиться разделять функционал на серверную и клиентскую часть для service и action.

Задание к лабораторной работе №2

Часть 1. Сервис (Service)

ISUID - это ваш табельный номер (ITMO ID), далее в задании необходимо его заменить на 6-значное число.

1. Создать собственный тип сервиса (`AISUID.srv`) в пакете `lab2_interfaces`, позволяющий передавать текущие координаты `Xcurrent` и `Ycurrent` в качестве запроса, а в качестве ответа — расстояние до цели `Distance` и угловую ошибку `Angle`.
2. Реализовать в пакете `lab2_controller` сервер сервиса, принимающий сообщение заданного выше типа и возвращающий угол и расстояние, которые необходимо пройти. Положение целевой точки известно только серверу.
3. Реализовать в пакете `lab2_controller` клиент сервиса, отправляющий запросы с текущим положением и получающий отклонения по расстоянию и углу. Управление черепахой должен выполнять только клиент, а сервер фактически рассчитывает, на какое расстояние и под каким углом необходимо переместиться. Координаты целевых точек приведены в ЛР №1 в соответствии с вашим вариантом.

4. Проверить работу узлов с помощью команды `ros2 service call` в терминале и собственного клиента.

5. Создать launch-файл с именем `service_server.launch.py` и launch-файл с именем `service_client.launch.py`, запускающие соответствующие исполняемые файлы. В файле `service_server.launch.py` дополнительно укажите запуск имитационной модели черепахи.

Обратите внимание. Управление черепахой необходимо реализовать с обратной связью по положению. При этом клиент не должен иметь доступа к целевым точкам. Переключение на следующую целевую точку выполняется на стороне сервера, если расстояние от черепахи до цели менее 0.1 метра. Также учтите, что в момент отправки запроса клиентом, черепаха должна стоять неподвижно, поскольку сервер выдаст ошибку отклонения от цели именно относительно этой точки.

Рекомендации.

Сервер сервиса может дополнительно вернуть False в случае, если не была достигнута цель в момент получения запроса. А если в момент получения запроса черепаха в точке цели (допустимое отклонение 0.2 метра) - возвращать True. Клиент, в свою очередь, отправляет запросы, пока не получит True, а когда получит True, переключается на следующую цель.

Допускается выносить фактический контроллер черепахи в отдельную ноду. Также разрешается ставить небольшую задержку времени в обработчике Сервера и вывод отладочной информации в терминал на стороне сервера. Не забывайте, что сервис - это блокирующий тип взаимодействия (callback-и внутри ноды сервиса не срабатывают, пока сервис не вернет какой-либо ответ).

Часть 2. Действие (Action)

Обратите внимание. Вторая часть данной лабораторной работы является не обязательной, так как относится к продвинутому уровню и может вызывать сложности для начинающих изучение ROS 2, однако её тоже можно выполнить, если вы уже изучили Executors и мультипоточные вычисления.

1. Создать собственный тип действия (`AISUID.action`) в пакете lab2_interfaces, в котором в качестве цели задаются координаты целевой точки для черепахи.

2. Реализовать в пакете lab2_controller сервер действия (Action), который будет управлять черепахой, перемещая её к заданной позиции, публикуя промежуточные значения (ошибку по расстоянию) в качестве *feedback* и в качестве результата возвращая `True`, когда цель будет достигнута.

3. Реализовать в пакете lab2_controller клиента действия (Action), отправляющего цель и выводящего промежуточные значения (полученные от сервера) на экран, при достижении цели выводить сообщение «Миссия выполнена, но это не точно».

4. Проверить работу узлов с помощью `ros2 action send_goal` (через командную строку) и собственного клиента.

5. Создать launch-файл с именем action_server.launch.py и launch-файл с именем action_client.launch.py, запускающие соответствующие исполняемые файлы.

