

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению лабораторных работ
по дисциплине «Управление автономными системами» для студентов направления
подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	3
Лабораторная работа 1. Установка и настройка ROS.....	4
Лабораторная работа 2. Создание первого ROS-пакета.....	7
Лабораторная работа 3. Издатель (Publisher) на C++.....	10
Лабораторная работа 4. Подписчик (Subscriber) на C++.....	13
Лабораторная работа 5. Издатель и подписчик на Python.....	16
Лабораторная работа 6. Пользовательские сообщения (Custom Messages).....	19
Лабораторная работа 7. Сервисы (Services): клиент и сервер.....	22
Лабораторная работа 8. Действия (Actions): клиент и сервер.....	25
Лабораторная работа 9. Работа с симулированным лидаром в Gazebo.....	28
Лабораторная работа 10. Визуализация сенсоров в RViz.....	31
Лабораторная работа 11. Одометрия и её визуализация.....	34
Лабораторная работа 12. Статическая карта (Occupancy Grid).....	37
Лабораторная работа 13. AMCL — адаптивная локализация Монте-Карло.....	40
Лабораторная работа 14. Слияние одометрии и IMU (фильтр Калмана).....	43
Лабораторная работа 15. Реализация алгоритма Дейкстры на карте.....	46
Лабораторная работа 16. Реализация алгоритма A*.....	49
Лабораторная работа 17. Интеграция A* в ROS-узел.....	52
Лабораторная работа 18. Dynamic Window Approach (DWA) — локальный планировщик.....	55
Лабораторная работа 19. DWA в ROS.....	58
Лабораторная работа 20. Timed Elastic Band (TEB) — альтернатива DWA.....	61
Лабораторная работа 21. PID-регулятор для управления роботом.....	64
Лабораторная работа 22. ROS-узел управления (Controller) с PID.....	67
Лабораторная работа 23. Конечный автомат для поведения робота.....	70
Лабораторная работа 24. Дерево поведения (Behavior Tree) в ROS.....	73
Лабораторная работа 25. Навигационный стек ROS: полная настройка.....	76
Лабораторная работа 26. Анализ производительности навигации.....	79
Лабораторная работа 27. Обработка динамических препятствий.....	82
Лабораторная работа 28. Повторное планирование (Replanning).....	85
Лабораторная работа 29. Запись и воспроизведение rosbag.....	88
Лабораторная работа 30. Создание пользовательского costmap layer.....	91
Лабораторная работа 31. Интеграция с компьютерным зрением (детекция объектов).....	94
Лабораторная работа 32. Многоагентная система: два робота.....	97
Лабораторная работа 33. Простая реализация SLAM (GMapping).....	100
Лабораторная работа 34. Комплексный проект: навигация к цели с объездом препятствий.....	103

Лабораторная работа 35. Проект: дерево поведения для комплексной миссии	106
Лабораторная работа 36. Итоговая защита: демонстрация автономной системы.....	109

ВВЕДЕНИЕ

Методические указания предназначены для студентов направления подготовки 09.03.04 «Программная инженерия» (направленность «Разработка программного обеспечения автономных и робототехнических систем»). Дисциплина «Управление автономными системами» изучается в рамках подготовки специалистов в области разработки программного обеспечения для робототехнических и автономных систем.

Дисциплина формирует у студентов системное представление о принципах построения и программирования автономных мобильных систем на базе Robot Operating System 2 (ROS 2). Студенты приобретают навыки разработки ROS-узлов, настройки коммуникационных механизмов, работы с сенсорными данными, построения карт, локализации, планирования движения и реализации алгоритмов управления.

Цель лабораторного практикума — закрепить теоретические знания и сформировать устойчивые практические навыки работы с инструментами ROS 2, симулятором Gazebo, средством визуализации RViz и навигационным стеком Nav2.

Практикум построен по принципу последовательного усложнения: от базовых механизмов ROS 2 (топики, сервисы, действия) через работу с сенсорами и картами к алгоритмам планирования пути и управления, деревьям поведения, многоагентным системам и комплексным проектам.

Каждая лабораторная работа включает цель и содержание работы, теоретическое обоснование, разбор типового примера, порядок выполнения индивидуального задания, варианты заданий, требования к отчёту и вопросы для защиты.

Результатом выполнения каждой работы является оформленный отчёт, который сдаётся преподавателю и защищается в форме собеседования по контрольным вопросам.

Лабораторная работа 1.

УСТАНОВКА И НАСТРОЙКА ROS

Цель и содержание работы: Овладеть навыками установки и первичной настройки среды Robot Operating System 2 (ROS 2) дистрибутива Humble Hawksbill на операционной системе Ubuntu. Изучить структуру рабочего пространства ROS 2, освоить команды управления переменными окружения и проверить работоспособность установки с помощью штатных тестовых узлов.

Теоретическое обоснование

ROS 2 — это промышленная программная платформа для разработки робототехнических приложений, построенная на архитектуре распределённых процессов (узлов), взаимодействующих через механизмы публикации/подписки (топики), службы и действия. В отличие от первой версии ROS, ROS 2 лишён центрального координирующего узла (roscore), использует протокол DDS для межузлового обмена и обладает улучшенной поддержкой реального времени, безопасности и многоплатформенности.

Дистрибутив Humble Hawksbill ориентирован на Ubuntu 22.04 LTS и обеспечивает долгосрочную поддержку. Установка выполняется из официальных deb-репозиториях, после чего становятся доступны базовые пакеты, утилиты командной строки (ros2), инструмент сборки colcon и менеджер зависимостей rosdep.

Центральным элементом организации кода является рабочее пространство (workspace) — иерархия каталогов с исходными текстами пакетов, файлами сборки и сгенерированными артефактами. Каждое рабочее пространство содержит папку src для исходников и, после сборки, папки build, install и log. Активация рабочего пространства осуществляется командой source install/setup.bash, которая дополняет переменные окружения

(PATH, LD_LIBRARY_PATH, PYTHONPATH, AMENT_PREFIX_PATH и др.) путями к исполняемым файлам и разделяемым библиотекам данного пространства.

Ключевые переменные окружения ROS 2:

- ROS_DISTRO — идентификатор активного дистрибутива (humble);
- ROS_DOMAIN_ID — номер домена DDS, используемый для изоляции групп узлов в одной сети;
- AMENT_PREFIX_PATH — список префиксов установленных пакетов;
- RMW_IMPLEMENTATION — реализация промежуточного слоя DDS (по умолчанию rmw_fastrtps_cpp).

Для исторической справки: в ROS 1 запуск системы начинался с команды `roscore`, запускавшей главный узел. В ROS 2 `roscore` не требуется, вместо этого используется `ros2 run <пакет> <узел>`. Однако в некоторых учебных и производственных задачах студент может столкнуться с обоими поколениями ROS, поэтому полезно уметь запускать `roscore` (для ROS 1) и `ros2 run` (для ROS 2) и понимать принципиальную разницу.

Пример

Рассмотрим типовой сценарий установки ROS 2 Humble на Ubuntu 22.04, создание рабочего пространства и проверку работоспособности.

Исходные данные: компьютер с установленной Ubuntu 22.04 LTS (нативная или виртуальная машина), доступ в интернет, пользователь с правами `sudo`.

1. Настройка репозитория и установка ROS 2 Humble.

Откройте терминал и выполните:

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

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

sudo apt update
sudo apt install ros-humble-desktop

```

Этот набор команд добавляет ключ репозитория, прописывает источник пакетов и устанавливает полную версию рабочего стола ROS 2 Humble (включает инструменты визуализации, симуляторы и библиотеки).

2. Настройка окружения.

Для автоматической активации ROS 2 при каждом запуске терминала добавьте в файл ~/.bashrc строку:

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

Примените изменения:

```
source ~/.bashrc
```

Проверьте установку:

```
printenv ROS_DISTRO
```

Ожидается вывод: humble.

3. Создание и сборка рабочего пространства.

Создайте каталог рабочего пространства и папку src:

```
mkdir -p ~/ros2_ws/src
```

Перейдите в корень рабочего пространства и выполните сборку (даже если папка src пуста, colcon создаст необходимые структуры):

```
cd ~/ros2_ws
colcon build --symlink-install
```

Ключ --symlink-install позволяет использовать символические ссылки на исходные файлы в установочном каталоге, что ускоряет разработку на Python.

После успешной сборки активируйте рабочее пространство:

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

Для автоматизации этого шага также можно добавить эту команду в ~/.bashrc после строки активации глобального окружения.

4. Проверка работоспособности.

Запустите первый узел-издатель (talker) из штатного пакета demo_nodes_cpp:

```
ros2 run demo_nodes_cpp talker
```

В другом терминале (не забудьте активировать окружение) запустите узел-подписчик (listener):

```
ros2 run demo_nodes_cpp listener
```

Убедитесь, что в терминале с listener появляются сообщения «Hello World: номер». Нажмите Ctrl+C в обоих окнах для остановки.

5. Историческая демонстрация ROS 1 (если установлен ROS 1 Noetic):

Запуск roscore и классического talker/listener позволит сравнить подходы. В ROS 2 roscore отсутствует, его функцию распределённого обнаружения берёт на себя DDS.

Работа считается выполненной, если команды ros2 run запускают узлы без ошибок, рабочее пространство собирается и активируется корректно.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

Варианты индивидуальных заданий определяются преподавателем.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое ROS 2 и чем он отличается от ROS 1?
2. Какой дистрибутив ROS 2 используется в работе и каков срок его поддержки?
3. Что такое рабочее пространство ROS 2? Каковы функции каталогов src, build, install, log?
4. Что делает команда `source /opt/ros/humble/setup.bash`?
5. Что такое DDS и зачем он используется в ROS 2?
6. Что такое `ROS_DOMAIN_ID` и как он влияет на взаимодействие узлов?
7. Как проверить корректность установки ROS 2?
8. Почему в ROS 2 не требуется запускать `roscore`?

Лабораторная работа 2.

СОЗДАНИЕ ПЕРВОГО ROS-ПАКЕТА

Цель и содержание работы: Приобретение практических навыков создания собственного ROS 2 пакета на языке C++, ознакомление со структурой пакета, файлами описания зависимостей (package.xml, CMakeLists.txt), сборкой посредством colcon и запуском минимального узла.

Теоретическое обоснование

Рабочее пространство и пакеты

Программное обеспечение в ROS 2 организуется в виде пакетов — минимальных единиц сборки и распространения. Каждый пакет содержит исходный код, описание зависимостей и инструкции для системы сборки. Пакеты размещаются в рабочих пространствах. Типовая иерархия выглядит следующим образом:

```
workspace/
├── src/      # исходные тексты пакетов
├── build/    # промежуточные файлы сборки
├── install/  # установленные файлы после colcon build
└── log/     # журналы сборки
```

Создание пакета выполняется утилитой `ros2 pkg create`. Для пакета на C++ указывается тип сборки `ament_cmake` и список зависимостей. Команда генерирует каркас, включающий файлы `package.xml` и `CMakeLists.txt`, а также каталоги `src` и `include`.

Файл package.xml

Содержит метаинформацию о пакете и объявляет зависимости. В ROS 2 для большинства случаев используется один тип тега зависимостей —

<depend>, который объединяет этапы сборки, экспорта и выполнения. Допускаются также более детальные теги <build_depend>, <build_export_depend>, <exec_depend>. Основные поля:

- <name> — имя пакета;
- <version> — версия (обычно 0.0.0 на начальном этапе);
- <description> — краткое описание;
- <maintainer email="..."> — данные сопровождающего;
- <license> — лицензия;
- <depend> — зависимость от другого пакета.

Файл CMakeLists.txt

Содержит команды CMake, необходимые для сборки пакета с использованием ament_cmake. Минимальное содержание выглядит так:

```
cmake_minimum_required(VERSION 3.8)
project(my_package)
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
add_executable(empty_node src/empty_node.cpp)
ament_target_dependencies(empty_node rclcpp std_msgs)
install(TARGETS empty_node DESTINATION lib/${PROJECT_NAME})
ament_package()
```

Здесь find_package ищет необходимые пакеты, add_executable создает исполняемую цель, ament_target_dependencies подключает все зависимости (заголовочные файлы, библиотеки, флаги компиляции), install устанавливает узел в каталог lib, а ament_package() завершает конфигурацию.

Минимальный узел

Пустой узел не выполняет полезной работы, но позволяет проверить корректность настройки окружения и сборки. Типичный исходный файл:

```
#include "rclcpp/rclcpp.hpp"
int main(int argc, char ** argv) {
    rclcpp::init(argc, argv);
    auto node = rclcpp::Node::make_shared("empty_node");
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

Узел инициализирует клиентскую библиотеку, создает объект Node с заданным именем и переходит в режим ожидания (spin) до получения сигнала завершения.

Сборка и запуск

Сборка выполняется командой `colcon build` из корня рабочего пространства. Рекомендуется использовать флаг `--packages-select` для сборки только одного пакета. После успешной сборки необходимо активировать окружение командой `source install/setup.bash`. Запуск узла осуществляется инструментом `ros2 run <имя_пакета> <имя_исполняемого_файла>`. Проверить присутствие узла можно командой `ros2 node list`.

Пример

Рассмотрим типовой сценарий создания пакета с именем `example_pkg` и зависимостями `rclcpp`, `std_msgs`.

Подготовка рабочего пространства. Создадим каталог и перейдём в подкаталог `src`:

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

Генерация пакета. Воспользуемся стандартной утилитой:

```
ros2 pkg create example_pkg --build-type ament_cmake --dependencies rclcpp
std_msgs
```

Будет создан каталог `example_pkg` с файлами `package.xml` и `CMakeLists.txt`, а также пустыми каталогами `src` и `include`.

Корректировка `package.xml` (при необходимости). В сгенерированном файле можно уточнить описание и данные сопровождающего:

```
<description>Пример пустого пакета для первой лабораторной</description>
<maintainer email="user@example.com">User</maintainer>
<license>Apache-2.0</license>
```

Зависимости `rclcpp` и `std_msgs` уже присутствуют в виде тегов `<depend>`.

Создание исходного файла пустого узла. В каталоге `src` пакета разместим файл `empty_node.cpp`:

```
#include "rclcpp/rclcpp.hpp"
int main(int argc, char ** argv) {
    rclcpp::init(argc, argv);
    auto node = rclcpp::Node::make_shared("empty_node");
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

Редактирование `CMakeLists.txt`. Убедимся, что файл содержит сборку исполняемого файла. В секцию после `find_package` добавим:

```
add_executable(empty_node src/empty_node.cpp)
ament_target_dependencies(empty_node rclcpp std_msgs)
install(TARGETS empty_node DESTINATION lib/${PROJECT_NAME})
```

Остальной сгенерированный шаблон (вызов `find_package` для `ament_cmake`, `rclcpp`, `std_msgs`, а также `ament_package()`) оставляем без изменений.

Сборка. Перейдём в корень рабочего пространства и выполним:

```
cd ~/ros2_ws
colcon build --packages-select example_pkg
```

При успешной сборке в консоли появятся сообщения `Summary: 1 package finished`.

Настройка окружения. Активируем `overlay` текущего рабочего пространства:

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

Запуск узла и проверка. Запускаем узел:

```
ros2 run example_pkg empty_node
```

В другой консоли (после повторного `source install/setup.bash`) выполняем:

```
ros2 node list
```

В выводе должно присутствовать `/empty_node`. Узел работает, задание выполнено.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Создайте пакет с именем <code>robot_base_pkg</code> и зависимостями <code>rclcpp</code> , <code>geometry_msgs</code> . Реализуйте пустой узел с именем <code>base_node</code> . В исходном файле узла добавьте строку <code>#include <geometry_msgs/msg/twist.hpp></code> (для проверки зависимости). Выполните сборку пакета, настройте окружение и запустите узел. Убедитесь, что узел отображается в выводе <code>ros2 node list</code> .
2	Создайте пакет с именем <code>sensor_demo</code> и зависимостями <code>rclcpp</code> , <code>sensor_msgs</code> . Реализуйте пустой узел с именем <code>sensor_node</code> . В

	исходном файле узла добавьте строку <code>#include <sensor_msgs/msg/image.hpp></code>. Выполните сборку, настройку окружения и запуск. Проверьте наличие узла.
3	Создайте пакет с именем <code>nav_tester</code> и зависимостями <code>rclcpp</code>, <code>nav_msgs</code>. Реализуйте пустой узел с именем <code>nav_node</code>. Подключите заголовок <code><nav_msgs/msg/odometry.hpp></code>. Проведите сборку и проверку работоспособности.
4	Создайте пакет с именем <code>vis_tools</code> и зависимостями <code>rclcpp</code>, <code>visualization_msgs</code>. Реализуйте пустой узел с именем <code>vis_node</code>. В исходном коде используйте <code>#include <visualization_msgs/msg/marker.hpp></code>. Соберите и запустите узел.
5	Создайте пакет с именем <code>trajectory_pkg</code> и зависимостями <code>rclcpp</code>, <code>trajectory_msgs</code>. Реализуйте пустой узел с именем <code>traj_node</code>. Подключите заголовок <code><trajectory_msgs/msg/joint_trajectory.hpp></code>. Выполните сборку и проверьте появление узла в списке.
6	Создайте пакет с именем <code>diagnostic_pkg</code> и зависимостями <code>rclcpp</code>, <code>diagnostic_msgs</code>. Реализуйте пустой узел с именем <code>diag_node</code>. В код добавьте <code>#include <diagnostic_msgs/msg/diagnostic_array.hpp></code>. Соберите, настройте окружение и запустите.
7	Создайте пакет с именем <code>shape_pkg</code> и зависимостями <code>rclcpp</code>, <code>shape_msgs</code>. Реализуйте пустой узел с именем <code>shape_node</code>. Подключите заголовок <code><shape_msgs/msg/mesh.hpp></code>. Выполните сборку и убедитесь, что узел работает.
8	Создайте пакет с именем <code>stereo_pkg</code> и зависимостями <code>rclcpp</code>, <code>stereo_msgs</code>. Реализуйте пустой узел с именем <code>stereo_node</code>. Используйте <code>#include <stereo_msgs/msg/disparity_image.hpp></code>. Соберите и запустите узел.
9	Создайте пакет с именем <code>action_pkg</code> и зависимостями <code>rclcpp</code>,

	action_msgs. Реализуйте пустой узел с именем action_node. В исходный файл добавьте <code>#include <action_msgs/msg/goal_status.hpp></code>. Произведите сборку и проверьте работу узла.
10	Создайте пакет с именем time_utils и зависимостями rclcpp, builtin_interfaces. Реализуйте пустой узел с именем time_node. Подключите заголовок <code><builtin_interfaces/msg/time.hpp></code>. Выполните стандартные шаги сборки и запуска.
11	Создайте пакет с именем uuid_pkg и зависимостями rclcpp, unique_identifier_msgs. Реализуйте пустой узел с именем uuid_node. В коде укажите <code>#include <unique_identifier_msgs/msg/uuid.hpp></code>. Соберите и проверьте функционирование.
12	Создайте пакет с именем tf_pkg и зависимостями rclcpp, tf2_msgs. Реализуйте пустой узел с именем tf_node. Добавьте строку <code>#include <tf2_msgs/msg/tf_message.hpp></code>. Проведите сборку и запуск.
13	Создайте пакет с именем example_iface_pkg и зависимостями rclcpp, example_interfaces. Реализуйте пустой узел с именем iface_node. В исходном файле подключите <code><example_interfaces/msg/int64.hpp></code>. Осуществите сборку, настройку окружения и проверку узла.
14	Создайте пакет с именем rcl_iface_pkg и зависимостями rclcpp, rcl_interfaces. Реализуйте пустой узел с именем rcl_iface_node. Используйте <code>#include <rcl_interfaces/msg/parameter_descriptor.hpp></code>. Соберите и запустите узел.
15	Создайте пакет с именем lifecycle_pkg и зависимостями rclcpp,

	lifecycle_msgs. Реализуйте пустой узел с именем lifecycle_node. Добавьте <code>#include <lifecycle_msgs/msg/state.hpp></code> . Выполните сборку и проверьте работоспособность стандартными средствами.
--	--

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое пакет ROS 2 и каковы его обязательные файлы?
2. Какова роль файла package.xml и как объявить зависимость?
3. Как создать пакет командой `ros2 pkg create`?
4. Что делает `colcon build` и флаг `--packages-select`?
5. Зачем нужна команда `source install/setup.bash` после сборки?
6. Чем `overlay` отличается от `underlay` в ROS 2?
7. Как убедиться, что узел доступен после сборки?
8. Для чего используется `ament_cmake`?

Лабораторная работа 3.

ИЗДАТЕЛЬ (PUBLISHER) НА C++

Цель и содержание работы: Освоить создание узла ROS2 на C++, выполняющего функцию издателя (Publisher). Научиться публиковать сообщения стандартного типа в заданный топик с требуемой частотой, компилировать и запускать узел, а также проверять публикацию с помощью утилиты `ros2 topic echo`.

Теоретическое обоснование

В ROS2 коммуникация между узлами строится на основе анонимной передачи сообщений через топики. Узел-издатель отправляет сообщения определённого типа в топик, а все узлы-подписчики, заинтересованные в этом топике, асинхронно их получают.

Ключевые понятия:

- Узел (Node) — основной вычислительный элемент в графе ROS. Каждый узел имеет уникальное имя в рамках графа.
- Топик (Topic) — именованный канал, через который передаются сообщения. Издатели и подписчики связываются по имени топика.
- Сообщение (Message) — структура данных, передаваемая по топику. Тип сообщения определяется интерфейсным файлом (например, `std_msgs/msg/String`).
- `rclcpp` — клиентская библиотека ROS2 для C++, предоставляющая API для создания узлов, издателей, подписчиков и работы со временем.
- Частота публикации — количество отправляемых сообщений в секунду. Регулируется с помощью механизма ожидания (Rate), чтобы не нагружать систему избыточными вызовами.

Типовой издатель содержит:

- 1. Инициализацию ROS2 и создание объекта узла.**
- 2. Создание объекта издателя с указанием типа сообщения, имени топика и размера очереди (QoS-глубины истории).**
- 3. Циклическую публикацию сообщений по таймеру или в цикле с задержкой, управляемой объектом `rclcpp::Rate` или таймером `wall_timer`.**

Пример

Рассмотрим типовой сценарий создания узла-издателя на C++, публикующего сообщения типа `std_msgs/msg/String` в топик `/talker` с частотой 2 Гц.

Шаг 1. Создание рабочего пространства и пакета

Если рабочее пространство ещё не создано, выполните:

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake --license Apache-2.0 my_publisher_pkg --dependencies rclcpp std_msgs
```

Шаг 2. Написание кода издателя

В каталоге `~/ros2_ws/src/my_publisher_pkg/src` создайте файл `publisher_node.cpp` со следующим содержимым:

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class Talker : public rclcpp::Node
{
public:
    Talker()
    : Node("talker"), count_(0)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("talker", 10);
        timer_ = this->create_wall_timer(
            500ms, std::bind(&Talker::timer_callback, this));
    }

private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_->publish(message);
    }
}
```

```

    }

    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    rclcpp::TimerBase::SharedPtr timer_;
    size_t count_;
};

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<Talker>());
    rclcpp::shutdown();
    return 0;
}

```

Логика работы:

- Конструктор узла Talker создаёт издателя для топика talker с типом String и размером очереди 10.
- Создается wall-таймер, срабатывающий каждые 500 миллисекунд (2 Гц). К моменту срабатывания вызывается метод timer_callback.
- В колбэке формируется сообщение с порядковым номером, оно публикуется и дублируется в лог.
- main выполняет инициализацию, запускает узел в режиме spin (бесконечный опрос событий) и завершает работу.

Шаг 3. Настройка сборки

Отредактируйте CMakeLists.txt в корне пакета так, чтобы он выглядел следующим образом:

```

cmake_minimum_required(VERSION 3.8)
project(my_publisher_pkg)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(talker src/publisher_node.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

install(TARGETS
    talker
    DESTINATION lib/${PROJECT_NAME})

```

```
ament_package()
```

Файл `package.xml` уже содержит зависимости `roscpp` и `std_msgs`, добавленные командой создания пакета; дополнительных правок не требуется.

Шаг 4. Сборка пакета

```
cd ~/ros2_ws  
colcon build --packages-select my_publisher_pkg  
source install/setup.bash
```

Шаг 5. Запуск и проверка

Запустите узел:

```
ros2 run my_publisher_pkg talker
```

В другом терминале, после настройки окружения, проверьте публикацию:

```
source ~/ros2_ws/install/setup.bash  
ros2 topic echo /talker
```

На экране будут появляться строки вида:

```
data: 'Hello, world! 0'  
---  
data: 'Hello, world! 1'  
...
```

Таким образом, издатель работает штатно.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализовать узел <code>chat_publisher</code> , публикующий сообщения типа <code>std_msgs/msg/String</code> в топик <code>/chat</code> с частотой 5 Гц. Каждое сообщение должно содержать текст "Message N", где N — порядковый номер сообщения, начиная с 0.
2	Реализовать узел <code>int_publisher</code> , публикующий сообщения типа <code>std_msgs/msg/Int32</code> в топик <code>/counter</code> с частотой 2 Гц. Значение сообщения должно линейно возрастать: 0, 1, 2, ... с каждой итерацией.
3	Реализовать узел <code>sine_publisher</code> , публикующий сообщения типа <code>std_msgs/msg/Float64</code> в топик <code>/sensor/value</code> с частотой 10 Гц. Значение должно представлять синусоидальный сигнал: $\sin(2 * \pi * 0.1 * t)$, где t — время в секундах с момента старта узла (используйте <code>this->now()</code>).
4	Реализовать узел <code>bool_toggle</code> , публикующий сообщения типа <code>std_msgs/msg/Bool</code> в топик <code>/flag</code> с частотой 1 Гц. Сообщение должно переключаться между <code>true</code> и <code>false</code> каждую итерацию.
5	Реализовать узел <code>dual_publisher</code> , который публикует сообщения типа <code>std_msgs/msg/String</code> одновременно в два топика: <code>/topic_A</code> и <code>/topic_B</code> с частотой 2 Гц. В <code>/topic_A</code> отправлять строку "Hello A", в <code>/topic_B</code> — "Hello B".
6	Реализовать узел <code>array_publisher</code> , публикующий сообщения типа <code>std_msgs/msg/Int32MultiArray</code> в топик <code>/array_data</code> с частотой 0.5 Гц. Массив должен содержать 5 элементов, заполненных значениями 1, 2, 3, 4, 5 (неизменными).
7	Реализовать узел <code>timestamp_pub</code> , публикующий сообщения типа

	std_msgs/msg/String в топик /config_info с частотой 0.2 Гц. Содержимое строки: "Timestamp: <секунды.наносекунды>", формируемое на основе this->now().
8	Реализовать узел temp_sensor_sim, публикующий сообщения типа std_msgs/msg/Float32 в топик /temperature с частотой 4 Гц. Значение — случайное число в диапазоне 20.0 ... 30.0, получаемое с помощью стандартного генератора C++ (std::rand или <random>).
9	Реализовать узел color_cycler, публикующий сообщения типа std_msgs/msg/ColorRGBA в топик /led_color с частотой 3 Гц. Циклически менять цвет сообщения каждые три публикации по схеме: красный (r=1,g=0,b=0), зелёный (r=0,g=1,b=0), синий (r=0,g=0,b=1). Альфа-канал установить в 1.0.
10	Реализовать узел header_pub, публикующий сообщения типа std_msgs/msg/Header (только заголовок) в топик /frame_info с частотой 10 Гц. Поле stamp заполнять текущим временем, поле frame_id устанавливать в "base_link".
11	Реализовать узел alpha_pub, публикующий сообщения типа std_msgs/msg/String в топик /alphabet с частотой 1 Гц. Каждое сообщение содержит очередную букву латинского алфавита: 'a', 'b', 'c', ..., 'z', после 'z' — снова 'a'.
12	Реализовать узел square_pub, публикующий сообщения типа std_msgs/msg/UInt32 в топик /square с частотой 1 Гц. Значение — квадрат номера итерации (0, 1, 4, 9, 16, ...).
13	Реализовать узел binary_pub, публикующий сообщения типа std_msgs/msg/Int8 в топик /binary с частотой 2 Гц. Публиковать попеременно 0 и 1.
14	Реализовать узел row_pub, публикующий сообщения типа

	std_msgs/msg/Float64MultiArray в топик /matrix_row с частотой 1 Гц. Массив содержит неизменные три значения: 1.0, 2.0, 3.0.
15	Реализовать узел log_pub, публикующий сообщения типа std_msgs/msg/String в топик /log с частотой 5 Гц. Сообщение: "Log entry N", где N — счётчик. Дополнительно, когда N кратно 10 (0, 10, 20, ...), выводить предупреждение в лог ROS с уровнем WARN (макрос RCLCPP_WARN).

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое Publisher (издатель) в ROS 2?
2. Как создать узел-издатель на C++ с использованием rclcpp?
3. Что такое топик и как определяется его тип?
4. Как настроить частоту публикации сообщений?
5. Как проверить, что издатель публикует сообщения (ros2 topic echo)?
6. Какова роль QoS в ROS 2?
7. Чем std_msgs/msg/String отличается от пользовательского сообщения?
8. Как корректно завершить работу узла?

Лабораторная работа 4.

ПОДПИСЧИК (SUBSCRIBER) НА C++

Цель и содержание работы: Освоение механизма подписки на топики в ROS 2 с использованием языка C++. Реализация узла-подписчика, принимающего сообщения заданного типа, обработка входящих данных в callback-функции и вывод результатов в терминал. Закрепление понимания асинхронной модели обмена сообщениями между узлами.

Теоретическое обоснование

В ROS 2 взаимодействие между узлами организуется через механизм топиков, реализующий модель издатель-подписчик. Узел, желающий получать данные, объявляет подписку на топик с определённым именем и типом сообщения. При поступлении нового сообщения в топик выполняется функция обратного вызова (callback), в которую передаётся содержимое сообщения. Функция обратного вызова регистрируется при создании подписки и не должна содержать долгих блокирующих операций, так как выполняется в потоке обработки событий.

Для работы подписчика на C++ необходимо подключить заголовочный файл типа сообщения и использовать класс `rclcpp::Node`. Подписка создаётся методом `create_subscription`, принимающим имя топика, размер очереди и функцию-callback. После настройки подписчика следует запустить цикл обработки сообщений с помощью `rclcpp::spin()`, который блокирует основной поток и вызывает зарегистрированные callback-функции по мере поступления данных. Альтернативой `spin()` является использование однопроходного вызова `spin_some()` или асинхронного исполнителя.

Для успешного обмена издатель и подписчик должны использовать одинаковый тип сообщений и совпадающее имя топика. Совместный запуск узлов позволяет наблюдать потоковую передачу данных в реальном времени.

Пример

Рассмотрим типовой сценарий создания подписчика на топик `/chatter` типа `std_msgs/msg/String`. Узел будет принимать строковые сообщения, выводить их в консоль с префиксом и подсчитывать количество полученных сообщений.

Исходный код узла (например, `subscriber_node.cpp`):

```
#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>

using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
public:
    MinimalSubscriber()
        : Node("minimal_subscriber"), count_(0)
    {
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "chatter", 10, std::bind(&MinimalSubscriber::topic_callback, this,
_1));
    }

private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg)
    {
        count_++;
        RCLCPP_INFO(this->get_logger(), "Получено сообщение #d: '%s'", count_,
msg->data.c_str());
    }

    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
    int count_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalSubscriber>());
    rclcpp::shutdown();
    return 0;
}
```

Для компиляции узла необходимо добавить его в `CMakeLists.txt` пакета.

Минимальный фрагмент:

```
add_executable(subscriber_node src/subscriber_node.cpp)
ament_target_dependencies(subscriber_node rclcpp std_msgs)
install(TARGETS subscriber_node DESTINATION lib/${PROJECT_NAME})
```

В package.xml должны быть объявлены зависимости от rclcpp и std_msgs. После сборки пакета командой colcon build узел запускается:

```
ros2 run <имя_пакета> subscriber_node
```

Для проверки связи следует в другом терминале запустить узел-издатель из лабораторной работы 3, публикующий сообщения в топик /chatter с типом String. Например:

```
ros2 run <имя_пакета> publisher_node
```

При корректной настройке в терминале подписчика начнут отображаться принимаемые строки с порядковыми номерами.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Создайте узел-подписчик, принимающий сообщения типа std_msgs/msg/String из топика /input_text. Реализуйте callback-функцию, которая выводит полученную строку, добавляя перед ней префикс "Received: ".

2	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Int32</code> из топика <code>/numbers</code> . В <code>callback</code> -функции вычисляйте и выводите квадрат полученного числа.
3	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Float64</code> из топика <code>/raw_value</code> . В <code>callback</code> -функции выводите значение, округлённое до ближайшего целого.
4	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Bool</code> из топика <code>/flag</code> . В <code>callback</code> -функции выводите слово "True", если флаг установлен, и "False" в противном случае.
5	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/String</code> из топика <code>/lines</code> . В <code>callback</code> -функции выводите номер полученного сообщения (по порядку) и сам текст в формате "N: <текст>".
6	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Int32MultiArray</code> из топика <code>/array_data</code> . В <code>callback</code> -функции вычисляйте и выводите сумму всех элементов массива.
7	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Float64MultiArray</code> из топика <code>/sensor_readings</code> . В <code>callback</code> -функции вычисляйте и выводите среднее арифметическое элементов массива.
8	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/String</code> из топика <code>/text_upper</code> . В <code>callback</code> -функции преобразуйте строку к верхнему регистру и выводите результат.
9	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Int32</code> из топика <code>/even_filter</code> . В <code>callback</code> -функции выводите число только если оно чётное; нечётные числа игнорируйте.

10	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Float64</code> из топика <code>/temperature</code> . В callback-функции выводите значение с точностью до двух знаков после запятой.
11	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/String</code> из топика <code>/full_name</code> . Разбейте строку по пробелам и выведите в консоль первое слово.
12	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Int32</code> из топика <code>/accumulator</code> . В callback-функции накапливайте сумму всех принятых чисел и выводите текущую сумму.
13	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/String</code> из топика <code>/alphanumeric</code> . Проверьте, содержит ли строка хотя бы одну цифру, и выведите результат проверки.
14	Создайте узел-подписчик, принимающий сообщения типа <code>geometry_msgs/msg/Point</code> из топика <code>/position</code> . Вычислите расстояние от точки до начала координат по формуле $\sqrt{x^2 + y^2 + z^2}$ и выведите его.
15	Создайте узел-подписчик, принимающий сообщения типа <code>std_msgs/msg/Int32</code> из топика <code>/binary_view</code> . В callback-функции выводите полученное число в двоичном представлении (в виде строки из 0 и 1).

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа `.docx` или `.pdf`.

Вопросы для защиты работы

1. Что такое Subscriber (подписчик) в ROS 2?
2. Как организована обработка входящих сообщений (callback)?
3. Как проверить список активных подписок (ros2 topic info)?
4. Что произойдёт при несовпадении QoS у издателя и подписчика?
5. Чем `rclcpp::spin` отличается от `rclcpp::spin_some`?
6. Как разместить несколько подписчиков в одном узле?
7. Что такое `latched topic`?
8. Как убедиться, что подписчик получает сообщения от издателя?

Лабораторная работа 5.

ИЗДАТЕЛЬ И ПОДПИСЧИК НА PYTHON

Цель и содержание работы: Освоить создание издателей и подписчиков на языке Python с использованием библиотеки `rclpy` в ROS 2. Получить практический опыт разработки, запуска и отладки коммуникационных узлов, а также провести сравнение синтаксиса и особенностей реализации с узлами на C++ для осознанного выбора языка под конкретную задачу.

Теоретическое обоснование

В ROS 2 взаимодействие между узлами организуется через механизм публикации и подписки на топики. Издатель (`publisher`) отправляет сообщения определённого типа в именованный топик, подписчик (`subscriber`) принимает эти сообщения, выполняя заданную функцию обратного вызова. Библиотека `rclpy` предоставляет Python-интерфейс к клиентской библиотеке ROS 2, построенной на основе `rcl`.

Ключевые этапы создания узла на `rclpy`:

1. Инициализация клиентской библиотеки вызовом `rclpy.init()`.
2. Создание объекта узла наследованием от `rclpy.node.Node` или инстанцированием непосредственно `Node`.
3. Для издателя: создание объекта `publisher` с помощью `node.create_publisher(MsgType, 'topic_name', qos_profile)`. Публикация выполняется вызовом `publisher.publish(msg)`.
4. Для подписчика: создание объекта `subscription` с помощью `node.create_subscription(MsgType, 'topic_name', callback, qos_profile)`. Функция `callback` принимает единственный аргумент — принятое сообщение.

5. Запуск цикла обработки событий `rclpy.spin(node)`, который блокирует поток и вызывает коллбэки по мере поступления сообщений. Для периодической публикации используется таймер `node.create_timer(period, callback)`.

6. Завершение работы: уничтожение узла и вызов `rclpy.shutdown()`.

Сравнение с C++ (`rclcpp`):

- Синтаксис Python более лаконичен: не требуется явного управления указателями, проще работа со строками и структурами данных.
- В C++ необходимо компилировать узел, в Python достаточно интерпретируемого скрипта, что ускоряет цикл разработки.
- Python-узлы могут иметь чуть большую задержку и более высокое потребление ресурсов, что не критично для прототипирования и высокоуровневой логики. C++ предпочтителен для задач с жёсткими требованиями по производительности и для низкоуровневого взаимодействия с аппаратурой.
- Отладка Python-узлов проще за счёт встроенных средств трассировки и возможности интерактивного запуска в интерпретаторе.

При запуске Python-узлов необходимо убедиться, что пакет корректно установлен в рабочее пространство, а скрипты имеют права на исполнение и начинаются с корректного шебанга `#!/usr/bin/env python3`.

Пример

Рассмотрим типовой сценарий: создание минимального издателя, публикующего строку «Hello from Python» с частотой 1 Гц, и подписчика, выводящего полученное сообщение.

Создание пакета и узлов выполняется в рабочем пространстве ROS 2 (например, `ros2_ws/src`). Предполагается, что пакет уже существует, либо создаётся новый командой:

```
ros2 pkg create --build-type ament_python lab5_python_demo --dependencies rclpy std_msgs
```

В каталоге пакета `lab5_python_demo/lab5_python_demo` создаются два файла: `publisher_node.py` и `subscriber_node.py`.

Содержимое `publisher_node.py`:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'chatter', 10)
        timer_period = 1.0 # секунд
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.get_logger().info('Издатель запущен')

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello from Python'
        self.publisher_.publish(msg)
        self.get_logger().info(f'Опубликовано: {msg.data}')

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

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

Содержимое `subscriber_node.py`:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String, 'chatter', self.listener_callback, 10)
```

```

        self.get_logger().info('Подписчик запущен')

    def listener_callback(self, msg):
        self.get_logger().info(f'Получено: {msg.data}')

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

if __name__ == '__main__':
    main()

```

Необходимо сделать файлы исполняемыми:

```

chmod +x publisher_node.py subscriber_node.py

```

В файле `setup.py` пакета (находится в корне пакета) в функции `setup` необходимо указать точки входа для консольных скриптов:

```

entry_points={
    'console_scripts': [
        'publisher = lab5_python_demo.publisher_node:main',
        'subscriber = lab5_python_demo.subscriber_node:main',
    ],
},

```

Сборка пакета:

```

cd ~/ros2_ws
colcon build --packages-select lab5_python_demo
source install/setup.bash

```

```

Запуск узлов:
ros2 run lab5_python_demo publisher
ros2 run lab5_python_demo subscriber

```

В терминале подписчика будут отображаться принимаемые сообщения. Для проверки можно использовать `ros2 topic echo /chatter` или `ros2 topic list`. Данный пример демонстрирует типовой процесс реализации коммуникации на Python.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;

- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Издатель публикует в топик /time_now сообщение типа String с текущей датой и временем в формате ISO каждые 2 секунды. Подписчик принимает сообщение и выводит в логи временную метку вместе с временем приёма.
2	Издатель каждую секунду генерирует случайное целое число в диапазоне от 0 до 100 и публикует в топик /random_int (тип Int32). Подписчик суммирует все принятые значения и после каждых 10 сообщений выводит накопленную сумму.
3	Издатель с периодом 500 мс отправляет в топик /sensor_value случайное вещественное число (тип Float64) от -10.0 до 10.0. Подписчик вычисляет скользящее среднее последних десяти значений и выводит его при каждом новом значении.
4	Издатель читает строки, вводимые пользователем с клавиатуры, и публикует их в топик /raw_input (тип String). Подписчик преобразует каждую полученную строку к верхнему регистру и выводит результат в лог.
5	Первый издатель публикует в топик /coord_x случайное вещественное число (Float64), второй издатель — в топик /coord_y случайное вещественное число. Подписчик

	подписывается на оба топика, при получении пары значений формирует строку вида "Point(x, y)" и выводит её.
6	Издатель с частотой 2 Гц публикует в топик /velocity сообщение типа Twist (linear.x присваивается постоянное значение 0.5). Подписчик интегрирует скорость по времени (считая шаг дискретизации 0.5 с) и выводит текущее перемещение по оси X.
7	Издатель-счётчик с интервалом 1 с публикует в топик /count значение типа Int32, увеличивающееся на 1 с каждой итерацией. Подписчик выводит сообщение "Кратное 5" только когда принятое число кратно 5.
8	Издатель публикует сообщение "Hello ROS" (String) в топик /greeting с частотой 2 Гц. Подписчик подсчитывает общее количество принятых сообщений и каждые 10 сообщений выводит это количество в лог.
9	Издатель публикует в топик /array_data массив из пяти целых чисел (тип Int32MultiArray), заполненный случайными значениями. Подписчик вычисляет сумму элементов каждого массива и выводит её.
10	Издатель генерирует случайные координаты x и y (тип Float64MultiArray из двух элементов) и публикует в топик /position. Подписчик вычисляет расстояние точки от начала координат ($\sqrt{x^2 + y^2}$) и выводит это значение.
11	Издатель с периодом 1 с случайным образом публикует в топик /status сообщение String: "Sensor OK" или "Sensor Fail". Подписчик ведёт статистику: считает количество сообщений каждого типа и каждые 5 секунд выводит итоговые счётчики.
12	Издатель публикует два топика: /temperature со значением Float64 (от 20 до 40) и /humidity со значением Float64 (от 30 до

	90). Подписчик подписывается на оба, вычисляет упрощённый индекс тепла по формуле: $HI = 0.5 * (T + 61.0 + ((T-68.0)*1.2) + (H*0.094))$ и выводит результат при обновлении любого из значений.
13	Издатель публикует в топик /scan массив расстояний (Float32MultiArray) из 8 элементов, моделируя показания дальномера. Подписчик находит минимальное расстояние в каждом массиве и выводит его.
14	Реализовать узел-ретранслятор (комбинированный узел). Узел подписывается на топик /input (тип String), в коллбэке добавляет к полученной строке префикс "Echo: " и публикует изменённую строку в топик /output.
15	Издатель моделирует угол поворота шарнира в градусах как синусоиду с амплитудой 90 градусов и периодом 10 с, публикует значение типа Float64 в топик /joint_angle с частотой 10 Гц. Подписчик переводит градусы в радианы и выводит оба значения.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Чем Python-узлы отличаются от C++-узлов в ROS 2?
2. Как создать издатель и подписчик в одном Python-узле?

3. Какую роль играет `rclpy.init()` и `rclpy.shutdown()`?
4. Как задать периодический таймер в Python?
5. Как запустить Python-узел без компиляции?
6. Как проверить активные топики (`ros2 topic list`)?
7. Для чего в Python-пакете нужен файл `setup.py`?
8. Какие ограничения производительности у Python-узлов по сравнению с C++?

Лабораторная работа 6.

ПОЛЬЗОВАТЕЛЬСКИЕ СООБЩЕНИЯ (CUSTOM MESSAGES)

Цель и содержание работы: Освоить создание пользовательских типов сообщений в ROS, настройку пакета для их генерации и практическое применение собственных сообщений в узлах-издателях и подписчиках.

Теоретическое обоснование

Сообщения в ROS являются основным механизмом обмена данными между узлами. В стандартной поставке имеется множество готовых типов (`std_msgs`, `sensor_msgs`, `geometry_msgs` и др.), однако при разработке реальных робототехнических систем часто возникает необходимость передавать структурированные данные специфического формата. Для этого служат пользовательские сообщения, определяемые в файлах с расширением `.msg`.

Файл `.msg` содержит перечень полей, каждое из которых описывается типом и именем. Допускаются целочисленные типы (`int8`, `int16`, `int32`, `int64` и беззнаковые варианты), числа с плавающей точкой (`float32`, `float64`), строки (`string`), логический тип (`bool`), массивы (обозначаются квадратными скобками, например `int32[] data`) и вложенные сообщения (использование другого определённого типа). Каждое поле записывается на отдельной строке.

После создания `.msg`-файла необходимо настроить пакет для его компиляции и генерации программного кода (заголовочных файлов C++ или модуля Python). Настройка выполняется в двух файлах: `package.xml` и `CMakeLists.txt`. В `package.xml` требуется объявить зависимости от пакетов `message_generation` (этап сборки) и `message_runtime` (этап выполнения). В `CMakeLists.txt` нужно выполнить следующие действия:

- найти пакет `message_generation` с помощью `find_package`;

- добавить созданные .msg-файлы в переменную с помощью `add_message_files`;
- активировать генерацию сообщений вызовом `generate_messages` с указанием зависимостей, например от `std_msgs`, если используются стандартные типы;
- экспортировать зависимость времени выполнения в `catkin_package`, включив `message_runtime` в `CATKIN_DEPENDS`.

