Практическое программирование

Python, ROS2 и нейросети

лабораторный практикум по программированию
на примерах задач управления автономными объектами

Ю.Н.Поликарпов

ктн, снс

самостоятельное учебное электронное издание
предназначено для использования в учебном процессе
школьниками, студентами колледжей и ВУЗов,
обучающихся по физико-техническим специальностям,
а также всем интересующимся робототехникой и программированием

СПб
Издатель Ю.Н.Поликарпов
2026

ISBN 978-5-6050573-5-2

© Ю.Н.Поликарпов, 2026

УДК 372.862
ББК 74.263.2
П50

П50 **Поликарпов, Ю.Н.** Практическое программирование. Python, ROS2 и нейросети : лабораторный практикум по программированию на примерах задач управления автономными объектами : самостоятельное учебное электронное издание / Ю.Н.Поликарпов. — 1-е изд. — СПб. : издатель Ю.Н.Поликарпов, 2026. — 1 CD-R (3 Мб). — ISBN 978-5-6050573-5-2. — Текст : электронный.

В практикуме рассматривается программирование робота на ROS2 с лидаром, камерой и нейропроцессором с использованием трех программных симуляторов.

В первую очередь практикум адресован школьникам и студентам колледжей и ВУЗов, обучающимся по физико-техническим специальностям, но будет полезен всем интересующимся робототехникой и программированием.

Практикум распространяется под открытой лицензией на условиях использования для личных некоммерческих учебных целей при соблюдении всех трех условий одновременно. Использование для иных целей с письменного разрешения автора.

Минимальные системные требования:

Процессор Intel® или AMD с частотой не менее 1,5 ГГц;
Windows 11, Windows 10 версии 1809 или более поздней, Windows Server 2016 или Windows Server 2019; 4 ГБ оперативной памяти (16 ГБ желательно); 20 ГБ свободного пространства на жестком диске; разрешение экрана 1024x768; аппаратное ускорение видеоплаты (желательно); Adobe Acrobat Standard.

Издательская система LaTeX.
Компьютерная верстка, редактура, корректура Ю.Н.Поликарпова.

Дата подписания к использованию 01.08.2026.
Объем издания 3 МБ.
Комплектация 1 CD-R, пластиковый бокс.
Тираж 100 экз.
Издатель Ю.Н.Поликапов, newwave50@bk.ru, www.dec.su, +79516739548.

Оглавление

Введение: О практикуме, роботе и кибернетическом мышлении . . . **6**

Ведение в Linux . . . **10**

Занятие 1: Первое знакомство с Linux-роботом . . . **10**

Занятие 2: Файловый менеджер и права доступа . . . **18**

Краткое введение в Python и ООП . . . **27**

Занятие 3: Повторение Python — переменные, типы, ввод/вывод, turtle . . . **27**

Занятие 4: Циклы for и range. Правильные многоугольники . . . **35**

Занятие 5: Циклы while, окружность, спираль, break, continue . . . **42**

Занятие 6: Условные операторы if-elif-else, раскрашивание по условию . . . **49**

Занятие 7: Списки и словари . . . **56**

Занятие 8: Функции: def, аргументы, возврат значения . . . **65**

Занятие 9: Модули и импорт . . . **73**

Занятие 10: Основы ООП: класс Artist . . . **80**

Занятие 11: Наследование, итоговое занятие, зачёт . . . **87**

Аппаратная платформа робота . . . **96**

Занятие 12: Обзор платформы: компьютер, контроллер, меканум-колёса . . . **96**

Занятие 13: Лидар и камера: устройство, принципы, характеристики . . . **101**

Занятие 14: Нейропроцессор Hailo-8 . . . **107**

Занятие 15: Сборка робота: крепление, коммутация, питание . . . **112**

Занятие 16: Тестовый запуск и первичная диагностика . . . **117**

Основы ROS2: базовые понятия и инструменты . . . **125**

Занятие 17: Архитектура ROS2, инструменты командной строки, teleop . . . **125**

Занятие 18: Издатель, пакет, colcon, первый Twist . . . **131**

Занятие 19: Свой телеоператор: захват клавиш . . . **139**

Занятие 20: Подписчик, связка, rqt_graph, частота . . . **145**

Занятие 21: Калибровка прямолинейного движения (выполняется на оригинальном роботе) . . . **152**

Занятие 22: Калибровка поворота (на оригинальном роботе) . . . **160**

Занятие 23: Типы сообщений, сервисы, ros2 bag . . . **169**

Занятие 24: Комплексная практика ROS2 . . . **176**

Занятие 25: Зачёт по ROS2 . . . **183**

Программирование движения на меканум-базе 188

Занятие 26: Архитектура автопилота: сервисы go_time, go_relative, stop 188

Занятие 27: Первый клиент: вызов go_time из Python-ноды, heartbeat 194

Занятие 28: Сервис go_relative и чтение телеметрии 200

Занятие 29: Сервис stop и базовый класс RobotDriver 208

Занятие 30: Движение по квадрату через последовательный вызов сервисов 213

Занятие 31: Деревья поведения: движок, Sequence, MoveRelativeAction, MoveTimeAction, StopAction 219

Занятие 32: Condition, Fallback, RetryAction: ветвление и обработка ошибок в дереве 224

Занятие 33: Parallel: одновременные действия 229

Занятие 34: Квадрат на дереве поведения: полный сценарий с обработкой ошибок 237

Занятие 35: Итоговое испытание: комплексная траектория на деревьях 243

Сенсоры: лидар и камера 253

Занятие 36: Зачем сенсоры? Лидар: устройство, первый запуск, визуализация 253

Занятие 37: Геттеры лидара в RobotDriver: первый тест без движения 257

Занятие 38: Сегментация данных лидара по секторам и Condition-узлы 263

Занятие 39: Остановка перед препятствием по лидару: первая BT-миссия с сенсором 268

Занятие 40: Камера: устройство, запуск, калибровка (на оригинальном роботе) 274

Занятие 41: Геттеры камеры в RobotDriver: фотофиксация по лидару 279

Занятие 42: Движение вдоль стены по лидару 284

Занятие 43: Выход из лабиринта по правилу правой руки 290

Занятие 44: Зачёт по сенсорам 296

Введение в OpenCV 305

Занятие 45: Проверка установки OpenCV, загрузка изображений, цветовые пространства 305

Занятие 46: Фильтрация: размытие, выделение границ Canny 312

Занятие 47: Бинаризация и морфологические операции 318

Занятие 48: Поиск контуров, геометрические примитивы 324

Занятие 49: CvBridge, обработка видеострима в реальном времени 330

Занятие 50: Детектирование линии разметки 336

Занятие 51: Трекинг объекта по цвету в HSV 342

Занятие 52: Выделение зон интереса (ROI) 347

Занятие 53: Оптический поток (Лукас-Канаде) 353

Занятие 54: Детектирование препятствий: BackgroundSubtractor MOG2 360

Занятие 55: ArUco-маркеры 365

Занятие 56: Объединение линии и ArUco-маркеров . . . 369
Занятие 57: Интеграция детекторов в единый Action-узел . . . 376
Занятие 58: Зачёт по OpenCV . . . 382

Введение в PyTorch и нейропроцессор Hailo-8 . . . 390

Занятие 59: Нейросети: базовые понятия, CNN, установка PyTorch . . . 390
Занятие 60: Предобученные модели: YOLO, инференс на CPU . . . 394
Занятие 61: Обработка результатов YOLO: boxes, классы, confidence, NMS . . . 400
Занятие 62: Hailo-8: архитектура, HailoRT, конвертация модели . . . 405
Занятие 63: Сравнение CPU vs Hailo-8: FPS, latency . . . 411
Занятие 64: Интеграция детектора в миссию на дереве поведения . . . 418
Занятие 65: Реакция робота на детектирование . . . 423
Занятие 66: Зачёт по PyTorch и Hailo-8 . . . 429

Приложение: Работа с RViz2 . . . 437

Тема: Визуализация данных робота в реальном времени . . . 437
1. Запуск RViz2 . . . 437
2. Визуализация LaserScan (лидар) . . . 438
3. Визуализация RobotModel (3D-модель робота) . . . 438
4. Визуализация одометрии . . . 439
5. Визуализация TF (систем координат) . . . 439
6. Визуализация видеопотока с камеры (Image) . . . 440
7. Визуализация карты . . . 441
8. Сохранение и загрузка конфигурации RViz . . . 441
9. Полезные приёмы . . . 441
10. Типичные проблемы и решения . . . 442

Введение: О практикуме, роботе и кибернетическом мышлении

О чём этот практикум

Этот лабораторный практикум, являющийся логическим продолжением учебного пособия «Практическое программирование» — не лекции и не «программирование по учебнику». Это **66 лабораторных работ**, в каждой из которых вы работаете с роботом: подключаетесь к нему по SSH, пишете код, запускаете, наблюдаете результат, отлаживаете.

Робот — идеальная модель для изучения того, как программный код взаимодействует с физическим миром. Здесь нельзя «закрыть глаза» на ошибку: если команда неверна, робот врежется в стену. Это учит думать, прежде чем запускать.

К концу практикума вы сможете:

- Заставить робота следовать по линии и ориентироваться по ArUco-маркерам.
- Детектировать людей в реальном времени через нейропроцессор.
- Строить сложные миссии на деревьях поведения.
- Понимать, как работают промышленные системы вроде Nav2.

На каком оборудовании можно работать

Практикум разработан для **оригинального робота** на меканум-базе с Raspberry Pi 4/5, STM32, лидаром, камерой и Hailo-8. Этот робот создан специально для данного курса и обеспечивает полное погружение во все темы.

Однако практикум **не привязан жёстко** к конкретному железу. Вы можете работать:

1. На оригинальном роботе (полный функционал). Все 66 работ выполняются с реальными сенсорами и движением. Вы видите облако точек лидара, видеопоток с камеры, реакцию робота на команды.

2. На любом ROS2-совместимом роботе (с адаптацией). Если у вас есть другой робот на ROS2 с дифференциальным или меканум-приводом, вы можете использовать его. Единственное требование — наличие **стандартных ROS2-топиков**:

- `/scan` (`sensor_msgs/LaserScan`) — данные лидара.
- `/odom` (`nav_msgs/Odometry`) — одометрия.
- `/cmd_vel` (`geometry_msgs/Twist`) — управление движением.
- `/camera/image_raw` (`sensor_msgs/Image`) — видеопоток с камеры.
- `/robot_description` — описание модели робота.

Описания таких роботов можно найти в интернете: TurtleBot4, LeoRover, JetBot с лидаром, самодельные платформы на ROS2. Если робот публикует эти четыре топика — практикум применим.

3. В симуляторе (без физического робота). Практикум поддерживает работу в симуляции. У вас есть три варианта:

- **Облегчённая симуляция ROS2** — робот присутствует как набор топиков (`/scan`, `/odom`, `/cmd_vel`, `/camera/image_raw`), видеопоток формируется с помощью библиотеки OpenCV. Подходит для отработки навигации, деревьев поведения. Не требует мощного компьютера.
- **Симуляция ROS2 с 3D-видеопотоком** — камера публикует реалистичный видеопоток с помощью библиотеки `panda3d`. Подходит для полного цикла, включая компьютерное зрение и нейросети. Требует компьютер с бОльшим объемом ОЗУ, чем в первом случае.
- **Максимально упрощенная симуляция** — видеопоток формируется с помощью библиотеки OpenCV. Подходит для отработки навигации, деревьев поведения. Для слабых Windows машин.

Симулятор позволяет проходить практикум полностью удалённо, без физического доступа к роботу. Все программы пишутся и отлаживаются точно так же, как на реальном роботе — разница лишь в том, что вместо реальных стен вы видите виртуальные.

Ваш лабораторный стенд (оригинальный робот)

Если вы работаете с оригинальным роботом курса, вот его ключевые компоненты:

Платформа: меканум-база с 4 всенаправленными колёсами. Может ехать боком, по диагонали, вращаться на месте.

Вычислительные модули:

- **Raspberry Pi 4/5** — одноплатный компьютер под Ubuntu 26.04. Вы выполняете на нём свои программы.
- **STM32** — микроконтроллер реального времени. Вы не трогаете его код, но именно он управляет моторами.

Сенсоры:

- **Лидар (LIDAR)** — лазерный дальномер 360°. Данные доступны в топике `/scan`.
- **Камера** — 30 FPS. Видеопоток в топике `/camera/image_raw`.

Ускоритель ИИ:

- **Hailo-8** — нейропроцессор для быстрого инференса нейросетей.

Структура практикума

66 лабораторных работ по 90 минут (два академических часа), объединённые в тематические блоки:

Блок	Работы	Что делаем
Linux и инструменты	1–2	Подключаемся к роботу, работаем в терминале
Python и ООП	3–11	Пишем код, рисуем черепашкой, осваиваем классы
Аппаратная платформа	12–16	Устройство робота, сборка, диагностика
Основы ROS2	17–25	Узлы, топики, сервисы, калибровка, зачёт
Сервисы автопилота	26–35	Высокоуровневое управление, деревья поведения, зачёт
Сенсоры: лидар и камера	36–44	Читаем данные, реагируем на препятствия, зачёт
OpenCV	45–58	Обработка изображений, детекторы, ArUco, зачёт
Нейросети и Hailo-8	59–66	PyTorch, YOLO, инференс на NPU, зачёт

Три уровня сложности в каждой работе

Каждая лабораторная работа содержит три уровня:

Уровень «База» — обязательный минимум. Конкретные инструменты и навыки.

Уровень «Углублённый» — понимание «что под капотом».

Уровень «Исследовательский» — выход за границы: эксперименты, сравнения, улучшения.

Форматы использования:

- **Обычные классы** — 34 часа в год (только База).
- **Профильные классы** — 34 часа в год (База + Углублённый).
- **Дополнительное образование** — 144 часа в год (все три уровня).

Чему вы научитесь помимо программирования

Главный результат практикума — **кибернетическое мышление**:

1. **Системность.** Изменили коэффициент регулятора — изменилось поведение всей системы.
2. **Обратная связь.** Сенсор → решение → действие → сенсор. Этот цикл в каждой работе.
3. **Модель ≠ реальность.** Одометрия говорит «метр», рулетка — 85 см. Вы научитесь измерять и компенсировать эту разницу.
4. **Абстракция.** От ручного `Twist` к сервисам, от сервисов к деревьям поведения.
5. **Отказоустойчивость.** `RetryAction`, `Fallback`, watchdog — вы освоите это на практике.

Как построена лабораторная работа

Каждая работа оформлена как трех-уровневая. Вы:

1. Читаете теорию.
2. Подключаетесь к роботу (или симулятору) по SSH.
3. Пишете код в своём каталоге `~/ws/rbt/scripts/`.
4. Запускаете и наблюдаете результат.

5. При необходимости смотрите HTTP-стрим для визуализации.
6. Если какие-то моменты недостаточно ясны ищете дополнительную информацию в различных источниках, например, в сети Internet. Помните: вы активный субъект — искатель знаний, а не овощь, ожидающий когда его унавозят.

Код примеров намеренно прост — без сложных абстракций, чтобы вы могли сосредоточиться на сути.

Приступайте к работе!

Ведение в Linux

Занятие 1: Первое знакомство с Linux-роботом

Содержание занятия: SSH, навигация в терминале, текстовый редактор nano

Цели занятия

- **Образовательная**: Освоить базовые принципы взаимодействия с роботом через терминал: удалённое подключение, навигацию по файловой системе, создание и редактирование файлов.
- **Развивающая**: Сформировать понимание операционной системы робота как «мозга», которым можно управлять текстовыми командами.
- **Воспитательная**: Продемонстрировать важность точности и аккуратности при работе с командной строкой.

План занятия (90 минут)

1. **Организационный момент и введение (10 мин)**
2. **Уровень «База»: Первое подключение и навигация (35 мин)**
   - Что такое Linux и почему он на роботе?
   - SSH: первое подключение к роботу.
   - Смена пароля командой `passwd`.
   - Файловая система: `pwd`, `ls -l`, `cd`, `mkdir`.
   - Текстовый редактор nano: создание и редактирование первого файла.
   - Завершение сеанса `exit` и повторный вход с новым паролем.
3. **Уровень «Углублённый»: Скрипты и сессии (25 мин)**
   - Создание и запуск простейшего Python-скрипта на роботе.
   - Менеджер терминалов tmux: запуск, создание окон, отключение и восстановление сессии.
   - Создание структуры проекта с помощью `mkdir -p`.
4. **Уровень «Исследовательский»: Автоматизация и анализ (15 мин)**
   - Разработка Python-скрипта-визитки для сбора системной информации.
   - Установка автопилота и симулятора.
   - Мини-исследование: почему процессы в tmux не умирают при разрыве SSH.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение в операционную систему робота

Наш робот — это не просто набор моторов и датчиков. Его главный «мозг» — одноплатный компьютер Raspberry Pi, работающий под управлением операционной системы **Ubuntu Linux**. Почему именно Linux?

- Это стандарт в мире робототехники и искусственного интеллекта.
- Это очень гибкая, мощная и надёжная система, которой мы можем полностью управлять.
- Мы не будем использовать мышь и графический интерфейс. Всё взаимодействие — через **терминал**, текстовые команды. Это самый быстрый и точный способ поставить задачу роботу.

Сегодня мы сделаем первый шаг — научимся «общаться» с нашим роботом.

2. Уровень «База»: Первое подключение и навигация

(Инструмент, навык, интерфейс)

Этот раздел обязателен для всех. Мы научимся входить в систему, перемещаться по папкам, создавать и изменять файлы.

2.1. Удалённое подключение по SSH

SSH (Secure Shell) — это программа, которая позволяет безопасно подключаться к другому компьютеру и выполнять на нём команды. Представьте, что вы открываете окно в командную строку робота со своего ноутбука.

> **Важно:** Учитель уже запустил базовое ПО робота и сообщил его IP-адрес. Ваш робот должен быть включён. Если физический робот недоступен используйте подключение к компьютеру с запущенным симулятором. Как установить и запустить симулятор написано в конце занятия - раздел 4.2.

Откройте терминал на своём компьютере и введите команду для подключения. Формат команды:

```
ssh <имя_пользователя>@<ip-адрес_робота>
```

Например: `ssh pi@172.22.17.30`

При первом подключении система спросит, доверяете ли вы этому компьютеру. Введите `yes` и нажмите Enter. Затем введите временный пароль, который выдаст преподаватель. **Важно:** при вводе пароля вы не увидите ни звёздочек, ни точек — это нормально, так задумано для безопасности. Просто введите пароль и нажмите Enter.

Если всё прошло успешно, вы увидите приглашение командной строки, которое выглядит примерно так: `login@ubuntu:~$`

Это означает, что вы внутри робота!

2.2. Первые действия: смена пароля и ориентирование

Первое, что нужно сделать — сменить временный пароль на свой собственный.

```
passwd
```

Команда попросит ввести текущий пароль, а затем дважды новый. Запомните свой новый пароль!

Теперь давайте осмотримся. Вы находитесь в своей домашней папке. В Linux нет дисков C: или D:, всё начинается с корня `/`.

- **Где я?** `pwd` (print working directory) — покажет ваше текущее местоположение.

  ```
  pwd
  # /home/pi
  ```

- **Что здесь есть?** `ls` (list) — покажет файлы и папки. Флаг `-l` покажет подробный список с правами, размером и датой.

  ```
  ls -l
  ```

- **Как перейти?** `cd` (change directory) — меняет папку.

  ```
  cd /home       # Перейти в конкретную папку по полному пути
  cd ..             # Подняться на уровень выше, в /
  cd ~              # Вернуться в домашнюю папку (тильда ~ - это её символ)
  ```

- **Как создать папку?** `mkdir` (make directory).

  ```
  mkdir ws
  ```

2.3. Текстовый редактор nano: создаём первый файл

nano — это простой текстовый редактор прямо в терминале. Сейчас мы создадим наш первый файл на роботе.

```
nano hello_robot.txt
```

Откроется окно редактора. Напишите что-нибудь, например: `Hello from student!`.

Внизу экрана вы видите подсказки, где `^` означает клавишу `Ctrl`:

- `Ctrl+O` — сохранить файл (WriteOut). Нажмите, затем Enter для подтверждения имени.
- `Ctrl+X` — выйти из редактора.

Давайте проверим, что файл создался и в нём есть наш текст:

```
cat hello_robot.txt
```

Команда `cat` выведет содержимое файла на экран.

И наконец, завершим сеанс работы с роботом:

```
exit
```

Теперь попробуйте подключиться снова, используя уже **ваш новый пароль**.

3. Уровень «Углублённый»: Скрипты и долгоживущие задачи

(Что под капотом)

Для тех, кто успешно справился с базовой частью. Мы превратим робота из просто компьютера в исполнителя программ.

3.1. Создание и запуск Python-скрипта

На роботе установлен Python. Давайте напишем простую программу. Создайте новый файл `my_script.py` с помощью nano:

```
nano my_script.py
```

И напишите в нём код:

```
print("Привет от твоего робота!")
print("Я умею складывать числа: 2 + 3 =", 2 + 3)
```

Сохраните и выйдите (`Ctrl+O` , `Enter` , `Ctrl+X`). Запустите скрипт:

```
python3 my_script.py
```

Вы увидите вывод программы. Вы только что заставили робота выполнить ваш код!

3.2. Менеджер терминалов tmux: работа без разрывов

Представьте: вы запустили на роботе программу, которая будет работать час, и ваше SSH-соединение разорвалось. Программа умрёт. **tmux** решает эту проблему. Он создаёт сессию, которая живёт на роботе независимо от вашего подключения.

1. **Запуск tmux**:

   ```
   tmux
   ```

   Снизу появится зелёная полоса — вы внутри tmux.

2. **Создание нового окна**: Все команды tmux начинаются с префикса `Ctrl+B` .

   - Нажмите `Ctrl+B` , отпустите, затем нажмите `C` (Create).
   - Внизу появится вкладка `1:bash` . Вы создали новое окно.

3. **Переключение между окнами**:

   - `Ctrl+B` , затем `0` — перейти на окно 0.
   - `Ctrl+B` , затем `1` — перейти на окно 1.

4. **Отключение от сессии (Detach)**: `Ctrl+B` , затем `D` . Вы вернётесь в обычный терминал, но всё, что было внутри tmux, продолжит работать.

5. **Восстановление сессии (Attach)**:

```
tmux attach
```

Вы снова увидите свои окна и работающие в них программы.

Попробуйте создать сессию, запустить в одном окне `python3 my_script.py`, а в другом — текстовый редактор nano. Отключитесь и подключитесь заново, чтобы убедиться, что всё работает.

3.3. Структура проекта

Хороший разработчик всегда организует свои файлы. Команда `mkdir -p` создаёт цепочку вложенных папок за один раз:

```
mkdir -p ~/ws/rbt/src ~/ws/rbt/logs ~/ws/rbt/config ~/ws/rbt/scripts
ls -l ~/ws/rbt
```

Теперь у вас есть чёткая структура для будущих программ.

4. Уровень «Исследовательский»: Системное мышление и автоматизация

(Новое применение, прорыв за границы)

Для будущих системотехников-исследователей, которые хотят не просто пользоваться, а понимать и улучшать систему.

4.1. Скрипт-визитка: сбор системной информации

Мы напишем Python-программу, которая выводит визитную карточку робота, собирая информацию из системы.

Создайте файл `robot_bio.py`:

```
nano robot_bio.py
```

И напишите следующий код. В нём мы используем модуль `subprocess` для выполнения команд bash прямо из Python.

```
import subprocess

def run_cmd(cmd):
    # Выполняет команду в терминале и возвращает её вывод
    result = subprocess.run(cmd, shell=True, capture_output=True, text=Tru
e)
    return result.stdout.strip()

print("="*40)
print("   ВИЗИТНАЯ КАРТОЧКА РОБОТА")
print("="*40)
print(f"Имя хоста: {run_cmd('hostname')}")
print(f"Мой IP-адрес: {run_cmd('hostname -I')}")
print(f"Версия ОС: {run_cmd('lsb_release -ds')}")
print(f"Версия Python: {run_cmd('python3 --version')}")
print(f"Свободное место: {run_cmd('df -h / | tail -1 | awk \'{print $4}
```

```
\'')}")
print("="*40)
```

Запустите скрипт и увидите системную информацию о роботе. Подумайте, зачем такая программа может понадобиться, если у вас целый парк роботов?

4.2.1. Установка автопилота и ROS2 симулятора (заметки для sudo-epa)

В качестве среды исполнения подойдет любой робот с ubuntu 26.04 и ROS2, получающий в качестве управления команды в топик `/cmd_vel` и публикующий стандартные корректно работающие топики `/odom`, `/scan`, `/camera/image_raw`, `/robot_description` (как собрать такого робота можно найти в сети, например TurtleBot) или ПК с ubuntu 26.04 и ROS2. Ниже описана процедура установки автопилота и симулятора, необходимых для работы по этому пособию. Все действия выполняются от лица sudo-epa как на роботе, так и на ПК (предполагается, что ROS2 и ubuntu 26.04 уже установлены, кроме того робот удовлетворяет описанным выше требованиям - `/cmd_vel`, `/odom`, `/scan`, `/camera/image_raw`, `/robot_description`. Также плюсом для робота будет связка RPI5 + Hailo8.).

```
# Подготавливаем каталог и скачиваем симулятор
mkdir -p ~/ws/git
cd ~/ws/git
git clone http://dec.su/upload/wysiwyg/bare_moebius_sim.git sim
cd sim
sudo cp -r drv /opt/

echo 'export PYTHONPATH=$PYTHONPATH:/opt/drv' >> ~/.bashrc
export PYTHONPATH=$PYTHONPATH:/opt/drv


# Устанавливаем автопилот для робота (на котором уже установлены базовые пакеты, формирующие указанные выше топики) (если мы на роботе) симулятор на роботе не устанавливаем только автопилот
sudo -i
source /opt/ros/lyrical/setup.bash
colcon build --packages-select ifra_mol_1 --install-base /opt/ros/custom_nodes --merge-install
source /opt/ros/custom_nodes/setup.bash
colcon build --packages-select robot_autopilot --install-base /opt/ros/custom_nodes --merge-install
find /opt/drv -type d -exec chmod 755 {} +
find /opt/drv -type f -exec chmod 644 {} +
find /opt/ros/custom_nodes -type d -exec chmod 755 {} +
find /opt/ros/custom_nodes -type f -exec chmod 644 {} +
exit

# Запуск автопилота на роботе (если мы на роботе)
source /opt/ros/lyrical/setup.bash
source /opt/ros/custom_nodes/setup.bash
ros2 run robot_autopilot autopilot_node.py # Нужен для изучения материала начиная с 5 главы

# Устанавливаем симулятор и автопилот на ПК (если мы на ПК)
sudo -i
source /opt/ros/lyrical/setup.bash
```

```
colcon build --install-base /opt/ros/custom_nodes --merge-install
find /opt/drv -type d -exec chmod 755 {} +
find /opt/drv -type f -exec chmod 644 {} +
find /opt/ros/custom_nodes -type d -exec chmod 755 {} +
find /opt/ros/custom_nodes -type f -exec chmod 644 {} +
exit

# Запуск симулятора на ПК (если мы на ПК)
source /opt/ros/lyrical/setup.bash
source /opt/ros/custom_nodes/setup.bash
ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml # Без
автопилота - в первых четырех главах
ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_autop
ilot:=true # С автопилотом - начиная с 5 главы
ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml
start_rviz:=true use_lite:=true start_autopilot:=true # с автоматическим
стартом rviz2 и облегченной версией видеосимуляции
```

Это общая схема. Если при запуске симулятор или автопилот на что то ругается - доставить необходимые стандартные пакеты (например `em` или `panda3d`). Если есть желание создать новую сцену - создаем новый файл .yaml в конфиге, перекомпилируем и включаем его вместо `hall.yaml`. При добавлении текстур также необходима перекомпиляция симулятора.

Rviz2 для робота запускается просто:

```
rviz2
```

для симулятора:

```
rviz2 --ros-args -p use_sim_time:=true
```

Файл сцены содержит информацию о препятствиях, исходном положении робота, разметке черной линии, ArUco метках и различных знаках. Формат можно понять из `hall.yaml`, идущего в комплекте. При добавлении новой сцены или изменении существующей необходима перекомпиляция симулятора. То же касается и текстур. НО! метки, знаки, линия появляются только в продвинутой версии видеосимуляции, то есть при запуске без аргумента `use_lite:=true`.

4.2.2. Установка симулятора без ROS2 на Windows машины.

Если найти Linux машину нет возможности, упрощенный симулятор можно развернуть на Windows машине с установленным Python, для этого надо скачать пакет `http://dec.su/upload/wysiwyg/pond-0.1.0-py3-none-any.whl` и установить его с помощью команды `pip install pond-0.1.0-py3-none-any.whl`.

Запуск симуляции осуществляется из каталога где расположен управляющий скрипт командой:

```
pond scene:=lab.yaml ctrl:=test.py auto:=n
```

`test.py` - управляющий скрипт, `lab.yaml` - сцена, которая должна находиться в том же каталоге, что и управляющий скрипт. В приведенном виде симуляция запускается с выключенным автопилотом. При необходимости запустить симуляцию с включенным автопилотом необходимо использовать команду:

```
pond scene:=lab.yaml ctrl:=test.py
```

4.3. Мини-исследование: Живучесть процессов tmux

Почему программы в tmux не умирают при разрыве связи, а программы в обычном SSH-терминале — умирают? Это связано с тем, как Linux управляет процессами.

Задание-исследование:

1. **Без tmux:** В обычном терминале запустите бесконечный процесс, который пишет время в файл:

   ```
   while true; do date >> /tmp/regular_test.log; sleep 2; done &
   ```

   (Символ `&` в конце запускает процесс в фоне. Запомните его PID, который выведет терминал).

2. Проверьте, что файл наполняется: `cat /tmp/regular_test.log`.

3. **Закройте терминал (SSH-сессию) и подключитесь заново.**

4. Проверьте файл `cat /tmp/regular_test.log`. Записи прекратились, как только вы отключились. Ваш процесс убит.

5. **С tmux:** Повторите эксперимент внутри tmux-сессии. Запустите `tmux`, выполните `while true; do date >> /tmp/tmux_test.log; sleep 2; done`. Отключитесь от tmux (`Ctrl+B, D`), разорвите SSH, подключитесь заново, восстановите сессию (`tmux attach`).

6. Проверьте файл `cat /tmp/tmux_test.log`. Записи продолжались всё время вашего отсутствия!

Объяснение: Когда вы закрываете терминал, ядро Linux посылает сигнал `SIGHUP` (hang-up) всем процессам, привязанным к этому терминалу. Tmux же создаёт свой собственный, виртуальный терминал-сессию, который не привязан ни к какому физическому SSH-соединению. Поэтому при разрыве связи `SIGHUP` получает ваш SSH-клиент, а процессы внутри tmux продолжают спокойно работать.

5. Выводы по занятию

1. **Linux — основа управления роботом.** Вы научились «входить внутрь» этой системы через SSH.
2. **Терминал — главный интерфейс инженера.** Вы освоили базовые команды для навигации (`pwd, ls, cd`), создания объектов (`mkdir, nano`) и контроля (`cat`).
3. **Python — язык, понятный роботу.** Вы написали и выполнили свою первую программу непосредственно на бортовом компьютере.
4. **tmux — инструмент для настоящих проектов.** Вы научились решать проблему долгоживущих задач и сделали первый шаг к профессиональной организации рабочей среды.
5. **Системное мышление начинается с вопросов.** Исследование на тему «почему процессы в tmux живут дольше» — это пример кибернетического подхода, который связывает наблюдение, теорию (сигналы ОС) и практику (надёжные системы).

Домашнее задание (по желанию):

- **Базовое**: Повторите все команды навигации и создайте в своей домашней папке структуру `week01/exercise01` с одним текстовым файлом внутри.
- **Продвинутое**: Установите симулятор на домашний компьютер.
- **Исследовательское**: Найдите в интернете, что такое команда `screen` и в чём её ключевое отличие от `tmux`. Какая из них, по-вашему, удобнее для робототехники и почему?

Занятие 2: Файловый менеджер и права доступа

Содержание занятия: Midnight Commander, права доступа, запуск скриптов

Цели занятия

- **Образовательная**: Освоить файловый менеджер Midnight Commander (mc) для эффективной навигации по файловой системе робота. Изучить базовую модель прав доступа Linux и научиться делать скрипты исполняемыми.
- **Развивающая**: Сравнить графический (mc) и командный (терминал) способы выполнения задач, выбирая оптимальный инструмент.
- **Воспитательная**: Привить понимание важности прав доступа как основы безопасности и порядка в многопользовательской системе робота.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Интерфейс Midnight Commander и chmod (35 мин)**
    - Запуск mc и его двухпанельный интерфейс.
    - Навигация: Tab, стрелки, Enter.
    - Функциональные клавиши: F3 (просмотр), F5 (копирование), F6 (перемещение), F8 (удаление), F10 (выход).
    - Создание папок и файлов через mc.
    - Понятие прав доступа. `chmod +x` — делаем скрипт исполняемым.
    - Запуск скрипта `./script.py`.
3. **Уровень «Углублённый»: Интеграция и удалённая работа (25 мин)**
    - Встроенный редактор: nano через mc (F4).
    - Полный цикл: создать файл в mc, отредактировать в nano, дать права, запустить.
    - Редактирование прав через меню mc (F9 → File → Chmod).
    - Копирование файлов между роботом и ПК с помощью `scp`.
4. **Уровень «Исследовательский»: Автоматизация и анализ (20 мин)**
    - Bash-скрипт для создания структуры проекта одной командой.
    - Хронометраж: сравнение скорости работы в mc и терминале.

• Мини-исследование: почему стандартные права — 644 для файлов и 755 для папок.

5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Терминал — это мощно, но не всегда удобно

На прошлом занятии мы делали всё «вслепую»: вводили команды и получали текстовый вывод. Это очень гибко, но для обзора десятков файлов хочется видеть картину целиком.

Midnight Commander (mc) — легендарный файловый менеджер, который работает прямо в терминале. Он даёт удобный двухпанельный интерфейс, не требуя графической оболочки. А ещё сегодня мы разберёмся, почему файлы не запускаются просто так — и научимся управлять их правами.

2. Уровень «База»: Midnight Commander и права доступа

(Инструмент, навык, интерфейс)

Обязательный уровень. Мы освоим mc, научимся управлять файлами и запускать скрипты.

2.1. Запуск и интерфейс Midnight Commander

Подключитесь к роботу по SSH и запустите mc:

```
mc
```

Вы увидите две панели. Это как два открытых окна проводника одновременно. Панели можно менять местами и использовать для быстрого копирования.

Основы управления:

- **Tab** — переключение между левой и правой панелью. Активная панель подсвечена.
- **Стрелки** ↑↓ — перемещение по файлам и папкам.
- **Enter** — войти в папку или открыть файл (если это не папка, mc предложит программу для открытия).
- **cd ..** (в командной строке внизу) — выйти на уровень выше. Но в mc для этого есть специальная кнопка в верхней части панели — `/ ..` — просто наведите на неё и нажмите Enter.

Давайте осмотримся. В верхней части каждой панели написан текущий путь. Найдите свою домашнюю папку (`/home/ваш_логин`).

2.2. Функциональные клавиши: ваши главные инструменты

Внизу экрана подписаны действия для клавиш F1–F10. Самые важные:

1. **F3 (Просмотр)**: Выделите файл и нажмите F3. Вы увидите его содержимое. Это как `cat` , но с возможностью листать. Для выхода нажмите F3 ещё раз или F10.

    • *Практика:* Просмотрите файл `/etc/hostname` .

2. **F5 (Копирование)**: Выделите файл, нажмите F5. В появившемся окне будет указан путь во *второй* (неактивной) панели. Нажмите Enter для подтверждения.

    • *Практика:* Скопируйте файл `hello_robot.txt` (с прошлого занятия) из домашнего каталога `~/` в `/tmp`.

3. **F6 (Перемещение/Переименование)**: Работает как F5, но исходный файл удаляется.

    • *Практика:* Переименуйте копию в `/tmp` в `hello_backup.txt`.

4. **F8 (Удаление)**: Выделите файл, нажмите F8. Подтвердите удаление.

    • *Практика:* Удалите `hello_backup.txt` из `/tmp`.

5. **F10 (Выход)**: Выход из mc с возвратом в обычный терминал.

6. **F7 (Создание папки)**: Быстрое создание директории без команды `mkdir`.

2.3. Создание файлов и папок в mc

Создадим в домашней папке новую директорию для сегодняшнего занятия:

1. Нажмите **F7** и введите имя папки `lesson02`, нажмите Enter.
2. Войдите в папку `lesson02` (нажав Enter на ней).
3. Создайте новый файл: нажмите **Shift+F4**. Это горячая клавиша для создания и редактирования нового файла. Введите имя `hello_mc.txt`, нажмите Enter.
4. Откроется редактор (по умолчанию — встроенный, но мы настроим nano). Напишите текст `Я создал этот файл в Midnight Commander!`.
5. Нажмите F2 для сохранения и F10 для выхода из редактора.
6. Проверьте, что файл появился в панели mc, и просмотрите его по F3.

2.4. Права доступа и запуск скриптов

В Linux каждый файл имеет три набора прав: для владельца (u), группы (g) и всех остальных (o). А права бывают трёх типов: чтение (r), запись (w) и исполнение (x).

Посмотрите на вывод `ls -l`:

```
ls -l hello_mc.txt
# -rw-rw-r-- 1 pi pi 40 Sep 14 10:00 hello_mc.txt
```

Символы `rw-` означают: владелец может читать и писать, но не исполнять.

Чтобы Python-скрипт (или bash-скрипт) можно было запустить как программу, у него должен быть **флаг исполнения (x)**.

Практика:

1. Выйдите из mc (F10) или откройте встроенный терминал (Ctrl+O — переключение между панелями и терминалом).

2. Создайте простейший скрипт в nano:

   ```
   cd ~/lesson02
   nano my_prog.py
   ```

   Содержимое:

   ```
   #!/usr/bin/env python3
   print("Скрипт запущен!")
   print("Я нахожусь в:", __import__('os').getcwd())
   ```

3. Попробуйте запустить его как программу:

   ```
   ./my_prog.py
   ```

   Вы получите ошибку `Permission denied`! Это потому что нет прав на исполнение.

4. Делаем файл исполняемым:

   ```
   chmod +x my_prog.py
   ```

   Проверьте `ls -l` — теперь там `-rwxrwxr-x`. Буква `x` появилась.

5. Запускаем снова:

   ```
   ./my_prog.py
   ```

   Работает! Строка `#!/usr/bin/env python3` (шебанг) в начале скрипта подсказывает системе, какой интерпретатор использовать.

3. Уровень «Углублённый»: Интеграция и удалённая работа

(Что под капотом)

Связываем mc с nano, осваиваем копирование по сети и глубокое управление правами.

3.1. Настройка и использование редактора в mc

По умолчанию mc использует свой внутренний редактор. Но мы уже знаем nano. Давайте сделаем его редактором по умолчанию.

1. В mc нажмите **F9** для входа в верхнее меню.
2. Стрелками выберите **Command** → **Menu file** (или просто нажмите `o`).
3. Найдите строчку `edit=` и замените её на:

   ```
   edit=nano %s
   ```

4. Сохраните (F2) и выйдите (F10). Теперь при нажатии **F4** на файле будет открываться nano.

Полный цикл разработки в mc:

- **Shift+F4**: Создаём новый файл `sensor_test.py`.
- **Пишем код** (вы уже в nano):

```
#!/usr/bin/env python3
import random
sensor_value = random.uniform(20.0, 25.0)
print(f"Температура сенсора: {sensor_value:.1f} °C")
if sensor_value > 24.0:
    print("   Внимание: Перегрев!")
```

- Сохраняем и выходим из nano (Ctrl+O, Enter, Ctrl+X).
- В панели mc выделяем `sensor_test.py`.
- **Управление правами прямо из mc**: Нажмите **F9** → **File** → **Chmod** (или Ctrl+X, C). Вы увидите окошко с галочками `[x]`. Вы можете визуально выставить права. Например, поставьте все три `[x]` для владельца. Нажмите **Set**.
- Запускаем файл из mc: выделите его и нажмите **Enter**. Так как файл теперь исполняемый, mc его выполнит.
- Альтернативно, нажмите **Ctrl+O**, чтобы «уронить» панели и увидеть обычный терминал, и запустите `./sensor_test.py`.

3.2. SCP: копирование файлов между роботом и вашим ПК

Иногда код удобнее писать на своём ноутбуке в красивом редакторе, а потом закидывать на робота. **SCP (Secure Copy)** — аналог SSH для передачи файлов.

ВНИМАНИЕ: Эту команду вы выполняете на своём ПК (не на роботе)!

Формат команды:

```
scp <путь_к_файлу_на_пк> <логин>@<ip_робота>:<путь_на_роботе>
```

Задача:

1. На вашем ПК создайте файл `test_from_pc.txt` с любым содержимым.
2. Выполните копирование (замените `<логин>` и `<ip_робота>` на реальные):

   ```
   scp test_from_pc.txt student@192.168.1.100:~/lesson02/
   ```

3. Подключитесь к роботу по SSH и проверьте, что файл появился в `~/lesson02`.
4. **Скачивание с робота** (тоже с ПК):

   ```
   scp student@192.168.1.100:~/lesson02/my_prog.py ./
   ```

   Точка в конце означает «текущая папка».

4. Уровень «Исследовательский»: Автоматизация и системный анализ

(Новое применение, прорыв за границы)

Инженерный подход: не повторять рутину, а писать скрипты. И понимать, почему система работает именно так.

4.1. Bash-скрипт для создания структуры проекта

Мы создадим инструмент, который одной командой развернёт шаблон проекта. Это пригодится для будущих ROS2-пакетов.

Создайте скрипт `create_project.sh` в `~/lesson02/` (через `nano`).

```
#!/bin/bash
# Скрипт для создания структуры робототехнического проекта
PROJECT_NAME=$1

if [ -z "$PROJECT_NAME" ]; then
    echo "Использование: ./create_project.sh <имя_проекта>"
    exit 1
fi

echo "Создаю структуру проекта: $PROJECT_NAME"

# Создание папок одной командой
mkdir -p ~/ws/$PROJECT_NAME/{src,logs,config,scripts}

# Создание README.md с кратким описанием
cat > ~/ws/$PROJECT_NAME/README.md << EOF
# $PROJECT_NAME

## Описание
Проект для изучения мобильной робототехники.

## Структура
- \`src/\` - исходный код
- \`logs/\` - файлы журналов
- \`config/\` - конфигурационные файлы
- \`scripts/\` - вспомогательные скрипты

## Сборка и запуск
TODO: добавить инструкции
EOF

echo "Готово! Дерево проекта:"
tree ~/ws/$PROJECT_NAME 2>/dev/null || ls -R ~/ws/$PROJECT_NAME
```

Сделайте скрипт исполняемым и протестируйте:

```
chmod +x create_project.sh
./create_project.sh my_test_project
ls -la ~/ws/my_test_project
```

4.2. Хронометраж: mc против терминала

Что быстрее: выполнить операцию в визуальном менеджере или написать команду? Давайте проверим научно.

Задача: Скопировать 10 файлов из одной папки в другую.

1. Подготовим 10 файлов в папке `~/lesson02/test_source`:

```
mkdir -p ~/lesson02/test_source ~/lesson02/test_dest_term ~/lesson02/test_dest_mc
for i in {1..10}; do echo "Файл $i" > ~/lesson02/test_source/file_$i.txt; done
```

2. **Тест в терминале:**

```
time cp ~/lesson02/test_source/* ~/lesson02/test_dest_term/
```

   Запомните время (параметр `real`).

3. **Тест в mc:**

   • Запустите mc.

   • В левой панели откройте `~/lesson02/test_source`.

   • В правой панели — `~/lesson02/test_dest_mc`.

   • Нажмите `+` на цифровой клавиатуре (или `Insert` на ноутбуке), чтобы отметить все файлы.

   • Нажмите F5 и Enter.

   • Время придётся засекать по секундомеру. Был ли процесс быстрее субъективно?

Вывод: Терминал выигрывает для однотипных массовых операций (`*` — маска). mc выигрывает для выборочного копирования разнородных файлов. Инженер должен владеть обоими инструментами.

4.3. Мини-исследование: почему права 644 для файлов и 755 для папок?

Вы наверняка видели эти «магические числа». Они называются **восьмеричной записью прав**.

- `4` — чтение (r)
- `2` — запись (w)
- `1` — исполнение (x)

Число собирается как сумма:

- `7` = 4+2+1 (rwx)
- `6` = 4+2 (rw-)
- `5` = 4+1 (r-x)
- `4` = 4 (r--)

Три цифры: для владельца, группы, остальных.

Исследовательская задача:

1. Создайте папку `mkdir test_perm_dir` и файл `touch test_perm_file` в `/tmp`.

2. Посмотрите их права по умолчанию: `ls -ld /tmp/test_perm_dir /tmp/test_perm_file`.
    - Скорее всего, вы увидите `rwxr-xr-x` (755) для папки и `rw-r--r--` (644) для файла.
3. **Почему так?**
    - **Для папки «исполнение» (x) означает право войти внутрь (`cd`)**. Без `x` на папку вы не сможете перейти в неё, даже если у вас есть права на чтение (`r`), которые позволяют лишь увидеть список файлов. Поэтому папки по умолчанию создаются с `x`.
    - **Для файла `x` — это право выполнить его как программу.** Ради безопасности Linux никогда не делает новый текстовый файл или картинку исполняемым по умолчанию. Вдруг вы скачали вирус? Вы должны явно дать права через `chmod +x`, подтверждая своё намерение.

Проверьте гипотезу:

1. Уберите право исполнения с папки:

    ```
    chmod 644 /tmp/test_perm_dir
    ```

2. Попробуйте войти в неё:

    ```
    cd /tmp/test_perm_dir
    ```

    Система скажет `Permission denied`, хотя `ls /tmp/test_perm_dir` сработает.

3. Верните права обратно: `chmod 755 /tmp/test_perm_dir`.

5. Выводы по занятию

1. **Midnight Commander — мост между мирами.** Вы освоили инструмент, который сочетает удобство визуальной оболочки с мощью терминала. Теперь вы можете быстро ориентироваться в файловой системе робота.
2. **Права доступа — азбука безопасности.** Вы узнали, что такое `rwx` и как они записываются числами. Теперь вы понимаете, почему скрипты нужно «запускать» явно (`chmod +x`), а папки — нет.
3. **SCP — перенос данных без флешки.** Вы освоили удалённое копирование файлов между ПК и роботом по тому же защищённому каналу SSH.
4. **Инженерное мышление — это автоматизация.** Мы создали скрипт, который штампует заготовки проектов. Это первый шаг к тому, чтобы рутину делали машины, а мы — решали творческие задачи.
5. **Критическое мышление — это проверка гипотез.** Мы не просто запомнили, что `644` и `755` — «стандарт». Мы провели эксперимент и поняли **почему** это так.

Домашнее задание:

- **Базовое**: Используя mc, создайте в домашней папке структуру `week02/exercise02` и поместите туда копию файла `/etc/hostname`.
- **Продвинутое**: Напишите на своём ПК Python-программу, которая выводит «Hello from my PC!», и скопируйте её на робота с помощью `scp`. Запустите эту программу на роботе, сделав её исполняемой.

- **Исследовательское**: Изучите команду `umask`. Введите `umask` в терминале и объясните, как значение `022` связано с тем, что новые файлы получают права `644`, а папки `755`.

Краткое введение в Python и ООП

Занятие 3: Повторение Python — переменные, типы, ввод/вывод, turtle

Содержание занятия: Основы синтаксиса Python и модуль turtle. Первый рисунок

Цели занятия

- **Образовательная**: Вспомнить и систематизировать базовые конструкции Python: переменные, основные типы данных, ввод и вывод. Познакомиться с графическим модулем turtle как инструментом визуализации алгоритмов.
- **Развивающая**: Перейти от «слепого» текстового ввода-вывода к наглядному результату в виде рисунка. Заложить связь «код → геометрия».
- **Воспитательная**: Показать красоту программирования через мгновенную визуальную обратную связь.

План занятия (90 минут)

1. **Организационный момент и цели (5 мин)**
2. **Уровень «База»: Переменные, типы и первый квадрат (40 мин)**
   - Python как калькулятор: переменные, типы int, float, str, bool.
   - Ввод и вывод: функции `input()` и `print()`.
   - Знакомство с turtle: холст, черепашка, система координат.
   - Команды движения: `forward()`, `backward()`, `left()`, `right()`.
   - Первый рисунок: квадрат из последовательных команд.
3. **Уровень «Углублённый»: Параметризация и циклы (25 мин)**
   - Параметризация размера стороны через переменную.
   - Изменение цвета линии: `pencolor()`.
   - Цикл `for` для рисования квадрата.
   - Разноцветный квадрат: смена цвета на каждой стороне.
4. **Уровень «Исследовательский»: Функции и цветовые модели (15 мин)**
   - Функция `square(size, color)` с параметрами по умолчанию.
   - Рисование сетки из квадратов вложенными циклами.
   - Мини-исследование: RGB-модель в turtle и плавный градиент.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Python как язык управления роботом

На прошлых занятиях мы учились «общаться» с операционной системой робота через терминал. Теперь мы переходим к программированию его поведения.

Почему Python?

- Это основной язык в ROS2 — нашей робототехнической платформе.
- Простой и читаемый синтаксис, похожий на обычный английский.
- Огромное количество библиотек: от компьютерного зрения до нейросетей.

Сегодня мы начнём с азов, но с прицелом на робототехнику. Вместо скучного вывода в консоль мы будем управлять графической «черепашкой» — это как виртуальный робот, который рисует на экране. Все принципы (координаты, углы, перемещения) напрямую переносятся на управление настоящим роботом.

2. Уровень «База»: Переменные, типы данных и первый рисунок

(Инструмент, навык, интерфейс)

Обязательный уровень для всех. Вспоминаем Python и делаем первый шаг в графику.

2.1. Python как калькулятор: переменные и типы данных

Подключитесь к роботу по SSH. Откройте интерпретатор Python (интерактивный режим) (это можно делать и на ПК с симулятором):

```
python3
```

Вы увидите `>>>` — приглашение Python. Попробуем:

```
# Переменные — это именованные «коробочки» для хранения данных
x = 10          # int — целое число
y = 3.14        # float — число с плавающей точкой
name = "Robot"  # str — строка (текст)
is_on = True    # bool — логическое значение (да/нет)

# Операции
print(x + 5)        # 15
print(x / 3)        # 3.333... (обычное деление)
print(x // 3)       # 3 (целочисленное деление)
print(x ** 2)       # 100 (возведение в степень)
print(name + " XYZ") # Robot XYZ (склеивание строк)
print(f"x = {x}, y = {y}")  # f-строка (форматирование)
```

Практика: Вычислите в интерпретаторе, сколько проедет робот за 5 секунд при скорости 0.2 м/с. (`distance = 0.2 * 5`).