Отправка решения

Решение оформляется в виде двух пакетов ROS 2 и загружается в систему контроля версий (git) согласно инструкции курса в папку lab2: первый ROS-пакет с файлами интерфейсов (на базе ament_cmake) и второй с программным кодом (на основе ament_python), оба пакета должны лежать в папке lab2 (все лишнее из lab2 удалите).

12 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.

Описание лабораторной работы

1. Создайте ROS-пакет с названием my_best_model в рабочем пространстве, то есть в папке `<your_ros2_ws>/src/`.
2. **Создание модели робота:**
   (a) Создать папку `launch` и `urdf` в ROS-пакете.
   (b) В папке `urdf` создать файл `robot_model.xacro` и внутри составить XML-описание мобильного робота с закрепленным на нем манипулятором. Параметры модели (габариты базы, количество и типы звеньев) соответствуют варианту в таблице 12.1.
   (c) Создать файл запуска `rviz.launch.py` и прописать загрузку модели робота в сервер параметров.
   (d) В файле `rviz.launch.py` также указать запуск необходимых пакетов для корректной визуализации модели робота: `robot_state_publisher` и `joint_state_publisher_gui`.
   (e) Добавить запуск `rviz` с автоматической подгрузкой конфигурационного файла rviz, в которой уже настроена визуализация модели робота и визуализация TF.
3. Готовый пакет должен запускаться с помощью команды:

```
ros2 launch my_best_model rviz.launch.py
```

4. Скопируйте свой ROS-пакет my_best_model в папку `lab3` (если там уже что-то есть, удалите все из этой папки) в локальной копии своего GITLAB репозитория.

Варианты заданий. Номера вариантов для студентов соответствуют последней цифре ISU ID, значения параметров указаны в Таблице 12.1.

Таблица 12.1. 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

Обратите внимание на рашифровку обозначений.
Типы шарниров: R - вращательный шарнир, P - призматический шарнир, C - вращательный шарнир без ограничений на угол поворота (постоянного вращения). Если у вас в варианте задании указано "PRR CC" это значит, что у манипулятора первый шарнир - типа P второй и третий -типа R, CC - два вращательных шарнира постоянного вращения (на колеса), если CCCC - четыре колеса с шарнирами постоянного вращения (continuous).

Отправка решения

Решение оформляется в виде одного пакета ROS 2 и загружается в систему контроля версий (git) согласно инструкции

курса в папку lab3. Решением является ROS пакет с файлами запуска launch, файлы модели робота, ROS-пакет должен лежать в папке lab3 (все лишнее из lab3 удалите).

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

Цель работы

- Создание имитационной модели робота в Gazebo.
- Разработка управляющей программы для шарниров (без учета кинематики) с помощью ros2_control.
- Реализация собственного diff_drive управления

Создание нового пакета в рабочем пространстве

Необходимо перейти в рабочую папку окружения и создать пакет с названием **my_best_robot_simulation_control**.

Затем необходимо выполнить сборку созданного ROS-пакета.

Внутри ROS пакета создайте подпапки config, launch и urdf. Затем необходимо создать файл запуска с названием simulation_control.launch.py.

Подготовка модели робота

1. Скопируйте в новый пакет файлы модели робота с дифференциальным приводом, которые были созданы на практических заданиях 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. Управление шарнирами реализовать с помощью ros2_control. Также необходимо реализовать узел, обеспечивающий управление мобильным роботом через топик /cmd_vel с учетом его кинематики (diff_drive).

Написание управляющей программы

Требуется написать ноду, которая будет слушать команды через топик /cmd_vel, и будет отправлять необходимые скорости вращения колес в топик /velocity_controller/commands. Расчет необходимых скоростей вращения колес нужно производить с учетом кинематики мобильной платформы (diff_drive) и параметров системы (см. Таблицу 13.1)

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_best_robot_simulation_control** при выполнении файла запуска с названием **simulation_control.launch.py**.