После успешной сборки в пространстве разработки становятся доступны классы сообщений, которые можно импортировать в код узлов. Для C++ заголовочный файл размещается в каталоге `devel/include/имя_пакета/имя_сообщения.h`, для Python — импортируется как `имя_пакета.msg.имя_сообщения`.

Пример

Рассмотрим типовой сценарий создания пользовательского сообщения, содержащего координаты и угол ориентации, и его использование в узлах.

Создадим пакет с именем `my_pose_msgs`, зависящий от `message_generation` и `message_runtime`. Для этого в рабочем пространстве `catkin_ws/src` выполним команду:

```
catkin_create_pkg my_pose_msgs std_msgs message_generation message_runtime
```

Внутри пакета создадим каталог `msg` и файл `MyPose.msg` со следующим содержимым:

```
float64 x
float64 y
float64 theta
```

Теперь отредактируем `package.xml`, добавив (или раскомментировав) следующие строки:

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

Затем изменим CMakeLists.txt. В разделе find_package к списку добавляем message_generation:

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

Раскомментируем и заполним блок add_message_files:

```
add_message_files(
  FILES
  MyPose.msg
)
```

Раскомментируем generate_messages, указав зависимость от std_msgs, так как наши поля используют встроенные типы:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

В catkin_package укажем message_runtime:

```
catkin_package(
  CATKIN_DEPENDS std_msgs message_runtime
)
```

Теперь создадим узлы. Разместим в каталоге scripts пакета файл publisher.py:

```
#!/usr/bin/env python
import rospy
from my_pose_msgs.msg import MyPose

if __name__ == '__main__':
    rospy.init_node('pose_publisher')
    pub = rospy.Publisher('pose_topic', MyPose, queue_size=10)
    rate = rospy.Rate(1)
    msg = MyPose()
    msg.x = 1.0
    msg.y = 2.0
    msg.theta = 0.5
    while not rospy.is_shutdown():
        pub.publish(msg)
        rospy.loginfo("Published custom pose")
        rate.sleep()
```

Сделаем файл исполняемым: chmod +x publisher.py. Аналогично создадим subscriber.py:

```
#!/usr/bin/env python
import rospy
from my_pose_msgs.msg import MyPose
```

```
def callback(data):
    rospy.loginfo(f"Received:           x={data.x:.2f},           y={data.y:.2f},
theta={data.theta:.2f}")

if __name__ == '__main__':
    rospy.init_node('pose_subscriber')
    rospy.Subscriber('pose_topic', MyPose, callback)
    rospy.spin()
```

Выполним сборку в корне рабочего пространства командой `catkin_make`, затем активируем окружение `source devel/setup.bash`. После запуска `roscore` можно выполнить узлы:

```
roslaunch my_pose_msgs publisher.py
roslaunch my_pose_msgs subscriber.py
```

В терминале подписчика будет наблюдаться вывод полученных данных, подтверждающий корректную работу пользовательского сообщения.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Сообщение <code>BatteryStatus</code> : float32 voltage float32 current float32 charge_percent bool is_charging
2	Сообщение <code>MotorSpeeds</code> : int32 left_rpm int32 right_rpm float32 linear_velocity

3	Сообщение SensorReading: float32 temperature float32 humidity float32 pressure string unit
4	Сообщение Waypoint: float64 latitude float64 longitude float64 altitude
5	Сообщение ObstacleInfo: float32 distance float32 angle string obstacle_type
6	Сообщение ImuData: float64 linear_accel_x float64 linear_accel_y float64 linear_accel_z float64 angular_vel_x float64 angular_vel_y float64 angular_vel_z
7	Сообщение ControlCommand: float64 linear_x float64 angular_z
8	Сообщение SystemHealth: int32 cpu_load int32 memory_free bool ros_master_connected
9	Сообщение NavigationGoal: float64 x float64 y float64 theta string frame_id
10	Сообщение JointStateSimple: float64 joint1 float64 joint2 float64 joint3 float64 joint4
11	Сообщение PowerConsumption: float64 power float64 energy_used float64 time_elapsed
12	Сообщение EnvironmentData: float32 light_level float32 noise_level bool smoke_detected
13	Сообщение DiagnosticMessage: string component int32 error_code string message bool is_critical
14	Сообщение RobotPose2D: float64 x float64 y float64 orientation
15	Сообщение SpeedWithCovariance: float64 speed float64 variance

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2,

используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое Custom Message в ROS 2?
2. Как описывается структура сообщения в файле .msg?
3. Как подключить пользовательский тип к пакету издателя?
4. Какие примитивные типы допустимы в .msg файлах?
5. Как сгенерировать заголовочный файл из .msg?
6. Зачем нужен пакет rosidl_default_generators?
7. Как проверить тип сообщения (ros2 interface show)?
8. Чем Custom Message отличается от Custom Service?

Лабораторная работа 7.

СЕРВИСЫ (SERVICES): КЛИЕНТ И СЕРВЕР

Цель и содержание работы: Освоить механизм синхронного взаимодействия узлов в ROS через сервисы: научиться описывать формат запроса и ответа в .srv-файле, реализовывать узел-сервер, обрабатывающий запросы, и узел-клиент, осуществляющий синхронный вызов, а также проверять работоспособность связки клиент-сервер штатными средствами ROS.

Теоретическое обоснование

Сервисы в ROS представляют собой механизм синхронной коммуникации между узлами по принципу «запрос – ответ». В отличие от топиков, где данные передаются в одностороннем потоке, сервис предполагает, что клиент отправляет запрос и ожидает ответа от сервера. Такой подход уместен в задачах, требующих немедленного результата (например, запрос конфигурации, запуск калибровки, единичное измерение).

Формат сервиса задаётся специальным файлом с расширением .srv, который располагается в подкаталоге srv пакета. Внутри файла описываются две секции, разделённые тремя дефисами: первая секция определяет поля запроса, вторая – поля ответа. Синтаксис аналогичен описанию сообщений ROS.

На стороне сервера создаётся объект, который регистрирует имя сервиса в мастер-узле и связывает с ним функцию обратного вызова. При поступлении запроса вызывается эта функция, формируется ответ, и он отсылается обратно клиенту. Клиент создаёт прокси-объект для нужного сервиса и при вызове передаёт заполненный запрос. Вызов блокирует выполнение клиента до получения ответа или истечения таймаута.

Сервисы обеспечивают гарантированную доставку результата конкретному клиенту, что удобно для коротких атомарных операций, но не рассчитано на продолжительные вычисления, так как блокируют вызывающий поток.

Пример

Рассмотрим типовой сценарий реализации сервиса сложения двух целых чисел. В рамках сценария создаётся пакет, описывается .srv-файл, пишется серверный узел, клиентский узел, производится сборка и проверка вызова сервиса.

Создание пакета. В рабочем пространстве ROS (например, catkin_ws/src) выполняется команда:

```
catkin_create_pkg my_adder_service rospy std_msgs message_generation
```

В пакете создаётся каталог srv, в нём — файл AddTwoInts.srv со следующим содержимым:

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

Для того чтобы система сборки увидела сервисный файл, необходимо отредактировать CMakeLists.txt: добавить message_generation в компоненты find_package, в секцию add_service_files внести AddTwoInts.srv, раскомментировать generate_messages с зависимостью от std_msgs, а также в catkin_package указать message_runtime. В файле package.xml нужно убедиться в наличии тегов <build_depend>message_generation</build_depend> и <exec_depend>message_runtime</exec_depend>.

Серверный узел. В каталоге scripts пакета размещается файл adder_server.py со следующим кодом:

```
#!/usr/bin/env python
import rospy
from my_adder_service.srv import AddTwoInts, AddTwoIntsResponse
```

```
def handle_add_two_ints(req):
    rospy.loginfo("Request: a=%d, b=%d" % (req.a, req.b))
    return AddTwoIntsResponse(req.a + req.b)

def adder_server():
    rospy.init_node('adder_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    rospy.loginfo("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    adder_server()
```

Файл должен быть исполняемым: `chmod +x adder_server.py`.

Клиентский узел. Там же создаётся файл `adder_client.py`:

```
#!/usr/bin/env python
import sys
import rospy
from my_adder_service.srv import AddTwoInts, AddTwoIntsRequest

def adder_client(a, b):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp = add_two_ints(a, b)
        return resp.sum
    except rospy.ServiceException as e:
        rospy.logerr("Service call failed: %s" % e)

if __name__ == "__main__":
    rospy.init_node('adder_client')
    if len(sys.argv) != 3:
        rospy.loginfo("Usage: adder_client A B")
        sys.exit(1)
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    result = adder_client(a, b)
    rospy.loginfo("Request: %d + %d = %d" % (a, b, result))
```

Сборка и запуск. Из корня рабочего пространства выполняется `catkin_make`, затем источник девелопмент-скрипта: `source devel/setup.bash`. Далее запускается мастер-узел (`roscore`), сервер (`roslaunch my_adder_service adder_server.py`) и в другом терминале — клиент (`roslaunch my_adder_service adder_client.py 5 7`). Ожидается вывод: `Request: 5 + 7 = 12`.

Проверка без клиентского узла. Сервис можно вызвать напрямую утилитой `rosservice call add_two_ints "a: 3 b: 8"`. Команда вернёт `sum: 11`.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Сервис вычисления суммы двух целых чисел. Запрос: int64 a, int64 b. Ответ: int64 sum.
2	Сервис вычисления разности двух целых чисел. Запрос: int64 a, int64 b. Ответ: int64 difference.
3	Сервис умножения двух целых чисел. Запрос: int64 a, int64 b. Ответ: int64 product.
4	Сервис деления двух чисел с плавающей точкой. Запрос: float64 dividend, float64 divisor. Ответ: float64 quotient, bool success (true, если деление выполнено без ошибок).
5	Сервис конкатенации двух строк. Запрос: string first, string second. Ответ: string combined.
6	Сервис определения длины строки. Запрос: string input_string. Ответ: int64 length.
7	Сервис вычисления площади прямоугольника. Запрос: float64 width, float64 height. Ответ: float64 area.
8	Сервис вычисления периметра прямоугольника. Запрос: float64 width, float64 height. Ответ: float64 perimeter.

9	Сервис вычисления площади круга. Запрос: float64 radius. Ответ: float64 area.
10	Сервис вычисления объёма куба. Запрос: float64 edge. Ответ: float64 volume.
11	Сервис преобразования температуры из градусов Цельсия в Фаренгейты. Запрос: float64 celsius. Ответ: float64 fahrenheit.
12	Сервис проверки целого числа на чётность. Запрос: int64 number. Ответ: bool is_even.
13	Сервис возведения числа в квадрат. Запрос: float64 value. Ответ: float64 square.
14	Сервис вычисления гипотенузы по двум катетам. Запрос: float64 leg_a, float64 leg_b. Ответ: float64 hypotenuse.
15	Сервис вычисления факториала целого неотрицательного числа. Запрос: int64 n. Ответ: int64 factorial, bool success (true, если $n \geq 0$). Для каждого варианта необходимо: создать отдельный ROS-пакет, описать соответствующий .srv-файл, реализовать серверный и клиентский узлы, выполнить сборку, продемонстрировать работоспособность вызовом сервиса с клиентского узла и утилитой rosservice call.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое сервис ROS 2 и чем он отличается от топика?
2. Как определяется структура сервиса в файле .srv?
3. Как создать сервер и клиент сервиса на C++?
4. Как протестировать сервис (ros2 service call)?
5. Что такое синхронный и асинхронный вызов сервиса?
6. Как обработать ситуацию, когда сервис недоступен?
7. Когда предпочтительнее использовать сервис, а не топик?
8. Как убедиться, что сервер сервиса запущен?

Лабораторная работа 8.

ДЕЙСТВИЯ (ACTIONS): КЛИЕНТ И СЕРВЕР

Цель и содержание работы: Изучение механизма actions в ROS 2 как средства организации длительных асинхронных задач с обратной связью и возможностью отмены. Получение практических навыков описания action-интерфейса, реализации action-сервера, выполняющего модельную длительную операцию с периодической публикацией статуса, и action-клиента с функцией досрочного прерывания.

Теоретическое обоснование

Actions (действия) в ROS 2 предназначены для задач, время выполнения которых существенно превышает типичное время обработки одного запроса в сервисах, и при этом требуется информировать вызывающую сторону о ходе выполнения и/или предоставить возможность безопасной отмены. В отличие от топиков (асинхронная однонаправленная передача сообщений) и сервисов (синхронный запрос-ответ), actions базируются на трехкомпонентной модели: цель (Goal), отправляемая клиентом; обратная связь (Feedback), периодически публикуемая сервером; и результат (Result), выдаваемый по завершении задачи. Все три компонента описываются в едином файле с расширением .action, на основе которого системой сборки генерируются необходимые типы сообщений и служебные классы.

Архитектура action-взаимодействия строится на паре клиент-сервер. Сервер регистрирует action-интерфейс по заданному имени, получает цели и выполняет асинхронный обработчик, который может длиться произвольное время. В процессе обработки сервер с заданной периодичностью обновляет состояние обратной связи, делая его доступным клиенту. Клиент отправляет цель и может либо асинхронно дожидаться результата, либо подписаться на

поток обратной связи, либо в любой момент инициировать отмену цели. Важной особенностью является то, что сервер обязан корректно обрабатывать запрос на отмену: периодически проверять флаг `is_cancel_requested()` и при его установке прекращать выполнение, по возможности с частичным результатом или статусом прерывания. Состояние action (принята, выполняется, отменена, завершена успешно и т.д.) отслеживается на уровне middleware и доступно через инструменты командной строки `ros2 action`.

Типичный сценарий использования actions — навигация мобильного робота к заданной точке: цель содержит координаты и ориентацию, обратная связь включает оставшееся расстояние и текущую оценку позы, результат информирует об успехе или неудаче. Другие примеры: управление манипулятором по сложной траектории, длительная калибровка датчиков, выполнение последовательности манипуляций, загрузка больших данных с внешнего устройства.

Для реализации action-сервера в ROS 2 (`rclpy` для Python) используется класс `ActionServer`, наследуемый от `Node`. Обработчик цели реализуется как функция обратного вызова, принимающая объект цели и возвращающая результат по завершении. Публикация обратной связи осуществляется вызовом метода `publish_feedback` на объекте обратной связи сервера. Проверка запроса отмены выполняется через проверку свойства `is_cancel_requested` у обработчика цели. Клиент создается через `ActionClient`, отправка цели — асинхронный вызов `send_goal_async` с последующим ожиданием будущего объекта. Обратная связь может обрабатываться через колбэк, регистрируемый при отправке цели. Отмена цели выполняется вызовом метода `cancel_goal_async`.

Пример

Рассмотрим типовой сценарий: движение робота к целевой точке на плоскости с постоянной скоростью, с публикацией оставшегося расстояния и возможностью отмены.

Первым шагом создаётся рабочее пространство и пакет с поддержкой action. Выполняется инициализация workspace, в src размещается пакет, например, my_action_pkg, создаваемый с зависимостями rclpy, action_msgs и встроенным генератором action. В пакете создаётся каталог action, в нём файл MoveToGoal.action следующего содержания:

```
# Цель
float64 x
float64 y
float64 theta
---
# Результат
bool success
string message
---
# Обратная связь
float64 distance_remaining
float64 current_x
float64 current_y
```

В CMakeLists.txt (для C++) или setup.py/package.xml для Python пакета необходимо убедиться, что файл .action указан в процедуре генерации интерфейсов. Для Python-пакета достаточно наличия записи в package.xml `<buildtool_depend>roscpp</buildtool_depend>` и `<exec_depend>roscpp</exec_depend>`, а в setup.py — указание подпакета с action-файлами.

Далее реализуется action-сервер. Узел на Python наследуется от Node, создаёт экземпляр ActionServer, передавая тип действия (MoveToGoal) и имя действия, например, 'move_to_goal'. Обработчик цели моделирует перемещение с постоянной скоростью. В цикле с малым шагом по времени (0.1 с) вычисляется новое положение текущего агента как приближение к цели на фиксированное расстояние за итерацию (например, 0.05 единиц). На каждом шаге вычисляется оставшееся расстояние до цели, формируется сообщение обратной связи и публикуется. Также проверяется is_cancel_requested: если установлен, обработчик устанавливает результат с

флагом `success=False` и сообщением о прерывании и завершает цикл, дав серверу обработать отмену. По достижении расстояния менее порога (например, 0.01) цикл завершается с успешным результатом. Важно: при отмене сервер должен проинформировать клиента о прерванном статусе, возвращая результат с соответствующим сообщением.

Клиентская часть реализуется в отдельном узле. Создаётся `ActionClient`, ожидается подключение сервера через `wait_for_server`. Формируется цель с координатами `x=5.0`, `y=5.0`, `theta=0.0`. Цель отправляется методом `send_goal_async`, к будущему объекту добавляется колбэк для обработки ответа сервера о принятии цели. При успешном принятии регистрируется колбэк на получение обратной связи, который выводит в лог оставшееся расстояние и текущие координаты. Также асинхронно ожидается результат через `goal_handle.get_result_async` с выводом финального статуса. Дополнительно в клиенте можно инициировать отмену по истечении некоторого времени (например, через 3 секунды) вызовом `goal_handle.cancel_goal_async`, демонстрируя возможность досрочного прерывания.

Для запуска примера необходимо в одном терминале запустить сервер: `ros2 run my_action_pkg move_to_goal_server`, в другом — клиент: `ros2 run my_action_pkg move_to_goal_client`. Мониторинг активных `actions` выполняется командой `ros2 action list`. Информация о конкретном действии: `ros2 action info /move_to_goal`. Принудительная отправка цели из командной строки: `ros2 action send_goal /move_to_goal my_action_pkg/action/MoveToGoal "{x: 5.0, y: 5.0, theta: 0.0}" --feedback`. Эта команда показывает поток обратной связи в реальном времени.

Демонстрация отмены: во время выполнения цели можно в другом терминале выполнить `ros2 action send_goal` с ключом `--cancel-on-timeout` и указанием небольшого таймаута, или через клиентский код с вызовом

cancel_goal_async. Сервер корректно завершит цикл и вернёт результат с success=false и пояснением.

Таким образом, студент осваивает полный цикл работы с actions: описание интерфейса, серверную логику с фидбеком и обработкой отмены, клиентское взаимодействие и диагностические утилиты.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Создайте action-интерфейс для движения робота по квадратной траектории со стороной, передаваемой в цели. Обратная связь должна содержать пройденный процент пути и текущие координаты. Сервер моделирует движение по сторонам квадрата с постоянной скоростью, периодически публикуя процент и координаты. Реализуйте клиент, отправляющий цель со стороной 4.0, выводящий обратную связь и предоставляющий возможность отмены по нажатию Ctrl+C или по таймауту 5 секунд. Продемонстрируйте мониторинг через ros2 action.
2	Разработайте action для постепенного нагрева объекта до заданной температуры. Цель: target_temperature (float64).

	Обратная связь: <code>current_temperature</code>, <code>time_elapsed</code>. Результат: <code>success</code>, <code>message</code>. Сервер моделирует нагрев с шагом +5 градусов в секунду, начиная от 20°C. Каждые 0.2 секунды публикуется текущая температура. При отмене нагрев немедленно прекращается, возвращается достигнутая температура в <code>message</code>. Клиент отправляет цель 100°C, печатает фидбек и позволяет отменить нагрев досрочно (например, через 3 секунды). Проверьте отмену.
3	Реализуйте action для зарядки батареи до заданного уровня. Цель: <code>target_level</code> (проценты от 0 до 100). Обратная связь: <code>current_level</code>, <code>estimated_time_remaining</code>. Результат: <code>success</code>, <code>final_level</code>. Сервер имитирует зарядку со скоростью 10% в секунду, обновляя фидбек каждые 0.5 с. При отмене зарядка останавливается, результат содержит достигнутый уровень. Клиент отправляет цель 80%, подписывается на обратную связь, отменяет через 4 секунды. Продемонстрируйте <code>ros2 action send_goal</code> с опцией <code>--feedback</code>.
4	Создайте action для сканирования окружения лидаром с подсчётом обнаруженных объектов. Цель: <code>scan_duration</code> (в секундах). Обратная связь: <code>elapsed_time</code>, <code>objects_detected_so_far</code>. Результат: <code>total_objects</code>, <code>success</code>. Сервер каждую секунду случайным образом добавляет от 0 до 3 объектов, публикует текущие показатели. При отмене возвращается количество объектов, обнаруженных до отмены. Клиент запрашивает сканирование длительностью 10 секунд, каждую секунду выводит обратную связь и по истечении 5 секунд инициирует отмену. Проверьте список actions и детализацию.
5	Реализуйте action для медленной сортировки массива целых

	чисел с визуализацией прогресса. Цель: <code>unsorted_array</code> (массив <code>int32</code>). Обратная связь: <code>progress</code> (процент завершённых сравнений), <code>partially_sorted_array</code>. Результат: <code>sorted_array</code>, <code>success</code>. Сервер использует пузырьковую сортировку с искусственной задержкой 0.1 с на каждой итерации внутреннего цикла, публикуя прогресс каждые 5% завершения. При отмене сортировка прерывается, результат содержит массив в текущем состоянии с <code>success=false</code>. Клиент отправляет массив из 20 случайных чисел, отображает обратную связь и отменяет после 3 секунд. Отследите взаимодействие <code>ros2 action info</code>.
6	Создайте action для перемещения руки манипулятора по трёхточечной траектории. Цель: три точки (x,y,z) — <code>point1</code>, <code>point2</code>, <code>point3</code>. Обратная связь: <code>current_point_index</code>, <code>completion_percentage</code>. Результат: <code>success</code>, <code>message</code>. Сервер моделирует движение от точки к точке с постоянной скоростью, между точками проходит 2 секунды, публикуя каждые 0.5 с текущий индекс точки и процент внутри отрезка. При отмене движение прекращается. Клиент задаёт три произвольные точки, подписывается на фидбек и отменяет цель через 3 секунды. Проверьте отображение обратной связи через <code>ros2 action send_goal</code> с флагом <code>--feedback</code>.
7	Разработайте action для эмуляции загрузки файла по сети с докачкой. Цель: <code>file_size_mb</code> (<code>float64</code>). Обратная связь: <code>downloaded_mb</code>, <code>speed_mbps</code>. Результат: <code>success</code>, <code>total_downloaded</code>. Сервер каждые 0.25 секунды увеличивает объём загруженных данных на случайную величину (0.5–2 МБ), вычисляет скорость. При отмене загрузка останавливается, результат содержит фактический объём. Клиент отправляет цель 100 МБ, выводит

	фидбек, через 4 секунды отменяет. Проверьте состояние активных actions.
8	Создайте action для симуляции печати документа. Цель: pages_count (int32). Обратная связь: printed_pages, pages_per_second. Результат: success, message. Сервер моделирует печать со скоростью 3 страницы в секунду, публикует обновления каждые 0.5 с. При отмене печать прерывается. Клиент отправляет цель на 50 страниц, показывает обратную связь, отменяет через 5 секунд. Продемонстрируйте мониторинг через ros2 action list и info.
9	Реализуйте action для движения робота по спирали Архимеда с заданными параметрами. Цель: max_radius, revolutions. Обратная связь: current_radius, current_angle, percentage. Результат: success. Сервер вычисляет точки спирали с шагом 0.1 рад, перемещается по ним с задержкой 0.1 с, публикуя текущие параметры каждые 0.5 с. Отмена останавливает движение. Клиент задаёт max_radius=5.0, revolutions=3, выводит фидбек и отменяет через 4 секунды.
10	Разработайте action для плавного изменения цвета RGB-светодиода от начального к целевому за заданное время. Цель: target_r, target_g, target_b, duration_sec. Обратная связь: current_r, current_g, current_b, progress. Результат: success. Сервер линейно интерполирует значения, обновляя фидбек 10 раз в секунду. При отмене светодиод остаётся в текущем цвете. Клиент отправляет цель перехода от (255,0,0) к (0,255,0) за 5 секунд, выводит фидбек и отменяет через 2 секунды.
11	Создайте action для обхода препятствий: робот движется вперёд и при обнаружении препятствия (случайная задержка 2–4

	секунды) объезжает его. Цель: <code>total_distance</code>. Обратная связь: <code>distance_travelled</code>, <code>obstacles_encountered</code>. Результат: <code>success</code>, <code>message</code>. Сервер каждые 0.2 секунды продвигается на 0.1 м; раз в 1–3 секунды случайно генерирует препятствие, задерживая продвижение на 1 с. Публикует пройденное расстояние и число препятствий. При отмене движение прекращается. Клиент отправляет цель 10 м, отображает фидбек и отменяет через 4 с.
12	Реализуйте action для последовательного посещения нескольких опорных точек с подтверждением достижения каждой. Цель: массив точек (x,y). Обратная связь: <code>current_point_index</code>, <code>remaining_points</code>. Результат: <code>success</code>, <code>visited_all</code>. Сервер моделирует прибытие в очередную точку через 1.5 с после начала движения к ней, публикует индекс после каждого достижения. Отмена прерывает обход. Клиент задаёт 5 точек, подписывается на фидбек, через 4 с отменяет цель.
13	Разработайте action для циклического выполнения операции с возможностью досрочного выхода. Цель: <code>cycle_count</code>. Обратная связь: <code>completed_cycles</code>, <code>current_cycle_progress</code>. Результат: <code>total_completed</code>, <code>success</code>. Сервер внутри каждого цикла выполняет 5 шагов с задержкой 0.2 с, публикуя прогресс. При отмене завершает текущий цикл и возвращает количество полностью выполненных циклов. Клиент запрашивает 10 циклов, отменяет через 3 с, наблюдает фидбек.
14	Создайте action для вычисления числа Фибоначчи с большой глубиной рекурсии и длительными шагами. Цель: <code>index</code> (номер). Обратная связь: <code>current_index</code>, <code>current_value</code>, <code>estimated_time_left</code>. Результат: <code>fibonacci_number</code>, <code>success</code>. Сервер вычисляет числа Фибоначчи последовательно с задержкой 0.1 с на каждое число,

	публикуя прогресс каждые 5 индексов. Отмена прерывает вычисление, результат содержит последнее вычисленное число. Клиент запрашивает 50-е число, показывает фидбек и отменяет через 3 с.
15	Реализуйте action для медленной калибровки сенсора, состоящей из нескольких этапов. Цель: sensor_id (строка). Обратная связь: stage (строка), progress_in_stage. Результат: calibration_success, offset_value. Сервер проходит три этапа: "инициализация", "сбор данных", "расчёт смещения", каждый длится 2 с, публикуя название этапа и процент внутри этапа каждые 0.5 с. Отмена на любом этапе прерывает процесс, результат содержит success=false. Клиент запускает калибровку сенсора "front_lidar", подписывается на обратную связь и отменяет через 4 с.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое Actions и чем они отличаются от сервисов?
2. Из каких частей состоит файл .action?
3. Как реализовать сервер действия на C++?
4. Как клиент отслеживает прогресс выполнения?
5. Что такое feedback и result в Action?
6. Как отменить выполняемое действие?

7. Когда Action предпочтительнее сервиса?
8. Как проверить выполнение (`ros2 action send_goal`)?

Лабораторная работа 9.

РАБОТА С СИМУЛИРОВАННЫМ ЛИДАРОМ В GAZEBO

Цель и содержание работы: Освоить запуск симулированного робота с лидаром в среде Gazebo под управлением ROS2, научиться получать и анализировать данные лазерного сканирования из топика `/scan`, визуализировать облака точек в RViz и разрабатывать программные узлы для обработки лидарных данных. По итогам выполнения работы студент должен демонстрировать умение взаимодействовать с симулированным сенсором и применять полученные сканы в простейших алгоритмах.

Теоретическое обоснование

Лидар (Light Detection and Ranging) представляет собой дальномер, формирующий массив измерений в пределах некоторого углового сектора. В симуляторе Gazebo модель лидара публикует данные в топик стандартного типа `sensor_msgs/msg/LaserScan`, определённый в пакете `sensor_msgs`. Структура сообщения содержит: временную метку `header`, углы начала и конца сканирования `angle_min`, `angle_max`, шаг угла `angle_increment`, время между двумя последовательными замерами `time_increment`, время полного оборота `scan_time`, минимальную и максимальную измеряемую дальность `range_min`, `range_max`, а также массив дальностей `ranges` и массив интенсивностей `intensities`. Дальности упорядочены от начального угла к конечному, пропущенные или ошибочные измерения могут принимать значение `inf`.

Готовый проект TurtleBot3 предоставляет настроенный мир с моделью робота, оснащённой лидаром, публикующим сканы в топик `/scan`. Среда RViz позволяет подписываться на этот топик и отображать облака точек, выбирая соответствующую визуализацию `LaserScan`. Программный доступ к данным реализуется через подписчика ROS2, написанного на Python (`rclpy`) или C++

(rcldcpp). Подписчик получает объект сообщения LaserScan и может обращаться к его полям для извлечения дальностей или интенсивностей. Наиболее типовой сценарий обработки включает выделение минимального, максимального или среднего расстояния, фильтрацию значений по порогу, конвертацию полярных координат в декартовы или сохранение сканов в файл.

Пример

Рассмотрим типовой сценарий получения, визуализации и обработки данных симулированного лидара. В качестве предварительного условия предполагается, что установлены ROS2 Humble и необходимые пакеты TurtleBot3, а окружение настроено командами `source /opt/ros/humble/setup.bash` и `export TURTLEBOT3_MODEL=burger`.

Запуск симуляции выполняется командой:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

После успешного старта в терминале наблюдаются сообщения о готовности мира и робота.

Для просмотра содержимого топика `/scan` в реальном времени используется:

```
ros2 topic echo /scan
```

Терминал отображает последовательные сообщения с полями `ranges`, интенсивностей и временных меток.

Для графической визуализации в отдельном терминале запускается RViz:

```
ros2 run rviz2 rviz2
```

В интерфейсе добавляется дисплей типа `LaserScan`, в качестве топика указывается `/scan`, задаётся фиксированный фрейм (обычно `base_scan` или `odom`). После нажатия `Apply` в окне появляются цветные точки, соответствующие текущему скану.

Написание простейшего подписчика выполняется на Python. Создаётся пакет, например `lidar_listener`, внутри которого размещается скрипт `scan_subscriber.py`. Код узла инициализирует `rclpy`, создаёт объект подписчика на топик `/scan` с типом сообщения `LaserScan` и в коллбеке извлекает массив дальностей. Типовой пример коллбека выводит количество лучей и минимальную измеренную дальность, игнорируя бесконечные значения:

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

class ScanSubscriber(Node):
    def __init__(self):
        super().__init__('scan_subscriber')
        self.subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.scan_callback,
            10
        )

    def scan_callback(self, msg):
        ranges = [r for r in msg.ranges if r > msg.range_min and r <
msg.range_max]
        count = len(ranges)
        if count > 0:
            min_range = min(ranges)
            self.get_logger().info(f'Получено {count} валидных лучей, минимум:
{min_range:.2f} м')
        else:
            self.get_logger().warn('Нет валидных измерений в скане')

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

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

После сборки пакета и его запуска командой `ros2 run lidar_listener scan_subscriber` терминал показывает периодические сообщения с минимальной дальностью. Данный пример демонстрирует базовое взаимодействие с лидарными данными и служит основой для выполнения вариантных заданий.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Создайте узел, подписывающийся на топик /scan, который при получении каждого скана выводит в лог общее количество лучей с конечными значениями дальности и максимальное измеренное расстояние.
2	Напишите подписчика, вычисляющего среднее арифметическое всех валидных дальностей скана. Узел должен выводить среднее значение с точностью до двух знаков после запятой.
3	Разработайте узел, который отфильтровывает дальности, превышающие заданный порог (например, 3.0 м), и публикует новый массив отфильтрованных дальностей в собственный топик типа sensor_msgs/LaserScan, сохраняя остальные параметры неизменными.
4	Создайте подписчика, преобразующего полярные координаты скана (угол, дальность) в декартовы координаты (x, y) и записывающего их в текстовый файл, по одному скану за строку

	в формате «угол дальность x y».
5	Реализуйте узел, подсчитывающий количество лучей, попадающих в заданный угловой сектор (например, от -30 до 30 градусов относительно оси X робота). Выводите это число в лог при каждом скане.
6	Напишите подписчика, который отслеживает минимальную дальность среди лучей, направленных в секторе, соответствующем задней полусфере робота (углы от 135 до 225 градусов).
7	Создайте узел, публикующий в топик /closest_point сообщение типа geometry_msgs/PointStamped с координатами ближайшей к роботу точки из текущего скана в системе координат лидара.
8	Напишите узел, вычисляющий временной интервал между двумя последовательными сканами, используя временные метки из заголовка сообщения, и выводящий его в миллисекундах.
9	Разработайте подписчика, ведущего журнал всех сканов: сохраняйте массив дальностей и временную метку в CSV-файл, дополняя его с каждым новым сообщением.
10	Реализуйте узел, который, получив первый скан, запоминает его в качестве эталона и для каждого последующего выводит количество лучей, чьё абсолютное изменение дальности по сравнению с эталонным превышает 0.2 м.
11	Напишите подписчика, публикующего в топик /scan_marker визуальную маркерную линию (Marker) типа LINE_STRIP, представляющую контур скана в декартовых координатах, для визуализации в RViz.
12	Создайте узел, который на основании данных скана определяет наличие препятствия в полосе шириной 0.5 м непосредственно

	перед роботом (от -0.25 до 0.25 м по оси Y в системе координат лидара) и публикует в топик /obstacle_warning сообщение типа Bool (True – препятствие есть).
13	Напишите подписчика, отображающего в лог угловое положение луча с минимальной дальностью относительно оси X робота.
14	Разработайте узел, который принимает параметр angular_range (в градусах) и отображает количество лучей, попадающих в этот угловой диапазон, равномерно отложенный от направления вперёд. При отсутствии параметра используется значение по умолчанию 60 градусов.
15	Создайте узел-подписчик, который при получении каждого десятого скана печатает среднюю дальность только по лучам, интенсивность которых превышает 50% от максимальной интенсивности в данном скане.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое лидар и какое сообщение ROS 2 используется для его данных?
2. Как запустить симуляцию в Gazebo и подключить ROS 2?
3. Что такое LaserScan и каковы его ключевые поля?
4. Как установить плагин симулируемого лидара?

5. Что такое TF и зачем он нужен для сенсоров?
6. Как визуализировать данные лидара в RViz?
7. Каковы типичные параметры симулируемого лидара?
8. Как проверить, что данные лидара публикуются?

Лабораторная работа 10.

ВИЗУАЛИЗАЦИЯ СЕНСОРОВ В RVIZ

Цель и содержание работы: Освоение базовых принципов визуализации данных сенсоров и навигационного состояния автономного мобильного робота в среде RViz2. Приобретение практических навыков запуска RViz, добавления и настройки типовых дисплеев (LaserScan, Odometry, Path, Map), управления системами координат (frame_id, fixed frame), а также сохранения конфигурации для повторного использования.

Теоретическое обоснование

RViz2 является штатным инструментом визуализации в экосистеме ROS 2. Он получает данные через топики и отображает их в трёхмерном пространстве, опираясь на информацию о трансформациях из tf2. Каждый визуализируемый элемент представляет собой дисплей, сконфигурированный на определённый топик и имеющий набор параметров отрисовки.

Важнейшим понятием является fixed frame — неподвижная система координат, относительно которой позиционируются все объекты сцены. Обычно в качестве fixed frame выбирают map или odom для глобального представления, либо base_link для отображения данных в системе координат робота. Параметр frame_id в каждом дисплее указывает, в какой системе координат интерпретируются поступающие сообщения.

Дисплей LaserScan предназначен для визуализации данных лидара, публикуемых в сообщениях sensor_msgs/msg/LaserScan. Он отрисовывает лучи, точки или облако точек в зависимости от настроек стиля. Дисплей Odometry визуализирует оценку положения робота на основе сообщений nav_msgs/msg/Odometry, отображая текущую позу и ковариационную эллипсу. Дисплей Path строит траекторию из последовательности поз, полученных из сообщений nav_msgs/msg/Path. Дисплей Map отображает

карту занятости, передаваемую в сообщениях `nav_msgs/msg/OccupancyGrid`, в виде цветного растрового изображения.

Для корректной работы всех дисплеев необходимо, чтобы дерево `tf2` содержало преобразования между используемыми системами координат. Перед запуском `RViz` следует убедиться, что запущены все узлы, публикующие необходимые данные и трансформации.

Сохранение конфигурации `RViz` выполняется в файл с расширением `.rviz`. В этом файле хранятся настройки всех добавленных дисплеев, параметры видов, цветовые схемы и прочие элементы интерфейса. Сохранённая конфигурация может быть загружена при последующих запусках, что позволяет быстро восстановить рабочее окружение.

Пример

Рассмотрим типовой сценарий настройки `RViz` для визуализации состояния автономной системы. Для выполнения задания потребуется заранее запущенная симуляция или реальный робот, публикующий данные лидара, одометрии, путь и карту.

Первым шагом осуществляется запуск `RViz`. В терминале выполняется команда:

```
rviz2
```

После открытия главного окна в левой панели дисплеев нажимается кнопка `Add`. В открывшемся диалоговом окне последовательно добавляются дисплеи `By topic` или `By display type`.

Для добавления `LaserScan` в разделе `By display type` выбирается `rviz_default_plugins/LaserScan`, задаётся имя дисплея, указывается топик, например `/scan`. В поле `Reliability Policy` устанавливается значение `System Default` или `Best Effort` в зависимости от издателя. В разделе `Style` выбирается пункт `Boxes`, `Points` или `Lines`. Параметр `Size` задаётся `0.05` м, цвет

подбирается по предпочтению. В поле Frame Id вводится система координат лидара, например laser_frame.

Дисплей Odometry добавляется выбором типа rviz_default_plugins/Odometry. Указывается топик одометрии, обычно /odom. Параметр Keep задаёт количество отображаемых последних положений, например 100. Опция Shape определяет форму стрелки, а настройки цвета и альфа-канала позволяют выделить траекторию одометрии. Frame Id устанавливается в odom.

Дисплей Path добавляется как rviz_default_plugins/Path. Топик выбирается из доступных, к примеру /plan или /trajectory. На вкладке Style можно задать цвет линии, тип линии (сплошная, пунктирная), ширину. Frame Id должен совпадать с системой координат, в которой публикуется путь, обычно map.

Дисплей Map добавляется типом rviz_default_plugins/Map. Топик задаётся /map, Topic QoS подбирается под издатель карты, обычно Durability: Transient Local. Параметр Alpha определяет прозрачность. Палитра map определяет отрисовку occupied/free/unknown. Frame Id задаётся map.

После добавления дисплеев в разделе Global Options правой панели устанавливается Fixed Frame в map (или odom, если карта публикуется в odom). Проверяется, что все дисплеи корректно получают данные. При отсутствии сообщений строке статуса дисплея отображается предупреждение.

Завершающим шагом выполняется сохранение конфигурации: File → Save Config As, указывается имя файла, например sensor_viz.rviz. В дальнейшем данную конфигурацию можно загрузить командой:

```
rviz2 -d sensor_viz.rviz
```

Таким образом, получается настроенная визуализация, включающая лидар, одометрию, запланированный путь и карту.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите RViz. Добавьте дисплей LaserScan, настроив его на топик /scan. Установите Fixed Frame в base_link. Измените стиль отрисовки на Points, размер точки 0.02, цвет зелёный. Сохраните конфигурацию под именем lidar_base.rviz.
2	Запустите RViz. Добавьте дисплей Odometry на топик /odom, количество сохраняемых положений Keep = 200, форму Arrow. Добавьте дисплей LaserScan на топик /scan, стиль Lines, цвет красный. Fixed Frame задайте odom. Сохраните конфигурацию как odom_lidar.rviz.
3	Запустите RViz. Добавьте дисплей Map на топик /map с альфа-каналом 0.7 и схемой map. Установите Fixed Frame в map. Добавьте дисплей Path на топик /trajectory, цвет линии жёлтый, ширина 0.05. Сохраните конфигурацию как map_traj.rviz.
4	Запустите RViz. Добавьте три дисплея: LaserScan на /scan, Odometry на /odom, Map на /map. Для LaserScan выберите стиль Boxes (Size 0.1), для Odometry форму Cylinder, цвет голубой. Fixed Frame = map. Сохраните конфигурацию как

	multi_display.rviz.
5	Запустите RViz. Добавьте дисплей Path на топик /plan, тип линии Dash, ширина 0.03. Добавьте LaserScan на /scan_filtered (если топик недоступен, эмулируйте /scan), стиль Spheres, цвет фиолетовый. Fixed Frame = odom. Сохраните как path_lidar.rviz.
6	Запустите RViz. Добавьте дисплей Map с топином /occupancy_grid, палитра costmap. Добавьте Odometry на /odometry/global, Keep = 50. Fixed Frame = map. Добавьте второй дисплей LaserScan на топик /scan_back, стиль Points, размер 0.03, оранжевый. Сохраните как global_view.rviz.
7	Запустите RViz. Настройте визуализацию, используя два дисплея LaserScan: один на /scan_front (цвет синий, стиль Boxes), второй на /scan_rear (цвет красный, стиль Lines). Fixed Frame = base_link. Сохраните как dual_lidar.rviz.
8	Запустите RViz. Добавьте дисплей Odometry на топик /wheel_odom, Shape = Arrow, Alpha = 0.8. Добавьте дисплей Map на топик /rtk_map. Настройте палитру raw. Fixed Frame = map. Сохраните как wheel_rtk.rviz.
9	Запустите RViz. Добавьте Path на топик /recorded_path, цвет белый, ширина 0.02. Добавьте LaserScan на /scan, стиль Points с цветом морской волны. Fixed Frame = odom. Сохраните как recorded_path.rviz.
10	Запустите RViz. Добавьте Map на топик /static_map, Alpha = 1.0, palette = map. Добавьте Odometry на /ekf_odom, Keep = 0 (показывать только последнюю позу). Fixed Frame = map. Сохраните как static_map_ekf.rviz.
11	Запустите RViz. Добавьте дисплей LaserScan на /scan, стиль Lines, Decay Time = 2.0. Добавьте дисплей Path на

	/move_base/GlobalPlanner/plan. Цвет пути — magenta, ширина 0.04. Fixed Frame = map. Сохраните как planned_scan.rviz.
12	Запустите RViz. Добавьте Odometry на /odom, Shape = Axes, длина осей 0.2. Добавьте Map на /map. Настройте карту с альфа-каналом 0.5. Fixed Frame = map. Добавьте LaserScan на /scan_rear, стиль Boxes, размер 0.08, цвет серый. Сохраните как axes_scan.rviz.
13	Запустите RViz. Используйте дисплей Map с топиком /floor_map, измените цветовую схему на «costmap» с инвертированием. Добавьте LaserScan на /scan, стиль Points, размер 0.01, цвет жёлтый. Fixed Frame = base_link. Сохраните как floor_local.rviz.
14	Запустите RViz. Добавьте три дисплея: Path на /local_path (зелёный, пунктир), Path на /global_path (синий, сплошной), LaserScan на /scan (красные точки). Fixed Frame = map. Сохраните как dual_path.rviz.
15	Запустите RViz. Добавьте Map на /map с альфа-каналом 0.9. Добавьте Odometry на /odom, отображение ковариационной эллипсы (Covariance = true), цвет эллипсы фиолетовый. Добавьте LaserScan на /scan, стиль Spheres, Decay Time = 1.0. Fixed Frame = map. Сохраните как full_state.rviz.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое RViz и для чего он используется?
2. Как добавить визуализацию топики в RViz?
3. Какие типы визуализаций поддерживает RViz?
4. Что такое Fixed Frame и как его выбрать?
5. Как сохранить конфигурацию RViz?
6. Что такое MarkerArray?
7. Как визуализировать систему координат TF?
8. Как настроить отображение сенсора через Display панель?

Лабораторная работа 11.

ОДОМЕТРИЯ И ЕЁ ВИЗУАЛИЗАЦИЯ

Цель и содержание работы: Получить понимание принципов одометрии в ROS2, научиться подписываться на топик /odom, извлекать данные о положении и ориентации робота, вычислять пройденный путь на основе одометрических данных, а также визуализировать траекторию движения в RViz с использованием инструментов Arrow, RobotModel и Path.

Теоретическое обоснование

Одометрия — метод оценки изменения положения робота во времени на основе данных датчиков вращения колёс (энкодеров) или других сенсоров. В ROS2 одометрическая информация передаётся через сообщения типа nav_msgs/msg/Odometry. Это сообщение содержит:

- pose.pose.position (x, y, z) — текущее положение робота в системе координат odom;
- pose.pose.orientation (кватернион) — ориентацию;
- twist.twist.linear и angular — оценки линейной и угловой скоростей;
- массивы ковариаций для положения и скорости.

Топик /odom публикуется драйвером робота или симуляцией (например, gazebo). Преобразование координат (tf) связывает фрейм odom с фреймом base_footprint или base_link. Для визуализации положения робота в RViz используются дисплеи RobotModel (отображение 3D-модели) и Arrow (отображение ориентации). Для отображения траектории можно использовать сообщения nav_msgs/msg/Path или визуализацию через Marker.

Вычисление пройденного пути основывается на интегрировании расстояния между последовательными положениями. Если получены позиции (x_i, y_i) в моменты времени t_i , то путь $S = \sum \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$.

$(y_i - y_{i-1})^2$). Для точности необходимо учитывать частоту обновления одометрии.

Пример

Рассмотрим типовой сценарий: запуск симуляции TurtleBot3, создание узла для подписки на /odom, вычисление пройденного пути и визуализация траектории в RViz.