2.2. Ввод и вывод: общение с программой

Выйдите из интерпретатора (`exit()` или Ctrl+D) и создайте файл `hello_input.py` :

```
mkdir -p ~/ws/rbt/scripts
nano ~/ws/rbt/scripts/hello_input.py
```

```
#!/usr/bin/env python3
# Программа для знакомства
user_name = input("Введите ваше имя: ")
age = input("Введите ваш возраст: ")

# input всегда возвращает строку. Для чисел нужно преобразование:
age_int = int(age)
year_of_birth = 2026 - age_int

print(f"Привет, {user_name}!")
print(f"Ты примерно {year_of_birth} года рождения.")
```

Запустите:

```
python3 ~/ws/rbt/scripts/hello_input.py
```

> **Запомните:** `input()` всегда даёт строку (`str`). Для математики преобразуйте: `int()` — в целое, `float()` — в дробное.

2.3. Знакомство с turtle: виртуальный робот-художник

turtle — это модуль Python, который создаёт графическое окно с «черепашкой». Черепашка понимает простые команды: иди вперёд, поверни налево, подними хвост. Это отличная модель для понимания того, как мы будем управлять роботом (только робот будет настоящим, с колёсами).

> **Важно для удалённой работы:** Окно turtle требует графический дисплей. Если вы подключены по SSH, окно не откроется. Есть два варианта:
>
> 1. **Запускать код на ПК**, если у вас установлен Python и turtle.
> 2. Использовать **X11 forwarding**: `ssh -X student@<ip_робота>`, затем `python3 script.py`. (Учитель подскажет, настроено ли это).
> 3. Писать код на роботе, а запускать через VNC (если настроен).
>
> *В этом занятии предполагается, что вы можете запустить графику локально или через X11-forwarding.*

Создайте файл `first_square.py`:

```
nano ~/ws/rbt/scripts/first_square.py
```

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

# Создаём экран и черепашку
screen = turtle.Screen()
screen.setup(700, 700) #для форвада
screen.screensize(600, 600) #для форварда
screen.title("Мой первый рисунок")
t = turtle.Turtle()
```

```
# Настройки черепашки
t.shape("turtle")  # вид: стрелка, turtle, circle, square
t.speed(2)          # скорость: 1 (медленно) ... 10 (быстро)

# Рисуем квадрат (сторона 100 пикселей)
t.forward(100)  # вперёд на 100
t.left(90)      # поворот налево на 90 градусов
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)

# Закрываем окно по клику
screen.exitonclick()
```

Запустите программу. Получился квадрат? Отлично! Это ваш первый «робот», выполняющий геометрическую миссию.

Основные команды turtle:

- `forward(distance)` / `fd(distance)` — вперёд
- `backward(distance)` / `bk(distance)` — назад
- `right(angle)` / `rt(angle)` — направо
- `left(angle)` / `lt(angle)` — налево
- `penup()` / `pu()` — поднять перо (не рисовать)
- `pendown()` / `pd()` — опустить перо (рисовать)

3. Уровень «Углублённый»: Параметризация и циклы

(Что под капотом)

Для тех, кто справился с базой. Делаем код гибким и лаконичным.

3.1. Параметризация: размер через переменную

В предыдущей программе сторона квадрата «зашита» как 100. Чтобы нарисовать квадрат другого размера, нужно менять 4 строки. Сделаем размер управляемым:

Создайте `square_variable.py`:

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

screen = turtle.Screen()
t = turtle.Turtle()
t.speed(3)

side = 150  # ⟵ меняем только здесь!
```

```
t.forward(side)
t.left(90)
t.forward(side)
t.left(90)
t.forward(side)
t.left(90)
t.forward(side)
t.left(90)

screen.exitonclick()
```

Поменяйте `side` на 50, запустите снова. Квадрат стал меньше.

3.2. Цвет и цикл for

4 раза повторять `forward` и `left` — это идеальный случай для цикла `for`.

Создайте `color_square.py`:

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

screen = turtle.Screen()
t = turtle.Turtle()
t.speed(5)
t.pensize(3)  # толщина линии

side = 120
colors = ["red", "green", "blue", "orange"]  # список цветов

for i in range(4):           # i будет 0, 1, 2, 3
    t.pencolor(colors[i])    # берём цвет по индексу
    t.forward(side)
    t.left(90)

screen.exitonclick()
```

Что мы сделали:

- `range(4)` даёт последовательность 0, 1, 2, 3 — ровно 4 итерации.
- `colors[i]` — обращение к элементу списка по индексу. На первом шаге `colors[0]` — "red", на втором `colors[1]` — "green", и так далее.
- `pencolor()` меняет цвет пера.

4. Уровень «Исследовательский»: Функции и цветовые модели

(Новое применение, прорыв за границы)

Для будущих архитекторов робототехнического ПО.

4.1. Функция square() с параметрами по умолчанию

Обернём рисование квадрата в функцию, чтобы использовать многократно. Создайте `square_func.py`:

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

def square(t, size, color="black"):
    """Рисует квадрат заданного размера и цвета."""
    t.pencolor(color)
    for _ in range(4):
        t.forward(size)
        t.left(90)

screen = turtle.Screen()
screen.setup(700, 700) #для форвада
screen.screensize(600, 600) #для форварда
t = turtle.Turtle()
t.speed(7)

# Использование функции с разными параметрами
square(t, 100, "red")       # красный квадрат 100x100
t.penup()
t.goto(150, 0)               # перемещаемся в новую точку
t.pendown()
square(t, 50, "blue")       # синий квадрат 50x50
t.penup()
t.goto(0, -150)
t.pendown()
square(t, 75)               # цвет по умолчанию (чёрный)

screen.exitonclick()
```

4.2. Сетка из квадратов вложенными циклами

Создайте `grid.py`:

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

def square(t, size):
    for _ in range(4):
        t.forward(size)
        t.left(90)

screen = turtle.Screen()
screen.tracer(0)  # отключаем анимацию для скорости
t = turtle.Turtle()
t.speed(0)
t.hideturtle()

cell_size = 30
grid_size = 5  # сетка 5x5
```

```
for row in range(grid_size):
    for col in range(grid_size):
        # Вычисляем позицию клетки
        x = col * cell_size - (grid_size * cell_size) / 2
        y = row * cell_size - (grid_size * cell_size) / 2

        t.penup()
        t.goto(x, y)
        t.pendown()

        # Цвет зависит от позиции
        if (row + col) % 2 == 0:
            t.fillcolor("lightblue")
        else:
            t.fillcolor("lightcoral")

        t.begin_fill()
        square(t, cell_size)
        t.end_fill()

screen.update()  # показываем результат
screen.exitonclick()
```

Что мы освоили:

- Вложенные циклы (`for row in ...` внутри `for col in ...`) для двумерных структур.
- `tracer(0)` и `update()` — приём для ускорения отрисовки (полезно для сложной графики).
- Заливка фигур: `begin_fill()` / `end_fill()` .

4.3. Мини-исследование: RGB-модель и плавный градиент

До сих пор мы использовали именованные цвета: `"red"` , `"blue"` . Но turtle поддерживает и RGB-формат: `(red, green, blue)` , где каждое число от 0 до 255 (или от 0.0 до 1.0).

Создайте `rgb_gradient.py` :

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

screen = turtle.Screen()
screen.colormode(255)  # включаем режим RGB 0-255
screen.tracer(0)

t = turtle.Turtle()
t.speed(0)
t.hideturtle()
t.pensize(2)

def draw_color_line(t, y, color):
    """Рисует горизонтальную линию заданным цветом."""
    t.penup()
    t.goto(-300, y)
    t.pendown()
```

```
    t.pencolor(color)
    t.forward(600)

# Градиент от красного (255,0,0) к синему (0,0,255)
steps = 100
for i in range(steps):
    ratio = i / (steps - 1)  # от 0.0 до 1.0
    r = int(255 * (1 - ratio))  # красный убывает
    g = 0
    b = int(255 * ratio)        # синий растёт

    y = -200 + i * 4  # распределяем линии по вертикали
    draw_color_line(t, y, (r, g, b))

screen.update()
screen.exitonclick()
```

Исследовательские вопросы:

1. Как получить градиент от зелёного к жёлтому? (Подсказка: красный растёт, зелёный постоянен).
2. Что будет, если смешивать все три компонента одновременно?
3. Почему RGB-модель так популярна в компьютерной графике? (Связано с устройством человеческого глаза и пикселей монитора).

Это знание пригодится нам позже, когда мы будем обрабатывать изображения с камеры робота в OpenCV.

5. Выводы по занятию

1. **Python — это просто и мощно.** Мы повторили переменные, типы данных, ввод/вывод — фундамент для любой программы.
2. **Turtle — тренажёр робототехники.** Управление «черепашкой» через команды `forward`, `left`, `right` — это точная модель того, как мы будем управлять роботом: задавать линейную и угловую скорость.
3. **Циклы — враг рутины.** Вместо 4 (или 360) строк кода мы используем `for i in range( ... )`. В робототехнике это превратится в циклы обработки данных с датчиков.
4. **Функции — кирпичики сложных систем.** Функция `square()` инкапсулирует логику. Мы можем переиспользовать её в сетках, узорах — так же как будем переиспользовать модули управления движением.
5. **Цвет — это данные.** RGB-модель — это три числа. Анализ и генерация этих чисел — первый шаг к компьютерному зрению, где каждый пиксель изображения — это вектор RGB.

Связь со следующим занятием: Мы научились рисовать квадрат циклом `for`. На следующем занятии мы расширим этот принцип на любые правильные многоугольники — треугольники, шестиугольники, окружности.

Домашнее задание:

- **Базовое**: Нарисуйте прямоугольник (не квадрат) со сторонами 150 и 80 пикселей.

- **Продвинутое**: Нарисуйте «лесенку» из 5 ступенек, используя цикл `for`.
- **Исследовательское**: Создайте функцию `rectangle(t, width, height, color)`, которая рисует залитый прямоугольник. С её помощью нарисуйте диаграмму из 3 столбцов разной высоты и цвета.

Занятие 4: Циклы for и range. Правильные многоугольники

Содержание занятия: Параметризация геометрических фигур с помощью цикла for

Цели занятия

- **Образовательная**: Освоить цикл `for` с функцией `range()` как универсальный инструмент для повторения действий. Научиться рисовать правильные многоугольники с произвольным количеством сторон.
- **Развивающая**: Установить связь между количеством повторений в цикле и геометрическими свойствами фигуры (угол поворота = 360 / количество сторон).
- **Воспитательная**: Показать, как одна универсальная программа заменяет множество частных решений.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Цикл for и range (35 мин)**
    - Синтаксис цикла `for i in range(n)`.
    - Рисование квадрата через цикл: 4 итерации вместо 4 команд.
    - Треугольник через `range(3)`, шестиугольник через `range(6)`.
    - Угол поворота как функция количества сторон: `360 / n`.
3. **Уровень «Углублённый»: Параметризация и цвет (25 мин)**
    - Параметризация: переменные `sides` и `size`.
    - Одна программа для рисования любого правильного многоугольника.
    - Список цветов и смена `pencolor` внутри цикла.
4. **Уровень «Исследовательский»: Функции и окружность (20 мин)**
    - Функция `polygon(sides, size)`.
    - Окружность как многоугольник с большим числом сторон.
    - Мини-исследование: при каком минимальном `sides` многоугольник визуально неотличим от окружности.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: От частного к общему

На прошлом занятии мы нарисовали квадрат четырьмя командами `forward()` и `left()`. Это работало для квадрата. Но что если нам понадобится треугольник? Пятиугольник? Десятиугольник?

Копировать и вставлять команды — путь в никуда. Настоящий инженер ищет **закономерность** и поручает повторяющуюся работу циклу. Сегодня мы научимся рисовать **любой** правильный многоугольник одной программой.

2. Уровень «База»: Цикл for и range

(Инструмент, навык, интерфейс)

Обязательный уровень. Осваиваем цикл `for` и рисуем разные фигуры.

2.1. Синтаксис цикла for и функция range

Цикл `for` повторяет блок кода заданное количество раз. Функция `range(n)` генерирует последовательность чисел от 0 до n-1.

Создайте файл `for_demo.py`:

```
nano ~/ws/rbt/scripts/for_demo.py
```

```
#!/usr/bin/env python3
# Демонстрация цикла for и range

print("range(5) генерирует числа:")
for i in range(5):
    print(f"Итерация {i}: Привет, робот!")

print("\nrange(2, 6) генерирует числа от 2 до 5:")
for i in range(2, 6):
    print(f"i = {i}")

print("\nrange(0, 10, 2) генерирует чётные числа:")
for i in range(0, 10, 2):
    print(f"i = {i}")
```

Запустите:

```
python3 ~/ws/rbt/scripts/for_demo.py
```

Три формы range:

- `range(n)` — числа от 0 до n-1 (n штук).
- `range(start, stop)` — от start до stop-1.
- `range(start, stop, step)` — с шагом step.

2.2. Квадрат через цикл for

Заменим 4 строки с `forward` и `left` на цикл из 4 итераций.

Создайте `square_for.py`:

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

screen = turtle.Screen()
screen.title("Квадрат через for")
t = turtle.Turtle()
t.speed(3)
t.pensize(2)

# 4 итерации — по одной на каждую сторону
for i in range(4):
    t.forward(120)
    t.left(90)

screen.exitonclick()
```

Результат тот же, но код стал короче. А главное — мы видим структуру: квадрат = 4 раза {вперёд, поворот на 90}.

2.3. Треугольник и шестиугольник: меняем число повторений и угол

Ключевая идея: Для любого правильного N-угольника:

- Количество сторон = N, значит `range(N)`.
- Сумма внешних углов = 360°, значит угол поворота = `360 / N`.

Создайте `polygons_basic.py`:

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

screen = turtle.Screen()
screen.title("Треугольник и шестиугольник")
t = turtle.Turtle()
t.speed(3)
t.pensize(2)

# Треугольник: 3 стороны, угол = 360 / 3 = 120
t.pencolor("red")
for i in range(3):
    t.forward(150)
    t.left(120)

# Переместимся правее
t.penup()
t.goto(200, 0)
t.pendown()
```

```
# Шестиугольник: 6 сторон, угол = 360 / 6 = 60
t.pencolor("blue")
for i in range(6):
    t.forward(80)
    t.left(60)

screen.exitonclick()
```

Проверьте себя:

- Какой угол нужен для пятиугольника? (360 / 5 = 72°)
- Для десятиугольника? (360 / 10 = 36°)

3. Уровень «Углублённый»: Параметризация и цветовые списки

(Что под капотом)

Делаем одну программу, которая рисует любой многоугольник по заданным параметрам.

3.1. Параметризация через переменные

Вместо того чтобы «зашивать» число 3 или 6 в код, сделаем переменные `sides` и `size`.

Создайте `any_polygon.py`:

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

screen = turtle.Screen()
screen.title("Любой правильный многоугольник")
t = turtle.Turtle()
t.speed(5)
t.pensize(2)

# Параметры — меняем только их!
sides = 7      # количество сторон
size = 100     # длина стороны

angle = 360 / sides  # угол поворота вычисляется автоматически

for i in range(sides):
    t.forward(size)
    t.left(angle)

print(f"Нарисован {sides}-угольник со стороной {size}")
screen.exitonclick()
```

Поменяйте `sides` на 8 — получите восьмиугольник. На 12 — двенадцатиугольник. Одна программа — бесконечное разнообразие фигур.

3.2. Список цветов для каждой стороны

Мы уже использовали список цветов на прошлом занятии. Теперь применим его к N-угольнику.

Создайте `colorful_polygon.py`:

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

screen = turtle.Screen()
screen.title("Разноцветный многоугольник")
t = turtle.Turtle()
t.speed(5)
t.pensize(3)

sides = 9
size = 100
angle = 360 / sides

# Список цветов — больше, чем сторон, чтобы хватило на любую фигуру
colors = ["red", "orange", "gold", "green", "blue", "purple", "violet", "p
ink", "cyan", "magenta"]

for i in range(sides):
    t.pencolor(colors[i])  # цвет меняется каждую итерацию
    t.forward(size)
    t.left(angle)

screen.exitonclick()
```

Что будет, если `sides` больше, чем цветов в списке? Будет ошибка `IndexError`. Подумайте, как этого избежать. (Подсказка: `colors[i % len(colors)]` — остаток от деления зациклит список).

4. Уровень «Исследовательский»: Функция polygon и предел окружности

(Новое применение, прорыв за границы)

Для будущих исследователей, которые хотят не просто использовать, а создавать инструменты.

4.1. Функция polygon(sides, size)

Обернём нашу универсальную логику в функцию. Создайте `polygon_func.py`:

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

def polygon(t, sides, size):
    """Рисует правильный многоугольник с заданным числом сторон и
размером."""
    angle = 360 / sides
    for i in range(sides):
        t.forward(size)
        t.left(angle)

# Тест функции
screen = turtle.Screen()
screen.title("Функция polygon")
```

```
t = turtle.Turtle()
t.speed(7)

# Рисуем 3 разные фигуры одной функцией
t.pencolor("red")
polygon(t, 3, 100)   # треугольник

t.penup()
t.goto(150, 0)
t.pendown()
t.pencolor("green")
polygon(t, 5, 70)    # пятиугольник

t.penup()
t.goto(-150, -150)
t.pendown()
t.pencolor("blue")
polygon(t, 8, 50)    # восьмиугольник

screen.exitonclick()
```

4.2. Окружность как многоугольник

Что будет, если `sides` сделать очень большим? Визуально многоугольник начнёт напоминать окружность.

Создайте `circle_as_polygon.py`:

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

def polygon(t, sides, size):
    angle = 360 / sides
    for i in range(sides):
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
screen.title("Многоугольник → Окружность")
screen.tracer(0)  # ускоряем отрисовку
t = turtle.Turtle()
t.speed(0)

# Рисуем 4 «окружности» с разным числом сторон
test_sides = [12, 24, 48, 96]
start_x = -300
y = 0

for sides in test_sides:
    t.penup()
    t.goto(start_x, y)
    t.pendown()

    # Чем больше сторон, тем меньше длина каждой стороны
    # чтобы общий размер оставался примерно одинаковым
```

```
    size = 600 / sides

    polygon(t, sides, size)

    # Подписываем
    t.penup()
    t.goto(start_x, -50)
    t.write(f"sides={sides}", align="center", font=("Arial", 12, "normal")
)

    start_x += 180

screen.update()
screen.exitonclick()
```

4.3. Мини-исследование: предел визуальной неотличимости

Вопрос: При каком минимальном количестве сторон многоугольник выглядит как идеальная окружность?

Методика исследования:

1. Напишите программу, которая рисует многоугольники с `sides` = 8, 12, 16, 20, 24, 32, 48, 64.
2. Для каждого случая зафиксируйте визуальное впечатление:
   - Отчётливо видны грани.
   - Грани заметны, но фигура уже «круглая».
   - Грани едва различимы.
   - Выглядит как идеальная окружность.
3. Заполните таблицу:

sides	Визуальное впечатление
8	Отчётливый восьмиугольник
12	...
16	...
20	...
24	...
32	...
48	...
64	...

Ожидаемый результат: Большинство людей перестают замечать грани при 24–32 сторонах. Это зависит от размера фигуры и расстояния до экрана.

Почему это важно для робототехники? Многие алгоритмы (планирование пути, SLAM) аппроксимируют сложные формы многоугольниками. Выбор правильного количества вершин — компромисс между точностью и вычислительной сложностью. Этот же принцип мы увидим при обработке облака точек с лидара.

5. Выводы по занятию

1. **Цикл for — универсальный инструмент повторения.** `for i in range(n)` позволяет выполнить блок кода ровно n раз. Это основа автоматизации.
2. **Геометрия = формула + цикл.** Для любого правильного многоугольника: `range(стороны)`, угол поворота = `360 / стороны`. Одна закономерность заменяет бесконечное множество частных случаев.
3. **Параметризация — шаг к инженерии.** Переменные `sides` и `size` делают программу гибкой. Так же мы будем параметризовать скорость и дистанцию для робота.
4. **Функции — переиспользуемые блоки.** `polygon(t, sides, size)` можно вызывать когда угодно и где угодно. На следующих занятиях мы соберём библиотеку таких функций.
5. **Окружность — это многоугольник с большим N.** Это не просто факт, а инженерный принцип аппроксимации, который лежит в основе цифрового моделирования реального мира.

Связь со следующим занятием: Мы освоили цикл `for`, когда число повторений известно заранее. Но что делать, если мы не знаем, сколько шагов потребуется? Например, рисовать спираль, пока она не упрётся в границу экрана? Для этого существует цикл `while` — тема следующего занятия.

Домашнее задание:

- **Базовое**: Нарисуйте девятиугольник со стороной 80.
- **Продвинутое**: Нарисуйте композицию из трёх вложенных многоугольников (например, треугольник внутри шестиугольника внутри двенадцатиугольника) с общим центром.
- **Исследовательское**: Проведите исследование «предела окружности» с замером визуального впечатления для разных `sides`. Оформите результат в виде комментария в коде или небольшого отчёта.

Занятие 5: Циклы while, окружность, спираль, break, continue

Содержание занятия: Цикл с неизвестным числом повторений. Спирали и преждевременный выход

Цели занятия

- **Образовательная**: Освоить цикл `while` и понять его принципиальное отличие от `for`. Изучить операторы `break` и `continue` для управления потоком выполнения цикла.
- **Развивающая**: Применить цикл `while` для создания динамических рисунков (спиралей), длина которых заранее не фиксирована.
- **Воспитательная**: Показать, что выбор между `for` и `while` — это инженерное решение, зависящее от задачи.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Цикл while и окружность (35 мин)**
   • Синтаксис цикла `while`.
   • Рисование окружности через `while` с фиксированным числом шагов (360).
   • Отличие `while` от `for`: когда число повторений неизвестно заранее.
3. **Уровень «Углублённый»: break, continue и спираль (25 мин)**
   • `break` — выход из цикла досрочно.
   • `continue` — пропуск оставшейся части итерации.
   • Спираль: увеличение шага, остановка при выходе за границу.
4. **Уровень «Исследовательский»: Цветоспираль и анализ (20 мин)**
   • Спираль с плавным изменением цвета и толщины.
   • Мини-исследование: влияние шага радиуса на плотность витков.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Когда мы не знаем, сколько шагов впереди

На прошлом занятии мы использовали `for i in range(n)`. Это удобно, когда мы **точно знаем**, сколько будет повторений: у квадрата их 4, у шестиугольника — 6.

Но представьте, что робот едет вперёд, пока не встретит препятствие. Мы не знаем заранее, сколько секунд ему ехать — 2 или 20. Или мы рисуем спираль, пока она не упрётся в край экрана. Для таких ситуаций нужен **цикл `while`** — он выполняется, пока верно некоторое **условие**.

2. Уровень «База»: Цикл while и окружность

(Инструмент, навык, интерфейс)

Обязательный уровень. Осваиваем синтаксис `while` и рисуем окружность.

2.1. Синтаксис цикла while

Цикл `while` имеет простое правило: «Пока условие истинно, выполняй тело цикла».

Создайте файл `while_demo.py`:

```
nano ~/ws/rbt/scripts/while_demo.py
```

```
#!/usr/bin/env python3
# Демонстрация цикла while

print("Обратный отсчёт:")
counter = 5
while counter > 0:
    print(f"  Осталось: {counter}")
```

```
    counter = counter - 1  # уменьшаем счётчик
print("  Пуск!")

print("\nБесконечный цикл с ручным прерыванием:")
print("Чтобы остановить программу, нажмите Ctrl+C")
# while True:
#     pass  # заглушка — не делаем ничего, чтобы не зависло
print("  (этот цикл закомментирован для безопасности)")
```

Ключевое отличие от for:

- `for` — «сделай 10 раз» (число повторений известно).
- `while` — «делай, пока условие верно» (число повторений неизвестно, может быть 0).

Опасность: Если условие всегда истинно, получится **бесконечный цикл**. Программа зависнет. Всегда проверяйте, что условие когда-нибудь станет ложным!

2.2. Окружность через while

Окружность — это 360 маленьких шагов с поворотом на 1°. Конечно, мы знаем, что шагов будет ровно 360, и могли бы использовать `for i in range(360)`. Но давайте применим `while` со счётчиком — это подготовит нас к спирали, где число шагов неизвестно.

Создайте `circle_while.py`:

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

screen = turtle.Screen()
screen.title("Окружность через while")
t = turtle.Turtle()
t.speed(5)
t.pensize(2)
t.pencolor("blue")

# Будем считать шаги, пока их не станет 360
steps_done = 0
while steps_done < 360:
    t.forward(2)    # длина шага (чем меньше, тем глаже окружность)
    t.left(1)       # поворот на 1 градус
    steps_done = steps_done + 1

print("Окружность нарисована!")
screen.exitonclick()
```

Этот же результат можно получить через `for i in range(360)`. Здесь `while` использован для демонстрации. Настоящая сила `while` — дальше.

2.3. Отличие while от for на практике

Правило выбора:

- Если вы **знаете** точное число повторений → используйте `for i in range(n)`.

- Если число повторений зависит от **условия**, которое вычисляется в процессе → используйте `while`.

Пример робототехнической задачи: «Робот едет вперёд, пока расстояние до препятствия больше 30 см». Мы не можем знать заранее, сколько секунд это займёт. Это идеальный случай для `while distance > 0.3:`.

3. Уровень «Углублённый»: break, continue и спираль

(Что под капотом)

Осваиваем управляющие операторы и применяем цикл `while` для настоящей динамики.

3.1. break — аварийный выход из цикла

Иногда нужно прекратить цикл досрочно. Например, робот заметил препятствие. Для этого есть `break`.

Создайте `break_demo.py`:

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

screen = turtle.Screen()
screen.setup(width=500, height=500)
t = turtle.Turtle()
t.speed(7)

# Робот едет вправо, пока не упрётся в край экрана
t.penup()
t.goto(-200, 0)
t.pendown()

while True:  # бесконечный цикл
    t.forward(5)
    x = t.xcor()  # текущая X-координата
    if x > 200:
        print(f"Достигнута граница! x = {x:.0f}")
        break  # немедленно выходим из цикла

print("Цикл завершён.")
screen.exitonclick()
```

3.2. continue — пропустить итерацию

`continue` не выходит из цикла, а переходит к следующей итерации, пропуская всё, что ниже.

Пример: рисуем пунктирную линию, пропуская каждый 10-й шаг.

Создайте `continue_demo.py`:

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

```
screen = turtle.Screen()
t = turtle.Turtle()
t.speed(7)
t.pensize(3)

t.penup()
t.goto(-200, 0)
t.pendown()

step = 0
while step < 100:
    step = step + 1
    if step % 10 == 0:  # каждый 10-й шаг
        t.penup()
        t.forward(4)
        t.pendown()
        continue  # пропускаем рисование, но цикл продолжается
    t.forward(4)

screen.exitonclick()
```

Получится линия с пробелами.

3.3. Спираль: рост шага и выход по границе

Классическая задача для `while`: рисуем спираль, увеличивая длину шага, пока черепашка не выйдет за пределы экрана.

Создайте `spiral.py`:

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

screen = turtle.Screen()
screen.setup(width=600, height=600)
screen.title("Спираль с выходом по границе")
t = turtle.Turtle()
t.speed(7)
t.pensize(1)

step = 5  # начальная длина шага
angle = 89  # угол поворота (чуть меньше 90 даёт плавную спираль)

while True:
    t.forward(step)
    t.left(angle)
    step = step + 1  # каждый шаг чуть длиннее

    # Проверяем выход за границы экрана
    x, y = t.xcor(), t.ycor()
    if abs(x) > 280 or abs(y) > 280:
        print(f"Выход за границу! x={x:.0f}, y={y:.0f}, шагов={step}")
        break
```

```
print(f"Спираль остановлена. Финальная длина шага: {step}")
screen.exitonclick()
```

Эксперимент: Попробуйте менять `angle`:

- `angle = 90` — квадратная спираль.
- `angle = 120` — треугольная спираль.
- `angle = 89` — плавная архимедова спираль.

4. Уровень «Исследовательский»: Цветоспираль и анализ витков

(Новое применение, прорыв за границы)

Для исследователей, соединяющих математику и эстетику кода.

4.1. Спираль с изменяющимся цветом и толщиной

Соединим всё вместе: `while`, изменение параметров, RGB-цвета.

Создайте `rainbow_spiral.py`:

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

screen = turtle.Screen()
screen.setup(width=600, height=600)
screen.colormode(255)  # RGB 0-255
screen.bgcolor("black")  # тёмный фон для эффекта
screen.title("Радужная спираль")

t = turtle.Turtle()
t.speed(0)  # максимальная скорость
t.hideturtle()

step = 3
angle = 89
hue = 0  # оттенок (будем менять)

while abs(t.xcor()) < 290 and abs(t.ycor()) < 290:
    # Преобразуем hue в RGB (упрощённо: радуга)
    # Используем HSV-логику: hue 0-255 даёт полный круг
    if hue < 85:
        r, g, b = 255 - int(hue * 3), int(hue * 3), 0
    elif hue < 170:
        r, g, b = 0, 255 - int((hue - 85) * 3), int((hue - 85) * 3)
    else:
        r, g, b = int((hue - 170) * 3), 0, 255 - int((hue - 170) * 3)

    t.pencolor(r, g, b)
    t.pensize(1 + step / 20)  # толщина растёт медленно
    t.forward(step)
    t.left(angle)
```

```
    step = step + 0.5
    hue = (hue + 1) % 256  # циклический сдвиг оттенка

screen.exitonclick()
```

4.2. Мини-исследование: Шаг радиуса и плотность витков

Вопрос: Как шаг увеличения радиуса (прирост `step`) влияет на «плотность» спирали?

Методика:

1. Напишите программу, которая рисует 3 спирали в разных местах экрана.
2. Для каждой спирали используйте свой **прирост шага**:
   - Спираль A: `step = step + 0.5` (медленный рост, плотные витки).
   - Спираль B: `step = step + 2` (средний рост).
   - Спираль C: `step = step + 5` (быстрый рост, редкие витки).
3. Замерьте для каждой:
   - Сколько полных витков она сделала до выхода за границу.
   - Визуальное впечатление («очень плотно», «нормально», «редко»).

Создайте `spiral_density.py`:

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

def draw_spiral(t, start_x, start_y, step_growth, color_name):
    """Рисует одну спираль и возвращает статистику."""
    t.penup()
    t.goto(start_x, start_y)
    t.pendown()
    t.pencolor(color_name)
    t.pensize(1)

    step = 3
    angle = 89
    count = 0  # счётчик шагов

    while abs(t.xcor() - start_x) < 150 and abs(t.ycor() - start_y) < 150:
        t.forward(step)
        t.left(angle)
        step = step + step_growth
        count = count + 1

    return count

screen = turtle.Screen()
screen.setup(width=700, height=400)
screen.title("Сравнение плотности спиралей")
t = turtle.Turtle()
t.speed(0)

print("Исследование плотности спиралей:")
```

```
print("-" * 40)

count_a = draw_spiral(t, -250, 0, 0.5, "blue")
print(f"Прирост 0.5: {count_a} шагов — очень плотно")

count_b = draw_spiral(t, 0, 0, 2.0, "green")
print(f"Прирост 2.0: {count_b} шагов — нормально")

count_c = draw_spiral(t, 250, 0, 5.0, "red")
print(f"Прирост 5.0: {count_c} шагов — редко")

print("-" * 40)
print("Вывод: чем меньше прирост, тем плотнее витки и больше шагов до
границы.")

screen.exitonclick()
```

Инженерная аналогия: Прирост шага в спирали — это как ускорение робота. Маленькое ускорение даёт плавное, контролируемое движение. Большое — быстрый разгон, но меньшую точность.

5. Выводы по занятию

1. **`while` — цикл для неизвестности.** В отличие от `for`, `while` выполняется, пока условие истинно. Это идеально для ситуаций, где мы не знаем число повторений заранее.
2. **`break` — аварийный выход.** Позволяет прервать цикл при наступлении критического события (граница экрана, препятствие, команда остановки). В робототехнике это основа безопасности.
3. **`continue` — пропуск хода.** Позволяет избирательно выполнять тело цикла. Пригодится для фильтрации данных.
4. **Спираль — модель накопления ошибки.** Каждый шаг чуть длиннее предыдущего — так же работает одометрия робота: маленькие погрешности накапливаются, и траектория «расползается».
5. **Параметры управляют поведением.** Изменяя `angle` и прирост `step`, мы получаем качественно разные спирали. Так же, меняя коэффициенты ПИД-регулятора, мы будем получать разное поведение робота.

Связь со следующим занятием: У нас появилась спираль, которая выходит за границу. А что, если мы хотим раскрашивать фигуры по хитрым правилам? «Если сторона чётная — красным, если нечётная — синим». Для этого нужны **условные операторы** **`if-elif-else`** — тема следующего занятия.

Домашнее задание:

- **Базовое**: Нарисуйте окружность через `while` со счётчиком до 180 (получится полуокружность — дуга).
- **Продвинутое**: Нарисуйте «улитку» — спираль с `angle = 60` (треугольная спираль).
- **Исследовательское**: Проведите замеры плотности спирали для 5 разных значений `step_growth` (от 0.2 до 8.0). Постройте таблицу зависимости числа шагов от прироста.

Занятие 6: Условные операторы if-elif-else, раскрашивание по условию

Содержание занятия: Принятие решений в программе. Условная логика в графике

Цели занятия

- **Образовательная**: Освоить условные операторы `if`, `elif`, `else` и операторы сравнения. Научиться менять поведение программы в зависимости от условий.
- **Развивающая**: Применить условную логику для создания адаптивных рисунков, где цвет и толщина линии зависят от номера стороны.
- **Воспитательная**: Показать, что ветвление — это основа «интеллектуального» поведения программы, фундамент для будущих алгоритмов управления роботом.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: if-elif-else и операторы сравнения (35 мин)**
   - Синтаксис `if`, `elif`, `else`.
   - Операторы сравнения: `==`, `!=`, `<`, `>`, `<=`, `>=`.
   - Раскрашивание сторон многоугольника: чётные — красный, нечётные — синий (`i % 2 == 0`).
3. **Уровень «Углублённый»: Логические операторы и адаптация (25 мин)**
   - Логические операторы `and`, `or`, `not`.
   - Изменение `pensize` по условию (толще на чётных).
   - Проверка выхода за границы экрана с предупреждением.
4. **Уровень «Исследовательский»: Гибкая раскраска (20 мин)**
   - Функция с параметром-списком цветов, раскраска сторон по кругу.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Программа, которая думает

До сих пор наши программы были «прямолинейными»: сделай то, потом это, повтори N раз. Но настоящий робот должен **принимать решения**: «Если расстояние до препятствия меньше 30 см — остановись. Иначе — продолжай движение».

Сегодня мы освоим **условные операторы** — инструмент, который позволяет программе выбирать, какой код выполнять, в зависимости от ситуации. И сделаем наши рисунки по-настоящему «умными».

2. Уровень «База»: if-elif-else и операторы сравнения

(Инструмент, навык, интерфейс)

Обязательный уровень. Осваиваем синтаксис ветвления и раскрашиваем многоугольник.

2.1. Синтаксис if, elif, else

Условный оператор проверяет истинность выражения и выполняет соответствующий блок кода.

Создайте файл `if_demo.py`:

```
nano ~/ws/rbt/scripts/if_demo.py
```

```
#!/usr/bin/env python3
# Демонстрация условных операторов

temperature = float(input("Введите температуру: "))

if temperature > 30:
    print("Жарко! Включить вентилятор.")
elif temperature > 20:
    print("Тепло. Оптимальный режим.")
elif temperature > 10:
    print("Прохладно. Надеть кофту.")
else:
    print("Холодно! Включить обогрев.")

print("Проверка завершена.")
```

Правила:

- `if` — обязательная первая проверка.
- `elif` (else if) — дополнительная проверка, если предыдущая ложна. Может быть сколько угодно.
- `else` — выполняется, если ни одно условие не истинно. Необязателен.
- Отступы (4 пробела) — **обязательны**. Они определяют, что относится к блоку условия.

2.2. Операторы сравнения

Условия строятся на операторах сравнения:

Оператор	Значение	Пример
`==`	равно	`x == 5`
`≠`	не равно	`x ≠ 0`
`<`	меньше	`speed < 0.5`
`>`	больше	`distance > 1.0`
`⩽`	меньше или равно	`angle ⩽ 90`
`⩾`	больше или равно	`battery ⩾ 20`

Распространённая ошибка: Не путайте `=` (присваивание) и `==` (сравнение).

```
x = 5       # присвоили значение
if x == 5:  # проверили, равно ли пяти
    print("x равен пяти")
```

2.3. Раскрашивание по чётности: оператор %

Оператор `%` возвращает **остаток от деления**. Это идеальный способ проверить чётность:

- `i % 2 == 0` — число чётное (остаток 0).
- `i % 2 == 1` — число нечётное (остаток 1).

Создайте `odd_even_polygon.py`:

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

screen = turtle.Screen()
screen.title("Чётные — красный, нечётные — синий")
t = turtle.Turtle()
t.speed(5)
t.pensize(3)

sides = 12
size = 80
angle = 360 / sides

for i in range(sides):
    # Проверяем чётность номера стороны
    if i % 2 == 0:
        t.pencolor("red")    # чётные стороны (0, 2, 4...)
    else:
        t.pencolor("blue")   # нечётные стороны (1, 3, 5...)

    t.forward(size)
    t.left(angle)

print("Двенадцатиугольник с чередованием цветов нарисован!")
screen.exitonclick()
```

Эксперимент: Поменяйте `sides` на 7. Если сторон нечётное количество, то первой и последней стороне достанется один цвет — они будут соседними.

3. Уровень «Углублённый»: Логические операторы и адаптация

(Что под капотом)

Усложняем условия и добавляем реакцию на окружение.

3.1. Логические операторы and, or, not

Для сложных условий комбинируем проверки:

- `and` — истинно, если **оба** условия истинны.
- `or` — истинно, если **хотя бы одно** истинно.
- `not` — переворачивает истинность.

Создайте `logic_demo.py`:

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

# Пример: допуск к управлению роботом
age = 16
has_license = True
is_instructor_nearby = False

if age >= 14 and has_license:
    print("Допуск разрешён (возраст и права в порядке)")

if age >= 14 or is_instructor_nearby:
    print("Допуск разрешён (либо возраст, либо инструктор рядом)")

if not is_instructor_nearby:
    print("Внимание: инструктора нет поблизости!")
```

3.2. Изменение pensize по условию

Сделаем стороны с разной толщиной: чётные — толстые, нечётные — тонкие.

Создайте `thick_thin_polygon.py`:

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

screen = turtle.Screen()
screen.title("Толстые и тонкие стороны")
t = turtle.Turtle()
t.speed(5)

sides = 8
size = 100
angle = 360 / sides

for i in range(sides):
    # Условие: номер стороны чётный ИЛИ это первая сторона
    if i % 2 == 0:
        t.pensize(5)   # толстая линия
        t.pencolor("darkgreen")
    else:
        t.pensize(1)   # тонкая линия
        t.pencolor("lightgreen")

    t.forward(size)
```

```
    t.left(angle)

screen.exitonclick()
```

3.3. Проверка выхода за границы экрана

Объединим `while`, `break` из прошлого занятия и `if` для контроля безопасной зоны.

Создайте `safe_walk.py`:

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

screen = turtle.Screen()
screen.setup(width=500, height=500)
screen.title("Безопасная прогулка")
t = turtle.Turtle()
t.speed(5)
t.pensize(2)

def is_safe(t, limit=200):
    """Проверяет, находится ли черепашка в безопасной зоне."""
    x, y = t.xcor(), t.ycor()
    # Опасно, если X ИЛИ Y вышли за предел
    if abs(x) > limit or abs(y) > limit:
        return False
    return True

step = 10
angle = 45

while is_safe(t):
    t.forward(step)
    t.left(angle)
    step += 10

    # Дополнительное предупреждение при приближении к границе
    x, y = t.xcor(), t.ycor()
    if abs(x) > 150 or abs(y) > 150:
        t.pencolor("orange")  # предупреждение
        print(f"   Приближаюсь к границе! x={x:.0f}, y={y:.0f}")
    else:
        t.pencolor("black")

print("Достигнута граница. Остановка.")
screen.exitonclick()
```

Что мы сделали:

- Вынесли проверку границы в отдельную функцию `is_safe()`.
- Использовали `or` для объединения условий по X и Y.
- Добавили зону предупреждения (оранжевый цвет) перед аварийной остановкой.

4. Уровень «Исследовательский»: Гибкая раскраска по списку

(Новое применение, прорыв за границы)

Для тех, кто хочет создавать переиспользуемые и настраиваемые решения.

4.1. Функция с параметром-списком цветов

Вместо жёсткой логики «чётный — красный» сделаем функцию, которая принимает список цветов и раскрашивает стороны по кругу.

Создайте `flex_color_polygon.py`:

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

def colored_polygon(t, sides, size, colors):
    """
    Рисует многоугольник, раскрашивая стороны цветами из списка по кругу.
    Если список короче числа сторон, цвета повторяются.
    """
    angle = 360 / sides
    for i in range(sides):
        # Выбираем цвет по кругу: i % len(colors)
        color_index = i % len(colors)
        t.pencolor(colors[color_index])
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
screen.title("Гибкая раскраска по списку")
t = turtle.Turtle()
t.speed(7)
t.pensize(3)

# Палитра 1: тёплые цвета
warm_colors = ["red", "orange", "gold"]
t.penup()
t.goto(-150, 0)
t.pendown()
colored_polygon(t, sides=9, size=70, colors=warm_colors)

# Палитра 2: холодные цвета
cool_colors = ["blue", "cyan", "navy"]
t.penup()
t.goto(150, 0)
t.pendown()
colored_polygon(t, sides=9, size=70, colors=cool_colors)

screen.exitonclick()
```

Ключевая идея: `i % len(colors)` — остаток от деления номера стороны на количество цветов. Это даёт индекс, который циклически бегает по списку: 0, 1, 2, 0, 1, 2... Такой подход — основа для работы с любыми повторяющимися паттернами.

5. Выводы по занятию

1. **Условные операторы — мозг программы.** `if-elif-else` позволяет коду выбирать действие в зависимости от ситуации. Это фундамент любого «умного» поведения.
2. **Операторы сравнения — язык условий.** `==`, `!=`, `<`, `>` и другие позволяют формализовать проверки: «равно ли?», «больше ли?».
3. **Логические операторы — комбинаторы.** `and`, `or`, `not` строят сложные условия из простых, как в реальной логике управления роботом («если скорость мала И препятствие далеко — ускориться»).
4. **Остаток от деления — мощный паттерн.** `i % 2` для чётности, `i % len(colors)` для циклического перебора. Этот же принцип мы будем использовать для циклических структур данных в сенсорах.
5. **Функции с параметрами — гибкость.** `colored_polygon(t, sides, size, colors)` можно вызвать с любой палитрой. Это шаг к созданию библиотек переиспользуемого кода.

Связь со следующим занятием: Мы использовали простые списки цветов. Но списки в Python умеют гораздо больше: хранить любые данные, изменяться в размере. А их «старший брат» — словари (`dict`) — позволяют связывать ключи и значения (например, имя фигуры → количество сторон). Этому посвящено следующее занятие: «Списки и словари».

Домашнее задание:

- **Базовое**: Нарисуйте десятиугольник, где стороны с 0-й по 4-ю — зелёные, а с 5-й по 9-ю — жёлтые (подсказка: `if i < 5`).
- **Продвинутое**: Нарисуйте «светофор»: три залитых круга (красный, жёлтый, зелёный), расположенных вертикально. Используйте `if` для выбора цвета в зависимости от номера круга.
- **Исследовательское**: Модифицируйте `colored_polygon` так, чтобы она принимала не список, а строку цветов через запятую (например, `"red,blue,green"`) и разбирала её внутри.

Занятие 7: Списки и словари

Содержание занятия: Структуры данных для хранения наборов и соответствий

Цели занятия

- **Образовательная**: Освоить два фундаментальных типа данных Python: списки (`list`) и словари (`dict`). Понять их синтаксис, способы создания и ключевые операции.
- **Развивающая**: Научиться выбирать подходящую структуру данных для задачи: список — для упорядоченного набора, словарь — для пар «ключ-значение».

- **Воспитательная**: Показать, что правильный выбор контейнера для данных — это инженерное решение, влияющее на читаемость и эффективность кода.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Списки и словари — первые шаги (35 мин)**
   - Списки: создание, индексация `[0]`, `[1]`, добавление `append()`.
   - Хранение набора цветов в списке.
   - Словари: пары «ключ-значение», получение значения по ключу.
3. **Уровень «Углублённый»: Циклы, random и таблицы (25 мин)**
   - Циклический перебор списка цветов для раскраски.
   - Словарь как таблица: фигура → количество сторон.
   - Случайный выбор цвета из списка (`random.choice`).
4. **Уровень «Исследовательский»: Конфигурации и скорость (20 мин)**
   - Чтение конфигурации из файла в список и словарь.
   - Мини-исследование: скорость поиска в списке и в словаре.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: От одиночных переменных к наборам данных

До сих пор мы хранили одно значение в одной переменной: `size = 100`, `color = "red"`. Но что если у нас 20 цветов? Или таблица соответствий «название фигуры → количество сторон»?

Для этого в Python есть **структуры данных** — контейнеры, которые держат много значений сразу. Два главных:

- **Список (`list`)** — упорядоченный набор элементов. Как полка с книгами: первая, вторая, третья.
- **Словарь (`dict`)** — набор пар «ключ → значение». Как телефонная книга: имя → номер телефона.

Сегодня мы освоим оба и поймём, когда какой использовать.

2. Уровень «База»: Списки и словари — первые шаги

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём списки и словари, учимся брать из них данные.

2.1. Списки: создание, индексация, добавление

Список создаётся квадратными скобками `[]`. Элементы нумеруются с нуля.

Создайте файл `list_demo.py`:

```
nano ~/ws/rbt/scripts/list_demo.py
```

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

# Создание списка
colors = ["red", "green", "blue", "yellow"]
numbers = [10, 20, 30, 40, 50]
mixed = [1, "hello", 3.14, True]  # список может содержать разные типы

# Доступ по индексу (начинается с 0!)
print(f"Первый цвет: {colors[0]}")   # red
print(f"Третий цвет: {colors[2]}")   # blue
print(f"Последний цвет: {colors[-1]}") # yellow (отрицательный индекс — с
конца)

# Длина списка
print(f"Количество цветов: {len(colors)}")  # 4

# Добавление элемента
colors.append("purple")
print(f"После добавления: {colors}")
print(f"Новый последний: {colors[-1]}")  # purple

# Изменение элемента
colors[0] = "darkred"
print(f"После изменения: {colors}")
```

Запустите:

```
python3 ~/ws/rbt/scripts/list_demo.py
```

2.2. Список цветов для рисования

Применим список к нашей графике. Создайте `list_colors_turtle.py`:

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

screen = turtle.Screen()
screen.title("Цвета из списка")
t = turtle.Turtle()
t.speed(5)
t.pensize(3)

# Список любимых цветов
my_colors = ["red", "orange", "gold", "green", "blue", "violet"]

# Рисуем линию из отрезков разных цветов
for i in range(len(my_colors)):
    t.pencolor(my_colors[i])
    t.forward(50)
    t.left(45)

screen.exitonclick()
```

2.3. Словари: пары «ключ-значение»

Словарь создаётся фигурными скобками `{}`. Каждый элемент — пара `ключ: значение`.

Создайте `dict_demo.py`:

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

# Создание словаря
robot = {
    "name": "MecanumBot",
    "speed": 0.5,
    "battery": 85,
    "is_moving": False
}

# Доступ по ключу
print(f"Имя робота: {robot['name']}")
print(f"Скорость: {robot['speed']} м/с")
print(f"Заряд: {robot['battery']}%")

# Добавление новой пары
robot["camera_resolution"] = "1920×1080"
print(f"Словарь после добавления: {robot}")

# Изменение значения
robot["battery"] = 80
print(f"Заряд обновлён: {robot['battery']}%")

# Проверка наличия ключа
if "speed" in robot:
    print("Ключ 'speed' есть в словаре")
```

Главное преимущество словаря: Мгновенный доступ к значению по ключу. Не нужно помнить индекс, как в списке.

3. Уровень «Углублённый»: Циклы, random и таблицы

(Что под капотом)

Комбинируем списки и словари с циклами и добавляем случайность.

3.1. Циклический перебор списка цветов

Список можно перебирать напрямую в `for`, без `range` и индексов:

Создайте `cycle_colors.py`:

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

screen = turtle.Screen()
screen.title("Перебор списка в цикле")
```

```
t = turtle.Turtle()
t.speed(7)
t.pensize(3)

colors = ["red", "orange", "gold", "green", "blue", "violet"]

t.penup()
t.goto(-200, 0)
t.pendown()

# Прямой перебор: color принимает значение каждого элемента по очереди
for color in colors:
    t.pencolor(color)
    t.forward(60)
    t.left(360 / len(colors))  # равномерный поворот

screen.exitonclick()
```

Это чище, чем `for i in range(len(colors))`, когда нам не нужен индекс.

3.2. Словарь фигур: программа рисует по таблице

Словарь отлично подходит для справочных таблиц. Создадим словарь «фигура → количество сторон» и одной программой нарисуем все.