Таблица 13.1. Параметры системы

Номер варианта	Радиус колеса в м	Габаритные размеры базы мобильной платформы в мм, Д Ш В
1	0.05 м	$200 \times 140 \times 90$
2	0.1 м	$200 \times 140 \times 120$
3	0.15 м	$260 \times 140 \times 100$
4	0.12 м	$260 \times 160 \times 100$
5	0.09 м	$190 \times 160 \times 130$
6	0.05 м	$200 \times 160 \times 90$
7	0.08 м	$200 \times 160 \times 90$
8	0.18 м	$230 \times 140 \times 90$
9	0.12 м	$220 \times 160 \times 90$
0	0.13 м	$200 \times 180 \times 120$

Все параметры, значения которых явно не указаны в лабораторной работе и таблице 13.1, выбираются произвольно.

14 Лабораторная работа №5. Создание имитационной модели робота, оснащенной датчиками

Цель работы

- Создание имитационной модели робота в Gazebo.
- Добавление датчиков на модель робота.
- Разработка управляющей программы с учетом данных с датчиков.

Создание нового пакета в рабочем пространстве

Необходимо перейти в рабочую папку окружения и создать пакет с названием **my_best_robot_simulation_setup**.

Внутри ROS пакета создайте подпапки config (конфигурационные файлы), launch (файлы запуска) и urdf (модель робота). Затем, необходимо создать файл запуска с названием simulation_final.launch.py.

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.
2. Также необходимо добавить в сцену минимум три объекта различной формы из реального мира (не примитивы: кубоид, сфера или цилиндр), например: дорожный конус, дерево, стол.

Подготовка модели робота

1. Скопируйте в новый пакет файлы модели робота с дифференциальным приводом, которые были созданы на практических работах 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 с использованием плагина diff_drive

3. Далее необходимо добавить в модель робота сенсоры следующих типов: лидар и камера.

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

Написание управляющей программы

В данной работе также необходимо написать управляющую программу, которая будет считывать данные с сенсоров и формировать управляющее воздействие для /cmd_vel. Цель алгоритма управления может быть любой, как и поведение робота, при этом задача робота должна быть понятной внешнему наблюдателю. Например (данный 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 и параметрам, указанным в Таблице 14.1. Пакет с лабораторной работой должен быть загружен в папку с названием **lab5** в приватном каталоге на сайте gitlab (согласно инструкции, предоставленной ранее). Имитационная модель должна запускаться из пакета **my_best_robot_simulation_setup** при выполнении файла запуска с названием **simulation_final.launch.py**.

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

Таблица 14.1. Параметры системы

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

СПИСОК ЛИТЕРАТУРЫ

[1] OMG Data Distribution Service (DDS) specification. — https://www.omg.org/spec/DDS/. — [Дата обращения: 1.03.2026].

[2] Ubuntu Сообщество. Операционная система Ubuntu. — https://ubuntu.com/. — [Дата обращения: 10.02.2026].

[3] Ислам Бжихатлов. Дополнительные материалы курса "Операционная система ROS 2". — https://likerobotics.ru/courses/ros2/. — [Дата обращения: 1.04.2026].

[4] eProsima Fast DDS documentation. — https://fast-dds.docs.eprosima.com/en/latest/. — [Дата обращения: 1.03.2026].

[5] RMW implementations — ROS 2 Jazzy. — https://docs.ros.org/en/jazzy/Installation/RMW-Implementations.html. — [Дата обращения: 27.02.2026].

[6] Документация по Colcon (colcon — collective construction). — https://colcon.readthedocs.io/en/released/user/installation.html. — [Дата обращения: 1.02.2026].

[7] rosdep — ROS dependency management tool. — https://docs.ros.org/en/jazzy/How-To-Guides/Installing-rosdep.html. — [Дата обращения: 1.03.2026].

[8] Recording and playing back data — ROS 2 Jazzy Tutorial. — https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html. — [Дата обращения: 1.03.2026].

[9] Gazebo Harmonic documentation. — https://gazebosim.org/docs/harmonic/. — [Дата обращения: 28.02.2026].