Необходимые действия:

1. Установка и запуск симуляции TurtleBot3 в ROS2

```
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

2. Создание нового пакета для узла:

```
ros2 pkg create --build-type ament_python lab11_odometry
cd lab11_odometry/lab11_odometry
Создать файл odometry_logger.py
```

3. Содержание узла (пример на Python):

```
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point
from nav_msgs.msg import Path
import math
```

```
class OdometryPathNode(Node):
    def __init__(self):
        super().__init__('odometry_path_node')
        self.subscription = self.create_subscription(Odometry, '/odom',
self.odom_callback, 10)
        self.path_pub = self.create_publisher(Path, '/trajectory', 10)
        self.total_distance = 0.0
        self.prev_x = None
        self.prev_y = None
        self.path = Path()
        self.path.header.frame_id = 'odom'

    def odom_callback(self, msg):
        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y
        if self.prev_x is not None:
            dx = x - self.prev_x
            dy = y - self.prev_y
            distance = math.sqrt(dx*dx + dy*dy)
            self.total_distance += distance
            self.get_logger().info(f'Пройдено: {self.total_distance:.3f} м')
            # Добавляем точку в траекторию
            p = Point()
            p.x = x
            p.y = y
```

```

        p.z = 0.0
        self.path.poses.append(self.create_pose_stamped(p,
msg.header.stamp))
        self.path.header.stamp = msg.header.stamp
        self.path_pub.publish(self.path)
        self.prev_x = x
        self.prev_y = y

    def create_pose_stamped(self, point, stamp):
        from geometry_msgs.msg import PoseStamped
        ps = PoseStamped()
        ps.header.stamp = stamp
        ps.header.frame_id = 'odom'
        ps.pose.position = point
        ps.pose.orientation.w = 1.0
        return ps

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

```

4. Настройка RViz для визуализации:

Запустить RViz: `ros2 run rviz2 rviz2`

Добавить дисплей RobotModel (выбрать описание TurtleBot3).

Добавить дисплей Path, в поле Topic указать /trajectory, Colour задать синий.

Добавить дисплей Odometry (для отображения стрелок), указать топик /odom.

Установить фиксированный фрейм odom.

5. Запуск узла:

```
ros2 run lab11_odometry odometry_logger
```

В результате в консоли будет выводиться накопленный пройденный путь, а в RViz отобразится синяя линия пройденной траектории. При движении робота (например, с помощью teleop) траектория обновляется.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;

- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Написать узел ROS2, подписывающийся на топик /odom. Робот движется прямолинейно вперёд на 5 метров (использовать команду /cmd_vel). Узел должен вычислять пройденный путь каждую секунду и выводить значение в консоль. В RViz настроить отображение траектории в виде линии (Path). Результат работы представить в виде скриншота RViz с траекторией и лога терминала.
2	Создать узел для вычисления пройденного пути при движении робота по квадрату со стороной 2 метра. После завершения движения (робот возвращается в начальную точку) узел выводит общий пройденный путь. Дополнительно опубликовать топик /square_path типа Path и визуализировать его в RViz зелёными маркерами.
3	Разработать узел, который при движении робота по окружности радиусом 1 метр (полный оборот) вычисляет пройденный путь и публикует его в топик /distance с периодичностью 0.5 секунды. В RViz добавить стрелку Arrow, отображающую текущее направление движения (подписаться на /odom и преобразовывать ориентацию в вектор).
4	Реализовать узел для траектории в виде восьмёрки. Узел должен

	сохранять все точки одометрии в текстовый файл trajectory.txt (формат: x y timestamp). Также узел вычисляет полный пройденный путь и записывает его в конец файла. В RViz отобразить траекторию с помощью Path.
5	Написать узел, который работает в режиме случайного движения (робот каждые 2 секунды меняет линейную и угловую скорость). Узел вычисляет пройденный путь и среднюю линейную скорость за всё время движения. В RViz отобразить текущее положение робота стрелкой и пройденный путь в виде текстовой метки (использовать дисплей TexturedMarker или выводить в консоль).
6	Создать узел для движения по спирали (увеличение радиуса). Узел должен предоставлять сервис reset_odometry, который сбрасывает счётчик пройденного пути в ноль. После сброса визуализация траектории начинается заново. Пройденный путь выводится в консоль каждые 2 метра.
7	Разработать узел, обрабатывающий движение вперед-назад: робот проезжает 1 метр вперед, затем 1 метр назад. Узел вычисляет суммарный абсолютный пройденный путь и чистый путь (смещение от начала). Оба значения выводятся по завершении. В RViz отобразить конечное положение робота маркером сферы.
8	Создать узел для движения по равностороннему треугольнику (сторона 1.5 метра). Узел строит график зависимости пройденного пути от времени и сохраняет его в файл distance_time.csv. В RViz показать траекторию, раскрашивая участки разными цветами (например, каждая сторона своим цветом через отдельные сообщения Marker).
9	Написать узел, который используется совместно с teleop (ручное

	управление). При каждой полной остановке робота (скорость близка к нулю) узел выводит в консоль пройденный путь с момента последней остановки. В RViz добавить текстовую панель (используя Marker с типом TEXT_VIEW_FACING) для отображения текущего пройденного пути.
10	Реализовать узел для движения по кривой Безье с тремя контрольными точками. Узел вычисляет пройденный путь и определяет момент, когда робот проходит ровно половину этого пути. При достижении половины пути узел публикует сообщение в топик /half_distance_reached. В RViz визуализировать всю траекторию и выделить точку половины маркером.
11	Разработать узел, обрабатывающий повороты на месте: робот совершает последовательно повороты на 90, 180, 90 и -180 градусов (суммарный угол). Узел вычисляет пройденный путь (должен быть близок к нулю) и суммарный угол рыскания, накопленный из ориентации одометрии. Результаты вывести в консоль. В RViz отобразить изменение ориентации стрелкой.
12	Создать узел для объезда препятствия по простому маршруту: вперёд 2 метра, вправо 1 метр, вперёд 2 метра. Узел вычисляет фактический пройденный путь и сравнивает его с прямым расстоянием от начала до конца (по теореме Пифагора). Выводит разницу. В RViz отобразить фактическую траекторию (синий Path) и прямую линию (красный Path).
13	Написать узел для движения по синусоиде (амплитуда 0.5 метра, период 2 метра). Узел вычисляет пройденный путь и среднюю скорость, выводя их в консоль каждые 0.5 секунды. В RViz добавить метки (маленькие сферы) в точках, где пройденный путь кратен 0.5 метра.

14	Реализовать узел для комбинированного движения: сначала робот движется по окружности (полный круг радиусом 0.8 м), затем по квадрату (сторона 1 м). Узел вычисляет пройденный путь отдельно для каждого этапа и общий путь. Переключение этапов осуществляется по достижении заданного пути (например, после 5 метров). В RViz отобразить траекторию разными цветами (круг — красный, квадрат — синий).
15	Создать узел, который симулирует сбой одометрии: в процессе движения (прямолинейно со скоростью 0.2 м/с) в одометрию искусственно вносится случайный выброс координат (смещение на 0.5 метра в случайном направлении). Узел должен обнаруживать выбросы по аномально большому приращению расстояния (порог > 0.3 м за один шаг) и фильтровать такие скачки, не учитывая их в пройденном пути. В RViz отобразить отфильтрованную траекторию без выбросов. Допускается модификация симуляции или использование отдельного узла для подмены данных /odom.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое одометрия и какое сообщение для неё используется?
2. Какие источники используются для получения одометрии?

3. Как накапливаются ошибки одометрии со временем?
4. Как визуализировать траекторию по одометрии в RViz?
5. Что такое `frame_id` и `child_frame_id` в сообщении `Odometry`?
6. Как публиковать трансформацию TF из одометрии?
7. Что такое `Path` и как его использовать для отображения маршрута?
8. Почему одна одометрия недостаточна для навигации?

Лабораторная работа 12.

СТАТИЧЕСКАЯ КАРТА (OCCUPANCY GRID)

Цель и содержание работы: Изучить способ представления окружения в виде плоской сетки занятости (Occupancy Grid) в ROS2, освоить запуск и настройку узла `map_server` для загрузки статической карты из графического файла с YAML-описанием, получить навыки проверки публикуемых данных через топик `/map`, визуализации карты в RViz и сопоставления карты с текущим положением робота.

Теоретическое обоснование

Карта занятости (Occupancy Grid) — это широко распространённый в мобильной робототехнике способ представления двумерной среды. Пространство разбивается на регулярную сетку ячеек, каждая из которых хранит оценку вероятности того, что данная область занята препятствием. В ROS2 такая карта передаётся сообщением типа `nav_msgs/msg/OccupancyGrid`. Поле `data` содержит линейный массив целых чисел со знаком, где значение `-1` означает неизвестную ячейку, `0` — свободное пространство, а положительные числа в диапазоне от 1 до 100 показывают степень уверенности в занятости (100 — полностью занято). Метаданные карты (ширина, высота, разрешение в метрах на ячейку, точка отсчёта и система координат) хранятся в заголовке сообщения и полях `info`.

Для загрузки готовой статической карты в ROS2 применяется узел `map_server` из состава Navigation2 (пакет `nav2_map_server`). Входными данными служат два файла: изображение в формате PGM (Portable Gray Map), каналы которого интерпретируются как занятость/свобода, и YAML-файл с параметрами карты. Типовой YAML-файл содержит ключи:

- `image` — путь к файлу изображения (относительный или абсолютный);

- resolution — разрешение карты, м/пиксель;
- origin — положение левого нижнего угла карты в глобальной системе координат, задаётся как [x, y, yaw];
- negate — флаг инверсии значений занятости (белое становится чёрным);
- occupied_thresh — порог, выше которого пиксель считается занятым (по умолчанию 0.65);
- free_thresh — порог, ниже которого пиксель считается свободным (по умолчанию 0.196);
- mode — режим интерпретации данных: trinary (значения -1/0/100), scale (масштабирует яркость в диапазон 0..100) или raw (передает исходные значения 0..255 как есть, где 0 — полностью занято, 255 — свободно).

Узел `map_server` после запуска публикует однократно (latched-публикация) или периодически тему `/map` с типом `OccupancyGrid` и, опционально, предоставляет сервисы для получения карты. Просмотр содержимого топика выполняется командой `ros2 topic echo /map`. Для графической визуализации используется `RViz2`, в котором добавляется дисплей типа `Map`, подписанный на топик `/map`, и выставляется фиксированная система координат, совпадающая с `frame_id` карты (обычно “map”). Сопоставление карты с положением робота требует наличия TF-преобразований от системы координат робота к карте; в простейшем случае без работающей локализации это может быть статический `transform publisher`, задающий смещение `base_link` относительно `map`.

Пример

Рассмотрим типовой сценарий загрузки и визуализации статической карты. Предварительно должен быть установлен пакет `nav2_map_server`. Рабочую область необходимо подготовить отдельно, либо выполнять действия в изолированном терминале с настроенным окружением ROS2.

Исходные файлы карты: room.pgm (серое изображение 400×400 пикселей) и room.yaml. Содержимое room.yaml:

```
image: room.pgm
resolution: 0.05
origin: [-10.0, -10.0, 0.0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Оба файла размещаются в одной директории, например, ~/maps. Для запуска map_server необходимо выполнить команду:

```
ros2 run nav2_map_server map_server --ros-args -p
yaml_filename:=~/maps/room.yaml
```

После успешного старта узел загрузит изображение, сформирует OccupancyGrid и начнёт публиковать топик /map. Для проверки публикуемых данных можно в другом терминале выполнить:

```
ros2 topic echo /map -once
```

В выводе отобразятся метаданные и массив data.

Для визуализации запускается RViz2:

```
ros2 run rviz2 rviz2
```

В интерфейсе RViz2 необходимо:

- в панели Displays добавить элемент “By display type” -> “Map”;
- в свойствах добавленного дисплея Map убедиться, что поле Topic указывает на /map;
- в разделе Global Options установить Fixed Frame в значение “map” (система координат карты);
- убедиться, что карта отображается корректно, цвета соответствуют занятым, свободным и неизвестным областям.

Чтобы сопоставить карту с воображаемым положением робота, требуется обеспечить публикацию TF. Если реальный робот или симуляция не запущены, можно опубликовать статическое преобразование, например:

```
ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 map base_link
```

После этого в RViz добавить дисплей “TF” и модель робота (например, RobotModel из соответствующего описания URDF или просто Axes). Убедиться, что начало координат робота совпадает с ожидаемой точкой на карте (в центре, так как задан нулевой сдвиг). При необходимости параметры origin можно скорректировать в YAML или изменить аргументы static_transform_publisher, чтобы разместить робота в конкретном месте карты.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Загрузите предоставленную карту office.pgm с соответствующим YAML-файлом. Запустите map_server, проверьте публикацию топика /map командой <code>ros2 topic echo --once</code> . В RViz добавьте дисплей Map, установите Fixed Frame в map и получите изображение карты. С помощью static_transform_publisher задайте положение робота в точке с координатами (2.0, 1.0, 0.0) относительно системы координат map и убедитесь в корректности отображения TF.
2	Используя ту же карту, измените в YAML-файле параметр

	resolution на 0.1 (метров на пиксель). Перезапустите map_server. Сравните визуально и по данным топика /map размерность полученной карты. Объясните, как изменение разрешения повлияло на размеры карты в метрических единицах.
3	В исходном YAML-файле карты установите параметр negate: 1. Запустите map_server и отобразите карту в RViz. Опишите, как инвертировались занятые и свободные области. Верните исходное значение и проверьте разницу.
4	Для карты room.yaml измените origin на [5.0, 5.0, 0.0]. Загрузите карту и визуализируйте в RViz. Добавьте в RViz дисплей Grid, чтобы видеть мировую систему координат. Определите, как сместилось изображение относительно начала координат. Сделайте вывод о влиянии параметра origin на положение карты.
5	Создайте launch-файл map_loader.launch.py, который запускает узел map_server с параметром yaml_filename, указывающим на файл warehouse.yaml. Выполните запуск через ros2 launch. Убедитесь, что топик /map публикуется, и карта корректно отображается в RViz.
6	Запустите map_server с картой по умолчанию. С помощью команды ros2 topic echo /map --no-arr --once зафиксируйте метаданные карты. Затем на основе полученного сообщения напишите однострочную команду (с использованием sed/jq или Python в конвейере), которая извлекает разрешение и размеры карты (ширину, высоту) из вывода ros2 topic echo в машиночитаемом виде.
7	При запущенном map_server откройте RViz, добавьте инструмент “Publish Point”. Опубликуйте точку в центре свободной области карты. Подпишитесь на топик /clicked_point (или используйте

	ros2 topic echo), чтобы увидеть координаты точки в системе map. Сопоставьте их с ожидаемым положением на карте.
8	Подготовьте собственное изображение карты в графическом редакторе: на белом фоне нарисуйте простой замкнутый контур (квадрат или окружность) чёрным цветом, сохраните в формате PGM с глубиной 8 бит. Создайте YAML-файл с разрешением 0.02 м/пиксель и нулевым origin. Загрузите карту через map_server и проверьте, что в RViz распознаны занятые и свободные ячейки. Убедитесь, что неизвестных областей нет.
9	Возьмите карту в формате PNG с альфа-каналом (или создайте изображение с полупрозрачными областями). Подготовьте YAML-файл, указав mode: raw. Запустите map_server и проанализируйте значения в массиве data топика /map для пикселей с частичной прозрачностью. Сравните с поведением режима trinary.
10	Запустите первый экземпляр map_server с картой floor1.yaml, ремапировав топик /map в /map_floor1. Запустите второй экземпляр с картой floor2.yaml, ремапировав топик в /map_floor2. В RViz добавьте два дисплея Map, каждый на свой топик, и разместите их в одной визуализации. Убедитесь в одновременном отображении двух разных карт.
11	Используя одну и ту же карту, создайте две конфигурации YAML: в первой occupied_thresh = 0.8, free_thresh = 0.2; во второй occupied_thresh = 0.4, free_thresh = 0.6. Поочерёдно загрузите обе и сравните карты в RViz. Объясните, как изменение порогов влияет на классификацию ячеек.
12	Измените в YAML-файле карты параметр frame_id (если используется параметр frame_id узла) или выполните ремапинг

	/map во frame_id с именем “world”. Запустите map_server. В RViz выставите Fixed Frame “world”. Убедитесь, что карта видна. Добавьте статический transform publisher, публикующий преобразование от world к map с нулевым смещением, и проверьте, что при Fixed Frame map карта также доступна.
13	Запустите map_server с любой картой. Создайте скрипт на Python с использованием rospy, который подписывается на топик /map и выводит значение ячейки в заданной пользователем координате (например, x=2.0, y=1.0). Координаты должны передаваться аргументами командной строки. Проверьте результат, сопоставив с визуализацией в RViz.
14	Для карты с режимом trinary измените YAML, задав mode: scale. Перезапустите map_server. С помощью ros2 topic echo /map --once проанализируйте несколько значений data. Убедитесь, что появляются промежуточные числа между 0 и 100. Отобразите карту в RViz и заметьте отличие цветовой шкалы от стандартного trinary.
15	Намеренно внесите ошибку в YAML-файл: укажите неверный путь к изображению. Запустите map_server. Зафиксируйте сообщение об ошибке в терминале. Исправьте путь и проверьте, что карта загружается. Затем укажите некорректное разрешение (отрицательное) и проанализируйте реакцию узла.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое Occupancy Grid?
2. Как кодируются значения ячеек (свободно/занято/неизвестно)?
3. Как загрузить статическую карту через map_server?
4. Какие параметры задаёт .yaml файл карты?
5. Что такое costmap и чем он отличается от статической карты?
6. Как отобразить карту в RViz?
7. Зачем нужна статическая карта для навигации?
8. Какова роль inflation layer в costmap?

Лабораторная работа 13.

AMCL — АДАПТИВНАЯ ЛОКАЛИЗАЦИЯ МОНТЕ-КАРЛО

Цель и содержание работы: Ознакомление с алгоритмом адаптивной локализации Монте-Карло (AMCL) в среде ROS 2. Получение практических навыков запуска узла AMCL, наблюдения за распределением частиц фильтра, оценки сходимости положения робота на заданной карте и влияния начальной инициализации.

Теоретическое обоснование

Локализация мобильного робота — задача определения его положения (позы) относительно известной карты среды по данным сенсоров и одометрии. Вероятностный подход позволяет обрабатывать неопределённость измерений. Фильтр частиц (particle filter) представляет апостериорную плотность вероятности позы набором взвешенных гипотез — частиц. Каждая частица содержит предполагаемую позу и вес, характеризующий её соответствие текущим наблюдениям.

Адаптивная локализация Монте-Карло (AMCL — Adaptive Monte Carlo Localization) является реализацией фильтра частиц с переменным размером выборки. Ключевая идея KLD-сэмплирования (Kullback-Leibler divergence sampling) заключается в динамическом изменении числа частиц: при высокой неопределённости позы число частиц увеличивается, при хорошей локализации — уменьшается. Это позволяет экономить вычислительные ресурсы без потери точности.

Алгоритм AMCL циклически выполняет следующие шаги:

- предсказание: новая поза каждой частицы вычисляется на основе одометрической модели движения с добавлением случайного шума;

- коррекция: вес частицы обновляется на основе функции правдоподобия измерений лидара относительно карты. Типичные модели сенсора — `likelihood_field` или `beam`;

- ресемплинг: частицы с низким весом заменяются копиями частиц с высоким весом, что предотвращает вырождение выборки. Частота ресемплинга регулируется параметром `resample_interval`;

- оценка позы робота формируется как среднее или наиболее вероятное значение по кластеру частиц.

Для решения проблемы «похищения робота» (kidnapping) AMCL использует два механизма введения случайных частиц: медленное и быстрое восстановление (`recovery_alpha_slow`, `recovery_alpha_fast`). Если средний вес частиц внезапно падает, добавляются случайные частицы, позволяющие перезахватить истинную позу.

Основные параметры узла AMCL в ROS 2 включают:

- `min_particles`, `max_particles` — границы числа частиц;
- `update_min_d`, `update_min_a` — минимальные линейное и угловое перемещения, при которых выполняется обновление фильтра;
- `laser_model_type` — тип модели сенсора;
- `alpha1...alpha5` — параметры шума одометрической модели движения.

Визуализация распределения частиц в RViz позволяет оценить сходимость фильтра и момент, когда робот однозначно локализован.

Пример

Рассмотрим типовой сценарий запуска AMCL для робота, оснащённого лидаром, в симуляторе с подготовленной картой. В качестве примера используется робот TurtleBot3, симулятор Gazebo, карта помещения в формате occupancy grid.

Подготовка. Убедитесь, что установлены пакеты ROS 2 Humble, turtlebot3_gazebo, turtlebot3_navigation2, а также среда симуляции. В терминале выполните экспорт модели робота:

```
export TURTLEBOT3_MODEL=waffle_pi
```

Запуск симуляции с картой. Запустите Gazebo с миром, содержащим карту:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

В отдельном терминале запустите узел карты (map_server), предоставляющий статическую карту:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True  
map:=/opt/ros/humble/share/turtlebot3_navigation2/map/map.yaml
```

Приведённый файл map.yaml должен быть предварительно создан или заменён на собственный.

Запуск AMCL. Узел AMCL входит в состав navigation2 и запускается автоматически в файле navigation2.launch.py. Для ручного запуска с настройкой параметров можно использовать:

```
ros2 run nav2_amcl amcl --ros-args --params-file amcl_params.yaml
```

Файл параметров определяет модель сенсора, границы частиц и другие настройки. Типовой минимальный набор:

```
amcl:  
  ros__parameters:  
    base_frame_id: "base_footprint"  
    odom_frame_id: "odom"  
    global_frame_id: "map"  
    use_sim_time: True  
    min_particles: 100  
    max_particles: 2000  
    update_min_d: 0.05  
    update_min_a: 0.01  
    laser_model_type: "likelihood_field"  
    laser_likelihood_max_dist: 2.0
```

Визуализация. Запустите RViz с конфигурацией навигации:

```
ros2 launch turtlebot3_navigation2 rviz.launch.py
```

В RViz добавьте дисплеи RobotModel, LaserScan, Map, а также PoseArray с топиком /particlecloud. Этот топик публикует текущий набор частиц. При запуске без начальной оценки позы облако частиц равномерно распределено по свободному пространству карты.

Задание начальной оценки позы. С помощью инструмента «2D Pose Estimate» в RViz укажите приблизительную позу робота на карте. Узел AMCL инициализирует частицы вокруг указанной точки с дисперсией, задаваемой параметрами `initial_pose.x`, `initial_pose.y`, `initial_pose.yaw` и их ковариациями. Альтернативно можно вызвать сервис:

```
ros2 service call /initialpose geometry_msgs/msg/PoseWithCovarianceStamped
"{header: {frame_id: 'map'}, pose: {pose: {position: {x: 0.0, y: 0.0, z: 0.0},
orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}}}"
```

После инициализации начните движение робота (например, телеоперацией) и наблюдайте в RViz, как облако частиц стягивается к истинной позе, а дисперсия уменьшается. Процесс сходимости оценивается визуально: когда частицы формируют плотное компактное скопление, локализацию можно считать успешной.

Завершение работы. Остановите симуляцию и все узлы.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите AMCL с параметрами по умолчанию. Инициализируйте позу с ошибкой 0.3 м по оси X и 15 градусов по рысканию. Зафиксируйте время сходимости облака частиц. Опишите динамику изменения числа частиц, если KLD-

	сэмплирование активно.
2	Установите <code>min_particles=100</code> , <code>max_particles=500</code> . Повторите эксперимент из варианта 1. Сравните время сходимости и компактность итогового скопления частиц. Объясните результат.
3	Задайте <code>update_min_d=0.01</code> , <code>update_min_a=0.005</code> . При медленном ручном движении робота определите, как часто фильтр выполняет коррекцию. Сравните с поведением при стандартных значениях 0.05 и 0.01.
4	Используйте карту большей площади (не менее 200 кв. м). Инициализируйте робота в точке, удалённой от истинного положения на 2 м и с угловой ошибкой 90 градусов. Удалось ли фильтру сойтись за разумное время? Проанализируйте распределение частиц в процессе сходимости.
5	В симуляторе введите дополнительный шум одометрии (увеличьте параметры <code>alpha1...alpha4</code>). Запустите AMCL и оцените, насколько возросла дисперсия частиц при движении по прямой длиной 5 м. Повторите без шума и сравните.
6	Вместо инструмента «2D Pose Estimate» используйте программный вызов сервиса <code>/initialpose</code> . Напишите команду или скрипт на Python, который публикует начальную позу с ковариацией 0.01 м^2 по позиции и 0.001 рад^2 по ориентации. Проверьте, что AMCL принимает инициализацию.
7	Сравните работу AMCL с KLD-сэмплированием и без него. Для отключения установите параметр <code>recovery_alpha_slow=0.0</code> , <code>recovery_alpha_fast=0.0</code> и зафиксируйте <code>max_particles=min_particles</code> . Проследите за числом частиц в течение минуты движения. Сделайте вывод о вычислительной эффективности.

8	Измените модель сенсора: установите <code>laser_model_type=beam</code> . Повторите локализацию в условиях узких коридоров. Сравните точность конечной позы с моделью <code>likelihood_field</code> , измерьте отклонение от истины.
9	В процессе локализованного движения робота с помощью инструмента телепортации в Gazebo переместите его в другую точку карты. Зафиксируйте, как быстро облако частиц перестраивается и восстанавливает правильную позу. Замерьте время повторной сходимости.
10	На короткое время (5–10 с) отключите публикацию данных лидара (например, убив узел драйвера лидара). Наблюдайте за ростом дисперсии частиц. После восстановления данных лидара определите, за какое время фильтр вновь сходится.
11	Если в предыдущих лабораторных работах была создана карта с помощью SLAM, используйте её для AMCL. Убедитесь, что карта корректно загружается <code>map_server</code> , и робот успешно локализуется на ней. Сравните сходимость с таковой на эталонной карте.
12	Смоделируйте ситуацию «похищения робота»: задайте <code>recovery_alpha_fast=0.1</code> и <code>recovery_alpha_slow=0.01</code> , затем в процессе движения телепортируйте робота на значительное расстояние (более 3 м). Опишите динамику облака частиц. Объясните роль параметров <code>recovery</code> .
13	Разработайте минимальный скрипт (Python или командный файл), который последовательно задаёт три разные начальные позы с интервалом 10 с. Для каждой позы замерьте количество итераций фильтра до сходимости. Проанализируйте зависимость скорости сходимости от величины начальной ошибки.

14	Установите параметр <code>resample_interval=2</code> . Сравните поведение фильтра с <code>resample_interval=1</code> при одинаковом начальном отклонении. Обратите внимание на возможное вырождение выборки и количество эффективных частиц.
15	Сконфигурируйте два независимых экземпляра AMCL для двух роботов в единой симуляции. Используйте пространства имён. Убедитесь, что каждый робот корректно локализуется на общей карте, а их облака частиц не влияют друг на друга. Зафиксируйте потребление ресурсов CPU.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа `.docx` или `.pdf`.

Вопросы для защиты работы

1. Что такое AMCL и на каком методе он основан?
2. Что такое частицы в алгоритме Монте-Карло?
3. Как инициализировать начальную позицию в AMCL?
4. Какие параметры AMCL влияют на точность локализации?
5. Как убедиться, что AMCL работает корректно в RViz?
6. Что происходит с распределением частиц при движении?
7. Как AMCL использует данные лидара для обновления гипотез?
8. Почему локализация ухудшается в симметричных помещениях?

Лабораторная работа 14.

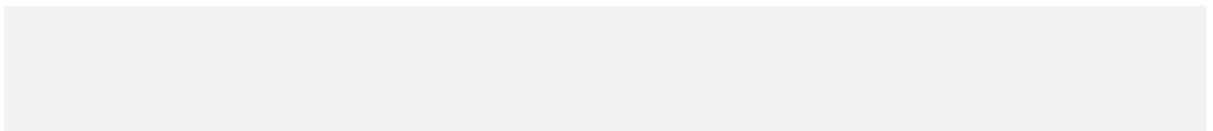
СЛИЯНИЕ ОДОМЕТРИИ И IMU (ФИЛЬТР КАЛМАНА)

Цель и содержание работы: Ознакомиться с принципом комплексирования сенсорных данных на примере расширенного фильтра Калмана (EKF). Реализовать узел, выполняющий слияние зашумлённых показаний одометрии и инерциального измерительного модуля (IMU) для повышения точности оценки положения робота на плоскости. Выполнить сравнение полученной оценки с истинной траекторией и визуализировать результаты.

Теоретическое обоснование

Оценка положения мобильного робота по одометрии страдает от накопления ошибки вследствие проскальзывания колёс и неточности модели. Инерциальные датчики (гироскопы, акселерометры) предоставляют информацию об ориентации и ускорениях, но их прямое интегрирование быстро приводит к дрейфу. Слияние двух источников позволяет компенсировать недостатки каждого из них.

Расширенный фильтр Калмана — рекурсивный байесовский оценщик, работающий с нелинейными моделями. Состояние системы на плоскости описывается вектором $x = [x, y, \theta]^T$, где x, y — координаты, θ — угол рыскания. Управление подаётся в виде одометрических скоростей: линейной v и угловой ω . Модель перехода (предсказание) в дискретном времени с шагом Δt :



Предсказание ковариационной матрицы состояния выполняется с использованием якобиана $F_k = \partial f / \partial x$. Шум процесса Q характеризует неопределённость управления.

Измерения от IMU поступают в виде оценки угла рыскания θ_{imu} (полученной, например, с гироскопа или магнитометра) либо вектора ускорений, позволяющего вычислить ориентацию. В простейшем случае вектор измерений $z = [\theta_{imu}]$. Матрица измерений $H = [0, 0, 1]$. Шум измерений R отражает точность датчика. При использовании нелинейного уравнения измерений применяется якобиан H_k .

Цикл EKF состоит из двух фаз:

- Прогноз:

- Коррекция:

В данной работе слияние реализуется в среде ROS2 на Python с применением библиотеки `filterpy` или самостоятельным кодированием уравнений фильтра.

Пример

Рассмотрим типовой сценарий моделирования и фильтрации. Истинная траектория генерируется как движение с постоянной линейной скоростью 0,5 м/с и угловой скоростью 0,2 рад/с в течение 60 секунд с шагом $\Delta t = 0,02$ с (50 Гц). На основе истинных скоростей формируются зашумлённые одометрические данные: к v добавляется гауссовский шум $\sigma_v = 0,1$ м/с, к ω — $\sigma_\omega = 0,05$ рад/с. Из истинного угла θ формируются измерения IMU с шумом $\sigma_\theta = 0,03$ рад. Считается, что IMU выдаёт оценку угла с той же частотой 50 Гц.

Для фильтрации используется библиотека `filterpy`. Импортируются `ExtendedKalmanFilter` из `filterpy.kalman`, `numpy` и `matplotlib`. Создаётся объект EKF с размерностью состояния 3, измерений 1. Функция перехода состояния $f(x, dt, v, \omega)$ реализует приведённые выше кинематические уравнения, возвращая новый вектор состояния. Её якобиан $F(x, dt, v, \omega)$ вычисляется аналитически или с помощью численного дифференцирования. Функция измерения $h(x)$ просто возвращает третий элемент состояния — угол θ , якобиан $H(x) = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$.

Начальное состояние задаётся нулевым, начальная ковариация $P = \text{np.eye}(3) * 0.1$. Матрица Q выбирается диагональной с элементами, соответствующими дисперсиям скоростных шумов, преобразованными через модель: $q_{11} = (\sigma_v \cdot \Delta t \cdot \cos\theta)^2$, $q_{22} = (\sigma_v \cdot \Delta t \cdot \sin\theta)^2$, $q_{33} = (\sigma_\omega \cdot \Delta t)^2$. Упрощённо полагают $Q = \text{np.diag}([0.01, 0.01, 0.001])$. Матрица $R = \text{np.array}([\sigma_\theta^2])$.

В цикле по времени для каждого шага:

- получаем управляющие скорости (v_noisy, ω_noisy);
- вызываем `predict` с аргументами $dt, v_noisy, \omega_noisy$;
- получаем измерение угла $z = \theta_truth + \text{шум}$;
- вызываем `update(z, HJacobian=H_jacobian, Hx=h_function)`.

На выходе сохраняются истинная траектория, одометрическая траектория (полученная интегрированием зашумлённых скоростей) и отфильтрованная траектория EKF.

Для оценки точности вычисляется среднеквадратическая ошибка (RMSE) положения (евклидово расстояние до истины) и ошибка по углу. Визуализация содержит три подграфика: траектории в плоскости XY, ошибка положения во времени, ошибка угла во времени. Графики демонстрируют, что EKF-оценка значительно ближе к истине, чем чистая одометрия,

особенно по угловому каналу, где дрейф полностью устранён благодаря коррекциям от IMU.

Код оформляется в виде ROS2-узла на Python. Узел подписывается на топики одометрии и IMU (или использует таймер для чтения сгенерированных данных), публикует отфильтрованное положение в топик /ekf_pose и записывает результаты в файл для последующего построения графиков. Графики строятся отдельным скриптом после завершения сбора данных.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Истинная траектория: движение по окружности радиусом 2 м с линейной скоростью 0,4 м/с. Шум одометрии: $\sigma_v = 0,08$ м/с, $\sigma_\omega = 0,04$ рад/с. Шум IMU по углу рыскания: $\sigma_\theta = 0,02$ рад. Частота одометрии и IMU 40 Гц. Длительность моделирования 80 с. Сравнить RMSE положения и угла.
2	Истинная траектория: лемниската Бернулли, задаваемая параметрически $x(t)=A \cdot \cos(t)/(1+\sin^2(t))$, $y(t)=A \cdot \sin(t) \cdot \cos(t)/(1+\sin^2(t))$ с $A=3$ м, t от 0 до 2π с шагом,

	соответствующим средней скорости около 0,3 м/с. Шум одометрии: $\sigma_v = 0,12$ м/с, $\sigma_\omega = 0,06$ рад/с. IMU выдаёт угол с $\sigma_\theta = 0,03$ рад, частота 30 Гц. Визуализировать истинную, одометрическую и EKF траектории на одном графике.
3	Прямолинейное движение вдоль оси X со скоростью 0,5 м/с с добавлением синусоидального рыскания амплитудой 0,3 рад и периодом 20 с. Шум одометрии: $\sigma_v = 0,1$ м/с, $\sigma_\omega = 0,05$ рад/с. IMU имеет не только шум $\sigma_\theta = 0,025$ рад, но и постоянное смещение (bias) 0,05 рад. В фильтре оценивать и компенсировать смещение, расширив состояние до $[x, y, \theta, b_\theta]$.
4	Движение «восьмёркой»: две сопряжённые окружности радиусом 1,5 м, проходимые последовательно. Скорость линейная 0,3 м/с, угловая меняет знак. Шум одометрии: $\sigma_v = 0,07$ м/с, $\sigma_\omega = 0,03$ рад/с. Измерения IMU: угол рыскания с шумом $\sigma_\theta = 0,015$ рад, частота 100 Гц. Частота одометрии 20 Гц. Организовать асинхронное поступление данных: предсказание выполнять на каждом одометрическом шаге, коррекцию — при появлении нового измерения IMU.
5	Траектория — случайное блуждание: на каждом шаге линейная скорость $v=0,4$ м/с, угловая скорость ω выбирается из нормального распределения $N(0, 0,1^2)$. Шум одометрии: $\sigma_v = 0,15$ м/с, $\sigma_\omega = 0,08$ рад/с. IMU измеряет не только угол рыскания ($\sigma_\theta = 0,04$ рад), но и угловую скорость ($\sigma_{\omega_imu} = 0,02$ рад/с). Реализовать EKF, использующий оба измерения с матрицей измерений 2×3 .
6	Движение по квадрату со стороной 4 м, скорость 0,4 м/с, повороты на 90° происходят мгновенно по достижении вершины (имитация идеального поворота). Шум одометрии: $\sigma_v = 0,06$ м/с,

	$\sigma_{\omega} = 0,02$ рад/с. IMU: $\sigma_{\theta} = 0,01$ рад, частота 50 Гц. Имитировать кратковременные потери измерений IMU: в интервале 20–25 с и 40–45 с данные IMU не поступают, фильтр работает только по предсказанию.
7	Использовать ручную реализацию EKF без filterpy. Истинная траектория — синусоида $y=2 \cdot \sin(0,2 \cdot x)$, движение вдоль X со скоростью 0,5 м/с. Шум одометрии: $\sigma_v = 0,09$ м/с, $\sigma_{\omega} = 0,045$ рад/с. IMU: угол рыскания $\sigma_{\theta} = 0,02$ рад. Построить графики изменения диагональных элементов ковариационной матрицы P во времени.
8	Истинная траектория — спираль Архимеда $r = 0,5 + 0,02 \cdot t$, движение с угловой скоростью 0,15 рад/с. Шум одометрии: $\sigma_v = 0,11$ м/с, $\sigma_{\omega} = 0,055$ рад/с. IMU выдаёт только угловую скорость (гироскоп) с шумом $\sigma_{\omega_imu} = 0,03$ рад/с и смещением 0,01 рад/с. Реализовать EKF с расширенным состоянием, включающим смещение гироскопа. Измерения — угловая скорость.
9	Траектория — движение по эллипсу с полуосями 3 м и 1,5 м, линейная скорость непостоянна, максимальна на вершинах. Шум одометрии: $\sigma_v = 0,1$ м/с, $\sigma_{\omega} = 0,04$ рад/с. IMU: угол рыскания $\sigma_{\theta} = 0,02$ рад. Провести сравнение двух стратегий настройки ковариационных матриц Q и R: «оптимистичной» (малые Q, большие R) и «консервативной» (большие Q, малые R). Проанализировать влияние на ошибку и гладкость траектории.
10	Движение по зигзагу: прямые отрезки длиной 2 м под углами 0° , 60° , -60° , повторяющиеся. Скорость 0,35 м/с. Шум одометрии: $\sigma_v = 0,08$ м/с, $\sigma_{\omega} = 0,04$ рад/с. IMU: угол рыскания $\sigma_{\theta} = 0,025$ рад, но частота IMU в два раза ниже частоты одометрии

	(одометрия 50 Гц, IMU 25 Гц). Учесть в реализации интерполяцию измерений IMU по времени.
11	Истина — движение, записанное реальным роботом (предоставить csv-файл с колонками $t, x, y, \theta, v, \omega$). Шум одометрии накладывается искусственно с $\sigma_v = 0,1$ м/с, $\sigma_\omega = 0,05$ рад/с. IMU формируется из истинного угла с шумом $\sigma_\theta = 0,03$ рад и случайными выбросами (10% измерений заменяются на равномерный шум в пределах $\pm 0,5$ рад). Реализовать отбраковку выбросов по критерию Махаланобиса.
12	Модель движения учитывает линейное ускорение: состояние расширено до $[x, y, \theta, v, \omega]$. Управление отсутствует, предсказание по модели постоянной скорости с шумом ускорения. Одометрия выступает в роли измерений v и ω с шумом $\sigma_v = 0,1$ м/с, $\sigma_\omega = 0,05$ рад/с. IMU измеряет угол θ с шумом 0,02 рад. Сравнить с вариантом, где одометрия используется как управление (как в варианте 1).
13	Генерация данных с ненулевым средним шумом одометрии (имитация систематической ошибки): $v_{\text{noisy}} = v_{\text{true}} + 0,05 + N(0, 0,08^2)$, $\omega_{\text{noisy}} = \omega_{\text{true}} + 0,02 + N(0, 0,03^2)$. IMU без смещения. Оценить способность EKF компенсировать систематический сдвиг, включив в состояние константные смещения скоростей.
14	Истинная траектория — восходящая раскручивающаяся спираль, но только в плоскости (зависимость радиуса от времени). Шум одометрии: $\sigma_v = 0,07$ м/с, $\sigma_\omega = 0,035$ рад/с. Измерения IMU: используются показания акселерометра для вычисления угла крена и тангажа, но в данной работе ограничиться углом рыскания, который вычисляется из проекций ускорений в

	предположении горизонтального движения. Реализовать вычисление $\theta_{imu} = \text{atan2}(a_y, a_x)$ и добавить соответствующий шум с $\sigma=0,05$ рад.
15	Движение по траектории «улитка Паскаля» ($r = 2 + 2 \cdot \cos(\theta)$), обход за 60 с. Шум одометрии: $\sigma_v = 0,09$ м/с, $\sigma_\omega = 0,045$ рад/с. IMU: угол рыскания $\sigma_\theta = 0,02$ рад, частота 60 Гц. Добавить задержку измерений IMU на 0,1 с относительно одометрии. Модифицировать EKF для работы с запаздывающими измерениями: хранить историю состояний и применять коррекцию задним числом (out-of-sequence update).

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое фильтр Калмана и для чего он применяется?
2. Как объединяются данные одометрии и IMU?
3. Что такое расширенный фильтр Калмана (EKF)?
4. Какой пакет ROS 2 реализует EKF?
5. Что такое ковариационная матрица?
6. Как настроить параметры robot_localization?
7. Какие проблемы возникают при несинхронизированных сенсорах?
8. Как проверить качество слияния данных в RViz?

Лабораторная работа 15.

РЕАЛИЗАЦИЯ АЛГОРИТМА ДЕЙКСТРЫ НА КАРТЕ

Цель и содержание работы: Реализовать алгоритм глобального планирования на основе метода Дейкстры для occupancy grid карты. Изучить устройство очереди с приоритетом на Python, организацию поиска кратчайшего пути на регулярной сетке, восстановление маршрута и визуализацию результата средствами matplotlib.

Теоретическое обоснование

Глобальное планирование в мобильной робототехнике решает задачу отыскания маршрута от текущего положения робота до заданной цели в известном статическом окружении. Дискретное представление среды в виде occupancy grid карты предполагает разбиение пространства на прямоугольные ячейки, каждая из которых характеризуется вероятностью занятости. В упрощённом бинарном случае ячейка либо свободна (0), либо занята препятствием (100). Поиск пути на такой сетке сводится к нахождению кратчайшего пути на графе, вершинами которого являются свободные ячейки, а рёбра соединяют соседние ячейки.

Алгоритм Дейкстры находит кратчайшие расстояния от стартовой вершины до всех остальных вершин графа с неотрицательными весами рёбер. В контексте сеточной карты вес ребра обычно равен единице (перемещение в соседнюю ячейку), что делает алгоритм эквивалентным поиску в ширину, однако использование общей схемы с приоритетной очередью позволяет впоследствии легко учитывать различные стоимости перемещения (например, пересечённая местность).

Алгоритм поддерживает две структуры данных: массив расстояний `dist` (инициализируется бесконечностью, кроме стартовой ячейки, где `dist = 0`) и очередь с приоритетом, извлекающую вершину с минимальным текущим

расстоянием. На каждом шаге из очереди извлекается вершина u , и для каждого её соседа v проверяется, можно ли улучшить расстояние $\text{dist}[v] = \text{dist}[u] + \text{weight}(u,v)$. Если улучшение возможно, значение обновляется и v помещается в очередь. Процесс продолжается до извлечения целевой вершины или опустошения очереди. Путь восстанавливается обратным ходом по сохранённым предшественникам.

На практике для очереди с приоритетом в Python применяют модуль `heapq`, позволяющий выполнять вставку и извлечение минимума за логарифмическое время. Матрица карты удобно представляется двумерным массивом NumPy. В операционной системе ROS 2 глобальный планировщик `nav2` использует аналогичный подход; данная лабораторная работа повторяет его базовую логику на уровне прототипа.

Пример

Рассмотрим типовой сценарий создания модуля `dijkstra_planner.py`. В качестве карты используем сгенерированный массив NumPy размером 25×25 , где нули обозначают свободные клетки, а 100 — занятые. Препятствия зададим в виде полосы вдоль строки 12 (от столбца 5 до 20) и квадрата со сторонами от 18 до 22 по обеим осям.

```
import numpy as np
import heapq
import matplotlib.pyplot as plt

class DijkstraPlanner:
    def __init__(self, grid):
        self.grid = grid
        self.rows, self.cols = grid.shape
        self.motion = self.get_motion_model()

    def get_motion_model(self):
        # 4-связность: вверх, вниз, влево, вправо
        return [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def plan(self, start, goal):
        start_node = (start[0], start[1], 0)
        goal_node = (goal[0], goal[1])
        if self.grid[start[0], start[1]] == 100 or self.grid[goal[0], goal[1]]
== 100:
            raise ValueError("Старт или цель находятся в препятствии")

        dist = np.full((self.rows, self.cols), np.inf)
```

```

        dist[start[0], start[1]] = 0
        prev = {}
        pq = [(0, start_node[0], start_node[1])]

        while pq:
            cur_dist, x, y = heapq.heappop(pq)
            if cur_dist > dist[x, y]:
                continue
            if (x, y) == (goal_node[0], goal_node[1]):
                break
            for dx, dy in self.motion:
                nx, ny = x + dx, y + dy
                if 0 <= nx < self.rows and 0 <= ny < self.cols and self.grid[nx,
ny] == 0:
                    new_dist = cur_dist + 1
                    if new_dist < dist[nx, ny]:
                        dist[nx, ny] = new_dist
                        prev[(nx, ny)] = (x, y)
                        heapq.heappush(pq, (new_dist, nx, ny))

        path = []
        node = goal_node
        if node not in prev and node != (start[0], start[1]):
            print("Путь не найден")
            return []
        while node != (start[0], start[1]):
            path.append(node)
            node = prev[node]
        path.append((start[0], start[1]))
        path.reverse()
        return path

```

Далее выполним построение карты, запуск планировщика и визуализацию.