Создайте `shape_table.py`:

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

def polygon(t, sides, size):
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
screen.title("Таблица фигур")
t = turtle.Turtle()
t.speed(7)

# Словарь: имя фигуры → количество сторон
shapes = {
    "треугольник": 3,
    "квадрат": 4,
    "пятиугольник": 5,
    "шестиугольник": 6,
    "восьмиугольник": 8
}

x_pos = -250
for name, sides in shapes.items():
    t.penup()
    t.goto(x_pos, 0)
    t.pendown()
```

```
    # Подписываем фигуру
    t.penup()
    t.goto(x_pos, -80)
    t.write(f"{name}\n({sides} сторон)", align="center", font=("Arial",
10, "normal"))
    t.goto(x_pos, 0)
    t.pendown()

    polygon(t, sides, 40)
    x_pos += 100

screen.exitonclick()
```

Что нового: `shapes.items()` возвращает пары (ключ, значение), которые мы распаковываем в `name, sides`.

3.3. Случайный выбор из списка: random.choice

Модуль `random` содержит функцию `choice()`, которая возвращает случайный элемент списка.

Создайте `random_artist.py`:

```
#!/usr/bin/env python3
import turtle
import random

screen = turtle.Screen()
screen.title("Случайный художник")
t = turtle.Turtle()
t.speed(10)
t.pensize(2)

colors = ["red", "blue", "green", "orange", "purple", "pink", "cyan", "bro
wn"]
angles = [60, 90, 120, 144]  # возможные углы поворота

for _ in range(50):
    t.pencolor(random.choice(colors))  # случайный цвет
    t.forward(30)
    t.left(random.choice(angles))      # случайный угол

print("Случайный узор готов!")
screen.exitonclick()
```

4. Уровень «Исследовательский»: Файлы конфигурации и скорость поиска

(Новое применение, прорыв за границы)

Для исследователей, которые проектируют системы с внешними настройками и думают о производительности.

4.1. Чтение конфигурации из файла

Настоящие робототехнические программы читают настройки из внешних файлов. Сделаем простой конфигурационный файл и программу, которая его читает.

Сначала создайте файл конфигурации `polygon_config.txt`:

```
nano ~/ws/rbt/config/polygon_config.txt
```

Содержимое (простой формат: ключ=значение):

```
# Конфигурация для рисования фигур
colors=red,green,blue,gold,violet
shapes=triangle:3,square:4,pentagon:5,hexagon:6
size=80
speed=7
```

Теперь программа, которая читает этот файл, — создайте `config_reader.py`:

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

def load_config(filename):
    """Читает простой конфигурационный файл."""
    config = {}
    with open(filename, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            # Пропускаем пустые строки и комментарии
            if not line or line.startswith("#"):
                continue
            # Разбираем строку ключ=значение
            key, value = line.split("=", 1)
            config[key.strip()] = value.strip()
    return config

def polygon(t, sides, size):
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

# Загрузка конфигурации - указать реальный путь до файла
config = load_config("/home/pi/ws/rbt/config/polygon_config.txt")

# Разбор строки цветов в список
colors_list = config["colors"].split(",")
print(f"Цвета: {colors_list}")

# Разбор строки фигур в словарь
shapes_dict = {}
for pair in config["shapes"].split(","):
    name, sides = pair.split(":")
    shapes_dict[name] = int(sides)
print(f"Фигуры: {shapes_dict}")
```

```
size = int(config["size"])

# Рисуем
screen = turtle.Screen()
screen.title("Рисование из конфига")
t = turtle.Turtle()
t.speed(int(config["speed"]))

x_pos = -250
for (name, sides), color in zip(shapes_dict.items(), colors_list):
    t.penup()
    t.goto(x_pos, 0)
    t.pendown()
    t.pencolor(color)
    polygon(t, sides, size)
    x_pos += 100

screen.exitonclick()
```

Что мы добились: Теперь, чтобы изменить цвета или набор фигур, не нужно лезть в код — достаточно отредактировать текстовый файл. Это принцип разделения кода и данных.

4.2. Мини-исследование: поиск в списке и в словаре

Вопрос: Что быстрее — найти элемент в списке из 10000 значений или найти значение по ключу в словаре из 10000 пар?

Создайте `speed_test.py`:

```
#!/usr/bin/env python3
import time
import random

# Подготовка данных
N = 10000
print(f"Создаю список и словарь из {N} элементов...")

# Список: просто числа
test_list = list(range(N))

# Словарь: ключи — числа, значения — строки
test_dict = {i: f"value_{i}" for i in range(N)}

# Выбираем случайный элемент для поиска
target = random.randint(0, N - 1)
print(f"Ищем элемент: {target}")

# Тест поиска в списке (проверка наличия)
start = time.time()
found_in_list = target in test_list
list_time = time.time() - start
print(f"Поиск в списке: {list_time:.6f} секунд. Найден: {found_in_list}")

# Тест поиска в словаре (проверка наличия ключа)
start = time.time()
```

```
found_in_dict = target in test_dict
dict_time = time.time() - start
print(f"Поиск в словаре: {dict_time:.6f} секунд. Найден: {found_in_dict}")

# Сравнение
if dict_time > 0:
    ratio = list_time / dict_time
    print(f"\nСловарь быстрее в {ratio:.0f} раз(а)!")
else:
    print("\nСловарь сработал мгновенно!")

print("\nОбъяснение:")
print("- Список проверяет элементы по порядку (O(n)).")
print("- Словарь использует хеш-таблицу (O(1)) — почти мгновенно.")
```

Запустите несколько раз с разными `target`. Словарь всегда будет значительно быстрее при большом объёме данных.

Вывод для робототехника: Когда нужно быстро находить параметры по имени (например, `params["max_speed"]`), используйте словарь. Когда важна упорядоченность (например, список путевых точек маршрута) — список.

5. Выводы по занятию

1. **Список (`list`) — упорядоченный набор.** Индексация с 0, добавление через `append()`, перебор в цикле. Идеален для хранения последовательностей (маршрутных точек, показаний датчиков, цветов).
2. **Словарь (`dict`) — таблица соответствий.** Мгновенный доступ по ключу. Идеален для хранения параметров, настроек, справочников.
3. **Выбор структуры — инженерное решение.** Список — для порядка, словарь — для скорости поиска. Сегодняшний эксперимент доказал это численно.
4. **Внешние конфигурации — шаг к профессионализму.** Вынос настроек в файл отделяет логику от данных и упрощает перенастройку системы.
5. **Модуль `random` — генератор неопределённости.** `random.choice()` добавляет вариативность, что пригодится для тестирования робота в симуляции.

Связь со следующим занятием: Мы активно использовали `def polygon( ... )`, но каждый раз писали её заново. На следующем занятии мы систематизируем знания о **функциях**: аргументы по умолчанию, возврат значений, и создадим библиотеку геометрических фигур, которую будем переиспользовать.

Домашнее задание:

- **Базовое**: Создайте список из 5 имён и выведите их в столбик через цикл `for`.
- **Продвинутое**: Создайте словарь «месяц → количество дней». Напишите программу, которая запрашивает название месяца и выводит количество дней.
- **Исследовательское**: Проведите замеры скорости поиска для списка и словаря размером 100, 1000, 10000, 100000 элементов. Постройте график зависимости времени от размера (можно вручную на бумаге или в электронных таблицах).

Занятие 8: Функции: def, аргументы, возврат значения

Содержание занятия: Создание собственных переиспользуемых блоков кода

Цели занятия

- **Образовательная**: Освоить синтаксис определения функции `def`, передачи аргументов и возврата результата через `return`.
- **Развивающая**: Научиться выделять повторяющуюся логику в функции, делать код модульным и переиспользуемым.
- **Воспитательная**: Привить инженерную культуру: не копировать код, а создавать функции.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**

2. **Уровень «База»: Определение и вызов функций (35 мин)**
    - Синтаксис `def имя(аргументы):`.
    - Функция `polygon(sides, size)` — рисует любой правильный многоугольник.
    - Вызов функции в цикле для рисования набора фигур.
3. **Уровень «Углублённый»: Аргументы по умолчанию и return (25 мин)**
    - Аргументы со значением по умолчанию (`size=100`).
    - Возврат значения `return`.
    - Функция `star(points, size)` — рисует звезду.
4. **Уровень «Исследовательский»: Композиция и анализ (20 мин)**
    - Функция `flower(petals)` — рисует цветок из ромбов.
    - Мини-исследование: что быстрее — передача параметра или глобальная переменная.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Кирпичики программы

Мы уже использовали функции: `t.forward(100)`, `print("Привет")`, `range(5)`. Но до сих пор мы не создавали свои.

Функция — это именованный блок кода, который можно вызвать много раз с разными параметрами. Зачем?

- **Избежать копирования:** написал один раз — используешь везде.
- **Структурировать код:** большая задача разбивается на маленькие понятные функции.
- **Легко менять:** исправил функцию — исправилось всё, что её использует.

Сегодня мы превратим наши рисунки в библиотеку переиспользуемых функций.

2. Уровень «База»: Определение и вызов функций

(Инструмент, навык, интерфейс)

Обязательный уровень. Учимся создавать и вызывать функции.

2.1. Синтаксис определения функции

Функция объявляется ключевым словом `def`, затем имя, скобки с параметрами и двоеточие. Тело функции — с отступом.

Создайте файл `function_demo.py`:

```
nano ~/ws/rbt/scripts/function_demo.py
```

```
#!/usr/bin/env python3
# Демонстрация создания и вызова функций

def greet(name):
```

```
    """Приветствует пользователя по имени."""
    print(f"Привет, {name}!")

def add(a, b):
    """Складывает два числа."""
    result = a + b
    print(f"{a} + {b} = {result}")

# Вызов функций
greet("Алиса")
greet("Боб")

add(3, 5)
add(10, -2)

# Функции можно вызывать в цикле
for name in ["Коля", "Оля", "Петя"]:
    greet(name)
```

Правила:

- Имя функции — глагол или существительное (что делает).
- Строка в тройных кавычках сразу после `def` — **документация** (docstring). Она помогает понять, что делает функция.
- Функцию нужно **определить до вызова** (код читается сверху вниз).

2.2. Функция polygon(sides, size)

Формализуем то, что мы уже много раз писали. Создайте `polygon_func.py`:

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

def polygon(t, sides, size):
    """Рисует правильный многоугольник.
    t     — черепашка
    sides — количество сторон
    size  — длина стороны
    """
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

# ── Основная программа ──
screen = turtle.Screen()
screen.title("Функция polygon")
t = turtle.Turtle()
t.speed(7)

# Рисуем разные фигуры одной функцией
t.pencolor("red")
polygon(t, 3, 100)   # треугольник
```

```
t.penup()
t.goto(150, 0)
t.pendown()
t.pencolor("blue")
polygon(t, 6, 60)    # шестиугольник

t.penup()
t.goto(-150, -150)
t.pendown()
t.pencolor("green")
polygon(t, 8, 50)    # восьмиугольник

screen.exitonclick()
```

2.3. Вызов функции в цикле

Создайте `polygon_gallery.py`:

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

def polygon(t, sides, size):
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
screen.title("Галерея многоугольников")
t = turtle.Turtle()
t.speed(10)

# Данные: список кортежей (стороны, размер, цвет)
figures = [
    (3, 80, "red"),
    (4, 80, "blue"),
    (5, 80, "green"),
    (6, 80, "orange"),
    (8, 80, "purple"),
]

x_start = -250
for sides, size, color in figures:
    t.penup()
    t.goto(x_start, 0)
    t.pendown()
    t.pencolor(color)
    polygon(t, sides, size)
    x_start += 110

screen.exitonclick()
```

3. Уровень «Углублённый»: Аргументы по умолчанию и return

(Что под капотом)

Делаем функции более гибкими и учим их возвращать результат.

3.1. Аргументы со значением по умолчанию

Иногда хочется, чтобы у параметра было «стандартное» значение, которое можно не указывать.

Создайте `default_args.py`:

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

def polygon(t, sides, size=100):
    """Рисует многоугольник. По умолчанию size=100."""
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
t = turtle.Turtle()
t.speed(7)

# Вызов с двумя аргументами (size по умолчанию = 100)
t.pencolor("red")
polygon(t, 4)          # квадрат со стороной 100

# Вызов с тремя аргументами (переопределяем size)
t.penup()
t.goto(150, 0)
t.pendown()
t.pencolor("blue")
polygon(t, 4, 50)      # квадрат со стороной 50

screen.exitonclick()
```

Правило: Параметры со значениями по умолчанию идут **после** обязательных.

3.2. Возврат значения return

Функция может не только делать, но и **вычислять** результат. Ключевое слово `return` возвращает значение в точку вызова.

Создайте `return_demo.py`:

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

def polygon_perimeter(sides, size):
    """Вычисляет периметр правильного многоугольника."""
    return sides * size
```

```
def circle_length(radius):
    """Вычисляет длину окружности."""
    return 2 * 3.14159 * radius

# Использование возвращаемых значений
p = polygon_perimeter(6, 50)
print(f"Периметр шестиугольника: {p}")

l = circle_length(10)
print(f"Длина окружности радиусом 10: {l:.2f}")

# Возвращаемое значение можно сразу использовать
total = polygon_perimeter(3, 40) + polygon_perimeter(4, 40)
print(f"Сумма периметров: {total}")
```

Важно: `return` немедленно завершает функцию. Всё, что после него в том же блоке, не выполнится.

3.3. Функция star(points, size)

Звезда — это многоугольник с самопересечением. Для N-конечной звезды угол поворота = 180 - 180 / N.

Создайте `star_func.py`:

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

def star(t, points, size):
    """Рисует звезду с заданным числом лучей."""
    angle = 180 - 180 / points
    for _ in range(points):
        t.forward(size)
        t.left(angle)

screen = turtle.Screen()
screen.title("Звёзды")
t = turtle.Turtle()
t.speed(7)
t.pensize(2)

# Рисуем звёзды с разным числом лучей
positions = [(-200, 0), (0, 0), (200, 0)]
points_list = [5, 7, 9]
colors = ["red", "gold", "blue"]

for (x, y), pts, col in zip(positions, points_list, colors):
    t.penup()
    t.goto(x, y)
    t.pendown()
    t.pencolor(col)
    star(t, pts, 80)

screen.exitonclick()
```

4. Уровень «Исследовательский»: Композиция и анализ

(Новое применение, прорыв за границы)

Строим сложные фигуры из простых и исследуем производительность.

4.1. Функция flower(petals) — композиция фигур

Цветок можно составить из ромбов, расположенных по кругу. Ромб — это два вызова `polygon` с поворотом, или просто цикл с углами.

Создайте `flower.py`:

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

def rhombus(t, size, angle):
    """Рисует ромб. angle — острый угол ромба."""
    for _ in range(2):
        t.forward(size)
        t.left(angle)
        t.forward(size)
        t.left(180 - angle)

def flower(t, petals, size):
    """Рисует цветок из ромбов."""
    angle = 60  # угол ромба
    turn = 360 / petals
    for _ in range(petals):
        t.pencolor("red")
        rhombus(t, size, angle)
        t.pencolor("darkred")
        rhombus(t, size * 0.5, angle)  # внутренний ромб поменьше
        t.left(turn)

screen = turtle.Screen()
screen.title("Цветок из ромбов")
screen.tracer(0)
t = turtle.Turtle()
t.speed(0)
t.hideturtle()
t.pensize(2)

# Рисуем цветы с разным числом лепестков
t.penup()
t.goto(-200, 0)
t.pendown()
flower(t, 6, 50)

t.penup()
t.goto(200, 0)
t.pendown()
flower(t, 8, 40)
```

```
screen.update()
screen.exitonclick()
```

Принцип композиции: `flower` вызывает `rhombus`. Сложное поведение строится из простых функций, как из кирпичиков.

4.2. Мини-исследование: параметр или глобальная переменная?

Вопрос: Что работает быстрее — передать значение в функцию как параметр или использовать глобальную переменную?

Создайте `param_vs_global.py`:

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

# Глобальная переменная
GLOBAL_SIZE = 100

def use_global():
    """Использует глобальную переменную."""
    total = 0
    for _ in range(1000000):
        total += GLOBAL_SIZE
    return total

def use_param(size):
    """Использует переданный параметр."""
    total = 0
    for _ in range(1000000):
        total += size
    return total

# Замер времени
start = time.time()
result1 = use_global()
time_global = time.time() - start
print(f"Глобальная переменная: {time_global:.4f} сек")

start = time.time()
result2 = use_param(100)
time_param = time.time() - start
print(f"Параметр функции:     {time_param:.4f} сек")

if time_param > 0:
    print(f"\nРазница: {(time_global / time_param - 1) * 100:.1f}%")

print("\nВывод: разница микроскопична.")
print("Лучше использовать параметры — код понятнее и безопаснее.")
print("Глобальные переменные могут быть случайно изменены,")
print("что приводит к трудным для поиска ошибкам.")
```

Инженерный вывод: Не гонитесь за микроскопической разницей в скорости. Пишите **понятный и безопасный** код. Явная передача параметров всегда лучше, чем неявная зависимость от глобальных переменных.

5. Выводы по занятию

1. **Функция — именованный блок кода.** `def имя(параметры):` создаёт переиспользуемый кирпичик. Вызов по имени с конкретными аргументами.
2. **Параметры — входные данные.** Функция `polygon(t, sides, size)` работает с любыми значениями. Это делает код универсальным.
3. **return — выходные данные.** Функция может вычислять и возвращать результат для дальнейшего использования.
4. **Умолчания — удобство.** `size=100` позволяет не указывать параметр, если стандартное значение подходит.
5. **Композиция — сила.** Функция `flower` вызывает `rhombus`. Сложные системы строятся из простых, хорошо протестированных компонентов. Так же мы будем собирать поведение робота из модулей навигации, зрения и управления.

Связь со следующим занятием: Мы создали функции `polygon` и `star`. Но они лежат в одном файле с основной программой. Как вынести их в отдельный файл и использовать в разных проектах? Для этого нужны **модули и импорт** — тема следующего занятия.

Домашнее задание:

- **Базовое**: Напишите функцию `square(t, size)`, которая рисует квадрат (вызовом `polygon`).
- **Продвинутое**: Напишите функцию `house(t, size)`, которая рисует домик (квадрат + треугольник-крыша) вызовом ранее созданных функций.
- **Исследовательское**: Напишите функцию `checkerboard(t, rows, cols, cell_size)`, которая рисует шахматную доску, используя `square`.

Занятие 9: Модули и импорт

Содержание занятия: Организация кода в отдельные файлы. Создание собственных библиотек

Цели занятия

- **Образовательная**: Освоить механизм импорта модулей в Python. Научиться выносить функции в отдельные файлы и использовать их в других программах.
- **Развивающая**: Понять принцип разделения кода на библиотеки (инструменты) и основную программу (их применение).
- **Воспитательная**: Привить культуру организации проектов: код должен быть не только работающим, но и аккуратно структурированным.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Создание и импорт модуля (35 мин)**
   • Что такое модуль и зачем он нужен.
   • Создание своего модуля `shapes.py` с функциями `polygon` и `star`.
   • Импорт модуля через `import shapes` и вызов функций.
3. **Уровень «Углублённый»: Варианты импорта и структура (25 мин)**
   • `from shapes import polygon, star` — импорт конкретных функций.
   • Организация кода: отдельный файл для функций и отдельный для основной программы.
4. **Уровень «Исследовательский»: Библиотека узоров и путь поиска (20 мин)**
   • Создание модуля `patterns.py` с готовыми композициями.
   • Мини-исследование: как Python находит модули — `sys.path`.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: От одного файла — к проекту

До сих пор мы писали всю программу в одном файле. Функции `polygon`, `star` и основная логика были перемешаны. Это работает для маленьких скриптов, но не для серьёзных проектов.

Модуль — это просто файл с расширением `.py`, который содержит функции, переменные и классы. Его можно **импортировать** в другой файл и использовать.

Зачем?

- **Переиспользование:** написал `polygon` один раз — используешь в десяти проектах.
- **Порядок:** библиотека фигур отдельно, программа их рисования отдельно.
- **Командная работа:** каждый может разрабатывать свой модуль независимо.

Сегодня мы превратим наши наработки в настоящую библиотеку.

2. Уровень «База»: Создание и импорт модуля

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём свой первый модуль и подключаем его.

2.1. Что такое модуль

Модуль — это файл `.py`. Его имя — это имя файла без расширения.

Мы уже использовали чужие модули:

```
import turtle       # модуль turtle (файл turtle.py где-то в системе)
import random       # модуль random
import time         # модуль time
```

Теперь создадим свой.

2.2. Создание модуля shapes.py

Создадим файл `shapes.py` в папке с исходным кодом. В него мы вынесем наши лучшие функции.

Создайте файл:

```
nano ~/ws/rbt/scripts/shapes.py
```

Содержимое модуля `shapes.py`:

```
# shapes.py — модуль с функциями для рисования геометрических фигур

def polygon(t, sides, size):
    """Рисует правильный многоугольник."""
    angle = 360 / sides
    for _ in range(sides):
        t.forward(size)
        t.left(angle)

def star(t, points, size):
    """Рисует звезду с заданным числом лучей."""
    angle = 180 - 180 / points
    for _ in range(points):
        t.forward(size)
        t.left(angle)
```

Обратите внимание: здесь **нет** кода, который что-то рисует. Только определения функций. Это и есть библиотека.

2.3. Импорт модуля и вызов функций

Теперь в **другом файле** (в той же папке) будем использовать наш модуль.

Создайте `use_shapes.py`:

```
nano ~/ws/rbt/scripts/use_shapes.py
```

```
#!/usr/bin/env python3
# use_shapes.py — основная программа, использующая модуль shapes

import turtle
import shapes  # импортируем наш модуль!

screen = turtle.Screen()
screen.title("Использование модуля shapes")
t = turtle.Turtle()
t.speed(7)

# Вызов функций из модуля: имя_модуля.имя_функции
```

```
t.pencolor("red")
shapes.polygon(t, 6, 70)  # шестиугольник

t.penup()
t.goto(150, 0)
t.pendown()
t.pencolor("gold")
shapes.star(t, 5, 80)     # пятиконечная звезда

t.penup()
t.goto(-150, -150)
t.pendown()
t.pencolor("blue")
shapes.polygon(t, 8, 50)  # восьмиугольник

screen.exitonclick()
```

Синтаксис: `shapes.polygon( ... )` — сначала имя модуля, потом точка, потом имя функции. Это как `turtle.forward( ... )`.

3. Уровень «Углублённый»: Варианты импорта и организация

(Что под капотом)

Осваиваем гибкие способы импорта и правильную структуру проекта.

3.1. from ... import — импорт конкретных функций

Если вы часто используете функцию, писать `shapes.polygon` утомительно. Можно импортировать функцию напрямую:

Создайте `use_shapes_short.py`:

```
#!/usr/bin/env python3
# Демонстрация from ... import

import turtle
from shapes import polygon, star  # импортируем только нужное

screen = turtle.Screen()
t = turtle.Turtle()
t.speed(7)

# Теперь можно вызывать без префикса shapes.
t.pencolor("green")
polygon(t, 5, 80)  # просто polygon!

t.penup()
t.goto(150, 0)
t.pendown()
t.pencolor("orange")
star(t, 7, 60)     # просто star!
```

```
screen.exitonclick()
```

Ещё варианты:

- `from shapes import *` — импортировать **всё** из модуля (не рекомендуется: можно случайно переопределить свои функции).
- `import shapes as sh` — дать модулю короткий псевдоним (`sh.polygon( ... )`).

3.2. Организация кода: библиотека и программа

Теперь у нас чёткое разделение:

```
~/ws/rbt/scripts/
├── shapes.py          # БИБЛИОТЕКА: только функции, без вызовов
├── use_shapes.py      # ПРОГРАММА: импорт и вызов функций
└── use_shapes_short.py # ПРОГРАММА: другой вариант импорта
```

Это стандартная практика в программировании:

- **Библиотека** не делает ничего сама по себе — она предоставляет инструменты.
- **Основная программа** использует инструменты для решения конкретной задачи.

Именно так устроен и ROS2: есть библиотеки (`rclpy`), а есть ваши узлы, которые их используют.

4. Уровень «Исследовательский»: Библиотека узоров и путь поиска

(Новое применение, прорыв за границы)

Создаём модуль с готовыми композициями и изучаем, как Python находит модули.

4.1. Модуль patterns.py с готовыми композициями

Создайте `patterns.py` — модуль, который использует `shapes` для создания готовых узоров:

```
nano ~/ws/rbt/scripts/patterns.py
```

```
# patterns.py — модуль с готовыми композициями

from shapes import polygon, star

def star_grid(t, rows, cols, star_size, spacing):
    """Рисует сетку из звёзд."""
    for row in range(rows):
        for col in range(cols):
            x = col * spacing - (cols * spacing) / 2
            y = row * spacing - (rows * spacing) / 2
            t.penup()
            t.goto(x, y)
            t.pendown()
            star(t, 5, star_size)

def square_grid(t, rows, cols, cell_size):
```

```
    """Рисует сетку из квадратов."""
    for row in range(rows):
        for col in range(cols):
            x = col * cell_size - (cols * cell_size) / 2
            y = row * cell_size - (rows * cell_size) / 2
            t.penup()
            t.goto(x, y)
            t.pendown()
            # Шахматная раскраска
            if (row + col) % 2 == 0:
                t.fillcolor("lightblue")
            else:
                t.fillcolor("lightcoral")
            t.begin_fill()
            polygon(t, 4, cell_size)
            t.end_fill()

def nested_polygons(t, sides_list, center_x, center_y):
    """Рисует вложенные многоугольники с общим центром."""
    t.penup()
    t.goto(center_x, center_y)
    t.pendown()
    for sides in sides_list:
        t.pencolor("blue")
        polygon(t, sides, 20 + sides * 10)
```

А теперь программа, которая использует этот модуль, — `use_patterns.py`:

```
nano ~/ws/rbt/scripts/use_patterns.py
```

```
#!/usr/bin/env python3
import turtle
from patterns import star_grid, square_grid

screen = turtle.Screen()
screen.title("Узоры из модуля patterns")
screen.tracer(0)
t = turtle.Turtle()
t.speed(0)
t.hideturtle()

# Рисуем узор из звёзд
t.pencolor("gold")
t.pensize(1)
star_grid(t, rows=3, cols=3, star_size=30, spacing=100)

# Перемещаемся и рисуем шахматную доску
t.penup()
t.goto(200, 0)
square_grid(t, rows=4, cols=4, cell_size=40)

screen.update()
screen.exitonclick()
```

Цепочка зависимостей: `use_patterns.py` → `patterns.py` → `shapes.py`. Каждый уровень добавляет свою абстракцию.

4.2. Мини-исследование: как Python находит модули

Вопрос: Почему `import shapes` работает, только если файл лежит в той же папке? Где Python вообще ищет модули?

Создайте `explore_path.py`:

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

print("Python ищет модули в следующих папках (sys.path):")
print("-" * 60)
for i, path in enumerate(sys.path):
    print(f"{i}: {path}")

print("\nПервая папка — это папка, из которой запущен скрипт.")
print("Поэтому модули из текущей папки находятся автоматически.")

# Попробуем временно добавить другую папку в путь поиска
import os
custom_path = os.path.expanduser("~/ws/rbt/src")
if custom_path not in sys.path:
    sys.path.append(custom_path)
    print(f"\nДобавлена папка: {custom_path}")
    print("Теперь Python будет искать модули и там!")
```

Задание:

1. Запустите скрипт и посмотрите на список папок.
2. Переместите `shapes.py` в папку `~/ws/rbt/src`.
3. Попробуйте `import shapes` из `~/ws/rbt/scripts` — получите ошибку `ModuleNotFoundError`.
4. Добавьте `sys.path.append( ... )` в начало скрипта — импорт заработает.

Вывод: Python ищет модули в `sys.path`. Первый элемент — текущая папка. Для ROS2-проектов мы будем настраивать пути через `setup.py` и `colcon build`.

5. Выводы по занятию

1. **Модуль — это файл .py с функциями и классами.** Создав `shapes.py`, мы получили библиотеку, которую можно импортировать в любой проект.
2. **import подключает модуль.** Есть несколько форм: `import module`, `from module import func`, `import module as alias`.
3. **Разделение кода — признак профессионализма.** Библиотека (`shapes.py`) не делает ничего сама, она предоставляет инструменты. Программа (`use_shapes.py`) их применяет.
4. **Многослойная архитектура.** `use_patterns.py` → `patterns.py` → `shapes.py`. Каждый слой добавляет уровень абстракции. Так же устроены ROS2-проекты: узлы → библиотеки → драйверы.

5. **sys.path управляет поиском модулей.** Понимание этого механизма поможет, когда мы перейдём к сборке ROS2-пакетов через colcon.

Связь со следующим занятием: У нас есть функции, которые работают с черепашкой `t`, переданной как параметр. Но черепашка — это объект с собственными атрибутами (цвет, скорость) и методами (`forward`, `left`). А что если мы создадим свой класс объектов, например, `Artist`? Это переход к **объектно-ориентированному программированию** — теме следующего занятия.

Домашнее задание:

- **Базовое**: Вынесите функцию `flower` из прошлого занятия в модуль `shapes.py`. Импортируйте её и нарисуйте цветок.
- **Продвинутое**: Создайте модуль `robot_math.py` с функциями `distance(x1, y1, x2, y2)` и `to_radians(degrees)`. Используйте их в программе.
- **Исследовательское**: Изучите содержимое папок из `sys.path`. Найдите, где лежит модуль `turtle.py` на вашем роботе. Посмотрите его размер.

Занятие 10: Основы ООП: класс Artist

Содержание занятия: Объектно-ориентированное программирование. Классы и объекты в Python

Цели занятия

- **Образовательная**: Освоить базовые понятия ООП: класс, объект, конструктор `__init__`, атрибуты и методы. Понять роль параметра `self`.
- **Развивающая**: Перейти от процедурного подхода (отдельные функции) к объектному (связка данных и поведения в одном объекте).
- **Воспитательная**: Показать, что ООП — это естественный способ моделирования реальных сущностей (робот, датчик, художник) в коде.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Класс Artist — первый объект (35 мин)**
   - Понятия «класс» и «объект» на аналогии с чертежом и изделием.
   - Синтаксис `class`, метод `__init__`, `self`.
   - Класс `Artist` с атрибутами (цвет, скорость) и методами `polygon()`, `move_to()`.
   - Создание одного объекта и рисование фигуры.
3. **Уровень «Углублённый»: Несколько объектов и изменение состояния (25 мин)**
   - Создание нескольких объектов `Artist` с разными цветами.
   - Одновременное рисование в разных углах экрана.
   - Метод `set_color()` для смены цвета после создания.

4. **Уровень «Исследовательский»: Фабрика объектов и пределы памяти (20 мин)**
    - Функция `create_artists(n)` — создание списка художников.
    - Мини-исследование: сколько объектов можно создать до заполнения памяти.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: От функций — к объектам

До сих пор мы передавали черепашку `t` в каждую функцию: `polygon(t, 6, 80)`. Это работает, но данные (цвет, скорость) и поведение (нарисовать, переместиться) существуют раздельно.

Объектно-ориентированное программирование (ООП) предлагает другой подход: объединить данные и поведение в одной сущности — **объекте**.

Аналогия:

- **Класс** — это чертёж. Например, «художник»: у него есть цвет, скорость, и он умеет рисовать много-угольники.
- **Объект** — это конкретный художник, созданный по чертежу. Художник с красным цветом, художник с синим цветом.

Сегодня мы создадим класс `Artist`, который будет самостоятельным «роботом-рисовальщиком».

2. Уровень «База»: Класс Artist — первый объект

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём класс, конструктор, атрибуты и методы.

2.1. Понятие класса и объекта

Проведём аналогию с реальным миром:

Чертёж (Класс)	Изделие (Объект)
`class Robot:`	`my_robot = Robot()`
`class Sensor:`	`lidar = Sensor()`
`class Artist:`	`painter = Artist()`

- **Класс** описывает, какие **атрибуты** (данные) и **методы** (функции) будут у объекта.
- **Объект** — конкретный экземпляр класса со своими значениями атрибутов.

2.2. Синтаксис класса, конструктор **init** и self

Создайте файл `artist_demo.py`:

```
nano ~/ws/rbt/scripts/artist_demo.py
```

```python
#!/usr/bin/env python3
# Демонстрация класса Artist

import turtle

class Artist:
    """Класс художника, который умеет рисовать фигуры."""

    def __init__(self, color, speed):
        """Конструктор — вызывается при создании объекта.
        self — ссылка на сам объект."""
        self.turtle = turtle.Turtle()  # у каждого художника своя
черепашка
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)
        print(f"Создан художник с цветом {color}")

    def polygon(self, sides, size):
        """Рисует правильный многоугольник."""
        angle = 360 / sides
        for _ in range(sides):
            self.turtle.forward(size)
            self.turtle.left(angle)

    def move_to(self, x, y):
        """Перемещает художника в точку, не рисуя."""
        self.turtle.penup()
        self.turtle.goto(x, y)
        self.turtle.pendown()

# ── Основная программа ──
screen = turtle.Screen()
screen.title("Мой первый класс Artist")

# Создаём объект (экземпляр класса)
artist1 = Artist("red", 5)  # вызывается __init__

# Используем методы объекта
artist1.move_to(-100, 0)
artist1.polygon(6, 70)

artist1.move_to(100, 0)
artist1.polygon(4, 80)

screen.exitonclick()
```

Ключевые моменты:

- `class Artist:` — объявление класса.
- `__init__` — специальный метод (конструктор), вызывается автоматически при создании объекта.
- `self` — первый параметр каждого метода, ссылается на конкретный объект. Через `self.` мы обращаемся к атрибутам и методам этого объекта.

2.3. Объект в действии

Обратите внимание на чистоту кода:

- Нам не нужно передавать `t` в каждый вызов — художник сам знает свою черепашку.
- Логика «переместиться» и «нарисовать» инкапсулирована внутри класса.

Этот подход станет основой для работы с ROS2: каждый узел — это объект, который подписывается на топики и публикует сообщения.

3. Уровень «Углублённый»: Несколько объектов и изменение состояния

(Что под капотом)

Создаём ансамбль художников и учим их менять цвет на лету.

3.1. Несколько объектов с разными цветами

Создайте `artist_ensemble.py`:

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

class Artist:
    def __init__(self, color, speed):
        self.turtle = turtle.Turtle()
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)

    def polygon(self, sides, size):
        angle = 360 / sides
        for _ in range(sides):
            self.turtle.forward(size)
            self.turtle.left(angle)

    def move_to(self, x, y):
        self.turtle.penup()
        self.turtle.goto(x, y)
        self.turtle.pendown()

screen = turtle.Screen()
screen.title("Ансамбль художников")
screen.tracer(0)  # ускоряем отрисовку

# Создаём трёх разных художников
red_artist = Artist("red", 7)
blue_artist = Artist("blue", 7)
green_artist = Artist("green", 7)

# Каждый рисует в своём углу экрана
red_artist.move_to(-200, 100)
```

```
red_artist.polygon(6, 60)

blue_artist.move_to(0, 100)
blue_artist.polygon(5, 70)

green_artist.move_to(200, 100)
green_artist.polygon(8, 50)

# Теперь каждый рисует внизу другую фигуру
red_artist.move_to(-200, -100)
red_artist.polygon(3, 80)

blue_artist.move_to(0, -100)
blue_artist.polygon(4, 70)

green_artist.move_to(200, -100)
green_artist.polygon(10, 40)

screen.update()
screen.exitonclick()
```

Наблюдение: Каждый объект хранит своё состояние (цвет, положение) независимо от других. Это как три настоящих робота, работающих параллельно.

3.2. Метод set_color() — изменение атрибута после создания

Добавим возможность менять цвет «на лету» без создания нового художника.

Добавьте в класс `Artist` новый метод:

```
def set_color(self, new_color):
    """Меняет цвет пера художника."""
    self.turtle.pencolor(new_color)
```

Создайте `color_change_demo.py`:

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

class Artist:
    def __init__(self, color, speed):
        self.turtle = turtle.Turtle()
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)

    def polygon(self, sides, size):
        angle = 360 / sides
        for _ in range(sides):
            self.turtle.forward(size)
            self.turtle.left(angle)

    def move_to(self, x, y):
        self.turtle.penup()
        self.turtle.goto(x, y)
```

```
        self.turtle.pendown()

    def set_color(self, new_color):
        """Меняет цвет пера."""
        self.turtle.pencolor(new_color)

screen = turtle.Screen()
screen.title("Смена цвета на лету")

a = Artist("black", 5)

colors = ["red", "green", "blue", "orange", "purple"]

for i, col in enumerate(colors):
    a.move_to(-200 + i * 80, 0)
    a.set_color(col)  # меняем цвет существующего художника
    a.polygon(4, 60)

screen.exitonclick()
```

4. Уровень «Исследовательский»: Фабрика объектов и пределы памяти

(Новое применение, прорыв за границы)

Автоматизируем создание художников и исследуем ограничения системы.

4.1. Фабрика create_artists(n)

Создадим функцию, которая генерирует заданное количество художников со случайными параметрами.

Создайте `artist_factory.py`:

```
#!/usr/bin/env python3
import turtle
import random

class Artist:
    def __init__(self, color, speed, start_x, start_y):
        self.turtle = turtle.Turtle()
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)
        self.turtle.penup()
        self.turtle.goto(start_x, start_y)
        self.turtle.pendown()

    def polygon(self, sides, size):
        angle = 360 / sides
        for _ in range(sides):
            self.turtle.forward(size)
            self.turtle.left(angle)
```

```
def create_artists(n):
    """Создаёт список из n художников со случайными цветами и
позициями."""
    colors = ["red", "blue", "green", "orange", "purple", "pink", "cyan",
"brown"]
    artists = []
    for _ in range(n):
        col = random.choice(colors)
        spd = random.randint(5, 10)
        x = random.randint(-250, 250)
        y = random.randint(-200, 200)
        artist = Artist(col, spd, x, y)
        artists.append(artist)
    return artists

screen = turtle.Screen()
screen.title("Фабрика художников")
screen.tracer(0)

# Создаём 10 случайных художников
team = create_artists(10)
print(f"Создано художников: {len(team)}")

# Каждый рисует случайную фигуру
for artist in team:
    sides = random.randint(3, 8)
    size = random.randint(30, 80)
    artist.polygon(sides, size)

screen.update()
screen.exitonclick()
```

4.2. Мини-исследование: сколько объектов можно создать?

Вопрос: Сколько объектов `Artist` можно создать, прежде чем Python исчерпает память?

Создайте `memory_test.py` (запускайте осторожно, программа может замедлить систему):

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

# Создаём простой класс для теста (без turtle, чтобы не открывать окна)
class SimpleBot:
    def __init__(self, robot_id):
        self.id = robot_id
        self.status = "idle"
        self.data = [0] * 100  # немного данных

print("Тест: создание объектов до исчерпания памяти")
print("Для остановки нажмите Ctrl+C")

bots = []
count = 0
try:
    while True:
```

```
        bot = SimpleBot(count)
        bots.append(bot)
        count += 1
        if count % 100000 == 0:
            print(f"Создано объектов: {count}")
except MemoryError:
    print(f"\nПамять закончилась! Максимум объектов: {count}")
except KeyboardInterrupt:
    print(f"\nОстановлено пользователем. Создано объектов: {count}")

print(f"\nПримерный размер одного объекта: {sys.getsizeof(bots[0])} байт")
print(f"Теоретически в 1 ГБ памяти поместится ~{1_000_000_000 // sys.getsi
zeof(bots[0])} объектов")
```

Ожидаемый результат: Вы создадите сотни тысяч или миллионы объектов до исчерпания памяти. Ограничение — не в Python, а в объёме оперативной памяти робота.

Практический вывод: Для реальных робототехнических задач вы никогда не упрётесь в ограничение количества объектов. Гораздо важнее — их правильная архитектура.

5. Выводы по занятию

1. **Класс — это чертёж, объект — изделие.** `class Artist:` описывает структуру, `artist1 = Artist( ... )` создаёт конкретный экземпляр со своими данными.
2. **`__init__` и `self` — основа ООП.** Конструктор инициализирует атрибуты, `self` связывает методы с конкретным объектом.
3. **Инкапсуляция — данные и поведение вместе.** Художник сам знает свой цвет и умеет рисовать. Не нужно передавать параметры через десяток функций.
4. **Независимость объектов.** Три художника работают параллельно, не мешая друг другу. Это модель для многозадачности в ROS2.
5. **Фабрика — массовое производство.** Функция `create_artists(n)` создаёт любое количество объектов с заданными свойствами. Это паттерн, который мы будем использовать для тестирования.

Связь со следующим занятием: `Artist` — это базовый класс. Но что если мы хотим создать «Радужного художника», который автоматически меняет цвет на каждой стороне? Можно скопировать код, а можно — **унаследовать** его от `Artist` и переопределить только нужное. Это тема финального занятия блока: «Наследование и зачёт».

Домашнее задание:

- **Базовое**: Создайте класс `Robot` с атрибутами `name` и `battery`. Добавьте метод `report()`, который выводит информацию о роботе.
- **Продвинутое**: Добавьте в класс `Artist` метод `star(points, size)` и нарисуйте композицию из звёзд.
- **Исследовательское**: Модифицируйте `create_artists(n)` так, чтобы все художники рисовали одновременно (каждый по одной стороне многоугольника за итерацию).

Занятие 11: Наследование, итоговое занятие, зачёт

Содержание занятия: Расширение классов через наследование. Подведение итогов блока Python и ООП

Цели занятия

- **Образовательная**: Освоить механизм наследования — создание нового класса на базе существующего. Изучить вызов методов родителя через `super()`.
- **Развивающая**: Спроектировать иерархию классов, демонстрирующую полиморфизм.
- **Воспитательная**: Провести зачёт в форме защиты мини-проекта, оценив свой прогресс в программировании.

План занятия (90 минут)

1. **Организационный момент (5 мин)**
2. **Уровень «База»: Наследование и RainbowArtist (30 мин)**
   - Понятие наследования.
   - Создание `RainbowArtist(Artist)` — переопределение цвета для каждой стороны.
   - `super().__init__()` — вызов конструктора родителя.
3. **Уровень «Углублённый»: Иерархия Shape и полиморфизм (25 мин)**
   - Базовый класс `Shape`, наследники `Polygon` и `Star`.
   - Класс `Artist` рисует любую фигуру через метод `draw(shape)`.
   - Демонстрация мини-проекта.
4. **Уровень «Исследовательский»: Творческий проект и MRO (25 мин)**
   - Создание собственного класса-наследника с уникальной логикой.
   - Презентация проекта.
   - Мини-исследование: порядок разрешения методов (MRO).
5. **Зачёт и рефлексия (5 мин)**

1. Введение: Не переписывать, а расширять

На прошлом занятии мы создали класс `Artist`. Он отлично работает. Но что если нам нужен «Радужный художник», который на каждой стороне многоугольника меняет цвет? Переписывать весь класс? Нет!

Наследование — это механизм, позволяющий создать новый класс на основе существующего. Новый класс (наследник) получает все атрибуты и методы родителя и может добавить свои или изменить существующие.

Это как с роботами: есть базовый класс «Мобильный робот», а от него наследуются «Робот-курьер» и «Робот-исследователь» — у каждого свои особенности, но общая база.

Сегодня мы освоим наследование и проведём зачёт по всему блоку Python.

2. Уровень «База»: Наследование и RainbowArtist

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём класс-наследник и используем `super()`.

2.1. Понятие наследования

Наследование — это отношение «является разновидностью»:

- `RainbowArtist` является разновидностью `Artist`.
- `Polygon` является разновидностью `Shape`.
- Робот-курьер является разновидностью мобильного робота.

Синтаксис:

```
class Parent:
    ...

class Child(Parent):  # в скобках — имя родительского класса
    ...
```

2.2. RainbowArtist — переопределение цвета для каждой стороны

Создайте файл `rainbow_artist.py`:

```
nano ~/ws/rbt/scripts/rainbow_artist.py
```

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

# ── Родительский класс (как на прошлом занятии) ──
class Artist:
    def __init__(self, color, speed):
        self.turtle = turtle.Turtle()
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)

    def polygon(self, sides, size):
        angle = 360 / sides
        for _ in range(sides):
            self.turtle.forward(size)
            self.turtle.left(angle)

    def move_to(self, x, y):
        self.turtle.penup()
        self.turtle.goto(x, y)
        self.turtle.pendown()

# ── Класс-наследник ──
class RainbowArtist(Artist):
```

```
    """Художник, который рисует каждую сторону новым цветом."""

    def __init__(self, speed):
        # Вызываем конструктор родителя, передаём начальный цвет
        super().__init__("red", speed)
        # Свой список цветов для перебора
        self.colors = ["red", "orange", "gold", "green", "blue",
"purple", "violet"]

    def polygon(self, sides, size):
        """Переопределяем метод: каждая сторона — новый цвет."""
        angle = 360 / sides
        for i in range(sides):
            # Выбираем цвет по индексу (с зацикливанием)
            color = self.colors[i % len(self.colors)]
            self.turtle.pencolor(color)
            self.turtle.forward(size)
            self.turtle.left(angle)

# ── Демонстрация ──
screen = turtle.Screen()
screen.title("RainbowArtist — наследование")

# Обычный художник
a1 = Artist("black", 5)
a1.move_to(-150, 0)
a1.polygon(8, 60)

# Радужный художник (создаём без указания цвета!)
a2 = RainbowArtist(7)
a2.move_to(150, 0)
a2.polygon(8, 60)  # тот же метод, но поведение другое!

screen.exitonclick()
```

Ключевые моменты:

- `class RainbowArtist(Artist):` — наследуем от `Artist`.
- `super().__init__("red", speed)` — вызываем конструктор родителя, чтобы создать черепашку.
- `polygon` переопределён: метод с тем же именем, но другой реализацией. Это **полиморфизм**.

2.3. super() — мост к родителю

`super()` возвращает временный объект родительского класса, позволяя вызвать его методы.

```
class Child(Parent):
    def __init__(self, extra_param):
        super().__init__(parent_params)  # инициализируем родительскую
часть
        self.extra = extra_param         # добавляем свою
```

Это гарантирует, что всё, что должен сделать родитель при создании, будет сделано.

3. Уровень «Углублённый»: Иерархия Shape и полиморфизм

(Что под капотом)

Строим иерархию геометрических фигур и учим `Artist` работать с любой фигурой.

3.1. Базовый класс Shape и наследники

Создайте модуль `shapes_oop.py`:

```
nano ~/ws/rbt/scripts/shapes_oop.py
```

```
# shapes_oop.py — иерархия геометрических фигур

class Shape:
    """Базовый класс для всех фигур."""
    def __init__(self, name):
        self.name = name

    def draw(self, t):
        """Метод, который должны переопределить наследники."""
        raise NotImplementedError("Метод draw должен быть переопределён!")

class Polygon(Shape):
    def __init__(self, sides, size):
        super().__init__(f"{sides}-угольник")
        self.sides = sides
        self.size = size

    def draw(self, t):
        angle = 360 / self.sides
        for _ in range(self.sides):
            t.forward(self.size)
            t.left(angle)

class Star(Shape):
    def __init__(self, points, size):
        super().__init__(f"{points}-конечная звезда")
        self.points = points
        self.size = size

    def draw(self, t):
        angle = 180 - 180 / self.points
        for _ in range(self.points):
            t.forward(self.size)
            t.left(angle)
```

3.2. Artist рисует любую фигуру через draw(shape)

Создайте `polymorph_demo.py`:

```
#!/usr/bin/env python3
import turtle
from shapes_oop import Polygon, Star
```

```
class Artist:
    def __init__(self, color, speed):
        self.turtle = turtle.Turtle()
        self.turtle.pencolor(color)
        self.turtle.speed(speed)
        self.turtle.pensize(2)

    def draw(self, shape):
        """Рисует любую фигуру, у которой есть метод draw(t)."""
        print(f"Рисую: {shape.name}")
        shape.draw(self.turtle)

    def move_to(self, x, y):
        self.turtle.penup()
        self.turtle.goto(x, y)
        self.turtle.pendown()

# ── Демонстрация полиморфизма ──
screen = turtle.Screen()
screen.title("Полиморфизм: Artist рисует всё")

artist = Artist("blue", 5)

# Создаём список разных фигур
shapes_to_draw = [
    Polygon(3, 80),
    Polygon(6, 60),
    Star(5, 70),
    Star(7, 50),
    Polygon(8, 40),
]

x_pos = -250
for shape in shapes_to_draw:
    artist.move_to(x_pos, 0)
    artist.draw(shape)  # один и тот же метод для разных фигур!
    x_pos += 100

screen.exitonclick()
```

Полиморфизм в действии: `artist.draw(shape)` работает с любым объектом, у которого есть метод `draw(t)`. Не важно, `Polygon` это или `Star` — код одинаковый. Это ключевой принцип для построения гибких систем.

4. Уровень «Исследовательский»: Творческий проект и MRO

(Новое применение, прорыв за границы)

Создаём собственный класс и разбираемся с множественным наследованием.

4.1. Собственный класс-наследник (зачётное задание)

Задача: Создайте свой класс, наследующий от `Artist` или `Shape`, с уникальной логикой рисования.

Идеи для проекта:

- `SpiralArtist(Artist)` — рисует спираль вместо многоугольника.
- `FractalArtist(Artist)` — рисует фрактальное дерево.
- `DashedPolygon(Polygon)` — многоугольник из пунктирных линий.
- `House(Shape)` — фигура «домик» из квадрата и треугольника.

Требования:

1. Класс-наследник должен использовать `super().__init__()`.
2. Должен быть переопределён хотя бы один метод.
3. Демонстрационная программа должна показать работу класса.

Пример: `SpiralArtist`

```
class SpiralArtist(Artist):
    def __init__(self, speed):
        super().__init__("cyan", speed)

    def spiral(self, turns, step_growth):
        """Рисует спираль."""
        step = 5
        for _ in range(turns * 10):
            self.turtle.forward(step)
            self.turtle.left(89)
            step += step_growth
```

4.2. Мини-исследование: MRO — порядок разрешения методов

Вопрос: Что произойдёт при множественном наследовании? В каком порядке Python ищет методы?