[10] Use ROS 2 to interact with Gazebo. — https://gazebosim.org/docs/latest/ros2_integration/. — [Дата обращения: 27.02.2026].

[11] SDFormat specification. — http://sdformat.org/spec. — [Дата обращения: 27.02.2026].

[12] ros2_control documentation — Jazzy. — https://control.ros.org/jazzy/. — [Дата обращения: 27.02.2026].

[13] Microsoft. Документация по подсистеме Windows для Linux. — https://learn.microsoft.com/ru-ru/windows/wsl/. — [Дата обращения: 13.02.2025].

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

[15] Introducing turtlesim — ROS 2 Jazzy Tutorial. — https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html. — [Дата обращения: 27.02.2026].

[16] Executror in ROS2 Jazzy. — https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Executors.html. — [Дата обращения: 23.03.2026].

[17] 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].

[18] ROS2 Control. — https://control.ros.org/jazzy/doc/gz_ros2_control/doc/index.html. — [Дата обращения: 28.09.2025].

[19] cv_bridge — ROS 2 package. — https://docs.ros.org/en/jazzy/p/cv_bridge/. — [Дата обращения: 27.02.2026].

[20] Официальная документация библиотеки OpenCV. — https://opencv.org/. — [Дата обращения: 04.03.2026].

[21] Образовательный робот-манипулятор с ROS 2. — https://hertzrobotics.ru/catalog/obrazovatelnye/chetyreh-osevoj-robot-hr-300-edu.html. — [Дата обращения: 19.03.2026].

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

15 Приложение А. Подготовка операционной системы

Предупреждение: работа с ROS2 на устройствах Mac может потребовать больших усилий в части системного администрирования, поэтому не рекомендуется использование компьютеров Mac.

В общем случае можно выделить 3 основных варианта подготовки операционной системы для использования ROS 2, каждый из которых имеет свои преимущества и недостатки.

- Установка Ubuntu 24.04 (рекомендуемый вариант для ROS 2 Jazzy Jalisco).
- Установка WSL 2 на операционную систему Windows.
- Установка Docker и использование готовых образов с ROS 2.

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

Пользователям Ubuntu 24.04 Docker не нужен — достаточно сразу перейти к разделу установки ROS 2. Если вы работаете на другой версии Ubuntu или в WSL, вам может потребоваться Docker [22].

15.1 Для Windows — настройка WSL 2

Для данного курса можно использовать WSL 2 внутри ОС Windows 10 (версия 2004 и выше) или Windows 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 -d Ubuntu-24.04
   ```

   и нажмите **Enter**. Эта команда установит WSL 2 с дистрибутивом Ubuntu 24.04. Во время установки или после

неё система может запросить ввод имени пользователя и пароля. Обязательно сохраните пароль — он понадобится позже.

3. После завершения установки перезагрузите компьютер, если система об этом попросит.

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

```
wsl
```

   чтобы войти в Linux. Также можно найти приложение **Ubuntu 24.04** в меню «Пуск».

5. Далее переходите к разделу установки ROS 2.

Замечание. WSL 2 по умолчанию может не поддерживать GUI-приложения без дополнительной настройки. В Windows 11 поддержка графических приложений Linux (WSLg) включена автоматически. В Windows 10 может потребоваться дополнительная настройка X-сервера. Также WSL имеет проблемы с доступом к подключенным через USB устройствам, кроме того, доступ к другим физическим устройствам через сеть требует дополнительной настройки, поэтому WSL рекомендуется только если вы планируете использовать ROS2 для учебных целей.

15.2 Для пользователей других версий Ubuntu

Если вы используете Ubuntu, отличную от версии 24.04, рекомендуем воспользоваться Docker для запуска ROS 2 Jazzy. Следуйте инструкции по установке Docker ниже.

Установка Docker (при необходимости)

Docker — это инструмент, позволяющий запускать изолированные среды (контейнеры) с нужной операционной системой и предустановленным ПО.

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, чтобы не требовался `sudo` для каждой команды: `https://docs.docker.com/engine/install/linux-postinstall/`

После установки переходите к разделу настройки работы в Docker.

Настройка работы в Docker

Авторы подготовили bash-скрипты для работы с Docker. Чтобы получить их, 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 -la
```

   Если файлы не являются исполняемыми (не зеленый цвет названия), выполните:

```
chmod +x *.bash
```

4. Выполните следующие команды:

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

Первая команда разрешает GUI-приложениям внутри контейнера отображаться на вашем экране. Остальные команды собирают образ и запускают контейнер.