```

def build_map():
    grid = np.zeros((25, 25), dtype=np.int8)
    grid[12, 5:21] = 100
    grid[18:23, 18:23] = 100
    return grid

grid = build_map()
planner = DijkstraPlanner(grid)
start = (2, 2)
goal = (22, 22)
path = planner.plan(start, goal)

plt.figure(figsize=(6,6))
plt.imshow(grid, cmap='gray_r', origin='upper')
if path:
    px = [p[1] for p in path]
    py = [p[0] for p in path]
    plt.plot(px, py, color='blue', linewidth=2, marker='o', markersize=2)
plt.plot(start[1], start[0], 'go', markersize=8)
plt.plot(goal[1], goal[0], 'ro', markersize=8)
plt.gca().invert_yaxis()
plt.grid(True)
plt.show()

```

В приведённом примере старт (2,2) и цель (22,22) успешно соединяются линией, огибающей препятствия. Обратите внимание: индексация массива

принята как (строка, столбец), начало координат в верхнем левом углу изображения.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Размер карты: 30x30. Старт: (2, 2). Цель: (27, 27). Препятствия: горизонтальная стена в строке 15 от столбца 5 до 25; вертикальная стена в столбце 20 от строки 10 до 20.
2	Размер карты: 25x25. Старт: (1, 1). Цель: (23, 23). Препятствия: квадрат со сторонами от 10 до 15 по обеим осям; диагональная линия из клеток с координатами (x, y), где $x = y$ для x от 18 до 22.
3	Размер карты: 20x20. Старт: (1, 18). Цель: (18, 1). Препятствия: вертикальные полосы в столбцах 5 и 15 между строками 5 и 15; горизонтальная полоса в строке 10 между столбцами 5 и 15.
4	Размер карты: 40x40. Старт: (2, 2). Цель: (37, 37). Препятствия: лабиринтообразный коридор: стены по строкам 10 и 30 во всю длину, с проходами в столбцах 10 и 30 соответственно; дополнительные блоки в виде квадрата (20:25, 20:25). Использовать 8-связность, диагональные перемещения считать

	стоимостью 1.4.
5	Размер карты: 35x35. Старт: (33, 1). Цель: (1, 33). Препятствия: две наклонные стены: линия клеток, где сумма индексов $x+y = 20$, и линия $x+y = 30$, на участках от края до края.
6	Размер карты: 28x28. Старт: (0, 0). Цель: (27, 27). Препятствия: центральная колонна из занятых клеток в квадрате (13:16, 13:16); кольцевая стена на расстоянии 5 клеток от границ (все клетки, где $x=5$ или $x=22$ или $y=5$ или $y=22$), с разрывом в середине каждой стороны длиной 3 клетки.
7	Размер карты: 32x32. Старт: (1, 1). Цель: (30, 30). Препятствия: множество случайно расположенных одиночных клеток (задать фиксированный список): (5,5), (5,7), (5,9), (6,10), (8,8), (10,5), (10,12), (12,10), (15,15), (15,18), (18,20), (20,22), (22,24), (25,25), (28,27). Гарантируется наличие пути.
8	Размер карты: 20x20. Старт: (1, 10). Цель: (18, 10). Препятствия: горизонтальные барьеры в строках 5, 8, 12, 15, каждый длиной 12 клеток со смещением: строка 5 – столбцы 2-14, строка 8 – столбцы 5-17, строка 12 – столбцы 1-13, строка 15 – столбцы 4-16.
9	Размер карты: 26x26. Старт: (1, 24). Цель: (24, 1). Препятствия: диагональная полоса шириной 2 клетки от (5,5) до (20,20), и зеркальная полоса от (20,5) до (5,20).
10	Размер карты: 50x50. Старт: (2, 2). Цель: (47, 47). Препятствия: сплошная граница по периметру (строки 0 и 49 полностью заняты, столбцы 0 и 49 заняты), внутренние стены: прямоугольник (15:20, 15:35) и прямоугольник (30:35, 15:35). Использовать 8-связность с единичной стоимостью всех соседей.
11	Размер карты: 22x22. Старт: (1, 1). Цель: (20, 20). Препятствия:

	лестничная структура: для каждого x от 2 до 19 с шагом 2 заняты клетки (x, x) , $(x+1, x)$, $(x, x+1)$.
12	Размер карты: 30x30. Старт: (28, 28). Цель: (1, 1). Препятствия: четыре прямоугольных блока: (8:12, 8:12), (8:12, 18:22), (18:22, 8:12), (18:22, 18:22); дополнительно две вертикальные стенки в столбцах 15 и 16 от строк 0 до 14 и от 16 до 29.
13	Размер карты: 24x24. Старт: (2, 2). Цель: (21, 21). Препятствия: все клетки, у которых сумма координат чётная (шахматная доска), за исключением узкого коридора вдоль главной диагонали шириной 1 клетка.
14	Размер карты: 27x27. Старт: (1, 25). Цель: (25, 1). Препятствия: три параллельные диагональные стены: $x+y=12$, $x+y=18$, $x+y=24$, каждая длиной от края до края карты.
15	Размер карты: 33x33. Старт: (1, 1). Цель: (31, 31). Препятствия: концентрические квадратные стены с толщиной в одну клетку: внешний периметр (по границе карты), следующий квадрат со стороной от 8 до 25, внутренний квадрат от 14 до 19, с разрывами в серединах сторон для прохода.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. В чём суть алгоритма Дейкстры?

2. Какая структура данных используется в алгоритме?
3. Как представить карту в виде графа?
4. Какова временная сложность алгоритма Дейкстры?
5. Чем он отличается от поиска в ширину (BFS)?
6. Как учитываются веса рёбер при навигации?
7. В чём ограничения для больших карт?
8. Как отобразить путь как `nav_msgs/Path`?

Лабораторная работа 16.

РЕАЛИЗАЦИЯ АЛГОРИТМА A*

Цель и содержание работы: Реализовать алгоритм A* для поиска кратчайшего пути на двумерной сетке с препятствиями, используя эвристику Манхэттенского расстояния. Сравнить работу A* с алгоритмом Дейкстры по числу посещённых узлов и времени вычисления на различных картах. Визуализировать найденные пути.

Теоретическое обоснование

Алгоритм Дейкстры находит кратчайший путь от начальной вершины до всех остальных в графе с неотрицательными весами рёбер. Он равномерно исследует узлы, увеличивая расстояние от старта, пока не достигнет цели. Это гарантирует оптимальность, но может быть медленным на больших пространствах, так как не использует информацию о направлении к цели.

Алгоритм A* расширяет Дейкстру за счёт эвристической функции $h(n)$, которая оценивает стоимость пути от узла n до цели. Полная оценка узла $f(n) = g(n) + h(n)$, где $g(n)$ – стоимость пути от старта до n . A* использует приоритетную очередь по $f(n)$. При допустимой эвристике (не переоценивающей истинное расстояние) A* гарантирует оптимальность, но обычно посещает меньше узлов, чем Дейкстра, благодаря направленному поиску.

В данной работе используется манхэттенское расстояние для сетки с четырьмя направлениями перемещения (вверх, вниз, влево, вправо). Вес перемещения между соседними клетками принят равным 1.

Сравнение алгоритмов проводится по:

- количеству узлов, добавленных в очередь (посещённых узлов);
- времени выполнения поиска (в миллисекундах);
- длине найденного пути (в количестве шагов).


```

        if use_astar:
            f = tentative_g + self.heuristic(neighbor,
self.goal)

        else:
            f = tentative_g
            heapq.heappush(pq, (f, neighbor))
            elapsed = time.perf_counter() - start_time
            return None, visited_count, elapsed

    def reconstruct_path(self, came_from: Dict, current: Tuple[int,int]) ->
List[Tuple[int,int]]:
        path = [current]
        while current in came_from:
            current = came_from[current]
            path.append(current)
        path.reverse()
        return path

    def visualize_paths(grid, start, goal, path_dijkstra, path_astar,
title="Сравнение путей"):
        plt.figure(figsize=(12,5))
        grid_np = np.array(grid)
        # первая карта - Дейкстра
        plt.subplot(1,2,1)
        plt.imshow(grid_np, cmap='gray_r', origin='upper')
        if path_dijkstra:
            path_arr = np.array(path_dijkstra)
            plt.plot(path_arr[:,1], path_arr[:,0], 'b-', linewidth=2, label='Путь
Дейкстры')
            plt.plot(start[1], start[0], 'go', markersize=10, label='Старт')
            plt.plot(goal[1], goal[0], 'ro', markersize=10, label='Цель')
            plt.title(f'Дейкстра (длина = {len(path_dijkstra)-1 if path_dijkstra else
"нет"})')
        plt.legend()
        # вторая карта - A*
        plt.subplot(1,2,2)
        plt.imshow(grid_np, cmap='gray_r', origin='upper')
        if path_astar:
            path_arr = np.array(path_astar)
            plt.plot(path_arr[:,1], path_arr[:,0], 'r-', linewidth=2, label='Путь
A*')
            plt.plot(start[1], start[0], 'go', markersize=10, label='Старт')
            plt.plot(goal[1], goal[0], 'ro', markersize=10, label='Цель')
            plt.title(f'A* (длина = {len(path_astar)-1 if path_astar else "нет"})')
            plt.legend()
            plt.suptitle(title)
            plt.show()

# Пример использования для заданной карты
grid_example = [
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,0,1,1,1,0,0,0],
    [0,0,0,0,1,0,0,0,1,0],
    [0,1,1,0,1,0,1,0,1,0],
    [0,0,0,0,0,0,1,0,0,0],
    [0,1,1,1,0,1,1,0,1,0],
    [0,0,0,0,0,0,0,0,1,0],
    [0,1,1,1,1,1,0,0,1,0],
    [0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,1,1,1,1,1,0]
]
start = (0,0)
goal = (9,9)

finder = GridPathfinder(grid_example, start, goal)
path_d, visited_d, time_d = finder.dijkstra()
path_a, visited_a, time_a = finder.astar()

```

```

        print(f"Дейкстра: узлов = {visited_d}, время = {time_d*1000:.2f} мс, длина пути = {len(path_d)-1 if path_d else 0}")
        print(f"A*: узлов = {visited_a}, время = {time_a*1000:.2f} мс, длина пути = {len(path_a)-1 if path_a else 0}")
        visualize_paths(grid_example, start, goal, path_d, path_a)

```

В результате выполнения кода выводится сравнительная статистика и два изображения: путь, найденный Дейкстрой, и путь, найденный A*. Ожидается, что A* посетит значительно меньше узлов и выполнится быстрее при сохранении оптимальной длины пути.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Карта 10x10:
2	grid = [
3	grid = [
4	grid = [
5	grid = [
6	grid = [
7	grid = [
8	grid = [
9	grid = [
10	grid = [

11	grid = [
12	grid = [
13	grid = [
14	grid = [
15	grid = [

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. В чём суть алгоритма A^* и чем он отличается от Дейкстры?
2. Что такое эвристическая функция и каковы требования к ней?
3. Какие эвристики используются для навигации по сетке?
4. Что такое открытый и закрытый список в A^* ?
5. Как выбор эвристики влияет на скорость и качество пути?
6. Что такое admissible heuristic?
7. Как A^* ведёт себя при нулевой эвристике?
8. Каковы ограничения A^* в динамических средах?

Лабораторная работа 17.

ИНТЕГРАЦИЯ A* В ROS-УЗЕЛ

Цель и содержание работы: Создать ROS-узел, предоставляющий сервис для построения глобального пути с использованием алгоритма A* (реализованного в лабораторной работе №16). Узел должен принимать запросы с координатами старта и цели, возвращать путь в виде списка точек и публиковать этот путь в топик для визуализации в RViz.

Теоретическое обоснование

В ROS2 взаимодействие между узлами осуществляется через топики (асинхронная передача сообщений) и сервисы (синхронный запрос-ответ). Для построения глобального пути часто используется сервис типа GetPlan, который принимает начальную и целевую позиции и возвращает путь.

В данной работе требуется реализовать ROS-узел, который:

- содержит реализацию A* (можно импортировать модуль из ЛР16);
- объявляет сервис с пользовательским типом сообщения (или использует стандартный nav_msgs/GetPlan);
- по запросу вычисляет путь на статической карте (передаваемой через параметр или загружаемой из файла);
- публикует полученный путь в топик /plan типа nav_msgs/Path для отображения в RViz.

Стандартный интерфейс планировщика в ROS2 часто использует сообщение nav_msgs/GetPlan (запрос: start, goal; ответ: plan). В качестве координат используются geometry_msgs/PoseStamped. Путь представляется как nav_msgs/Path, содержащий массив поз.

Для визуализации пути в RViz достаточно опубликовать сообщение Path в топик /plan и добавить в RViz дисплей Path.

Пример

Рассмотрим типовой сценарий создания ROS-узла с сервисом GetPlan, использующим реализацию A^* на двумерной сетке. Предполагается, что карта представлена в виде OccupancyGrid (загружается из YAML или передаётся параметром). Для простоты в примере карта задаётся в коде как двумерный список, а координаты start и goal задаются в пиксельных индексах.

Создайте пакет ros2_planner со структурой:

```
ros2_planner/  
  ros2_planner/  
    __init__.py  
    a_star.py  
    planner_node.py  
  setup.py  
  package.xml
```

Файл a_star.py содержит класс AStarPlanner, аналогичный реализации из LP16. Файл planner_node.py содержит ROS-узел.

Код a_star.py:

```
import heapq  
from typing import List, Tuple, Optional  
  
class AStarPlanner:  
    def __init__(self, grid: List[List[int]]):  
        self.grid = grid  
        self.rows = len(grid)  
        self.cols = len(grid[0])  
        self.directions = [(0,1), (1,0), (0,-1), (-1,0)]  
  
        def heuristic(self, a: Tuple[int,int], b: Tuple[int,int]) -> int:  
            return abs(a[0]-b[0]) + abs(a[1]-b[1])  
  
        def plan(self, start: Tuple[int,int], goal: Tuple[int,int]) -> Tuple[Optional[List[Tuple[int,int]]], int, float]:  
            open_set = []  
            start_g = 0  
            start_f = start_g + self.heuristic(start, goal)  
            heapq.heappush(open_set, (start_f, start))  
            came_from = {}  
            g_score = {start: 0}  
            visited = 0  
            import time  
            t0 = time.perf_counter()  
  
            while open_set:  
                current_f, current = heapq.heappop(open_set)  
                visited += 1  
                if current == goal:  
                    path = self.reconstruct(came_from, current)  
                    elapsed = time.perf_counter() - t0
```

```

        return path, visited, elapsed
    for dr, dc in self.directions:
        nr, nc = current[0]+dr, current[1]+dc
        neighbor = (nr, nc)
        if 0 <= nr < self.rows and 0 <= nc < self.cols and
self.grid[nr][nc] == 0:
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g <
g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f = tentative_g + self.heuristic(neighbor, goal)
                heapq.heappush(open_set, (f, neighbor))
    elapsed = time.perf_counter() - t0
    return None, visited, elapsed

def reconstruct(self, came_from: dict, current: Tuple[int,int]) ->
List[Tuple[int,int]]:
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

```

Код planner_node.py (ROS2 узел):

```

import rclpy
from rclpy.node import Node
from nav_msgs.srv import GetPlan
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped, Pose
from std_msgs.msg import Header
from .a_star import AStarPlanner

class PlannerNode(Node):
    def __init__(self):
        super().__init__('global_planner_node')
        # объявляем параметры: карта (список списков), разрешение (не
используется для простоты)
        self.declare_parameter('grid', [
            [0,0,0,0,0,0,0,0,0,0],
            [0,1,1,0,0,0,1,1,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,1,0,1,1,1,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,1,0,0,0,1,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,0,0,0,0,0,0]
        ])
        grid = self.get_parameter('grid').value
        self.planner = AStarPlanner(grid)
        self.service = self.create_service(GetPlan, 'get_plan',
self.get_plan_callback)
        self.path_pub = self.create_publisher(Path, '/plan', 10)
        self.get_logger().info('Планировщик запущен, сервис get_plan готов')

    def get_plan_callback(self, request, response):
        start_pose = request.start.pose.position
        goal_pose = request.goal.pose.position
        # в реальном приложении координаты привязываются к карте, здесь
используем целые индексы
        start = (int(start_pose.x), int(start_pose.y))
        goal = (int(goal_pose.x), int(goal_pose.y))
        self.get_logger().info(f'Запрос на планирование от {start} до {goal}')
        path, visited, elapsed = self.planner.plan(start, goal)

```

```

        if path is None:
            self.get_logger().warn('Путь не найден')
            response.plan = Path() # пустой путь
            return response
        # заполняем ответ (nav_msgs/GetPlan)
        ros_path = Path()
        ros_path.header = Header()
        ros_path.header.stamp = self.get_clock().now().to_msg()
        ros_path.header.frame_id = 'map'
        for (r,c) in path:
            pose = PoseStamped()
            pose.header = ros_path.header
            pose.pose.position.x = float(c) # предположим, что x соответствует
колонке
            pose.pose.position.y = float(r) # y соответствует строке
            pose.pose.orientation.w = 1.0
            ros_path.poses.append(pose)
        response.plan = ros_path
        # публикуем путь для визуализации в RViz
        self.path_pub.publish(ros_path)
        self.get_logger().info(f'Путь найден, длина {len(path)}, узлов посещено
{visited}, время {elapsed*1000:.2f} мс')
        return response

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

if __name__ == '__main__':
    main()

```

Файл `setup.py` должен включать точку входа:

```

entry_points={
    'console_scripts': [
        'planner = ros2_planner.planner_node:main',
    ],
},

```

После сборки пакета (`colcon build`) и источника окружения запуск узла:

```
ros2 run ros2_planner planner
```

Вызов сервиса из другого терминала:

```
ros2 service call /get_plan nav_msgs/srv/GetPlan "{start: {pose: {position: {x: 0, y: 0}}}, goal: {pose: {position: {x: 9, y: 9}}}}"
```

Для визуализации в RViz необходимо добавить дисплей типа Path, указать топик `/plan` и выбрать `frame_id` 'map'. Если карта не отображается, можно также опубликовать статическую карту через OccupancyGrid.

При необходимости координаты пересчитываются из реальных метров в пиксели с учётом разрешения и origin карты, но в учебных целях достаточно использовать целые индексы.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	grid = [
2	grid = [
3	grid = [
4	grid = [
5	grid = [
6	grid = [
7	grid = [
8	grid = [
9	grid = [
10	grid = [
11	grid = [
12	grid = [
13	grid = [
14	grid = [

15	grid = [
----	----------

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как реализовать A* в виде ROS-узла?
2. Какие топики принимает и публикует узел планировщика?
3. Как передать карту в узел планировщика?
4. Как принять запрос через сервис или действие?
5. Как опубликовать найденный путь как nav_msgs/Path?
6. Как тестировать интеграцию с RViz?
7. Что такое NavigateToPose в Nav2?
8. Как обработать случай, когда путь не найден?

Лабораторная работа 18.

DYNAMIC WINDOW APPROACH (DWA) — ЛОКАЛЬНЫЙ ПЛАНИРОВЩИК

Цель и содержание работы: Изучить алгоритм Dynamic Window Approach (DWA) для локального планирования движения мобильного робота с дифференциальным приводом. Получить практические навыки реализации генерации допустимых скоростей, построения прогнозных траекторий и многокритериальной оценки. Подготовить основу для последующего использования локального планировщика в навигационном стеке ROS2 (Nav2).

Теоретическое обоснование

Алгоритм DWA решает задачу выбора безопасных управляющих скоростей (v, ω) на коротком интервале времени в среде с препятствиями. Пространство поиска ограничено динамическим окном — множеством скоростей, достижимых с учётом текущих значений и предельных ускорений робота.

Модель движения дифференциального привода описывается системой:

$$x' = v * \cos(\theta)$$

$$y' = v * \sin(\theta)$$

$$\theta' = \omega$$

где v — линейная скорость, ω — угловая скорость.

Динамическое окно формируется как пересечение трёх множеств:

V_s — допустимые по кинематическим ограничениям скорости ($|v| \leq v_{\max}, |\omega| \leq \omega_{\max}$);

V_d — скорости, достижимые за шаг dt при заданных линейном и угловом ускорениях a_v, a_ω :

$v \in [v_{curr} - a_v \cdot dt, v_{curr} + a_v \cdot dt], \omega \in [\omega_{curr} - a_\omega \cdot dt, \omega_{curr} + a_\omega \cdot dt];$

V_a – скорости, при которых робот способен остановиться до столкновения с препятствием на прогнозном горизонте T . Условие безопасной остановки: $v \leq \sqrt{2 * dist(v, \omega) * a_v}$ и аналогично для ω , где $dist$ – минимальное расстояние до препятствий на прогнозной траектории.

Для каждой пары (v, ω) из динамического окна строится дуговая или прямолинейная траектория на интервале прогноза. Качество траектории оценивается целевой функцией:

$$G(v, \omega) = \alpha * heading(v, \omega) + \beta * clearance(v, \omega) + \gamma * velocity(v, \omega)$$

$heading$ – мера ориентации робота к целевой точке (например, $1 - |\Delta\theta|/\pi$, где $\Delta\theta$ – разность между курсом робота и направлением на цель);

$clearance$ – расстояние до ближайшего препятствия на траектории; нормируется на безопасную дистанцию;

$velocity$ – предпочтение более высокой линейной скорости.

Весы α, β, γ подбираются эмпирически. Выбирается пара (v^*, ω^*) , максимизирующая G .

Полученные скорости подаются на контроллеры, и цикл повторяется до достижения цели.

Пример

Рассмотрим типовой сценарий реализации DWA на Python для двумерной симуляции без использования ROS. Робот представлен точкой с ориентацией, препятствия – набором окружностей. Цель задана координатами.

Шаг 1. Импорт библиотек: `math`, `random`, `matplotlib` для визуализации.

Шаг 2. Определение класса робота с полями `x`, `y`, `theta`, `v`, `omega` и методами обновления состояния.

Шаг 3. Функция генерации выборки скоростей:

```
def generate_velocity_samples(v_curr, omega_curr, v_max, omega_max, a_v, a_omega, dt, samples):
    v_min = max(-v_max, v_curr - a_v * dt)
    v_max_eff = min(v_max, v_curr + a_v * dt)
    omega_min = max(-omega_max, omega_curr - a_omega * dt)
    omega_max_eff = min(omega_max, omega_curr + a_omega * dt)
    vs = []
    for _ in range(samples):
        v = random.uniform(v_min, v_max_eff)
        omega = random.uniform(omega_min, omega_max_eff)
        vs.append((v, omega))
    return vs
```

Шаг 4. Прогноз траектории. Для заданных (v , ω) и длительности `predict_time` модель дуговой траектории вычисляет последовательность поз с шагом `dt`. В каждой точке проверяется столкновение с препятствиями. Возвращается признак допустимости и минимальное расстояние до препятствий.

```
def predict_trajectory(x, y, theta, v, omega, predict_time, dt, obstacles, robot_radius):
    steps = int(predict_time / dt)
    min_dist = float('inf')
    for _ in range(steps):
        x += v * math.cos(theta) * dt
        y += v * math.sin(theta) * dt
        theta += omega * dt
        # проверка препятствий
        for (ox, oy, r) in obstacles:
            d = math.hypot(x - ox, y - oy) - r - robot_radius
            if d < min_dist:
                min_dist = d
            if d < 0:
                return False, min_dist # столкновение
    return True, min_dist
```

Шаг 5. Оценочная функция. Для цели (g_x, g_y):

```
def evaluate_trajectory(x, y, theta, v, omega, gx, gy, min_dist, alpha, beta, gamma, safe_dist):
    # heading: разница между курсом и азимутом на цель
    target_angle = math.atan2(gy - y, gx - x)
    heading_error = abs(normalize_angle(target_angle - theta))
    heading_score = 1.0 - heading_error / math.pi
    # clearance: нормализация
    clearance_score = min(min_dist / safe_dist, 1.0) if min_dist != float('inf') else 1.0
    # velocity
    velocity_score = v / v_max # v_max глобальная максимальная скорость
    return alpha * heading_score + beta * clearance_score + gamma * velocity_score
```

Шаг 6. Главный цикл управления. На каждом шаге генерируются выборки, для каждой вычисляется прогноз и оценка, выбирается лучшая допустимая пара, применяются v , ω для обновления робота на время dt . Условие остановки: расстояние до цели меньше порога. Для наглядности используется `matplotlib.animation` для отрисовки карты, препятствий, робота и выбранной траектории.

```
def main():
    # начальное состояние робота, цель, список препятствий
    robot = Robot(start_x, start_y, start_theta)
    while True:
        dist = math.hypot(goal_x - robot.x, goal_y - robot.y)
        if dist < goal_tolerance:
            print("Цель достигнута")
            break
        samples = generate_velocity_samples(robot.v, robot.omega, v_max,
omega_max, a_v, a_omega, dt, num_samples)
        best_score = -float('inf')
        best_v, best_omega = 0.0, 0.0
        best_min_dist = 0.0
        for v, omega in samples:
            feasible, min_dist = predict_trajectory(robot.x, robot.y,
robot.theta, v, omega, predict_time, dt, obstacles, robot_radius)
            if not feasible:
                continue
            score = evaluate_trajectory(robot.x, robot.y, robot.theta, v, omega,
goal_x, goal_y, min_dist, alpha, beta, gamma, safe_dist)
            if score > best_score:
                best_score = score
                best_v = v
                best_omega = omega
                best_min_dist = min_dist
        robot.update(best_v, best_omega, dt)
        # опционально: сохранение данных для анимации или отрисовка текущего
шага
```

Ключевые параметры: $dt = 0.1$ с, $predict_time = 2.0$ с, $v_max = 0.6$ м/с, $\omega_max = 1.0$ рад/с, $a_v = 0.3$ м/с², $a_\omega = 0.5$ рад/с², $\alpha=0.5$, $\beta=0.3$, $\gamma=0.2$, $safe_dist = 0.5$ м, $robot_radius=0.2$ м.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;

5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте DWA с равными весами оценочной функции ($\alpha=1.0$, $\beta=1.0$, $\gamma=1.0$). Проверьте работоспособность в среде с тремя статическими препятствиями. Зафиксируйте траекторию и время достижения цели.
2	Измените веса на $\alpha=1.0$, $\beta=0.5$, $\gamma=0.2$. Сравните с вариантом 1 по длине пути и минимальному расстоянию до препятствий. Объясните различия.
3	Добавьте в оценочную функцию четвёртый компонент – штраф за большие угловые скорости: $\text{penalty} = \lambda * \omega / \omega_{\text{max}}$. Подберите коэффициент λ , минимизирующий осцилляции робота на прямых участках.
4	Реализуйте ограничение не только по максимальным скоростям, но и по ускорениям, как описано в теории. Проведите два эксперимента: с ограничением и без. Отобразите графики скоростей от времени.
5	Смоделируйте датчик расстояния с максимальной дальностью 2 метра. Препятствия за пределами этой дальности игнорируются при расчёте clearance. Проверьте работу робота в среде с дальними и ближними препятствиями.
6	Вместо полного перебора дискретных скоростей используйте случайную выборку из динамического окна (500 сэмплов). Сравните время вычисления и качество управления с равномерной сеткой 20x20.

7	Добавьте в сценарий движущееся препятствие (например, круг, смещающийся с постоянной скоростью 0.2 м/с). Оцените способность робота безопасно разойтись.
8	Реализуйте визуализацию средствами matplotlib, где на каждом шаге отображаются: текущая поза робота, цель, препятствия, все сгенерированные допустимые траектории (полупрозрачными линиями) и выбранная траектория (жирной линией). Сохраните анимацию в файл.
9	Модифицируйте условие успешного достижения цели: робот должен находиться в радиусе 0.1 м от цели и иметь линейную скорость менее 0.05 м/с. Протестируйте, исключив случаи проезда мимо цели.
10	Сделайте веса оценочной функции адаптивными: при расстоянии до цели больше 3 м $\alpha=0.6$, $\beta=0.2$, $\gamma=0.2$; при расстоянии меньше 0.5 м $\alpha=0.1$, $\beta=0.7$, $\gamma=0.2$. Сравните поведение с постоянными весами.
11	Введите ограничение минимальной линейной скорости $v_{\min} = 0.1$ м/с (робот не может двигаться медленнее). Это исключает выбор очень малых скоростей. Проверьте, как изменится поведение в узком коридоре между препятствиями.
12	Реализуйте прохождение маршрута из трёх последовательных путевых точек. Цель переключается, когда робот входит в окрестность текущей точки. Используйте DWA на каждом сегменте.
13	Сгенерируйте случайную карту с 10 окружностями препятствий. Запустите 20 эпизодов со случайными начальными и целевыми точками. Подсчитайте долю успешных достижений цели без столкновений. Постройте гистограмму пройденного расстояния.

14	Измените расчёт clearance: вместо расстояния до всех препятствий, учитывайте только те, что находятся в секторе $\pm 45^\circ$ от направления движения робота. Сравните с полным обзором в лабиринтообразной среде.
15	Интегрируйте симулятор с библиотекой rpygame: окно реального времени, клик мышью задаёт новую цель. Робот непрерывно рассчитывает DWA и перемещается. Предусмотрите ручную расстановку препятствий через консоль.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. В чём суть алгоритма Dynamic Window Approach?
2. Что такое пространство скоростей и динамическое окно?
3. Как DWA учитывает кинематические ограничения робота?
4. Из каких компонентов состоит целевая функция DWA?
5. Чем локальный планировщик отличается от глобального?
6. Как DWA реагирует на динамические препятствия?
7. Каковы параметры, влияющие на поведение DWA?
8. В каких случаях DWA не находит безопасную траекторию?

Лабораторная работа 19.

DWA В ROS

Цель и содержание работы: Освоить настройку локального планировщика DWA в навигационном стеке ROS2 (Nav2). Научиться конфигурировать скоростные и оценочные параметры, запускать симуляцию с препятствиями, анализировать поведение робота и интерпретировать влияние отдельных параметров на качество движения.

Теоретическое обоснование

Nav2 — фреймворк навигации для ROS2, реализующий модульный стек: глобальный планировщик (путь), локальный планировщик (управление), контроллер, поведенческие деревья, восстановительные поведения. Локальный планировщик DWB (DWAPlanner) основан на алгоритме Dynamic Window Approach. Он генерирует допустимые пары скоростей (v , ω) для дифференциального привода, прогнозирует траектории на короткий интервал, оценивает их по набору критиков (critics) и публикует команду в топик /cmd_vel.

Ключевые параметры, определяющие динамическое окно:

- max_vel_x, min_vel_x — максимальная и минимальная линейные скорости.
- max_vel_theta (max_rot_vel) — максимальная угловая скорость.
- acc_lim_x, acc_lim_theta — предельные ускорения по линейной и угловой скоростям.
- min_speed_xy — минимальная абсолютная линейная скорость, подавляет нулевые команды.
- sim_time — длительность прогноза в секундах.
- vx_samples, vtheta_samples — количество сэмплов скоростей.

Оценка траекторий выполняется критиками (GoalDist, GoalAlign, PathAlign, ObstacleFootprint, Oscillation и др.), каждый из которых вычисляет свою оценку. Итоговая стоимость траектории — взвешенная сумма оценок с весами (scale). Настройка сводится к подбору весов и скоростных ограничений так, чтобы робот двигался к цели, избегал препятствия и не осциллировал.

Пример

Рассмотрим типовой сценарий настройки DWA в Nav2 для TurtleBot3 в Gazebo (пакет turtlebot3_gazebo). ОС Ubuntu 22.04, ROS2 Humble. Пакеты устанавливаются из стандартного репозитория.

Установка необходимых пакетов:

```
sudo apt install ros-humble-turtlebot3-gazebo ros-humble-nav2-bringup
```

Запуск симуляции с картой:

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

В отдельном терминале запуск Nav2 с локализацией AMCL:

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true
map:=/путь/к/вашему.yaml
```

Для изменения параметров DWA используется файл конфигурации. По умолчанию конфигурация находится в пакете nav2_bringup/params/nav2_params.yaml. Создадим собственный YAML-файл, содержащий секцию локального планировщика:

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: 0.0
      max_vel_x: 0.26
      max_vel_theta: 0.5
      min_speed_xy: 0.1
      acc_lim_x: 0.8
      acc_lim_theta: 1.0
      vx_samples: 20
      vtheta_samples: 40
```

```

sim_time: 1.5
critics: ["RotateToGoal", "Oscillation", "GoalAlign", "GoalDist",
"PathAlign", "PathDist", "ObstacleFootprint"]
GoalAlign.scale: 24.0
GoalDist.scale: 24.0
PathAlign.scale: 16.0
PathDist.scale: 16.0
ObstacleFootprint.scale: 1.0
Oscillation.scale: 10.0
RotateToGoal.scale: 32.0
RotateToGoal.slowness_factor: 5.0

```

Для запуска с этой конфигурацией используем:

```

ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=./map.yaml
params_file:=./my_dwa_params.yaml

```

После старта роботу задаётся цель через RViz (кнопка 2D Goal Pose). Наблюдаем в топике /cmd_vel публикуемые команды. Для замера поведения при изменении параметра увеличим max_vel_x до 0.5. Отредактируем файл, перезапустим Nav2. Сравним время прохождения одинакового маршрута, плавность движения (по графику скоростей, полученному через ros2 topic echo /cmd_vel). Анализ показывает, что увеличение максимальной скорости сокращает время, но в узком коридоре может приводить к сближению с препятствиями.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания

1	Запустите стандартную конфигурацию DWA с параметрами по умолчанию. Задайте две цели на противоположных концах карты. Замерьте время прохождения, постройте траекторию в RViz. Зафиксируйте базовые показатели.
2	Измените <code>max_vel_x</code> с 0.26 на 0.5. Повторите маршрут варианта 1. Сравните время, минимальное расстояние до препятствий и плавность графика линейной скорости. Объясните, почему возросла или снизилась безопасность.
3	Увеличьте <code>acc_lim_theta</code> до 2.0, оставив <code>acc_lim_x</code> без изменений. Зафиксируйте, как изменилась угловая скорость при обходе угла. Постройте график $\omega(t)$ из топика <code>/cmd_vel</code> до и после изменения.
4	Установите <code>min_vel_x</code> равной 0.1 (робот всегда движется вперёд с минимальной скоростью). Проверьте поведение в среде с узким проходом. Возникают ли заторы? Объясните результат.
5	Измените параметр <code>sim_time</code> на 3.0. Сравните с <code>sim_time=1.5</code> по дальности обнаружения препятствий и способности планировать длинные обходные пути. Проанализируйте влияние на вычислительную нагрузку (частота планировщика).
6	Полностью отключите критик <code>ObstacleFootprint</code> (закомментируйте его в списке <code>critics</code> и удалите его <code>scale</code>). Запустите робота в среде с несколькими цилиндрическими препятствиями. Опишите последствия. Почему робот перестаёт избегать препятствия?
7	Увеличьте вес <code>GoalAlign.scale</code> до 48.0, уменьшив <code>PathAlign.scale</code> до 8.0. Пронаблюдайте, как робот поворачивается на месте в сторону цели, игнорируя глобальный путь. Заснимите характерное поведение и сравните длину фактической

	траектории с вариантом 1.
8	Включите критик Oscillation и установите его вес 50.0. Зафиксируйте, подавляются ли колебания линейной скорости при движении вдоль прямой. Постройте график $v(t)$ в двух случаях: Oscillation.scale=0 и Oscillation.scale=50.
9	Настройте параметры для движения задним ходом: разрешите отрицательные min_vel_x (например, -0.26) и задайте цель позади робота. Проверьте, способен ли планировщик выбрать траекторию заднего хода. Запишите видео.
10	Измените количество выборок скоростей: vx_samples=10, vtheta_samples=20. Сравните с конфигурацией 20/40 по времени вычисления (ros2 topic hz /local_plan) и по качеству огибания препятствия. Сделайте вывод о минимально необходимой дискретизации.
11	В мире turtlebot3_world добавьте динамическое препятствие (запустите модель движущегося цилиндра через отдельный узел). Проверьте, как DWA реагирует на движущийся объект. Оцените задержку уклонения.
12	Создайте карту с лабиринтом (например, с помощью gazebo_walls). Проведите 10 пусков из фиксированной стартовой точки в целевую. Подсчитайте долю успешных проездов (достижение цели без столкновений) с параметрами по умолчанию. Затем измените max_vel_theta на 1.0 и повторите эксперимент. Сравните успешность.
13	Через динамическую реконфигурацию (rqt_reconfigure или ros2 param set) изменяйте параметр max_vel_x во время движения робота от 0.2 до 0.4 и обратно. Постройте график зависимости времени достижения цели от значения max_vel_x.

14	Сравните поведение DWA и встроенного локального планировщика ТЕВ (замените плагин FollowPath на <code>teb_local_planner::TEBLocalPlanner</code> , потребуется установить <code>ros-humble-teb-local-planner</code>). В одинаковой среде с поворотами зафиксируйте различия в плавности, колебаниях и длине пути.
15	Реализуйте узел-анализатор, который подписывается на <code>/cmd_vel</code> и вычисляет интегральные показатели: суммарный поворот (интеграл $ \omega dt$), максимальную скорость, среднее ускорение. Запустите три варианта конфигураций DWA (спокойная, агрессивная, сбалансированная) и сведите результаты в таблицу.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа `.docx` или `.pdf`.

Вопросы для защиты работы

1. Как интегрировать DWA в навигационный стек ROS 2?
2. Какой пакет реализует DWA в Nav2?
3. Как настроить параметры DWA через YAML?
4. Как проверить работу DWA в симуляции?
5. Что такое `costmap_ros`?
6. Как отобразить локальный план в RViz?
7. Как DWA взаимодействует с глобальным планировщиком?
8. Какие метрики оценивают качество локального планировщика?

Лабораторная работа 20.

TIMED ELASTIC BAND (TEB) — АЛЬТЕРНАТИВА DWA

Цель и содержание работы: Освоить настройку и применение локального планировщика Timed Elastic Band (TEB) в навигационном стеке Nav2. Провести сравнительный анализ поведения TEB и DWA в идентичных симуляционных сценариях. Научиться интерпретировать влияние ключевых параметров TEB на гладкость, время прохождения маршрута и способность огибать препятствия.

Теоретическое обоснование

Timed Elastic Band (TEB) — локальный планировщик, формулирующий задачу движения как нелинейную оптимизацию последовательности поз робота с временными интервалами между ними. В отличие от DWA, который на каждом шаге перебирает конечный набор скоростей и немедленно применяет лучшую, TEB строит и непрерывно оптимизирует упругую ленту (band) из промежуточных состояний от текущей позы до подцели на глобальном пути. В процессе оптимизации одновременно учитываются:

- ограничения по скоростям и ускорениям (`max_vel_x`, `max_vel_theta`, `acc_lim_x`, `acc_lim_theta`);
- кинематические ограничения (`min_turning_radius` для неголономных роботов);
- целевые функции: минимизация времени (`weight_optimaltime`), сглаживание траектории (`weight_kinematics`), удаление от препятствий (`weight_obstacle`), следование глобальному пути (`weight_viapoint`), штраф за рывки (`weight_acc_lim_x`).

Ключевые параметры, определяющие поведение TEB:

- `dt_ref` — желаемый временной интервал между соседними позами (влияет на детализацию ленты);

- max_vel_x, max_vel_theta — кинематические пределы;
- min_turning_radius — минимальный радиус поворота, ограничивает кривизну;
- xy_goal_tolerance, yaw_goal_tolerance — точность достижения цели;
- weight_optimaltime — вес компонента минимизации времени (чем больше, тем агрессивнее движение);
- weight_obstacle, obstacle_poses_affected — настройки обхода препятствий.

Результатом работы планировщика является команда скорости /cmd_vel, рассчитанная из первой пары оптимизированной ленты. Траектория пересчитывается с высокой частотой (обычно 10-20 Гц) и адаптируется к динамическим препятствиям за счёт быстрой переоптимизации.

Пример

Рассмотрим типовой сценарий настройки ТЕВ в Nav2 для TurtleBot3 в симуляции Gazebo. ОС Ubuntu 22.04, ROS2 Humble. Пакет teb_local_planner уже должен быть установлен (sudo apt install ros-humble-teb-local-planner). Карта и симуляция идентичны лабораторной работе 19.

Шаг 1. Запуск симуляции с миром turtlebot3_world:

```
export TURTLEBOT3_MODEL=waffle
```

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Шаг 2. Создание конфигурационного файла nav2_teb_params.yaml, в котором локальный планировщик заменён на ТЕВ. Содержимое файла:

```
controller_server:
  ros__parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "teb_local_planner::TebLocalPlannerROS"
      min_vel_x: -0.26
      max_vel_x: 0.26
      max_vel_theta: 0.5
      acc_lim_x: 0.8
      acc_lim_theta: 1.0
      xy_goal_tolerance: 0.15
      yaw_goal_tolerance: 0.1
```

```
dt_ref: 0.3
min_turning_radius: 0.0
weight_optimaltime: 1.0
weight_obstacle: 50.0
weight_kinematics: 1.0
weight_viapoint: 1.0
obstacle_poses_affected: 15
footprint_model: "point"
global_frame: "map"
odom_topic: "odom"
```

Шаг 3. Запуск Nav2 с использованием созданного файла параметров и карты (заранее построенной или идущей с turtlebot3_world):

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true
map:=путь/к/карте.yaml params_file:=./nav2_teb_params.yaml
```

Шаг 4. Задание цели через RViz. Робот начинает движение. Для количественного сравнения с DWA необходимо запустить тот же сценарий с конфигурацией DWA, использованной в ЛР 19 (базовые параметры), и проехать идентичный маршрут.

Шаг 5. Сбор метрик. Подписываемся на /cmd_vel для записи скоростей. Вычисляем гладкость пути как среднеквадратичное изменение угловой скорости (вариации ω) на единицу пути. Время прохождения фиксируется от момента отправки цели до достижения целевой окрестности. Траекторию из /local_plan можно сохранить с помощью `ros2 topic echo`.