Создайте `mro_demo.py`:

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

class A:
    def who(self):
        print("Метод класса A")

class B(A):
    def who(self):
        print("Метод класса B")

class C(A):
    def who(self):
        print("Метод класса C")

class D(B, C):  # множественное наследование
```

```
    pass

# Проверяем порядок разрешения методов
print("Порядок разрешения методов для класса D:")
for cls in D.__mro__:
    print(f"  {cls.__name__}")

d = D()
print("\nВызов d.who():")
d.who()  # какой метод вызовется?

print("\nОбъяснение:")
print("Python использует алгоритм C3-линеаризации (MRO).")
print("Метод ищется в порядке: D → B → C → A → object.")
print("Первый найденный метод побеждает — это B.who().")
```

Вывод: При множественном наследовании Python строит линейный порядок (Method Resolution Order), который гарантирует, что каждый класс-предок вызывается ровно один раз и в правильном порядке.

4.3. Зачёт: демонстрация мини-проекта

Критерии оценки зачётного проекта:

1. **Работоспособность** — программа запускается и выполняет заявленное.
2. **Использование ООП** — есть собственный класс, наследование, переопределение методов.
3. **Аккуратность кода** — переменные названы осмысленно, есть комментарии.
4. **Презентация** — ученик может объяснить, что делает его программа и как она устроена.

Темы для обсуждения на зачёте:

- Что такое класс и объект?
- Зачем нужен `self`?
- Что делает `__init__`?
- Чем наследование отличается от копирования кода?
- Что такое полиморфизм?
- Как Python ищет метод при вызове?

5. Выводы по занятию и блоку Python/ООП

Итоги занятия

1. **Наследование — расширение, а не переписывание.** `class Child(Parent):` даёт доступ ко всем методам родителя.
2. **super() — вызов родительской логики.** Позволяет не дублировать код конструктора или методов.
3. **Полиморфизм — «один интерфейс, много реализаций».** `artist.draw(shape)` работает с любой фигурой. Это основа для подключаемых модулей.

4. **MRO — алгоритм поиска метода.** При множественном наследовании Python строит линейный порядок, гарантирующий предсказуемость.

Итоги блока Python и ООП (занятия 3–11)

Мы прошли путь от простых переменных до объектно-ориентированного проектирования:

- **Базовый синтаксис:** переменные, циклы, условия, функции.
- **Структуры данных:** списки и словари.
- **Модульность:** разделение кода на файлы и импорт.
- **ООП:** классы, объекты, инкапсуляция, наследование, полиморфизм.

Этот фундамент необходим для следующего большого блока — **«Аппаратная платформа робота»**, где мы познакомимся с реальным железом: меканум-колёсами, лидаром, камерой и нейропроцессором.

Домашнее задание:

- **Базовое**: Повторите определения: класс, объект, наследование, полиморфизм.
- **Продвинутое**: Завершите зачётный проект, если не успели на занятии.
- **Исследовательское**: Изучите, что такое «абстрактный класс» в Python (модуль `abc`) и зачем он нужен.

Аппаратная платформа робота

Занятие 12: Обзор платформы: компьютер, контроллер, меканум-колёса

Содержание занятия: Архитектура мобильного робота. Знакомство с компонентами и всенаправленным движением

Цели занятия

- **Образовательная**: Изучить состав аппаратной платформы робота: одноплатный компьютер, микроконтроллер, моторы, меканум-колёса. Понять принцип всенаправленного движения.
- **Развивающая**: Связать абстрактное программирование с физическим устройством. Научиться идентифицировать компоненты и понимать их роль.
- **Воспитательная**: Сформировать бережное отношение к сложной технике через понимание её устройства.

План занятия (90 минут)

1. **Организационный момент и введение (5 мин)**
2. **Уровень «База»: Из чего состоит робот (35 мин)**
   - Одноплатный компьютер (Raspberry Pi) — «мозг».
   - Микроконтроллер STM32 — «спинной мозг».
   - Моторы и меканум-колёса — «ноги».
   - Чем меканум отличается от обычных колёс.
   - Визуальный осмотр компонентов на реальном роботе.
3. **Уровень «Углублённый»: Принцип всенаправленного движения (25 мин)**
   - Ролики на колёсах и векторы скоростей vx, vy, ωz.
   - Как STM32 принимает команды и управляет моторами.
   - Связь компьютер ↔ контроллер (UART/USB).
4. **Уровень «Исследовательский»: Расчёт скоростей и анализ (20 мин)**
   - Расчёт скоростей четырёх колёс из вектора (vx, vy, ωz).
   - Мини-исследование: почему меканум-платформа менее эффективна на неровной поверхности.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: От программ — к «железу»

Мы закончили большой блок по Python и ООП. Теперь мы переходим от виртуальной черепашки к настоящему роботу. Но прежде чем программировать движение, нужно понять, **из чего робот состоит** и **как эти части взаимодействуют**.

Наш робот — это сложная кибернетическая система: компьютер, контроллер, датчики, моторы. Сегодня мы познакомимся с «анатомией» робота и поймём, почему он может ездить боком.

2. Уровень «База»: Из чего состоит робот

(Инструмент, навык, интерфейс)

Обязательный уровень. Знакомимся с компонентами, их названиями и назначением.

2.1. Четыре ключевых компонента

Подойдите к роботу (учитель поставил его на демонстрационный стол). Рассмотрим его «анатомию».

Компонент	Аналогия	Где находится	Что делает
Одноплатный компьютер (Raspberry Pi 4/5)	Мозг	Верхняя плата в корпусе	Думает: считает траектории, обрабатывает изображения, запускает нейросети
Микроконтроллер STM32	Спинной мозг	Рядом с моторами, часто на нижней плате	Реагирует быстро: управляет скоростью каждого мотора, считает обороты колёс
Моторы	Мышцы	4 штуки, по одному на колесо	Вращают колёса
Механум-колёса	Ноги с роликами	4 штуки, по углам шасси	Позволяют ехать в любом направлении

Почему два вычислительных устройства?

- **Компьютер (RPi)** работает под управлением Linux. Он многозадачный, но может «задуматься».
- **Микроконтроллер (STM32)** работает в реальном времени. Он не отвлекается — только считает импульсы с датчиков и выдаёт ШИМ-сигналы на моторы. Ему доверяют самое критичное — управление движением.

2.2. Механум-колёса: что это и почему они особенные

Обычное колесо катится только вперёд-назад. Чтобы повернуть, нужно изменить его направление (как на автомобиле).

Механум-колесо — это колесо, на ободе которого закреплены **ролики** (маленькие бочонки) под углом 45°. Эти ролики вращаются свободно.

Что это даёт? Когда колесо вращается, ролики создают силу **под углом** к направлению вращения. Комбинируя вращение четырёх колёс, можно получить движение в любом направлении **без поворота корпуса**:

- Вперёд и назад.
- Влево и вправо (**крабовое движение**).
- По диагонали.
- Вращение на месте.

Посмотрите на колёса робота. Найдите ролики под углом. Обратите внимание: колёса установлены в определённом порядке — ролики левого переднего и правого заднего «смотрят» в одну сторону, правого переднего и левого заднего — в другую. Это важно для правильной кинематики.

2.3. Визуальный осмотр: найди компонент

Практическое задание (на реальном роботе):

1. Найдите и покажите: одноплатный компьютер, микроконтроллер STM32, моторы, механум-колёса.
2. Найдите: USB, лидар, камеру, Hailo-8.
3. Найдите аккумулятор.
4. Проследите провода: от аккумулятора к преобразователю, от преобразователя к компьютеру и моторам.

Запишите в тетрадь схему расположения компонентов.

3. Уровень «Углублённый»: Принцип всенаправленного движения

(Что под капотом)

Разбираем физику и математику механум-движения.

3.1. Векторы скоростей: vx, vy, ωz

Движение робота описывается **вектором скоростей**:

- `vx` — скорость вдоль оси X (вперёд/назад).
- `vy` — скорость вдоль оси Y (влево/вправо).
- `ωz` (омега-z) — угловая скорость вокруг вертикальной оси (вращение на месте).

В ROS2 этот вектор передаётся в сообщении `Twist` (топик `/cmd_vel`). Именно его мы будем заполнять в программах.

Как это работает физически: Каждый ролик на колесе преобразует вращение колеса в силу под углом 45°. Когда все четыре колеса вращаются согласованно, их векторы сил складываются, давая нужное направление движения.

3.2. Роль STM32: от вектора к оборотам

Компьютер думает в терминах `(vx, vy, ωz)`. Но моторы понимают только «сколько вольт подано». Кто переводит?

STM32 делает это в реальном времени:

1. Получает `Twist` от компьютера по UART/USB.
2. По формулам кинематики вычисляет, с какой скоростью должно вращаться каждое из четырёх колёс.
3. Выдаёт ШИМ-сигнал на драйверы моторов.

4. Считывает энкодеры (датчики оборотов) и корректирует скорость — это называется **замкнутый контур управления**.

Связь компьютер ↔ STM32:

```
Компьютер (RPi) ──UART/USB──> STM32 ──ШИМ──> Драйверы моторов ──> Моторы
                                STM32 <──Энкодеры──< Моторы
```

Это классическое разделение: компьютер даёт команду, контроллер её исполняет с высокой точностью.

4. Уровень «Исследовательский»: Расчёт и анализ ограничений

(Новое применение, прорыв за границы)

Выводим формулы и исследуем, где меканум-платформа даёт сбой.

4.1. Расчёт скоростей колёс из вектора (vx, vy, ωz)

Задача: Даны желаемые скорости робота `vx`, `vy`, `ωz`. Найти скорости вращения каждого из четырёх колёс: `ω1`, `ω2`, `ω3`, `ω4`.

Дано (геометрия робота):

- `a` — половина ширины колёсной базы (расстояние от центра до колеса по X).
- `b` — половина длины колёсной базы (расстояние от центра до колеса по Y).
- `r` — радиус колеса.

Формулы (прямая кинематика меканум):

Для каждого колеса скорость складывается из трёх составляющих:

1. Поступательное движение: `vx` и `vy`.
2. Вращательное движение: `ωz`, которое добавляет или вычитает скорость в зависимости от положения колеса.

Упрощённые формулы (без учёта угла роликов):

```
ω1 = (1/r) * (vx - vy - ωz * (a + b))   // переднее левое
ω2 = (1/r) * (vx + vy + ωz * (a + b))   // переднее правое
ω3 = (1/r) * (vx + vy - ωz * (a + b))   // заднее левое
ω4 = (1/r) * (vx - vy + ωz * (a + b))   // заднее правое
```

Практическое задание: Возьмите лист бумаги. Пусть `vx = 0.2 м/с`, `vy = 0`, `ωz = 0`, `r = 0.05 м`, `a = 0.15 м`, `b = 0.20 м`.

1. Подставьте в формулы и вычислите `ω1`, `ω2`, `ω3`, `ω4`.
2. Какие колёса вращаются быстрее при движении вперёд? (Все одинаково).
3. А при крабовом движении (`vx = 0`, `vy = 0.2`, `ωz = 0`)? (Противоположные колёса вращаются в разные стороны).

Проверьте себя: при `vx = 0.2, vy = 0, ωz = 0` все четыре колеса должны получить одинаковую скорость `0.2 / 0.05 = 4 рад/с`.

4.2. Мини-исследование: меканум на неровной поверхности

Вопрос: Почему меканум-платформа отлично работает на ровном полу, но буксует на ковре или гравии?

Методика исследования (мысленный эксперимент + наблюдение):

1. Вспомните устройство меканум-колеса: ролики вращаются свободно.
2. На ровной поверхности ролик касается пола в одной точке и катится.
3. На неровной поверхности (ковёр, стык плитки, гравий):
   • Ролик может провалиться в углубление.
   • Ворс ковра цепляется за ролик, мешая ему свободно вращаться.
   • Разные колёса теряют сцепление в разное время → нарушается согласованность векторов сил.
4. Результат: робот «плывёт» не туда, куда рассчитано, или вообще не может сдвинуться.

Инженерный компромисс: Меканум-колёса дают непревзойдённую манёвренность на **ровных** поверхностях (склады, цеха). Но за это приходится платить:

• Худшей проходимостью.
• Повышенным износом роликов.
• Более сложной математикой управления.

Задание: Осмотрите полигон для испытаний. Есть ли на нём неровности? Как вы думаете, как робот поведёт себя на стыке двух плиток?

5. Выводы по занятию

1. **Робот — это система из четырёх ключевых компонентов.** Компьютер думает, контроллер управляет, моторы крутят, меканум-колёса преобразуют вращение в любое направление.
2. **Разделение «мозг — спинной мозг» критично.** RPi может «зависнуть» на обработке нейросети, но STM32 продолжит удерживать скорость колёс.
3. **Меканум-колёса — инженерный шедевр.** Ролики под 45° позволяют движение в любом направлении без поворота корпуса.
4. **Вектор (vx, vy, ωz) — универсальный язык движения.** Именно в таком виде команды передаются в ROS2. Мы научимся заполнять его программно.
5. **У каждой технологии есть границы.** Меканум-платформа — чемпион манёвренности, но только на ровной поверхности. Инженер должен знать ограничения своего инструмента.

Связь со следующим занятием: Теперь мы знаем, как робот движется. Но чтобы двигаться осмысленно, ему нужно **видеть** окружающий мир. За это отвечают лидар и камера — тема следующего занятия.

Домашнее задание:

- **Базовое**: Зарисуйте схему робота с подписанными компонентами.
- **Продвинутое**: Вычислите скорости колёс для движения робота по диагонали (`vx = 0.2` , `vy = 0.2` , `ωz = 0`).
- **Исследовательское**: Найдите в интернете видео сравнения меканум-колёс и обычных «омни-колёс». В чём их конструктивное отличие и какие задачи решают омни-колёса?

Занятие 13: Лидар и камера: устройство, принципы, характеристики

Содержание занятия: Сенсоры робота. Как робот «видит» мир

Цели занятия

- **Образовательная**: Изучить устройство и принцип работы двух основных сенсоров робота: лидара (лазерного дальномера) и камеры. Понять их ключевые характеристики.
- **Развивающая**: Сформировать понимание разницы между активным и пассивным сенсором, их сильных и слабых сторон.
- **Воспитательная**: Показать, что данные с сенсоров — это основа для принятия решений автономным роботом.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Что такое лидар и камера (35 мин)**
    - Лидар: что измеряет, как выглядит, основные цифры.
    - Камера: зачем нужна, куда смотрит, основные цифры.
    - Визуальный осмотр датчиков на роботе.
3. **Уровень «Углублённый»: Физические принципы (25 мин)**
    - Лидар: принцип лазерной дальнометрии (ToF), угол обзора, облако точек.
    - Камера: матрица, фокусное расстояние, угол обзора, дисторсия.
4. **Уровень «Исследовательский»: Сравнение и анализ (20 мин)**
    - Сравнение характеристик разных моделей лидаров и камер.
    - Мини-исследование: почему лидар плохо видит стекло.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Как робот познаёт мир

На прошлом занятии мы изучили, из чего состоит робот и как он движется. Но чтобы двигаться **осмысленно**, роботу нужна информация об окружающем мире. За это отвечают **сенсоры**.

У нашего робота два главных «органа чувств»:

- **Лидар (LIDAR)** — измеряет расстояния до препятствий с помощью лазера. Это как летучая мышь, но с инфракрасным лучом.
- **Камера** — видит мир так же, как мы, но в виде матрицы чисел.

Сегодня мы разберём, как они устроены и почему роботу нужны оба.

2. Уровень «База»: Что такое лидар и камера

(Инструмент, навык, интерфейс)

Обязательный уровень. Знакомимся с датчиками визуально и узнаём ключевые цифры.

2.0. Настройка окружения

Для того, чтобы можно было выпонять задания этого пособия ROS2 lyrical должна быть установлена в системе

Основной стек робота должен быть запущен

- команда `ros2 launch mecanum_bringup bridge_ekf_lidar.launch.py start_autopilot:=false` - для оригинального робота;
- если это, например, TurtleBot со стандартными топиками `/cmd_vel`, `/odom`, `/scan`, `/camera/image_raw`, `/robot_description`, то команда запуска автопилота робота (если нужен) `ros2 run robot_autopilot autopilot_node.py` - см раздел 4.2 первого занятия;
- если вместо робота симулятор, то он запускается командами:
  - `ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_rviz:=true` - продвинутая версия видеосимуляции без автопилота;
  - `ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_rviz:=true use_lite:=true` - облегченная версия видеосимуляции без автопилота, в первых четырех главах рекомендуется использовать именно этот способ запуска симулятора;
  - `ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_rviz:=true start_autopilot:=true` - с автопилотом - начиная с 5 главы;
  - `ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_rviz:=true use_lite:=true start_autopilot:=true` - с автоматическим стартом rviz2 и облегченной версией видеосимуляции, с автопилотом.

Для того, чтобы команды ROS2 работали, нужно «активировать» окружение.

Подключитесь к роботу (симулятору) по SSH и выполните (установка симулятора на ПК и автопилота на робота с ROS2 описано в конце первого занятия):

```
if ! grep -q 'source /opt/ros/lyrical/setup.bash' ~/.bashrc; then
    echo "source /opt/ros/lyrical/setup.bash" >> ~/.bashrc
```

```
fi
if ! grep -q 'source /opt/ros/custom_nodes/setup.bash' ~/.bashrc; then
    echo "source /opt/ros/custom_nodes/setup.bash" >> ~/.bashrc
fi
if ! grep -q '/opt/drv' ~/.bashrc; then
    echo 'export PYTHONPATH=$PYTHONPATH:/opt/drv' >> ~/.bashrc
fi
source ~/.bashrc
```

(Путь может отличаться: `lyrical` — версия ROS2 для Ubuntu 26.04. Уточните у преподавателя.) После этого при каждом новом SSH-соединении окружение будет настроено автоматически.

Проверим, что топики ROS2 публикуют сообщения:

```
ros2 topic echo /odom
```

2.1. Лидар: лазерная рулетка на скорости

Лидар (Light Detection and Ranging) — это устройство, которое измеряет расстояние до объектов с помощью лазерного луча.

Как выглядит: На роботе лидар — это небольшая круглая башенка, которая постоянно вращается. Найдите её на роботе.

Что измеряет: Расстояние до ближайшего препятствия в направлении луча. За один оборот (обычно 10 раз в секунду) лидар делает сотни или тысячи измерений по кругу.

Основные цифры (для нашего лидара):

- **Дальность:** до 12–40 метров (в зависимости от модели и отражающей способности объекта).
- **Угол обзора:** 360° (полный круг).
- **Частота:** 10 Гц (10 оборотов в секунду).
- **Количество точек на оборот:** от 400 до 4000 (зависит от модели).

В ROS2 данные лидара публикуются в топик `/scan`.

2.2. Камера: глаз робота

Камера — это привычный нам датчик, который фиксирует двумерное изображение. Но для робота это не «картинка», а матрица чисел: яркость каждого пикселя в трёх цветовых каналах (RGB).

Где находится: Обычно закреплена спереди робота, «смотрит» вперёд. Найдите её на роботе.

Зачем нужна:

- Распознавать объекты (люди, знаки, другие роботы).
- Читать ArUco-маркеры для точного позиционирования.
- Следовать по линии.
- Строить карту окружения (в паре с лидаром).

Основные цифры (для нашей камеры):

- **Разрешение:** например, 1920×1080 (Full HD) — это 2 мегапикселя.
- **Частота кадров:** 30 FPS (кадров в секунду).
- **Угол обзора:** обычно 60–90° по горизонтали (не 360°, как у лидара!).

В ROS2 изображения публикуются в топик `/camera/image_raw`.

2.3. Визуальный осмотр датчиков

Практическое задание:

1. Найдите лидар. Обратите внимание: он вращается? (Если робот включён и стек ROS2 запущен — да).
2. Найдите камеру. Куда она направлена?
3. Сравните их положение: лидар обычно сверху, чтобы видеть на 360°. Камера — спереди, чтобы смотреть вперёд.
4. Подключитесь к роботу по SSH. Выполните:

   ```
   ros2 topic list | grep -E "scan|image"
   ```

   Вы должны увидеть топики `/scan` и `/camera/image_raw` — значит, датчики работают.

3. Уровень «Углублённый»: Физические принципы

(Что под капотом)

Разбираем, как именно работают лидар и камера.

3.1. Принцип лазерной дальнометрии (Time of Flight)

Лидар работает по принципу **измерения времени полёта (ToF)** лазерного импульса:

1. Лазерный диод испускает короткий инфракрасный импульс.
2. Импульс летит до препятствия, отражается и возвращается обратно.
3. Высокоточный фотоприёмник фиксирует отражённый сигнал.
4. Расстояние вычисляется по формуле: `D = c × t / 2`, где `c` — скорость света (≈300 000 км/с), `t` — измеренное время полёта.

Почему «/ 2»? Потому что луч проходит расстояние **туда и обратно**.

Как сканирует 360°? Вращающееся зеркало или сама головка лидара направляет луч в разные стороны. За один оборот получается **облако точек** — набор измерений с углами и расстояниями.

3.2. Облако точек и угловое разрешение

Результат одного оборота лидара — это массив `ranges[]`, где каждый элемент — расстояние в данном направлении. Направление вычисляется как `angle = angle_min + i × angle_increment`.

Угловое разрешение — это шаг между соседними измерениями. Например, если `angle_increment = 0.35°`, то на полный круг (360°) приходится `360 / 0.35 ≈ 1028` точек.

Практика (в терминале робота):

```
ros2 topic echo /scan --field ranges --once | head -20
```

Вы увидите массив чисел — это расстояния в метрах.

3.3. Камера: матрица, фокус, искажения

Матрица — это светочувствительный сенсор (CMOS), разбитый на миллионы пикселей. Каждый пиксель измеряет интенсивность света в своём цветовом канале (красный, зелёный, синий).

Фокусное расстояние — определяет угол обзора:

- Короткий фокус (широкоугольная линза) — большой угол обзора, но объекты кажутся дальше.
- Длинный фокус (телеобъектив) — узкий угол, но объекты «приближены».

Дисторсия — искривление прямых линий, особенно заметное по краям кадра:

- «Бочка» — края выгнуты наружу.
- «Подушка» — края вогнуты внутрь.

Для точных измерений (например, определение позы по ArUco-маркеру) камеру нужно **калибровать** — этот процесс исправляет дисторсию. Мы сделаем это на одном из следующих занятий.

4. Уровень «Исследовательский»: Сравнение и анализ

(Новое применение, прорыв за границы)

Сравниваем датчики по спецификациям и исследуем фундаментальные ограничения.

4.1. Сравнение характеристик по даташитам

Задание: Учитель предоставляет даташиты (технические спецификации) двух лидаров и двух камер. Заполните сравнительную таблицу:

Характеристика	Лидар А (наш)	Лидар Б (другой)	Камера А (наша)	Камера Б (другая)
Дальность / разрешение				
Угол обзора				
Частота				
Цена (примерно)				
Масса				
Энергопотребление				

Вывод: Вы увидите, что:

- Лидары с большей дальностью стоят кратно дороже.
- У камер с высоким разрешением растёт объём данных для обработки.
- Выбор датчика — всегда компромисс между точностью, скоростью и стоимостью.

4.2. Мини-исследование: почему лидар плохо видит стекло

Проблема: Если направить лидар на стеклянную дверь или окно, он может «не заметить» препятствия или измерить расстояние неправильно.

Причины:

1. **Прозрачность.** Стекло пропускает большую часть инфракрасного излучения (именно в ИК-диапазоне работает лидар). Луч просто проходит сквозь стекло, не отражаясь.
2. **Зеркальное отражение.** Если луч падает под острым углом, он может отразиться как от зеркала — в сторону, а не обратно в приёмник.
3. **Поглощение.** Некоторые стёкла с покрытием активно поглощают ИК-излучение.

Как это исправить:

- **Комбинация с камерой.** Камера видит стекло (блики, раму, наклейки). Нейросеть может детектировать «стеклянные» паттерны.
- **Ультразвуковые датчики.** Звук отражается от стекла, в отличие от лазера. Но у них хуже точность.
- **Предварительная карта.** Если робот знает, что в этом месте стеклянная дверь, он будет осторожнее.

Эксперимент (если есть прозрачная поверхность на полигоне):

1. Поставьте робота перед стеклом.
2. Посмотрите данные `/scan` (через `ros2 topic echo /scan --field ranges --once`).
3. Видит ли лидар стекло? Сравните с показаниями, когда перед роботом стена.
4. Сделайте снимок с камеры. Видно ли стекло на изображении?

5. Выводы по занятию

1. **Лидар — активный датчик расстояния.** Измеряет время полёта лазерного импульса. Даёт облако точек на 360°. Публикует данные в `/scan`.
2. **Камера — пассивный датчик изображения.** Фиксирует матрицу яркостей RGB. Даёт детальную картинку, но только вперёд и без точного расстояния. Публикует в `/camera/image_raw`.
3. **Лидар и камера дополняют друг друга.** Лидар даёт точную геометрию, камера — семантику (что это за объект). Вместе они — основа для автономной навигации.
4. **У каждого сенсора есть слабости.** Лидар плохо видит прозрачные объекты, камера — чувствительна к освещению.
5. **Выбор датчика — инженерный компромисс.** Мы анализируем характеристики и выбираем то, что оптимально для задачи.

Связь со следующим занятием: Теперь мы знаем, как робот видит мир. Но чтобы **понимать** увиденное (распознавать объекты, людей, знаки), нужен искусственный интеллект. Для этого на роботе установлен специальный ускоритель — **нейропроцессор Hailo-8**, о котором пойдёт речь на следующем занятии.

Домашнее задание:

- **Базовое**: Запишите в тетрадь основные характеристики нашего лидара и камеры (учитель их озвучит).
- **Продвинутое**: Подключитесь к роботу и изучите вывод `ros2 topic info /scan` и `ros2 topic info /camera/image_raw`. Какие типы сообщений используются?
- **Исследовательское**: Найдите в интернете, что такое «лидарная засветка» и почему несколько лидаров в одном помещении могут мешать друг другу.

Занятие 14: Нейропроцессор Hailo-8

Содержание занятия: Специализированный ускоритель для искусственного интеллекта на борту робота

Цели занятия

- **Образовательная**: Понять, что такое нейропроцессор, зачем он нужен роботу и чем отличается от CPU и GPU.
- **Развивающая**: Сформировать представление о гетерогенных вычислениях: разные процессоры для разных задач.
- **Воспитательная**: Показать, что современная робототехника — это симбиоз механики, электроники и искусственного интеллекта.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**

2. **Уровень «База»: Знакомство с Hailo-8 (35 мин)**
    • Что такое нейропроцессор и зачем он нужен.
    • Где находится Hailo-8 на роботе, как выглядит.
    • Основная функция: быстрое распознавание объектов.
3. **Уровень «Углублённый»: Архитектура и сравнение (25 мин)**
    • Потоковая архитектура Hailo-8.
    • Сравнение CPU vs GPU vs NPU.
    • Преимущества: энергопотребление и задержка.
4. **Уровень «Исследовательский»: Обзор рынка и анализ (20 мин)**
    • Обзор других edge-ускорителей.
    • Мини-исследование: почему низкая задержка критична для робота.
5. **Подведение итогов и рефлексия (5 мин)**

1. Введение: Мозг робота — не только процессор

На прошлых занятиях мы узнали, что у робота есть «мозг» (Raspberry Pi), «спинной мозг» (STM32) и «органы чувств» (лидар и камера).

Но представьте: камера выдаёт 30 кадров в секунду. На каждом кадре нужно найти людей, знаки, препятствия. Обычный процессор (CPU) будет делать это медленно, а графический (GPU) — быстро, но съест весь аккумулятор.

Решение — **нейропроцессор (NPU)**, специализированный чип, созданный только для одной задачи: быстро и экономно выполнять нейросети. На нашем роботе это **Hailo-8**.

Сегодня мы разберёмся, что это за устройство и почему оно — ключ к автономности.

2. Уровень «База»: Знакомство с Hailo-8

(Инструмент, навык, интерфейс)

Обязательный уровень. Узнаём, что такое NPU и где он находится.

2.1. Что такое нейропроцессор и зачем он роботу

Нейропроцессор (NPU — Neural Processing Unit) — это специализированный процессор, архитектура которого оптимизирована для операций, из которых состоят нейронные сети: умножение матриц, свёртки, функции активации.

Аналогия:

- **CPU (центральный процессор)** — универсал. Умеет всё, но ничего не умеет супер-быстро.
- **GPU (графический процессор)** — художник. Отлично считает параллельно, но прожорлив.

- **NPU (нейропроцессор)** — узкий специалист по ИИ. Считает нейросети в 10–100 раз эффективнее CPU и в разы экономичнее GPU.

Зачем он роботу:

- Камера видит пешехода → NPU за 10 миллисекунд говорит: «Это человек, координаты (x, y)».
- Без NPU этот же анализ занял бы 200 миллисекунд на CPU. За это время робот на скорости мог бы проехать 20 см и наткнуться на препятствие.

2.2. Где находится Hailo-8 и как выглядит

Практическое задание (на реальном роботе):

1. Найдите Hailo-8 на плате робота. Обычно это небольшая плата-ускоритель, вставленная в разъём M.2 или подключённая через USB.
2. Обратите внимание на маркировку: «Hailo-8».
3. Сравните размер с Raspberry Pi: Hailo-8 значительно компактнее.

Как проверить, что он работает (в терминале робота):

```
lspci | grep -i hailo
```

Если устройство определяется, вы увидите строку с упоминанием Hailo.

2.3. Основная функция: быстрое распознавание объектов

Hailo-8 предназначен для **инференса** — выполнения уже обученной нейросети. Он не обучает модели (это делают на мощных серверах), а применяет их.

Что он распознаёт (примеры):

- Люди, велосипедисты, автомобили.
- Дорожные знаки, ArUco-маркеры.
- Препятствия, свободное пространство.

Как это выглядит в коде (забегая чуть вперёд):

```
# Упрощённый пример (реальный код будет позже)
detections = hailo.detect(image)  # image — кадр с камеры
for obj in detections:
    print(f"Обнаружен {obj.class_name} с уверенностью {obj.confidence}%")
```

3. Уровень «Углублённый»: Архитектура и сравнение

(Что под капотом)

Разбираем, как Hailo-8 устроен внутри и почему он эффективнее.

3.1. Потоковая архитектура Hailo-8

В отличие от CPU, который выполняет инструкции последовательно, Hailo-8 построен по **потоковому (dataflow) принципу**:

1. Данные (пиксели изображения) «протекают» через конвейер вычислительных блоков.
2. Каждый блок выполняет свою операцию (свёртку, активацию, пулинг) и передаёт результат дальше.
3. Все блоки работают **одновременно** (параллельно).

Аналогия: Это как конвейер на заводе: пока один рабочий прикручивает колесо, другой уже красит кузов следующей машины. В CPU — один рабочий делает всё по очереди.

Ключевые цифры Hailo-8:

• Производительность: до 26 TOPS (триллионов операций в секунду).

• Энергопотребление: всего 2.5–5 Вт.

• Для сравнения: CPU Raspberry Pi 4 — около 0.1 TOPS, GPU — около 0.5 TOPS.

3.2. Сравнение CPU vs GPU vs NPU

Характеристика	CPU (RPi4)	GPU (видеокарта)	NPU (Hailo-8)
Универсальность	Абсолютная	Графика и параллельные вычисления	Только нейросети
Производительность на нейросетях	Низкая (FPS < 5)	Высокая (FPS 30+)	Очень высокая (FPS 30+)
Энергопотребление	5–10 Вт	50–200 Вт	2.5–5 Вт
Задержка (latency)	Большая (>100 мс)	Средняя (20–50 мс)	Малая (<10 мс)
Цена	~$50 (в составе RPi)	$200–$1000+	~$80–$150

Вывод: NPU — это «серебряная пуля» для задач ИИ на мобильном роботе: максимум производительности при минимуме энергии.

4. Уровень «Исследовательский»: Обзор рынка и анализ

(Новое применение, прорыв за границы)

Сравниваем Hailo-8 с конкурентами и думаем о требованиях реального времени.

4.1. Обзор других edge-ускорителей

Hailo-8 не единственный. Вот его конкуренты:

Устройство	Производительность	Энергопотребление	Интерфейс
Google Coral Edge TPU	4 TOPS	2 Вт	USB / M.2
Intel Neural Compute Stick 2	1 TOPS	2.5 Вт	USB
NVIDIA Jetson Nano	0.5 TOPS (GPU)	5–10 Вт	Полноценный компьютер
Hailo-8	26 TOPS	2.5–5 Вт	M.2 / Mini PCIe

Ключевое отличие:

- Google Coral и Intel NCS — это «флешки» с NPU внутри. Подключаются к существующему компьютеру.
- NVIDIA Jetson Nano — это **целый компьютер** со встроенным GPU. Дороже, больше, но умеет и обучать.
- Hailo-8 — высокопроизводительный сопроцессор, сравнительно новый игрок.

4.2. Мини-исследование: задержка vs точность

Вопрос: Что важнее для движущегося робота — максимальная точность распознавания или минимальная задержка?

Мысленный эксперимент: Робот едет со скоростью 1 м/с. Перед ним внезапно появляется препятствие.

- **Система А (CPU, точная модель):** Распознаёт препятствие за 200 мс с точностью 99.9%. За это время робот проехал **20 см** и уже врезался.
- **Система Б (NPU, быстрая модель):** Распознаёт препятствие за 10 мс с точностью 95%. За это время робот проехал всего **1 см** и успел затормозить.

Расчёт тормозного пути:

```
Время реакции системы: t_reaction
Тормозной путь при скорости v и ускорении торможения a:
S = v * t_reaction + v² / (2 * a)

При v = 1 м/с, a = 2 м/с²:
Система А (t_reaction = 0.2 с): S = 1 * 0.2 + 1 / 4 = 0.45 м
Система Б (t_reaction = 0.01 с): S = 1 * 0.01 + 1 / 4 = 0.26 м
```

Разница почти в 2 раза! Быстрая реакция спасла от столкновения.

Вывод: В робототехнике **задержка (latency) часто важнее точности**. Лучше быстро заметить препятствие с 95% уверенностью, чем точно понять, что это был за объект, но уже после столкновения.

5. Выводы по занятию

1. **NPU — специалист по нейросетям.** Hailo-8 выполняет задачу распознавания в десятки раз быстрее CPU при минимальном энергопотреблении.
2. **Гетерогенные вычисления — стандарт робототехники.** CPU для логики, STM32 для моторов, NPU для ИИ. Каждый делает то, в чём он лучший.

3. **Hailo-8 использует потоковую архитектуру.** Данные текут через конвейер вычислительных блоков, обеспечивая параллелизм без сложного программирования.
4. **Задержка — ключевой параметр для безопасности.** В мобильной робототехнике «быстро» часто означает «безопасно».
5. **Выбор ускорителя — инженерный компромисс.** Hailo-8, Google Coral, NVIDIA Jetson — у каждого свои сильные стороны для разных задач.

Связь со следующим занятием: Мы рассмотрели все компоненты робота по отдельности. Пришло время **собрать их в единую систему**. На следующем занятии — «Сборка робота: крепление, коммутация, питание» — мы физически соединим всё вместе.

Домашнее задание:

- **Базовое**: Запишите в тетрадь, зачем роботу NPU и чем он отличается от CPU.
- **Продвинутое**: Подключитесь к роботу и выполните `lsusb`. Найдите в выводе Hailo-8.
- **Исследовательское**: Сравните характеристики Hailo-8 и Google Coral Edge TPU. Какой лучше подходит для задачи детектирования пешеходов на скорости 2 м/с? Аргументируйте.

Занятие 15: Сборка робота: крепление, коммутация, питание

Содержание занятия: Физическая сборка мобильной платформы. Практическое занятие

Цели занятия

- **Образовательная**: Освоить порядок сборки робота: крепление компонентов на шасси, коммутация проводов, подключение питания. Научиться соблюдать меры предосторожности.
- **Развивающая**: Понять логику соединений: почему лидар в UART, а контроллер — в USB.
- **Воспитательная**: Сформировать культуру аккуратной работы с электроникой и ответственность за сохранность оборудования.

План занятия (90 минут)

1. **Организационный момент и инструктаж по ТБ (10 мин)**
2. **Уровень «База»: Порядок сборки (40 мин)**
   - Крепление компонентов на шасси.
   - Подключение датчиков и ускорителя.
   - Подключение аккумулятора.
3. **Уровень «Углублённый»: Логика коммутации и питание (25 мин)**
   - Что куда и почему: UART vs USB.
   - Преобразователи напряжения.
   - Меры предосторожности.

4. **Уровень «Исследовательский»: Энергопотребление и автономность (10 мин)**
    - Расчёт энергопотребления.
    - Кто главный потребитель?
5. **Подведение итогов и проверка сборки (5 мин)**

1. Введение: От теории — к практике

Мы изучили все компоненты робота по отдельности. Теперь самый ответственный этап — **физическая сборка**. Сегодня мы возьмём отвёртки, провода и превратим набор деталей в единую работающую систему. (Если физически робота нет и есть огромное желание его собрать, то в сети интернет есть огромное количество описаний подходящих конструкций, например, TurtleBot. Для консультаций можно обратиться к автору. Кроме того для дальнейшего изучения материала книги можно использовать симулятор.)

Важно: Это не просто «соединить провода». Это инженерная задача: обеспечить надёжный контакт, правильную полярность, защиту от помех и статического электричества.

> **Инструктаж по технике безопасности (обязателен перед началом!):**
>
> 1. Перед любыми подключениями **отключайте аккумулятор**.
> 2. Соблюдайте **полярность** (красный — плюс, чёрный — минус).
> 3. Перед работой **снимите статическое электричество**, прикоснувшись к заземлённому металлическому предмету.
> 4. При пайке или работе с горячим клеем соблюдайте осторожность.
> 5. Если вы не уверены — **спросите преподавателя**.

2. Уровень «База»: Порядок сборки

(Инструмент, навык, интерфейс)

Обязательный уровень. Практическая сборка робота под руководством учителя.

2.1. Крепление компонентов на шасси

Инструменты: отвёртка, стойки, винты М3.

Порядок действий:

1. **Шасси** — базовая рама. Убедитесь, что она ровная и все отверстия совпадают.
2. **Моторы с колёсами** — 4 штуки. Прикрутите каждый мотор к своему посадочному месту. Убедитесь, что:
    - Меканум-колёса установлены **правильным узором** (ролики «смотрят» в центр робота спереди и сзади, наружу — по бокам).
    - Колёса не задевают за шасси при вращении.

3. **Одноплатный компьютер (Raspberry Pi)** — закрепите на стойках в верхней части шасси. Не перетяните винты!
4. **Микроконтроллер STM32** (если отдельной платой) — закрепите рядом с моторным отсеком.
5. **Лидар** — закрепите на верхней площадке так, чтобы ему ничего не мешало вращаться.
6. **Камера** — закрепите спереди. Она должна «смотреть» вперёд, не быть загорожена проводами.
7. **Hailo-8** — вставьте в разъём M.2 на Raspberry Pi 5 или подключите через переходник.

2.2. Коммутация: что куда подключать

Проверьте, что аккумулятор ОТКЛЮЧЁН.

Выполните подключения в следующем порядке:

Устройство	Разъём на устройстве	Куда подключать на роботе
Лидар	UART	гребенка Raspberry Pi
Камера	CSI	CSI-порт Raspberry Pi
STM32 (управление моторами)	UART (часто через USB-UART переходник)	USB-порт Raspberry Pi
Hailo-8	M.2	M.2 разъём на RPi5
Аккумулятор	XT60 / клеммы	Входной разъём платы питания

Практическое задание: Подключите все устройства. Проверьте, что разъёмы вставлены до щелчка. Провода не должны болтаться и мешать вращению лидара.

2.3. Подключение аккумулятора

Самое ответственное подключение.

1. Найдите разъём аккумулятора и ответный разъём на роботе.
2. Убедитесь, что **полярность соблюдена** (красный к красному/плюсу, чёрный к чёрному/минусу).
3. Аккуратно, но плотно соедините разъём.
4. Проверьте индикаторы:
    - На Raspberry Pi должны загореться красный (питание) и зелёный (активность) светодиоды.
    - Лидаре начнёт очень тихо шуршать.
    - Hailo-8 может иметь свой индикатор.

Если что-то не горит — **немедленно отключите аккумулятор** и проверьте подключения.

3. Уровень «Углублённый»: Логика коммутации и питание

(Что под капотом)

Понимаем, почему всё подключено именно так.

3.1. Почему лидар — в UART, а STM32 — в USB?

USB (Universal Serial Bus):

- Высокая пропускная способность (до сотен мегабит/с).
- Подходит для передачи больших объёмов данных: облака точек лидара, видеопоток с камеры.
- Поддерживает «горячее подключение» (можно подключать при включённом питании, хотя лучше так не делать).

UART (Universal Asynchronous Receiver/Transmitter):

- Простой последовательный протокол.
- Низкая задержка, детерминированное время передачи.
- Идеален для команд управления моторами: `Twist` — это всего несколько байт, их нужно передать максимально быстро и без задержек.

Схема коммуникации:

```
[Камера] ──> CSI (видео, сотни МБ/с)──> [Raspberry Pi] ──> USB ──> UART
(команды, байты)──> [STM32] ──> Моторы
[Лидар]  ──> UART (облако точек) ──> [Raspberry Pi]
[Hailo-8] ──> M.2/PCIe (данные нейросети) ──> [Raspberry Pi]
```

3.2. Преобразователи напряжения: от 12В к 5В и 3.3В

Аккумулятор робота — обычно 12В (литий-полимерный, 3S). Но разные компоненты требуют разного напряжения:

Компонент	Требуемое напряжение	Как обеспечивается
Raspberry Pi	5В	Понижающий DC-DC преобразователь (step-down)
STM32	3.3В	Линейный стабилизатор или DC-DC
Моторы	12В (или меньше через ШИМ)	Напрямую от аккумулятора через драйверы
Лидар	5В (питается по USB)	От Raspberry Pi
Камера	(питается по CSI)	От Raspberry Pi

DC-DC преобразователь (step-down, «понижайка») — устройство, которое берёт 12В от аккумулятора и выдаёт стабильные 5В для компьютера. Без него Raspberry Pi сгорит от 12В!

3.3. Меры предосторожности: полярность и статика

Полярность:

- Красный провод — всегда «+» (плюс).
- Чёрный провод — всегда «−» (минус, земля, GND).
- **Переполюсовка (плюс на минус) — мгновенное убийство электроники!** Всегда перепроверяйте.

Статическое электричество (ESD):

- На вашем теле может накопиться заряд в тысячи вольт (особенно в сухом помещении, в синтетической одежде).
- При прикосновении к микросхеме этот разряд может пробить тончайшие транзисторы.
- **Защита:** перед работой прикоснитесь к заземлённому металлическому корпусу (батарея отопления, корпус компьютера). Можно использовать антистатический браслет.

4. Уровень «Исследовательский»: Энергопотребление и автономность

(Новое применение, прорыв за границы)

Для тех, кто хочет знать, как долго робот проживёт на одном заряде.

4.1. Расчёт энергопотребления

Дано:

- Аккумулятор: 12В, ёмкость 5000 мА·ч (миллиампер-часов).
- Энергия аккумулятора: `E = 12 В × 5 А·ч = 60 Вт·ч`.

Потребители (примерные значения):

Компонент	Мощность в покое	Мощность под нагрузкой
4 мотора	2 Вт	20 Вт (движение)
Raspberry Pi	5 Вт	8 Вт (пиковая)
Лидар	2 Вт	3 Вт
Камера	1 Вт	2 Вт
Hailo-8	2 Вт	5 Вт (инференс)
ИТОГО	**12 Вт**	**38 Вт**

Время автономной работы:

- В покое (простой): `60 Вт·ч / 12 Вт ≈ 5 часов`.
- При активном движении с распознаванием: `60 Вт·ч / 38 Вт ≈ 1.6 часа`.

4.2. Мини-исследование: кто главный потребитель?

Вопрос: Что потребляет больше всего энергии на роботе?

Посмотрите на таблицу выше. Ответ: **моторы**. При движении они могут забирать до 20 Вт — больше половины всего бюджета.

Инженерный вывод:

1. **Плавный стиль вождения экономит заряд.** Резкие разгоны и торможения — пиковые токи.

2. **Вес робота критичен.** Каждый лишний грамм — дополнительные ватты на движение.
3. **NPU экономит энергию.** Если бы мы гоняли нейросеть на CPU, он бы потреблял ещё на 3–5 Вт больше.

Задание: Если вы хотите увеличить время работы робота вдвое, что выгоднее: увеличить ёмкость аккумулятора (дорого и тяжело) или оптимизировать алгоритмы, чтобы реже включать моторы?

5. Выводы по занятию

1. **Сборка — это инженерная задача.** Правильная установка колёс, надёжное крепление, аккуратная коммутация — залог долгой и безотказной работы.
2. **Интерфейсы выбраны не случайно.** USB для больших данных (лидар, камера), UART для быстрых команд (STM32).
3. **Электропитание — кровеносная система робота.** Преобразователи напряжения, полярность, защита от статики — этому нужно уделять максимум внимания.
4. **Моторы — главные потребители энергии.** Оптимизация движения напрямую влияет на время автономной работы.

Связь со следующим занятием: Робот собран и запитан. Теперь самый волнующий момент — **тестовый запуск и диагностика**. Мы проверим, что каждый компонент работает, увидим данные с датчиков и заставим робота «дёрнуться» по команде.

Домашнее задание:

- **Базовое**: Запишите в тетрадь порядок сборки робота. Какие интерфейсы используются для подключения лидара и STM32?
- **Продвинутое**: Измерьте фактическое напряжение на аккумуляторе мультиметром (под контролем преподавателя). Сравните с номинальным.
- **Исследовательское**: Рассчитайте, как изменится время автономной работы, если заменить моторы на более эффективные (КПД 60% → 80%).

Занятие 16: Тестовый запуск и первичная диагностика

Содержание занятия: Проверка работоспособности всех систем робота

Цели занятия

- **Образовательная**: Научиться выполнять первичную диагностику робота: проверять питание, соединение, работу датчиков и контроллера.
- **Развивающая**: Освоить методику поиска неисправностей: от простого к сложному, изоляция проблемы.
- **Воспитательная**: Сформировать привычку систематической проверки оборудования перед началом работы.

План занятия (90 минут)

1. **Организационный момент и проверка сборки (5 мин)**
2. **Уровень «База»: Проверка питания и первые данные (40 мин)**
    • Визуальная проверка индикаторов.
    • Подключение по SSH и проверка загрузки ОС.
    • Запуск стека ROS2 и проверка топиков.
    • Просмотр данных лидара и тестовый снимок с камеры.
3. **Уровень «Углублённый»: Проверка движения и Hailo-8 (25 мин)**
    • Тестовая команда движения (Twist в `/cmd_vel`).
    • Просмотр сырых данных лидара (`ranges`).
    • Проверка обнаружения Hailo-8 через `lsusb`.
    • Логирование ошибок.
4. **Уровень «Исследовательский»: Скрипт диагностики (15 мин)**
    • Написание Python-скрипта для автоматической проверки компонентов.
    • Алгоритм поиска неисправности.
5. **Подведение итогов (5 мин)**

1. Введение: Момент истины

Робот собран, провода подключены. Настал самый волнительный момент — **первый запуск**. Сегодня мы не будем программировать сложное поведение. Наша задача — убедиться, что **каждый компонент работает** и робот готов к дальнейшим экспериментам.

Диагностика — это навык системного инженера. Мы будем проверять компоненты по порядку: от питания к датчикам, от датчиков к движению.

2. Уровень «База»: Проверка питания и первые данные

(Инструмент, навык, интерфейс)

Обязательный уровень. Учимся проверять работоспособность базовых систем.

2.1. Визуальная проверка индикаторов

Не подключая аккумулятор, ещё раз осмотрите все соединения. Убедитесь, что:

• Провода не пережаты.
• Разъёмы вставлены до конца.
• Нет оголённых контактов, которые могут замкнуть.

Подключите аккумулятор. Наблюдайте за индикаторами:

Компонент	Ожидаемый индикатор	Если не горит
Raspberry Pi	Красный (PWR) горит постоянно	Нет питания 5В. Проверьте преобразователь.
Raspberry Pi	Зелёный (ACT) мигает	Нормально (активность). Если не мигает и не горит — не загрузилась SD-карта.
Лидар	шуршит	Проверьте USB-кабель и питание.

Задание: Зафиксируйте в тетради состояние всех индикаторов.

2.2. Подключение по SSH и проверка загрузки ОС

После включения подождите 30–60 секунд, чтобы Raspberry Pi загрузился.

Подключитесь по SSH (как на занятии 1):

```
ssh <ваш_логин>@<ip_робота>
```

Если подключение успешно — ОС загружена. Проверим «здоровье» системы:

```
# Проверка времени работы (uptime)
uptime

# Проверка свободного места на диске
df -h /

# Проверка температуры процессора
vcgencmd measure_temp
```

Запишите вывод этих команд. Если температура выше 80°C — проверьте охлаждение.

2.3. Запуск стека ROS2 и проверка топиков

Учитель запускает основной стек ROS2 (на классной доске через проектор):

```
ros2 launch mecanum_bringup bridge_ekf_lidar.launch.py start_autopilot:=fa
lse
```

Если работа с симулятором, то он запускается командой:

```
ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml
start_rviz:=true use_lite:=true.
```

Ученики (на своих терминалах SSH) проверяют топики:

Проверка, что топики ROS2 публикуют сообщения:

```
ros2 topic echo /odom
```

Если сообщений нет необходимо выполнить пункт 2.0 Занятия 13.

```
# Посмотреть все активные топики
ros2 topic list
```

```
# Проверить, что ключевые топики присутствуют
ros2 topic list | grep -E "scan|image_raw|cmd_vel|odom"
```

Вы должны увидеть среди прочих: `/scan`, `/camera/image_raw`, `/cmd_vel`, `/odom`. Если какого-то нет — сообщите учителю.

2.4. Просмотр данных лидара и тестовый снимок с камеры

Данные лидара (один снимок):

```
ros2 topic echo /scan --field ranges --once
```

Вы увидите массив чисел — расстояния в метрах. Если массив пустой или все значения `0.0` — лидар что-то не видит (закрыт, не вращается).

Тестовый снимок с камеры (одним кадром, если утилита установлена):

```
ros2 run image_tools cam2image --ros-args -p show_camera:=true
# Или просто проверьте, что топик /camera/image_raw существует и публикует сообщения:
ros2 topic hz /camera/image_raw
```

Вы должны увидеть частоту ~30 Гц.

3. Уровень «Углублённый»: Проверка движения и Hailo-8

(Что под капотом)

Углублённая диагностика: заставляем робота двигаться и проверяем нейропроцессор.

3.1. Тестовая команда движения: «робот, дёрнись!»

Внимание! Робот должен стоять на полу, колёса не должны быть заблокированы. Убедитесь, что вокруг достаточно места.

Отправим тестовый `Twist` — команду на короткое движение вперёд:

```
ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```

Ожидаемый результат: Робот должен начать движение вперед, чтобы остановить робота надо дать ему команду:

```
ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```

Если робот не двинулся:

1. Проверьте, что стек ROS2 запущен (топик `/cmd_vel` существует).
2. Проверьте связь с STM32 (обычно через топик `/x10/motor_control` или диагностику).
3. Проверьте, подано ли питание на моторы (отдельный выключатель?).

3.2. Просмотр сырых данных лидара

В ROS2 сообщение `LaserScan` содержит не только `ranges`, но и метаданные:

```
ros2 topic echo /scan --once
```

Найдите в выводе:

- `angle_min` и `angle_max` — границы сканирования (обычно -π до π, т.е. 360°).
- `angle_increment` — шаг между измерениями в радианах.
- `range_min` и `range_max` — минимальная и максимальная измеряемая дальность.
- `ranges` — массив расстояний.

Задание: Посчитайте, сколько точек в одном скане: `(angle_max - angle_min) / angle_increment`.

3.3. Проверка Hailo-8

Проверим, определяется ли нейропроцессор системой:

```
lspci | grep -i hailo
```

Если устройство отображается (например, `ID 03e7:2485`), значит, физически оно подключено.

Проверим, загружен ли драйвер:

```
lsmod | grep hailo
```

Если Hailo-8 не обнаружен, но физически подключён — проблема в драйверах. Это задача преподавателя.

3.4. Логирование ошибок

Приучитесь всегда проверять диагностический топик:

```
ros2 topic echo /diagnostics --once
```

Сообщения в этом топике содержат статус (`OK`, `WARN`, `ERROR`) для каждого компонента. Если что-то не в порядке — это первый источник информации.

4. Уровень «Исследовательский»: Скрипт диагностики

(Новое применение, прорыв за границы)

Автоматизируем проверку и учимся алгоритмическому поиску неисправностей.

4.1. Python-скрипт автоматической диагностики

Напишем скрипт, который проверяет всё за нас. Создайте `diagnostic_check.py`:

```
nano ~/ws/rbt/scripts/diagnostic_check.py
```

```python
#!/usr/bin/env python3
"""Скрипт первичной диагностики робота."""
import subprocess
import sys

def check(name, command, success_word):
    """Выполняет команду и проверяет, есть ли success_word в выводе."""
    try:
        result = subprocess.run(command, shell=True,
                                capture_output=True, text=True,
timeout=10)
        output = result.stdout + result.stderr
        if success_word in output:
            print(f"  [{name}] OK")
            return True
        else:
            print(f"  [{name}] FAIL — {success_word} не найден")
            return False
    except Exception as e:
        print(f"  [{name}] ERROR — {e}")
        return False

print("=" * 50)
print("ДИАГНОСТИКА РОБОТА")
print("=" * 50)

results = []

# 1. Проверка ROS2 (топики)
print("\n1. Проверка ROS2-топиков:")
results.append(check("Лидар /scan", "ros2 topic list", "/scan"))
results.append(check("Камера /camera/image_raw", "ros2 topic list", "/
camera/image_raw"))
results.append(check("Управление /cmd_vel", "ros2 topic list", "/cmd_vel")
)
results.append(check("Одометрия /odom", "ros2 topic list", "/odom"))

# 2. Проверка Hailo-8
print("\n2. Проверка Hailo-8:")
results.append(check("Hailo-8 (lspci)", "lspci", "Hailo"))

# 3. Проверка системы
print("\n3. Проверка системы:")
# Проверим, что Python доступен
results.append(check("Python3", "python3 --version", "Python"))

# Итог
print("\n" + "=" * 50)
passed = sum(results)
total = len(results)
print(f"ИТОГО: {passed}/{total} проверок пройдено.")
if passed == total:
    print("Робот к работе ГОТОВ!")
else:
```

```
    print("Обнаружены проблемы. Проверьте FAIL-компоненты.")
print("=" * 50)
```

Запустите скрипт:

```
python3 ~/ws/rbt/scripts/diagnostic_check.py
```

4.2. Алгоритм поиска неисправности

Методика «от простого к сложному»:

Если компонент не работает, задайте вопросы по порядку:

```
1. Питание: горит ли индикатор? Есть ли напряжение?
   └─ Нет → Проверить аккумулятор, преобразователь, кабель.
   └─ Да → Идём дальше.