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

Запуск дополнительного терминала внутри Docker

Если вам нужен ещё один терминал внутри уже запущенного контейнера Docker, нельзя использовать скрипт `docker_run.bash` повторно. Вместо этого:

1. Откройте новый терминал и перейдите в каталог Docker:

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

2. Выполните команду:

```
./docker_new.bash
```

При использовании Docker-скриптов курса ROS 2 Jazzy будет уже предустановлен внутри контейнера.

15.3 Установка ROS 2 Jazzy для Ubuntu 24.04

Если вы работаете на Ubuntu 24.04 (нативно или в WSL 2), ROS 2 Jazzy можно установить напрямую из deb-пакетов. Ниже приведена пошаговая инструкция.

Шаг 1. Настройка локали

Убедитесь, что ваша система поддерживает UTF-8:

```
locale
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale
```

Шаг 2. Подключение репозитория ROS 2

Активируйте репозиторий Ubuntu Universe и добавьте источник пакетов ROS 2:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Затем установите пакет `ros2-apt-source`, который автоматически настроит ключи и источники:

```
sudo apt update && sudo apt install curl -y
export ROS_APT_SOURCE_VERSION=$(curl -s \
https://api.github.com/repos/ros-infrastructure\
/ros-apt-source/releases/latest \
| grep -F "tag_name" | awk -F'"' '{print $4}')
curl -L -o /tmp/ros2-apt-source.deb \
"https://github.com/ros-infrastructure\
/ros-apt-source/releases/download\
/${ROS_APT_SOURCE_VERSION}\
/ros2-apt-source_${ROS_APT_SOURCE_VERSION}\
.$(. /etc/os-release \
&& echo ${UBUNTU_CODENAME:-${VERSION_CODENAME}})\
_all.deb"
sudo dpkg -i /tmp/ros2-apt-source.deb
```

Шаг 3. Установка ROS 2

Обновите кэш пакетов и установите ROS 2 Jazzy:

```
sudo apt update
sudo apt upgrade
sudo apt install ros-jazzy-desktop
```

Пакет `ros-jazzy-desktop` включает в себя основные библиотеки, RViz, демонстрационные пакеты и утилиты. Если вам не нужен графический интерфейс (например, на сервере), можно установить минимальную версию:

```
sudo apt install ros-jazzy-ros-base
```

Шаг 4. Установка инструментов разработки (опционально)

Для сборки собственных пакетов рекомендуется установить набор инструментов:

```
sudo apt install ros-dev-tools
```

Шаг 5. Настройка окружения

Для использования ROS 2 необходимо подключать (source) файл настройки окружения в каждом терминале:

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

Чтобы не выполнять эту команду каждый раз, добавьте её в `~/.bashrc`:

```
echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc
```

Шаг 6. Проверка установки

Для проверки откройте два терминала. В первом запустите:

```
ros2 run demo_nodes_cpp talker
```

Во втором терминале запустите:

```
ros2 run demo_nodes_py listener
```

Если `talker` выводит сообщения вида `Publishing: 'Hello World: N'`, а `listener` — `I heard: [Hello World: N]`, установка выполнена успешно.

Бжихатлов Ислам Асланович, Ткачев Игорь Юрьевич, Зименко Константин Александрович, Власов Сергей Михайлович, Маргун Алексей Анатольевич

Операционная система для роботов ROS 2

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