Типовые команды сравнения (DWA) – запуск с параметрами DWA из ЛР 19:

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=./map.yaml
params_file:=./nav2_dwa_params.yaml
```

Результаты сводятся в таблицу: время в пути, максимальная скорость, интегральная мера гладкости (например, сумма квадратов $\Delta\omega$). Визуальное сравнение траекторий в RViz показывает, что ТЕВ генерирует более плавные дуги, в то время как DWA иногда допускает угловатые манёвры.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Установите ТЕВ-планировщик и запустите робота с параметрами по умолчанию (как в примере). Задайте маршрут с двумя поворотами. Запишите время прохождения и визуализируйте траекторию из /local_plan. Это будет базовый эксперимент.
2	Повторите маршрут варианта 1, но с DWA (параметры из ЛР 19). Сравните время прохождения, гладкость (по графику $\omega(t)$) и минимальное расстояние до препятствий. Сделайте вывод, какой планировщик быстрее, а какой безопаснее.
3	Измените параметр <code>dt_ref</code> на 0.5 и на 0.15. Проведите по три заезда для каждого значения. Постройте график зависимости времени прохождения от <code>dt_ref</code> . Объясните, как частота реоптимизации влияет на скорость и колебания.
4	Установите вес <code>weight_optimaltime</code> равным 10.0 (агрессивное движение) и сравните с базовым 1.0. Оцените, насколько быстрее робот проходит маршрут, и как меняется зазор до стен.
5	Настройте ограничение <code>min_turning_radius</code> равным 0.3 м. Протестируйте в коридоре с крутыми поворотами. Опишите, удаётся ли роботу вписаться в поворот, или он вынужден

	совершать манёвры задним ходом.
6	Активируйте строгую ориентацию в целевой точке (<code>yaw_goal_tolerance = 0.02</code>, <code>weight_goal_orientation = 50</code>). Сравните поведение робота при заезде на парковочное место с разворотом. Проверьте, не возникает ли осцилляций на финальном участке.
7	В среде с движущимся препятствием (запустите простой узел, публикующий геометрию движущегося цилиндра) сравните способность ТЕВ и DWA уклониться. Опишите, какой планировщик лучше адаптируется к динамике.
8	Постройте график зависимости гладкости траектории (дисперсия угловой скорости) от веса <code>weight_kinematics</code>. Протестируйте значения 0.1, 1.0, 10.0. Определите значение, при котором робот движется плавно без ударов, но не срезает повороты.
9	Добавьте в конфигурацию параметр <code>visualize_with_time_as_z_axis_scale = 0.5</code> и публикуйте траекторию с временной осью. В RViz отобразите трёхмерную визуализацию ленты. Опишите, как меняется плотность точек ленты на поворотах и прямых.
10	Проведите 15 экспериментов в мире с лабиринтом (один и тот же маршрут). В половине случаев используйте ТЕВ, в половине DWA. Сравните доли успешных проездов без столкновений и без застреваний. Проанализируйте причины неудач.
11	Создайте узел, динамически переключающий планировщик через сервис <code>lifecycle_manager</code>. Запустите робота, в середине маршрута переключите с DWA на ТЕВ. Опишите переходный процесс: есть ли рывок, как быстро ТЕВ принимает управление.
12	Настройте ТЕВ на движение задним ходом, разрешив <code>min_vel_x</code>

	= -0.26 и задав цель позади робота. Сравните с аналогичным экспериментом для DWA (из ЛР 19, вариант 9). Оцените, какой планировщик строит более плавный задний путь.
13	Реализуйте визуализатор областей оптимизации ТЕВ: подпишитесь на /teb_markers (если публикуются) или напишите отрисовку планируемой ленты с градиентом по времени. Сохраните анимацию прохождения лабиринта.
14	Смоделируйте робота с прицепом (turtlebot3_waffle с прицепом в Gazebo). Примените ТЕВ с неголономной кинематикой (параметр wheelbase и т.д.). Проверьте, способен ли планировщик избежать столкновения прицепа с препятствиями. Сравните с DWA в том же сценарии (прицеп неизбежно задевает препятствия).
15	Разработайте скрипт на Python, который автоматически запускает симуляцию, задаёт последовательность из пяти целей, записывает /cmd_vel и /local_plan для ТЕВ и DWA, вычисляет метрики (время, гладкость, максимальную кривизну) и выводит сводный отчёт в CSV. Проведите тестирование для трёх различных карт.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. В чём суть Timed Elastic Band (TEB)?
2. Как ТЕВ учитывает временные и пространственные ограничения?

3. Чем ТЕВ отличается от DWA?
4. Что такое параметр `dt_ref` и как он влияет на поведение?
5. Как ТЕВ учитывает кинематические ограничения?
6. Как настроить ТЕВ для прохождения узких проходов?
7. Как ТЕВ ведёт себя при движущихся препятствиях?
8. Каковы вычислительные требования ТЕВ по сравнению с DWA?

Лабораторная работа 21.

PID-РЕГУЛЯТОР ДЛЯ УПРАВЛЕНИЯ РОБОТОМ

Цель и содержание работы: Освоить принципы пропорционально-интегрально-дифференциального (ПИД) регулирования применительно к управлению движением мобильного робота. Реализовать цифровой ПИД-регулятор на Python, провести экспериментальное исследование влияния коэффициентов на качество переходного процесса, получить навыки настройки регулятора и визуализации ошибки.

Теоретическое обоснование

ПИД-регулятор формирует управляющий сигнал $u(t)$ по ошибке $e(t)$ как сумму трёх составляющих:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(\tau) d\tau + K_d \cdot de(t)/dt,$$

где K_p – пропорциональный коэффициент, K_i – интегральный коэффициент, K_d – дифференциальный коэффициент.

Пропорциональная составляющая реагирует на текущую ошибку, обеспечивая основное управляющее воздействие. Слишком большой K_p вызывает перерегулирование и колебания.

Интегральная составляющая накапливает прошлую ошибку и устраняет статическую ошибку, однако может привести к интегральному насыщению и замедлению реакции.

Дифференциальная составляющая противодействует резким изменениям ошибки, демпфируя колебания и улучшая устойчивость, но чувствительна к шуму измерений.

Для цифровой реализации применяются дискретные аппроксимации: интеграл заменяется суммой (метод прямоугольников или трапеций), производная — разностью. Временной шаг dt должен быть постоянным.

В задаче следования робота вдоль вертикальной линии $x = x_target$ с постоянной линейной скоростью v управление реализуется через угловую скорость ω . Ошибка $e = x_target - x_robot$. Уравнение движения: $dx/dt = v \cdot \cos(\theta)$, $dy/dt = v \cdot \sin(\theta)$, $d\theta/dt = \omega$.

Цель управления: удерживать ошибку $e(t)$ вблизи нуля, то есть стабилизировать робота на линии.

Пример

Рассмотрим типовой сценарий реализации ПИД-регулятора для дифференциального робота в двумерной симуляции без ROS. Робот стартует с начальным отклонением от линии $x=0$, движется с постоянной скоростью $v=0.2$ м/с, управление реализуется угловой скоростью ω , вычисляемой ПИД-регулятором. Симуляция и визуализация выполняются средствами Python (numpy, matplotlib).

Код реализации:

```
import numpy as np
import matplotlib.pyplot as plt

class Robot:
    def __init__(self, x, y, theta):
        self.x = x
        self.y = y
        self.theta = theta
    def update(self, v, omega, dt):
        self.x += v * np.cos(self.theta) * dt
        self.y += v * np.sin(self.theta) * dt
        self.theta += omega * dt

class PID:
    def __init__(self, Kp, Ki, Kd, dt, output_limit=None):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.dt = dt
        self.output_limit = output_limit
        self.integral = 0.0
        self.prev_error = 0.0
    def compute(self, error):
        self.integral += error * self.dt
        derivative = (error - self.prev_error) / self.dt
        output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        self.prev_error = error
        if self.output_limit is not None:
            output = max(self.output_limit[0], min(self.output_limit[1], output))
        return output
```

```

dt = 0.05
t_end = 30.0
v = 0.2
target_x = 0.0
robot = Robot(2.0, 0.0, 0.0)
pid = PID(Kp=0.8, Ki=0.01, Kd=0.2, dt=dt, output_limit=(-1.5, 1.5))

time_hist, err_hist, omega_hist = [], [], []
t = 0.0
while t < t_end:
    error = target_x - robot.x
    omega = pid.compute(error)
    robot.update(v, omega, dt)
    time_hist.append(t)
    err_hist.append(error)
    omega_hist.append(omega)
    t += dt

plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(time_hist, err_hist)
plt.xlabel('Время, с'); plt.ylabel('Ошибка, м')
plt.grid(True)
plt.subplot(1,2,2)
plt.plot(time_hist, omega_hist)
plt.xlabel('Время, с'); plt.ylabel('ω, рад/с')
plt.grid(True)
plt.tight_layout()
plt.show()

```

Запуск демонстрирует переходный процесс: ошибка быстро убывает с небольшим перерегулированием, затем стабилизируется около нуля, управляющий сигнал при этом стремится к нулевой угловой скорости. Экспериментируя с K_p , K_i , K_d , можно наблюдать изменение характера переходного процесса: увеличение K_p ускоряет реакцию, но вызывает колебания; введение K_i ликвидирует остаточную ошибку; увеличение K_d сглаживает колебания.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте П-регулятор ($K_i=0$, $K_d=0$). Подберите K_p , обеспечивающий минимальное время выход в окрестность $ \text{error} < 0.05$ м без устойчивых колебаний. Зафиксируйте остаточную ошибку.
2	Добавьте интегральную составляющую ($K_i=0.02$). Покажите, что статическая ошибка варианта 1 устранена. Проанализируйте появление перерегулирования и длительности переходного процесса.
3	Введите дифференциальную составляющую ($K_d=0.3$). Сравните три графика ошибки: П, ПИ, ПИД. Определите, при каком сочетании достигается наименьшая колебательность.
4	Исследуйте влияние K_p на перерегулирование. Постройте зависимость максимального отклонения от K_p при фиксированных $K_i=0.01$, $K_d=0.2$. Диапазон K_p : 0.2...2.0.
5	Исследуйте влияние K_i на время выхода на установившийся режим. Постройте график $t_{\text{уст}}(K_i)$ для K_i от 0.0 до 0.1. Объясните, почему слишком большой K_i ухудшает устойчивость.
6	Реализуйте защиту от интегрального насыщения (anti-windup): если управляющий сигнал достиг насыщения <code>output_limit</code> , интеграл не накапливается. Сравните переходные процессы с насыщением и без.
7	Используйте трапецеидальную аппроксимацию интеграла вместо прямоугольной. Сравните точность и поведение при большом dt (0.1 с) для двух методов.
8	Внесите в измерение координаты x нормальный шум с $\sigma=0.01$ м.

	Примените фильтр низких частот к производной (K_d компонента). Продемонстрируйте снижение шума в управлении.
9	Ограничьте максимальную угловую скорость значением 1.0 рад/с. Протестируйте при $K_p=2.5$, когда управление стремится к большим значениям. Отобразите работу ограничения на графике $\omega(t)$.
10	Реализуйте адаптивный коэффициент K_p : при $ \text{error} > 0.5$ м $K_p=1.5$, иначе $K_p=0.5$. Сравните со стандартным ПИД с постоянным $K_p=0.8$ по быстродействию и перерегулированию.
11	Замените постоянную целевую линию $x=0$ на синусоидальную траекторию $x_{\text{target}}(t) = 1.0 \cdot \sin(0.2 \cdot t)$. Робот должен отслеживать эту траекторию. Настройте ПИД, минимизирующий среднеквадратичную ошибку.
12	Примените метод Циглера-Никольса для настройки регулятора. Доведите систему до автоколебаний, увеличивая K_p ($K_i=K_d=0$). По критическому коэффициенту и периоду вычислите K_p , K_i , K_d . Сравните с эмпирически подобранными значениями.
13	Реализуйте ПИД-регулятор с двумя степенями свободы (задание и обратная связь имеют разные веса для пропорциональной части). Сравните отклик на ступенчатое изменение уставки с классической схемой.
14	Добавьте в симуляцию запись в CSV-файл данных времени, ошибки и управления. Постройте скрипт для пакетной обработки: автоматический запуск для 5 наборов коэффициентов и сохранение графиков.
15	Визуализируйте на одном 3D-графике (время, ошибка, ω) траектории работы регулятора при разных K_p . Отобразите фазовый портрет (ошибка, производная ошибки) для

	установившегося режима.
--	-------------------------

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое ПИД-регулятор и для чего он применяется?
2. Как каждая из трёх составляющих (П, И, Д) влияет на управление?
3. Что такое перерегулирование (overshoot)?
4. Как выбрать начальные значения коэффициентов?
5. Что такое интегральное насыщение (integral windup)?
6. Как устраняется интегральное насыщение?
7. Что происходит при слишком большом дифференциальном коэффициенте?
8. Как проверить устойчивость системы с ПИД?

Лабораторная работа 22.

ROS-УЗЕЛ УПРАВЛЕНИЯ (CONTROLLER) С PID

Цель и содержание работы: Разработать ROS2-узел контроллера движения на базе ПИД-регулятора. Узел должен подписываться на одометрию и глобальный путь, вычислять ошибку положения и ориентации, формировать управляющие команды скорости и публиковать их в топик /cmd_vel. Провести тестирование в симуляции Gazebo с роботом TurtleBot3, оценить качество отслеживания траектории при различных настройках.

Теоретическое обоснование

В стеке навигации ROS2 контроллер отвечает за преобразование глобального плана (последовательность путевых точек) в управляющие воздействия на исполнительные механизмы робота. Глобальный план публикуется в топик /plan (тип nav_msgs/msg/Path). Текущая оценка положения поступает из одометрии в топик /odom (nav_msgs/msg/Odometry). Команды скорости передаются в топик /cmd_vel (geometry_msgs/msg/Twist) и принимаются драйверами двигателей.

Задача следования по пути может быть решена с помощью ПИД-регулирования поперечного отклонения и угловой ошибки. В классическом подходе выделяют:

- ошибку бокового смещения e_{lat} – кратчайшее расстояние от текущей позы робота до прямолинейного сегмента пути или перпендикуляр к касательной в ближайшей точке траектории;
- ошибку ориентации e_{θ} – разность между текущим курсом робота и направлением касательной к пути.

Управление строится по закону:

$$\omega = Kp_{lat} * e_{lat} + Ki_{lat} * \int e_{lat} dt + Kd_{lat} * de_{lat}/dt + Kp_{\theta} * e_{\theta}$$

Линейная скорость может быть константной или регулироваться отдельным ПИ-контроллером по продольной ошибке.

Цифровая реализация требует дискретизации с постоянным периодом (например, 20 Гц), корректной обработки интегральных составляющих и защиты от насыщения. Интеграция в ROS2 выполняется с использованием класса Node, подписчиков, издателей и таймера для цикла управления.

Пример

Рассмотрим реализацию узла `pure_pursuit_pid_controller` на Python для робота с дифференциальным приводом. Узел подписывается на `/odom` и `/plan`, вычисляет поперечную ошибку и ошибку ориентации относительно ближайшей точки пути, с помощью ПИД формирует ω и задаёт постоянную линейную скорость $v=0.15$ м/с. Регулятор применяется только после получения первого плана.

Код узла (файл `pid_controller.py`):

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

import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry, Path
from geometry_msgs.msg import Twist
import math
import numpy as np

class PIDController(Node):
    def __init__(self):
        super().__init__('pid_controller')
        self.odom_sub = self.create_subscription(Odometry, '/odom',
self.odom_cb, 10)
        self.path_sub = self.create_subscription(Path, '/plan', self.path_cb,
10)

        self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.timer = self.create_timer(0.05, self.control_loop)

        self.path = None
        self.robot_x = 0.0
        self.robot_y = 0.0
        self.robot_theta = 0.0
        self.linear_velocity = 0.15

        # Параметры ПИД
        self.Kp_lat = 0.8
        self.Ki_lat = 0.01
        self.Kd_lat = 0.2
        self.Kp_theta = 0.5
        self.integral_lat = 0.0
        self.prev_error_lat = 0.0
```

```

self.dt = 0.05
self.max_omega = 1.0

def odom_cb(self, msg):
    self.robot_x = msg.pose.pose.position.x
    self.robot_y = msg.pose.pose.position.y
    q = msg.pose.pose.orientation
    siny_cosp = 2.0 * (q.w * q.z + q.x * q.y)
    cosy_cosp = 1.0 - 2.0 * (q.y * q.y + q.z * q.z)
    self.robot_theta = math.atan2(siny_cosp, cosy_cosp)

def path_cb(self, msg):
    self.path = msg

def get_closest_point(self):
    if self.path is None or len(self.path.poses) == 0:
        return None
    min_dist = float('inf')
    closest_idx = 0
    for i, pose_stamped in enumerate(self.path.poses):
        px = pose_stamped.pose.position.x
        py = pose_stamped.pose.position.y
        d = math.hypot(self.robot_x - px, self.robot_y - py)
        if d < min_dist:
            min_dist = d
            closest_idx = i
    return self.path.poses[closest_idx].pose.position

def control_loop(self):
    if self.path is None:
        self.publish_zero()
        return
    target = self.get_closest_point()
    if target is None:
        self.publish_zero()
        return
    # поперечная ошибка (вектор на цель в системе координат робота)
    dx = target.x - self.robot_x
    dy = target.y - self.robot_y
    # поворачиваем вектор в систему робота
    lat_error = dy * math.cos(self.robot_theta) - dx *
math.sin(self.robot_theta)
    # ошибка ориентации
    path_angle = math.atan2(dy, dx)
    theta_error = self.normalize_angle(path_angle - self.robot_theta)

    # ПИД по поперечной ошибке
    self.integral_lat += lat_error * self.dt
    derivative_lat = (lat_error - self.prev_error_lat) / self.dt
    omega_lat = (self.Kp_lat * lat_error +
                self.Ki_lat * self.integral_lat +
                self.Kd_lat * derivative_lat)
    self.prev_error_lat = lat_error

    # компонент от ориентации
    omega_theta = self.Kp_theta * theta_error

    omega = omega_lat + omega_theta
    omega = max(-self.max_omega, min(self.max_omega, omega))

    twist = Twist()
    twist.linear.x = self.linear_velocity
    twist.angular.z = omega
    self.cmd_pub.publish(twist)

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 publish_zero(self):
    twist = Twist()
    twist.linear.x = 0.0
    twist.angular.z = 0.0
    self.cmd_pub.publish(twist)

def main(args=None):
    rclpy.init(args=args)
    controller = PIDController()
    rclpy.spin(controller)
    controller.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Для запуска в составе Nav2 создаётся пакет с зависимостями `rclpy`, `nav_msgs`, `geometry_msgs`. Узел запускается отдельно от стандартного контроллера (локальный планировщик отключается). Настройка Nav2: в файле параметров контроллер `FollowPath` заменяется на заглушку, а данный узел запускается как отдельный жизненный цикл или через `launch`-файл. Альтернативно, можно использовать `move_base_flex` с кастомным контроллером.

Тестирование в симуляции TurtleBot3: запуск мира, AMCL, карта, затем узел контроллера. Задание цели в RViz. Наблюдение за траекторией, оценка точности прибытия и плавности хода. Рекомендуется записывать топики `/odom` и `/cmd_vel` для последующего анализа.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите предоставленный код контроллера с параметрами по умолчанию. Оцените время движения по прямому участку пути и точность финальной позиции. Постройте график боковой ошибки во времени.
2	Добавьте П-регулятор линейной скорости: $v = K_v * (\text{расстояние до конечной точки пути})$, но ограничьте $v_{\text{max}}=0.3$. Сравните время прохождения маршрута с постоянной скоростью.
3	Вместо ближайшей точки используйте lookahead (чистый Pure Pursuit) – цель смещается вперёд на фиксированное расстояние $L=0.5$ м от ближайшей точки. Реализуйте вычисление поперечной ошибки на основе lookahead-точки. Сравните поведение на поворотах.
4	Реализуйте интегральное насыщение (anti-windup) для поперечной ошибки: при выходе ω на ограничение интеграл замораживается. Проверьте, уменьшаются ли перерегулирования на резких поворотах.
5	Добавьте в узел динамическую реконфигурацию параметров ПИД через rclpy параметры (K_p_lat , K_i_lat , K_d_lat). В процессе движения через командную строку изменяйте K_p_lat , наблюдайте за изменением поведения робота. Предоставьте скриншоты.
6	Реализуйте переключение между двумя наборами коэффициентов ПИД в зависимости от скорости робота: если $v > 0.2$, использовать один набор, иначе другой. Проверьте гипотезу о необходимости разных настроек для разных скоростей.

7	Подпишитесь на топик /local_plan (или используйте глобальный путь) и реализуйте ПИД для отслеживания не только поперечного смещения, но и ориентации каждого сегмента пути. Сравните траекторию с вариантом 1.
8	Проведите тест на помехоустойчивость: в симуляции добавьте шум в одометрию (искусственно через ноду, добавляющую гауссов шум в /odom). Подберите фильтр низких частот для производной (Kd). Покажите, как фильтрация влияет на плавность управления.
9	Реализуйте ПДД (пропорционально-дифференциальный с двумя степенями свободы) для угла: введите предварительную компенсацию по целевой угловой скорости (если известна кривизна пути). Сравните ошибку слежения на поворотах.
10	Включите в управление компенсацию трения: добавьте постоянное смещение ω_{bias} , определяемое экспериментально (робот без управления должен оставаться на месте или слабо дрейфовать). Покажите уменьшение статической ошибки.
11	Организируйте проезд по сложному маршруту (восьмёрка) с помощью публикации промежуточных целей. Интегрируйте узел с action-сервером FollowPath. Сравните качество с нативным контроллером Nav2 (DWB или TEB) по показателю overshoot.
12	Добавьте визуализацию: публикуйте маркер текущей целевой точки, поперечной ошибки, lookahead-точки. В RViz отобразите векторы ошибок.
13	Реализуйте алгоритм автоматической настройки ПИД методом Циглера-Никольса в процессе движения (эвристический запуск автоколебаний). Запишите процесс настройки и сравните с

	ручным подбором.
14	Измените структуру регулятора: отдельные ПИД для бокового смещения и ориентации заменяются единым ПИД по кривизне (управление = $K * (\text{эталонная кривизна} - \text{фактическая кривизна})$). Реализуйте вычисление эталонной кривизны из пути. Оцените результат.
15	Проведите сравнительное тестирование вашего контроллера и стандартного контроллера Nav2 (DWB) на 10 различных путях. Зафиксируйте показатели: среднее боковое отклонение, максимальное отклонение, время выполнения. Составьте отчёт с рекомендациями, когда следует применять самописный ПИД.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как реализовать ПИД-регулятор в виде ROS-узла?
2. Какие топики принимает и публикует узел-контроллер?
3. Как вычислить ошибку управления из одометрии?
4. Как задать уставку через параметр или топик?
5. Что такое управляющее воздействие `cmd_vel`?
6. Как протестировать контроллер в Gazebo?
7. Как избежать скачков при запуске контроллера?
8. Как записать и анализировать данные управления?

Лабораторная работа 23.

КОНЕЧНЫЙ АВТОМАТ ДЛЯ ПОВЕДЕНИЯ РОБОТА

Цель и содержание работы: Освоить принципы событийно-ориентированного управления роботом на базе конечного автомата. Разработать ROS2-узел на языке C++, реализующий логику переключения поведений с помощью конструкции switch-case. Изучить механизмы подписки на данные лазерного дальномера и одометрии, публикации команд скорости, а также условия смены состояний. Провести тестирование в симуляционной среде с препятствиями.

Теоретическое обоснование

Конечный автомат (finite state machine, FSM) – математическая модель, описывающая поведение системы в виде совокупности состояний, переходов между ними и действий, выполняемых в каждом состоянии. В робототехнике FSM широко применяется для реализации гибридных архитектур управления, где робот может находиться в одном из ограниченного набора поведений: движение к цели, объезд препятствия, возврат на базу, ожидание команд.

Состояния определяют текущий режим работы. Переходы инициируются событиями – логическими условиями, основанными на данных сенсоров (расстояние до препятствия, достижение позиции, тайм-аут) или внешними командами. Действия в состоянии обычно включают вычисление и публикацию управляющих сигналов.

При реализации на C++ в среде ROS2 удобно использовать перечисляемый тип для состояний, а в цикле управления, запускаемом таймером, размещать оператор switch. В каждой ветви switch выполняются действия текущего состояния и проверяются условия перехода в другое состояние. Такой подход обеспечивает чёткую структуру программы и лёгкость модификации логики.

Основные компоненты узла: подписка на /odom (текущая поза), /scan (данные лазерного дальномера), издатель /cmd_vel, таймер для периодического вызова функции управления.

Пример

Рассмотрим типовой сценарий реализации узла behaviour_fsm на C++. Узел содержит состояния IDLE, MOVE_TO_GOAL, AVOID_OBSTACLE, RETURN_HOME. В IDLE робот неподвижен и ожидает внешней команды (например, сообщения в топик /start). По команде переходит в MOVE_TO_GOAL, где движется к фиксированной цели (x=3.0, y=0.0) с пропорциональным регулятором ориентации и постоянной линейной скоростью. При обнаружении препятствия ближе 0.5 м в секторе $\pm 45^\circ$ по курсу происходит переход в AVOID_OBSTACLE. В этом состоянии робот поворачивается на месте до тех пор, пока путь не станет свободным, после чего возвращается в MOVE_TO_GOAL. При достижении цели (расстояние менее 0.2 м) выполняется переход в RETURN_HOME, где робот движется в начальную точку (0,0). После прибытия домой автомат возвращается в IDLE.

Код узла (файл behaviour_fsm.cpp):

```
#include <rclcpp/rclcpp.hpp>
#include <nav_msgs/msg/odometry.hpp>
#include <sensor_msgs/msg/laser_scan.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <std_msgs/msg/bool.hpp>
#include <cmath>

using namespace std::chrono_literals;

class BehaviourFSM : public rclcpp::Node {
public:
    enum State {IDLE, MOVE_TO_GOAL, AVOID_OBSTACLE, RETURN_HOME};
    BehaviourFSM() : Node("behaviour_fsm"), state_(IDLE) {
        odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
            "/odom", 10, std::bind(&BehaviourFSM::odom_cb, this,
std::placeholders::_1));
        scan_sub_ = this->create_subscription<sensor_msgs::msg::LaserScan>(
            "/scan", 10, std::bind(&BehaviourFSM::scan_cb, this,
std::placeholders::_1));
        start_sub_ = this->create_subscription<std_msgs::msg::Bool>(
            "/start", 10, std::bind(&BehaviourFSM::start_cb, this,
std::placeholders::_1));
        cmd_pub_ = this->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel",
10);
```

```

        timer_ = this->create_wall_timer(100ms,
std::bind(&BehaviourFSM::control_loop, this));

    // параметры цели
    goal_x_ = 3.0;
    goal_y_ = 0.0;
    home_x_ = 0.0;
    home_y_ = 0.0;

    robot_x_ = robot_y_ = robot_theta_ = 0.0;
    obstacle_detected_ = false;
    goal_reached_ = false;
    home_reached_ = false;
    start_received_ = false;
}

private:
void odom_cb(const nav_msgs::msg::Odometry::SharedPtr msg) {
    robot_x_ = msg->pose.pose.position.x;
    robot_y_ = msg->pose.pose.position.y;
    double qx = msg->pose.pose.orientation.x;
    double qy = msg->pose.pose.orientation.y;
    double qz = msg->pose.pose.orientation.z;
    double qw = msg->pose.pose.orientation.w;
    robot_theta_ = std::atan2(2.0*(qw*qz + qx*qy), 1.0 - 2.0*(qy*qy + qz*qz));
}

void scan_cb(const sensor_msgs::msg::LaserScan::SharedPtr msg) {
    // определяем наличие препятствия в переднем секторе
    obstacle_detected_ = false;
    int center_idx = static_cast<int>((-msg->angle_min) / msg->angle_increment);
    int range = static_cast<int>((M_PI/4.0) / msg->angle_increment); // ±45°
    for (int i = center_idx - range; i <= center_idx + range; ++i) {
        int idx = (i + msg->ranges.size()) % msg->ranges.size();
        if (msg->ranges[idx] < 0.5 && msg->ranges[idx] > msg->range_min) {
            obstacle_detected_ = true;
            break;
        }
    }
}

void start_cb(const std_msgs::msg::Bool::SharedPtr msg) {
    if (msg->data && state_ == IDLE) {
        start_received_ = true;
    }
}

void control_loop() {
    geometry_msgs::msg::Twist cmd;
    cmd.linear.x = 0.0;
    cmd.angular.z = 0.0;

    switch (state_) {
        case IDLE: {
            if (start_received_) {
                state_ = MOVE_TO_GOAL;
                start_received_ = false;
                RCLCPP_INFO(this->get_logger(), "Start command received. Moving to
goal.");
            }
            break;
        }
        case MOVE_TO_GOAL: {
            double dx = goal_x_ - robot_x_;
            double dy = goal_y_ - robot_y_;
            double dist = std::hypot(dx, dy);
            if (dist < 0.2) {
                goal_reached_ = true;
            }
        }
    }
}

```

```

    }
    if (goal_reached_) {
        state_ = RETURN_HOME;
        goal_reached_ = false;
        RCLCPP_INFO(this->get_logger(), "Goal reached. Returning home.");
    } else if (obstacle_detected_) {
        state_ = AVOID_OBSTACLE;
        RCLCPP_INFO(this->get_logger(), "Obstacle ahead. Avoiding.");
    } else {
        double target_angle = std::atan2(dy, dx);
        double angle_error = normalize_angle(target_angle - robot_theta_);
        cmd.linear.x = 0.15;
        cmd.angular.z = 0.8 * angle_error;
    }
    break;
}

case AVOID_OBSTACLE: {
    // поворот на месте до исчезновения препятствия
    cmd.angular.z = 0.5; // поворачиваемся влево
    if (!obstacle_detected_) {
        state_ = MOVE_TO_GOAL;
        RCLCPP_INFO(this->get_logger(), "Path clear. Resuming goal.");
    }
    break;
}

case RETURN_HOME: {
    double dx = home_x_ - robot_x_;
    double dy = home_y_ - robot_y_;
    double dist = std::hypot(dx, dy);
    if (dist < 0.2) {
        home_reached_ = true;
    }
    if (home_reached_) {
        state_ = IDLE;
        home_reached_ = false;
        RCLCPP_INFO(this->get_logger(), "Home reached. Entering IDLE.");
    } else {
        double target_angle = std::atan2(dy, dx);
        double angle_error = normalize_angle(target_angle - robot_theta_);
        cmd.linear.x = 0.15;
        cmd.angular.z = 0.8 * angle_error;
    }
    break;
}
}
cmd_pub_->publish(cmd);
}

double normalize_angle(double angle) {
    while (angle > M_PI) angle -= 2.0 * M_PI;
    while (angle < -M_PI) angle += 2.0 * M_PI;
    return angle;
}

rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_sub_;
rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr scan_sub_;
rclcpp::Subscription<std_msgs::msg::Bool>::SharedPtr start_sub_;
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_pub_;
rclcpp::TimerBase::SharedPtr timer_;

State state_;
double goal_x_, goal_y_, home_x_, home_y_;
double robot_x_, robot_y_, robot_theta_;
bool obstacle_detected_;
bool goal_reached_, home_reached_;
bool start_received_;
};

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);

```

```

rclcpp::spin(std::make_shared<BehaviourFSM>());
rclcpp::shutdown();
return 0;
}

```

Для компиляции необходимо добавить зависимости в CMakeLists.txt: rclcpp, nav_msgs, sensor_msgs, geometry_msgs, std_msgs. Запуск узла выполняется в составе симуляции TurtleBot3 с публикацией данных одометрии и лазерного сканера. Робот начинает движение после публикации сообщения Bool со значением true в топик /start:

```

ros2 topic pub /start std_msgs/msg/Bool "data: true"

```

Поведение робота можно наблюдать в RViz и по изменению состояний в КОНСОЛЬНОМ ВЫВОДЕ.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, ВЫВОДЫ;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Добавьте состояние CHARGING. Эмулируйте уровень заряда (переменная, уменьшающаяся с каждым таймером). При падении ниже 20% робот из любого состояния, кроме RETURN_HOME, переходит в CHARGING и движется к зарядной станции (x=0, y=2). После достижения станции заряд восстанавливается до 100%, и автомат возвращается в IDLE.

2	Измените переход AVOID_OBSTACLE -> MOVE_TO_GOAL: теперь робот должен двигаться вдоль препятствия (следуя стене), пока угол на цель не станет свободным, а не просто поворачиваться на месте. Используйте показания боковых секторов лазера.
3	Реализуйте запись пройденной траектории в MOVE_TO_GOAL (сохраняйте массив точек одометрии). В состоянии RETURN_HOME используйте эту траекторию для движения обратно, а не прямую линию.
4	Добавьте состояние FOLLOW_WALL, в которое автомат входит из AVOID_OBSTACLE, если препятствие сохраняется более 5 секунд. FOLLOW_WALL реализует движение на постоянном расстоянии 0.4 м от стены, пока не будет достигнуто направление на цель. После этого возвращается в MOVE_TO_GOAL.
5	Вместо фиксированной цели используйте подписку на топик /target (тип geometry_msgs/Point). При получении нового сообщения разрешите переход из IDLE в MOVE_TO_GOAL. Соответственно измените условие начала движения, убрав топик /start.
6	Добавьте состояние EMERGENCY_STOP, переход в которое происходит из любого состояния при расстоянии до препятствия менее 0.2 м. В этом состоянии робот тормозит и ждёт ручного сброса через топик /reset. После сброса возвращается в предыдущее состояние.
7	Перепишите конечный автомат с использованием паттерна "Состояние" (state pattern). Каждое состояние оформите в виде отдельного класса с общим интерфейсом, а узел вызывает метод handle() текущего состояния. Продемонстрируйте работу на тех

	же четырёх состояниях.
8	Реализуйте состояние SEARCH, в котором робот поворачивается на месте на 360 градусов, сканируя лазером окружающее пространство. Если в секторе обнаруживается объект ближе 1.0 м, робот переходит в состояние APPROACH и приближается к объекту. Добавьте соответствующие переходы.
9	Визуализируйте текущее состояние с помощью маркеров в RViz: опубликуйте текстовый маркер над роботом с названием состояния. Для этого добавьте в узел издателя visualization_msgs::msg::Marker.
10	Создайте сервис /set_state (тип std_srvs/SetBool или кастомный) для ручного переключения состояния из IDLE в MOVE_TO_GOAL или наоборот. Обработайте сервис в узле и обеспечьте безопасный переход.
11	Реализуйте иерархический автомат: внутри состояния AVOID_OBSTACLE добавьте подсостояния TURN_LEFT, TURN_RIGHT, MOVE_BACK, выбираемые на основе свободного пространства. Переход между ними осуществляется без выхода из AVOID_OBSTACLE.
12	Вынесите пороговые значения (расстояние до препятствия для избегания, дистанция достижения цели) в параметры ROS (rclcpp::Parameter). Сделайте их динамически изменяемыми через rqt_reconfigure или ros2 param set.
13	Реализуйте запись лога состояний и временных меток в CSV-файл. При каждом переключении состояния добавляйте строку с текущим временем и новым состоянием. После завершения работы узла обеспечьте сохранение файла.
14	Добавьте детектор движущихся препятствий: анализируйте

	изменение положения препятствий во времени. При обнаружении движущегося объекта, приближающегося к роботу, переходите в состояние EVADE (убегание в направлении от объекта). Проверьте в симуляции с динамическими препятствиями.
15	Интегрируйте конечный автомат с Nav2: используйте action клиент NavigateToPose для движения к цели в состоянии MOVE_TO_GOAL. Состояние AVOID_OBSTACLE заменяется на ожидание завершения локального планирования, а переход в RETURN_HOME осуществляется через вызов NavigateToPose с координатами дома. Остальные состояния могут обрабатываться параллельно.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое конечный автомат (FSM)?
2. Каковы его основные элементы?
3. Как реализовать FSM на Python в ROS 2?
4. Что такое переход между состояниями?
5. Как FSM управляет поведением робота?
6. Как визуализировать текущее состояние?
7. Каковы ограничения FSM при большом числе состояний?

8. Когда FSM предпочтительнее дерева поведения?

Лабораторная работа 24.

ДЕРЕВО ПОВЕДЕНИЯ (BEHAVIOR TREE) В ROS

Цель и содержание работы: Изучить концепцию деревьев поведения (Behavior Trees) как средства организации событийно-управляемой логики робота. Научиться применять библиотеку BehaviorTree.CPP в среде ROS2 для создания модульных, масштабируемых и легко модифицируемых поведенческих моделей. Реализовать набор типовых узлов действия и условий, скомпоновать их в дерево, запустить и протестировать в симуляции, сравнить с конечным автоматом.

Теоретическое обоснование

Дерево поведения (Behavior Tree, BT) — это направленный ациклический граф, представляющий иерархию поведений робота. Основные типы узлов: корень (root), композиты (Sequence, Fallback, Parallel), декораторы и листовые узлы (Action, Condition). При каждом тике (тактыном обновлении) корневой узел рекурсивно обходит дерево, и узлы возвращают статус: SUCCESS, FAILURE или RUNNING.

Sequence выполняет дочерние узлы последовательно; если какой-то возвращает FAILURE, Sequence прерывается и возвращает FAILURE. Если все вернули SUCCESS, Sequence возвращает SUCCESS. Fallback (селектор) выполняет дочерние узлы по порядку до первого, вернувшего SUCCESS или RUNNING; если все вернули FAILURE, Fallback возвращает FAILURE. Действие (Action) выполняет команду (например, движение к точке) и возвращает RUNNING, пока не завершит, после чего SUCCESS или FAILURE. Условие (Condition) немедленно возвращает SUCCESS или FAILURE, не меняя состояние робота.

Деревья поведения обеспечивают модульность: каждое поддерево — переиспользуемый компонент. В сравнении с конечным автоматом (FSM)

они упрощают добавление новых ветвей поведения, так как не требуют переопределения переходов между состояниями.

В ROS2 библиотека BehaviorTree.CPP интегрируется через отдельный узел, который на каждом шаге вызывает tick() корня. Узлы действий могут публиковать команды в топики и взаимодействовать с другими частями системы через черный ящик (Blackboard) для обмена данными между узлами.

Пример

Рассмотрим типовой сценарий: робот должен двигаться к целевой точке, но если уровень заряда батареи низкий, то сначала направиться к зарядной станции. Дерево состоит из Fallback в корне, который сначала проверяет условие низкого заряда, и, если оно истинно, выполняет последовательность «ехать к станции и заряжаться»; иначе выполняется действие «ехать к цели».

Установка библиотеки BehaviorTree.CPP и ROS2-привязки:

```
sudo apt install ros-humble-behaviortree-cpp
```

Библиотека предоставляет шаблоны для создания узлов, синхронизацию с ROS2.

Реализуем три класса узлов: CheckBattery (Condition), MoveToPose (Action), ChargeBattery (Action). Для простоты используем фиктивное время и симуляцию без реального робота: управление ограничится выводом в лог и эмуляцией перемещения.

Код узлов и главного файла (bt_controller.cpp):

```
#include <behaviortree_cpp/bt_factory.h>
#include <behaviortree_cpp/behavior_tree.h>
#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>
#include <chrono>
#include <thread>

using namespace BT;

// Условие: проверка заряда батареи (данные из Blackboard)
class CheckBattery : public BT::ConditionNode {
public:
    CheckBattery(const std::string& name) : BT::ConditionNode(name, {}) {}
```

```

        BT::NodeStatus tick() override {
            double battery_level;
            if (!config().blackboard->get<double>("battery", battery_level)) {
                return BT::NodeStatus::FAILURE;
            }
            return (battery_level < 0.25) ? BT::NodeStatus::SUCCESS :
BT::NodeStatus::FAILURE;
        }
    };

    // Действие: движение к заданной точке (имитация)
    class MoveToPose : public BT::StatefulActionNode {
    public:
        MoveToPose(const std::string& name) : BT::StatefulActionNode(name, {}) {}
        BT::NodeStatus onStart() override {
            std::string target;
            if (!config().blackboard->get<std::string>("target", target)) {
                return BT::NodeStatus::FAILURE;
            }
            RCLCPP_INFO(rclcpp::get_logger("bt"), "Starting to move to %s",
target.c_str());
            start_time_ = std::chrono::steady_clock::now();
            duration_ = std::chrono::seconds(3); // имитация длительности движения
            return BT::NodeStatus::RUNNING;
        }
        BT::NodeStatus onRunning() override {
            auto elapsed = std::chrono::steady_clock::now() - start_time_;
            if (elapsed >= duration_) {
                RCLCPP_INFO(rclcpp::get_logger("bt"), "Arrived at target");
                return BT::NodeStatus::SUCCESS;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            return BT::NodeStatus::RUNNING;
        }
        void onHalted() override {}
    private:
        std::chrono::steady_clock::time_point start_time_;
        std::chrono::seconds duration_;
    };

    // Действие: зарядка батареи
    class ChargeBattery : public BT::StatefulActionNode {
    public:
        ChargeBattery(const std::string& name) : BT::StatefulActionNode(name, {}) {}
        BT::NodeStatus onStart() override {
            RCLCPP_INFO(rclcpp::get_logger("bt"), "Charging battery...");
            start_time_ = std::chrono::steady_clock::now();
            return BT::NodeStatus::RUNNING;
        }
        BT::NodeStatus onRunning() override {
            auto elapsed = std::chrono::steady_clock::now() - start_time_;
            if (elapsed >= std::chrono::seconds(2)) {
                // устанавливаем заряд в 1.0
                config().blackboard->set<double>("battery", 1.0);
                RCLCPP_INFO(rclcpp::get_logger("bt"), "Battery charged to 100%");
                return BT::NodeStatus::SUCCESS;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            return BT::NodeStatus::RUNNING;
        }
        void onHalted() override {}
    private:
        std::chrono::steady_clock::time_point start_time_;
    };

    int main(int argc, char* argv[]) {
        rclcpp::init(argc, argv);
        auto node = rclcpp::Node::make_shared("bt_node");
        // Blackboard
        BT::Blackboard::Ptr blackboard = BT::Blackboard::create();

```

```

blackboard->set<double>("battery", 1.0); // начальный заряд 100%
blackboard->set<std::string>("target", "goal_point"); // цель по умолчанию

// Фабрика регистрирует пользовательские узлы
BehaviorTreeFactory factory;
factory.registerNodeType<CheckBattery>("CheckBattery");
factory.registerNodeType<MoveToPose>("MoveToPose");
factory.registerNodeType<ChargeBattery>("ChargeBattery");

// XML-описание дерева
std::string xml_tree = R"(
<root>
  <BehaviorTree ID="MainTree">
    <Fallback name="root_fallback">
      <Sequence name="low_battery_seq">
        <CheckBattery name="battery_check"/>
        <Sequence name="charge_seq">
          <MoveToPose name="go_to_station" target="charging_station"/>
          <ChargeBattery name="charge"/>
        </Sequence>
      </Sequence>
      <MoveToPose name="go_to_goal" target="goal_point"/>
    </Fallback>
  </BehaviorTree>
</root>
)";

auto tree = factory.createTreeFromText(xml_tree, blackboard);

// Основной цикл с периодичностью 10 Гц
rclcpp::WallRate loop_rate(10);
while (rclcpp::ok()) {
  BT::NodeStatus status = tree.tickOnce();
  if (status == BT::NodeStatus::SUCCESS) {
    RCLCPP_INFO(node->get_logger(), "Behavior tree finished successfully.");
    break;
  } else if (status == BT::NodeStatus::FAILURE) {
    RCLCPP_ERROR(node->get_logger(), "Behavior tree failed.");
    break;
  }
  loop_rate.sleep();
}
rclcpp::shutdown();
return 0;
}

```

Сценарий демонстрирует: если заряд ниже 25% (значение можно изменить через консольный ввод, модифицируя Blackboard), то выполняется сначала движение к станции, затем зарядка; если заряд достаточен, сразу выполняется движение к цели. Дерево масштабируется добавлением новых узлов без изменения существующей логики. В отличие от FSM, здесь не требуется вручную описывать все возможные переходы между состояниями — логика вытекает из структуры дерева.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте действие MoveToPose, которое публикует команды в топик /cmd_vel для реального робота (используйте простой P-регулятор). Подключите одометрию и симулируйте достижение цели. Сохраните базовое дерево из примера.
2	Добавьте узел условия IsObstacleDetected, который проверяет данные лазерного сканера (подписка на /scan, обнаружение препятствия ближе 0.5 м). Вставьте его в дерево перед движением к цели, обернув в Fallback: если препятствие есть, выполняется AvoidObstacle (поворот на месте), иначе обычное движение.
3	Реализуйте декоратор RetryUntilSuccessful, который повторяет попытку заряда батареи до трёх раз, если зарядка не увеличила уровень заряда. Создайте кастомный декоратор или используйте готовый Retry.
4	Замените Fallback на ReactiveFallback (реактивный селектор), который прерывает выполняющиеся дочерние узлы, если условие более высокого приоритета становится истинным. Проверьте, как робот реагирует на внезапное падение заряда во время движения

	к цели.
5	Добавьте Sequence для патрулирования: робот последовательно объезжает три точки (waypoints). Используйте массив координат в Blackboard и итератор. Движение между точками осуществляется через MoveToPose.
6	Реализуйте обмен данными через Blackboard между узлами: действие MoveToPose после выполнения записывает пройденную дистанцию, а следующий узел вычисляет общий путь и сохраняет в файл. Визуализируйте маршрут в RViz.
7	Создайте кастомный узел Parallel, который одновременно выполняет движение к цели и мониторинг заряда (печатает предупреждение, если заряд упал ниже 30%). Используйте Parallel с правилом успеха при успехе первого дочернего.
8	Интегрируйте дерево поведения с ROS2 action: узел действия оборачивает вызов NavigateToPose из Nav2. Дерево должно ждать завершения действия. Сравните с нативной реализацией Nav2.
9	Преобразуйте конечный автомат из лабораторной работы 23 в эквивалентное дерево поведения. Сохраните те же состояния IDLE, MOVE_TO_GOAL, AVOID_OBSTACLE, RETURN_HOME, но реализуйте их как поддеревья. Сравните читаемость кода.
10	Реализуйте подписку на топик /battery_status (тип sensor_msgs/BatteryState) и сделайте условие CheckBattery реальным. При разряде ниже 20% инициируйте возврат на базу.
11	Создайте узел для голосового оповещения (текст в речь): при переходе в состояние зарядки публикуйте сообщение в топик /speak. Остальную часть дерева не меняйте.

12	Разработайте узел-декоратор Timeout, который ограничивает выполнение дочернего узла заданным временем. Примените его к MoveToPose, чтобы робот не зависал, если не может достичь цели. Настройте тайм-аут 10 секунд.
13	Сравните поведение робота под управлением дерева поведения и конечного автомата в одной и той же среде с динамическим препятствием. Оцените, насколько легко внести изменение (добавить обход препятствия) в каждую архитектуру.
14	Реализуйте поддереву для исследования неизвестной территории: робот движется в случайном направлении, избегая препятствий. Добавьте условие «обследовано достаточно» по времени. Используйте ReactiveFallback для переключения на возврат домой при низком заряде.
15	Создайте три различных дерева для одного сценария (патруль, зарядка, уклонение) и загружайте их динамически через параметр ROS. Реализуйте сервис для переключения между деревьями во время выполнения без остановки узла.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое Behavior Tree?
2. Какие типы узлов существуют в дереве поведения?

3. Что означают статусы SUCCESS, FAILURE, RUNNING?
4. Как работает тикание (tick) дерева?
5. Что такое Sequence и Fallback узлы?
6. Какой пакет реализует BT в ROS 2?
7. Как интегрировать BehaviorTree.CPP с ROS 2?
8. Чем дерево поведения лучше конечного автомата?

Лабораторная работа 25.

НАВИГАЦИОННЫЙ СТЕК ROS: ПОЛНАЯ НАСТРОЙКА

Цель и содержание работы: Освоить запуск и настройку полного навигационного стека ROS2 (Nav2) в симуляционной среде Gazebo на примере робота TurtleBot3. Научиться конфигурировать компоненты: map_server, AMCL, глобальный планировщик, локальный планировщик и контроллер. Осуществить автономное перемещение робота к целевой точке с помощью RViz.

Теоретическое обоснование

Навигационный стек Nav2 предназначен для автономного перемещения мобильного робота в известной среде. В его состав входят:

- map_server – загружает статическую карту в формате Occupancy Grid и публикует её в топик /map.
- AMCL (Adaptive Monte Carlo Localization) – оценивает позу робота на основе данных одометрии и лазерного сканера с учётом карты.
- Глобальный планировщик (planner_server) – строит путь из начальной точки в целевую, используя алгоритмы поиска (SmacPlanner, NavFn, Theta* и др.).
- Локальный планировщик и контроллер (controller_server) – генерирует управляющие команды скорости (/cmd_vel) для движения вдоль глобального пути, избегая препятствий (DWB, TEB).
- Сервер поведений (behavior_server) – реализует восстановительные действия при сбоях (очистка карты затрат, вращение на месте).
- Сглаживатель путей (smoother_server) – постобработка маршрута для уменьшения резких поворотов.

Все компоненты обмениваются данными через топики и сервисы, а запуск осуществляется единым launch-файлом, считывающим YAML-конфигурацию параметров.

Пример

Рассмотрим типовой сценарий запуска Nav2 для TurtleBot3 в Gazebo (ROS2 Humble, пакет turtlebot3_gazebo). Карта используется из стандартного мира turtlebot3_world. После запуска роботу задаётся цель через инструмент "2D Goal Pose" в RViz.

Шаг 1. Установка необходимых пакетов (если ещё не установлены):