2. Подключение: вставлен ли разъём до конца? Тот ли порт?
   └─ Нет → Переподключить.
   └─ Да → Идём дальше.

3. Драйвер: определяется ли устройство системой (lsusb, lspci, dmesg)?
   └─ Нет → Проблема с драйвером или железом.
   └─ Да → Идём дальше.

4. ROS2: есть ли топик? Публикуются ли данные?
   └─ Нет → Проверить launch-файл, перезапустить стек.
   └─ Да → Компонент работает!
```

Задание: Примените этот алгоритм к своему роботу. Для каждого компонента запишите результат каждого шага.

5. Выводы по занятию и блоку «Аппаратная платформа»

Итоги занятия

1. **Диагностика — обязательный ритуал.** Перед каждой работой с роботом проверяйте индикаторы, SSH и ключевые топики.
2. **ROS2 даёт прозрачность.** Все данные с датчиков доступны через топики, их можно просмотреть «глазами».
3. **Тестовая команда подтверждает готовность.** Если робот «дёрнулся» по `/cmd_vel` — канал управления работает.
4. **Автоматизация диагностики экономит время.** Скрипт за секунды проверяет то, что вручную заняло бы 10 минут.

Итоги блока «Аппаратная платформа робота» (занятия 12–16)

Мы прошли путь от незнакомого набора деталей до проверенного, работающего робота:

- **Занятие 12:** Архитектура: компьютер, STM32, меканум-колёса.

- **Занятие 13:** Сенсоры: лидар (ToF) и камера (матрица).
- **Занятие 14:** Нейропроцессор Hailo-8 — специалист по ИИ.
- **Занятие 15:** Физическая сборка и коммутация.
- **Занятие 16:** Тестовый запуск и диагностика.

Теперь мы знаем робота «в лицо». Со следующего занятия начинается новый большой блок — **«Основы ROS2»**, где мы научимся программировать поведение робота через узлы, топики и сообщения.

Домашнее задание:

- **Базовое**: Запишите в тетрадь пять топиков ROS2, которые вы проверили сегодня.
- **Продвинутое**: Модифицируйте скрипт `diagnostic_check.py`, добавив проверку частоты публикации топика `/scan` (через `ros2 topic hz`).
- **Исследовательское**: Изучите вывод `dmesg | grep -i error` на роботе. Есть ли там ошибки, связанные с USB-устройствами? О чём они говорят?

Основы ROS2: базовые понятия и инструменты

Занятие 17: Архитектура ROS2, инструменты командной строки, teleop

Содержание занятия: Введение в Robot Operating System 2. Узлы, топики, сообщения

Цели занятия

- **Образовательная**: Понять базовую архитектуру ROS2: узлы, топики, сообщения. Освоить основные инструменты командной строки и научиться управлять роботом с клавиатуры.
- **Развивающая**: Сформировать кибернетическое представление о роботе как о сети взаимодействующих узлов.
- **Воспитательная**: Показать мощь модульной архитектуры: каждый компонент — отдельный узел, их можно перезапускать независимо.

План занятия (90 минут)

1. **Организационный момент и введение в ROS2 (10 мин)**
2. **Уровень «База»: Знакомство с ROS2 и teleop (40 мин)**
   - Что такое ROS2: узлы, топики, сообщения.
   - Переменные окружения и `source setup.bash`.
   - Запуск teleop_twist_keyboard — ручное управление роботом.
   - Инструменты: `ros2 node list`, `ros2 topic list`, `ros2 topic echo`.
3. **Уровень «Углублённый»: Архитектура DDS и рабочее пространство (20 мин)**
   - DDS — децентрализованное обнаружение узлов.
   - Отличие ROS2 от ROS1 (нет главного сервера).
   - Рабочее пространство и структура папок.
4. **Уровень «Исследовательский»: Сравнение и замер задержки (15 мин)**
   - ROS2 vs MQTT vs gRPC.
   - Замер задержки передачи сообщения.
5. **Подведение итогов (5 мин)**

1. Введение: Операционная система для роботов

Мы уже знаем, что робот состоит из множества компонентов: компьютер, контроллер, лидар, камера, NPU. Как заставить их работать вместе, обмениваться данными и не конфликтовать?

Ответ — **ROS2 (Robot Operating System 2)**. Это не совсем операционная система в привычном смысле. Это **фреймворк для создания робототехнического ПО**. Он предоставляет:

- Способ разбить программу на независимые **узлы**.
- Механизм обмена **сообщениями** между узлами через **топики**.
- Инструменты для отладки, визуализации и записи данных.

Сегодня мы начнём осваивать ROS2 с самых основ.

2. Уровень «База»: Знакомство с ROS2 и teleop

(Инструмент, навык, интерфейс)

Обязательный уровень. Узнаём базовые понятия и управляем роботом с клавиатуры.

2.1. Что такое ROS2: узлы, топики, сообщения

Представьте себе редакцию газеты:

- **Узлы (Nodes)** — это журналисты, редакторы, верстальщики. Каждый занят своим делом.
- **Топики (Topics)** — это доска объявлений. Журналист вешает статью на доску (публикует). Все, кому интересно, читают (подписываются).
- **Сообщения (Messages)** — это формат статьи: заголовок, текст, фото.

В роботе:

- Узел лидара публикует облако точек в топик `/scan`.
- Узел навигации подписывается на `/scan`, чтобы знать, где препятствия.
- Узел управления подписывается на `/cmd_vel` и отправляет команды моторам.

Ключевое преимущество: узлы можно запускать, останавливать и заменять по отдельности. Обновили алгоритм навигации — перезапустили только узел навигации, а лидар продолжает работать.

2.2. Переменные окружения

Проверка, что топики ROS2 публикуют сообщения:

```
ros2 topic echo /odom
```

Если сообщений нет необходимо выполнить пункт 2.3 Занятия 16.

2.3. Запуск teleop_twist_keyboard — ручное управление

Теперь мы запустим узел телеоператора, чтобы управлять роботом с клавиатуры:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

На экране появится инструкция:

```
   Управление:
     ↑ : вперёд
     ↓ : назад
     ← : поворот влево
     → : поворот вправо
     q : выход
     (другие клавиши для изменения скорости)
```

Осторожно! Робот должен стоять на полу, вокруг достаточно места. Попробуйте аккуратно подвигать робота.

2.4. Инструменты командной строки

Пока робот движется (или стоит), откройте **второй терминал** по SSH. Посмотрим, что происходит внутри ROS2.

```
# Какие узлы запущены?
ros2 node list
# Вы увидите: /teleop_twist_keyboard, /bridge_node, /lidar_driver, ...

# Какие топики активны?
ros2 topic list
# Вы увидите: /cmd_vel, /scan, /odom, /camera/image_raw, ...

# Что публикуется в /cmd_vel (команды движения)?
ros2 topic echo /cmd_vel
# Нажимайте стрелки в окне teleop — здесь будут появляться сообщения!
```

Сообщение `Twist` выглядит примерно так:

```
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Это означает: «ехать вперёд со скоростью 0.5 м/с, не поворачивать».

3. Уровень «Углублённый»: Архитектура DDS и рабочее пространство

(Что под капотом)

Разбираемся, как узлы находят друг друга, и изучаем структуру ROS2-проекта.

3.1. DDS — децентрализованное обнаружение

В первой версии (ROS1) был **rosmaster** — центральный сервер, который знал обо всех узлах. Если он падал — система теряла связь.

ROS2 использует протокол **DDS (Data Distribution Service)**. В нём **нет главного сервера**. Узлы сами находят друг друга по сети с помощью механизма «обнаружения» (discovery).

Аналогия: ROS1 — это звонки через оператора (коммутатор). ROS2 — это рации: настроились на одну частоту (топик) и общаемся напрямую.

Преимущества DDS:

- Нет единой точки отказа.
- Настоящая распределённость: узлы на разных компьютерах общаются так же, как на одном.
- Гибкие настройки **Quality of Service (QoS)**: надёжность, приоритет, время жизни сообщений.

3.2. Рабочее пространство (workspace) и структура папок

Все ROS2-пакеты живут в **рабочем пространстве**. На оригинальном роботе оно находится в `/home/pi/moebius_ws`. У каждого ученика — своё в `~/ws/rbt`.

Структура типичного рабочего пространства:

```
~/ws/rbt/
├── src/                    # Исходный код пакетов
│   └── my_package/         # Ваш пакет
│       ├── package.xml     # Метаданные (имя, зависимости)
│       ├── setup.py        # Скрипт сборки
│       └── my_package/     # Код Python
│           └── __init__.py
├── build/                  # Промежуточные файлы сборки
├── install/                # Установленные файлы
└── log/                    # Логи сборки
```

4. Уровень «Исследовательский»: Сравнение и замер задержки

(Новое применение, прорыв за границы)

Сравниваем ROS2 с альтернативами и измеряем скорость.

4.1. ROS2 vs MQTT vs gRPC

ROS2 — не единственный способ обмена сообщениями. Вот краткое сравнение:

Характеристика	ROS2 (DDS)	MQTT	gRPC
Архитектура	Децентрализованная (P2P)	Центральный брокер	Клиент-сервер
Модель общения	Публикация/подписка	Публикация/подписка	Удалённый вызов процедур
Скорость	Очень высокая (UDP)	Средняя (TCP)	Высокая (HTTP/2)
Где используется	Робототехника, автономные системы	IoT, умный дом	Микросервисы, облака
Сложность настройки	Высокая	Низкая	Средняя

Вывод: ROS2 выбран для роботов, потому что он обеспечивает минимальную задержку и не зависит от центрального сервера.

4.2. Мини-исследование: замер задержки сообщения

Вопрос: Сколько времени проходит от публикации сообщения одним узлом до его получения другим узлом на том же компьютере?

Создайте скрипт `latency_test.py`:

```
nano ~/ws/rbt/scripts/latency_test.py
```

```
#!/usr/bin/env python3
"""Замер задержки передачи сообщения в ROS2."""
import rclpy
from rclpy.node import Node
from rclpy.executors import SingleThreadedExecutor
from std_msgs.msg import Header

class PublisherNode(Node):
    def __init__(self):
        super().__init__('latency_publisher')
        self.pub = self.create_publisher(Header, 'latency_test', 10)
        self.timer = self.create_timer(1.0, self.publish_message)

    def publish_message(self):
        msg = Header()
        # Записываем текущее системное время ROS2 в структуру сообщения
        msg.stamp = self.get_clock().now().to_msg()
        self.pub.publish(msg)

class SubscriberNode(Node):
    def __init__(self):
        super().__init__('latency_subscriber')
        self.sub = self.create_subscription(Header, 'latency_test', self.c
allback, 10)

    def callback(self, msg):
        # Получаем текущее время как полноценный объект Time
        now = self.get_clock().now()

        # Конвертируем штамп из сообщения обратно в объект Time
        sent_time = rclpy.time.Time.from_msg(msg.stamp)

        # Вычитаем объекты напрямую! На выходе получаем объект Duration
        duration = now - sent_time

        # Переводим наносекунды в миллисекунды.
        # Целочисленное деление исключает плавающую погрешность больших
float-чисел
        latency_ms = duration.nanoseconds / 1e6

        self.get_logger().info(f'Задержка: {latency_ms:.4f} мс')

def main():
```

```
    rclpy.init()

    pub_node = PublisherNode()
    sub_node = SubscriberNode()

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and pub_node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(pub_node, timeout_sec=0.02)

    # SingleThreadedExecutor последовательно обработает паблиш и колбэк,
    # что даст максимально стабильный результат для одного процесса
    executor = SingleThreadedExecutor()
    executor.add_node(pub_node)
    executor.add_node(sub_node)

    try:
        executor.spin()
    except KeyboardInterrupt:
        pass
    finally:
        executor.shutdown()
        pub_node.destroy_node()
        sub_node.destroy_node()
        rclpy.shutdown()

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

Запустите скрипт:

```
python3 ~/ws/rbt/scripts/latency_test.py

# В случае использования симулятора команда запуска скрипта
python3 ~/ws/rbt/scripts/latency_test.py --ros-args -p use_sim_time:=true
```

Ожидаемый результат: Задержка на локальной машине — порядка 0.1–1 мс. Это очень быстро! Именно поэтому ROS2 подходит для управления в реальном времени.

5. Выводы по занятию

1. **ROS2 — модульный фреймворк.** Узлы обмениваются сообщениями через топики. Каждый узел — независимая программа.
2. **DDS — децентрализованная основа.** Нет главного сервера, узлы находят друг друга автоматически.
3. **teleop_twist_keyboard — простой способ управления.** Публикует `Twist` в `/cmd_vel` по нажатию клавиш.
4. **Инструменты командной строки — окно в систему.** `ros2 node list`, `ros2 topic list`, `ros2 topic echo` позволяют «видеть» обмен данными.
5. **Задержка ROS2 — доли миллисекунды.** Это позволяет роботу реагировать на препятствия практически мгновенно.

Связь со следующим занятием: Мы управляли роботом вручную. Но настоящая робототехника — это автономность. На следующем занятии мы напишем свой первый **узел-издатель**, который будет отправлять команды движения без участия человека.

Домашнее задание:

- **Базовое**: Запишите в тетрадь определения: узел, топик, сообщение.
- **Продвинутое**: Подключитесь к роботу и изучите вывод `ros2 topic info /cmd_vel`. Сколько узлов подписано? Сколько публикуют?
- **Исследовательское**: Запустите `latency_test.py` (с разрешения учителя) и запишите среднюю задержку. Как она меняется при добавлении ещё 3–4 узлов-подписчиков?

Занятие 18: Издатель, пакет, colcon, первый Twist

Содержание занятия: Создание ROS2-узла, публикующего команды движения

Цели занятия

- **Образовательная**: Научиться создавать простейший Python-скрипт — узел-издатель, отправляющий сообщения `Twist` в топик `/cmd_vel`.
- **Развивающая**: Понять структуру ROS2-пакета и процесс сборки, даже если мы работаем с отдельными скриптами.
- **Воспитательная**: Сделать первый шаг от ручного управления к автономному: робот выполняет программу.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Первый издатель Twist (40 мин)**
   - Что такое издатель (Publisher).
   - Написание скрипта: `move_forward.py`.
   - Запуск и наблюдение через `ros2 topic echo` и `ros2 topic hz`.
3. **Уровень «Углублённый»: Как устроен пакет и QoS (25 мин)**
   - Структура пакета: `package.xml`, `setup.py`, `entry_points`.
   - `colcon build` — процесс сборки.
   - Quality of Service (QoS) — надёжность и очередь.
4. **Уровень «Исследовательский»: Плавный разгон и частота (15 мин)**
   - Скрипт с плавным изменением скорости (ramp).
   - Мини-исследование: частота публикации и плавность движения.
5. **Подведение итогов (5 мин)**

1. Введение: От ручного управления — к программному

На прошлом занятии мы управляли роботом с клавиатуры через `teleop_twist_keyboard`. Но настоящая робототехника — это **автономность**. Робот должен выполнять команды без участия человека.

Для этого мы напишем **узел-издатель (Publisher)** — программу, которая сама отправляет сообщения `Twist` в топик `/cmd_vel`. Сегодня робот впервые «оживёт» под управлением нашего кода.

2. Уровень «База»: Первый издатель Twist

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем простой скрипт и запускаем робота.

2.1. Что такое издатель (Publisher)

В ROS2:

- **Издатель** — узел, который создаёт сообщения и отправляет их в топик.
- **Подписчик** — узел, который принимает сообщения из топика.

Один топик может иметь много издателей и много подписчиков. Они не знают друг о друге — только о топике.

Наша задача: создать узел, который публикует `Twist` в `/cmd_vel`. Этот топик уже «слушает» драйвер STM32, который управляет моторами.

2.2. Скрипт move_forward.py

Создайте файл скрипта:

```
nano ~/ws/rbt/scripts/move_forward.py
```

```
#!/usr/bin/env python3
"""Узел: робот едет вперёд заданное время и гарантированно
останавливается."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class MoveForward(Node):
    def __init__(self):
        super().__init__('move_forward')
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)
        self.get_logger().info('Узел move_forward запущен!')

    def move(self, speed, duration_sec):
        """Движение вперёд с заданной скоростью в течение duration_sec
секунд."""

        # 1. Ожидание подключения подписчика с отправкой нулевых скоростей
```

```
        self.get_logger().info('Ожидание подключения драйвера моторов ... ')
        stop_msg = Twist() # Все поля равны 0.0 по умолчанию

        while rclpy.ok() and (self.publisher.get_subscription_count() ==
0):
            self.publisher.publish(stop_msg)  # Шлем 0, пока драйвер не
увидит узел
            rclpy.spin_once(self, timeout_sec=0.05)

        self.get_logger().info('Драйвер подключен! Начинаю движение.')

        # 2. Подготовка сообщения движения
        msg = Twist()
        msg.linear.x = speed

        # Засекаем время СТРОГО после того, как связь установилась
        start_time = self.get_clock().now()
        duration_ns = duration_sec * 1e9  # Переводим секунды в
наносекунды

        self.get_logger().info(f'Еду вперёд со скоростью {speed} м/с в
течение {duration_sec} с ... ')

        # Основной цикл движения
        while rclpy.ok():
            current_time = self.get_clock().now()
            elapsed_ns = (current_time - start_time).nanoseconds

            if elapsed_ns >= duration_ns:
                break

            self.publisher.publish(msg)
            # Оживляем узел на 100 мс (частота публикации 10 Гц)
            rclpy.spin_once(self, timeout_sec=0.1)

        # 3. Гарантированная остановка серией сообщений
        self.stop_robot()

    def stop_robot(self):
        """Серия публикаций нулевой скорости для гарантированной
остановки."""
        self.get_logger().info('Остановка робота (безопасный цикл) ... ')
        stop_msg = Twist()  # по умолчанию все скорости равны 0.0

        for _ in range(5):
            self.publisher.publish(stop_msg)
            rclpy.spin_once(self, timeout_sec=0.05)

        self.get_logger().info('Робот успешно остановлен.')

def main():
    rclpy.init()
    node = MoveForward()

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
```

```
        rclpy.spin_once(node, timeout_sec=0.02)

    try:
        # Едем вперёд со скоростью 0.2 м/с 2 секунды
        node.move(speed=0.2, duration_sec=2.0)
    except KeyboardInterrupt:
        # Экстренная остановка при Ctrl+C в терминале
        # ИСПРАВЛЕНО: .warning() вместо падающего .warn()
        node.get_logger().warning('Прерывание оператором! Останавливаю
моторы ... ')
        node.move(speed=0.0, duration_sec=0.1)
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

Важно: Топик `/cmd_vel` — «скользящий»: если в него перестали публиковать, робот остановится через таймаут (обычно 0.5–1 с, но это не точно - таймаут может быть ооочень большим). Поэтому мы публикуем в цикле, а не один раз.

2.3. Запуск и наблюдение

Проверка, что топики ROS2 публикуют сообщения:

```
ros2 topic echo /odom
```

Если сообщений нет необходимо выполнить пункт 2.3 Занятия 16.

Такую проверку следует проводить регулярно.

Запустите скрипт:

```
python3 ~/ws/rbt/scripts/move_forward.py

# В случае использования симулятора команда запуска скрипта ()
python3 ~/ws/rbt/scripts/move_forward.py --ros-args -p use_sim_time:=true
```

Во всех дальнейших ситуациях использования симулятора при запуске скриптов используйте аргумент командной строки `--ros-args -p use_sim_time:=true`

Робот должен проехать вперёд около 40 см (0.2 м/с × 2 с) и остановиться.

Наблюдение за топиком (в другом терминале):

```
# Сколько сообщений в секунду?
ros2 topic hz /cmd_vel

# Информация о топике
ros2 topic info /cmd_vel
```

3. Уровень «Углублённый»: Как устроен пакет и QoS

(Что под капотом)

Понимаем структуру пакета и настройки качества обслуживания.

3.1. Структура ROS2-пакета (обзорно на оригинальном роботе)

Хотя мы пишем отдельные скрипты, полезно понимать, как устроен «настоящий» ROS2-пакет. Заглянем в пакет `mecanum_bringup`:

```
ls /home/pi/moebius_ws/src/mecanum_bringup/
```

Типичная структура:

```
mecanum_bringup/
├── package.xml          # Метаданные: имя, версия, зависимости
├── setup.py             # Скрипт сборки (entry_points — где лежат узлы)
├── launch/              # Launch-файлы для запуска комплекса узлов
│   └── bridge_ekf_lidar.launch.py
└── mecanum_bringup/     # Код Python
    ├── __init__.py
    └── bridge_node.py
```

package.xml (фрагмент):

```
<package format="3">
  <name>mecanum_bringup</name>
  <version>0.1.0</version>
  <depend>rclpy</depend>
  <depend>geometry_msgs</depend>
</package>
```

setup.py (фрагмент с entry_points):

```
entry_points={
    'console_scripts': [
        'bridge_node = mecanum_bringup.bridge_node:main',
    ],
},
```

Это связывает команду `bridge_node` с функцией `main` в файле `bridge_node.py`.

3.2. colcon build — как собирается пакет

colcon — инструмент сборки ROS2-пакетов. Учитель запускает его так:

```
cd /home/pi/moebius_ws
colcon build
```

Что происходит:

1. colcon читает `package.xml` — узнаёт зависимости.
2. Запускает `setup.py` — копирует файлы и создаёт скрипты запуска.

3. Результат кладёт в папки `build/` и `install/`.
4. После сборки нужно выполнить `source install/setup.bash`, чтобы система узнала о новом пакете.

Наши скрипты мы не собираем через colcon — запускаем напрямую через `python3`.

3.3. Quality of Service (QoS)

При создании издателя мы указали `10` — это **глубина очереди** (queue size). Это часть настроек QoS.

Основные параметры QoS:

- **Reliability** — надёжность:
  - `RELIABLE` — сообщение будет доставлено обязательно (с переповторами).
  - `BEST_EFFORT` — «я постарался». Если потерялось — ничего страшного.
- **Durability** — сохранность:
  - `VOLATILE` — сообщение живёт только в момент отправки.
  - `TRANSIENT_LOCAL` — новые подписчики получат последнее сообщение.

Для `/cmd_vel` обычно используют `RELIABLE`, чтобы команда точно дошла. Для `/scan` — `BEST_EFFORT`, так как важна скорость, а не потеря одного скана.

4. Уровень «Исследовательский»: Плавный разгон и частота

(Новое применение, прорыв за границы)

Улучшаем движение и исследуем влияние частоты.

4.1. Издатель с плавным изменением скорости (ramp)

Резкий старт вреден для механики и вызывает проскальзывание. Сделаем плавный разгон.

Создайте `move_smooth.py`:

```
nano ~/ws/rbt/scripts/move_smooth.py
```

```
#!/usr/bin/env python3
"""Узел: плавный разгон и торможение по системному времени ROS 2."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class MoveSmooth(Node):
    def __init__(self):
        super().__init__('move_smooth')
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)
        self.get_logger().info('Узел move_smooth запущен!')

    def move_with_ramp(self, target_speed, ramp_time_sec, cruise_time_sec)
:
```

```
        """Плавный разгон, движение, плавное торможение на основе ROS 2
часов."""

        # 1. Ожидание подключения подписчика
        self.get_logger().info('Ожидание подключения драйвера моторов ... ')
        stop_msg = Twist()
        while rclpy.ok() and (self.publisher.get_subscription_count() ==
0):
            self.publisher.publish(stop_msg)
            rclpy.spin_once(self, timeout_sec=0.05)

        self.get_logger().info('Драйвер подключен! Начинаю профиль
движения.')

        # Переводим параметры времени в наносекунды
        ramp_ns = ramp_time_sec * 1e9
        cruise_ns = cruise_time_sec * 1e9
        total_ns = (ramp_time_sec * 2 + cruise_time_sec) * 1e9

        msg = Twist()
        start_time = self.get_clock().now()

        # Основной цикл профиля скорости
        while rclpy.ok():
            current_time = self.get_clock().now()
            elapsed_ns = (current_time - start_time).nanoseconds

            # Условие выхода — общее время вышло
            if elapsed_ns >= total_ns:
                break

            # ФАЗА 1: Разгон
            if elapsed_ns < ramp_ns:
                ratio = elapsed_ns / ramp_ns
                msg.linear.x = target_speed * ratio

            # ФАЗА 2: Круиз (равномерное движение)
            elif elapsed_ns < (ramp_ns + cruise_ns):
                msg.linear.x = target_speed

            # ФАЗА 3: Торможение
            else:
                # Сколько времени прошло с момента начала торможения
                time_in_braking = elapsed_ns - (ramp_ns + cruise_ns)
                ratio = 1.0 - (time_in_braking / ramp_ns)
                # Ограничиваем снизу нулем на случай микровылетов по
времени
                msg.linear.x = max(0.0, target_speed * ratio)

            self.publisher.publish(msg)
            # Частота обновления профиля ~20 Гц (шаг 50 мс)
            rclpy.spin_once(self, timeout_sec=0.05)

        # 2. Гарантированный финишный сброс в ноль
        self.stop_robot()
```

```
    def stop_robot(self):
        """Серия публикаций нулевой скорости для фиксации остановки."""
        stop_msg = Twist()
        for _ in range(5):
            if not rclpy.ok():
                break
            self.publisher.publish(stop_msg)
            try:
                rclpy.spin_once(self, timeout_sec=0.02)
            except:
                break
        self.get_logger().info('Плавное движение успешно завершено.')

def main():
    rclpy.init()
    node = MoveSmooth()

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(node, timeout_sec=0.02)

    try:
        # Разгон 1с до 0.3 м/с, круиз 1с, торможение 1с (итого 3 секунды
движения)
        node.move_with_ramp(target_speed=0.3, ramp_time_sec=1.0, cruise_ti
me_sec=1.0)
    except KeyboardInterrupt:
        node.get_logger().warning('Прерывание оператором! Мгновенная
остановка ... ')

# При Ctrl+C сбрасываем скорость мгновенно без плавного торможения ради
безопасности
        node.stop_robot()
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

4.2. Мини-исследование: частота публикации и плавность

Вопрос: Как частота публикации `Twist` влияет на плавность движения робота?

Методика:

1. Модифицируйте `move_forward.py`: измените `rclpy.spin_once(self, timeout_sec=0.1)` (10 Гц) на `rclpy.spin_once(self, timeout_sec=0.02)` (50 Гц).
2. Запустите робота с одной и той же скоростью, но разной частотой публикации.
3. Оцените визуально: какое движение более плавное?
4. Проверьте `ros2 topic hz /cmd_vel` для обоих случаев.

Ожидаемый результат:

- При 10 Гц движение может быть слегка «дёрганым».
- При 50 Гц движение более плавное, но процессор нагружен сильнее.
- Оптимум для нашего робота — 20–30 Гц. Выше — рост нагрузки без заметного улучшения плавности.

Почему это важно: В реальных системах управления частота контура управления (control loop) напрямую влияет на качество стабилизации. Слишком низкая — рывки. Слишком высокая — перегруз CPU.

5. Выводы по занятию

1. **Издатель — узел, отправляющий сообщения.** Создаётся через `create_publisher(Twist, '/cmd_vel', 10)`.
2. **Робот движется, пока получает команды.** Топик `/cmd_vel` требует непрерывной публикации. Остановились публиковать — робот остановится.
3. **Структура пакета стандартизирована.** `package.xml` + `setup.py` + код. colcon собирает это в исполняемые скрипты.
4. **QoS управляет надёжностью и сохранностью.** Для команд движения важна надёжность (`RELIABLE`), для сенсоров — скорость (`BEST_EFFORT`).
5. **Плавный разгон (ramp) — забота о механике.** Линейное изменение скорости снижает износ и проскальзывание.

Связь со следующим занятием: `teleop_twist_keyboard` — чужая программа. На следующем занятии мы напишем **свой телеоператор**, который будет захватывать нажатия клавиш и превращать их в команды движения.

Домашнее задание:

- **Базовое**: Измените `move_forward.py` так, чтобы робот ехал назад 1 секунду.
- **Продвинутое**: Измените `move_smooth.py`, добавив параметр `direction` ('forward' или 'backward').
- **Исследовательское**: Проведите замеры плавности движения при 5, 10, 20, 50 Гц. Запишите наблюдения.

Занятие 19: Свой телеоператор: захват клавиш

Содержание занятия: Создание интерактивного узла управления с клавиатуры

Цели занятия

- **Образовательная**: Научиться перехватывать нажатия клавиш в Python без прав root и преобразовывать их в команды `Twist` .

- **Развивающая**: Понять принцип неблокирующего ввода: программа не ждёт нажатия, а опрашивает клавиатуру в цикле.
- **Воспитательная**: Создать свой инструмент, который полностью заменяет стандартный `teleop_twist_keyboard`.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Захват клавиш и Twist (40 мин)**
   - Библиотека `curses` для захвата клавиш.
   - Связь клавиш со значениями `Twist`.
   - Запуск и управление роботом.
3. **Уровень «Углублённый»: Неблокирующий ввод и режимы (25 мин)**
   - Как работает буфер клавиатуры.
   - Обработка Shift для ускорения.
   - Плавное изменение скорости.
4. **Уровень «Исследовательский»: Режимы и задержка (15 мин)**
   - Добавление «точного» и «быстрого» режимов.
   - Сравнение задержки со стандартным teleop.
5. **Подведение итогов (5 мин)**

1. Введение: Сделай сам

На позапрошлом занятии мы управляли роботом через `teleop_twist_keyboard` — готовую программу. Но настоящий разработчик создаёт свои инструменты. Сегодня мы напишем **свой телеоператор** с нуля.

Зачем?

- Чтобы понимать, как работает захват клавиш.
- Чтобы добавить функции, которых нет в стандартном (режимы точности, плавный газ).
- Чтобы гордиться: «Я сам сделал пульт управления роботом!»

2. Уровень «База»: Захват клавиш и Twist

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем свой телеоператор.

2.1. Библиотека curses для захвата клавиш

Обычный `input()` ждёт нажатия Enter — это неудобно для телеоператора. Нужно ловить нажатие **мгновенно**, без Enter.

Библиотека `curses` (встроена в Python) позволяет:

- Перевести терминал в «сырой» режим (raw mode) — клавиши считываются сразу.
- Отключить эхо (нажатые клавиши не печатаются на экране).
- Работать без прав root (в отличие от некоторых других методов).

Простой пример curses:

```
import curses

screen = curses.initscr()       # инициализация
curses.cbreak()                 # не ждать Enter
screen.keypad(True)             # включить спецклавиши (стрелки)

key = screen.getch()            # ждать нажатия
print(f"Нажата клавиша с кодом: {key}")

curses.endwin()                 # восстановить обычный режим терминала
```

2.2. Скрипт телеоператора my_teleop.py

Создайте файл:

```
nano ~/ws/rbt/scripts/my_teleop.py
```

```
#!/usr/bin/env python3
"""Телеоператор управления роботом с фиксацией команд и интеграцией ROS
2."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import curses

class MyTeleop(Node):
    def __init__(self):
        super().__init__('my_teleop')
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)
        self.speed = 0.2        # линейная скорость по умолчанию
        self.turn = 0.5         # угловая скорость по умолчанию
        self.get_logger().info('Мой телеоператор запущен!')

    def publish_twist(self, linear_x, angular_z):
        msg = Twist()
        msg.linear.x = linear_x
        msg.angular.z = angular_z
        self.publisher.publish(msg)

def run_teleop(screen, node):
    """Основной цикл curses, запущенный через безопасный wrapper."""
    curses.cbreak()
    screen.keypad(True)
    screen.nodelay(True)  # неблокирующий режим ввода

    screen.clear()
```

```
    screen.addstr(0, 0, "=== УПРАВЛЕНИЕ РОБОТОМ ===", curses.A_BOLD)
    screen.addstr(2, 0, "Управление: стрелки (фиксация движения)")
    screen.addstr(3, 0, "Пробел = СТОП | Q = Выход")
    screen.refresh()

    running = True
    while running:
        key = screen.getch()

        if key == -1:  # ничего не нажато
            # Сохраняем вашу логику: просто пропускаем и ничего не меняем
            pass
        elif key == curses.KEY_UP:
            node.publish_twist(node.speed, 0.0)
            screen.addstr(5, 0, "Текущая команда: Вперед          ")
        elif key == curses.KEY_DOWN:
            node.publish_twist(-node.speed, 0.0)
            screen.addstr(5, 0, "Текущая команда: Назад           ")
        elif key == curses.KEY_LEFT:
            node.publish_twist(0.0, node.turn)
            screen.addstr(5, 0, "Текущая команда: Поворот Влево   ")
        elif key == curses.KEY_RIGHT:
            node.publish_twist(0.0, -node.turn)
            screen.addstr(5, 0, "Текущая команда: Поворот Вправо  ")
        elif key == ord(' '):  # пробел
            node.publish_twist(0.0, 0.0)
            screen.addstr(5, 0, "Текущая команда: Остановка       ")
        elif key == ord('q') or key == ord('Q'):
            running = False

        # РЕШЕНИЕ: Оживляем ROS 2 на каждом шаге цикла.
        # Шаг в 20 миллисекунд обеспечивает стабильную частоту цикла ~50
Гц.
        rclpy.spin_once(node, timeout_sec=0.02)
        screen.refresh()

def main():
    rclpy.init()
    node = MyTeleop()

    try:
        # curses.wrapper гарантирует, что если код упадет с ошибкой,
        # терминал Linux не сломается и сохранит нормальный вид.
        curses.wrapper(run_teleop, node)
    except KeyboardInterrupt:
        pass
    finally:
        # Гарантированная остановка робота при закрытии программы
        print("\nЗакрытие телеоператора. Остановка моторов...")
        stop_msg = Twist()
        node.publisher.publish(stop_msg)

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

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

Запуск:

```
cd ~/ws/rbt/scripts
python3 my_teleop.py
```

Важно: Для выхода нажмите `q`. Если терминал «испортился» (не печатает буквы), выполните в слепую команду `reset`.

2.3. Тестирование

Проверьте управление:

- Стрелка вверх — робот едет вперёд.
- Стрелка вниз — назад.
- Стрелка влево — поворот влево.
- Стрелка вправо — поворот вправо.
- Пробел — стоп.

Если робот не реагирует, проверьте:

1. Запущен ли основной стек (`ros2 launch ...`).
2. Работает ли ваш скрипт (нет ли ошибок в терминале).
3. Есть ли сообщения в `/cmd_vel` (`ros2 topic echo /cmd_vel`).

3. Уровень «Углублённый»: Неблокирующий ввод и режимы

(Что под капотом)

Понимаем, как работает ввод, и добавляем фичи.

3.1. Как работает неблокирующий ввод

Обычный `getch()` **ждёт**, пока пользователь нажмёт клавишу. Это «блокирующий» режим — программа засыпает.

`screen.nodelay(True)` переводит `getch()` в **неблокирующий** режим:

- Если клавиша нажата — возвращает её код.
- Если ничего не нажато — мгновенно возвращает `-1`.

Это позволяет публиковать `Twist` непрерывно (50 Гц), даже когда клавиши не нажимаются. Иначе робот будет двигаться рывками.

3.2. Обработка Shift для ускорения

Добавим в `my_teleop.py` поддержку Shift для увеличения скорости:

```
# Внутри цикла while, после проверки обычных стрелок:
elif key == curses.KEY_SR:    # Shift+Вверх (может отличаться)
    node.publish_twist(node.speed * 2, 0.0)
elif key == curses.KEY_SF:    # Shift+Вниз
    node.publish_twist(-node.speed * 2, 0.0)
```

Примечание: Коды Shift+стрелка зависят от терминала. Альтернативный подход — использовать буквенные клавиши для режимов.

4. Уровень «Исследовательский»: Режимы и задержка

(Новое применение, прорыв за границы)

Добавляем профессиональные функции и измеряем производительность.

4.1. Режимы: «точный» и «быстрый»

Добавим переключение режимов по клавишам `1` и `2`:

```
# В классе MyTeleop добавить:
self.mode = "normal"
# В цикле while:
elif key == ord('1'):
    node.speed = 0.05
    node.turn = 0.2
    node.mode = "precise"
    screen.addstr(1, 0, f"Режим: ТОЧНЫЙ (speed={node.speed}, turn={node.tu
rn})  ")
elif key == ord('2'):
    node.speed = 0.4
    node.turn = 1.0
    node.mode = "fast"
    screen.addstr(1, 0, f"Режим: БЫСТРЫЙ (speed={node.speed}, turn={node.t
urn})  ")
```

4.2. Мини-исследование: задержка своего и стандартного телеоператоров

Вопрос: Есть ли разница в задержке между нашим телеоператором и `teleop_twist_keyboard`?

Методика:

1. Запустите `teleop_twist_keyboard`. В другом терминале: `ros2 topic echo /cmd_vel`.
2. Нажмите стрелку и засеките время от нажатия до появления сообщения в `echo`.
3. Повторите с `my_teleop.py`.

Ожидаемый результат: Задержка практически одинакова (50–100 мс), так как оба публикуют с частотой ~20–50 Гц. Разница — в удобстве дополнительных функций.

Почему задержка не нулевая?

- Обработка клавиатуры операционной системой (несколько мс).
- Цикл публикации (мы публикуем каждые 20 мс).
- Сетевой стек DDS (доли мс).

5. Выводы по занятию

1. **curses — стандартный способ захвата клавиш.** Работает без root, неблокирующий режим через `nodelay(True)`.
2. **Телеоператор — это издатель Twist.** Та же модель, что и `move_forward.py`, но команды зависят от нажатых клавиш.
3. **Неблокирующий ввод — ключ к плавности.** Публикация продолжается, даже когда клавиши не нажимаются.
4. **Свой инструмент лучше стандартного.** Мы добавили режимы точности, и это только начало.

Связь со следующим занятием: Теперь у нас есть и издатель, и телеоператор. Но чтобы «видеть», что происходит в системе, нужен **подписчик** — узел, который принимает сообщения и выводит их. На следующем занятии мы создадим связку «телеоп → подписчик → лог» и визуализируем граф узлов через `rqt_graph`.

Домашнее задание:

- **Базовое**: Добавьте в `my_teleop.py` обработку клавиш `w`, `a`, `s`, `d` как альтернативу стрелкам.
- **Продвинутое**: Реализуйте «инерцию»: при отпускании клавиши робот плавно замедляется, а не останавливается мгновенно.
- **Исследовательское**: Измерьте задержку своего телеоператора с помощью методики из п. 4.2. Оформите результат.

Занятие 20: Подписчик, связка, rqt_graph, частота

Содержание занятия: Приём сообщений и визуализация графа ROS2-узлов

Цели занятия

- **Образовательная**: Научиться создавать узел-подписчик (Subscriber), который принимает сообщения `Twist` и обрабатывает их в callback-функции.
- **Развивающая**: Освоить инструмент `rqt_graph` для визуализации связей между узлами. Понять влияние частоты публикации на систему.
- **Воспитательная**: Сформировать системное видение: робот — это граф взаимодействующих узлов.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Подписчик Twist и rqt_graph (40 мин)**
   - Что такое подписчик и callback.
   - Скрипт `twist_logger.py`.
   - Связка: телеоп → `/cmd_vel` → робот + подписчик.
   - `rqt_graph` — визуализация.
3. **Уровень «Углублённый»: Частота и пропуски (25 мин)**
   - Частота публикации и её измерение.
   - Пропуск сообщений при медленной обработке.
   - Callback-функции: однопоточность.
4. **Уровень «Исследовательский»: График и перегрузка (15 мин)**
   - Подписчик с графиком скорости в реальном времени.
   - Что будет, если callback длиннее интервала публикации.
5. **Подведение итогов (5 мин)**

1. Введение: Слышать, что говорит робот

До сих пор мы только **отправляли** команды (издатель, телеоператор). Но в реальной системе нужно и **принимать** данные: скорость, одометрию, данные лидара.

Сегодня мы создадим **узел-подписчик** — программу, которая «слушает» топик и реагирует на новые сообщения. Мы построим полную связку: человек → телеоп → `/cmd_vel` → робот + наш подписчик-логгер. И увидим эту связь на графе.

2. Уровень «База»: Подписчик Twist и rqt_graph

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём подписчика и визуализируем граф.

2.1. Что такое подписчик и callback

Подписчик (Subscriber) — узел, который регистрирует свой интерес к топику. Когда в топик приходит новое сообщение, автоматически вызывается **callback-функция**.

```
# Упрощённая схема
def callback(msg):
    print(f"Получена скорость: {msg.linear.x}")

sub = node.create_subscription(Twist, '/cmd_vel', callback, 10)
```

Когда кто-то публикует в `/cmd_vel`, ROS2 вызывает `callback(msg)`.

2.2. Скрипт twist_logger.py

Создайте подписчика, который выводит в терминал всё, что летит в `/cmd_vel`:

```
nano ~/ws/rbt/scripts/twist_logger.py
```

```
#!/usr/bin/env python3
"""Подписчик: эффективно логирует команды движения без перегрузки
процессора."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class TwistLogger(Node):
    def __init__(self):
        super().__init__('twist_logger')
        self.subscription = self.create_subscription(
            Twist,
            '/cmd_vel',
            self.callback,
            10
        )
        self.get_logger().info('Логгер запущен. Слушаю /cmd_vel ... ')
        self.msg_count = 0

        # Запоминаем предыдущие значения, чтобы отслеживать изменения
        self.last_linear_x = None
        self.last_angular_z = None

    def callback(self, msg):
        """Вызывается при каждом новом сообщении в /cmd_vel."""
        self.msg_count += 1

        # Проверяем, изменилась ли скорость по сравнению с прошлым
сообщением
        speed_changed = (msg.linear.x != self.last_linear_x or
                         msg.angular.z != self.last_angular_z)

        if speed_changed:
            # Если скорость изменилась, логируем мгновенно (важное
событие!)
            self.get_logger().info(
                f'[{self.msg_count}] ИЗМЕНЕНИЕ: '
                f'linear.x={msg.linear.x:.2f} м/с, '
                f'angular.z={msg.angular.z:.2f} рад/с'
            )
            # Обновляем историю
            self.last_linear_x = msg.linear.x
            self.last_angular_z = msg.angular.z
        else:
            # Если скорость та же (робот просто продолжает ехать),
            # используем THROTTLE-логирование: пишем в консоль не чаще,
чем раз в 1000 мс (1 секунду)
            self.get_logger().info(
                f'[{self.msg_count}] Удержание: '
                f'linear.x={msg.linear.x:.2f} м/с, '
```

```
                f'angular.z={msg.angular.z:.2f} рад/с',
                throttle_duration_sec=1.0
            )

def main():
    rclpy.init()
    node = TwistLogger()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

Ключевое отличие от издателя: `rclpy.spin(node)` — это бесконечный цикл, который ждёт сообщений и вызывает `callback`.

2.3. Связка: телеоп → /cmd_vel → робот + подписчик

Запустите в разных терминалах:

```
# Терминал 1: телеоператор (ваш my_teleop.py)
python3 ~/ws/rbt/scripts/my_teleop.py

# Терминал 2: подписчик-логгер
python3 ~/ws/rbt/scripts/twist_logger.py
```

Теперь нажимайте стрелки в терминале 1. В терминале 2 будут появляться строки со скоростями. Робот будет двигаться.

Вы построили связку: человек → телеоп → `/cmd_vel` → (робот + логгер). Три узла, один топик.

2.4. rqt_graph — визуализация графа узлов

Учитель запускает `rqt_graph` на компьютере с графическим интерфейсом (проекция на доску):

```
ros2 run rqt_graph rqt_graph
```

Вы увидите схему:

- Овалы — **узлы** (Nodes): `/my_teleop`, `/twist_logger`, `/bridge_node`, `/lidar_driver`, ...
- Прямоугольники — **топики** (Topics): `/cmd_vel`, `/scan`, `/odom`, ...
- Стрелки — **направление передачи сообщений**: от издателя к топику, от топика к подписчику.

Найдите на графе свой узел `/my_teleop` и `/twist_logger`. Убедитесь, что они соединены через `/cmd_vel`.

3. Уровень «Углублённый»: Частота и пропуски

(Что под капотом)

Изучаем частоту публикации и ограничения однопоточной модели.

3.1. Частота публикации и её измерение

```
ros2 topic hz /cmd_vel
```

Вы увидите среднюю частоту за последние секунды. При активном телеоператоре она должна быть ~20–50 Гц.

Почему не ровно 50 Гц? Потому что телеоп публикует в цикле с `time.sleep(0.02)`, но сама публикация тоже занимает время. Реальная частота чуть ниже.

3.2. Пропуск сообщений и однопоточность callback

Важный факт: По умолчанию callback-функции выполняются **в одном потоке**. Если callback обрабатывает сообщение дольше, чем интервал между публикациями, сообщения накапливаются в очереди.

Эксперимент: Добавим искусственную задержку в callback:

```
def callback(self, msg):
    self.msg_count += 1
    time.sleep(0.5)  # притворяемся, что обрабатываем долго
    self.get_logger().info(f'[{self.msg_count}] Обработано')
```

Запустите подписчик с задержкой и телеоператор. Вы увидите, что:

- Логгер отстаёт: выводит не каждое сообщение.
- Сообщения копятся в очереди (можно увидеть по `ros2 topic info /cmd_vel` — растёт `queue size`).

Вывод: Callback должен быть **быстрым**. Тяжёлую обработку — в отдельный поток или другой узел.

4. Уровень «Исследовательский»: График и перегрузка

(Новое применение, прорыв за границы)

Визуализируем данные и изучаем поведение системы под нагрузкой.

4.1. Подписчик, который строит график скорости

Создайте `twist_plotter.py`:

```
nano ~/ws/rbt/scripts/twist_plotter.py
```

```
#!/usr/bin/env python3
"""Подписчик: безопасно выводит ASCII-график скорости в реальном
времени."""
```

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class TwistPlotter(Node):
    def __init__(self):
        super().__init__('twist_plotter')
        self.subscription = self.create_subscription(
            Twist, '/cmd_vel', self.callback, 10
        )
        self.get_logger().info('График скорости запущен.')

        # Константы для визуализации
        self.MAX_BAR_WIDTH = 40  # Максимальная длина шкалы в символах
        self.SCALE = 40          # Масштаб: 40 символов на 1.0 м/с

    def callback(self, msg):
        # Рассчитываем длину шкалы и жестко ограничиваем её (clamp) от 0
до MAX_BAR_WIDTH
        raw_bar_len = int(abs(msg.linear.x) * self.SCALE)
        bar_len = max(0, min(raw_bar_len, self.MAX_BAR_WIDTH))

        # Вычисляем остаток свободного места на шкале
        remaining_space = self.MAX_BAR_WIDTH - bar_len

        if msg.linear.x >= 0:
            # Движение вперед: график растет вправо
            bar = '#' * bar_len + '>' + ' ' * remaining_space
        else:
            # Движение назад: график растет влево
            bar = ' ' * remaining_space + '<' + '#' * bar_len

        # \033[K — это ANSI-код, который очищает строку от курсора до
конца экрана.
        # Это полностью убирает проблему "остаточных хвостов" от прошлых
длинных сообщений.
        # flush=True гарантирует мгновенный вывод на экран без задержек
буфера Linux.
        print(f"\rСкорость: [{bar}] {msg.linear.x:+.2f} м/с\033[K",
end='', flush=True)

def main():
    rclpy.init()
    node = TwistPlotter()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    print("\nГрафик остановлен.")  # Чистый перевод строки при выходе
    node.destroy_node()
    rclpy.shutdown()

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

Запустите вместе с телеоператором — вы увидите «живой» столбик скорости.

4.2. Мини-исследование: перегрузка callback

Вопрос: Что произойдёт, если callback обрабатывает дольше, чем интервал публикации?

Эксперимент:

1. В `twist_logger.py` добавьте `import time` и `time.sleep(0.2)` в начало `callback`.
2. Запустите телеоператор (публикует каждые 20 мс).
3. Запустите модифицированный логгер.
4. Понаблюдайте за `ros2 topic info /cmd_vel` — посмотрите на `queue size`.

Результат:

- Callback не успевает обрабатывать 50 сообщений в секунду (успевает только 5).
- Сообщения накапливаются в очереди.
- Когда очередь переполняется (больше 10), старые сообщения выбрасываются.
- Логгер отстаёт, но не мешает другим узлам — это преимущество ROS2.

Решение для реальных систем:

- Использовать `MultiThreadedExecutor` для параллельных callback.
- Вынести тяжёлую обработку в отдельный узел.
- Увеличить глубину очереди (но это откладывает проблему).

5. Выводы по занятию

1. **Подписчик — приёмник сообщений.** Создаётся через `create_subscription()`. Callback вызывается при каждом новом сообщении.
2. **rqt_graph — рентген системы.** Позволяет увидеть все узлы, топики и связи между ними.
3. **Связка издатель-подписчик — основа ROS2.** Один публикует, многие подписываются. Узлы не знают друг о друге.
4. **Частота публикации влияет на плавность.** Измеряется через `ros2 topic hz`.
5. **Callback должен быть быстрым.** Медленная обработка приводит к накоплению очереди и пропуску сообщений.

Связь со следующим занятием: Мы научились отправлять команды и принимать данные. Но есть проблема: `Twist` задаёт скорость в м/с, а реальная скорость робота может отличаться. Почему? И как это исправить? Об этом — **калибровка прямолинейного движения**.

Домашнее задание:

- **Базовое**: Напишите подписчик, который считает общее количество полученных сообщений `/scan` за 10 секунд.
- **Продвинутое**: Модифицируйте `twist_plotter.py`, чтобы он показывал и линейную, и угловую скорость на одной строке.

- **Исследовательское**: Проведите эксперимент с перегрузкой callback (п. 4.2) и запишите, при каком `time.sleep` очередь начинает переполняться.

Занятие 21: Калибровка прямолинейного движения (выполняется на оригинальном роботе)

Содержание занятия: Экспериментальное определение параметров колёсной базы: ticks_per_rev и cmd_scale_vx

Цели занятия

- **Образовательная**: Понять разницу между калибровкой энкодеров (одометрия) и калибровкой команд движения. Освоить методику последовательной калибровки двух параметров моста.
- **Развивающая**: Научиться планировать двухэтапный эксперимент: сначала добиться точных показаний одометрии, затем — точного исполнения команд.
- **Воспитательная**: Показать важность порядка калибровки: нельзя калибровать исполнение команд, пока нет доверия к измерениям.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База» — Этап 1: Калибровка ticks_per_rev по одометрии (35 мин)**
   - Проблема: одометрия врёт.
   - Эксперимент: ручное движение по телеоп, замер рулеткой, сравнение с `/odom`.
   - Вычисление нового `ticks_per_rev`.
3. **Уровень «База» — Этап 2: Калибровка cmd_scale_vx по времени (30 мин)**
   - Проблема: робот проезжает не то расстояние за заданное время.
   - Эксперимент: скрипт с фиксированной скоростью и длительностью.
   - Вычисление нового `cmd_scale_vx`.
4. **Уровень «Углублённый»: Анализ причин ошибок (10 мин)**
   - Почему оба параметра нуждаются в калибровке.
   - Порядок калибровки: сначала одометрия, потом команды.
5. **Уровень «Исследовательский»: Покрытие и батарея (5 мин — задание на дом)**
   - Калибровка на разных покрытиях.
   - Влияние напряжения батареи.
6. **Подведение итогов (5 мин)**

1. Введение: Два источника ошибок движения

Робот получает команду `Twist` — и едет. Но как узнать, сколько он **на самом деле** проехал? И почему он проезжает не столько, сколько мы хотим?

Есть два независимых источника ошибок:

1. **Одометрия врёт.** Энкодеры считают обороты колёс, но мост неправильно переводит их в метры. Параметр `ticks_per_rev` (тиков энкодера на оборот колеса) может быть неточен.
2. **Команды неточны.** Мы задаём `linear.x = 0.2`, но реальная скорость не 0.2 м/с. Параметр `cmd_scale_vx` (масштабный коэффициент преобразования м/с в команды мотора) нуждается в коррекции.

Правило калибровки: Сначала добиваемся, чтобы **измерения** (одометрия) были верны. Затем — чтобы **исполнение команд** соответствовало заданию.

Сегодня мы проведём оба этапа.

2. Уровень «База» — Этап 1: Калибровка ticks_per_rev

(Инструмент, навык, интерфейс)

Калибруем одометрию: робот должен правильно сообщать, сколько он проехал.

2.1. Проблема: одометрия не совпадает с реальностью

Мост `mecanum_bridge` получает от STM32 количество тиков энкодера. Чтобы перевести тики в метры, он использует формулу:

```
расстояние (м) = (тики / ticks_per_rev) × (2 × π × wheel_radius)
```

Если `ticks_per_rev` (тиков на один оборот колеса) указан неверно, одометрия будет систематически ошибаться. Например, робот проехал 1 метр, а одометрия показывает 0.9 м.

Первый этап калибровки — исправить это.

2.2. Эксперимент: ручное движение и замер

Инструменты: рулетка (3–5 м), мел/малярный скотч.

Порядок действий:

1. Отметьте на полу стартовую линию.
2. Поставьте робота передней частью ровно на линию.
3. Запустите **ваш телеоператор** (`my_teleop.py`).
4. В другом терминале запустите **наблюдение за одометрией**:

   ```
   ros2 topic echo /odom --field pose.pose.position.x --once
   ```

   Запишите начальное значение X (должно быть близко к 0).
5. Аккуратно, на малой скорости, проведите робота **строго вперёд** на расстояние примерно 2 метра (по рулетке).

6. Остановите робота. Запишите конечное значение X из одометрии.

7. Измерьте **реальное** расстояние рулеткой от стартовой линии до передней части робота.

8. Повторите заезд 3 раза (вперёд) и запишите результаты.

2.3. Запись данных и вычисление нового ticks_per_rev

Таблица измерений (пример):

Заезд	Реальное расстояние (рулетка), м	Расстояние по одометрии, м	Соотношение (реал/одом)
1	2.10	1.85	2.10 / 1.85 = 1.135
2	1.95	1.72	1.95 / 1.72 = 1.134
3	2.05	1.80	2.05 / 1.80 = 1.139
Среднее			**1.136**

Вычисление нового `ticks_per_rev`:

```
Если одометрия занижает (показывает меньше, чем реально),
значит ticks_per_rev завышен.