```
sudo apt install ros-humble-turtlebot3 ros-humble-turtlebot3-gazebo ros-humble-nav2-bringup
```

Шаг 2. Запуск симуляции с картой по умолчанию:

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Шаг 3. Создание YAML-файла параметров nav2_params.yaml. В нём указываются конфигурации плагинов. Минимально необходимый набор для работы:

```
amcl:
  ros__parameters:
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    global_frame_id: "map"
    laser_likelihood_max_dist: 2.0
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    robot_model_type: "differential"
    set_initial_pose: True
    update_min_d: 0.2
    update_min_a: 0.2
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05

bt_navigator:
```

```

ros_parameters:
  default_bt_xml_filename: "navigate_to_pose_w_replanning_and_recovery.xml"
  bt_xml_filename: ""
  plugin_lib_names: ["nav2_compute_path_to_pose_action_bt_node",
                    "nav2_follow_path_action_bt_node",
                    "nav2_back_up_action_bt_node",
                    "nav2_spin_action_bt_node",
                    "nav2_wait_action_bt_node",
                    "nav2_clear_costmap_service_bt_node",
                    "nav2_is_stuck_condition_bt_node",
                    "nav2_goal_reached_condition_bt_node",
                    "nav2_initial_pose_received_condition_bt_node",
                    "nav2_are_error_codes_active_condition_bt_node",
                    "nav2_is_battery_low_condition_bt_node"]

controller_server:
  ros_parameters:
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      min_vel_x: 0.0
      max_vel_x: 0.26
      max_vel_theta: 0.5
      min_speed_xy: 0.1
      acc_lim_x: 0.8
      acc_lim_theta: 1.0
      vx_samples: 20
      vtheta_samples: 40
      sim_time: 1.5
      critics: ["RotateToGoal", "Oscillation", "GoalAlign", "GoalDist",
               "PathAlign", "PathDist", "ObstacleFootprint"]
      GoalAlign.scale: 24.0
      GoalDist.scale: 24.0
      PathAlign.scale: 16.0
      PathDist.scale: 16.0
      ObstacleFootprint.scale: 1.0
      Oscillation.scale: 10.0
      RotateToGoal.scale: 32.0

planner_server:
  ros_parameters:
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_smac_planner/SmacPlanner"
      tolerance: 0.5
      downsample_costmap: false
      allow_unknown: true
      max_iterations: -1
      max_on_approach_iterations: 1000
      max_planning_duration: 2.0
      analytic_expansion_ratio: 3.0
      analytic_expansion_max_length: 2.0
      smooth_path: true
      minimum_turning_radius: 0.4
      lookup_table_size: 20.0

smoother_server:
  ros_parameters:
    smoother_plugins: ["simple_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 0.001

behavior_server:
  ros_parameters:
    local_costmap_topic: "local_costmap/costmap_raw"
    global_costmap_topic: "global_costmap/costmap_raw"
    footprint_topic: "local_costmap/published_footprint"
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "wait"]

```

```

    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    wait:
      plugin: "nav2_behaviors/Wait"

local_costmap:
  ros__parameters:
    global_frame: "odom"
    robot_base_frame: "base_footprint"
    update_frequency: 5.0
    publish_frequency: 2.0
    rolling_window: true
    width: 3
    height: 3
    resolution: 0.05
    footprint: "[[0.15, 0.15], [0.15, -0.15], [-0.15, -0.15], [-0.15, 0.15]]"
    plugins: ["voxel_layer", "inflation_layer"]
    voxel_layer:
      plugin: "nav2_costmap_2d::VoxelLayer"
      enabled: True
      footprint_clearing_enabled: True
      max_obstacle_height: 2.0
      observation_sources: "scan"
      scan:
        topic: "/scan"
        max_obstacle_height: 2.0
        clearing: True
        marking: True
        data_type: "LaserScan"
        min_obstacle_height: 0.0
    inflation_layer:
      plugin: "nav2_costmap_2d::InflationLayer"
      inflation_radius: 0.55
      cost_scaling_factor: 2.0

global_costmap:
  ros__parameters:
    global_frame: "map"
    robot_base_frame: "base_footprint"
    update_frequency: 1.0
    publish_frequency: 0.5
    rolling_window: false
    resolution: 0.05
    footprint: "[[0.15, 0.15], [0.15, -0.15], [-0.15, -0.15], [-0.15, 0.15]]"
    plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
    static_layer:
      plugin: "nav2_costmap_2d::StaticLayer"
      map_subscribe_transient_local: True
    obstacle_layer:
      plugin: "nav2_costmap_2d::ObstacleLayer"
      enabled: True
      observation_sources: "scan"
      scan:
        topic: "/scan"
        max_obstacle_height: 2.0
        clearing: True
        marking: True
        data_type: "LaserScan"
        min_obstacle_height: 0.0
    inflation_layer:
      plugin: "nav2_costmap_2d::InflationLayer"
      inflation_radius: 0.55
      cost_scaling_factor: 2.0

map_server:
  ros__parameters:
    yaml_filename: "/путь/к/вашей/карте.yaml"

```

Шаг 4. Запуск навигационного стека с подготовленным файлом параметров и картой (необходимо предварительно скопировать карту из turtlebot3_gazebo или сохранить свою):

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true  
map:=путь/к/карте.yaml params_file:=./nav2_params.yaml
```

Шаг 5. Запуск RViz для визуализации и задания целей:

```
rviz2 -d $(ros2 pkg prefix  
nav2_bringup)/share/nav2_bringup/rviz/nav2_default_view.rviz
```

В RViz отображается карта, облако частиц AMCL, глобальный и локальный costmap, планируемый путь. С помощью инструмента "2D Goal Pose" задаётся целевая позиция и ориентация. Робот начинает движение, обходит препятствия и достигает цели. В консоли можно наблюдать информационные сообщения о смене состояний и возможных восстановительных действиях.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите навигационный стек с параметрами по умолчанию, как в примере. Установите цель на противоположном конце карты. Зафиксируйте время движения и итоговую точность

	позиционирования. Постройте траекторию с помощью сохранённого топика /plan.
2	Измените глобальный планировщик на NavFn (замените SmacPlanner на NavfnPlanner в секции planner_server). Сравните форму планируемых путей и время построения маршрута для двух разных целей. Объясните различия.
3	Замените локальный планировщик DWB на TEB (ros-humble-teb-local-planner должен быть установлен). Настройте базовые параметры TEB (max_vel_x, max_vel_theta, dt_ref). Сравните с DWB по гладкости траектории и способности вписываться в узкие проходы.
4	Включите сглаживатель пути simple_smoother с параметром tolerance 0.001. Проведите 5 экспериментов в мире с большим количеством поворотов. Оцените визуально и количественно (сумма квадратов кривизны) извилистость пути до и после сглаживания.
5	Измените ограничения скорости робота в локальном планировщике: max_vel_x = 0.5, max_vel_theta = 1.0. Повторите маршрут варианта 1 и зафиксируйте, как изменилось время достижения цели и минимальное расстояние до стен.
6	Создайте собственную карту в формате pgm/yaml с помощью ручного редактора или SLAM. Запустите SLAM (slam_toolbox) в симуляции turtlebot3_world, сохраните карту. Затем используйте эту карту для навигации и проверьте работоспособность.
7	Настройте параметры AMCL: уменьшите число частиц до 500 и увеличьте до 5000. Для каждого случая замерьте среднеквадратичную ошибку локализации (сравнение с ground truth из /ground_truth_odom). Сделайте вывод о влиянии

	количества частиц на точность и вычислительную нагрузку.
8	Добавьте динамическое препятствие в симуляцию (запустите отдельный узел с движущимся цилиндром). Проверьте, как локальный планировщик реагирует на появление препятствия на пути. Запишите видео и отметьте задержку перепланирования.
9	Измените параметры costmap локального уровня: уменьшите rolling_window width/height до 2 метров. Проанализируйте, успевает ли робот вовремя замечать препятствия на высокой скорости. Зафиксируйте момент столкновения или резкого торможения.
10	Активируйте поведение резервного движения (backup) в behavior_server. Искусственно создайте ситуацию застревания (поместите близкий цилиндр перед роботом после начала движения). Убедитесь, что запускается отъезд назад. Логируйте срабатывания через /behavior_server/transition_event.
11	Реализуйте узел-анализатор, подписывающийся на /plan, /local_plan и /cmd_vel. Он должен вычислять интегральное время до цели после каждой поставленной задачи и строить график зависимости времени от длины прямого пути. Проведите 10 запусков с разными целями.
12	Настройте навигационный стек без использования глобальной карты (только локальная), используя rolling_window для global_costmap и плагин SmacPlanner с allow_unknown = true. Дайте роботу команду движения в прямом направлении. Наблюдайте, как планировщик строит путь через неизвестное пространство.
13	Измените конфигурацию, чтобы робот всегда ориентировался вперёд (запрет разворота на месте). Для этого установите

	минимальную линейную скорость 0.05 и отключите поведение Spin. Проверьте, сможет ли робот корректно доехать до цели, требующей разворота.
14	Запишите данные из /odom и /cmd_vel во время миссии. Постройте графики скорости и траектории. Сравните плавность управления для двух вариантов локального планировщика: DWB и ТЕВ. Вычислите рывок (производную ускорения) как меру комфорта.
15	Создайте launch-файл, который единым запуском стартует симуляцию, Nav2 с вашими параметрами и RViz. Прдемонстрируйте полностью автоматизированный запуск автономной навигации. Документируйте структуру launch-файла и использованные аргументы.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Из каких компонентов состоит навигационный стек Nav2?
2. Какую роль выполняет lifecycle_manager?
3. Как настроить планировщики через YAML?
4. Что такое behavior_server и какие поведения он предоставляет?
5. Как задать цель навигации через RViz и через API?
6. Что такое costmap_2d и каковы его слои?

7. Как отлаживать проблемы навигации через логи?
8. Как настроить `inflation_layer`?

Лабораторная работа 26.

АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ НАВИГАЦИИ

Цель и содержание работы: Освоить методы измерения и анализа производительности навигационного стека ROS2. Получить практические навыки сбора метрик времени глобального и локального планирования, частоты выдачи управляющих команд, влияния параметров costmap на вычислительную нагрузку. Научиться интерпретировать результаты и выбирать компромисс между качеством пути и быстродействием.

Теоретическое обоснование

Производительность навигационной системы мобильного робота характеризуется следующими ключевыми показателями:

- Время глобального планирования – интервал от запроса нового пути до публикации результата. Зависит от размера и разрешения costmap, выбранного алгоритма (A*, Dijkstra, SmacPlanner) и вычислительных ресурсов.

- Частота обновления локального плана – количество перестроений траектории в секунду. Определяет способность реагировать на динамические препятствия.

- Частота публикации /cmd_vel – стабильность управляющего сигнала. Низкая частота или джиттер вызывают рывки.

- Задержка между сенсорными данными и командой – общая латентность контура управления.

На производительность влияют параметры карт затрат (costmap): разрешение (resolution) определяет количество ячеек; радиус инфляции и количество слоёв увеличивают время обработки каждого пикселя. Глобальный планировщик, работающий на large-scale карте, критичен к разрешению. Локальный планировщик, ограниченный скользящим окном,

более чувствителен к частоте поступления данных сканера и сложности целевой функции.

Для измерения времени планирования можно использовать action-интерфейс `ComputePathToPose`, который предоставляет сервер планировщика. Засекая время между отправкой цели и получением результата, получаем чистую длительность вычисления пути без учёта коммуникации. Частоту публикации топиков измеряют, вычисляя интервалы между последовательными сообщениями.

Пример

Рассмотрим типовой сценарий анализа производительности навигации TurtleBot3 в Gazebo. Создадим узел `performance_analyzer` на Python, который будет измерять время глобального планирования через action и частоту команд `/cmd_vel`. Затем внесём изменения в `costmap` и сравним показатели.

Шаг 1. Создание пакета и узла. В рабочем пространстве создаём пакет:

```
ros2 pkg create --build-type ament_python performance_analyzer --dependencies rclpy nav2_msgs geometry_msgs std_msgs
```

В файле `performance_analyzer/performance_analyzer_node.py` разместим код:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient
from nav2_msgs.action import ComputePathToPose
from geometry_msgs.msg import PoseStamped, Twist
import time
import statistics

class PerformanceAnalyzer(Node):
    def __init__(self):
        super().__init__('performance_analyzer')
        self.action_client = ActionClient(self, ComputePathToPose,
            'compute_path_to_pose')
        self.cmd_sub = self.create_subscription(Twist, '/cmd_vel',
            self.cmd_callback, 10)
        self.cmd_timestamps = []
        self.planning_times = []

    def cmd_callback(self, msg):
        self.cmd_timestamps.append(time.time())
```

```

def send_goal(self, x, y, theta=0.0):
    goal_msg = ComputePathToPose.Goal()
    goal_msg.goal = PoseStamped()
    goal_msg.goal.header.frame_id = 'map'
    goal_msg.goal.header.stamp = self.get_clock().now().to_msg()
    goal_msg.goal.pose.position.x = x
    goal_msg.goal.pose.position.y = y
    goal_msg.goal.pose.orientation.z = 0.0
    goal_msg.goal.pose.orientation.w = 1.0
    goal_msg.planner_id = ''
    start_time = time.time()
    self._action_client.wait_for_server()
    send_goal_future = self._action_client.send_goal_async(goal_msg)
    send_goal_future.add_done_callback(lambda future:
self.goal_response_callback(future, start_time))

def goal_response_callback(self, future, start_time):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected')
        return
    get_result_future = goal_handle.get_result_async()
    get_result_future.add_done_callback(lambda
self.get_result_callback(future, start_time))

def get_result_callback(self, future, start_time):
    result = future.result()
    end_time = time.time()
    duration = end_time - start_time
    self.planning_times.append(duration)
    self.get_logger().info(f'Planning time: {duration*1000:.2f} ms')
    if result.result.result == 0: # SUCCESS
        pass

def report_stats(self):
    if len(self.cmd_timestamps) > 1:
        intervals = [self.cmd_timestamps[i] - self.cmd_timestamps[i-1] for i
in range(1, len(self.cmd_timestamps))]
        freq = 1.0 / statistics.mean(intervals) if intervals else 0.0
        self.get_logger().info(f'Average /cmd_vel frequency: {freq:.2f} Hz')
    if self.planning_times:
        avg = statistics.mean(self.planning_times) * 1000
        self.get_logger().info(f'Average planning time: {avg:.2f} ms')

def main(args=None):
    rclpy.init(args=args)
    analyzer = PerformanceAnalyzer()
    # Отправим несколько целей для теста
    import threading
    def send_goals():
        import time
        time.sleep(2) # ждём запуск системы
        for (x,y) in [(2.0, 0.0), (1.0, 2.0), (-1.0, 1.0)]:
            analyzer.send_goal(x, y)
            time.sleep(5.0) # пауза между запросами
        analyzer.report_stats()
    threading.Thread(target=send_goals).start()
    rclpy.spin(analyzer)
    analyzer.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Шаг 2. Запуск симуляции и навигации:

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=...
params_file:=nav2_params.yaml
```

Затем запускаем узел анализатора: `ros2 run performance_analyzer performance_analyzer_node`

Узел выполнит запросы и выведет среднее время планирования и частоту `/cmd_vel`.

Шаг 3. Изменение параметров costmap для оценки влияния. В `nav2_params.yaml` изменим `resolution` в секции `global_costmap` с 0.05 на 0.1:

```
global_costmap:
  ros__parameters:
    resolution: 0.1
```

После перезапуска Nav2 повторяем измерения. Сравниваем время планирования: при разрешении 0.1 оно должно уменьшиться, но возрастёт погрешность пути. Аналогично можно варьировать `inflation_radius`.

Шаг 4. Сравнение алгоритмов глобального планирования. По умолчанию используется SmacPlanner (A*). Заменяем на NavfnPlanner:

```
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false # Dijkstra
```

После перезапуска выполняем те же запросы целей и сравниваем среднее время. Узел `PerformanceAnalyzer` зафиксирует разницу.

Собранные метрики позволяют выбрать параметры, обеспечивающие своевременное планирование без излишней нагрузки на бортовой компьютер.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;

- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите узел анализатора производительности, соберите метрики для конфигурации по умолчанию. Запишите среднее время глобального планирования, частоту <code>/cmd_vel</code> и частоту <code>/local_plan</code> . Это будет базовый уровень.
2	Сравните время планирования SmacPlanner (A*) и NavfnPlanner с отключенным A* (<code>use_astar: false</code>) для трёх целей, расположенных на разных расстояниях. Постройте гистограмму.
3	Исследуйте влияние разрешения глобальной costmap. Установите <code>resolution: 0.025, 0.05, 0.1</code> . Для каждого разрешения измерьте среднее время планирования и размер карты (число ячеек). Объясните нелинейную зависимость.
4	Изменяйте радиус инфляции локальной costmap: 0.3, 0.55, 1.0. Замерьте частоту <code>/local_plan</code> и загрузку CPU (через <code>top</code> или <code>ps</code>). Определите максимально допустимый радиус, при котором частота не падает ниже 5 Гц.
5	Проведите нагрузочный тест: запустите параллельно три узла, одновременно запрашивающих <code>ComputePathToPose</code> с разными целями. Измерьте деградацию времени планирования относительно одного запроса.
6	Включите сглаживатель пути (<code>smoother_server</code>) и сравните время

	получения финального глобального пути с включенным и отключенным сглаживателем. Постройте график для 10 запросов.
7	Измерьте время, затрачиваемое на восстановительные поведения. Искусственно создайте ситуацию застревания (заблокируйте путь цилиндром) и зафиксируйте время от входа в состояние backup до его завершения.
8	Сравните производительность локальных планировщиков DWB и ТЕВ по частоте /cmd_vel и задержке публикации относительно /scan. Используйте ros2 topic delay.
9	Напишите скрипт для автоматического варьирования параметра max_vel_x от 0.1 до 0.5 с шагом 0.1. Для каждого значения измерьте частоту /cmd_vel и дальность прогноза локального планировщика. Представьте зависимости.
10	Отключите obstacle_layer в global_costmap (оставьте только static_layer). Измерьте время планирования и сравните с полной конфигурацией. Обсудите, в каких сценариях допустимо такое упрощение.
11	Ограничьте время планирования, установив max_planning_duration = 0.5 с. Проанализируйте, как часто планировщик не успевает найти путь и возвращает FAILURE. Запишите статистику успешных и провальных попыток.
12	Измерьте потребление памяти процессом planner_server при трёх значениях resolution. Используйте утилиту ps или ros2 node info. Свяжите потребление с количеством ячеек карты.
13	Создайте узел, сохраняющий все измеренные метрики в CSV-файл для последующей обработки в Python (pandas, matplotlib). Постройте графики времени планирования от номера запроса при разных конфигурациях.

14	Визуализируйте в реальном времени время планирования с помощью <code>rqt_plot</code> , подписавшись на самодельный топик, в который анализатор публикует измеренную длительность. Добейтесь обновления графика 1 Гц.
15	Разработайте <code>launch</code> -файл для автоматического регрессионного тестирования производительности: последовательно загружаются три различные карты (разных размеров), для каждой выполняется 5 запросов цели, собираются метрики. Результаты записываются в отчёт.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа `.docx` или `.pdf`.

Вопросы для защиты работы

1. Какие метрики используются для оценки навигации?
2. Как измерить время достижения цели?
3. Что такое длина пути?
4. Как использовать `rosvbag` для сбора данных?
5. Что такое CPU usage и как он влияет на навигацию?
6. Как сравнить два алгоритма планирования?
7. Каковы причины деградации производительности?
8. Как построить графики метрик с Python?

Лабораторная работа 27.

ОБРАБОТКА ДИНАМИЧЕСКИХ ПРЕПЯТСТВИЙ

Цель и содержание работы: Освоить методы обеспечения безопасной навигации робота в среде с движущимися объектами. Научиться добавлять динамические препятствия в симуляцию Gazebo, настраивать локальный планировщик Nav2 (DWB или TEB) для своевременного обнаружения и объезда, анализировать поведение робота при различных сценариях столкновения (встречное движение, пересечение курса, попутное движение). Получить практические навыки выбора параметров costmap и поведения при возникновении динамических препятствий.

Теоретическое обоснование

Обработка динамических препятствий в навигационном стеке Nav2 базируется на локальных картах затрат (local costmap), которые обновляются с высокой частотой на основе данных сенсоров (лазерный сканер, RGB-D камеры). При появлении движущегося объекта в поле зрения сенсора его проекция вносится в costmap как препятствие, и локальный планировщик на ближайших циклах пересчитывает траекторию, огибая запретную область. Ключевую роль играют параметры:

- observation_sources: определение источника данных (scan), гарантирующее, что движущийся объект будет отмечен как препятствие.
- clearing: true – препятствие удаляется из costmap, как только сенсор перестаёт его видеть; важно для динамических объектов.
- max_obstacle_height, min_obstacle_height – фильтрация по высоте для игнорирования пола и потолка.
- inflation_radius – область вокруг препятствия, которую робот не должен пересекать.

- Параметры локального планировщика (`sim_time`, `max_vel_x`, `acc_lim_x`) определяют способность робота экстренно затормозить или маневрировать.

В условиях динамических препятствий может потребоваться увеличить частоту обновления `local costmap` (`update_frequency`) и уменьшить `sim_time`, чтобы планировщик быстрее реагировал на изменения. При недостаточной реакции робот либо сталкивается, либо выполняет резкое торможение, что оценивается по графикам линейной скорости и траектории.

Пример

Рассмотрим типовой сценарий: в симуляции TurtleBot3 добавляется движущийся цилиндр, который пересекает путь робота. Для этого создаётся отдельный ROS-узел на Python, публикующий команды скорости для модели цилиндра, либо используется плагин Actor в Gazebo. Будем использовать узел-издатель на `/cmd_vel_moving_obstacle` и модель `turtlebot3`, замаскированную под препятствие (цилиндр). Чтобы не усложнять, можно добавить простую модель цилиндра в мир и управлять ею через топик.

Шаг 1. Добавление модели движущегося препятствия в Gazebo. В файл `turtlebot3_world.world` или свой мир вставим модель:

```
<model name='moving_cylinder'>
  <pose>1.0 2.0 0 0 0 0</pose>
  <link name='link'>
    <collision name='collision'>
      <geometry><cylinder radius='0.2' length='0.5'></geometry>
    </collision>
    <visual name='visual'>
      <geometry><cylinder radius='0.2' length='0.5'></geometry>
      <material><ambient>1 0 0 1</ambient></material>
    </visual>
  </link>
</model>
```

Управлять позицией цилиндра будем через ROS-узел, публикующий `geometry_msgs/TransformStamped` или сразу изменяющий модель через сервис Gazebo. Для простоты напомним узел, который публикует в топик `/moving_obstacle/cmd_vel`, а плагин `differential_drive` перемещает цилиндр.

Для этого в модель цилиндра надо добавить плагин libgazebo_ros_diff_drive.so. Альтернативно, можно использовать Gazebo напрямую через set_model_state.

Проще использовать gazebo_ros для перемещения: запускаем узел, который периодически вызывает сервис /gazebo/set_entity_state. Пример узла dynamic_obstacle.py:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from gazebo_msgs.srv import SetEntityState
from gazebo_msgs.msg import EntityState
import time

class DynamicObstacle(Node):
    def __init__(self):
        super().__init__('dynamic_obstacle')
        self.cli = self.create_client(SetEntityState,
'/gazebo/set_entity_state')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting for /gazebo/set_entity_state...')
        self.timer = self.create_timer(0.1, self.update_position)
        self.start_time = time.time()
        self.initial_x = 1.0
        self.initial_y = 2.0
        self.velocity = 0.3 # м/с вдоль оси y

    def update_position(self):
        elapsed = time.time() - self.start_time
        new_y = self.initial_y + self.velocity * elapsed
        if new_y > 4.0:
            new_y = 2.0 # зациклить
            self.start_time = time.time()
        req = SetEntityState.Request()
        req.state.name = 'moving_cylinder'
        req.state.pose.position.x = self.initial_x
        req.state.pose.position.y = new_y
        req.state.pose.position.z = 0.0
        req.state.pose.orientation.w = 1.0
        self.cli.call_async(req)

def main():
    rclpy.init()
    node = DynamicObstacle()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

Этот узел заставляет цилиндр двигаться по прямой, пересекая путь робота.

Шаг 2. Настройка локального планировщика для работы с динамическими препятствиями. В файле nav2_params.yaml нужно

убедиться, что `local_costmap` использует `observation_sources` с `clearing: True` и `mark: True`. Также важно установить `update_frequency` локальной `costmap` не менее 5 Гц, чтобы своевременно видеть движущийся объект. Дополнительно в параметрах локального планировщика можно уменьшить `sim_time` до 1.0, чтобы уменьшить инерционность прогноза, и увеличить `max_vel_theta` для быстрых манёвров.

Пример фрагмента параметров (секция `local_costmap`):

```
local_costmap:
  ros_parameters:
    update_frequency: 5.0
    publish_frequency: 2.0
    rolling_window: true
    width: 4
    height: 4
    resolution: 0.05
    plugins: ["voxel_layer", "inflation_layer"]
  voxel_layer:
    plugin: "nav2_costmap_2d::VoxelLayer"
    enabled: True
    footprint_clearing_enabled: True
    max_obstacle_height: 2.0
    observation_sources: "scan"
  scan:
    topic: "/scan"
    max_obstacle_height: 2.0
    clearing: True
    marking: True
    data_type: "LaserScan"
    min_obstacle_height: 0.0
  inflation_layer:
    plugin: "nav2_costmap_2d::InflationLayer"
    inflation_radius: 0.55
    cost_scaling_factor: 2.0
```

Шаг 3. Тестирование. Запускаем симуляцию `TurtleBot3` с добавленным цилиндром, `Nav2` с настроенными параметрами, узел `dynamic_obstacle`, `RViz`. Задаём цель для робота так, чтобы его траектория пересекалась с маршрутом цилиндра. Наблюдаем реакцию: робот должен затормозить или объехать движущееся препятствие, не столкнувшись. В логах можно отслеживать изменение скоростей. Сравниваем поведение с выключенным `clearing` или заниженной частотой `costmap`, где робот может не успеть среагировать.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте одиночное поперечно движущееся препятствие, как в примере. Проверьте реакцию робота при различных скоростях движения препятствия (0.1, 0.3, 0.5 м/с). Замерьте минимальное расстояние между роботом и препятствием в момент прохождения.
2	Настройте локальный планировщик с минимальным <code>inflation_radius</code> (0.2) и проверьте поведение робота при встречном движении объекта. Сравните с стандартным радиусом 0.55 по числу столкновений или резких торможений.
3	Отключите <code>clearing</code> в локальной <code>costmap</code> (<code>clearing: false</code>). Проверьте, как долго "фантом" препятствия сохраняется на карте после его ухода. Как это влияет на планирование пути? Предоставьте скриншоты <code>costmap</code> во времени.
4	Добавьте одновременно два движущихся объекта, движущихся по разным траекториям (например, по синусоиде и по прямой). Проверьте, может ли локальный планировщик найти проход между ними.
5	Используйте ТЕВ в качестве локального планировщика.

	Включите параметр <code>include_dynamic_obstacles</code> (или настройте <code>obstacle_poses_affected</code>). Сравните плавность объезда с DWB.
6	Реализуйте узел, который управляет движущимся препятствием, отслеживая позицию робота и намеренно двигаясь на пересечение его курса (агрессивное поведение). Проверьте, успевает ли робот избежать столкновения.
7	Варьируйте параметр <code>update_frequency</code> локальной <code>costmap</code> : 2 Гц, 5 Гц, 10 Гц. Для каждого случая замерьте время от появления препятствия в поле зрения до начала манёвра (используя топики <code>/scan</code> и <code>/cmd_vel</code>). Постройте график задержки.
8	Настройте поведение <code>recovery</code> : при обнаружении динамического препятствия слишком близко (менее 0.3 м) пусть робот выполняет экстренную остановку и ждёт 2 секунды. Реализуйте это через параметры или собственный ВТ-узел, добавленный в дерево поведения Nav2.
9	Создайте модель "пешехода", который движется по случайной траектории с изменением направления. Проверьте способность робота безопасно перемещаться в такой среде, измерив долю успешных доездов до цели без столкновений за 10 попыток.
10	Увеличьте размер робота (<code>footprint</code>) до 0.5x0.5 м. Как это сказывается на возможности объезда динамических препятствий? Сравните с штатным размером.
11	Подключите облако точек RGB-D камеры (<code>depthimage_to_laserscan</code> или напрямую <code>obstacle_layer</code> с <code>pointcloud</code>) и наблюдайте, как быстро появляется и исчезает динамическое препятствие по сравнению с однолучевым сканером. Проведите замеры.
12	Реализуйте прогнозирование движения динамического

	препятствия: напишите узел, который по последовательным наблюдениям оценивает скорость объекта и публикует предсказанную позицию в отдельную costmap. Интегрируйте это в локальный планировщик через дополнительные слои.
13	Проведите сравнительный анализ поведения DWB с включённым и выключенным критиком GoalAlign (изменение scale на 0). Как это влияет на способность объезжать движущиеся препятствия, сохраняя направление на цель?
14	Настройте локальную costmap так, чтобы она очищалась не только по данным лазера, но и с учётом времени (voxel_layer с таймаутом clearing). Сравните с режимом clearing: true.
15	Разработайте комплексный тестовый сценарий: робот должен доставить груз из точки А в точку Б, при этом на его пути появляются три динамических препятствия с разными сценариями (попутное, встречное, пересекающее). Запишите видео и подготовьте отчёт с рекомендациями по настройке планировщика для динамических сред.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое динамическое препятствие?
2. Как costmap обновляет информацию о динамических препятствиях?

3. Что такое clearing operation в costmap?
4. Как обнаружить движущееся препятствие по данным лидара?
5. Чем obstacle_layer отличается от voxel_layer?
6. Как настроить obstacle_range и raytrace_range?
7. Как симулировать движущееся препятствие в Gazebo?
8. Каковы ограничения Nav2 при высокой плотности препятствий?

Лабораторная работа 28.

ПОВТОРНОЕ ПЛАНИРОВАНИЕ (REPLANNING)

Цель и содержание работы: Изучить механизмы глобального повторного планирования пути в навигационном стеке ROS2 (Nav2). Научиться настраивать триггеры перепланирования, анализировать поведение робота при внезапном появлении статического препятствия на заранее известной карте, сравнивать работу с периодическим перепланированием и без него. Освоить мониторинг событий планировщика и измерение времени реакции на изменения среды.

Теоретическое обоснование

Глобальное перепланирование (replanning) — это построение нового глобального маршрута при изменении условий, делающих текущий путь недействительным или неоптимальным. В классическом стеке move_base за это отвечал параметр `planner_frequency`, задающий частоту повторного запуска глобального планировщика независимо от событий. В ROS2 Nav2 подход событийно-ориентированный: перепланирование инициируется деревом поведения при обнаружении невалидности пути.

Условиями для перепланирования служат:

- Появление новых статических препятствий в глобальной `costmap` (через `obstacle_layer`).
- Значительное отклонение робота от глобального пути (например, после длительного локального обхода препятствия).
- Изменение цели или получение новой команды.
- Явный вызов `ComputePathToPose` через `action`.

Периодическое перепланирование может быть реализовано декоратором `RateController` в BT, что позволяет гарантировать обновление пути с заданной частотой, даже если формально старый путь ещё проходим.

Это полезно в высокодинамичных средах, но увеличивает вычислительную нагрузку.

Ключевые компоненты, участвующие в replanning: глобальный планировщик (planner_server), сервер поведений (bt_navigator), глобальная costmap (global_costmap). Глобальная costmap должна быть настроена на приём данных о препятствиях (obstacle_layer с clearing), чтобы новые объекты оперативно попадали в неё. Частота обновления costmap (update_frequency) определяет, как быстро новое препятствие будет учтено. При обнаружении препятствия вблизи текущего глобального пути условие isPathValid возвращает FAILURE, и BT запускает новое ComputePathToPose.

Пример

Рассмотрим типовой сценарий: робот TurtleBot3 движется к цели по заранее известной карте. Во время движения в симуляции Gazebo появляется новое статическое препятствие (цилиндр), перегораживающее исходный путь. Наблюдается, как глобальный планировщик строит альтернативный маршрут.

Шаг 1. Создание файла параметров replanning_params.yaml с настроенной глобальной costmap, принимающей данные сканера как дополнительный слой препятствий, и деревом поведения, поддерживающим перепланирование при невалидности пути.

```
global_costmap:
  ros__parameters:
    global_frame: map
    robot_base_frame: base_footprint
    update_frequency: 3.0          # повышенная для быстрого обнаружения
    publish_frequency: 1.0
    rolling_window: false
    resolution: 0.05
    footprint: "[[0.15,0.15],[0.15,-0.15],[-0.15,-0.15],[-0.15,0.15]]"
    plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
    static_layer:
      plugin: "nav2_costmap_2d::StaticLayer"
      map_subscribe_transient_local: True
    obstacle_layer:
      plugin: "nav2_costmap_2d::ObstacleLayer"
      enabled: True
      observation_sources: "scan"
    scan:
      topic: /scan
```

```

        max_obstacle_height: 2.0
        clearing: True
        marking: True
        data_type: "LaserScan"
        min_obstacle_height: 0.0
inflation_layer:
    plugin: "nav2_costmap_2d::InflationLayer"
    inflation_radius: 0.55
    cost_scaling_factor: 2.0

planner_server:
    ros__parameters:
        planner_plugins: ["GridBased"]
        GridBased:
            plugin: "nav2_smac_planner/SmacPlanner"
            tolerance: 0.5
            max_planning_duration: 2.0
            smooth_path: true

bt_navigator:
    ros__parameters:
        default_bt_xml_filename: "navigate_to_pose_w_replanning_and_recovery.xml"
        plugin_lib_names: ["nav2_compute_path_to_pose_action_bt_node",
                           "nav2_follow_path_action_bt_node",
                           "nav2_back_up_action_bt_node",
                           "nav2_spin_action_bt_node",
                           "nav2_wait_action_bt_node",
                           "nav2_clear_costmap_service_bt_node",
                           "nav2_is_stuck_condition_bt_node",
                           "nav2_goal_reached_condition_bt_node",
                           "nav2_is_path_valid_condition_bt_node"] # важный узел
        global_frame: map
        robot_base_frame: base_footprint
        transform_tolerance: 0.2

controller_server:
    ros__parameters:
        controller_plugins: ["FollowPath"]
        FollowPath:
            plugin: "dwb_core::DWBLocalPlanner"
            # ... стандартные параметры DWB

```

Обратите внимание на включение узла isPathValid. Стандартное дерево navigate_to_pose_w_replanning_and_recovery.xml содержит логику: если путь невалиден, вызывается ComputePathToPose.

Шаг 2. Узел для динамического добавления препятствия. Напишем скрипт add_obstacle.py, использующий сервис Gazebo для создания цилиндра в заданной точке в определённый момент времени.

```

#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from gazebo_msgs.srv import SpawnEntity, DeleteEntity
import time
import threading

class ObstacleAdder(Node):
    def __init__(self):
        super().__init__('obstacle_adder')
        self.spawn_cli = self.create_client(SpawnEntity, '/spawn_entity')

```

```

self.delete_cli = self.create_client(DeleteEntity, '/delete_entity')
self.timer = self.create_timer(5.0, self.add_obstacle) # через 5 сек

def add_obstacle(self):
    self.timer.cancel()
    sdf = """<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="dynamic_obstacle">
    <static>true</static>
    <link name="link">
      <collision name="collision">
        <geometry><cylinder radius="0.3" length="0.5"/></geometry>
      </collision>
      <visual name="visual">
        <geometry><cylinder radius="0.3" length="0.5"/></geometry>
        <material><ambient>1 0 0 1</ambient></material>
      </visual>
    </link>
  </model>
</sdf>"""
    req = SpawnEntity.Request()
    req.name = "dynamic_obstacle"
    req.xml = sdf
    req.initial_pose.position.x = 1.5
    req.initial_pose.position.y = 0.0
    req.initial_pose.position.z = 0.0
    self.spawn_cli.call_async(req)
    self.get_logger().info("Obstacle added at (1.5, 0.0)")

def main():
    rclpy.init()
    node = ObstacleAdder()
    rclpy.spin(node)

if __name__ == '__main__':
    main()

```

Этот узел добавляет красный цилиндр на пути робота (цель робота находится дальше по прямой). При появлении цилиндра лазерный сканер обнаруживает его, глобальная costmap обновляется, и условие isPathValid перестаёт выполняться, что вызывает перепланирование.

Шаг 3. Запуск сценария и анализ. В трёх терминалах:

1. Симуляция: `export TURTLEBOT3_MODEL=waffle; ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py`

2. Навигация с нашими параметрами: `ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=/путь/к/карте.yaml params_file:=./replanning_params.yaml`

3. RViz: rviz2

Задаём цель роботу на расстояние ~3 м вперёд. После начала движения запускаем узел add_obstacle: `ros2 run <pkg> add_obstacle.py`. Через 5 секунд

цилиндр появляется. В RViz наблюдаем: глобальный путь (зелёная линия) сначала проходит через будущее препятствие, но после его обнаружения и обновления costmap глобальный планировщик запускается повторно, и путь перестраивается в обход. Локальный планировщик немедленно реагирует, но без нового глобального плана робот мог бы упереться в препятствие. Зафиксируем время от появления цилиндра до публикации нового /plan.

Шаг 4. Сравнение с отключённым перепланированием. Для этого можно временно заменить дерево на упрощённое без isPathValid (создав копию XML), либо закомментировать obstacle_layer в global_costmap, чтобы препятствие вообще не учитывалось глобально. Робот будет пытаться следовать старому пути, пока локальный планировщик не упрётся в цилиндр и не выполнит recovery. Сравните время достижения цели и плавность обхода.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте сценарий из примера. Замерьте интервал времени от появления препятствия до публикации нового /plan. Запишите

	базовое время реакции.
2	Увеличьте частоту обновления <code>global_costmap</code> до 10 Гц. Повторите замер. Объясните, как частота влияет на задержку перепланирования.
3	Настройте глобальный планировщик на периодическое перепланирование через ВТ, добавив декоратор <code>RateController(1.0)</code> к <code>ComputePathToPose</code> в дереве. Сравните нагрузку на CPU с событийной моделью.
4	Отключите <code>obstacle_layer</code> в <code>global_costmap</code> , оставив только <code>static_layer</code> . Запустите тот же сценарий. Объясните, почему робот не перепланирует глобальный путь и как справляется локальный планировщик.
5	Используйте сервис <code>/global_costmap/clear_except_region</code> для симуляции "забывания" препятствия. Проверьте, как быстро глобальный путь возвращается к исходному после исчезновения объекта.
6	Настройте параметр <code>inflation_radius</code> глобальной <code>costmap</code> на 0.2. Повторите сценарий и оцените, стал ли новый глобальный путь более агрессивным (ближе к препятствию).
7	Измените планировщик на <code>NavfnPlanner</code> с <code>use_astar: true</code> . Сравните время перепланирования и длину пути с <code>SmacPlanner</code> .
8	Добавьте два препятствия последовательно (с интервалом 3 с). Проверьте, инициируется ли перепланирование после каждого. Сколько раз публикуется новый <code>/plan</code> ?
9	Реализуйте узел, который принудительно вызывает перепланирование через action <code>ComputePathToPose</code> при обнаружении застревания робота (остановка на 2 с). Сравните со встроенной логикой <code>isStuck</code> .

10	Используйте параметр <code>replanning_period</code> (если доступен в вашей версии Nav2) или реализуйте его аналог через таймер в ВТ. Проверьте, как частое перепланирование (1 Гц) сказывается на плавности движения.
11	Создайте сценарий с открывающейся дверью: сначала проход закрыт, робот ждёт; через 10 с препятствие исчезает, и робот продолжает движение. Опишите, какую роль играет <code>clearing</code> в <code>costmap</code> .
12	Сравните <code>replanning</code> с разными глобальными планировщиками (<code>SmacPlanner</code> , <code>Theta*</code> , <code>Navfn</code>) в лабиринте, где после старта перекрывается один коридор. Оцените, какой планировщик быстрее находит новый путь.
13	Напишите анализатор, подписанный на <code>/plan</code> , который вычисляет степень отличия нового пути от предыдущего (метрика Хаусдорфа). Визуализируйте эту величину при появлении препятствия.
14	Включите параметр <code>allow_unknown</code> в <code>global_costmap</code> . Проверьте, будет ли робот планировать путь через неизвестные области после добавления препятствия, если свободного пространства нет.
15	Проведите стресс-тест: в цикле добавляйте и удаляйте препятствие каждые 2 секунды. Измерьте процент успешных перепланирований и стабильность навигации. Запишите видео и сформулируйте рекомендации по предельной динамике среды.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги

команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое replanning и когда он необходим?
2. Как Nav2 запускает повторное планирование?
3. Что такое controller_frequency?
4. Как настроить условие запуска нового плана?
5. Что такое recovery behaviors в Nav2?
6. Как реализовать стратегию при неудаче планирования?
7. Как отследить момент активации replanning?
8. Какие параметры влияют на агрессивность replanning?

Лабораторная работа 29.

ЗАПИСЬ И ВОСПРОИЗВЕДЕНИЕ ROSBAG

Цель и содержание работы: Освоить инструменты записи и воспроизведения данных в ROS2 с помощью rosbag2. Получить практические навыки захвата сообщений заданных топиков, их последующего воспроизведения в режиме симуляции времени, а также программного чтения bag-файлов для анализа и отладки. Научиться верифицировать поведение навигационных узлов на записанных сенсорных данных без повторного запуска робота или симулятора.

Теоретическое обоснование

Rosbag2 — штатное средство записи и воспроизведения сообщений в ROS2. Оно позволяет сериализовать произвольные топики в бинарный файл формата sqlite3 (или mcap) с сохранением временных меток и служебных заголовков. Запись инициируется командой `ros2 bag record`, воспроизведение — `ros2 bag play`. Для корректного воспроизведения данных с учетом временных задержек используется опция `--clock`, публикующая время из bag-файла в топик `/clock`. Все узлы, подписанные на `/clock` и настроенные на `use_sim_time:=true`, воспринимают это время как системное, что позволяет повторно исполнять алгоритмы в точности так, как они работали в момент записи.

Основные сценарии применения:

- отладка алгоритмов: запись проблемного сеанса и многократное воспроизведение с изменёнными параметрами;
- регрессионное тестирование: проверка, что новая версия стека не ухудшает поведение на эталонной записи;
- обучение восприятия: запись сырых сенсорных данных в реальном мире для тренировки нейросетей офлайн.

Фильтрация топиков при записи уменьшает размер файла: можно указать явный список (-t) или использовать регулярное выражение (-x). При воспроизведении также доступны опции ограничения длительности, скорости воспроизведения и заикливания.

Для программного анализа rosbag2 предоставляет Python API (rosbag2_py), позволяющее итерировать сообщения по временной шкале и десериализовать их в объекты ROS-сообщений.

Пример

Рассмотрим типовой сценарий: запись сеанса автономной навигации TurtleBot3 в Gazebo, затем её воспроизведение с работающим AMCL и Nav2 для отладки параметров локального планировщика.

Шаг 1. Запуск симуляции и навигации.

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=./map.yaml
params_file:=./nav2_params.yaml
```

В RViz задаём цель роботу, наблюдаем за движением.

Шаг 2. Запись всех топиков в bag-файл.

```
ros2 bag record -a -o turtlebot3_navigation_bag
```

Опция -a записывает все топики. После достижения роботом цели нажимаем Ctrl+C для остановки записи. В текущей директории создастся папка turtlebot3_navigation_bag (или с добавленным суффиксом даты).

Шаг 3. Анализ записанных данных.

```
ros2 bag info turtlebot3_navigation_bag
```

Вывод покажет список топиков, типы сообщений, общую длительность и размер. Допустим, нас интересуют только /odom, /scan и /cmd_vel. Можно сделать выборочную перезапись с фильтром:

```
ros2 bag record -o selected_bag -t /odom /scan /cmd_vel
```

Шаг 4. Воспроизведение bag-файла вместо живой симуляции.

Останавливаем симуляцию (Gazebo и Nav2). Запускаем AMCL, map_server и другие необходимые узлы в отдельном терминале с параметром use_sim_time:=true. Затем:

```
ros2 bag play turtlebot3_navigation_bag --clock
```

Параметр --clock указывает bag-плееру публиковать время в /clock, что синхронизирует таймеры узлов с записанным временем. В RViz видим, как робот (поза из /odom) движется, обновляются данные сканера, а Nav2 может строить путь и публиковать /cmd_vel. Это позволяет менять конфигурационные файлы и наблюдать реакцию планировщика на одни и те же сенсорные данные.

Шаг 5. Программное чтение bag-файла на Python.