новый_ticks_per_rev = текущий_ticks_per_rev / среднее_соотношение

Пример:
новый_ticks_per_rev = 689.0 / 1.136 ≈ 606.5
```

Почему делим? Одометрия считает: `путь = тики / ticks_per_rev × окружность`. Если `ticks_per_rev` уменьшить, одометрия покажет **больше** метров при тех же тиках.

Важно: Вы передаёте вычисленное значение преподавателю. Он изменит параметр в launch-файле и перезапустит стек.

2.4. Проверка: повторный заезд

После того как преподаватель обновил `ticks_per_rev` и перезапустил стек, проведите ещё один заезд и убедитесь, что одометрия стала точнее (соотношение реал/одом близко к 1.0). Если нет — повторите калибровку.

3. Уровень «База» — Этап 2: Калибровка cmd_scale_vx

(Инструмент, навык, интерфейс)

Теперь, когда одометрия точна, калибруем исполнение команд.

3.1. Проблема: робот не проезжает расчётное расстояние

Мы отправляем `linear.x = 0.2` на 5 секунд. Ожидаем: `0.2 × 5 = 1.0 м`. Но робот проезжает, скажем, 0.85 м. Одометрия теперь исправна и подтверждает: реальная скорость не 0.2, а 0.17 м/с.

Параметр `cmd_scale_vx` преобразует `linear.x` (м/с) во внутренние единицы команды для моторов. Если он неточен, реальная скорость отличается от заданной.

3.2. Скрипт для калибровочных заездов

Создайте `calib_straight.py`:

```
nano ~/ws/rbt/scripts/calib_straight.py
```

```
#!/usr/bin/env python3
"""Скрипт для калибровки cmd_scale_vx с автоматическим замером по
одометрии."""
import rclpy
from rclpy.node import Node
from rclpy.duration import Duration
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import sys
import math

class CalibStraight(Node):
    def __init__(self, speed, duration):
        super().__init__('calib_straight')

        self.speed = speed
        self.duration = duration

        # Стандартные издатель и подписчик. Работают в одном главном
потоке.
        self.pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.odom_sub = self.create_subscription(Odometry, '/odom', self.o
dom_callback, 10)

        # Переменные одометрии
        self.start_x = None
        self.start_y = None
        self.current_x = None
        self.current_y = None

    def odom_callback(self, msg):
        """Запоминаем текущую позицию из одометрии."""
        self.current_x = msg.pose.pose.position.x
        self.current_y = msg.pose.pose.position.y

        if self.start_x is None:
            self.start_x = self.current_x
            self.start_y = self.current_y

    def get_odom_distance(self):
        """Вычисляет пройденное расстояние по одометрии от старта."""
        if self.start_x is None or self.current_x is None:
            return 0.0
        dx = self.current_x - self.start_x
        dy = self.current_y - self.start_y
        return math.sqrt(dx * dx + dy * dy)
```

```
def main():
    rclpy.init()

    # Исправлен парсинг аргументов командной строки (строго по индексам)
    speed = float(sys.argv[1]) if len(sys.argv) > 1 else 0.2
    duration = float(sys.argv[2]) if len(sys.argv) > 2 else 5.0

    node = CalibStraight(speed, duration)

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(node, timeout_sec=0.02)

    expected = node.speed * node.duration
    print(f"\n[ИНФО] Заезд: скорость={node.speed} м/с,
длительность={node.duration} с")
    print(f"[ИНФО] Ожидаемый путь: {expected:.2f} м")
    input("Поставьте робота на стартовую линию и нажмите Enter для
запуска ... ")

    # 1. Ожидание одометрии с помощью spin_once
    node.get_logger().info('Ожидание данных одометрии ... ')
    waited = 0
    while node.start_x is None and waited < 50 and rclpy.ok():
        rclpy.spin_once(node, timeout_sec=0.1)
        waited += 1

    if node.start_x is None:
        node.get_logger().error('Не удалось получить одометрию! Выход.')
        node.destroy_node()
        rclpy.shutdown()
        return

    node.get_logger().info(f'СТАРТ: v={node.speed:.2f} м/с, t={node.durati
on} с')

    # 2. Движение с использованием системных часов ROS 2
    msg = Twist()
    for _ in range(10):
        rclpy.spin_once(node, timeout_sec=0.1)
        node.pub.publish(msg)
    msg.linear.x = node.speed

    start_time = node.get_clock().now()
    duration_ns = node.duration * 1e9

    # Цикл публикации команд скорости с частотой ~10 Гц через таймаут
spin_once
    while rclpy.ok():
        current_time = node.get_clock().now()
        elapsed_ns = (current_time - start_time).nanoseconds

        if elapsed_ns ≥ duration_ns:
            break
```

```
            node.pub.publish(msg)
            # Обрабатывает одометрию и одновременно заменяет сон на 0.1
секунды
            rclpy.spin_once(node, timeout_sec=0.1)

        # 3. Остановка
        stop_msg = Twist()
        node.pub.publish(stop_msg)

        # Даем роботу полностью остановиться физически (0.5 сек собираем
финальный выбег одометрии)
        for _ in range(5):
            rclpy.spin_once(node, timeout_sec=0.1)
            node.pub.publish(stop_msg)

        # 4. Расчет результатов
        odom_dist = node.get_odom_distance()

        node.get_logger().info('=' * 50)
        node.get_logger().info(f'СТОП. Ожидаемое расстояние: {expected:.2f}
м')
        node.get_logger().info(f'Пройдено по одометрии: {odom_dist:.3f} м')
        node.get_logger().info(f'Реальная скорость (по одом): {odom_dist / nod
e.duration:.3f} м/с')
        node.get_logger().info(f'Соотношение (реал/ожид): {odom_dist / expecte
d:.3f}')
        node.get_logger().info('')
        node.get_logger().info('А ТЕПЕРЬ ИЗМЕРЬТЕ РЕАЛЬНОЕ РАССТОЯНИЕ
РУЛЕТКОЙ!')
        node.get_logger().info(f'Если рулетка показывает {expected:.2f} м —
калибровка cmd_scale_vx не нужна.')
        node.get_logger().info(f'Если нет — вычислите новый cmd_scale_vx по
формуле:')
        node.get_logger().info(f'новый_cmd_scale_vx = текущий * (ожидаемое /
реальное_по_рулетке)')
        node.get_logger().info('=' * 50)

        # Корректное последовательное уничтожение без InvalidHandle
        node.destroy_node()
        rclpy.shutdown()

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

Запуск (три заезда с разными скоростями):

```
python3 ~/ws/rbt/scripts/calib_straight.py 0.1 5.0
python3 ~/ws/rbt/scripts/calib_straight.py 0.2 5.0
python3 ~/ws/rbt/scripts/calib_straight.py 0.3 5.0
```

3.3. Запись данных и вычисление нового cmd_scale_vx

Таблица измерений (пример):

Заданная скорость, м/с	Длительность, с	Ожидаемый путь, м	Реальный путь, м	Реальная скорость, м/с
0.1	5.0	0.50	0.44	0.088
0.2	5.0	1.00	0.88	0.176
0.3	5.0	1.50	1.33	0.266

Вычисление нового `cmd_scale_vx`:

```
соотношение = реальная_скорость / заданная_скорость

Для 0.1: 0.088 / 0.1 = 0.88
Для 0.2: 0.176 / 0.2 = 0.88
Для 0.3: 0.266 / 0.3 = 0.887
Среднее = 0.882

новый_cmd_scale_vx = текущий_cmd_scale_vx / среднее_соотношение
новый_cmd_scale_vx = 13.055 / 0.882 ≈ 14.80
```

Почему делим? `cmd_scale_vx` — это коэффициент усиления. Если реальная скорость **меньше** заданной, нужно **увеличить** коэффициент, чтобы давать моторам больше «газа».

Передайте вычисленное значение преподавателю.

4. Уровень «Углублённый»: Почему оба параметра важны

(Что под капотом)

Системный взгляд на калибровку.

4.1. Порядок калибровки имеет значение

```
Сначала: ticks_per_rev (одометрия)
   └─ Без точной одометрии мы не знаем, сколько робот ПРОЕХАЛ.

Потом: cmd_scale_vx (исполнение команд)
   └─ Теперь мы точно измеряем реальную скорость и корректируем усиление.
```

Если сделать наоборот:

- Мы подгоним `cmd_scale_vx` так, чтобы робот проезжал 1 м за 5 секунд.
- Но одометрия всё ещё врёт.
- Когда мы начнём использовать одометрию для навигации, робот будет думать, что он в другом месте.

Вывод: Одометрия — фундамент. Команды — надстройка.

4.2. Откуда берутся ошибки?

- **`ticks_per_rev`:** производственный допуск энкодера, проскальзывание колёс, деформация шины.
- **`cmd_scale_vx`:** масса робота, трение, КПД драйверов, напряжение батареи, износ моторов.

Оба параметра могут «уплывать» со временем. Хорошая практика — проводить калибровку раз в несколько месяцев.

5. Уровень «Исследовательский»: Покрытие и батарея

(Новое применение, прорыв за границы)

Задания для самостоятельного исследования.

5.1. Калибровка на разных покрытиях

Гипотеза: На ковре `cmd_scale_vx` потребуется больше (нужно больше «газа» для преодоления трения).

План: Проведите Этап 2 (три заезда) на линолеуме и на ковре. Сравните полученные коэффициенты.

5.2. Мини-исследование: влияние напряжения батареи

Вопрос: Будет ли `cmd_scale_vx` меняться при разряде батареи?

План:

1. Проведите заезд (0.2 м/с, 5 с) на полностью заряженной батарее. Запишите реальную скорость.
2. Поработайте 30–40 минут. Повторите заезд.
3. Сравните результаты.

Ожидание: На разряженной батарее скорость упадёт. В идеале контроллер должен компенсировать это автоматически, измеряя напряжение.

6. Выводы по занятию

1. **Калибровка — двухэтапный процесс.** Сначала `ticks_per_rev` (точность измерений), затем `cmd_scale_vx` (точность исполнения).
2. **Одометрия — фундамент.** Без точной одометрии невозможна точная навигация.
3. **Эксперимент — источник истины.** Теория говорит «1 метр», но только рулетка и замеры дают правильный ответ.
4. **Параметры моста — рычаги управления точностью.** `ticks_per_rev` и `cmd_scale_vx` позволяют адаптировать модель к реальному железу.
5. **Внешние факторы влияют.** Покрытие и заряд батареи меняют динамику. Инженер это учитывает.

Связь со следующим занятием: Мы откалибровали прямолинейное движение. Теперь — **калибровка поворота**. Параметр `cmd_scale_omega` также нуждается в коррекции.

Домашнее задание:

- **Базовое**: Проведите Этап 1 и Этап 2, заполните обе таблицы, вычислите новые `ticks_per_rev` и `cmd_scale_vx`.
- **Продвинутое**: Проверьте, улучшилась ли точность после калибровки (повторный заезд).
- **Исследовательское**: Исследуйте влияние покрытия или заряда батареи на `cmd_scale_vx`.

Занятие 22: Калибровка поворота (на оригинальном роботе)

Содержание занятия: Экспериментальное определение параметров вращения: base_width и cmd_scale_omega

Цели занятия

- **Образовательная**: Освоить методику калибровки углового движения робота. Понять роль параметров `base_width` и `cmd_scale_omega`.
- **Развивающая**: Применить двухэтапную логику калибровки (сначала измерения, потом команды) к вращательному движению.
- **Воспитательная**: Показать, что точный поворот критичен для навигации: ошибка в 5° на каждом повороте за 4 поворота квадрата даст 20°.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База» — Этап 1: Калибровка base_width по одометрии (35 мин)**
   - Проблема: одометрия врёт по углу.
   - Эксперимент: ручной поворот по телеоп, замер транспортиром, сравнение с `/odom` (угол yaw).
   - Вычисление нового `base_width`.
3. **Уровень «База» — Этап 2: Калибровка cmd_scale_omega по времени (30 мин)**
   - Проблема: робот поворачивает не на тот угол за заданное время.
   - Эксперимент: скрипт с фиксированной угловой скоростью и длительностью.
   - Вычисление нового `cmd_scale_omega`.
4. **Уровень «Углублённый»: Причины ошибок поворота (10 мин)**
   - Момент инерции, сцепление, проскальзывание.
   - Почему ошибка растёт с увеличением скорости.
5. **Уровень «Исследовательский»: Таблица поправок (5 мин — задание на дом)**
   - Калибровка на разных угловых скоростях.
   - Почему при быстром развороте точность падает сильнее.
6. **Подведение итогов (5 мин)**

1. Введение: Почему робот недоворачивает?

На прошлом занятии мы откалибровали прямолинейное движение. Теперь займёмся поворотом. Ошибки вращения накапливаются быстрее и критичнее: ошибка в 5° на каждом из 4 поворотов квадрата — это 20° к концу маршрута.

Как и с линейным движением, есть два источника ошибок:

1. **Одометрия врёт по углу.** Параметр `base_width` (половина ширины колёсной базы) используется в кинематических формулах для расчёта угла поворота. Если он неточен — одометрия показывает неверный угол.
2. **Команды неточны.** Параметр `cmd_scale_omega` преобразует `angular.z` (рад/с) в команды моторов. Нуждается в коррекции.

Порядок — как и раньше: сначала одометрия, потом исполнение команд.

2. Уровень «База» — Этап 1: Калибровка base_width

(Инструмент, навык, интерфейс)

Калибруем одометрию по углу: робот должен правильно сообщать, на сколько он повернулся.

2.1. Проблема: одометрия показывает не тот угол

Для расчёта угла поворота по энкодерам мост использует геометрию робота:

```
угол = (путь_правого_колеса - путь_левого_колеса) / (2 * base_width)
```

где `base_width` — половина расстояния между центрами левых и правых колёс.

Если `base_width` в конфиге не совпадает с реальным, одометрия будет систематически ошибаться по углу.

2.2. Эксперимент: ручной поворот и замер

Инструменты: транспортир (или лист с нарисованными углами), мел.

Порядок действий:

1. Отметьте на полу начальное положение робота и направление (линия «вперёд»).
2. Запустите **ваш телеоператор** и **наблюдение за одометрией** (в отдельном терминале):

   ```
   ros2 topic echo /odom --field pose.pose.orientation
   ```

   Обратите внимание на поле `z` кватерниона — это связано с углом рысканья (yaw). Для простоты используем `ros2 topic echo /odom_fused`, если он есть, или напишем скрипт-помощник.
3. На малой скорости поверните робота **на месте** на 90° (по транспортиру).

4. Запишите, какой угол показала одометрия.

5. Повторите для 180° и 360°.

Упрощённый способ (скрипт-помощник): Создайте `check_yaw.py`:

```
nano ~/ws/rbt/scripts/check_yaw.py
```

```
#!/usr/bin/env python3
"""Выводит текущий угол yaw из одометрии с контролируемой частотой."""
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
import math
import sys

class YawChecker(Node):
    def __init__(self):
        super().__init__('yaw_checker')

        # Подписка на топик одометрии
        self.sub = self.create_subscription(
            Odometry,
            '/odom',
            self.odom_callback,
            10
        )

        # Переменная для хранения угла
        self.yaw = 0.0
        self.has_data = False

        # Таймер для вывода в консоль (5 Гц / каждые 0.2 секунды)
        self.timer = self.create_timer(0.2, self.timer_callback)

    def odom_callback(self, msg):
        q = msg.pose.pose.orientation


# Полная формула перехода из кватерниона в Yaw с защитой от Gimbal Lock
        sinr_cosp = 2.0 * (q.w * q.x + q.y * q.z)
        cosr_cosp = 1.0 - 2.0 * (q.x * q.x + q.y * q.y)
        sinp = 2.0 * (q.w * q.y - q.z * q.x)

        # Защита от выхода за пределы [-1, 1] из-за погрешности float
        sinp = max(min(sinp, 1.0), -1.0)

        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)

        # Проверка на критический наклон (Pitch ~ 90 градусов)
        if abs(sinp) >= 0.9999:
            self.yaw = 2.0 * math.atan2(q.z, q.w)
        else:
            self.yaw = math.atan2(siny_cosp, cosy_cosp)
```

```
        self.has_data = True

    def timer_callback(self):
        """Вывод данных в терминал с фиксированной частотой."""
        if not self.has_data:
            sys.stdout.write("\rОжидание данных из /odom ... ")
            sys.stdout.flush()
            return

        # Переводим радианы в градусы
        deg = math.degrees(self.yaw)

        # Красивый вывод с затиранием строки без засорения системного лога
        sys.stdout.write(f"\rТекущий угол Yaw: {deg:.1f}°   ")
        sys.stdout.flush()

def main():
    rclpy.init()
    node = YawChecker()

    print("Нода запущена. Нажмите Ctrl+C для выхода.")
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        print("\nЗавершение работы ноды.")
        node.destroy_node()
        rclpy.shutdown()

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

2.3. Запись данных и вычисление нового base_width

Таблица измерений (пример):

Поворот	Реальный угол (транспортир), °	Угол по одометрии, °	Соотношение (реал/одом)
1	90	82	90 / 82 = 1.098
2	180	165	180 / 165 = 1.091
3	360	330	360 / 330 = 1.091
Среднее			**1.093**

Вычисление нового `base_width`:

```
Если одометрия занижает угол (показывает меньше реального),
значит base_width в конфиге ЗАВЫШЕН.

новый_base_width = текущий_base_width / среднее_соотношение

Пример:
новый_base_width = 0.0929 / 1.093 ≈ 0.0850
```

Почему делим? В формуле `угол = разница_путей / (2 * base_width)`. Если `base_width` уменьшить, одометрия покажет **больший** угол при той же разнице путей.

3. Уровень «База» — Этап 2: Калибровка cmd_scale_omega

(Инструмент, навык, интерфейс)

Теперь, когда одометрия точна по углу, калибруем исполнение команд поворота.

3.1. Проблема: робот недоворачивает по команде

Мы отправляем `angular.z = 0.5` на 2 секунды. Ожидаем: `0.5 × 2 = 1.0 рад ≈ 57.3°`. Но робот поворачивает, скажем, на 50°. Одометрия (уже откалиброванная) это подтверждает.

Параметр `cmd_scale_omega` преобразует `angular.z` (рад/с) во внутренние команды моторов. Если он неточен — реальная угловая скорость отличается.

3.2. Скрипт для калибровочных поворотов

Создайте `calib_rotation.py`:

```
nano ~/ws/rbt/scripts/calib_rotation.py
```

```
#!/usr/bin/env python3
"""Скрипт для калибровки cmd_scale_omega. Определение угла через unwrap
yaw."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import sys
import math

class CalibRotation(Node):
    def __init__(self, omega, duration, current_scale):
        super().__init__('calib_rotation')

        self.omega = omega
        self.duration = duration
        self.current_scale = current_scale

        # Стандартные издатель и подписчик в дефолтной группе. Без
многопоточности.
        self.pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.odom_sub = self.create_subscription(
            Odometry, '/odom', self.odom_callback, 10
        )

        # Переменные для unwrap yaw
        self.start_yaw_raw = None
        self.prev_yaw_raw = None
        self.unwrap_offset = 0
        self.current_unwrapped = None
```

```
    def odom_callback(self, msg):
        q = msg.pose.pose.orientation
        siny = 2.0 * (q.w * q.z + q.x * q.y)
        cosy = 1.0 - 2.0 * (q.y * q.y + q.z * q.z)
        yaw_raw = math.atan2(siny, cosy)

        if self.start_yaw_raw is None:
            self.start_yaw_raw = yaw_raw
            self.prev_yaw_raw = yaw_raw
            self.unwrap_offset = 0
            self.current_unwrapped = yaw_raw
            return

        delta = yaw_raw - self.prev_yaw_raw

        # Оригинальный алгоритм обнаружения переходов через ±π (без
изменений)
        if delta > math.pi:
            self.unwrap_offset -= 1
        elif delta < -math.pi:
            self.unwrap_offset += 1

        self.current_unwrapped = yaw_raw + self.unwrap_offset * 2.0 *
math.pi
        self.prev_yaw_raw = yaw_raw

    def get_total_rotation(self):
        """Возвращает полный угол поворота от старта (рад)."""
        if self.start_yaw_raw is None or self.current_unwrapped is None:
            return 0.0
        return self.current_unwrapped - self.start_yaw_raw

def main():
    rclpy.init()

    # Считываем аргументы: скорость, время, ТЕКУЩИЙ коэффициент
    omega = float(sys.argv[1]) if len(sys.argv) > 1 else 0.5
    duration = float(sys.argv[2]) if len(sys.argv) > 2 else 4.0
    current_scale = float(sys.argv[3]) if len(sys.argv) > 3 else 1.0

    node = CalibRotation(omega, duration, current_scale)

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(node, timeout_sec=0.02)

    expected_rad = node.omega * node.duration
    expected_deg = math.degrees(expected_rad)

    print(f"\n[ИНФО] Поворот: omega={node.omega} рад/с,
длительность={node.duration} с")
    print(f"[ИНФО] Ожидаемый угол: {expected_deg:.1f}°")
    input("Поставьте робота на свободную площадку и нажмите Enter для
старта ... ")
```

```
    # 1. Ждём одометрию через spin_once главного потока
    node.get_logger().info('Ожидание данных одометрии...')
    waited = 0
    while node.start_yaw_raw is None and waited < 50 and rclpy.ok():
        rclpy.spin_once(node, timeout_sec=0.1)
        waited += 1

    if node.start_yaw_raw is None:
        node.get_logger().error('Нет одометрии! Выход.')
        node.destroy_node()
        rclpy.shutdown()
        return

    node.get_logger().info(f'СТАРТ: omega={node.omega:.2f} рад/с,
t={node.duration} с')
    node.get_logger().info(f'Стартовый yaw: {math.degrees(node.start_yaw_r
aw):.1f}°')

    # 2. Вращение робота
    msg = Twist()
    for _ in range(20):
        rclpy.spin_once(node, timeout_sec=0.05)
        node.pub.publish(msg)
    msg.angular.z = node.omega

    start_time = node.get_clock().now()
    duration_ns = node.duration * 1e9

    # Таймаут spin_once на 0.05 секунды удерживает стабильную частоту
отправки скорости ~20 Гц
    while rclpy.ok():
        current_time = node.get_clock().now()
        elapsed_ns = (current_time - start_time).nanoseconds

        if elapsed_ns >= duration_ns:
            break

        node.pub.publish(msg)
        rclpy.spin_once(node, timeout_sec=0.05)

    # 3. Остановка
    stop_msg = Twist()
    node.pub.publish(stop_msg)

    # Даем роботу время физически затушить инерцию вращения (0.5 сек
выгребаем остатки одометрии)
    for _ in range(10):
        rclpy.spin_once(node, timeout_sec=0.05)
        node.pub.publish(stop_msg)

    # 4. Сбор результатов
    total_rad = node.get_total_rotation()
    total_deg = math.degrees(total_rad)
    ratio = total_rad / expected_rad if expected_rad != 0 else 0.0
    real_omega = total_rad / node.duration
```

```
    suggested_scale = node.current_scale / ratio if ratio != 0 else node.c
urrent_scale

    # Вывод результатов
    node.get_logger().info('=' * 55)
    node.get_logger().info(f'РЕЗУЛЬТАТЫ:')
    node.get_logger().info(f'  Ожидаемый угол:       {expected_deg:8.1f}
°')
    node.get_logger().info(f'  Реальный угол (unwrap):{total_deg:8.1f}°')
    node.get_logger().info(f'  Переходов через ±π:    {abs(node.unwrap_off
set):8}')
    node.get_logger().info(f'  Заданная угл. скорость: {node.omega:8.3f}
рад/с')
    node.get_logger().info(f'  Реальная угл. скорость: {real_omega:8.3f}
рад/с')
    node.get_logger().info(f'  Соотношение (реал/ожид):{ratio:8.3f}')
    node.get_logger().info('')

    if abs(1.0 - ratio) > 0.01:
        node.get_logger().info(f'Реальная скорость отличается от
заданной.')
        node.get_logger().info(f'Текущий cmd_scale_omega: {node.current_sc
ale:.3f}')
        node.get_logger().info(f'Рекомендуемый cmd_scale_omega: {suggested
_scale:.3f}')
    else:
        node.get_logger().info('Калибровка не требуется — соотношение
близко к 1.0.')
    node.get_logger().info('=' * 55)

    node.destroy_node()
    rclpy.shutdown()

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

Запуск (три поворота с разными скоростями):

```
python3 ~/ws/rbt/scripts/calib_rotation.py 0.5 2.0
python3 ~/ws/rbt/scripts/calib_rotation.py 1.0 2.0
python3 ~/ws/rbt/scripts/calib_rotation.py 1.5 2.0
```

3.3. Запись данных и вычисление нового cmd_scale_omega

Таблица измерений (пример):

Заданная omega, рад/с	Длительность, с	Ожидаемый угол, °	Реальный угол, °	Реальная omega, рад/с
0.5	2.0	57.3	50	0.436
1.0	2.0	114.6	102	0.890
1.5	2.0	171.9	155	1.353

Вычисление нового `cmd_scale_omega`:

```
соотношение = реальная_omega / заданная_omega

Для 0.5: 0.436 / 0.5 = 0.872
Для 1.0: 0.890 / 1.0 = 0.890
Для 1.5: 1.353 / 1.5 = 0.902
Среднее = 0.888

новый_cmd_scale_omega = текущий_cmd_scale_omega / среднее_соотношение
новый_cmd_scale_omega = 3.0 / 0.888 ≈ 3.38
```

Почему делим? `cmd_scale_omega` — усиление. Если реальный угол меньше ожидаемого, нужно **увеличить** коэффициент.

4. Уровень «Углублённый»: Почему угол отличается

(Что под капотом)

Разбираем физические причины ошибок поворота.

4.1. Факторы, влияющие на точность поворота

- **Момент инерции.** При резком старте вращения колёса могут проскальзывать, особенно на гладкой поверхности. Робот «промахивается» мимо расчётного угла.
- **Сцепление.** На ковре или неровной поверхности разные колёса имеют разное сцепление → неравномерный поворот.
- **Люфт редукторов.** Есть мёртвая зона, где мотор уже вращается, а колесо ещё нет.
- **Центр масс.** Если центр масс смещён относительно геометрического центра, поворот в одну сторону может отличаться от поворота в другую.

Проверка: Проведите калибровку отдельно для поворота влево и вправо. Если коэффициенты различаются — центр масс смещён.

5. Уровень «Исследовательский»: Таблица поправок

(Новое применение, прорыв за границы)

Задания для углублённого анализа.

5.1. Калибровка на разных скоростях

Проведите Этап 2 для 5 разных значений `angular.z` : 0.2, 0.5, 1.0, 1.5, 2.0. Запишите реальную угловую скорость для каждой.

Вопрос: Одинаково ли соотношение (реал/ожид) для всех скоростей? Если нет — зависимость нелинейна, и одним коэффициентом не обойтись.

5.2. Мини-исследование: почему на высокой скорости точность падает

Гипотеза: При больших угловых скоростях механум-колёса проскальзывают сильнее (центробежная сила, ролики не успевают сцепляться).

Проверка: Сравните соотношение (реал/ожид) для omega=0.5 и omega=2.0. Если для 2.0 соотношение заметно меньше (робот сильнее недоворачивает) — гипотеза подтверждается.

Инженерное решение: Для точных манёвров использовать низкую угловую скорость. Для быстрых разворотов — задавать угол с запасом.

6. Выводы по занятию

1. **Калибровка поворота — два этапа.** Сначала `base_width` для точной одометрии по углу, затем `cmd_scale_omega` для точного исполнения команд.
2. **Порядок важен.** Без точной одометрии мы не знаем истинного угла. Сначала измерения, потом команды.
3. **Поворот чувствительнее прямолинейного движения.** Ошибки накапливаются быстрее: 5° × 4 = 20° за квадрат.
4. **Скорость влияет на точность.** При быстрых разворотах проскальзывание выше — соотношение (реал/ожид) падает.
5. **Асимметрия — признак проблем.** Разные коэффициенты для левого и правого поворота указывают на смещение центра масс.

Связь со следующим занятием: Мы откалибровали и линейное, и угловое движение. Теперь мы умеем точно ездить и точно измерять. На следующем занятии мы изучим другие типы сообщений, сервисы и запись данных через `ros2 bag`.

Домашнее задание:

- **Базовое**: Проведите Этап 1 и Этап 2 калибровки поворота, заполните таблицы, вычислите новые `base_width` и `cmd_scale_omega`.
- **Продвинутое**: Проведите калибровку отдельно для левого и правого поворота. Отличаются ли коэффициенты?
- **Исследовательское**: Проведите калибровку `cmd_scale_omega` для 5 скоростей и постройте график зависимости реальной угловой скорости от заданной.

Занятие 23: Типы сообщений, сервисы, ros2 bag

Содержание занятия: Разнообразие сообщений, синхронное взаимодействие и запись данных

Цели занятия

- **Образовательная**: Познакомиться с разными типами сообщений ROS2, понять разницу между топиками и сервисами, освоить запись и воспроизведение данных через `ros2 bag`.
- **Развивающая**: Научиться выбирать тип сообщения под задачу и способ взаимодействия (асинхронный/синхронный).
- **Воспитательная**: Показать важность логирования данных для отладки и анализа.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Сообщения, монитор, сервисы, ros2 bag (45 мин)**
   - Стандартные типы сообщений: `String`, `Int32`, `Float64`.
   - Узел-монитор: подписчик `/cmd_vel`, расчёт и публикация пройденной дистанции.
   - Сервисы: `ros2 service list`, `ros2 service call`.
   - `ros2 bag`: запись и воспроизведение.
3. **Уровень «Углублённый»: Устройство сообщений и сервисов (20 мин)**
   - Структура `.msg` и `.srv` файлов.
   - Сервис vs топик: когда что использовать.
   - Формат `ros2 bag` (sqlite3).
4. **Уровень «Исследовательский»: Своё сообщение и анализ (15 мин)**
   - Идея пользовательского типа `RobotStatus`.
   - Мини-исследование: размер файла bag при разной частоте записи.
5. **Подведение итогов (5 мин)**

1. Введение: Не только Twist

До сих пор мы работали только с одним типом сообщений — `Twist`. Но ROS2 предоставляет десятки стандартных типов: `String` для текста, `Int32` для чисел, `LaserScan` для лидара, `Image` для камеры.

Кроме топиков (асинхронная публикация), есть **сервисы** — синхронный механизм «запрос → ответ». И есть **ros2 bag** — «чёрный ящик», который записывает все данные для последующего анализа.

Сегодня мы расширим арсенал инструментов.

2. Уровень «База»: Сообщения, монитор, сервисы, ros2 bag

(Инструмент, навык, интерфейс)

Обязательный уровень. Пробуем разные сообщения, создаём полезный узел, работаем с сервисами и записью.

2.1. Стандартные типы сообщений

Кроме `geometry_msgs/msg/Twist`, часто используются:

Тип сообщения	Пакет	Для чего
`String`	`std_msgs`	Текстовые данные
`Int32`, `Float64`	`std_msgs`	Числа
`Header`	`std_msgs`	Временная метка + frame_id
`LaserScan`	`sensor_msgs`	Данные лидара
`Image`	`sensor_msgs`	Изображение с камеры
`Odometry`	`nav_msgs`	Одометрия

Практика в терминале:

```
# Посмотреть структуру сообщения
ros2 interface show std_msgs/msg/String
ros2 interface show geometry_msgs/msg/Twist

# Опубликовать текстовое сообщение вручную
ros2 topic pub /test_string std_msgs/msg/String "data: 'Привет, робот'" --
once

# Подписаться и увидеть его
ros2 topic echo /test_string --once
```

2.2. Узел-монитор: расчёт пройденной дистанции

Создадим полезный узел, который слушает `/cmd_vel` и считает, сколько робот **должен был** проехать (по заданным скоростям), и публикует это в новый топик.

Создайте `distance_monitor.py`:

```
nano ~/ws/rbt/scripts/distance_monitor.py
```

```
#!/usr/bin/env python3
"""Узел: считает расчётную дистанцию по /cmd_vel по фиксированному
таймеру."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from std_msgs.msg import Float64
```

```
class DistanceMonitor(Node):
    def __init__(self):
        super().__init__('distance_monitor')

        # Подписчик на команды движения
        self.sub = self.create_subscription(Twist, '/cmd_vel', self.cmd_ca
llback, 10)

        # Издатель расчётной дистанции
        self.pub = self.create_publisher(Float64, '/estimated_distance', 1
0)

        # Переменные состояния
        self.current_linear_x = 0.0
        self.total_distance = 0.0
        self.last_time = self.get_clock().now()

        # Таймер для интеграции пути с частотой 50 Гц (каждые 0.02 сек)
        # Высокая частота обеспечивает максимальную точность
        self.dt_sec = 0.02
        self.timer = self.create_timer(self.dt_sec, self.timer_callback)

        self.get_logger().info('Монитор дистанции запущен (расчет по
таймеру 50 Гц).')

    def cmd_callback(self, msg):
        """Просто сохраняем последнюю полученную скорость."""
        self.current_linear_x = msg.linear.x

    def timer_callback(self):
        """Интегрируем расстояние по фиксированным промежуткам времени."""
        now = self.get_clock().now()

        # Используем реальное дельта-время таймера на случай системных
задержек
        dt = (now - self.last_time).nanoseconds / 1e9
        self.last_time = now

        # Расстояние = модуль скорости × честное дельта-время
        distance = abs(self.current_linear_x) * dt
        self.total_distance += distance

        # Публикуем накопленную дистанцию
        dist_msg = Float64()
        dist_msg.data = self.total_distance
        self.pub.publish(dist_msg)

        # Логируем раз в секунду, чтобы не спамить в консоль
        self.get_logger().info(
            f'Расчетный путь: {self.total_distance:.3f} м (v={self.current
_linear_x:.2f} м/с)',
            throttle_duration_sec=1.0
        )

def main():
    rclpy.init()
```

```
    node = DistanceMonitor()

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(node, timeout_sec=0.02)
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

Запуск (в отдельном терминале):

```
python3 ~/ws/rbt/scripts/distance_monitor.py
```

Теперь, когда робот движется (телеоп или скрипт), монитор считает расчётную дистанцию и публикует её в `/estimated_distance`.

2.3. Сервисы: запрос → ответ

Сервис — это синхронное взаимодействие: узел А отправляет **запрос**, узел Б обрабатывает и возвращает **ответ**. В отличие от топиков, где сообщения летят в одну сторону.

Практика в терминале:

```
# Посмотреть список сервисов
ros2 service list

# Пример: сбросить одометрию (если есть такой сервис)
ros2 service call /reset_odometry std_srvs/srv/Empty "{}"

# Узнать тип сервиса
ros2 service type /reset_odometry
```

Когда использовать:

- **Топик** — поток данных (скорость, лидар, камера).
- **Сервис** — единичное действие (сброс, калибровка, запрос статуса).

2.4. ros2 bag: запись и воспроизведение

ros2 bag — утилита для записи всех сообщений в выбранных топиках в файл. Потом этот файл можно «проиграть», и узлы будут думать, что данные идут в реальном времени.

Запись:

```
# Записываем /cmd_vel и /odom
ros2 bag record /cmd_vel /odom -o my_test_bag
# Пусть робот что-то делает... Нажмите Ctrl+C для остановки записи.
```

Просмотр информации:

```
ros2 bag info my_test_bag
```

Воспроизведение:

```
# Остановите реального робота! При воспроизведении данные идут в те же топики.
ros2 bag play my_test_bag
```

Зачем это нужно:

- Отладка: записать данные с датчиков и потом анализировать.
- Тестирование: записать реальные данные и прогонять на них новые алгоритмы без робота.
- Демонстрация: показать, как работал робот, без его запуска.

3. Уровень «Углублённый»: Устройство сообщений и сервисов

(Что под капотом)

Как создаются типы сообщений и чем сервис отличается от топика.

3.1. Как создаются пользовательские типы

Стандартные сообщения хранятся в пакетах ROS2. Их структура описывается в файлах `.msg`:

```
# Посмотрим структуру Twist
ros2 interface show geometry_msgs/msg/Twist
```

Вывод:

```
   Vector3 linear
   Vector3 angular
```

А `Vector3`:

```
   float64 x
   float64 y
   float64 z
```

Для создания своего типа нужно:

1. Создать файл `.msg` в пакете.
2. Описать поля (каждое с новой строки: `тип имя`).
3. Собрать пакет через `colcon build`.

Но мы не создаём пакеты, поэтому пользуемся стандартными типами.

3.2. Сервис vs топик

Характеристика	Топик (Topic)	Сервис (Service)
Направление	Однонаправленное	Запрос → Ответ
Время	Асинхронное	Синхронное (ждём ответ)
Использование	Потоки данных	Команды, запросы
Пример	`/cmd_vel`, `/scan`	`/reset_odometry`, `/set_pose`

Аналогия: Топик — это радиостанция (вещает, кто хочет — слушает). Сервис — это телефонный звонок (спросил — получил ответ).

3.3. Формат ros2 bag

Файл `.bag` — это база данных **SQLite3**. Внутри:

- Таблица `topics` — имена и типы топиков.
- Таблица `messages` — сами сообщения с временными метками.

Можно открыть bag-файл обычным браузером SQLite и посмотреть данные напрямую. Это удобно для продвинутого анализа.

4. Уровень «Исследовательский»: Своё сообщение и анализ

(Новое применение, прорыв за границы)

Идеи для самостоятельного расширения.

4.1. Идея: тип сообщения RobotStatus

Представьте, что мы хотим публиковать состояние робота одним сообщением:

```
float64 speed        # текущая скорость м/с
float64 battery      # заряд батареи %
float64 temperature  # температура процессора °C
string  mode         # режим: "idle", "moving", "charging"
```

Это потребовало бы создания своего пакета с `.msg` файлом. Но мы можем эмулировать это, публикуя четыре отдельных топика или используя `String` с JSON-строкой.

Упрощённый вариант (JSON в String):

```python
import json
msg = String()
msg.data = json.dumps({"speed": 0.2, "battery": 85, "temp": 42})
pub.publish(msg)
```

4.2. Мини-исследование: размер bag при разной частоте

Вопрос: Как растёт размер bag-файла при увеличении частоты публикации?

Методика:

1. Запустите издатель, публикующий `Float64` с частотой 10 Гц в течение 60 секунд. Запишите bag.
2. Повторите с частотами 50 Гц и 100 Гц.
3. Сравните размеры файлов `ros2 bag info`.

Ожидаемый результат: Размер растёт **пропорционально** частоте. 100 Гц → файл в 10 раз больше, чем 10 Гц.

Вывод: Не стоит записывать все топики на максимальной частоте без необходимости. Выбирайте только нужные топики и разумную частоту.

5. Выводы по занятию

1. **Сообщения бывают разные.** `String`, `Int32`, `Float64` — для простых данных. `Twist`, `LaserScan` — для специализированных.
2. **Монитор дистанции — полезный узел.** Считает расчётный путь по командам и публикует в отдельный топик.
3. **Сервисы — для запросов.** Синхронное взаимодействие: отправил запрос — получил ответ.
4. **ros2 bag — чёрный ящик.** Записывает данные для отладки, тестирования и демонстрации.
5. **Размер bag растёт с частотой.** Записывайте только нужное.

Связь со следующим занятием: Мы освоили отдельные инструменты ROS2. На следующем занятии — **комплексная практика**: напишем программу, которая проведёт робота по квадрату, используя всё, чему мы научились.

Домашнее задание:

- **Базовое**: Запишите bag-файл с топиками `/cmd_vel` и `/odom` во время ручного управления. Проверьте его размер.
- **Продвинутое**: Модифицируйте `distance_monitor.py`, чтобы он также считал **угловой путь** (сумму `|angular.z| × dt`) и публиковал в `/estimated_rotation`.
- **Исследовательское**: Проведите эксперимент с замером размера bag при 10, 50, 100 Гц. Постройте график.

Занятие 24: Комплексная практика ROS2

Содержание занятия: Автономное движение по траектории «квадрат» с контролем по одометрии

Цели занятия

- **Образовательная**: Объединить навыки работы с ROS2: издатель `Twist`, подписчик `/odom`, расчёт пройденного пути и угла в реальном времени, движение по траектории с обратной связью.

- **Развивающая**: Провести натурный эксперимент, сравнить траекторию по одометрии с реальными замерами, понять природу накопления ошибок.
- **Воспитательная**: Показать, что даже с обратной связью по одометрии точность ограничена — необходимость внешних сенсоров.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Программа «Квадрат» по одометрии (45 мин)**
   - Постановка задачи: проехать квадрат 1×1 м, контролируя движение по `/odom`.
   - Логика: едем вперёд, пока одометрия не покажет 1.0 м, затем поворачиваем на месте, пока yaw не изменится на 90°.
   - Написание скрипта `square_odom.py`.
   - Запуск, наблюдение, замеры.
3. **Уровень «Углублённый»: Анализ ошибок (25 мин)**
   - Сравнение траектории по одометрии с реальной.
   - Где робот отклонился и почему одометрия «врёт».
   - Влияние проскальзывания и люфтов.
4. **Уровень «Исследовательский»: Накопление ошибки (10 мин)**
   - Эксперимент: несколько квадратов подряд.
   - График роста ошибки.
5. **Подведение итогов (5 мин)**

1. Введение: От слепого движения — к движению с обратной связью

На прошлых занятиях мы ездили «по времени»: `move_forward(1.0, 0.2)` — едем 5 секунд и надеемся, что проехали ровно метр. Это **разомкнутое управление**: мы не проверяем, сколько проехали на самом деле.

Сегодня мы реализуем **замкнутое управление по одометрии**:

- Едем вперёд, **пока** одометрия не покажет, что пройден 1 метр.
- Поворачиваем, **пока** угол yaw не изменится на 90°.

Робот сам решает, когда остановиться, на основе данных с энкодеров. Это кибернетика в действии: обратная связь!

2. Уровень «База»: Программа «Квадрат» по одометрии

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем скрипт с обратной связью по `/odom`.

2.1. Постановка задачи

Робот должен проехать квадрат 1×1 м, используя одометрию для контроля:

1. Ехать вперёд, пока пройденное расстояние (по одометрии) не станет ≥ 1.0 м.
2. Повернуть влево на месте, пока накопленный угол поворота (по одометрии) не станет ≥ 90°.
3. Повторить 4 раза.

Отличие от «слепого» метода:

- Раньше: «ехать 5 секунд» — если колёса проскальзывают, робот недоезжает.
- Теперь: «ехать, пока одометрия не скажет 1 метр» — если колёса проскальзывают, робот едет дольше, компенсируя проскальзывание (в пределах возможностей энкодеров).

2.2. Скрипт square_odom.py

Создайте файл:

```
nano ~/ws/rbt/scripts/square_odom.py
```

```
#!/usr/bin/env python3
"""Автономный проезд квадрата 1x1 м с контролем по одометрии."""
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import math

class SquareOdomRunner(Node):
    def __init__(self):
        super().__init__('square_odom_runner')
        self.pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.odom_sub = self.create_subscription(Odometry, '/odom', self.o
dom_callback, 10)

        # Позиция робота
        self.current_x = 0.0
        self.current_y = 0.0
        self.start_x = 0.0
        self.start_y = 0.0
        self.got_first_odom = False

        # Переменные для unwrap yaw (непрерывного угла)
        self.prev_yaw_raw = 0.0
        self.unwrap_offset = 0.0
        self.current_unwrapped = 0.0

    def odom_callback(self, msg):
        self.current_x = msg.pose.pose.position.x
        self.current_y = msg.pose.pose.position.y

        # Кватернион в Yaw
        q = msg.pose.pose.orientation
```

```
        siny = 2.0 * (q.w * q.z + q.x * q.y)
        cosy = 1.0 - 2.0 * (q.y * q.y + q.z * q.z)
        yaw_raw = math.atan2(siny, cosy)

        if not self.got_first_odom:
            self.start_x = self.current_x
            self.start_y = self.current_y
            self.prev_yaw_raw = yaw_raw
            self.current_unwrapped = yaw_raw
            self.got_first_odom = True
            return

        # Корректный алгоритм Unwrap без ручных if/else условий на pi
        delta = yaw_raw - self.prev_yaw_raw
        if delta > math.pi:
            self.unwrap_offset -= 2.0 * math.pi
        elif delta < -math.pi:
            self.unwrap_offset += 2.0 * math.pi

        self.current_unwrapped = yaw_raw + self.unwrap_offset
        self.prev_yaw_raw = yaw_raw

    def get_distance(self):
        dx = self.current_x - self.start_x
        dy = self.current_y - self.start_y
        return math.sqrt(dx*dx + dy*dy)

    def reset_distance_origin(self):
        self.start_x = self.current_x
        self.start_y = self.current_y

    def stop_robot(self):
        msg = Twist()  # Все поля по умолчанию 0.0
        self.pub.publish(msg)

    def wait_and_stabilize(self, duration_sec=0.5):
        """Безопасная пауза без блокировки timeout'ом."""
        self.get_logger().info(f'Пауза {duration_sec} с для
стабилизации...')
        steps = int(duration_sec / 0.02)
        for _ in range(steps):
            self.stop_robot()
            rclpy.spin_once(self, timeout_sec=0.02)

    def move_forward_until(self, target_distance, speed):
        self.reset_distance_origin()
        msg = Twist()
        msg.linear.x = speed

        self.get_logger().info(f'Еду вперёд до {target_distance} м...')
        while rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.02)
            if not self.got_first_odom:
                continue

            if self.get_distance() >= target_distance:
```

```
                break
            self.pub.publish(msg)

        self.stop_robot()
        self.get_logger().info(f'Пройдено: {self.get_distance():.3f} м')
        self.wait_and_stabilize(0.5)

    def turn_left_until(self, target_angle_deg, omega):
        # Одометрия инкрементальна. Нам важно не просто повернуться на 90
градусов
        # от текущего положения, а компенсировать перелет прошлого шага!
        target_rad = math.radians(target_angle_deg)
        start_unwrapped = self.current_unwrapped

        msg = Twist()
        msg.angular.z = omega

        self.get_logger().info(f'Поворот влево на {target_angle_deg}° ... ')
        while rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.02)
            if not self.got_first_odom:
                continue

            angle_change = self.current_unwrapped - start_unwrapped
            if angle_change >= target_rad:
                break
            self.pub.publish(msg)

        self.stop_robot()
        final_change = self.current_unwrapped - start_unwrapped
        self.get_logger().info(f'Повёрнуто: {math.degrees(final_change):.1
f}° ')
        self.wait_and_stabilize(0.5)

    def run_square(self, side_length, speed, omega):
        self.get_logger().info('СТАРТ КВАДРАТА')

        # Ждем первую одометрию перед стартом, чтобы не ехать в "никуда"
        while rclpy.ok() and not self.got_first_odom:
            self.get_logger().info('Ожидание данных одометрии ... ')
            rclpy.spin_once(self, timeout_sec=0.5)

        for i in range(4):
            self.get_logger().info(f'\n—— Сторона {i+1}/4 —— ')
            self.move_forward_until(side_length, speed)
            self.turn_left_until(90.0, omega)

        self.get_logger().info('\nКВАДРАТ ЗАВЕРШЁН!')


def main():
    rclpy.init()
    node = SquareOdomRunner()

    side = 1.0
    speed = 0.2
```

```
    omega = 0.5

    print("Поставьте робота в начальную точку и нажмите Enter...")
    input()

    try:
        node.run_square(side, speed, omega)
    except KeyboardInterrupt:
        pass
    finally:
        node.stop_robot()
        node.destroy_node()
        rclpy.shutdown()

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

2.3. Запуск и наблюдение

Перед запуском:

1. Убедитесь, что одометрия откалибрована (занятия 21–22).

Запуск:

```
python3 ~/ws/rbt/scripts/square_odom.py
```

Во время движения:

- Следите за логами: видно, сколько реально проехал и повернул по одометрии.

После завершения:

- Измерьте рулеткой отклонение от стартовой точки.
- Запишите: отклонение по X, Y и угол.

3. Уровень «Углублённый»: Анализ ошибок

(Что под капотом)

Сравниваем траекторию по одометрии с реальностью.

3.1. Одометрия vs реальность

После проезда квадрата у вас есть два набора данных:

1. **По одометрии:** робот думает, что он в точке (0, 0), угол 0°.
2. **По рулетке:** реальное положение робота.

Заполните таблицу:

Параметр	По одометрии	Реально (рулетка)	Ошибка
X, м	0.0		
Y, м	0.0		
Угол, °	0		

Главный вопрос: Если одометрия говорит «я в нуле», а робот реально сместился — значит, одометрия накопила ошибку. Откуда она?

3.2. Почему одометрия «врёт»

Даже откалиброванная одометрия имеет фундаментальные ограничения:

1. **Проскальзывание колёс.** Энкодер считает обороты колеса, но если колесо прокрутилось на месте — одометрия показывает движение, которого не было.
2. **Боковой увод.** Меканум-колёса могут слегка смещаться вбок, но энкодеры этого не измеряют.
3. **Неровности пола.** Стык плитки может отклонить робота — одометрия этого не заметит.
4. **Дискретизация.** Энкодер имеет конечное разрешение (ticks_per_rev). Маленькие перемещения теряются.

Вывод: Одометрия хороша для **кратковременного** контроля (секунды). Для длительного движения (минуты) ошибка неизбежно накапливается.

4. Уровень «Исследовательский»: Накопление ошибки

(Новое применение, прорыв за границы)

Количественный эксперимент.

4.1. Эксперимент: несколько квадратов подряд

Модифицируйте `run_square`, чтобы робот проезжал 3 или 5 квадратов подряд (12 или 20 сторон).

После каждого квадрата записывайте:

- Где робот реально оказался (рулетка).
- Что показывает одометрия.

Постройте график:

- Ось X — количество сторон.
- Ось Y — ошибка позиции (расстояние от реальной точки до ожидаемой).

Ожидаемый результат: Ошибка растёт. Первые 4 стороны — ошибка 5–10 см. Ещё 4 стороны — ошибка 20–30 см. Это называется **дрейф одометрии**.

5. Выводы по занятию

1. **Замкнутое управление по одометрии — шаг вперёд.** Робот сам решает, когда остановиться, на основе измерений, а не по секундомеру.
2. **Одометрия лучше слепого движения, но не идеальна.** Проскальзывания и уводы накапливаются.
3. **Ошибка одометрии растёт со временем.** Для длинных маршрутов нужны внешние сенсоры (лидар, камера).
4. **Квадрат — отличный тест.** Если робот возвращается в начало с ошибкой < 10 см — калибровка выполнена хорошо.

Связь со следующим занятием: Мы завершаем блок ROS2 **зачётом**. Каждый продемонстрирует своего телеоператора и проезд по траектории с контролем по одометрии.

Домашнее задание:

- **Базовое**: Запустите `square_odom.py`, измерьте отклонение, запишите.
- **Продвинутое**: Сравните точность «слепого» квадрата (занятие 24 старая версия) и квадрата по одометрии. Какой точнее?
- **Исследовательское**: Проведите эксперимент с 5 квадратами, постройте график роста ошибки. Запрограммируйте змейку и восьмерку.

Занятие 25: Зачёт по ROS2

Содержание занятия: Демонстрация практических навыков работы с Robot Operating System 2

Цели занятия

- **Образовательная**: Проверить усвоение ключевых концепций ROS2: узлы, топики, сообщения, издатели, подписчики, калибровка, одометрия.
- **Развивающая**: Продемонстрировать умение самостоятельно написать узел для движения по траектории с контролем по одометрии.
- **Воспитательная**: Сформировать уверенность в своих силах: «Я могу запрограммировать робота на автономное выполнение миссии».

План занятия (90 минут)

1. **Организационный момент и регламент зачёта (5 мин)**
2. **Зачётная сессия (75 мин)**
   - Каждый ученик выполняет индивидуальное задание.
   - Демонстрирует работу своего телеоператора.
   - Отвечает на теоретические вопросы.
3. **Подведение итогов блока ROS2 (10 мин)**

1. Введение: Чему мы научились в блоке ROS2

За 8 занятий (17–24) мы прошли путь от «что такое ROS2» до автономного проезда квадрата с контролем по одометрии:

Занятие	Тема	Ключевой навык
17	Архитектура ROS2, teleop	Узлы, топики, управление с клавиатуры
18	Издатель, пакет, colcon	Публикация Twist из кода
19	Свой телеоператор	Захват клавиш, curses
20	Подписчик, rqt_graph	Приём сообщений, визуализация графа
21	Калибровка прямолинейного движения	Точность перемещений
22	Калибровка поворота	Точность вращения
23	Типы сообщений, сервисы, ros2 bag	Разнообразие данных, запись
24	Комплексная практика	Автономный квадрат по одометрии

Сегодня — **зачёт**. Вы покажете, что умеете управлять роботом программно, понимаете, как данные текут в ROS2, и можете объяснить свои действия.

2. Зачётные задания

Три уровня сложности. Каждый ученик выбирает уровень по своим силам или по указанию преподавателя.

2.1. Уровень «База»: Индивидуальная траектория

(Инструмент, навык, интерфейс)

Задание: Написать и запустить скрипт, который проводит робота по заданной траектории с контролем по одометрии.

Варианты траекторий (преподаватель назначает или ученик выбирает):

Вариант	Траектория	Параметры
1	Треугольник	Сторона 0.5 м, углы 120°
2	Прямоугольник	1.0 × 0.5 м
3	Квадрат (малый)	Сторона 0.5 м
4	Линия туда-обратно	Вперёд 1.5 м, назад 1.5 м

Требования:

1. Скрипт должен использовать **контроль по одометрии** (не по времени).
2. После завершения робот должен **явно остановиться** (нулевая скорость).
3. В логах должно быть видно, сколько реально пройдено и повёрнуто по одометрии.
4. Замерьте рулеткой отклонение от конечной точки.

Дополнительно: Продемонстрируйте работу **своего телеоператора** (`my_teleop.py`).

2.2. Теоретические вопросы (уровень «База»)

Ответьте устно:

1. Что такое **узел (Node)** в ROS2? Приведите пример узлов на нашем роботе.
2. Что такое **топик (Topic)**? Какие топики вы использовали?
3. Какие поля есть в сообщении `Twist` ? За что отвечает `linear.x` и `angular.z` ?
4. Как посмотреть список активных топиков? Как посмотреть данные, летящие в топик?
5. Чем отличается управление «по времени» от управления «по одометрии»?

3. Уровень «Углублённый»: Усложнённое задание

(Что под капотом)

Для тех, кто справился с базовым заданием.

3.1. Задание: траектория с разными скоростями

Модифицируйте скрипт: робот должен проехать квадрат, но **каждая сторона с разной скоростью**:

- Сторона 1: 0.1 м/с (медленно).
- Сторона 2: 0.2 м/с.
- Сторона 3: 0.3 м/с.
- Сторона 4: 0.1 м/с.

Повороты — с одинаковой скоростью 0.5 рад/с.

Вопрос: Влияет ли скорость на точность движения? На какой стороне ошибка больше?

3.2. Теоретические вопросы (уровень «Углублённый»)

1. **Почему важна калибровка?** Что будет, если запустить квадрат с неоткалиброванными `cmd_scale_vx` и `cmd_scale_omega` ?
2. **Как работает связка издатель-подписчик?** Нарисуйте на доске схему: ваш узел → `/cmd_vel` → `mecanum_bridge` → STM32 → моторы.
3. **Что такое QoS?** Какие настройки QoS вы использовали при создании издателя? Что означает число 10 в `create_publisher(Twist, '/cmd_vel', 10)` ?
4. **Чем отличается топик от сервиса?** Приведите пример, когда нужен сервис, а не топик.
5. **Как работает `ros2 bag` ?** Зачем записывать данные с робота?

4. Уровень «Исследовательский»: Улучшения и анализ

(Новое применение, прорыв за границы)

Для тех, кто хочет выйти за рамки стандартного задания.

4.1. Задание: предложить способ улучшения точности

Проблема: Даже с контролем по одометрии робот не возвращается точно в начало квадрата. Ошибка накапливается.

Задание: Предложите **свой способ** улучшить точность движения **без использования лидара и камеры** (только одометрия и `cmd_vel`).

Возможные идеи (для вдохновения, но придумайте свою):

- Замедлять робота перед достижением цели (плавное торможение).
- Делать несколько попыток поворота: повернул на 85° — довернул ещё на 5°.
- Учитывать «мёртвую зону»: давать чуть большую команду и останавливаться по одометрии.
- Калибровать отдельно левый и правый поворот.

Форма сдачи: Устное объяснение или короткий скрипт, демонстрирующий идею.

4.2. Мини-исследование: предсказание ошибки

Вопрос: Можно ли предсказать итоговую ошибку квадрата по ошибке на первой стороне?

Методика:

1. Запустите квадрат (4 стороны).
2. После первой стороны измерьте отклонение (рулеткой).
3. После четвёртой стороны измерьте итоговое отклонение.
4. Связаны ли эти величины? Если после первой стороны ошибка 5 см, означает ли это, что после четырёх будет 20 см?

Ожидаемый вывод: Ошибка не обязательно линейна. Поворот может как компенсировать, так и усиливать ошибку предыдущей прямой. Это зависит от геометрии ошибки (смещение по X или по Y, угловая ошибка).

5. Подведение итогов блока «Основы ROS2»

Чему мы научились

1. **Архитектура ROS2:** узлы, топики, сообщения, DDS.
2. **Инструменты командной строки:** `ros2 node list`, `ros2 topic list/echo/hz/info`.
3. **Создание узлов:** издатель `Twist`, подписчик на `/odom`, телеоператор.

4. **Калибровка:** приведение модели робота в соответствие с реальностью (`ticks_per_rev` , `cmd_scale_vx` , `base_width` , `cmd_scale_omega`).
5. **Замкнутое управление:** движение до достижения заданного расстояния/угла по одометрии.
6. **Запись и анализ данных:** `ros2 bag` .

Что дальше

Следующий большой блок — **«Программирование движения на меканум-базе»**.

Это переход от «скриптов» к настоящей программной архитектуре робота.

Домашнее задание (на каникулы между блоками):

- Повторите определения: узел, топик, сообщение, издатель, подписчик, сервис.
- Убедитесь, что ваш телеоператор и скрипт квадрата работают и вы понимаете каждую строчку.
- Подумайте: что бы вы улучшили в своём коде?

Программирование движения на меканум-базе

Занятие 26: Архитектура автопилота: сервисы go_time, go_relative, stop

Содержание занятия: Знакомство с сервисным управлением роботом. Запуск и вызов из командной строки

Цели занятия

- **Образовательная**: Понять, что такое ROS2-сервис, чем он отличается от топика. Научиться вызывать сервисы автопилота из командной строки.
- **Развивающая**: Освоить три сервиса управления роботом: движение по времени, движение относительно, экстренная остановка.
- **Воспитательная**: Показать эволюцию управления: от ручной публикации `Twist` к высокоуровневым командам через сервисы.

План занятия (90 минут)

1. **Организационный момент и введение (5 мин)**
2. **Уровень «База»: Сервисы автопилота из командной строки (45 мин)**
   - Что такое сервис в ROS2. Request и Response.
   - Запуск ноды `robot_autopilot`.
   - Вызов `go_time`: робот едет вперёд 2 секунды.
   - Вызов `go_relative`: проехать 1 метр, повернуть на 90°.
   - Вызов `stop`: экстренная остановка.
   - Чтение телеметрии.
3. **Уровень «Углублённый»: Как устроен сервис и автопилот (25 мин)**
   - Структура сервиса: `.srv` файл, Request, Response.
   - Сервис vs топик.
   - Три контура безопасности: watchdog, stuck detector, timeout.
   - Статусы автопилота.
4. **Уровень «Исследовательский»: Точность и гонка сервисов (10 мин)**
   - Сравнение точности `go_time` vs `go_relative`.
   - Что будет при одновременном вызове двух сервисов.
5. **Подведение итогов (5 мин)**

1. Введение: Новый уровень управления роботом

В предыдущем блоке мы управляли роботом «вручную»: публиковали `Twist` в `/cmd_vel`, подписывались на `/odom`, писали циклы «пока не доедем». Это дало нам понимание основ ROS2.

Теперь мы переходим на новый уровень абстракции. Вместо того чтобы думать о скоростях и одометрии, мы будем отдавать **высокоуровневые команды**:

- «Проедь 1 метр вперёд».
- «Поверни на 90°».
- «Стоп!».

Эти команды принимает **сервисный автопилот** (`robot_autopilot`) — готовая нода, которая сама управляет движением по одометрии. Мы будем вызывать её через **сервисы ROS2**.

2. Уровень «База»: Сервисы автопилота из командной строки

(Инструмент, навык, интерфейс)

Обязательный уровень. Учимся запускать автопилот и вызывать его сервисы.

2.1. Что такое сервис в ROS2

Сервис — это синхронный механизм взаимодействия: клиент отправляет **запрос (Request)**, сервер обрабатывает и возвращает **ответ (Response)**.

В отличие от топика (где сообщения летят в одну сторону), сервис — это диалог:

```
Клиент: «Проедь 1 метр вперёд»
Сервер: «Принято, начинаю движение» (мгновенный ответ)
 ... робот едет ...
Статус в телеметрии меняется: MOVING → SUCCESS
```

Наш автопилот предоставляет три сервиса:

Сервис	Тип	Что делает
`autopilot/go_time`	`AutopilotTime`	Ехать с заданными скоростями заданное время
`autopilot/go_relative`	`AutopilotRelative`	Переместиться на (dx, dy) и повернуть на dyaw_deg
`autopilot/stop`	`Trigger`	Экстренная остановка, сброс в IDLE

2.2. Запуск ПО робота

Учитель запускает основной стек и автопилот:

```
ros2 launch mecanum_bringup bridge_ekf_lidar.launch.py
```

Ученики проверяют, что сервисы появились:

```
ros2 service list | grep autopilot
```

Должны увидеть:

```
/autopilot/go_time
/autopilot/go_relative
/autopilot/stop
```

При использовании симулятора он запускается командой `ros2 launch moebius_mock mock_debug_launch.py scene:=hall.yaml start_rviz:=true use_lite:=true start_autopilot:=true`.

При использовании стороннего робота, например TurtleBot автопилот запускается командой `ros2 run robot_autopilot autopilot_node.py`.

2.3. Вызов go_time: движение по времени

Структура сервиса `go_time`:

- Request: `vx`, `vy`, `omega` (скорости), `duration` (секунды).
- Response: `success` (bool), `status_message` (string).

Вызов из командной строки:

```
ros2 service call /autopilot/go_time ifra_mol_1/srv/AutopilotTime \
  "{vx: 0.2, vy: 0.0, omega: 0.0, duration: 2.0}"
```

Ожидаемый результат:

- Терминал покажет ответ:

```
waiting for service to become available...
requester: making request: ifra_mol_1.srv.AutopilotTime_Request(vx=0.2,
vy=0.0, omega=0.0, duration=2.0)

response:
ifra_mol_1.srv.AutopilotTime_Response(success=True, status_message='Accept
ed TIME_DRIVE: 2.0s')
```

- Робот начнёт ехать вперёд 2 секунды.
- Через 2 секунды остановится.

Попробуйте другие параметры:

```
# Поворот на месте 3 секунды
ros2 service call /autopilot/go_time ifra_mol_1/srv/AutopilotTime \
  "{vx: 0.0, vy: 0.0, omega: 0.5, duration: 3.0}"

# Крабовое движение (меканум!) вправо 2 секунды
ros2 service call /autopilot/go_time ifra_mol_1/srv/AutopilotTime \
  "{vx: 0.0, vy: -0.2, omega: 0.0, duration: 2.0}"
```

2.4. Вызов go_relative: относительное перемещение

Структура сервиса `go_relative`:

- Request: `dx` (метры вперёд), `dy` (метры влево), `dyaw_deg` (градусы поворота).
- Response: `success` (bool), `status_message` (string).

Вызов из командной строки:

```
# Проехать 1 метр вперёд
ros2 service call /autopilot/go_relative ifra_mol_1/srv/AutopilotRelative
\
  "{dx: 1.0, dy: 0.0, dyaw_deg: 0.0}"

# Повернуть на 90° влево
ros2 service call /autopilot/go_relative ifra_mol_1/srv/AutopilotRelative
\
  "{dx: 0.0, dy: 0.0, dyaw_deg: 90.0}"

# Проехать 0.5 м вперёд и 0.3 м влево (диагональ), повернуть на 45°
ros2 service call /autopilot/go_relative ifra_mol_1/srv/AutopilotRelative
\
  "{dx: 0.5, dy: 0.3, dyaw_deg: 45.0}"
```

Важно: Сервис отвечает **мгновенно** (принял/отклонил). Фактическое выполнение отслеживается через телеметрию.

2.5. Вызов stop: экстренная остановка

```
ros2 service call /autopilot/stop std_srvs/srv/Trigger "{}"
```

Эффект: Робот немедленно останавливается, автопилот сбрасывается в IDLE. Можно вызвать в любой момент, даже во время выполнения `go_time` или `go_relative`.

2.6. Чтение статуса через телеметрию

Автопилот публикует телеметрию в топик `/autopilot/telemetry` (тип `AutopilotTelemetry`):

```
ros2 topic echo /autopilot/telemetry
```

Вы увидите:

```
x: 0.523
y: -0.014
yaw: 0.02
status: MOVING
***
x: 0.998
y: -0.018
yaw: 0.03
status: SUCCESS
```

Статусы:

- `IDLE` — ничего не делает, ждёт команду.
- `MOVING` — выполняет команду.
- `SUCCESS` — команда выполнена успешно.
- `ABORTED_TIMEOUT` — потерян heartbeat от клиента.
- `ABORTED_STUCK` — робот застрял (детектор застревания).

3. Уровень «Углублённый»: Как устроен сервис и автопилот

(Что под капотом)

Разбираем структуру сервисов и контуры безопасности.

3.1. Структура сервиса: Request и Response

Сервис описывается в файле '.srv'. Например, 'AutopilotRelative.srv':

```
float32 dx
float32 dy
float32 dyaw_deg
***
bool success
string status_message
```

Всё, что **до** '***' — Request (запрос). Всё, что **после** — Response (ответ).

Посмотреть структуру любого сервиса:

```
ros2 interface show ifra_mol_1/srv/AutopilotRelative
```

3.2. Сервис vs Топик

Характеристика	Топик (Topic)	Сервис (Service)
Направление	Однонаправленное	Запрос → Ответ
Время	Асинхронное (не ждём)	Синхронное (ждём ответ)
Для чего	Потоковые данные (лидар, камера)	Команды, запросы (движение, стоп)
Кто кому	Издатель → многие подписчики	Клиент → один сервер

Почему для движения выбран сервис, а не топик?

- Клиент **знает**, принята ли команда (`success=True/False`).
- Нельзя случайно отправить две команды одновременно (сервер обрабатывает последовательно).
- Сервер может отклонить команду (нет одометрии).

3.3. Три контура безопасности автопилота

Автопилот — это не просто исполнитель команд. Он защищает робота:

1. **Watchdog (сторожевой таймер).** Клиент должен слать `heartbeat` (пустое сообщение в `/autopilot/heartbeat`). Если heartbeat не приходит **1.5 секунды** — автопилот считает, что программа-клиент зависла, и экстренно останавливает робота.

2. **Stuck Detector (детектор застревания).** Если робот выполняет `go_relative`, но его одометрия не меняется более **3 секунд** (сдвиг < 1 см, поворот < 1.1°) — автопилот считает, что робот застрял, и прекращает попытку.

3. **Высокоуровневый Перехватчик Аварий (Логика Дерева Поведения).** Клиент (`MoveRelativeAction`) следит за выполнением. Если нижний уровень уже сообщил о поломке - топик телеметрии приносит статус, отличный от MOVING или SUCCESS (например, ABORTED_STUCK), корневому узлу возвращается статус FAILURE. Дерево Поведения мгновенно блокирует выдачу следующих команд, останавливает сценарий и вызывает встроенный в main() безопасный метод деструкции ноды, исключая хаотичное движение робота (но это не точно:).

4. Уровень «Исследовательский»: Точность и гонка сервисов

(Прорыв за границы)

Сравниваем методы и исследуем поведение системы.

4.1. Сравнение точности go_time vs go_relative

Эксперимент:

1. Вызовите `go_time(vx=0.2, duration=5.0)` — цель: проехать 1 м. Замерьте рулеткой.
2. Вызовите `go_relative(dx=1.0, dy=0, dyaw_deg=0)` — та же цель. Замерьте.

Ожидаемый результат: `go_relative` точнее, потому что использует **обратную связь по одометрии** (едет, пока не достигнет 1 м). `go_time` просто едет 5 секунд и не корректируется.

Когда нужен `go_time`? Для манёвров, где не важна точная дистанция (например, «отъехать немного назад»), или для дуг (одновременно ехать и поворачиваться).

Примечание: Чтобы сервис не останавливался по сторожевому таймеру в отдельном терминале следует публиковать сердцебиение:

```
ros2 topic pub -r 5 /autopilot/heartbeat std_msgs/msg/Empty "{}"
```

4.2. Мини-исследование: гонка сервисов

Вопрос: Что будет, если из двух разных терминалов **одновременно** вызвать два сервиса?

```
# Терминал 1:
ros2 service call /autopilot/go_relative ...
```

```
"{dx: 1.0, dy: 0.0, dyaw_deg: 0.0}"

# Терминал 2 (сразу же):
ros2 service call /autopilot/go_relative ...
"{dx: 0.0, dy: 0.0, dyaw_deg: 90.0}"
```

Гипотеза: Автопилот обрабатывает сервисы в одном потоке (с `lock`). Победит тот, кто первый захватил `lock`. Второй либо дождётся освобождения (если сервер в MOVING — не примет новую команду), либо получит отказ.

Проверьте: Запустите оба вызова и посмотрите ответы.

5. Выводы по занятию

1. **Сервис — это запрос-ответ.** В отличие от топика, клиент получает подтверждение, что команда принята.
2. **Три сервиса автопилота:** `go_time` (движение по времени), `go_relative` (относительное перемещение), `stop` (экстренная остановка).
3. **Вызов из командной строки:** `ros2 service call /autopilot/go_relative ...` — быстрый способ управлять роботом без написания кода.
4. **Безопасность встроена:** watchdog по heartbeat, детектор застревания, таймауты выполнения.
5. **`go_relative` точнее `go_time`** благодаря обратной связи по одометрии.

Связь со следующим занятием: Из командной строки вызывать сервисы удобно для тестов, но настоящие миссии требуют программного вызова. На следующем занятии мы напишем **первый Python-клиент**, который вызывает `go_time` и шлёт heartbeat.

Домашнее задание:

- **Базовое**: Вызовите все три сервиса из командной строки. Запишите ответы.
- **Продвинутое**: Вызовите `go_relative` с ненулевыми `dx` и `dy` одновременно. Куда поехал робот?
- **Исследовательское**: Проведите эксперимент по сравнению точности `go_time` vs `go_relative`. Запишите результаты.

Занятие 27: Первый клиент: вызов go_time из Python-ноды, heartbeat

Содержание занятия: Программный вызов сервиса. Асинхронные запросы и сторожевой таймер

Цели занятия

- **Образовательная**: Научиться создавать Python-ноду с клиентом сервиса `go_time`. Освоить асинхронный вызов `call_async` и публикацию heartbeat.

- **Развивающая**: Понять разницу между синхронным и асинхронным вызовом. Осознать роль heartbeat как механизма безопасности.
- **Воспитательная**: Привить культуру безопасного программирования: если программа управляет роботом, она должна непрерывно подтверждать, что «жива».

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Python-клиент для go_time (40 мин)**
   - Структура ноды с клиентом сервиса.
   - Асинхронный вызов `call_async`.
   - Публикация heartbeat (пульс).
   - Программа: движение с разными скоростями.
3. **Уровень «Углублённый»: Асинхронность и watchdog (25 мин)**
   - Синхронный vs асинхронный вызов.
   - `Future` в ROS2.
   - Heartbeat как watchdog (сторожевой таймер).
4. **Уровень «Исследовательский»: Задержка и частота heartbeat (15 мин)**
   - Измерение задержки.
   - Эксперимент с частотой heartbeat.
5. **Подведение итогов (5 мин)**

1. Введение: От командной строки — к коду

На прошлом занятии мы вызывали сервисы автопилота из командной строки. Это удобно для тестов, но настоящая программа должна делать это сама.

Сегодня мы напишем **первый Python-клиент** для сервиса `go_time`. Программа будет запускать робота на заданное время и одновременно **слать heartbeat** — сигнал «я жива». Если программа зависнет, автопилот остановит робота.

2. Уровень «База»: Python-клиент для go_time

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём ноду с клиентом и heartbeat.

2.1. Как устроен клиент сервиса в ROS2

Чтобы вызвать сервис из Python-ноды, нужно:

1. Создать **клиент** (`create_client`) для нужного типа сервиса.
2. Дождаться, пока сервер станет доступен (`wait_for_service`).

3. Создать **запрос** (`Request`) и заполнить поля.
4. Вызвать сервис: **синхронно** (`call`) или **асинхронно** (`call_async`).
5. Получить **ответ** (`Response`) и проверить `success` .

Мы будем использовать **асинхронный** вызов, чтобы программа не «зависала» в ожидании ответа.

2.2. Скрипт: go_time_client.py

Создайте файл:

```
nano ~/ws/rbt/scripts/go_time_client.py
```

```
#!/usr/bin/env python3
"""Первый клиент автопилота: вызов go_time с heartbeat (время ROS2)."""
import rclpy
from rclpy.node import Node
from ifra_mol_1.srv import AutopilotTime
from std_msgs.msg import Empty

class GoTimeClient(Node):
    def __init__(self):
        super().__init__('go_time_client')

        # Клиент сервиса go_time
        self.cli = self.create_client(AutopilotTime, '/autopilot/go_time')

        # Издатель heartbeat
        self.hb_pub = self.create_publisher(Empty, '/autopilot/
heartbeat', 10)

        # Таймер для heartbeat (каждые 0.5 секунды)
        self.hb_timer = self.create_timer(0.5, self.send_heartbeat)

        self.get_logger().info('Ожидание сервиса go_time ... ')
        while (not self.cli.wait_for_service(timeout_sec=1.0)) and rclpy.o
k():
            self.get_logger().info('Сервис ещё не готов ... ')
        self.get_logger().info('Сервис go_time доступен!')

    def send_heartbeat(self):
        """Отправляет heartbeat — сигнал 'я жива'."""
        self.hb_pub.publish(Empty())

    def call_go_time(self, vx, vy, omega, duration):
        """Вызывает сервис go_time асинхронно."""
        req = AutopilotTime.Request()
        req.vx = vx
        req.vy = vy
        req.omega = omega
        req.duration = duration

        self.get_logger().info(f'Отправка: vx={vx}, duration={duration}c')
        future = self.cli.call_async(req)
        return future
```

```
    def wait_seconds(self, seconds):
        """Ждёт заданное время, не блокируя обработку сообщений."""
        start = self.get_clock().now()
        while rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.05)
            elapsed = (self.get_clock().now() - start).nanoseconds / 1e9
            if elapsed >= seconds:
                break

def main():
    rclpy.init()
    node = GoTimeClient()


    print("Тест сервиса go_time с heartbeat ")
    print("Робот поедет 3 раза с разными скоростями")
    print("Нажмите Enter ... ")
    input()

    node.wait_seconds(0.1)

    # Тест 1: медленно вперёд 2 секунды
    print("\n── Тест 1: vx=0.1, 2 секунды ──")
    future1 = node.call_go_time(0.1, 0.0, 0.0, 2.0)

    # Ждём 3 секунды (2 с движения + запас)
    node.wait_seconds(3.0)

    if future1.done():
        resp = future1.result()
        print(f"Ответ: success={resp.success}, {resp.status_message}")

    # Тест 2: быстрее вперёд 1 секунду
    print("\n── Тест 2: vx=0.3, 1 секунда ──")
    future2 = node.call_go_time(0.3, 0.0, 0.0, 1.0)

    node.wait_seconds(2.0)

    if future2.done():
        resp = future2.result()
        print(f"Ответ: success={resp.success}, {resp.status_message}")

    # Тест 3: поворот на месте
    print("\n── Тест 3: omega=0.5, 2 секунды ──")
    future3 = node.call_go_time(0.0, 0.0, 0.5, 2.0)

    node.wait_seconds(3.0)

    if future3.done():
        resp = future3.result()
        print(f"Ответ: success={resp.success}, {resp.status_message}")

    print("\nВсе тесты завершены.")
    node.destroy_node()
    rclpy.shutdown()
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/go_time_client.py
```

2.3. Наблюдение за работой

В отдельном терминале следите за телеметрией:

```
ros2 topic echo /autopilot/telemetry --field status
```

Вы увидите смену статусов:

```
IDLE
MOVING
MOVING
 ...
SUCCESS
IDLE
```

Heartbeat работает в фоне: таймер каждые 0.5 секунды публикует `Empty` в `/autopilot/heartbeat`. Пока heartbeat идёт, автопилот знает, что клиент жив.

3. Уровень «Углублённый»: Асинхронность и watchdog

(Что под капотом)

Разбираемся в механизмах работы.

3.1. Синхронный vs асинхронный вызов

```
# СИНХРОННЫЙ (блокирующий) — программа ЗАМИРАЕТ, пока сервер не ответит
response = cli.call(request)  # может висеть секунды!

# АСИНХРОННЫЙ (неблокирующий) — программа продолжает работать
future = cli.call_async(request)  # мгновенно возвращает Future
# ... делаем другие дела (heartbeat, проверка статуса) ...
if future.done():  # проверяем, готов ли ответ
    response = future.result()
```

Почему асинхронно? Если вызвать `call` синхронно, программа замрёт. Heartbeat перестанет отправляться, и через 1.5 секунды автопилот экстренно остановит робота!

3.2. Future: обещание результата

`call_async` возвращает объект **Future**. Это «контейнер», в котором **когда-нибудь** появится результат.

- `future.done()` — `True`, если ответ получен.

- `future.result()` — получить ответ (если не готов — ждёт!). Вызывайте только после `done() == True`.

Аналогия: Future — это квитанция из химчистки. Вы отдали рубашку, получили квитанцию. Можете заниматься другими делами. Когда по квитанции рубашка готова — забираете.

3.3. Heartbeat как watchdog

Watchdog — это механизм, который следит, чтобы программа не зависла.

В нашем случае:

- Клиент шлёт heartbeat каждые 0.5 с.
- Автопилот проверяет: если heartbeat не приходил **1.5 секунды** — считает, что клиент завис, и останавливает робота.

Почему heartbeat в отдельном таймере? Таймер работает независимо от основного кода. Даже если основная логика «задумалась», таймер продолжает слать heartbeat. Это гарантирует, что кратковременные задержки не вызовут ложное срабатывание.

4. Уровень «Исследовательский»: Задержка и частота heartbeat

(Прорыв за границы)

Эксперименты с производительностью.

4.1. Измерение задержки между вызовом go_time и началом движения

Методика:

1. Засеките время `t0` непосредственно перед `call_async`.
2. В телеметрии найдите первое появление статуса `MOVING` и зафиксируйте `t1`.
3. Задержка = `t1 - t0`.

Ожидаемый результат: 50–100 мс (время на передачу запроса, обработку сервером, публикацию телеметрии).

Модифицируйте скрипт для автоматического замера:

```
import time
t0 = time.time()
future = node.call_go_time(0.1, 0.0, 0.0, 2.0)
# ждём ответ ...
print(f"Ответ получен через {time.time() - t0:.3f} с")
```

4.2. Мини-исследование: частота heartbeat и срабатывание watchdog

Вопрос: При какой частоте heartbeat автопилот ещё не глушит моторы, а при какой уже срабатывает?

Эксперимент:

1. Установите таймер heartbeat на 1.0 с. Запустите движение. Watchdog (1.5 с) не должен сработать.
2. Установите таймер heartbeat на 2.0 с. Watchdog **сработает** — робот остановится через 1.5 с.

Вывод: Период heartbeat должен быть **меньше** таймаута watchdog (1.5 с). Рекомендуется 0.5 с — двойной запас.

5. Выводы по занятию

1. **Клиент сервиса** создаётся через `create_client` и вызывается через `call_async`.
2. **Асинхронный вызов** не блокирует программу — heartbeat продолжает слаться.
3. **Future** — это «обещание» результата. Проверяйте `done()` перед `result()`.
4. **Heartbeat** — обязательный элемент безопасности. Без него автопилот остановит робота через 1.5 с.
5. **Таймер** для heartbeat надёжнее, чем ручная отправка в цикле.

Связь со следующим занятием: `go_time` — это движение вслепую. На следующем занятии мы освоим **`go_relative`** — движение с контролем по одометрии, и научимся ждать статус `SUCCESS` через телеметрию.

Домашнее задание:

- **Базовое**: Запустите `go_time_client.py`. Убедитесь, что heartbeat работает (проверьте через `ros2 topic echo /autopilot/heartbeat`).
- **Продвинутое**: Модифицируйте скрипт: добавьте крабовое движение (`vy`) и проверьте, едет ли робот боком.
- **Исследовательское**: Проведите эксперимент с разной частотой heartbeat (0.3, 0.7, 1.0, 1.5, 2.0 с). При какой частоте watchdog срабатывает?

Занятие 28: Сервис go_relative и чтение телеметрии

Содержание занятия: Относительное перемещение с обратной связью. Ожидание статуса через телеметрию

Цели занятия

- **Образовательная**: Научиться вызывать сервис `go_relative` из Python-ноды и отслеживать выполнение команды через телеметрию.
- **Развивающая**: Понять преимущество контроля по статусу перед «слепым» ожиданием по времени.
- **Воспитательная**: Сформировать привычку проверять результат команды, а не надеяться, что «всё пройдёт как задумано».

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**

2. **Уровень «База»: Клиент go_relative с ожиданием SUCCESS (45 мин)**
    - Параметры `dx`, `dy`, `dyaw_deg`.
    - Подписка на `/autopilot/telemetry`.
    - Программа: проехать 1 метр, дождаться SUCCESS, повернуть на 90°.
3. **Уровень «Углублённый»: Как автопилот считает целевую точку (20 мин)**
    - Проекция `dx`, `dy` на глобальные координаты через yaw.
    - Почему важно ждать SUCCESS, а не `sleep(duration)`.
4. **Уровень «Исследовательский»: Разброс и скорость (15 мин)**
    - Эксперимент: 5 заездов по 1 метру.
    - Зависимость точности от скорости.
5. **Подведение итогов (5 мин)**

1. Введение: Умное движение вместо движения по таймеру

На прошлом занятии мы использовали `go_time` — робот едет заданное время и останавливается. Это просто, но неточно:

- Если колесо проскальзывает — робот недоезжает.
- Если батарея села — скорость ниже, дистанция меньше.

Сервис `go_relative` работает иначе: вы говорите **«проедь 1 метр»**, и автопилот сам едет, пока одометрия не покажет, что метр пройден. Это **движение с обратной связью**.

А чтобы знать, когда команда выполнена, мы читаем **телеметрию** — топик, в котором автопилот сообщает своё состояние.

2. Уровень «База»: Клиент go_relative с ожиданием SUCCESS

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем ноду: клиент + подписчик телеметрии.

2.1. Параметры go_relative

Сервис `go_relative` принимает три числа:

- `dx` — метры **вперёд** (относительно текущего курса робота).
- `dy` — метры **влево** (крабовое движение меканум-базы).
- `dyaw_deg` — градусы **поворота влево** на месте.

Примеры:

Команда	Что делает робот
`dx=1, dy=0, dyaw=0`	Едет вперёд 1 метр
`dx=0, dy=0, dyaw=90`	Поворачивает влево на 90°
`dx=0.5, dy=0.3, dyaw=0`	Едет по диагонали (вперёд-влево)
`dx=0, dy=0, dyaw=-45`	Поворачивает вправо на 45°

2.2. Скрипт: go_relative_client.py

Создайте файл:

```
nano ~/ws/rbt/scripts/go_relative_client.py
```

```
#!/usr/bin/env python3
"""Клиент go_relative: движение с ожиданием статуса SUCCESS (время
ROS2)."""
import sys
sys.path.append('/home/sh')
import rclpy
from rclpy.node import Node
from ifra_mol_1.srv import AutopilotRelative
from ifra_mol_1.msg import AutopilotTelemetry
from std_msgs.msg import Empty
from std_srvs.srv import Trigger

class GoRelativeClient(Node):
    def __init__(self):
        super().__init__('go_relative_client')

        # Клиенты сервисов
        self.cli = self.create_client(AutopilotRelative, '/autopilot/
go_relative')
        self.stop_cli = self.create_client(Trigger, '/autopilot/stop')

        # Подписчик телеметрии
        self.status = "IDLE"
        self.sub = self.create_subscription(
            AutopilotTelemetry,
            '/autopilot/telemetry',
            self._on_telemetry,
            10
        )

        # Издатель heartbeat — 2 Гц
        self.hb_pub = self.create_publisher(Empty, '/autopilot/
heartbeat', 10)
        self.hb_timer = self.create_timer(0.5, self._send_hb)

        self.get_logger().info('Ожидание сервисов ... ')
        while not self.cli.wait_for_service(timeout_sec=1.0) and rclpy.ok(
):
            self.get_logger().info('go_relative ещё не готов ... ')
        while not self.stop_cli.wait_for_service(timeout_sec=1.0) and rclp
```

```
y.ok():
            self.get_logger().info('stop ещё не готов ... ')
        self.get_logger().info('Сервисы доступны!')

        # Сброс автопилота в IDLE (если он в ABORTED_TIMEOUT)
        self._reset_autopilot()

    def _on_telemetry(self, msg):
        self.status = msg.status

    def _send_hb(self):
        self.hb_pub.publish(Empty())

    def _elapsed(self, start):
        return (self.get_clock().now() - start).nanoseconds / 1e9

    def _reset_autopilot(self):
        """Вызывает stop для сброса автопилота в IDLE."""
        self.get_logger().info('Сброс автопилота в IDLE ... ')
        future = self.stop_cli.call_async(Trigger.Request())
        start = self.get_clock().now()
        while not future.done() and self._elapsed(start) < 3.0 and rclpy.o
k():
            rclpy.spin_once(self, timeout_sec=0.02)
        if future.done():
            resp = future.result()
            self.get_logger().info(f'Сброс: {resp.message}')
        # Ждём, пока статус станет IDLE
        start = self.get_clock().now()
        while self._elapsed(start) < 3.0 and rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.02)
            if self.status == "IDLE":
                self.get_logger().info('Автопилот в IDLE, готов к
работе.')
                return
        self.get_logger().warn(f'Не удалось дождаться IDLE, текущий
статус: {self.status}')

    def call_and_wait(self, dx, dy, dyaw_deg, timeout=30.0):
        """Вызывает go_relative и ждёт статус SUCCESS или ошибку."""
        req = AutopilotRelative.Request()
        req.dx = dx
        req.dy = dy
        req.dyaw_deg = dyaw_deg

        self.status = "IDLE"

        self.get_logger().info(f'Отправка: dx={dx}, dy={dy}, dyaw={dyaw_de
g}°')
        future = self.cli.call_async(req)

        # Ждём ответа сервиса
        start = self.get_clock().now()
        while not future.done() and self._elapsed(start) < 5.0 and rclpy.o
k():
            rclpy.spin_once(self, timeout_sec=0.02)
```

```
        if not future.done():
            self.get_logger().error('Сервис не ответил!')
            return False

        resp = future.result()
        if not resp.success:
            self.get_logger().error(f'Отклонено: {resp.status_message}')
            return False

        self.get_logger().info('Принято. Ожидание начала движения...')

        # Ждём MOVING
        start = self.get_clock().now()
        while self._elapsed(start) < 3.0 and rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.02)
            if self.status == "MOVING":
                break
            elif self.status in ("ABORTED_STUCK", "ABORTED_TIMEOUT"):
                self.get_logger().error(f'Ошибка при старте:
{self.status}')
                return False

        if self.status ≠ "MOVING":
            self.get_logger().error(f'Робот не начал движение! Статус: {se
lf.status}')
            return False

        self.get_logger().info('Робот движется. Ожидание завершения...')

        # Ждём SUCCESS
        start = self.get_clock().now()
        while self._elapsed(start) < timeout and rclpy.ok():
            rclpy.spin_once(self, timeout_sec=0.02)
            if self.status == "SUCCESS":
                self.get_logger().info('Команда выполнена успешно!')
                return True
            elif self.status in ("ABORTED_STUCK", "ABORTED_TIMEOUT"):
                self.get_logger().error(f'Ошибка: {self.status}')
                return False

        self.get_logger().error('Таймаут выполнения!')
        return False


def main():
    rclpy.init()
    node = GoRelativeClient()

    print("Программа: проехать 1 м вперёд, затем повернуть на 90°")
    print("Нажмите Enter...")
    input()

    # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and node.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(node, timeout_sec=0.02)
```

```
    if node.call_and_wait(1.0, 0.0, 0.0):
        print("√ 1 метр пройден")
    else:
        print("✗ Ошибка при движении вперёд")
        node.destroy_node()
        rclpy.shutdown()
        return

    start = node.get_clock().now()
    while node._elapsed(start) < 0.5 and rclpy.ok():
        rclpy.spin_once(node, timeout_sec=0.02)

    if node.call_and_wait(0.0, 0.0, 90.0):
        print("√ Поворот на 90° выполнен")
    else:
        print("✗ Ошибка при повороте")

    print("Программа завершена.")
    node.destroy_node()
    rclpy.shutdown()


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

Запуск:

```
python3 ~/ws/rbt/scripts/go_relative_client.py
```

2.3. Что происходит при запуске

1. Нода создаёт клиента `go_relative` и подписывается на телеметрию.
2. `call_and_wait(1.0, 0.0, 0.0)` отправляет запрос и **ждёт**.
3. Автопилот принимает запрос, отвечает `success=True`, начинает движение.
4. Нода в цикле читает телеметрию: `MOVING` → `MOVING` → ... → `SUCCESS`.
5. Когда статус стал `SUCCESS` — функция возвращает `True`, программа идёт дальше.
6. Если статус стал `ABORTED_STUCK` (робот застрял) — возвращает `False`, программа останавливается.

Ключевое отличие от `sleep(duration)`: мы не гадаем, сколько времени займёт движение. Мы **ждём факта** завершения.

3. Уровень «Углублённый»: Как автопилот считает целевую точку

(Что под капотом)

Разбираем математику и логику безопасности.

3.1. Проекция относительных координат на глобальные

Когда вы вызываете `go_relative(dx=1, dy=0, dyaw=0)`, автопилот:

1. Запоминает текущую позицию `(start_x, start_y)` и курс `start_yaw`.
2. Вычисляет **целевую точку** в глобальных координатах:

```
target_x = start_x + dx * cos(start_yaw) - dy * sin(start_yaw)
target_y = start_y + dx * sin(start_yaw) + dy * cos(start_yaw)
target_yaw = start_yaw + dyaw_deg (в радианах)
```

3. Включает моторы и едет, пока `(current_x, current_y)` не окажется в допуске от `(target_x, target_y)`.

Почему через косинус/синус? Потому что `dx` и `dy` заданы **относительно робота**, а не относительно комнаты. Если робот стоит под углом 45°, команда «вперёд 1 метр» означает движение под 45° в глобальных координатах.

3.2. Почему важно ждать SUCCESS, а не sleep

Сценарий: робот пытается проехать 1 метр, но упирается в препятствие.

- **С `sleep(5.0)`:** программа ждёт 5 секунд и идёт дальше. Робот стоит на месте (или буксует). Следующая команда (поворот) выполняется, но робот уже не там, где ожидалось.
- **С ожиданием SUCCESS:** через 3 секунды автопилот обнаруживает, что позиция не меняется (stuck detector). Статус становится `ABORTED_STUCK`. Программа видит это и **не выполняет** следующую команду. Она может вызвать `stop` и сообщить об ошибке.

Вывод: `sleep` предполагает, что всё идёт по плану. Ожидание статуса **проверяет** это.

4. Уровень «Исследовательский»: Разброс и скорость

(Прорыв за границы)

Эксперименты с точностью.

4.1. Эксперимент: разброс финальной позиции

Задача: 5 раз вызвать `go_relative(dx=1, dy=0, dyaw=0)`, каждый раз замеряя реальную позицию рулеткой.

Заполните таблицу:

Попытка	Цель X	Реальный X	Ошибка X	Цель Y	Реальный Y	Ошибка Y
1	1.0			0.0		
2	1.0			0.0		
...						

Постройте график: точки (реальный X, реальный Y) на плоскости. Насколько плотно они группируются?

4.2. Мини-исследование: зависит ли точность от скорости

Автопилот использует фиксированные скорости (0.2 м/с для `dx`, 0.4 рад/с для `dyaw`).

Гипотеза: Если бы скорость была выше, робот проскальзывал бы сильнее, и точность упала.

Проверка (требует модификации автопилота — под руководством преподавателя): измените `tgt_vx` в `_srv_go_relative_callback` на 0.4 и повторите эксперимент. Сравните разброс.

Ожидаемый результат: при большей скорости разброс увеличивается. Инженерный компромисс: скорость vs точность.

5. Выводы по занятию

1. **`go_relative` — движение с обратной связью.** Робот едет, пока не достигнет цели, а не заданное время.
2. **Телеметрия — источник правды.** Статус `SUCCESS` означает, что автопилот подтвердил выполнение. `ABORTED_STUCK` — что-то пошло не так.
3. **Ждать статус надёжнее, чем `sleep`.** Статус отражает реальность, `sleep` — только предположение.
4. **Точность зависит от скорости.** Быстрее — менее точно. Это фундаментальный компромисс.

Связь со следующим занятием: Теперь у нас есть два сервиса (`go_time`, `go_relative`). На следующем занятии мы добавим третий — `stop` для экстренной остановки — и познакомимся с **базовым классом `RobotDriver`**, объединяющим все три клиента и телеметрию.

Домашнее задание:

- **Базовое**: Запустите `go_relative_client.py`. Убедитесь, что робот проезжает 1 м и поворачивает на 90°.
- **Продвинутое**: Модифицируйте программу для проезда треугольника (3 стороны по 0.5 м, углы 120°).
- **Исследовательское**: Проведите 5 заездов по 1 м, замерьте разброс, постройте график.

Занятие 29: Сервис stop и базовый класс RobotDriver

Содержание занятия: Экстренная остановка и единый интерфейс к автопилоту из библиотеки driver_lib

Цели занятия

- **Образовательная**: Научиться пользоваться сервисом `stop` и готовым классом `RobotDriver` из библиотеки `/home/sh/driver_lib.py`.
- **Развивающая**: Понять принцип инкапсуляции: библиотека прячет детали ROS2 за простыми методами.
- **Воспитательная**: Привить привычку использовать готовые проверенные компоненты, а не писать всё с нуля.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Знакомство с RobotDriver и stop (50 мин)**
   - Импорт `RobotDriver` из `driver_lib`.
   - Методы: `move_time()`, `move_relative()`, `stop()`, `get_status()`, `get_pose()`.
   - Тест: разгон через `move_time`, остановка через `stop`.
3. **Уровень «Углублённый»: Что внутри RobotDriver (20 мин)**
   - Три клиента, телеметрия, heartbeat, автосброс в IDLE.
   - Почему `stop` — отдельный сервис, а не `move_time(vx=0)`.
4. **Уровень «Исследовательский»: Тормозной путь (10 мин)**
   - Измерение тормозного пути на разных скоростях.
   - Сравнение `stop` и `move_time(vx=0)`.
5. **Подведение итогов (5 мин)**

1. Введение: Не изобретаем велосипед

На прошлых занятиях мы вручную создавали клиентов, подписчиков, таймеры heartbeat. Это было нужно для понимания, как всё работает внутри.

Теперь мы переходим на следующий уровень: используем **готовую библиотеку** `/home/sh/driver_lib.py`, которая содержит класс `RobotDriver`. Он уже умеет всё то, что мы писали вручную:

- Три клиента (`go_time`, `go_relative`, `stop`).
- Подписка на телеметрию (`/autopilot/telemetry`).
- Heartbeat каждые 0.5 секунды.
- Автоматический сброс автопилота в `IDLE` при старте.

Наша задача — научиться **использовать** этот класс.

2. Уровень «База»: Знакомство с RobotDriver и stop

(Инструмент, навык, интерфейс)

Обязательный уровень. Импортируем готовый класс и тестируем аварийную остановку.

2.1. Импорт RobotDriver из библиотеки

Класс лежит в `/opt/drv/driver_lib.py`. Чтобы его импортировать, пишем:

```
from driver_lib import RobotDriver
```

Если возникает ошибка внимательно читаем занятие 13.

Методы RobotDriver:

Метод	Что делает
`move_time(vx, vy, omega, duration)`	Движение по времени. Возвращает `Future`
`move_relative(dx, dy, dyaw_deg)`	Относительное перемещение. Возвращает `Future`
`stop()`	Экстренная остановка. Возвращает `Future`
`get_status()`	Текущий статус автопилота (`IDLE`, `MOVING`, `SUCCESS`, ...)
`get_pose()`	Текущая позиция `(x, y, yaw)`
`send_heartbeat()`	Явная отправка heartbeat
`is_connected()`	Получена ли телеметрия

2.2. Тест: разгон через move_time, остановка через stop

Создайте файл:

```
nano ~/ws/rbt/scripts/test_stop.py
```

```
#!/usr/bin/env python3
"""Тест драйвера: разгон через move_time, остановка через stop."""
import sys
import rclpy
from driver_lib import RobotDriver

def main():
    rclpy.init()
    driver = RobotDriver('test_stop')

    print("Тест: разгон через move_time, остановка через stop")
    print("Робот поедет вперёд. Через 2 секунды — экстренный stop.")
    print("Нажмите Enter ... ")
    input()
```

```
        # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and driver.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(driver , timeout_sec=0.02)
    start =  driver.get_clock().now().nanoseconds
    while (driver.get_clock().now().nanoseconds - start) / 1e9 < 0.1:
        rclpy.spin_once(driver , timeout_sec=0.02)

    # Разгон: едем вперёд 5 секунд
    print("\nЗапуск движения (move_time: 5 секунд)...")
    driver.move_time(0.2, 0.0, 0.0, 5.0)

    print("Робот едет...")
    stopped = False
    start = driver.get_clock().now()

    while rclpy.ok():
        # Главный цикл: шлём heartbeat (я жив!), обрабатываем сообщения
        driver.send_heartbeat()
        rclpy.spin_once(driver, timeout_sec=0.5)

        elapsed = (driver.get_clock().now() - start).nanoseconds / 1e9

        # Через 2 секунды — экстренный stop (один раз)
        if elapsed ≥ 2.0 and not stopped:
            print("ЭКСТРЕННЫЙ СТОП!")
            driver.stop()
            stopped = True

        # После stop ждём ещё секунду и выходим
        if stopped and elapsed ≥ 3.0:
            break

    print(f"Финальный статус: {driver.get_status()}")
    print("Тест завершён.")
    driver.destroy_node()
    rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/test_stop.py
```

2.3. Наблюдение

1. Робот начинает ехать вперёд.
2. Через 2 секунды программа вызывает `stop`.
3. Робот резко останавливается.
4. Статус становится `IDLE`.

Сравните с прошлым занятием: этот скрипт в 3 раза короче, потому что вся сложность спрятана в `RobotDriver`.

3. Уровень «Углублённый»: Что внутри RobotDriver

(Что под капотом)

Заглянем в библиотеку и поймём, почему `stop` — особенный.

3.1. Что делает RobotDriver при создании

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import Empty
from std_srvs.srv import Trigger
#import time

from ifra_mol_1.srv import AutopilotTime, AutopilotRelative
from ifra_mol_1.msg import AutopilotTelemetry


#
================================================================================
====
# 1. БАЗОВЫЙ КЛАСС-КЛИЕНТ АВТОПИЛОТА (DRIVER)
#
================================================================================
====

class RobotDriver(Node):
    def __init__(self, node_name: str = 'robot_driver'):
        super().__init__(node_name)

        # Сервис-клиенты
        self._cli_time = self.create_client(AutopilotTime, 'autopilot/
go_time')
        self._cli_relative = self.create_client(AutopilotRelative, 'autopi
lot/go_relative')
        self._cli_stop = self.create_client(Trigger, 'autopilot/stop')

        # Подписчик телеметрии
        self._telemetry_sub = self.create_subscription(
            AutopilotTelemetry,
            '/autopilot/telemetry',
            self._on_telemetry,
            10
        )
        self._current_status = "IDLE"
        self._current_x = 0.0
        self._current_y = 0.0
        self._current_yaw = 0.0
        self._has_telemetry = False

        # Издатель heartbeat
        self._hb_pub = self.create_publisher(Empty, '/autopilot/
heartbeat', 10)
```

```
        # Блокирующее ожидание готовности сервисов
        self.get_logger().info("Ожидание автопилота...")
        while not self._cli_time.wait_for_service(timeout_sec=1.0) and rcl
py.ok():
            self.get_logger().info("Повторное ожидание...")
        self.get_logger().info("Автопилот подключён.")
```

Всё это происходит автоматически. Мы просто пишем `driver = RobotDriver()`, и через пару секунд драйвер готов к работе.