Создадим скрипт read_bag.py для извлечения позы робота и построения траектории.

```
#!/usr/bin/env python3
import rclpy
from rclpy.serialization import deserialize_message
from rosbag2_py import SequentialReader, StorageOptions, ConverterOptions
from nav_msgs.msg import Odometry
import matplotlib.pyplot as plt

def main():
    storage_options = StorageOptions(uri='turtlebot3_navigation_bag',
storage_id='sqlite3')
    converter_options = ConverterOptions(input_serialization_format='cdr',
output_serialization_format='cdr')
    reader = SequentialReader()
    reader.open(storage_options, converter_options)

    x_vals, y_vals = [], []
    while reader.has_next():
        topic, data, timestamp = reader.read_next()
        if topic == '/odom':
            msg = deserialize_message(data, Odometry)
            x_vals.append(msg.pose.pose.position.x)
            y_vals.append(msg.pose.pose.position.y)

    plt.plot(x_vals, y_vals, '-o')
    plt.axis('equal')
    plt.xlabel('X'); plt.ylabel('Y')
    plt.title('Robot trajectory from bag')
    plt.grid(True)
    plt.show()

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

Установка зависимостей: `pip install matplotlib` (в виртуальном окружении ROS). Скрипт отобразит траекторию, записанную в одометрии. Аналогично можно извлекать данные лазера или команд скорости для сравнения с желаемым поведением.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Выполните полную запись сеанса навигации (запись всех топиков). Воспроизведите bag-файл при запущенном AMCL и map_server. Убедитесь, что в RViz робот отображается корректно. Зафиксируйте размер bag-файла и общее число сообщений.
2	При записи отфильтруйте только топики /scan, /odom и /tf. Сравните размер файла с полной записью. Воспроизведите, добавив Nav2. Работает ли планировщик с урезанным набором топиков? Объясните, почему отсутствие /tf_static может нарушить навигацию.
3	Запишите bag-файл с опцией --max-cache-size 5000000 (5 МБ). Проверьте, как запись разбивается на несколько файлов.

	Объедините их с помощью <code>ros2 bag convert</code> и проверьте целостность.
4	Запустите воспроизведение с параметром <code>--rate 0.5</code> (замедленное в два раза). Замерьте время прохождения роботом известного отрезка в реальном времени при записи и при замедленном воспроизведении. Соответствует ли оно ожидаемому?
5	Используйте <code>ros2 bag play --clock --loop</code> для зацикленного воспроизведения. Добейтесь, чтобы Nav2 непрерывно перепланировал путь на повторяющихся данных. Запишите второй bag с результатом.
6	Во время воспроизведения bag-файла меняйте параметр локального планировщика <code>max_vel_x</code> (через <code>ros2 param set</code>) и наблюдайте за изменениями в <code>/cmd_vel</code> . Сохраните график линейной скорости для трёх разных значений.
7	Напишите Python-скрипт, который читает bag-файл и вычисляет статистику: пройденное расстояние (интегрирование скорости), максимальную скорость, среднее значение датчиков. Визуализируйте гистограмму расстояний лазерного сканера.
8	Запишите bag-файл только с топиками <code>/scan</code> и <code>/tf</code> (без <code>/odom</code>). Попробуйте запустить AMCL. Объясните, почему локализация не работает или работает с ошибками, и какую роль играет одометрия.
9	Произведите две записи с разными значениями <code>inflation_radius</code> (0.2 и 0.8). С помощью программного анализа сравните среднее расстояние до препятствий в планируемом пути в обоих случаях.
10	Используйте <code>rosbag2_py</code> для извлечения всех сообщений <code>/scan</code> и записи их в CSV-файл с временными метками. Постройте 3D-график (время, угол, расстояние) интенсивности сканов.

11	Создайте узел, который при воспроизведении bag-файла слушает /clock и публикует в /diagnostics информацию о рассинхронизации (разница между /clock и системным временем). Проанализируйте задержки.
12	Запишите bag-файл с движущимся препятствием (из ЛР 27). Воспроизведите его пять раз с разными параметрами локального планировщика и определите наилучший с точки зрения минимального расстояния до препятствия.
13	Воспроизведите bag-файл в полностью автономном режиме: запустите RViz в конфигурации, показывающей карту и траекторию. Напишите скрипт, который автоматически подаёт цель через /navigate_to_pose и проверяет, достигнута ли она (по одометрии из bag).
14	Примените фильтрацию при воспроизведении: <code>ros2 bag play --exclude '/camera/*'</code> , чтобы исключить данные видеокамеры. Проверьте, что навигация не нарушена. Сравните нагрузку на CPU.
15	Разработайте полный тестовый сценарий: запуск Nav2 на воспроизводимых данных из bag-файла, автоматический прогон по трём заданным целям, запись результата в новый bag-файл и генерация отчёта о времени прохождения и плавности.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое rosbag и для чего он используется?
2. Как записать данные топиков в файл rosbag?
3. Как воспроизвести записанный rosbag?
4. Какие форматы поддерживает rosbag2?
5. Как выбрать только нужные топики для записи?
6. Как использовать rosbag для регрессионного тестирования?
7. Что такое rate при воспроизведении rosbag?
8. Как анализировать данные из rosbag с Python?

Лабораторная работа 30.

СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО COSTMAP LAYER

Цель и содержание работы: Изучить механизм расширения карты затрат (costmap) в навигационном стеке ROS2 Nav2 путём создания пользовательского слоя-плагина. Разработать и интегрировать собственный слой, добавляющий виртуальные препятствия в заданной области, скомпилировать его как динамическую библиотеку, подключить к глобальной или локальной costmap и проверить влияние на планирование пути в симуляции TurtleBot3.

Теоретическое обоснование

Costmap в Nav2 представляет собой многослойную структуру, где каждый слой независимо обновляет значения ячеек основной карты. Слои реализованы как плагины на основе класса `nav2_costmap_2d::Layer`. Основные методы, требующие переопределения в наследнике:

- `onInitialize()` – вызывается однократно, получает указатель на costmap, позволяет объявить параметры ROS.

- `updateBounds(double robot_x, double robot_y, double robot_yaw, double* min_x, double* min_y, double* max_x, double* max_y)` – пересчитывает границы области, которую слой намерен изменить (обновляет переданные min/max).

- `updateCosts(nav2_costmap_2d::Costmap2D& master_grid, int min_i, int min_j, int max_i, int max_j)` – вносит изменения в ячейки основной карты затрат, ограниченные индексами.

Слой может также реагировать на входящие сообщения через подписки внутри `onInitialize`.

Виртуальные препятствия – это искусственно добавляемые запретные области, которых в действительности нет, но которые робот должен

объезжать. Типичные примеры: зоны безопасности вокруг персонала, виртуальные стены (keep-out zones), динамически определяемые запретные маршруты. Плагин получает описание зон через параметры или топики и проставляет в соответствующих ячейках costmap максимальные значения (LETHAL_OBSTACLE = 254) или заданные пользователем.

Для создания плагина требуется описать класс, экспортировать его через макрос `PLUGINLIB_EXPORT_CLASS`, собрать как shared library и задекларировать в pluginlib-описании (XML). На стороне конфигурации Nav2 в YAML-файле слой подключается в список plugins соответствующей costmap с указанием типа плагина и его параметров.

Пример

Рассмотрим реализацию плагина VirtualWallLayer, который добавляет прямоугольную запретную зону, заданную параметрами: координаты центра (x, y), ширина, высота. Слой будет работать в глобальной costmap.

Шаг 1. Создание пакета virtual_wall_layer с зависимостями:

```
ros2 pkg create --build-type ament_cmake virtual_wall_layer --dependencies
rclcpp nav2_costmap_2d pluginlib
```

Шаг 2. Файл include/virtual_wall_layer/virtual_wall_layer.hpp:

```
#ifndef VIRTUAL_WALL_LAYER_HPP_
#define VIRTUAL_WALL_LAYER_HPP_

#include <rclcpp/rclcpp.hpp>
#include <nav2_costmap_2d/layer.hpp>
#include <nav2_costmap_2d/costmap_2d.hpp>

namespace virtual_wall_layer
{
class VirtualWallLayer : public nav2_costmap_2d::Layer
{
public:
    VirtualWallLayer();
    virtual ~VirtualWallLayer();
    virtual void onInitialize() override;
    virtual void updateBounds(double robot_x, double robot_y, double robot_yaw,
                             double* min_x, double* min_y,
                             double* max_x, double* max_y) override;
    virtual void updateCosts(nav2_costmap_2d::Costmap2D& master_grid,
                             int min_i, int min_j, int max_i, int max_j) override;
};
}
```

```

        virtual void activate() override;
        virtual void deactivate() override;
        virtual void reset() override;

private:
    double center_x_, center_y_, width_, height_;
    bool enabled_;
    rclcpp::Node::SharedPtr node_;
};
} // namespace virtual_wall_layer

#endif // VIRTUAL_WALL_LAYER_HPP_

```

Шаг 3. Файл src/virtual_wall_layer.cpp:

```

#include "virtual_wall_layer/virtual_wall_layer.hpp"
#include <pluginlib/class_list_macros.hpp>
#include <algorithm>

PLUGINLIB_EXPORT_CLASS(virtual_wall_layer::VirtualWallLayer,
nav2_costmap_2d::Layer)

using namespace virtual_wall_layer;

VirtualWallLayer::VirtualWallLayer() : enabled_(true) {}
VirtualWallLayer::~VirtualWallLayer() {}

void VirtualWallLayer::onInitialize()
{
    node_ = rclcpp::Node::make_shared("virtual_wall_layer_node_internal");
    // Объявление параметров с значениями по умолчанию
    node_>declare_parameter("center_x", 1.0);
    node_>declare_parameter("center_y", 0.0);
    node_>declare_parameter("width", 0.5);
    node_>declare_parameter("height", 0.5);
    node_>declare_parameter("enabled", true);

    // Чтение параметров
    center_x_ = node_>get_parameter("center_x").as_double();
    center_y_ = node_>get_parameter("center_y").as_double();
    width_ = node_>get_parameter("width").as_double();
    height_ = node_>get_parameter("height").as_double();
    enabled_ = node_>get_parameter("enabled").as_bool();

    current_ = true; // слой активен
    RCLCPP_INFO(node_>get_logger(), "VirtualWallLayer initialized");
}

void VirtualWallLayer::updateBounds(double robot_x, double robot_y, double
robot_yaw,
double* min_x, double* min_y, double* max_x,
double* max_y)
{
    if (!enabled_) return;

    double half_w = width_ / 2.0;
    double half_h = height_ / 2.0;
    // Расширяем границы, чтобы включить прямоугольник
    *min_x = std::min(*min_x, center_x_ - half_w);
    *min_y = std::min(*min_y, center_y_ - half_h);
    *max_x = std::max(*max_x, center_x_ + half_w);

```

```

    *max_y = std::max(*max_y, center_y_ + half_h);
}

void VirtualWallLayer::updateCosts(nav2_costmap_2d::Costmap2D& master_grid,
                                   int min_i, int min_j, int max_i, int max_j)
{
    if (!enabled_) return;

    double half_w = width_ / 2.0;
    double half_h = height_ / 2.0;
    double origin_x = layered_costmap_->getCostmap()->getOriginX();
    double origin_y = layered_costmap_->getCostmap()->getOriginY();
    double resolution = layered_costmap_->getCostmap()->getResolution();

    for (int j = min_j; j < max_j; ++j) {
        for (int i = min_i; i < max_i; ++i) {
            double wx, wy;
            layered_costmap_->getCostmap()->indexToWorld(i, j, wx, wy);
            if (wx >= center_x_ - half_w && wx <= center_x_ + half_w &&
                wy >= center_y_ - half_h && wy <= center_y_ + half_h) {
                unsigned char cost = nav2_costmap_2d::LETHAL_OBSTACLE;
                master_grid.setCost(i, j, cost);
            }
        }
    }
}

void VirtualWallLayer::activate() { enabled_ = true; }
void VirtualWallLayer::deactivate() { enabled_ = false; }
void VirtualWallLayer::reset() {}

```

Шаг 4. Файл src/plugin_description.xml для экспорта плагина:

```

<library path="virtual_wall_layer">
  <class name="virtual_wall_layer/VirtualWallLayer"
        type="virtual_wall_layer::VirtualWallLayer"
        base_class_type="nav2_costmap_2d::Layer">
    <description>Custom layer adding a virtual rectangular
obstacle</description>
  </class>
</library>

```

Шаг 5. CMakeLists.txt:

```

cmake_minimum_required(VERSION 3.8)
project(virtual_wall_layer)

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(nav2_costmap_2d REQUIRED)
find_package(pluginlib REQUIRED)

include_directories(include)

add_library(${PROJECT_NAME} SHARED
    src/virtual_wall_layer.cpp
)

```

```

ament_target_dependencies(${PROJECT_NAME}
  rclcpp
  nav2_costmap_2d
  pluginlib
)

# Установка заголовков
install(DIRECTORY include/ DESTINATION include/${PROJECT_NAME})

# Установка библиотек
install(TARGETS ${PROJECT_NAME}
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)

# Установка описания плагина
install(FILES src/plugin_description.xml
  DESTINATION share/${PROJECT_NAME}
)

ament_export_include_directories(include)
ament_export_libraries(${PROJECT_NAME})
ament_export_dependencies(rclcpp nav2_costmap_2d pluginlib)

# Регистрация плагина
pluginlib_export_plugin_description_file(nav2_costmap_2d
src/plugin_description.xml)

ament_package()

```

Шаг 6. package.xml должен содержать зависимости от nav2_costmap_2d, pluginlib, rclcpp. Например:

```

<package format="3">
  <name>virtual_wall_layer</name>
  <version>0.1.0</version>
  <description>Custom costmap layer for virtual obstacles</description>
  <maintainer email="user@example.com">User</maintainer>
  <license>Apache-2.0</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>nav2_costmap_2d</depend>
  <depend>pluginlib</depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>

```

Шаг 7. Сборка пакета:

```

colcon build --packages-select virtual_wall_layer
source install/setup.bash

```

Шаг 8. Подключение слоя в конфигурацию глобальной costmap. В YAML-файле параметров Nav2 добавляем в global_costmap в секцию plugins новый плагин после static_layer, но до inflation_layer. Пример:

```
global_costmap:
  ros__parameters:
    plugins: ["static_layer", "virtual_wall_layer", "obstacle_layer",
"inflation_layer"]
    virtual_wall_layer:
      plugin: "virtual_wall_layer/VirtualWallLayer"
      center_x: 1.5
      center_y: 0.5
      width: 0.6
      height: 0.6
      enabled: true
```

Важно: порядок слоёв важен. Слои обрабатываются последовательно, виртуальная стена должна применяться после статической карты и до инфляции, чтобы инфляция учла новое препятствие.

Шаг 9. Тестирование в симуляции.

```
Запуск TurtleBot3 в Gazebo:
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Запуск навигации с изменённым YAML-файлом:

```
ros2 launch nav2_bringup bringup_launch.py use_sim_time:=true map:=./map.yaml
params_file:=./nav2_params_with_virtual_wall.yaml
```

В RViz задаём цель за виртуальной стеной. Глобальный планировщик должен построить путь в обход прямоугольника. Если стена перекрывает единственный проход, планировщик вернёт неудачу. Слой также можно временно отключить через параметр enabled (ros2 param set).

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;

- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте слой виртуального кругового препятствия (радиус задаётся параметром). Разместите его в центре коридора и проверьте, как планировщик его огибает. Сравните траекторию с прямоугольным препятствием из примера.
2	Добавьте возможность задания нескольких зон через массив параметров (например, список центров и размеров). Слой должен одновременно добавлять все препятствия. Протестируйте с тремя кругами.
3	Реализуйте динамическое изменение положения виртуального препятствия: подпишитесь в слое на топики <code>geometry_msgs/msg/Point</code> , и при получении нового сообщения обновляйте координаты центра. Продемонстрируйте, как глобальный план перестраивается в реальном времени.
4	Создайте слой «запретная зона вокруг робота»: слой добавляет виртуальное препятствие в виде окружности радиуса 1 м вокруг текущей позы робота (читайте одометрию из внутреннего буфера). Проверьте, что робот не может подъехать близко к самому себе (это не имеет смысла для одного робота, но демонстрирует концепцию зоны безопасности).
5	Реализуйте слой, который считывает координаты виртуальных стен из файла (CSV) и строит соответствующие препятствия в

	costmap. Формат: x1,y1,x2,y2 для каждой стены. Стены могут быть наклонными.
6	Создайте слой, который по данным лазерного сканера строит «зоны повышенной осторожности»: если в секторе много точек, он добавляет LETHAL в этой области, но с задержкой (имитация прогноза). Проверьте в динамической среде.
7	Интегрируйте пользовательский слой в локальную costmap, а не в глобальную. Понаблюдайте, как локальный планировщик реагирует на виртуальное препятствие непосредственно перед роботом. Сравните влияние на плавность.
8	Реализуйте слой, который инвертирует функцию: добавляет «разрешённую зону» (все ячейки вне зоны становятся LETHAL, а внутри — FREE). Это виртуальный коридор, по которому робот должен двигаться. Проверьте планирование.
9	Добавьте поддержку dynamic_reconfigure (rclcpp параметров) для изменения размеров/положения препятствия без перезапуска. Покажите изменение стены в реальном времени через rqt_reconfigure.
10	Создайте слой, который получает список виртуальных препятствий из топика типа nav_msgs/msg/GridCells и обновляет costmap. Напишите отдельный узел-публикатор для тестирования.
11	Реализуйте слой, который постепенно «заращивает» ячейки вокруг виртуального препятствия со временем (имитация медленного увеличения зоны опасности). Используйте таймер внутри слоя для расширения зоны.
12	Сравните поведение робота с виртуальной стеной в глобальной costmap и тем же препятствием, физически добавленным в

	Gazebo. Оцените разницу в прохождении поворотов и задержках.
13	Разработайте слой, который добавляет виртуальное препятствие в форме эллипса с ориентацией (параметры: длины полуосей и угол). Проверьте в симуляции.
14	Реализуйте слой, который считывает полигон из GeoJSON (через параметр) и заполняет его LETHAL. Используйте библиотеку для проверки попадания точки в полигон.
15	Произведите стресс-тест: создайте слой, который быстро перемещает виртуальное препятствие (через подписку), и измерьте, насколько часто планировщик успевает перестраивать путь. Запишите частоту replanning'a.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое costmap layer и как слои объединяются?
2. Как создать пользовательский слой?
3. Какой интерфейс необходимо реализовать?
4. Как зарегистрировать новый слой в Nav2?
5. Какие данные доступны в методе updateCosts?
6. Как кастомный слой взаимодействует с топиками?
7. Для каких задач полезны пользовательские слои?
8. Как отлаживать работу кастомного слоя?

Лабораторная работа 31.

ИНТЕГРАЦИЯ С КОМПЬЮТЕРНЫМ ЗРЕНИЕМ (ДЕТЕКЦИЯ ОБЪЕКТОВ)

Цель и содержание работы: Освоить интеграцию данных компьютерного зрения в систему управления мобильным роботом в ROS2. Научиться захватывать изображения с симулированной RGB-камеры в Gazebo, обрабатывать их средствами OpenCV для обнаружения объектов заданного цвета, публиковать события детекции и изменять поведение робота (остановка, объезд, оповещение). Получить практический опыт связывания восприятия и управления в едином навигационном стеке.

Теоретическое обоснование

Интеграция технического зрения в робототехнические системы позволяет роботу реагировать на визуальные стимулы, недоступные лазерным дальномерам: цветные метки, дорожные знаки, сигнальные конусы. Процесс включает несколько этапов:

- Получение изображений: симулированная камера в Gazebo публикует топик типа `sensor_msgs/msg/Image`.
- Предобработка: конвертация из ROS-сообщения в формат OpenCV (`cv_bridge`), перевод в цветовое пространство HSV, фильтрация шумов.
- Детекция: пороговая обработка по цвету (`inRange`), выделение связанных областей, фильтрация по площади.
- Генерация события: при обнаружении объекта публикуется сообщение в специальный топик (`std_msgs/msg/Bool`, `vision_msgs/msg/Detection2DArray` или пользовательский тип).
- Реакция: узел управления подписывается на событие и изменяет поведение: останавливается, включает сигнал тревоги, строит объезд через модификацию локальной `costmap` или переключение состояний.

В ROS2 передача изображений стандартизирована, а библиотека cv_bridge обеспечивает совместимость с OpenCV. Для цветовой детекции часто применяется фильтрация в HSV, устойчивая к изменениям освещения (в симуляции это не критично, но полезно). Логика поведения может быть реализована через конечный автомат, дерево поведения или прямое изменение параметров локального планировщика.

Пример

Рассмотрим типовой сценарий: робот TurtleBot3 оснащён RGB-камерой (модель waffle уже имеет камеру). В симуляции перед роботом размещается красный конус. Узел-детектор распознаёт красные объекты в поле зрения. При обнаружении публикуется true в топик /detection_event. Узел управления движением подписан на событие: в штатном режиме робот движется к фиксированной цели, а при появлении объекта останавливается, ждёт 3 секунды и продолжает движение.

Шаг 1. Подготовка симуляции. Запускаем мир TurtleBot3 с красным конусом (можно добавить модель в файл мира или через spawn). Для простоты используем стандартный turtlebot3_world и вручную добавим красный цилиндр через отдельный скрипт или встроенные модели. Альтернативно, разместим красный объект в Gazebo: в окне Gazebo выберем Insert -> Cylinder, зададим красный цвет и поместим перед роботом.

Шаг 2. Установка пакетов для работы с изображениями:

```
sudo apt install ros-humble-cv-bridge ros-humble-image-transport python3-opencv
```

Шаг 3. Узел детекции (detector_node.py):

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import Bool
import cv2
```

```

from cv_bridge import CvBridge

class ColorDetector(Node):
    def __init__(self):
        super().__init__('color_detector')
        self.subscription = self.create_subscription(Image, '/camera/image_raw',
self.image_callback, 10)
        self.detection_pub = self.create_publisher(Bool, '/detection_event', 10)
        self.bridge = CvBridge()
        self.object_detected = False

    def image_callback(self, msg):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        except Exception as e:
            self.get_logger().error(f'CV Bridge error: {e}')
            return

        # Преобразование в HSV и поиск красного цвета
        hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
        # Диапазон красного в HSV (два интервала)
        lower_red1 = (0, 100, 100)
        upper_red1 = (10, 255, 255)
        lower_red2 = (160, 100, 100)
        upper_red2 = (180, 255, 255)
        mask1 = cv2.inRange(hsv, lower_red1, upper_red1)
        mask2 = cv2.inRange(hsv, lower_red2, upper_red2)
        mask = cv2.bitwise_or(mask1, mask2)

        # Морфологические операции для удаления шумов
        mask = cv2.erode(mask, None, iterations=2)
        mask = cv2.dilate(mask, None, iterations=2)

        # Поиск контуров
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        detected = False
        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area > 500: # минимальная площадь в пикселях
                detected = True
                break

        # Публикация события при изменении состояния
        if detected != self.object_detected:
            self.object_detected = detected
            msg = Bool()
            msg.data = detected
            self.detection_pub.publish(msg)
            self.get_logger().info(f'Detection: {detected}')

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

if __name__ == '__main__':
    main()

```

Шаг 4. Узел управления (behavior_controller.py):

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from std_msgs.msg import Bool
import time

class BehaviorController(Node):
    def __init__(self):
        super().__init__('behavior_controller')
        self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.detection_sub = self.create_subscription(Bool, '/detection_event',
self.detection_callback, 10)
        self.timer = self.create_timer(0.1, self.control_loop)
        self.detected = False
        self.stop_start_time = None
        self.stop_duration = 3.0 # секунд ожидания

    def detection_callback(self, msg):
        self.detected = msg.data
        if self.detected:
            self.stop_start_time = time.time()
            self.get_logger().info('Object detected! Stopping...')

    def control_loop(self):
        cmd = Twist()
        if self.detected:
            cmd.linear.x = 0.0
            cmd.angular.z = 0.0
            # Проверка, истекло ли время ожидания
            if self.stop_start_time and (time.time() - self.stop_start_time) >
self.stop_duration:
                self.detected = False # Сбрасываем флаг для продолжения
движения
                self.get_logger().info('Resuming motion')
            else:
                # Движение вперёд к фиксированной цели
                cmd.linear.x = 0.2
                cmd.angular.z = 0.0
                self.cmd_pub.publish(cmd)

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

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

Шаг 5. Запуск системы.

Терминал 1: симуляция

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

В Gazebo добавляем красный объект (цилиндр) на пути робота.

Терминал 2: детектор

```
ros2 run image_processing detector_node.py
```

Терминал 3: контроллер

```
ros2 run image_processing behavior_controller.py
```

Робот начинает движение. Когда камера видит красный объект, детектор публикует True, контроллер останавливает робота, ждёт 3 секунды и продолжает. Если объект исчезает из поля зрения, движение возобновляется сразу.

Шаг 6. Интеграция с Nav2 (опционально). Вместо собственного контроллера можно использовать дерево поведения Nav2, добавив условие проверки топика /detection_event и действие остановки. Это потребует создания ВТ-узлов, но приведённый подход проще для понимания основ.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Используйте детектор синего цвета. Разместите синий конус и проверьте реакцию. Объясните, какие диапазоны HSV были выбраны.
2	Реализуйте детектор на основе контуров и цветовых гистограмм: объект считается обнаруженным, если цветовая гистограмма в

	выделенной области близка к эталонной (метод сравнения корреляцией). Проверьте на нескольких цветах.
3	Вместо полной остановки реализуйте объезд: при обнаружении красного конуса робот должен повернуть на месте на 90 градусов влево, проехать 0.5 м, затем вернуться на курс к цели. Интегрируйте в контроллер простую логику.
4	Добавьте звуковое оповещение: при обнаружении объекта узел управления публикует сообщение в топик /speaker (или использует play из системного вызова). Сделайте однократное предупреждение.
5	Опубликуйте координаты обнаруженного объекта (координаты центра ограничивающей рамки) в топик /detection_bbox. Визуализируйте эти рамки в RViz с помощью Marker. Убедитесь, что маркер накладывается на объект в сцене.
6	Используйте YOLO (через пакет yolo_ros или ultralytics) для обнаружения объектов класса "traffic cone" или "person". Сравните с цветовым детектором по устойчивости к освещению.
7	Настройте динамическую реакцию: чем ближе объект (по размеру рамки), тем сильнее замедляется робот (пропорциональное управление скоростью). Реализуйте плавную остановку.
8	Интегрируйте детекцию в Nav2 как слой costmap: напишите узел, который при обнаружении препятствия визуально публикует в локальную costmap дополнительные запретные ячейки через сервис или топик. Проверьте, планировщик строит объезд.
9	Создайте детектор мигающего света (анализ последовательности кадров). При обнаружении мигания (например, аварийного сигнала) робот должен остановиться и ждать, пока мигание не

	прекратится.
10	Реализуйте детекцию дорожных знаков (заранее подготовьте шаблоны знаков "стоп" или "уступи дорогу" с помощью template matching). При обнаружении знака «стоп» робот полностью останавливается на 2 секунды.
11	Объедините данные камеры и лазерного сканера: спроецируйте лазерный скан на изображение и проверьте, что обнаруженный объект подтверждается дальномером. Повысьте надёжность детекции.
12	Запустите три робота: один – оператор, публикующий изображения, два других – использующие общую детекцию для синхронной реакции (розыгрыш). Визуализируйте общую сцену.
13	Реализуйте детектор на основе нейросети (MobileNet SSD) из OpenCV DNN. Преобразуйте модель в формат, поддерживаемый ROS2, и запустите инференс на CPU. Сравните частоту кадров с цветовым детектором.
14	Напишите сервис /enable_detection, позволяющий дистанционно включать/выключать реакцию на визуальные стимулы. Интегрируйте его в контроллер.
15	Проведите серию экспериментов с разной скоростью робота и расстоянием до объекта. Измерьте время реакции от появления объекта в кадре до начала торможения. Постройте график и предложите рекомендации по максимальной допустимой скорости.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги

команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как передаются данные компьютерного зрения в ROS 2?
2. Что такое sensor_msgs/Image?
3. Как интегрировать детектор (YOLO) с ROS-узлом?
4. Что такое vision_msgs?
5. Как публиковать bounding boxes через ROS 2?
6. Как синхронизировать камеру и лидар?
7. Что такое cv_bridge?
8. Как использовать результаты детекции для навигации?

Лабораторная работа 32.

МНОГОАГЕНТНАЯ СИСТЕМА: ДВА РОБОТА

Цель и содержание работы: Освоить запуск и настройку многоагентной системы в ROS2 с использованием симуляции Gazebo на примере двух мобильных роботов TurtleBot3. Научиться изолировать топики, сервисы и системы координат с помощью пространств имён (namespaces), настраивать уникальные идентификаторы для каждого робота. Реализовать координацию движения, при которой один робот ведомый следует за лидирующим, и изучить базовые проблемы многоагентных систем (конфликты имён, коллизии, синхронизация).

Теоретическое обоснование

В многоагентных робототехнических системах несколько автономных или полуавтономных роботов оперируют в общей среде. ROS2 предоставляет встроенные механизмы для разделения ресурсов:

- Пространство имён (namespace) – логическая группировка топиков, сервисов и параметров. Каждый робот запускается в собственном пространстве имён (например, /robot1, /robot2), благодаря чему топики /cmd_vel, /odom и другие становятся уникальными (/robot1/cmd_vel, /robot2/cmd_vel).

- Префикс системы координат (tf_prefix) или задание параметров frame_id в конфигурационных файлах позволяет изолировать tf-фреймы (robot1/base_footprint, robot2/base_footprint).

- В Gazebo для каждого робота необходимо запустить отдельный узел-драйвер с уникальным именем робота в параметрах.

При запуске двух однотипных роботов без надлежащей изоляции произойдёт конфликт топиков, что приведёт к путанице в управлении и

локализации. После изоляции каждый робот может независимо принимать команды и сообщать о своём состоянии.

Координация поведения достигается путём обмена данными через общие топики (например, поза лидера) или через централизованный планировщик. В простейшем случае ведомый робот подписывается на одометрию лидера и вычисляет целевую точку на фиксированном расстоянии позади него, затем применяет регулятор для движения к этой точке. Это требует корректной трансформации координат и предотвращения столкновений за счёт контроля дистанции.

Пример

Рассмотрим типовой сценарий: в симуляции Gazebo запускаются два TurtleBot3 waffle в одном мире. Первый робот (robot1) является лидером и выполняет автономную миссию (движение по нескольким путевым точкам). Второй робот (robot2) — ведомый, который должен следовать за лидером на безопасной дистанции 1 метр, избегая столкновений с ним и с окружением. Для простоты используем программируемое движение лидера по прямой, а follower реализуем отдельным узлом.

Шаг 1. Создание launch-файла для запуска двух роботов. Файл multi_turtlebot3_launch.py:

```
import os
from launch import LaunchDescription
from launch.actions import ExecuteProcess, RegisterEventHandler, EmitEvent
from launch.event_handlers import OnProcessExit
from launch.events import Shutdown
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Переменные окружения
    env = {'TURTLEBOT3_MODEL': 'waffle'}

    # Запуск первого робота в namespace /robot1
    robot1 = ExecuteProcess(
        cmd=['ros2', 'launch', 'turtlebot3_gazebo',
            'turtlebot3_world.launch.py',
            'robot_name:=robot1', 'x_pose:=0.0', 'y_pose:=0.0'],
        env=env,
        output='screen'
    )
```

```

        # Запуск второго робота в namespace /robot2
        robot2 = ExecuteProcess(
            cmd=['ros2', 'launch', 'turtlebot3_gazebo',
                'turtlebot3_world.launch.py',
                'robot_name:=robot2', 'x_pose:=2.0', 'y_pose:=1.0'],
            env=env,
            output='screen'
        )

    return LaunchDescription([robot1, robot2])

```

Примечание: `turtlebot3_gazebo` поддерживает аргументы `robot_name`, `x_pose`, `y_pose` для запуска нескольких роботов. Каждый робот получит namespace `/robot_name` и `tf`-префикс. Если стандартный `launch` не поддерживает, можно модифицировать `world` и параметры.

Шаг 2. Узел движения лидера. Файл `leader.py`:

```

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

class Leader(Node):
    def __init__(self):
        super().__init__('leader', namespace='robot1')
        self.pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.timer = self.create_timer(0.1, self.move)
        self.start_time = self.get_clock().now()

    def move(self):
        cmd = Twist()
        # Простое движение вперёд в течение 20 секунд, затем поворот
        elapsed = (self.get_clock().now() - self.start_time).nanoseconds / 1e9
        if elapsed < 20.0:
            cmd.linear.x = 0.2
            cmd.angular.z = 0.0
        else:
            cmd.linear.x = 0.0
            cmd.angular.z = 0.5 # разворот на месте
        self.pub.publish(cmd)

def main():
    rclpy.init()
    node = Leader()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Шаг 3. Узел ведомого. Файл `follower.py`:

```

#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
import math

```

```

class Follower(Node):
    def __init__(self):
        super().__init__('follower', namespace='robot2')
        self.lead_om_sub = self.create_subscription(Odometry,
            '/robot1/odom', self.lead_cb, 10)
        self.self_om_sub = self.create_subscription(Odometry, '/robot2/odom',
            self.self_cb, 10)
        self.cmd_pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.lead_x = 0.0
        self.lead_y = 0.0
        self.self_x = 0.0
        self.self_y = 0.0
        self.distance = 1.0 # желаемая дистанция позади лидера
        self.timer = self.create_timer(0.1, self.control)

    def lead_cb(self, msg):
        self.lead_x = msg.pose.pose.position.x
        self.lead_y = msg.pose.pose.position.y

    def self_cb(self, msg):
        self.self_x = msg.pose.pose.position.x
        self.self_y = msg.pose.pose.position.y

    def control(self):
        # Вычисляем направление от ведомого к лидеру
        dx = self.lead_x - self.self_x
        dy = self.lead_y - self.self_y
        dist = math.hypot(dx, dy)
        if dist < 0.1:
            return
        # Целевая позиция: точка на distance позади лидера по прямой от ведомого
        # Упрощённо: двигаемся к лидеру, пока расстояние не станет равным
distance
        if dist > self.distance:
            speed = min(0.2, 0.5 * (dist - self.distance))
            target_angle = math.atan2(dy, dx)
            # Для простоты используем угол только для поворота
            # Получаем ориентацию из своей одометрии (если нужно)
            # но у нас нет текущего угла, возьмём из self_om
            # В полном коде нужно извлечь ориентацию, здесь упростим
            cmd = Twist()
            cmd.linear.x = speed
            cmd.angular.z = 0.0 # в реальности нужен регулятор угла
            self.cmd_pub.publish(cmd)

def main():
    rclpy.init()
    node = Follower()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Этот код не идеален, но демонстрирует концепцию. В реальном задании потребуется доработать управление угловой скоростью на основе разницы между текущим курсом и направлением на цель.

Шаг 4. Тестирование. Запускаем launch-файл для двух роботов, затем leader.py и follower.py. В RViz можно подключить оба namespaces и наблюдать за движением. При правильной настройке robot2 следует за robot1.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Реализуйте следование с сохранением точной дистанции 1.0 м позади лидера (по направлению движения лидера). Используйте историю поз лидера для вычисления его траектории. Сравните с простым движением к текущей позиции.
2	Добавьте третьего робота, который тоже следует за лидером, но сбоку (эшелон). Настройте пространство имён robot3. Проверьте отсутствие коллизий.
3	Реализуйте обмен информацией о препятствиях: первый робот сканирует пространство, и если обнаруживает препятствие, уходит от него, а второй повторяет его манёвр, читая только его траекторию. Покажите преимущество.
4	Вместо фиксированной траектории лидера запустите для него

	навигационный стек Nav2 с целью. Ведомый должен следовать за лидером, используя его одометрию. Проверьте работу в мире с препятствиями.
5	Реализуйте механизм предотвращения столкновений между роботами: если ведомый приближается к лидеру слишком близко (менее 0.5 м), он останавливается. Проверьте с резким торможением лидера.
6	Добавьте визуализацию в RViz: отобразите маркеры, показывающие желаемую позицию ведомого относительно лидера, и текущую траекторию обоих роботов.
7	Реализуйте алгоритм "конвоя" (platooning): несколько роботов (2-3) следуют цепочкой, каждый ориентируется на впереди идущего. Лидер может двигаться по заранее заданному маршруту. Исследуйте накопление ошибки позиционирования.
8	Настройте систему, в которой роботы обмениваются планами (топик /plan) и договариваются о приоритете проезда в узком коридоре с помощью простого протокола (например, запрос/ответ).
9	Запустите двух роботов с независимыми навигационными стеками Nav2. Задайте им пересекающиеся цели и убедитесь, что они не сталкиваются благодаря локальным планировщикам. Оцените задержки.
10	Реализуйте симуляцию патрулирования: два робота движутся по одной замкнутой траектории в противоположных направлениях. При встрече они должны обмениваться позициями и расходиться правыми бортами.
11	Создайте узел-диспетчер, который назначает цели роботам и следит за их выполнением, перераспределяя задачи в случае

	отказа. Используйте service/action.
12	Добавьте в сценарий движущееся препятствие (человек). Один робот должен уклоняться и сообщать об этом второму, чтобы тот заранее скорректировал путь.
13	Реализуйте формирование "строя" (клином или шеренгой) с заданными относительными координатами. Лидер движется к цели, ведомые поддерживают строй. Используйте систему координат лидера.
14	Настройте запись rosbag для всей системы и проанализируйте время задержки передачи одометрии от лидера к ведомому. Предложите меры по её уменьшению.
15	Проведите сравнение производительности при запуске двух роботов в одной Gazebo симуляции и в двух отдельных экземплярах Gazebo (связанных через ROS2). Измерьте частоту симуляции и реальное время.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое многоагентная система?
2. Как разделить namespace для нескольких роботов?
3. Как настроить Nav2 для каждого робота?
4. Что такое namespaced topic?

5. Как координировать движение нескольких роботов?
6. Как организовать коммуникацию между агентами?
7. Что такое `tf_prefix`?
8. Каковы ограничения Nav2 для нескольких роботов?

Лабораторная работа 33.

ПРОСТАЯ РЕАЛИЗАЦИЯ SLAM (GMAPPING)

Цель и содержание работы: Освоить методику одновременной локализации и построения карты (SLAM) в ROS2 с использованием пакета `slam_toolbox`. Научиться запускать симуляцию робота с лидаром, управлять им в режиме телеоперации, наблюдать за процессом построения карты в реальном времени и сохранять полученную карту для дальнейшего использования в навигации.

Теоретическое обоснование

SLAM (Simultaneous Localization and Mapping) – фундаментальная задача мобильной робототехники, в которой робот должен одновременно оценивать собственное положение в пространстве и строить карту неизвестного окружения, используя только бортовые сенсоры (одометрию и лазерный дальномер). Математически задача формулируется как вероятностная оценка совместного распределения позы робота и карты признаков или Occupancy Grid по последовательности измерений и управляющих воздействий. В ROS2 широко применяется пакет `slam_toolbox`, реализующий графовый SLAM: позы робота представляются узлами графа, а связи между ними – ограничения, полученные из сопоставления последовательных сканов (scan matching) и данных одометрии. При замыкании петли (loop closure) глобальная оптимизация графа (например, с помощью `g2o` или `Ceres`) исправляет накопленную ошибку. В отличие от классического GMapping (фильтр частиц), графовый подход обеспечивает более высокую точность и масштабируемость. Ключевые параметры: разрешение карты, минимальное расстояние/угол для добавления нового узла, метод scan matching (ICP, correlative). После завершения

картографирования карта сохраняется утилитой `map_saver` в формате `pgm/yaml`.

Пример

Рассмотрим типовой сценарий построения карты мира `turtlebot3_world` с помощью `slam_toolbox`. Используется симуляция Gazebo, робот TurtleBot3 с моделью `waffle`, оснащённый лидаром.

Шаг 1. Установка необходимых пакетов (если не установлены):

```
sudo apt install ros-humble-slam-toolbox ros-humble-turtlebot3-gazebo ros-humble-teleop-twist-keyboard
```

Шаг 2. Запуск симуляции:

```
export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Шаг 3. Запуск `slam_toolbox`. Для работы в синхронном режиме с визуализацией создадим файл параметров `slam_params.yaml` (на основе стандартного):

```
slam_toolbox:
  ros__parameters:
    # Режим синхронного онлайн SLAM
    solver_plugin: solver_plugins::CeresSolver
    ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
    ceres_preconditioner: SCHUR_JACOBI
    ceres_trust_strategy: LEVENBERG_MARQUARDT
    ceres_dogleg_type: TRADITIONAL_DOGLEG
    ceres_loss_function: None

    # Скорость публикации карты
    map_update_interval: 5.0
    resolution: 0.05

    # Параметры одометрии
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    use_scan_matching: true
    use_scan_barycenter: true
    minimum_time_interval: 0.5
    minimum_travel_distance: 0.2
    minimum_travel_heading: 0.1

    # Loop closure
    loop_closure: true
    loop_search_maximum_distance: 3.0
    loop_match_minimum_chain_size: 10
    loop_match_maximum_variance_coarse: 3.0
    loop_match_minimum_response_coarse: 0.35
```

```
loop_match_minimum_response_fine: 0.45
```

```
Запуск:  
ros2 launch slam_toolbox online_sync_launch.py params_file:=./slam_params.yaml  
use_sim_time:=true
```

Шаг 4. Телеоперация. В отдельном терминале запустим узел клавиатурного управления:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r  
/cmd_vel:=/cmd_vel
```

Перемещаем робота по территории, стараясь охватить все помещения. В RViz (запуск: rviz2) добавляем дисплей Map (топик /map) и Odometry, наблюдаем формирование карты.

Шаг 5. Сохранение карты. По завершении обхода останавливаем teleop (Ctrl+C). Сохраняем карту:

```
ros2 run nav2_map_server map_saver_cli -f ~/my_map
```

Будут созданы файлы my_map.yaml и my_map.pgm. Для проверки качества можно отобразить карту: ros2 run nav2_util map_server --ros-args -p yaml_filename:=./my_map.yaml.

Шаг 6. (Опционально) Для сравнения можно запустить GMapping, установив ros-humble-slam-gmapping, но основное задание ориентировано на slam_toolbox.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Выполните базовый сценарий из примера. Оцените время построения полной карты, её размер в пикселях и объём файла <code>pgm</code> . Сравните с картой, предоставляемой в <code>turtlebot3_gazebo</code> .
2	Сравните онлайн и офлайн режимы <code>slam_toolbox</code> . В онлайн режиме постройте карту, затем запустите офлайн SLAM на основе записанного <code>rosbag</code> 'а того же сеанса. Сравните полученные карты метрикой схожести (визуально, по перекрытию).
3	Измените разрешение карты <code>resolution</code> в файле параметров: 0.02, 0.05, 0.1. Для каждого запустите SLAM и сохраните карту. Оцените детализацию и занимаемую память. Постройте график зависимости размера карты от разрешения.
4	Внесите искусственный шум в одометрию (например, создайте узел, добавляющий гауссов шум в <code>/odom</code>). Сравните качество карты (прямолинейность стен, смыкание петель) при наличии шума и без него. Объясните, как SLAM компенсирует ошибки одометрии.
5	Установите пакет <code>slam_gmapping</code> (если доступен) и запустите GMapping с параметрами по умолчанию. Сравните скорость построения, нагрузку на CPU и качество замыкания петли с <code>slam_toolbox</code> . Представьте таблицу.
6	Запустите симуляцию в мире <code>turtlebot3_house</code> (дом с несколькими комнатами и длинными коридорами). Включите <code>loop closure</code> в <code>slam_toolbox</code> и выключите его (<code>loop_closure: false</code>). Покажите разницу в карте при возвращении робота в исходную точку. Оцените ошибку положения до и после замыкания петли.

7	Настройте параметр <code>minimum_travel_distance</code> (дистанция для добавления нового узла): 0.1, 0.5, 1.0 м. Замерьте количество узлов в графе (можно посмотреть в логах или через сервис) и частоту обновления карты. Найдите оптимальное значение для компромисса точность/нагрузка.
8	Включите <code>use_scan_matching</code> : true и false. При выключенном <code>scan matching</code> 'е SLAM опирается только на одометрию. Сравните траекторию робота (из <code>/map</code> -> <code>/odom</code> трансформации) с траекторией, полученной из одометрии. Оцените дрейф.
9	Смоделируйте «похищение робота»: во время движения телепортируйте робота через сервис Gazebo <code>set_entity_state</code> в новую позицию. Зафиксируйте, как <code>slam_toolbox</code> обрабатывает такое событие: возникает ли новая карта, перестраивается ли граф. Опишите поведение.
10	Запишите <code>rosvbag</code> 'ом процесс движения робота (топики <code>/scan</code> , <code>/odom</code> , <code>/tf</code>). Воспроизведите <code>bag</code> -файл, запустив <code>slam_toolbox</code> в офлайн режиме, и сравните карту с онлайн-версией. Измерьте время обработки в офлайн режиме.
11	Используя сохранённую карту, запустите навигационный стек Nav2 (AMCL + планировщики). Проверьте, может ли робот автономно перемещаться по построенной карте. Зафиксируйте успешность навигации и точность локализации.
12	Сравните два метода сопоставления сканов в <code>slam_toolbox</code> : корреляционный (CORRELATIVE) и ICP (доступные через настройки <code>solver</code>). Оцените их быстродействие и способность замыкать петли в коридорах с однотипной геометрией.
13	Измените параметр <code>minimum_time_interval</code> (0.1, 0.5, 1.0 с). Измерьте, как это влияет на частоту создания узлов и на

	плавность построения карты. Постройте график зависимости размера графа от временного порога.
14	Запустите два робота одновременно в одном мире (по аналогии с ЛР 32) с разными namespaces. Запустите для каждого свой экземпляр slam_toolbox (в разных пространствах имён). Проверьте, не мешают ли они друг другу. Опишите особенности multi-robot SLAM в данной конфигурации.
15	Напишите скрипт на Python, который автоматически управляет роботом по траектории «квадрат» или «змейка», чтобы полностью покрыть доступное пространство. В конце скрипт вызывает сохранение карты. Сравните качество автоматически построенной карты с ручным управлением.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Что такое SLAM и каковы его основные задачи?
2. Как работает алгоритм GMapping?
3. Какие данные необходимы для работы GMapping?
4. Что такое частицы в particle filter SLAM?
5. Как SLAM строит карту при движении?
6. Как сохранить карту через map_saver?
7. Какова роль одометрии и лидара в SLAM?

8. В чём разница между offline и online SLAM?

Лабораторная работа 34.

КОМПЛЕКСНЫЙ ПРОЕКТ: НАВИГАЦИЯ К ЦЕЛИ С ОБЪЕЗДОМ ПРЕПЯТСТВИЙ

Цель и содержание работы: Интегрировать все изученные компоненты навигационного стека ROS2 в единую автономную систему. Практически собрать и настроить полный контур управления: локализация (AMCL), карта (map_server), глобальный и локальный планировщики, слои затрат с учётом статических и динамических препятствий. Провести комплексное тестирование в симуляционной среде с препятствиями, включая случайные, и оценить качество навигации по метрикам успешности, времени и плавности движения.

Теоретическое обоснование

Автономная навигация мобильного робота в среде с препятствиями достигается согласованной работой следующих подсистем:

- map_server предоставляет статическую карту эталонного окружения.
- AMCL (Adaptive Monte Carlo Localization) оценивает позу робота, сопоставляя данные лидара с картой и одометрией.
- global_costmap объединяет статическую карту, динамические препятствия от сенсоров и виртуальные зоны; служит основой для глобального планировщика.
- global_planner (SmacPlanner с гибридным A*) строит оптимальный по стоимости путь в сетке занятости от текущего положения до цели.
- local_costmap в скользящем окне отслеживает ближайшие препятствия.
- local_planner (DWB или TEB) генерирует управляющие команды скоростей, следуя глобальному пути и избегая препятствий.

- Сервер поведений восстанавливает движение при застреваниях (spin, backup, wait).

Все компоненты объединяются через launch-файл, который загружает YAML-конфигурацию параметров. Цель задаётся через RViz или программно через action NavigateToPose.

Пример

Рассмотрим типовой сценарий комплексного проекта: симуляция TurtleBot3 Waffle в мире turtlebot3_world с добавлением случайных препятствий (цилиндров). Робот должен проехать из стартовой позиции в целевую. Интеграция осуществляется с помощью единого launch-файла.

Шаг 1. Подготовка конфигурации параметров. Создадим файл nav2_integration_params.yaml, основанный на стандартной поставке, но с оптимизированными параметрами для динамических препятствий.

Ключевые секции:

```
amcl:
  ros__parameters:
    use_sim_time: true
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    global_frame_id: "map"
    laser_likelihood_max_dist: 2.0
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "odom"
    robot_model_type: "differential"
    set_initial_pose: true
    update_min_d: 0.2
    update_min_a: 0.2
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05

bt_navigator:
  ros__parameters:
    use_sim_time: true
    default_bt_xml_filename: "navigate_to_pose_w_replanning_and_recovery.xml"
    bt_xml_filename: ""
    plugin_lib_names: ["nav2_compute_path_to_pose_action_bt_node",
                      "nav2_follow_path_action_bt_node",
                      "nav2_back_up_action_bt_node",
                      "nav2_spin_action_bt_node",
```

```

        "nav2_wait_action_bt_node",
        "nav2_clear_costmap_service_bt_node",
        "nav2_is_stuck_condition_bt_node",
        "nav2_goal_reached_condition_bt_node",
        "nav2_initial_pose_received_condition_bt_node",
        "nav2_are_error_codes_active_condition_bt_node",
        "nav2_is_battery_low_condition_bt_node",
        "nav2_is_path_valid_condition_bt_node"]

controller_server:
  ros__parameters:
    use_sim_time: true
    controller_plugins: ["FollowPath"]
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: true
      min_vel_x: 0.0
      max_vel_x: 0.26
      max_vel_theta: 0.5
      min_speed_xy: 0.1
      acc_lim_x: 0.8
      acc_lim_theta: 1.0
      vx_samples: 20
      vtheta_samples: 40
      sim_time: 1.5
      critics: ["RotateToGoal", "Oscillation", "GoalAlign", "GoalDist",
"PathAlign", "PathDist", "ObstacleFootprint"]
      GoalAlign.scale: 24.0
      GoalDist.scale: 24.0
      PathAlign.scale: 16.0
      PathDist.scale: 16.0
      ObstacleFootprint.scale: 1.0
      Oscillation.scale: 10.0
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0

planner_server:
  ros__parameters:
    use_sim_time: true
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_smac_planner/SmacPlanner"
      tolerance: 0.5
      downsample_costmap: false
      allow_unknown: true
      max_iterations: -1
      max_on_approach_iterations: 1000
      max_planning_duration: 2.0
      analytic_expansion_ratio: 3.0
      analytic_expansion_max_length: 2.0
      smooth_path: true
      minimum_turning_radius: 0.4
      lookup_table_size: 20.0

smoother_server:
  ros__parameters:
    use_sim_time: true
    smoother_plugins: ["simple_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 0.001

behavior_server:
  ros__parameters:
    use_sim_time: true
    local_costmap_topic: "local_costmap/costmap_raw"
    global_costmap_topic: "global_costmap/costmap_raw"
    footprint_topic: "local_costmap/published_footprint"
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "wait"]

```

```

    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    wait:
      plugin: "nav2_behaviors/Wait"

local_costmap:
  ros__parameters:
    use_sim_time: true
    global_frame: "odom"
    robot_base_frame: "base_footprint"
    update_frequency: 5.0
    publish_frequency: 2.0
    rolling_window: true
    width: 4
    height: 4
    resolution: 0.05
    footprint: "[[0.15, 0.15], [0.15, -0.15], [-0.15, -0.15], [-0.15, 0.15]]"
    plugins: ["voxel_layer", "inflation_layer"]
    voxel_layer:
      plugin: "nav2_costmap_2d::VoxelLayer"
      enabled: True
      footprint_clearing_enabled: True
      max_obstacle_height: 2.0
      observation_sources: "scan"
      scan:
        topic: "/scan"
        max_obstacle_height: 2.0
        clearing: True
        marking: True
        data_type: "LaserScan"
        min_obstacle_height: 0.0
    inflation_layer:
      plugin: "nav2_costmap_2d::InflationLayer"
      inflation_radius: 0.55
      cost_scaling_factor: 2.0

global_costmap:
  ros__parameters:
    use_sim_time: true
    global_frame: "map"
    robot_base_frame: "base_footprint"
    update_frequency: 2.0
    publish_frequency: 1.0
    rolling_window: false
    resolution: 0.05
    footprint: "[[0.15, 0.15], [0.15, -0.15], [-0.15, -0.15], [-0.15, 0.15]]"
    plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
    static_layer:
      plugin: "nav2_costmap_2d::StaticLayer"
      map_subscribe_transient_local: True
    obstacle_layer:
      plugin: "nav2_costmap_2d::ObstacleLayer"
      enabled: True
      observation_sources: "scan"
      scan:
        topic: "/scan"
        max_obstacle_height: 2.0
        clearing: True
        marking: True
        data_type: "LaserScan"
        min_obstacle_height: 0.0
    inflation_layer:
      plugin: "nav2_costmap_2d::InflationLayer"
      inflation_radius: 0.55
      cost_scaling_factor: 2.0

map_server:
  ros__parameters:

```

```
use_sim_time: true
yaml_filename: "/путь/к/вашей/карте.yaml"
```

Шаг 2. Launch-файл, запускающий все компоненты вместе с симулятором. Файл `comprehensive_nav.launch.py`:

```
import os
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, ExecuteProcess, TimerAction
from launch.launch_description_sources import PythonLaunchDescriptionSource
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Симуляция с выбранным миром (можно добавить аргумент)
    turtlebot3_gazebo_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('turtlebot3_gazebo'),
                        'launch/turtlebot3_world.launch.py')
        ),
        launch_arguments={ 'robot_name': 'waffle' }.items()
    )

    # Навигационный стек
    nav2_bringup_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('nav2_bringup'),
                        'launch/bringup_launch.py')
        ),
        launch_arguments={
            'use_sim_time': 'true',
            'params_file': '/path/to/nav2_integration_params.yaml',
            'map': '/path/to/map.yaml'
        }.items()
    )

    # Узел для добавления случайных препятствий через 10 секунд после старта
    spawn_obstacles = ExecuteProcess(
        cmd=['ros2', 'run', 'integration_project',
            'random_obstacle_spawner.py'],
        output='screen'
    )

    return LaunchDescription([
        turtlebot3_gazebo_launch,
        nav2_bringup_launch,
        TimerAction(period=10.0, actions=[spawn_obstacles])
    ])
```

Шаг 3. Скрипт `random_obstacle_spawner.py` для добавления нескольких цилиндров в случайных позициях (пакет `integration_project`):

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from gazebo_msgs.srv import SpawnEntity, DeleteEntity
import random
import math
```

```

class RandomObstacleSpawner(Node):
    def __init__(self):
        super().__init__('random_obstacle_spawner')
        self.spawn_cli = self.create_client(SpawnEntity, '/spawn_entity')
        while not self.spawn_cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting for /spawn_entity service...')
        self.spawn_obstacles()

    def spawn_obstacles(self, count=5, world_size=4.0):
        for i in range(count):
            x = random.uniform(-world_size, world_size)
            y = random.uniform(-world_size, world_size)
            # избегаем зоны старта робота (0,0)
            while math.hypot(x, y) < 1.5:
                x = random.uniform(-world_size, world_size)
                y = random.uniform(-world_size, world_size)
            radius = random.uniform(0.15, 0.3)
            sdf = f"""<?xml version="1.0" ?>
<sdf version="1.6">
    <model name="random_obs_{i}">
        <static>true</static>
        <link name="link">
            <collision name="collision">
                <geometry><cylinder
length="0.5"/></geometry>
                    radius="{radius}"
                </collision>
                <visual name="visual">
                    <geometry><cylinder
length="0.5"/></geometry>
                        radius="{radius}"
                    <material><ambient>1 0 0 1</ambient></material>
                </visual>
            </link>
        </model>
    </sdf>"""
            req = SpawnEntity.Request()
            req.name = f'random_obs_{i}'
            req.xml = sdf
            req.initial_pose.position.x = x
            req.initial_pose.position.y = y
            req.initial_pose.position.z = 0.0
            future = self.spawn_cli.call_async(req)
            self.get_logger().info(f'Spawed obstacle {i} at ({x:.2f},
{y:.2f})')

def main():
    rclpy.init()
    node = RandomObstacleSpawner()
    rclpy.spin_once(node, timeout_sec=1.0)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Шаг 4. Запуск RViz с правильной конфигурацией:

```

rviz2 -d $(ros2 pkg prefix
nav2_bringup)/share/nav2_bringup/rviz/nav2_default_view.rviz

```

Шаг 5. Задание цели и наблюдение.

В RViz используем инструмент "2D Goal Pose", устанавливаем цель на противоположной стороне карты. Робот начинает движение. Глобальный

планировщик прокладывает путь в обход статических стен, локальный планировщик огибает случайные цилиндры. При необходимости срабатывают восстановительные поведения. Успешное достижение фиксируется сообщением в терминале.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Запустите базовый сценарий с 5 случайными препятствиями. Зафиксируйте время достижения цели, длину пройденного пути (интегрируя одометрию) и минимальное расстояние до препятствий.
2	Увеличьте количество случайных препятствий до 15. Проведите 10 заездов, меняя начальную и целевую точки. Подсчитайте процент успешных завершений. Оцените необходимость recovery-поведений.
3	Замените глобальный планировщик на NavfnPlanner (Dijkstra) и сравните с SmacPlanner по длине пути и времени планирования для трёх сценариев.
4	Сравните локальные планировщики DWB и TEB в условиях

	комплекса с препятствиями. Оцените плавность (по производной угловой скорости) и количество активаций recovery.
5	Настройте виртуальные запретные зоны через пользовательский costmap-слой (из ЛР 30). Разместите зону, перекрывающую половину прохода. Проверьте, что робот выбирает другой маршрут.
6	Реализуйте узел-монитор, который подписывается на /plan и /local_plan и публикует статистику (время планирования, частоту обновления, длину). Отобразите её в rqt_plot.
7	Добавьте динамическое препятствие (движущийся цилиндр) по аналогии с ЛР 27. Проверьте, как робот объезжает его при движении к цели. Запишите видео.
8	Запустите комплексную систему в среде с узким коридором (создайте свою карту). Измените footprint робота на больший и оцените, сможет ли он протиснуться, не нарушая safety.
9	Реализуйте автоматическую постановку цели через скрипт, который циклически задаёт 5 целей, ожидая достижения каждой. Измерьте суммарное время и среднюю скорость.
10	Настройте параметры AMCL (уменьшите число частиц до 300, увеличьте до 5000). Оцените среднеквадратичную ошибку локализации (сравнивая с ground truth из /ground_truth_odom) для этих двух режимов.
11	Используйте запись rosbag (ЛР 29) сеанса с комплексным сценарием. Воспроизведите и добейтесь идентичного поведения навигации при различных скоростях воспроизведения.
12	Интегрируйте компьютерное зрение: перед роботом разместите цветной сигнал (красный – стоп, зелёный – старт). Узел-детектор публикует статус, и навигация включается только при зелёном

	сигнале.
13	Организируйте многоагентный сценарий (два робота). Первый выполняет миссию, второй должен следовать за ним, не сталкиваясь и объезжая общие препятствия.
14	Проведите стресс-тест: непрерывная навигация в течение 30 минут со случайными препятствиями, запись всех метрик. Постройте гистограмму времени выполнения отдельных маршрутов и выявите условия отказов.
15	Разработайте собственное поведение recovery (через ВТ или кастомный плагин), которое вместо стандартного backup выполняет разворот на 180 градусов и следует по пройденному пути назад. Сравните эффективность со стандартным recovery.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как организовать полный цикл навигации?
2. Как объединить SLAM, AMCL и Nav2?
3. Как запустить несколько узлов через launch-файл?
4. Что такое параметрический launch-файл?
5. Как обеспечить надёжный объезд препятствий?
6. Как логировать ключевые события навигации?
7. Как тестировать проект на серии маршрутов?

8. Какие метрики оценивают итоговую систему?

Лабораторная работа 35.

ПРОЕКТ: ДЕРЕВО ПОВЕДЕНИЯ ДЛЯ КОМПЛЕКСНОЙ МИССИИ

Цель и содержание работы: Разработать комплексную поведенческую логику автономного мобильного робота с использованием деревьев поведения (Behavior Tree) на базе библиотеки BehaviorTree.CPP в ROS2. Реализовать миссию последовательного объезда трёх целевых точек с паузой и подачей сигнала на каждой, а также параллельный мониторинг заряда батареи, обеспечивающий экстренный возврат на базу. Освоить композицию узлов Sequence, Fallback, ReactiveFallback и Parallel для создания гибких, устойчивых к сбоям систем управления.

Теоретическое обоснование

Дерево поведения (Behavior Tree) — иерархическая модель управления, в которой логика представляется ориентированным деревом. Основные типы узлов: композиты (Sequence, Fallback, Parallel), декораторы (Retry, Timeout, Reactive) и листья (Action, Condition). При каждом тике узлы возвращают статус SUCCESS, FAILURE или RUNNING.

- Sequence выполняет детей по порядку до первого FAILURE; возвращает SUCCESS, если все дети успешны.
- Fallback выполняет детей до первого SUCCESS; возвращает FAILURE, если все вернули FAILURE.
- ReactiveFallback действует как Fallback, но при этом, если во время выполнения текущего дочернего узла условие более высокого приоритета становится истинным, то активный узел прерывается и управление переключается.
- Parallel с правилом «успех при первом успехе» позволяет одновременно выполнять несколько ветвей, например, основную миссию и мониторинг батареи.

Для обмена данными между узлами используется Blackboard — ассоциированная с деревом таблица пар ключ-значение.

Преимущества деревьев поведения перед конечными автоматами: модульность, переиспользуемость поддеревьев, упрощённое добавление новых условий и действий без переопределения переходов, а также естественная поддержка реактивности.

Пример

Рассмотрим реализацию миссии на роботе TurtleBot3 в симуляции Gazebo. Используются пакеты: rclcpp, behaviortree_cpp, geometry_msgs, nav_msgs, std_msgs. Узлы дерева управляют движением через публикацию команд в /cmd_vel на основе одометрии, а сигнал реализован через топик /signal.

Шаг 1. Создание ROS2-пакета mission_bt с зависимостями:

```
ros2 pkg create --build-type ament_cmake mission_bt --dependencies rclcpp
std_msgs geometry_msgs nav_msgs behaviortree_cpp
```

Шаг 2. Разработка Action-узла MoveToWaypoint, который принимает координаты цели из Blackboard и публикует команды скорости, пока расстояние до цели не станет меньше порога. Файл include/mission_bt/move_to_waypoint.hpp:

```
#ifndef MOVE_TO_WAYPOINT_HPP
#define MOVE_TO_WAYPOINT_HPP

#include <behaviortree_cpp/action_node.h>
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <nav_msgs/msg/odometry.hpp>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <cmath>

namespace mission_bt {

class MoveToWaypoint : public BT::StatefulActionNode {
public:
    MoveToWaypoint(const std::string& name, const BT::NodeConfiguration& config)
        : BT::StatefulActionNode(name, config) {}
};

}
```

```

static BT::PortsList providedPorts() {
    return {BT::InputPort<std::string>("waypoint")};
}

BT::NodeStatus onStart() override {
    // Получение строки вида "x,y"
    std::string wp_str;
    if (!getInput("waypoint", wp_str)) {
        RCLCPP_ERROR(rclcpp::get_logger("MoveToWaypoint"), "No waypoint
specified");
        return BT::NodeStatus::FAILURE;
    }
    size_t comma = wp_str.find(',');
    if (comma == std::string::npos) return BT::NodeStatus::FAILURE;
    target_x_ = std::stod(wp_str.substr(0, comma));
    target_y_ = std::stod(wp_str.substr(comma + 1));

    // Инициализация издателя cmd_vel и подписки на odom, если еще не сделано
    if (!node_) {
        node_ = rclcpp::Node::make_shared("move_to_wp_node");
        cmd_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel",
10);

        odom_sub_ = node_->create_subscription<nav_msgs::msg::Odometry>(
            "/odom", 10,
            [this](const nav_msgs::msg::Odometry::SharedPtr msg) {
                current_x_ = msg->pose.pose.position.x;
                current_y_ = msg->pose.pose.position.y;
            });
    }
    return BT::NodeStatus::RUNNING;
}

BT::NodeStatus onRunning() override {
    rclcpp::spin_some(node_);
    double dx = target_x_ - current_x_;
    double dy = target_y_ - current_y_;
    double dist = std::hypot(dx, dy);
    if (dist < 0.2) {
        // Остановка
        auto cmd = geometry_msgs::msg::Twist();
        cmd_pub_>publish(cmd);
        return BT::NodeStatus::SUCCESS;
    }
    // Простейший П-регулятор
    double angle_to_target = std::atan2(dy, dx);
    double heading_error = angle_to_target; // Упрощение: курс робота не читаем,
но можем считать, что он уже примерно направлен
    // В реальности нужно получить текущий курс из одометрии, здесь для
краткости опустим,
    // добавив получение yaw через tf2. Допустим, робот ориентирован по
направлению движения.
    auto cmd = geometry_msgs::msg::Twist();
    cmd.linear.x = std::min(0.2, 0.5 * dist);
    cmd.angular.z = 0.8 * heading_error;
    cmd_pub_>publish(cmd);
    return BT::NodeStatus::RUNNING;
}

void onHalted() override {
    auto cmd = geometry_msgs::msg::Twist();
    cmd_pub_>publish(cmd);
}

private:
    rclcpp::Node::SharedPtr node_;
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_pub_;
    rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_sub_;

```

```

        double target_x_, target_y_;
        double current_x_ = 0.0, current_y_ = 0.0;
    };

    } // namespace mission_bt
#endif

```

Шаг 3. Action-узел WaitAction (пауза) и SignalAction (публикация сигнала). Их реализация проста. Файл include/mission_bt/wait_action.hpp:

```

#include <behaviortree_cpp/action_node.h>
#include <chrono>
#include <thread>

class WaitAction : public BT::StatefulActionNode {
public:
    WaitAction(const std::string& name) : BT::StatefulActionNode(name, {}) {}
    BT::NodeStatus onStart() override {
        start_ = std::chrono::steady_clock::now();
        return BT::NodeStatus::RUNNING;
    }
    BT::NodeStatus onRunning() override {
        if (std::chrono::steady_clock::now() - start_ >= std::chrono::seconds(3))
            return BT::NodeStatus::SUCCESS;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        return BT::NodeStatus::RUNNING;
    }
    void onHalted() override {}
private:
    std::chrono::steady_clock::time_point start_;
};

```

Шаг 4. Условие GoalReached (проверяет расстояние до точки, используя Blackboard). Здесь можно использовать те же данные одометрии, но проще передавать статус через Blackboard, обновляемый в MoveToWaypoint. Однако для демонстрации реализуем простую проверку на основе последней команды. Вместо этого в дереве будем сразу после MoveToWaypoint ставить паузу и сигнал, полагая, что если MoveToWaypoint завершился успехом, то цель достигнута.

Шаг 5. Условие BatteryLow и действие ReturnToBase. Используем Blackboard-ключ "battery". BatteryLow проверяет, что значение меньше 0.3. ReturnToBase — аналогично MoveToWaypoint, но с фиксированной базовой точкой (0,0).

Шаг 6. XML-описание дерева (файл bt_mission.xml):

```
<root>
  <BehaviorTree ID="MainTree">
    <ReactiveFallback name="top_level">
      <!-- Ветка экстренного возврата: если батарея низкая, возвращаемся на базу -->
      <Sequence name="low_battery_seq">
        <BatteryLow/>
        <ReturnToBase/>
      </Sequence>
      <!-- Основная миссия: последовательное выполнение трёх точек -->
      <Sequence name="mission">
        <!-- Точка 1 -->
        <Fallback name="wp1">
          <Sequence>
            <MoveToWaypoint waypoint="2.0,1.0"/>
            <WaitAction/>
            <SignalAction message="Reached WP1"/>
          </Sequence>
        </Fallback>
        <!-- Точка 2 -->
        <Fallback name="wp2">
          <Sequence>
            <MoveToWaypoint waypoint="3.0,-1.0"/>
            <WaitAction/>
            <SignalAction message="Reached WP2"/>
          </Sequence>
        </Fallback>
        <!-- Точка 3 -->
        <Fallback name="wp3">
          <Sequence>
            <MoveToWaypoint waypoint="1.0,-2.0"/>
            <WaitAction/>
            <SignalAction message="Reached WP3"/>
          </Sequence>
        </Fallback>
        <SignalAction message="Mission complete"/>
      </Sequence>
    </ReactiveFallback>
  </BehaviorTree>
</root>
```

Шаг 7. Главный узел (mission_bt_node.cpp):

```
#include <rclcpp/rclcpp.hpp>
#include <behaviortree_cpp/bt_factory.h>
#include "mission_bt/move_to_waypoint.hpp"
#include "mission_bt/wait_action.hpp"
#include "mission_bt/signal_action.hpp"
#include "mission_bt/battery_low.hpp"
#include "mission_bt/return_to_base.hpp"

int main(int argc, char* argv[]) {
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("mission_bt_node");

  BT::BehaviorTreeFactory factory;
  factory.registerNodeType<mission_bt::MoveToWaypoint>("MoveToWaypoint");
  factory.registerNodeType<WaitAction>("WaitAction");
  factory.registerNodeType<SignalAction>("SignalAction");
  factory.registerNodeType<BatteryLow>("BatteryLow");
  factory.registerNodeType<ReturnToBase>("ReturnToBase");
```

```

BT::Blackboard::Ptr blackboard = BT::Blackboard::create();
blackboard->set<double>("battery", 1.0); // 100%

auto tree = factory.createTreeFromFile("bt_mission.xml", blackboard);

// Таймер для тика дерева с частотой 10 Гц
rclcpp::WallRate loop_rate(10);
while (rclcpp::ok() && tree.tickOnce() == BT::NodeStatus::RUNNING) {
    loop_rate.sleep();
}

rclcpp::shutdown();
return 0;
}

```

Шаг 8. Сборка и запуск. В CMakeLists.txt добавить зависимости, установить файл дерева в share. После сборки запускаем симуляцию TurtleBot3:

```

export TURTLEBOT3_MODEL=waffle
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py

```

Затем узел: `ros2 run mission_bt mission_bt_node`

Робот начнёт движение по точкам, делая остановки и публикуя сигналы. При падении заряда (можно вручную установить через `ros2 param set /mission_bt_node battery 0.2`) дерево прервёт текущее движение и возвратится в начало координат.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№	Содержание индивидуального задания
---	------------------------------------

варианта	
1	Реализуйте миссию объезда трёх заданных преподавателем точек с паузами 5 секунд. Подайте звуковой сигнал через системный вызов (play). Запишите видео.
2	Замените реактивный возврат по батарее на возврат по внешнему сигналу остановки (топик /emergency_stop). При получении сообщения робот немедленно прекращает миссию и останавливается на месте.
3	Добавьте поддерево для обхода динамического препятствия: если во время движения к точке лидар обнаруживает препятствие ближе 0.5 м, робот выполняет манёвр объезда (объехать справа) и продолжает путь.
4	Реализуйте миссию патрулирования: робот объезжает точки циклически до получения команды стоп через сервис.
5	Используйте Nav2 (action NavigateToPose) для движения к точкам. Дерево должно отправлять цель и ожидать завершения навигации. Сравните плавность движения с самописным контроллером.
6	Добавьте мониторинг столкновений: если срабатывает bumper (топик /bumper), дерево отменяет текущий этап, отъезжает назад на 0.3 м и ждёт 2 с, после чего продолжает.
7	Реализуйте зарядку на базе: по окончании миссии робот возвращается на базу, сообщает "Mission complete", затем выключает двигатели.
8	Создайте дерево с параллельным узлом, которое одновременно выполняет движение к точке и сканирование местности. При обнаружении интересующего объекта (красный конус) публикует его координаты и переходит к следующей точке.

9	Интегрируйте детектор объектов (ЛР 31). Если во время движения камера видит красный знак, робот останавливается на 2 с, затем продолжает.
10	Реализуйте миссию «доставка»: робот должен посетить три точки, в каждой ожидать ручного подтверждения через сервис /confirm, после чего переходить к следующей.
11	Добавьте в дерево декоратор Retry: если движение к точке по какой-то причине не удалось (например, застревание), повторить попытку до двух раз, затем пропустить точку и перейти к следующей.
12	Разработайте дерево, в котором одна из точек является зарядной станцией. При достижении этой точки, если заряд ниже порога, запускается процедура стыковки и зарядки; иначе точка пропускается.
13	Создайте визуализацию текущего состояния дерева в RViz с помощью публикации маркеров: при выполнении каждого Action-узла публикуется текстовая метка с названием узла.
14	Проведите сравнительный анализ: реализуйте ту же миссию на конечном автомате (ЛР 23) и на дереве поведения. Оцените трудозатраты на изменение логики (добавление четвертой точки) в обоих подходах.
15	Реализуйте миссию с неожиданным препятствием, которое создаётся через 20 секунд после старта сценария. Дерево должно обрабатывать перепланирование и при необходимости пропускать точку, если она полностью заблокирована.

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2,

используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как спроектировать дерево поведения для сложной миссии?
2. Что такое SubTree?
3. Как реализовать условный переход в BT?
4. Как интегрировать BT с Nav2 NavigateToPose?
5. Что делать, если лист дерева возвращает FAILURE?
6. Как отображать состояние дерева во время работы?
7. Как реализовать прерывание миссии по сигналу?
8. Как тестировать дерево поведения без полного окружения?

Лабораторная работа 36.

ИТОГОВАЯ ЗАЩИТА: ДЕМОНСТРАЦИЯ АВТОНОМНОЙ СИСТЕМЫ

Цель и содержание работы: Разработать и защитить комплексный проект автономной мобильной робототехнической системы на платформе ROS2. Самостоятельно спроектировать архитектуру, включающую картографию, локализацию, глобальное и локальное планирование, поведенческую логику и взаимодействие с симулятором Gazebo. Продемонстрировать работоспособность системы на выбранном сценарии, подтвердить глубину понимания интеграции компонентов и умение аргументировать принятые проектные решения. Подготовить итоговый отчёт и презентацию, отвечающие требованиям портфолио инженера-робототехника.

Теоретическое обоснование

Итоговая защита синтезирует знания и навыки, полученные в течение всего курса лабораторных работ. Автономная мобильная система состоит из следующих базовых подсистем:

- Сенсорика: лидар, одометрия, IMU, камеры.
- Картография: статическая карта окружения (Occupancy Grid) или динамическое построение карты методом SLAM.
- Локализация: вероятностная оценка положения робота относительно карты (AMCL).
- Глобальное планирование: поиск оптимального маршрута по карте (A*, Dijkstra, SmacPlanner).
- Локальное планирование: генерация безопасных скоростей с учётом динамических препятствий (DWB, TEB).

- Поведенческий уровень: конечный автомат или дерево поведения, реализующее логику миссии (последовательность целей, реакции на события, обработку отказов).

- Интерфейс оператора: визуализация в RViz, целеуказание, мониторинг состояния.

Проектирование системы начинается с определения сценария и построения архитектурной диаграммы. На основе сценария выбираются необходимые компоненты, конфигурируются параметры навигационного стека Nav2, разрабатываются пользовательские узлы (например, для детекции объектов, следования по пути, коммуникации). Обязательной является проверка работоспособности в симуляторе Gazebo с реальной физикой и сенсорными моделями. Оцениваются метрики: успешность выполнения миссии, время, пройденный путь, минимальное расстояние до препятствий.

Пример

Рассмотрим типовой сценарий «Доставка посылки»: робот TurtleBot3 должен забрать груз на складе (точка А), доставить его в пункт выдачи (точка В), объехав препятствия, и вернуться на исходную позицию. Реализация включает карту известного окружения, Nav2 с глобальным SmacPlanner и локальным DWB, дерево поведения, управляющее последовательностью целей, и узел мониторинга состояния.

Шаг 1. Подготовка карты и симуляции. Запуск `turtlebot3_world` или создание собственного мира с необходимыми препятствиями. Построение карты с помощью `slam_toolbox` (ЛР 33), сохранение через `map_saver`.

Шаг 2. Настройка навигационного стека. Файл `params.yaml` конфигурирует AMCL, глобальную и локальную `costmap`, плагины планировщиков, сервер поведений. В глобальной `costmap` включается `obstacle_layer` для учёта динамических препятствий. Устанавливается

глобальный planner_server с SmacPlanner, controller_server с DWB, параметры которого подбираются в предшествующих работах.

Шаг 3. Разработка дерева поведения. Создаётся пакет delivery_bt (C++ или Python) с узлами Action: MoveToGoal (использует action NavigateToPose), PickUpPackage (имитация, таймер 2 с), DeliverPackage (аналогично). Условие BatteryLow проверяет заряд. ReactiveFallback на верхнем уровне прерывает миссию при низком заряде. Sequence описывает последовательность целей: MoveToGoal(A), PickUpPackage, MoveToGoal(B), DeliverPackage, ReturnToBase. Blackboard хранит координаты точек, уровень заряда, статус доставки.

Шаг 4. Интеграция и тестирование. Запуск симуляции, Nav2 с параметрами, узла дерева поведения. Целевые координаты задаются в конфигурационном файле или через параметры. Через RViz наблюдается движение робота по глобальному пути. Логируется выполнение каждого шага. При принудительном снижении заряда (ros2 param set) робот прерывает доставку и возвращается на базу.

Шаг 5. Документирование. Отчёт включает архитектурную схему, описание компонентов, графы дерева поведения, скриншоты RViz, анализ метрик (общее время миссии, длина пути, количество перепланирований). Презентация содержит демонстрационное видео и ответы на вопросы о выборе алгоритмов.

Порядок выполнения индивидуального задания

- 1) изучить теоретический материал по теме лабораторной работы;
- 2) разобрать представленный пример выполнения задания;
- 3) выполнить индивидуальное задание по полученному варианту;
- 4) оформить отчёт: ход выполнения, листинги команд, скриншоты, выводы;
- 5) подготовиться к защите лабораторной работы.

Варианты индивидуальных заданий

№ варианта	Содержание индивидуального задания
1	Сценарий «Доставка». Робот должен последовательно посетить склад, забрать груз (имитация), доставить его в заданную точку, вернуться на базу. Добавить мониторинг батареи с возвратом при разряде. Представить архитектуру в виде дерева поведения, задействовать Nav2 для каждой цели.
2	Сценарий «Патрулирование». Робот объезжает четыре контрольные точки по замкнутому маршруту, на каждой делает остановку для осмотра (скан лазером на 360°). При обнаружении движущегося объекта (через /scan, ЛР 27) следует за ним 5 секунд, затем продолжает маршрут. Использовать дерево поведения с Parallel для сканирования.
3	Сценарий «Сбор объектов». Робот получает список координат красных и синих конусов. Должен подъехать к каждому красному, остановиться на 3 с, синие игнорировать. Детекция цвета через OpenCV (ЛР 31). Карта строится в процессе SLAM (ЛР 33) во время выполнения миссии. Поведение: дерево поведения.
4	Сценарий «Экскурсовод». Робот ожидает команды начала экскурсии (топик /start). Затем последовательно перемещается к трём экспонатам, у каждого произносит текстовое описание (публикация в /speaker) и ждёт подтверждения от посетителя (топик /next). После финала возвращается в исходную точку. Интеграция с распознаванием речи не требуется, достаточно имитации команд через топики.
5	Сценарий «Спасательная миссия». В среде с разрушенными

	стенами (модифицированная карта) робот должен найти пострадавшего (красный цилиндр) и вернуться на базу. Поиск реализовать как случайное блуждание с уклонением от препятствий, детекция объекта по цвету. При обнаружении – движение к объекту и сигнал тревоги. Логика на конечном автомате.
6	Сценарий «Колонна с лидером». Два робота (ЛР 32): ведущий едет по маршруту из трёх точек (заранее заданных), ведомый следует за ним на дистанции 1.5 м. В случае расхождения – остановка и подтягивание. Дерево поведения на каждом роботе: у ведущего Sequence целей, у ведомого Fallback (следование, ожидание). Запись росбага для анализа.
7	Сценарий «Исследование неизвестной территории». Карта отсутствует. Робот должен исследовать замкнутый полигон, используя SLAM (slam_toolbox) и активное стремление к неизвестным областям (выбор точки на границе известной/неизвестной зоны, планирование пути). Миссия завершается, когда исследовано не менее 90% доступной площади. Использовать глобальный планировщик с allow_unknown.
8	Сценарий «Объезд препятствий на трассе». На прямолинейном треке появляются случайные статические препятствия (генерация через узел). Робот должен проехать трассу туда и обратно, не столкнувшись. Активировать поведение резервного движения при застревании. Измерить время прохождения. Сравнить производительность DWB и TEB (один из них использовать).
9	Сценарий «Парковка». Робот получает команду припарковаться на заданное парковочное место (прямоугольная область).

	Планировщик должен выполнить маневр с учётом габаритов (увеличенный footprint). При неудаче – повторная попытка с другой траекторией. Использовать дерево поведения с Retry.
10	Сценарий «Следование за человеком». В симуляции движется модель пешехода (динамическое препятствие). Робот должен обнаружить его через лазерный сканер и камеру, идентифицировать как цель (постоянный объект) и следовать за ним, сохраняя дистанцию 1 м. Поведение реализовать через конечный автомат.
11	Сценарий «Сортировка посылок». Робот находится в центре комнаты с тремя зонами разных цветов (маркеры на полу). Получает по топику команду: "доставить в зелёную зону". Должен подъехать к зелёной зоне, остановиться на 5 с, вернуться в центр. Детекция цветных зон по камере. Использовать дерево поведения.
12	Сценарий «Охрана периметра». Робот патрулирует квадрат (4 угловые точки). При обнаружении движущегося объекта (по изменению скана) следует за ним, пока тот не покинет охраняемую зону, затем возвращается к патрулированию. Дерево поведения должно быть реактивным.
13	Сценарий «Зарядка по расписанию». Робот выполняет обычный маршрут доставки (как вариант 1), но уровень заряда моделируется как величина, убывающая пропорционально пройденному расстоянию. Когда заряд падает ниже 30%, робот прерывает текущую задачу и направляется к зарядной станции (определённая координата), где остаётся 5 секунд (имитация зарядки), после чего возобновляет миссию с прерванного места. Для хранения контекста использовать Blackboard.

14	Сценарий «Светофор». Перед роботом на перекрёстке появляется визуальный сигнал светофора (красный/зелёный круг, отображаемый в симуляции или детектируемый по камере). Робот должен остановиться при красном и продолжить движение при зелёном. Миссия – проезд трёх перекрёстков. Детекция цвета через OpenCV. Использовать дерево поведения с реактивным Fallback.
15	Сценарий «Запись и воспроизведение миссии» (регрессионное тестирование). Студент сначала разрабатывает миссию доставки (как вариант 1) и записывает успешный сеанс в rosbag. Затем воспроизводит этот bag с изменённой версией навигационных параметров (например, сниженной скоростью или уменьшенным inflation radius) и демонстрирует, что система по-прежнему выполняет миссию. Оценивается качество отчёта и способность к отладке на записанных данных. ~~~

Содержание отчёта и его форма

Отчёт должен содержать название лабораторной работы, её цель, краткое описание используемого окружения (ОС, версия ROS 2, используемые инструменты и симуляторы), ход выполнения (листинги команд, конфигурационных файлов, скриншоты результатов), выводы. Отчёт оформляется в виде документа .docx или .pdf.

Вопросы для защиты работы

1. Как подготовить демонстрацию автономной системы?
2. Какие компоненты должны работать одновременно?
3. Как убедиться в устойчивой работе перед демонстрацией?
4. Что такое launch-файл верхнего уровня?

5. Как реагировать на отказ узла во время демонстрации?
6. Какие метрики фиксируются при демонстрации?
7. Как оформить итоговый отчёт?
8. Какие области применения автономных систем наиболее перспективны?

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Управление автономными системами» для студентов направления
подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	3
ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ	3
МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА	4
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ	5
ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ....	6
КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ	7

ВВЕДЕНИЕ

Настоящие методические указания определяют порядок организации и проведения самостоятельной работы студентов по дисциплине «Управление автономными системами». Дисциплина реализуется в рамках направления подготовки 09.03.04 «Программная инженерия» (направленность «Разработка программного обеспечения автономных и робототехнических систем»).

Цель самостоятельной работы — сформировать у студентов устойчивые навыки самостоятельного освоения современных инструментов разработки автономных робототехнических систем и готовность к практическому применению полученных знаний при решении инженерных задач.

Указания содержат общую характеристику видов самостоятельной работы, рекомендации по изучению теоретического материала, порядок подготовки к лабораторным работам, вопросы для собеседования и критерии оценивания компетенций.

ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студента по дисциплине включает проработку теоретического материала, подготовку к лабораторным работам, самостоятельное выполнение индивидуального задания и подготовку к защите. Форма контроля — защита отчёта и собеседование.

Таблица 1 — Виды самостоятельной работы и формы контроля

Вид самостоятельной работы	Содержание деятельности	Форма контроля
Проработка теоретического	Изучение лекций, учебной и официальной технической	Собеседование, тестирование

материала	документации по ROS 2, навигационным алгоритмам и методам управления автономными системами	
Подготовка к лабораторной работе	Изучение теоретического обоснования и примера, подготовка рабочего окружения (Ubuntu, ROS 2, Gazebo), уточнение варианта задания	Допуск к выполнению работы
Выполнение индивидуального задания	Самостоятельное выполнение задания по варианту, оформление отчёта с листингами и скриншотами	Проверка отчёта
Подготовка к защите	Подготовка ответов на контрольные вопросы, обоснование принятых технических решений	Защита отчёта

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

Изучение теоретического материала рекомендуется проводить последовательно, в соответствии с порядком следования лабораторных работ. По каждой теме необходимо ознакомиться с официальной документацией ROS 2 (docs.ros.org), материалами лекций и рекомендованной литературой.

Таблица 2 — Темы для самостоятельного изучения теоретического материала

№	Тема для самостоятельного изучения	Вопросы для проработки
1	Установка и настройка ROS	архитектура ROS 2, DDS, workspace,

	2	основные команды
2	Создание ROS-пакета	ament_cmake, colcon, package.xml, CMakeLists.txt
3–5	Коммуникационные механизмы	Publisher, Subscriber, QoS, типы сообщений
6	Пользовательские сообщения	файлы .msg, rosidl, генерация кода
7–8	Сервисы и действия	синхронные/асинхронные взаимодействия, .srv/.action
9–10	Симуляция и визуализация	Gazebo, RViz, плагины сенсоров, TF
11–12	Одометрия и карты	Odometry, Occupancy Grid, map_server, costmap_2d
13–14	Локализация и слияние сенсоров	AMCL, фильтр Калмана, robot_localization
15–17	Алгоритмы глобального планирования	Дейкстра, A*, интеграция в ROS 2
18–20	Алгоритмы локального планирования	DWA, TEB, настройка параметров
21–22	Управление роботом	ПИД-регулятор, cmd_vel, контроллер движения
23–24	Поведенческое программирование	FSM, Behavior Tree, BehaviorTree.CPP
25–28	Навигационный стек Nav2	компоненты Nav2, lifecycle, costmap, препятствия
29–30	Запись данных и расширения	rosbag2, custom costmap layer
31–33	Продвинутые темы	компьютерное зрение, мультиагентные системы, SLAM

34–36	Комплексные проекты	полный цикл навигации, ВТ-миссия, итоговая защита
-------	---------------------	---

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Выполнение каждой лабораторной работы осуществляется самостоятельно в соответствии с полученным вариантом. Рекомендуется придерживаться следующего порядка работы.

Таблица 3 — Этапы выполнения лабораторной работы

Этап	Содержание деятельности
Подготовительный	Изучение теоретического обоснования и примера, подготовка рабочего окружения (Ubuntu 22.04, ROS 2 Humble, Gazebo), уточнение варианта задания
Основной	Самостоятельное выполнение индивидуального задания, фиксация команд, конфигурационных файлов, исходного кода и полученных результатов
Оформление отчёта	Подготовка отчёта: цель, ход выполнения, листинги команд и кода, скриншоты, выводы
Защита	Ответы на контрольные вопросы, обоснование принятых технических решений

Отчёт оформляется в текстовом редакторе; листинги команд и кода приводятся моноширинным шрифтом. По итогам формулируются выводы, отражающие достигнутые результаты и возникшие трудности.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Для подготовки к собеседованию рекомендуется проработать следующие вопросы:

1. Что такое ROS 2? Чем он отличается от ROS 1 и какие преимущества даёт использование DDS?
2. Как устроено рабочее пространство ROS 2? Какие инструменты используются для сборки пакетов?
3. Назовите основные коммуникационные механизмы ROS 2. В каких случаях предпочтителен каждый из них?
4. Как определяются пользовательские типы сообщений, сервисов и действий?
5. Что такое Gazebo и RViz? Какова роль каждого инструмента в разработке автономного робота?
6. Что такое Occupancy Grid? Как строится и используется статическая карта для навигации?
7. Сформулируйте принципы работы AMCL. Как частицы отражают гипотезы о положении робота?
8. Что такое фильтр Калмана? Как он применяется для слияния данных одометрии и IMU?
9. В чём состоит алгоритм A*? Как эвристика влияет на качество и скорость поиска пути?
10. Сравните DWA и TEB. По каким критериям выбирается алгоритм?
11. Что такое ПИД-регулятор? Как каждая составляющая влияет на систему управления?
12. Что такое FSM и Behavior Tree? Сравните эти подходы к реализации поведения робота.
13. Из каких компонентов состоит Nav2? Какова роль lifecycle_manager?
14. Что такое SLAM? Как работает GMapping и какие данные ему необходимы?
15. Каковы принципы построения многоагентных систем на базе ROS 2?

16. Что такое rosbag2? Как он используется для тестирования и отладки?

17. Как пользовательский слой costmap расширяет возможности навигации?

18. Каковы принципы разработки дерева поведения для комплексной миссии?

КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Оценивание результатов самостоятельной работы осуществляется по итогам выполнения и защиты лабораторных работ, а также собеседования. Критерии оценивания приведены в таблице 4.

Таблица 4 — Критерии оценивания

Оценка	Критерии оценивания
«Отлично»	Задание выполнено в полном объёме и без ошибок; отчёт оформлен в соответствии с требованиями; студент свободно владеет теоретическим материалом, обосновывает принятые решения, уверенно отвечает на все вопросы.
«Хорошо»	Задание выполнено с незначительными недочётами; отчёт оформлен с небольшими замечаниями; студент владеет материалом, но допускает неточности при ответах.
«Удовлетворительно»	Задание выполнено частично; отчёт содержит ошибки; студент испытывает затруднения при ответах на контрольные вопросы.
«Неудовлетворительно»	Задание не выполнено или выполнено с грубыми ошибками; отчёт отсутствует или не соответствует требованиям; студент не владеет материалом.