3.2. Почему stop — отдельный сервис

Можно было бы останавливать командой `move_time(vx=0, vy=0, omega=0, duration=0.1)`. Но:

`stop`	`move_time(vx=0)`
Немедленная реакция	Немедленная реакция
Сбрасывает статус в IDLE	Не сбрасывает статус в IDLE
Не зависит от heartbeat	Зависит (watchdog)
Специальный приоритет	Обычная команда

Аналогия: `stop` — это красная кнопка пожарной сигнализации. `move_time(vx=0)` — это записка «уважаемый робот, пожалуйста, остановитесь, если не затруднит».

4. Уровень «Исследовательский»: Тормозной путь

(Прорыв за границы)

Эксперименты с аварийной остановкой.

4.1. Измерение тормозного пути

Методика:

1. Запустите `move_time` с разными скоростями (0.1, 0.2, 0.3, 0.4).
2. Через 1 секунду вызовите `stop`.
3. Замерьте рулеткой расстояние от точки вызова до полной остановки.

Скорость, м/с	Тормозной путь, см
0.1	
0.2	
0.3	
0.4	

Постройте график: ось X — скорость, ось Y — тормозной путь.

4.2. Мини-исследование: stop vs move_time(vx=0)

Вопрос: Что быстрее останавливает робота — `stop()` или `move_time(vx=0, duration=0.1)`?

Проверка:

1. Разгон до 0.2 м/с.
2. Способ А: `stop()`. Замерить тормозной путь.
3. Способ Б: `move_time(vx=0, duration=0.1)`. Замерить тормозной путь.

Ожидаемый результат: физически тормозной путь одинаков (робот не умеет тормозить быстрее, чем позволяет механика). Но `stop` надёжнее, потому что гарантированно сбрасывает состояние автопилота.

5. Выводы по занятию

1. **`RobotDriver` из `driver_lib` — готовый интерфейс к роботу.** Не нужно писать клиентов вручную.
2. **`stop()` — экстренная остановка.** Немедленная, сбрасывает автопилот в `IDLE`.
3. **Инкапсуляция экономит код.** 10 строк вместо 100. Меньше кода — меньше ошибок.
4. **Библиотека — результат нашей работы.** Мы прошли путь от ручных `Twist` до высокоуровневых сервисов, а теперь используем готовый драйвер.

Связь со следующим занятием: Теперь у нас есть `RobotDriver` с методами `move_relative`, `stop` и `get_status`. На следующем занятии мы напишем **проезд квадрата** с проверкой статуса после каждого манёвра и обработкой ошибок.

Домашнее задание:

- **Базовое**: Запустите `test_stop.py`. Убедитесь, что робот останавливается.
- **Продвинутое**: Модифицируйте скрипт: добавьте `move_relative` до вызова `move_time` для возврата в начало.
- **Исследовательское**: Проведите замеры тормозного пути для 4 скоростей. Постройте график.

Занятие 30: Движение по квадрату через последовательный вызов сервисов

Содержание занятия: Программный проезд квадрата с проверкой статуса и обработкой ошибок

Цели занятия

- **Образовательная**: Написать программу для проезда квадрата, используя `RobotDriver`, с ожиданием `SUCCESS` после каждого манёвра и обработкой ошибок.
- **Развивающая**: Понять ограничения линейного последовательного кода: почему добавление ветвлений и обработки ошибок делает его громоздким.

- **Воспитательная**: Подготовить мотивацию к переходу на деревья поведения.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Квадрат на RobotDriver (45 мин)**
   - Структура: 4 стороны + 4 поворота.
   - Ожидание `SUCCESS` после каждого `move_relative`.
   - Heartbeat в главном цикле.
   - Обработка `ABORTED` через `stop`.
3. **Уровень «Углублённый»: Проблемы последовательного кода (25 мин)**
   - Что будет без проверки статуса.
   - Почему добавление объезда препятствий ломает структуру.
4. **Уровень «Исследовательский»: Надёжность на разных покрытиях (10 мин)**
   - Эксперимент: 10 попыток квадрата.
   - Сравнение покрытий.
5. **Подведение итогов (5 мин)**

1. Введение: От ручного управления к автономной миссии

В блоке ROS2 мы уже проезжали квадрат — сначала «вслепую» по времени, потом с контролем по одометрии. Но там мы сами публиковали `Twist` и читали `/odom`.

Теперь у нас есть **автопилот** и **RobotDriver**. Квадрат пишется гораздо проще:

- `driver.move_relative(1, 0, 0)` — проехать метр вперёд.
- Ждать `SUCCESS`.
- `driver.move_relative(0, 0, 90)` — повернуть на 90°.
- Повторить 4 раза.

Но мы добавим **проверку ошибок**: если робот застрял — остановка и диагностика, а не попытка ехать дальше в стену.

2. Уровень «База»: Квадрат на RobotDriver

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем программу с контролем статуса.

2.1. Структура программы

Квадрат — это 8 манёвров: 4 прямых и 4 поворота.

Для каждого манёвра:

1. Вызвать `driver.move_relative( ... )`.
2. Ждать в цикле `spin_once`, пока статус не станет `SUCCESS` или `ABORTED_*`.
3. Если `ABORTED` — вызвать `driver.stop()` и завершить программу.

Heartbeat отправляется на каждой итерации цикла ожидания.

2.2. Скрипт square_services.py

Создайте файл:

```
nano ~/ws/rbt/scripts/square_services.py
```

```
#!/usr/bin/env python3
"""Проезд квадрата через RobotDriver с трехфазным ожиданием."""
import sys
import rclpy
from driver_lib import RobotDriver

def wait_for_result(driver, future, timeout=30.0):
    """Двухфазное ожидание: ответ сервиса → MOVING → SUCCESS."""
    # Фаза 1: ждём ответа сервиса
    start = driver.get_clock().now()
    while not future.done() and rclpy.ok():
        driver.send_heartbeat()
        rclpy.spin_once(driver, timeout_sec=0.02)
        if (driver.get_clock().now() - start).nanoseconds / 1e9 > 5.0:
            driver.get_logger().error('Сервис не ответил!')
            return False

    resp = future.result()
    if not resp.success:
        driver.get_logger().error(f'Отклонено: {resp.status_message}')
        return False

    # Фаза 2: ждём MOVING (робот реально начал движение)
    start = driver.get_clock().now()
    while rclpy.ok():
        driver.send_heartbeat()
        rclpy.spin_once(driver, timeout_sec=0.02)
        status = driver.get_status()
        if status == "MOVING":
            break
        if status in ("ABORTED_STUCK", "ABORTED_TIMEOUT"):
            driver.get_logger().error(f'Ошибка при старте: {status}')
            return False
        if (driver.get_clock().now() - start).nanoseconds / 1e9 > 3.0:
            driver.get_logger().error(f'Робот не начал движение! Статус:
{status}')
            return False

    # Фаза 3: ждём SUCCESS
    start = driver.get_clock().now()
```

```
    while rclpy.ok():
        driver.send_heartbeat()
        rclpy.spin_once(driver, timeout_sec=0.02)
        status = driver.get_status()
        if status == "SUCCESS":
            return True
        if status in ("ABORTED_STUCK", "ABORTED_TIMEOUT"):
            driver.get_logger().error(f'Манёвр прерван: {status}')
            return False
        if (driver.get_clock().now() - start).nanoseconds / 1e9 > timeout:
            driver.get_logger().error('Таймаут манёвра!')
            return False

def main():
    rclpy.init()
    driver = RobotDriver('square_services')
        # ── ДАЕМ СЕТИ DDS ВРЕМЯ НА СИНХРОНИЗАЦИЮ ──
    while rclpy.ok() and driver.get_clock().now().nanoseconds == 0:
        rclpy.spin_once(driver , timeout_sec=0.02)

    # Сброс автопилота в IDLE и ожидание heartbeat
    print("Сброс автопилота в IDLE ... ")
    driver.stop()
    start = driver.get_clock().now()
    while (driver.get_clock().now() - start).nanoseconds / 1e9 < 2.0 and r
clpy.ok():
        driver.send_heartbeat()
        rclpy.spin_once(driver, timeout_sec=0.02)
        if driver.get_status() == "IDLE":
            print("Автопилот готов (IDLE).")
            break

    print("Квадрат 1x1 м через сервисы автопилота")
    print("Нажмите Enter ... ")
    input()
    #Синхронизация после инпута
    start =  driver.get_clock().now().nanoseconds
    while (driver.get_clock().now().nanoseconds - start) / 1e9 < 0.1:
        rclpy.spin_once(driver , timeout_sec=0.02)

    maneuvers = [
        ("Сторона 1", 1.0, 0.0, 0.0),
        ("Поворот 1", 0.0, 0.0, 90.0),
        ("Сторона 2", 1.0, 0.0, 0.0),
        ("Поворот 2", 0.0, 0.0, 90.0),
        ("Сторона 3", 1.0, 0.0, 0.0),
        ("Поворот 3", 0.0, 0.0, 90.0),
        ("Сторона 4", 1.0, 0.0, 0.0),
        ("Поворот 4", 0.0, 0.0, 90.0),
    ]

    for name, dx, dy, dyaw in maneuvers:
        print(f"\n── {name} ──")
        future = driver.move_relative(dx, dy, dyaw)

        if wait_for_result(driver, future):
```

```
            print(f"√ {name} выполнен")
        else:
            print(f"✗ {name} провален! Экстренная остановка.")
            driver.stop()
            start = driver.get_clock().now()
            while (driver.get_clock().now() - start).nanoseconds / 1e9 < 1
.0 and rclpy.ok():
                driver.send_heartbeat()
                rclpy.spin_once(driver, timeout_sec=0.02)
            break

    print(f"\nФинальный статус: {driver.get_status()}")
    x, y, yaw = driver.get_pose()
    print(f"Позиция: x={x:.2f}, y={y:.2f}, yaw={yaw:.2f}")
    driver.destroy_node()
    rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/square_services.py
```

2.3. Наблюдение за выполнением

Программа выводит:

```
── Сторона 1 ──
√ Сторона 1 выполнен
── Поворот 1 ──
√ Поворот 1 выполнен
...
Финальный статус: SUCCESS
Позиция: x=0.03, y=-0.05, yaw=0.04
```

Если робот застревает:

```
── Сторона 2 ──
✗ Сторона 2 провален! Экстренная остановка.
```

3. Уровень «Углублённый»: Проблемы последовательного кода

(Что под капотом)

Почему этот подход не масштабируется.

3.1. Что будет без проверки статуса

Если убрать `wait_for_result` и просто делать `sleep(5)` после каждого манёвра:

- Робот застревает на первом метре.
- `sleep` истекает.

• Программа отправляет «поворот на 90°».

• Робот стоит на месте и крутится (хорошо, если не врежется).

• Программа продолжает слать команды, не подозревая о проблеме.

Проверка статуса — это обратная связь. Без неё программа слепа.

3.2. Почему это не масштабируется

Представьте, что мы хотим добавить:

• **Объезд препятствия:** если `ABORTED_STUCK` — отъехать назад, повернуть, попробовать снова.

• **Условный поворот:** если статус `SUCCESS` и впереди свободно — ехать прямо, иначе — повернуть.

В последовательном коде это требует `if-else` внутри цикла, флагов, дополнительных проверок. Код быстро становится запутанным.

Решение: деревья поведения. Они позволяют собирать сложную логику из готовых блоков, как конструктор. Об этом — следующие занятия.

4. Уровень «Исследовательский»: Надёжность на разных покрытиях

(Прорыв за границы)

Эксперимент по статистике отказов.

4.1. Эксперимент: 10 попыток квадрата

Методика:

1. Запустите `square_services.py` 10 раз подряд.
2. Каждый раз записывайте: успех/провал, на каком манёвре провал.

Таблица:

Попытка	Результат	Где провал
1		
2		
...		
10		

Посчитайте процент успешных проездов.

4.2. Сравнение покрытий

Проведите тот же эксперимент на другом покрытии (ковёр, плитка).

Гипотеза: На ковре процент отказов выше (больше трения → чаще `ABORTED_STUCK`).

Вывод: Надёжность автономного движения зависит от среды. Для ковра снизить скорость или повысить порог stuck detector'a.

5. Выводы по занятию

1. **`RobotDriver` упрощает код.** Квадрат пишется в 2 раза короче, чем с ручным `Twist`.
2. **Проверка статуса обязательна.** Без неё программа не знает, что робот застрял.
3. **Обработка `ABORTED` через `stop` — безопасность.** Лучше остановиться и сообщить об ошибке, чем ехать вслепую.
4. **Последовательный код плохо масштабируется.** Ветвления и повторы делают его громоздким. Нужен новый инструмент.

Связь со следующим занятием: Мы увидели ограничения линейного кода. На следующем занятии мы познакомимся с **деревьями поведения** — модульным подходом к построению миссий, где каждый манёвр, проверка и обработка ошибки — отдельный узел.

Домашнее задание:

- **Базовое**: Запустите `square_services.py`. Измерьте отклонение от старта.
- **Продвинутое**: Добавьте в программу подсчёт времени, затраченного на каждый манёвр.
- **Исследовательское**: Проведите 10 попыток, посчитайте процент успешных проездов.

Занятие 31: Деревья поведения: движок, Sequence, MoveRelativeAction, MoveTimeAction, StopAction

Содержание занятия: Модульный подход к построению миссий. Первые узлы и дерево из двух действий

Цели занятия

- **Образовательная**: Понять, что такое дерево поведения (Behavior Tree). Освоить узлы `Sequence`, `MoveRelativeAction`, `MoveTimeAction`, `StopAction` из библиотеки `driver_lib`.
- **Развивающая**: Перейти от линейного последовательного кода к модульной событийной архитектуре с неблокирующим `tick`.
- **Воспитательная**: Показать, как правильная архитектура решает проблемы предыдущего занятия: гонку команд, пропуск статуса, громоздкость.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Первое дерево (45 мин)**
    - BTNode и tick: SUCCESS/FAILURE/RUNNING.

- Sequence — последовательное «И».
- MoveRelativeAction, MoveTimeAction, StopAction.
- Сборка: проехать 1 метр → повернуть на месте.

3. **Уровень «Углублённый»: tick-механика и сравнение (25 мин)**
   - Таймер 5 Гц, неблокирующий вызов.
   - Состояния внутри Action-узлов.
   - Сравнение с последовательным кодом.
4. **Уровень «Исследовательский»: Замеры времени (10 мин)**
   - Логирование времени узлов.
   - go_time vs go_relative.
5. **Подведение итогов (5 мин)**

1. Введение: Конструктор вместо лапши

На прошлом занятии мы написали квадрат последовательным кодом. Он работал, но:

- Приходилось вручную ждать `MOVING`, потом `SUCCESS`.
- Легко было допустить гонку команд (отправить новую до завершения старой).
- Код был линейным — добавить ветвление или повтор было сложно.

Деревья поведения (Behavior Trees) решают эти проблемы. Миссия собирается из готовых **узлов**, как из кубиков LEGO. Каждый узел:

- Сам управляет своим состоянием.
- Гарантирует, что не пошлёт новую команду, пока не завершена предыдущая.
- Возвращает один из трёх статусов: `SUCCESS`, `FAILURE`, `RUNNING`.

Библиотека `driver_lib` уже содержит всё необходимое. Мы будем их **использовать**, а не писать с нуля.

2. Уровень «База»: Первое дерево

(Инструмент, навык, интерфейс)

Обязательный уровень. Знакомимся с узлами и собираем простую миссию.

2.1. Как работает дерево поведения

Дерево состоит из узлов. Каждый узел при вызове `tick(driver)` возвращает:

- **`SUCCESS`** — действие выполнено успешно.
- **`FAILURE`** — действие провалилось.
- **`RUNNING`** — действие ещё выполняется (продолжаем тикать).

Узлы, которые управляют другими узлами:

- **`Sequence`** — последовательное «И»: выполняет детей по порядку. Если один вернул `FAILURE` — весь Sequence возвращает `FAILURE`. Если все `SUCCESS` — Sequence `SUCCESS`.
- **`Fallback`** — «ИЛИ»: пробует детей, пока один не вернёт `SUCCESS`.
- **`Parallel`** — тикает всех детей одновременно.

Узлы-действия (листья дерева):

- **`MoveRelativeAction`** — вызывает `go_relative`, ждёт `SUCCESS`.
- **`MoveTimeAction`** — вызывает `go_time`, ждёт `SUCCESS`.
- **`StopAction`** — вызывает `stop`, мгновенный `SUCCESS`.
- **`HeartbeatAction`** — шлёт heartbeat, всегда `RUNNING`.

2.2. Скрипт: первое дерево

Создайте файл:

```
nano ~/ws/rbt/scripts/my_first_bt.py
```

```
#!/usr/bin/env python3
"""Первое дерево поведения: проехать 1 м, повернуть на месте 2 секунды."""
import sys
import rclpy
from driver_lib import RobotDriver, Sequence, MoveRelativeAction, MoveTime
Action, HeartbeatAction, Parallel

class MyFirstBT(RobotDriver):
    def __init__(self):
        super().__init__('my_first_bt')
        self._bt_root = Parallel([
            HeartbeatAction(),
            Sequence([
                MoveRelativeAction(dx=1.0, dy=0.0, dyaw_deg=0.0, name="Впе
рёд 1 м"),
                MoveTimeAction(vx=0.0, vy=0.0, omega=0.5, duration=3.14, n
ame="Поворот ~90°"),
            ])
        ], required_successes=1)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info("Дерево готово")

    def _loop(self):
        if not self._active:
            print("Дерево: проехать 1 м вперёд, повернуть на месте ~90°")
            input("Нажмите Enter ... ")
            self.get_logger().info(f"Запуск дерева ... {self.get_clock().no
w().nanoseconds}")
            self._active = True
            return
```

```
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Дерево выполнено успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Дерево провалено!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())


def main():
    rclpy.init()
    node = MyFirstBT()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/my_first_bt.py
```

2.3. Что происходит внутри

1. Каждые 0.2 секунды (5 Гц) вызывается `tree.tick(driver)`.
2. `Parallel` тикает всех детей:
   - `HeartbeatAction` — шлёт `Empty` в `/autopilot/heartbeat`, возвращает `RUNNING`.
   - `Sequence` — тикает первого ребёнка `MoveRelativeAction`.
3. `MoveRelativeAction` (конечный автомат внутри):
   - Первый `tick`: отправляет `go_relative(dx=1)`, переходит в `WAITING_RESPONSE`, возвращает `RUNNING`.
   - Следующие `tick`: ждёт ответа сервиса. Получив `success=True`, переходит в `MOVING`, возвращает `RUNNING`.
   - Дальше: ждёт статус `SUCCESS`. Получив — возвращает `SUCCESS`.
4. `Sequence` видит `SUCCESS` от первого ребёнка, переходит ко второму (`MoveTimeAction`).
5. Когда оба ребёнка вернули `SUCCESS`, `Sequence` возвращает `SUCCESS`.

6. `Parallel` с `required_successes=1` видит `SUCCESS` от `Sequence` и сам возвращает `SUCCESS`.

Ключевое преимущество: мы не писали ни одного цикла ожидания, ни одной проверки статуса. Всё это внутри готовых узлов.

3. Уровень «Углублённый»: tick-механика и сравнение

(Что под капотом)

Разбираем конечный автомат внутри Action-узлов.

3.1. Конечный автомат внутри MoveRelativeAction

```
START
  │ tick
  ▼
WAITING_RESPONSE ── timeout/error ──► FAILURE
  │ future.done() && response.success
  ▼
MOVING
  │ tick (каждые 0.2 с)
  ├── status == SUCCESS ──► SUCCESS
  ├── status == ABORTED_* ──► FAILURE
  └── timeout ──► FAILURE
```

Каждый `tick` — один шаг автомата. Блокировок нет. Таймер 5 Гц гарантирует, что heartbeat успевает уходить.

3.2. Сравнение с последовательным кодом

Проблема	**Последовательный код (занятие 30)**	**Дерево поведения**
Гонка команд	Надо вручную ждать ответ Future	Узел сам ждёт внутри WAITING_RESPONSE
Пропуск статуса	Надо двухфазное ожидание (MOVING → SUCCESS)	Встроено в конечный автомат
Heartbeat	Надо вызывать в каждом цикле	HeartbeatAction в Parallel
Добавление манёвра	Добавить строку в список	Добавить узел в Sequence
Обработка ошибок	if-else после каждого шага	Fallback, RetryAction (следующее занятие)

4. Уровень «Исследовательский»: Замеры времени

(Прорыв за границы)

Эксперименты с производительностью.

4.1. Логирование времени выполнения узлов

Добавьте `import time` и замеряйте время между первым `tick` узла и его `SUCCESS`. Узлы `MoveRelativeAction` и `MoveTimeAction` можно обернуть в свой класс, но проще — смотреть на логи `[BT] ...: Успешно` и считать время между ними.

Задание: Запишите, сколько времени занял `MoveRelativeAction(dx=1)` и `MoveTimeAction(omega=0.5, duration=3.14)`.

4.2. Мини-исследование: go_time vs go_relative

Вопрос: Что выполняется быстрее для поворота на 90° — `MoveTimeAction` с `go_time(omega=0.5, duration=3.14)` или `MoveRelativeAction` с `go_relative(dyaw_deg=90)`?

Проверка: Соберите два дерева, запустите по очереди, сравните время.

Ожидание: `go_relative` может быть быстрее или медленнее в зависимости от реальной угловой скорости. `go_time` длится ровно заданное время.

5. Выводы по занятию

1. **Дерево поведения — модульный конструктор миссий.** Узлы собираются в Sequence, Parallel, Fallback.
2. **Action-узлы уже готовы.** `MoveRelativeAction`, `MoveTimeAction`, `StopAction`, `HeartbeatAction` — в библиотеке `driver_lib`.
3. **Конечный автомат внутри узла решает проблемы гонки команд и пропуска статуса.** START → WAITING_RESPONSE → MOVING → SUCCESS.
4. **HeartbeatAction в Parallel** гарантирует, что пульс идёт всегда, независимо от основной миссии.
5. **Код стал проще и надёжнее.** Мы не пишем циклы ожидания — их выполняют готовые узлы.

Связь со следующим занятием: Мы собрали линейную цепочку действий. Но что если манёвр провалился? Нужно **повторить** или **выбрать альтернативу**. Для этого есть `Condition`, `Fallback` и `RetryAction` — тема следующего занятия.

Домашнее задание:

- **Базовое**: Запустите `my_first_bt.py`. Убедитесь, что робот проехал 1 м и повернул.
- **Продвинутое**: Замените `MoveTimeAction` на `MoveRelativeAction(dyaw_deg=90)`. Сравните точность поворота.
- **Исследовательское**: Замерьте время выполнения `MoveRelativeAction(dx=1)` и `MoveTimeAction(vx=0.2, duration=5)`.

Занятие 32: Condition, Fallback, RetryAction: ветвление и обработка ошибок в дереве

Содержание занятия: Принятие решений и восстановление после сбоев в дереве поведения

Цели занятия

- **Образовательная**: Освоить узлы ветвления: `Condition` (проверка условия), `Fallback` (логическое «ИЛИ»), `RetryAction` (повтор при FAILURE).
- **Развивающая**: Научиться строить деревья, которые не просто выполняют цепочку действий, а **реагируют** на нештатные ситуации.
- **Воспитательная**: Показать, что надёжная система — это не та, которая не ломается, а та, которая умеет восстанавливаться.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Три новых узла (45 мин)**
   - `Condition` — мгновенная проверка.
   - `Fallback` — перебор альтернатив.
   - `RetryAction` — повтор при FAILURE.
   - Пример: квадрат с RetryAction на прямых.
3. **Уровень «Углублённый»: Композиция и reset (25 мин)**
   - Fallback над Sequence.
   - Почему нужен `reset()`.
   - Цепочка обработки ошибки.
4. **Уровень «Исследовательский»: Оптимальное число повторов (10 мин)**
   - RetryAction с задержкой.
   - Эксперимент: 1, 3 или 5 retry.
5. **Подведение итогов (5 мин)**

1. Введение: Когда всё идёт не по плану

На прошлом занятии мы собрали линейное дерево: проехать метр, повернуть. Это работает, пока всё идёт гладко. Но что если робот застревает?

Сегодня мы добавим в арсенал три новых узла:

- **`Condition`** — «посмотреть по сторонам»: мгновенная проверка (например, «батарея > 20%?»).
- **`Fallback`** — «план Б»: если первый способ не сработал, пробуем второй.
- **`RetryAction`** — «попробовать ещё раз»: если действие провалилось, повторяем его N раз.

2. Уровень «База»: Три новых узла

(Инструмент, навык, интерфейс)

Обязательный уровень. Осваиваем `Condition`, `Fallback`, `RetryAction`.

2.1. Condition — мгновенная проверка

`Condition` принимает **функцию**, которая получает `driver` и возвращает `True` или `False`. На каждом `tick` функция вызывается, и узел мгновенно возвращает `SUCCESS` или `FAILURE`.

```
def battery_ok(driver):
    return driver.get_status() ≠ "ABORTED_TIMEOUT"

Condition(battery_ok)  # SUCCESS если статус не ABORTED_TIMEOUT
```

Важно: `Condition` не ждёт и не блокирует. Он просто проверяет и сразу отвечает. Это аналог `if` в обычном коде.

2.2. Fallback — перебор альтернатив (логическое «ИЛИ»)

`Fallback` содержит список детей. На каждом `tick` он пробует детей **по порядку**, пока один не вернёт `SUCCESS`. Если все вернули `FAILURE` — `Fallback` возвращает `FAILURE`.

```
pythonFallback([
    MoveRelativeAction(dx=1.0, ...),  # попробовать проехать прямо
    MoveRelativeAction(dx=-0.5, ...), # не вышло — отъехать назад
])
```

Аналогия: Это `try` без `except`. Пробуем первое, если не вышло — второе.

2.3. RetryAction — повтор при FAILURE

`RetryAction` оборачивает один дочерний узел. Если тот возвращает `FAILURE`, `RetryAction` **сбрасывает** его состояние (`reset()`) и пробует снова. До `max_attempts` раз.

```
RetryAction(
    MoveRelativeAction(dx=1.0, name="Вперёд"),
    max_attempts=3
)
```

Если робот застрял на первом метре (`ABORTED_STUCK` → `FAILURE`), `RetryAction` сбрасывает узел и пробует ещё раз. Со второй попытки может проехать.

2.4. Пример: прямая с повтором при ошибке

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_retry.py
```

```
#!/usr/bin/env python3
"""Дерево: проехать 1 м с 3 попытками, затем повернуть на 90°."""
```

```
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Parallel,
                        MoveRelativeAction, HeartbeatAction, RetryAction)

class BtRetry(RobotDriver):
    def __init__(self):
        super().__init__('bt_retry')
        self._bt_root = Parallel([
            HeartbeatAction(),
            Sequence([
                RetryAction(
                    MoveRelativeAction(dx=1.0, dy=0.0, dyaw_deg=0.0,
name="Вперёд 1 м"),
                    max_attempts=3
                ),
                MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="По
ворот 90°"),
            ])
        ], required_successes=1)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info("Дерево с Retry готово")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print("\nРобот готов к миссии 'Прямая с повтором'.")
            input("Нажмите Enter для старта ... ")
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Миссия выполнена успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Миссия провалена!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    rclpy.init()
    node = BtRetry()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/bt_retry.py
```

3. Уровень «Углублённый»: Композиция и reset

(Что под капотом)

Как узлы работают вместе и почему нужен сброс состояния.

3.1. Fallback над Sequence — «если... то... иначе...»

```
pythonFallback([
    Sequence([                                  # Попытка А:
        Condition(lambda d: d.get_status() == "IDLE"),  # всё спокойно?
        MoveRelativeAction(dx=1.0, ...),        # едем прямо
    ]),
    Sequence([                                  # Попытка Б (если А не вышла):
        LogAction("Объезжаем!"),
        MoveRelativeAction(dx=0.0, dy=0.5, ...),  # крабом вбок
    ]),
])
```

Если первая `Sequence` падает (например, `Condition` вернул `FAILURE` или `MoveRelativeAction` застрял) — `Fallback` пробует вторую.

3.2. Почему нужен reset()

У `MoveRelativeAction` есть внутреннее состояние: `START → WAITING_RESPONSE → MOVING → SUCCESS`. Если он вернул `FAILURE`, его состояние осталось в `WAITING_RESPONSE` или `MOVING`.

При повторном `tick` без сброса он попытается **продолжить** с того же места, а не начать заново. `RetryAction` вызывает `child.reset()`, который возвращает состояние в `START`.

Цепочка обработки:

```
MoveRelativeAction.tick() → FAILURE
RetryAction видит FAILURE
RetryAction вызывает child.reset()  ← состояние сброшено в START
RetryAction на следующем tick снова вызывает child.tick() → START → ...
```

4. Уровень «Исследовательский»: Оптимальное число повторов

(Прорыв за границы)

Эксперименты с параметрами восстановления.

4.1. RetryAction с задержкой между попытками

Стандартный `RetryAction` пробует немедленно. Но иногда полезно **подождать** перед повтором (например, чтобы робот успокоился после удара).

Идея для модификации (самостоятельно): добавить `TimerAction` внутрь `RetryAction`, чтобы между попытками была пауза 1 секунда.

4.2. Мини-исследование: 1, 3 или 5 retry?

Вопрос: Сколько повторов оптимально для `ABORTED_STUCK`?

Эксперимент:

1. Поставьте препятствие перед роботом.
2. Запустите `bt_retry.py` с `max_attempts=1`, `3`, `5`.
3. Замерьте: сколько раз робот реально пытается проехать? Через сколько секунд сдаётся?

Ожидание: `max_attempts=1` — сразу сдаётся. `3` — разумный компромисс. `5` — долго буксует, износ колёс.

Инженерный вывод: Количество повторов зависит от цены ошибки. Для учебного полигона 3 — нормально. Для склада с хрупким грузом — лучше 1 и вызов оператора.

5. Выводы по занятию

1. **`Condition` — мгновенная проверка.** `if` в мире деревьев. Не ждёт, не блокирует.
2. **`Fallback` — перебор альтернатив.** Логическое «ИЛИ»: пробуем первый способ, потом второй.
3. **`RetryAction` — повтор при FAILURE.** Сбрасывает состояние дочернего узла и пробует снова до N раз.
4. **`reset()` обязателен для повтора.** Без него узел продолжит с середины, а не начнёт заново.
5. **Деревья умеют восстанавливаться.** Это главное преимущество перед линейным кодом.

Домашнее задание:

- **Продвинутое**: Добавьте `Fallback` в дерево: если `MoveRelativeAction` провалился — выполнить `MoveTimeAction(vx=-0.1, duration=1.0)` для отъезда назад.
- **Исследовательское**: Проведите эксперимент с `max_attempts=1,3,5`. Запишите время до сдачи.

Занятие 33: Parallel: одновременные действия

Содержание занятия: Параллельное выполнение узлов в дереве поведения

Цели занятия

- **Образовательная**: Освоить узел `Parallel` для одновременного выполнения нескольких действий. Понять параметр `required_successes`.
- **Развивающая**: Научиться отделять «фоновые» задачи (heartbeat, мониторинг) от основной миссии.
- **Воспитательная**: Показать, что параллелизм в дереве — это не многопоточность, а кооперативная многозадачность на одном тике.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Parallel для фоновых задач (40 мин)**
   - Как работает `Parallel`.
   - `required_successes=1` — «хотя бы один».
   - `required_successes=len` — «все».
   - Пример: движение + heartbeat.
3. **Уровень «Углублённый»: Стратегии и FAILURE (25 мин)**
   - Когда FAILURE одного ребёнка должен прервать весь Parallel.
   - `required_successes` и поведение при ошибке.
4. **Уровень «Исследовательский»: Параллельное движение (15 мин)**
   - Два `MoveRelativeAction` одновременно.
   - Поведение меканум-базы при конфликтующих командах.
5. **Подведение итогов (5 мин)**

1. Введение: Одна голова хорошо, а две — одновременно

До сих пор мы использовали `Parallel` только для связки «heartbeat + миссия». Но его возможности шире.

Параллельный узел на каждом `tick` тикает **всех** своих детей. Они выполняются не в отдельных потоках, а **по очереди в рамках одного тика**, но с точки зрения дерева они работают «одновременно».

Это позволяет:

- Слать heartbeat во время движения.
- Двигаться И отслеживать условие остановки.

2. Уровень «База»: Parallel для фоновых задач

(Инструмент, навык, интерфейс)

Обязательный уровень. Разбираем параметр `required_successes`.

2.1. Как работает Parallel

```
Parallel([A, B, C], required_successes=2)
```

На каждом `tick`:

1. Тикаем `A`. Если вернул `SUCCESS` — запоминаем.
2. Тикаем `B`. Если вернул `SUCCESS` — запоминаем.
3. Тикаем `C`.
4. Если количество `SUCCESS` >= `required_successes` — `Parallel` возвращает `SUCCESS`.
5. Если любой ребёнок вернул `FAILURE` — `Parallel` возвращает `FAILURE`.
6. Иначе — `RUNNING`.

2.2. Две стратегии required_successes

Значение	Логика	Пример
`required_successes=1`	«Хотя бы один»	Heartbeat (всегда RUNNING) + миссия. Миссия SUCCESS → Parallel SUCCESS
`required_successes=len(children)`	«Все»	Два MoveRelativeAction. Оба должны завершиться → Parallel SUCCESS

Случай с HeartbeatAction:

```
Parallel([
    HeartbeatAction(),           # всегда RUNNING
    MoveRelativeAction(dx=1.0),  # RUNNING → SUCCESS
], required_successes=1)
```

`HeartbeatAction` никогда не возвращает `SUCCESS`, только `RUNNING`. `MoveRelativeAction` возвращает `SUCCESS`. `required_successes=1` достигнут — `Parallel` возвращает `SUCCESS`.

Если бы `required_successes=2`, `Parallel` **никогда бы не завершился**, потому что `HeartbeatAction` всегда `RUNNING`.

2.3. Пример: движение с контролем статуса

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_parallel_demo.py
```

```
#!/usr/bin/env python3
"""Дерево: ехать вперёд И одновременно проверять статус."""
```

```
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Parallel,
                        MoveRelativeAction, HeartbeatAction, Condition)

def status_ok(driver):
    """Условие: статус не аварийный."""
    s = driver.get_status()
    return s not in ("ABORTED_STUCK", "ABORTED_TIMEOUT")

class BtParallelDemo(RobotDriver):
    def __init__(self):
        super().__init__('bt_parallel_demo')
        self._bt_root = Parallel([
            HeartbeatAction(),
            Condition(status_ok),
            Sequence([
                MoveRelativeAction(dx=0.5, dy=0.0, dyaw_deg=0.0, name="Пол
метра"),
                MoveRelativeAction(dx=0.0, dy=0.5, dyaw_deg=0.0, name="Кра
бом вбок"),
                MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="По
ворот"),
            ])
        ], required_successes=2)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info("Дерево с параллельным контролем готово")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print("\nРобот готов: полметра вперёд, крабом вбок, поворот.")
            print("Если статус станет ABORTED — миссия прервётся.")
            input("Нажмите Enter для старта ... ")
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Миссия выполнена успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Миссия провалена (статус ABORTED)!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    rclpy.init()
    node = BtParallelDemo()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
```

```
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/bt_parallel_demo.py
```

Что делают три ребёнка Parallel'а:

1. `HeartbeatAction` — всегда `RUNNING` (фон).
2. `Condition(status_ok)` — `SUCCESS` пока статус не ABORTED. Если статус стал ABORTED — `FAILURE` → весь Parallel падает.
3. `Sequence` из трёх движений — `SUCCESS` когда все три выполнены.

`required_successes=2`: ждём `SUCCESS` от `Condition` И от `Sequence`. `HeartbeatAction` не учитывается (он `RUNNING`).

3. Уровень «Углублённый»: FAILURE в Parallel

(Что под капотом)

Когда ошибка одного ребёнка должна прервать всех.

3.1. Текущее поведение: один FAILURE — все FAILURE

В текущей реализации `Parallel` в `driver_lib`, если **любой** ребёнок возвращает `FAILURE`, весь `Parallel` немедленно возвращает `FAILURE`.

```
if failures > 0:
    return "FAILURE"
```

Это правильно для `Condition(status_ok)`: если статус стал `ABORTED`, мы **хотим** прервать миссию.

Но для `HeartbeatAction` это не проблема — он никогда не возвращает `FAILURE`.

3.2. Когда FAILURE должен прерывать весь Parallel, а когда нет

Всегда прерывать:

- `Condition` слежения за безопасностью (статус ABORTED).
- `MoveRelativeAction` в `required_successes=len` (один из манёвров провалился).

Не должен прерывать (но текущая реализация прерывает):

- Один из нескольких альтернативных манёвров в `required_successes=1`. Но для этого лучше использовать `Fallback`.

Вывод: Текущее поведение («один FAILURE — все FAILURE») достаточно для учебных задач. Для более сложных сценариев используйте `Fallback` вместо `Parallel` с `required_successes=1`.

3.3. StreamAction — непрерывное управление

Что под капотом: `Parallel` тикает всех детей одновременно. Стратегия `required_successes` — сколько детей должны вернуть `SUCCESS` для завершения. Если один ребёнок вернул `FAILURE`, весь `Parallel` падает. `StreamAction` в отличие от `MoveRelativeAction` не ждёт `SUCCESS` от контроллера — он вызывает `move_time` заново на каждом тике с обновлёнными параметрами. Контроллер перезаписывает цель, движение становится непрерывным и плавным.

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_stream_parallel.py
```

```
#!/usr/bin/env python3
"""Дерево: плавный разгон через StreamAction + контроль статуса в
Parallel."""
import sys
import rclpy
from driver_lib import (RobotDriver, Parallel,
                        StreamAction, HeartbeatAction, Condition)

def status_ok(driver):
    """Условие: статус не аварийный."""
    s = driver.get_status()
    return s not in ("ABORTED_STUCK", "ABORTED_TIMEOUT")

def ramped_speed(driver):
    """Плавно наращиваем скорость: от 0 до 0.3 м/с за 3 секунды."""
    elapsed = driver.get_clock().now().nanoseconds * 1e-9
    if not hasattr(ramped_speed, 'start'):
        ramped_speed.start = elapsed
    dt = elapsed - ramped_speed.start
    vx = min(0.3, dt * 0.1)   # 0 → 0.3 м/с за 3 сек
    return (vx, 0.0, 0.0, 0.1)

def time_to_stop(driver):
    """Остановка через 5 секунд."""
    elapsed = driver.get_clock().now().nanoseconds * 1e-9
    if not hasattr(ramped_speed, 'start'):
        return False
    return elapsed - ramped_speed.start > 5.0

class BtStreamParallel(RobotDriver):
    """Миссия: плавный разгон с контролем статуса."""
    def __init__(self):
        super().__init__('bt_stream_parallel')
        self._bt_root = Parallel([
```

```
            HeartbeatAction(),
            Condition(status_ok),
            StreamAction(command_func=ramped_speed,
                         stop_condition=time_to_stop,
                         name="Плавный разгон 5 сек"),
        ], required_successes=2)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info("Дерево с плавным разгоном готово")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print("\nРобот будет разгоняться плавно с 0 до 0.3 м/с.")
            print("Остановка через 5 секунд.")
            print("Если статус станет ABORTED — миссия прервётся.")
            input("Нажмите Enter для старта ... ")
            ramped_speed.start = self.get_clock().now().nanoseconds * 1e-9
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Разгон завершён!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Миссия прервана (статус ABORTED)!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    rclpy.init()
    node = BtStreamParallel()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/bt_stream_parallel.py
```

Что делают три ребёнка Parallel'а:

Узел	Поведение	Возвращает
`HeartbeatAction`	Шлёт пульс автопилоту каждый тик	Всегда `RUNNING`
`Condition(status_ok)`	Проверяет что статус не ABORTED	`SUCCESS` пока всё в порядке
`StreamAction`	Каждые 0.2 сек вызывает `move_time` с обновлённой скоростью	`RUNNING` всегда (пока не `stop_condition`)

`required_successes=2`: ждём `SUCCESS` от `Condition` И от `StreamAction`. `HeartbeatAction` не учитывается — он всегда `RUNNING`.

Ключевое отличие `StreamAction` от `MoveRelativeAction`:

	`MoveRelativeAction`	`StreamAction`
Как работает	Отправил `go_relative` → ждёт `SUCCESS`	Каждый тик шлёт `move_time` с новыми параметрами
Когда завершается	Когда контроллер достиг цели	Никогда (или по `stop_condition`)
Для чего	Точное перемещение на метр, поворот на 90°	Плавный разгон, движение по линии, объезд препятствия

`StreamAction` — это основа для работы с сенсорами в реальном времени. В разделе «Сенсоры» именно он будет читать данные лидара и камеры и корректировать движение на лету.

4. Уровень «Исследовательский»: Параллельное движение

(Прорыв за границы)

Эксперимент с двумя командами одновременно.

4.1. Два MoveRelativeAction в Parallel

Идея: Запустить `MoveRelativeAction(dx=1)` и `MoveRelativeAction(dyaw_deg=90)` одновременно. Меканум-база позволяет ехать вперёд и поворачиваться одновременно.

Но! `MoveRelativeAction` вызывает `go_relative`, который блокирует автопилот до завершения. Два одновременных вызова приведут к тому, что второй прервёт первый.

Правильный способ: Использовать `MoveTimeAction` с ненулевыми `vx` и `omega` одновременно:

```
pythonParallel([
    HeartbeatAction(),
    MoveTimeAction(vx=0.2, vy=0.0, omega=0.3, duration=5.0, name="Дуга"),
], required_successes=1)
```

Это заставит робота ехать по дуге.

4.2. Мини-исследование: конфликтующие команды

Эксперимент: Запустите два `MoveTimeAction` в `Parallel` с `required_successes=2`:

```
Parallel([
    MoveTimeAction(vx=0.2, vy=0.0, omega=0.0, duration=3.0, name="Вперёд")
,
    MoveTimeAction(vx=0.0, vy=0.0, omega=0.5, duration=3.0,
name="Поворот"),
], required_successes=2)
```

Вопрос: Как поведёт себя робот? Какая команда победит?

Ожидание: Оба узла на каждом тике вызывают `driver.move_time( ... )`. Автопилот получит два вызова почти одновременно. Победит последний (второй `tick` перезапишет цель). Робот будет только поворачиваться.

Вывод: `Parallel` не предназначен для отправки конфликтующих команд. Для дуг используйте **один** `MoveTimeAction` с обоими компонентами скорости.

5. Выводы по занятию

1. **`Parallel` тикает всех детей на каждом `tick`.** Это кооперативная многозадачность, не потоки.
2. **`required_successes` управляет завершением.** `1` — «хотя бы один» (для heartbeat). `len` — «все» (для группы манёвров).
3. **Один FAILURE прерывает весь Parallel.** Это правильно для безопасности, но для альтернатив лучше `Fallback`.
4. **Конфликтующие команды в Parallel не работают.** Для дуг используйте один `MoveTimeAction` с обоими компонентами.

Связь со следующим занятием: Мы освоили все базовые узлы. На следующем занятии — **полный квадрат на дереве поведения** с `RetryAction`, `Parallel`, `Sequence` и обработкой ошибок.

Домашнее задание:

- **Базовое**: Запустите `bt_parallel_demo.py`. Убедитесь, что `Condition` прерывает миссию при ABORTED.
- **Продвинутое**: Добавьте в дерево `RetryAction` для прямолинейного движения.
- **Исследовательское**: Проведите эксперимент с двумя `MoveTimeAction` в `Parallel`. Опишите поведение робота.

Занятие 34: Квадрат на дереве поведения: полный сценарий с обработкой ошибок

Содержание занятия: Сборка завершённой миссии из готовых узлов. Сравнение с последовательным кодом

Цели занятия

- **Образовательная**: Собрать полное дерево поведения для проезда квадрата с обработкой ошибок, используя все изученные узлы.
- **Развивающая**: Сравнить два подхода к одной задаче: последовательный код (занятие 30) и дерево поведения. Оценить, что легче отлаживать и расширять.
- **Воспитательная**: Показать, что выбор архитектуры влияет на сопровождаемость кода сильнее, чем выбор языка или библиотеки.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Дерево для квадрата (45 мин)**
   - Структура: Parallel (Heartbeat + Sequence из 8 действий).
   - RetryAction на прямых, простые повороты.
   - Запуск и демонстрация на полигоне.
3. **Уровень «Углублённый»: Сравнение с последовательным кодом (25 мин)**
   - Что легче отлаживать.
   - Что легче расширять.
   - Статус-машина vs ручная проверка.
4. **Уровень «Исследовательский»: Параметризация дерева (10 мин)**
   - Функция `build_polygon(sides, length)`.
   - Какой многоугольник точнее.
5. **Подведение итогов (5 мин)**

1. Введение: Собираем пазл

Мы изучили все базовые узлы дерева поведения:

- `Sequence` — последовательное «И».
- `Parallel` — одновременное выполнение.
- `Fallback` — перебор альтернатив.
- `RetryAction` — повтор при ошибке.
- `Condition` — мгновенная проверка.
- `MoveRelativeAction`, `MoveTimeAction`, `StopAction` — действия.

- `HeartbeatAction` — фоновый пульс.

Сегодня мы соберём из них **полноценную миссию** — квадрат с обработкой ошибок. И сравним с последовательным кодом из занятия 30.

2. Уровень «База»: Дерево для квадрата

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем дерево и запускаем на полигоне.

2.1. Архитектура дерева

```
Parallel (required_successes=1)
├── HeartbeatAction          ← всегда RUNNING, шлёт пульс
└── Sequence                 ← основная миссия
    ├── LogAction("Старт")
    ├── RetryAction(MoveRelativeAction(1,0,0))   ← сторона 1
    ├── MoveRelativeAction(0,0,90)                ← поворот 1
    ├── RetryAction(MoveRelativeAction(1,0,0))   ← сторона 2
    ├── MoveRelativeAction(0,0,90)                ← поворот 2
    ├── RetryAction(MoveRelativeAction(1,0,0))   ← сторона 3
    ├── MoveRelativeAction(0,0,90)                ← поворот 3
    ├── RetryAction(MoveRelativeAction(1,0,0))   ← сторона 4
    ├── MoveRelativeAction(0,0,90)                ← поворот 4
    └── LogAction("Финиш")
```

Почему RetryAction только на прямых? Поворот на месте реже застревает. Если нужно — можно обернуть и повороты.

2.2. Скрипт bt_square.py

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_square.py
```

```
#!/usr/bin/env python3
"""Квадрат на дереве поведения с обработкой ошибок."""
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Parallel,
                        MoveRelativeAction, HeartbeatAction,
                        RetryAction, LogAction)

def build_square_tree(side):
    """Создаёт дерево для квадрата со стороной side метров."""
    return Parallel([
        HeartbeatAction(),
        Sequence([
            LogAction("Старт миссии: Квадрат"),
            RetryAction(MoveRelativeAction(dx=side, dy=0.0, dyaw_deg=0.0,
name="Вперёд 1")),
            MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="Поворо
```

```
т 1"),
            RetryAction(MoveRelativeAction(dx=side, dy=0.0, dyaw_deg=0.0,
name="Вперёд 2")),
            MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="Поворо
т 2"),
            RetryAction(MoveRelativeAction(dx=side, dy=0.0, dyaw_deg=0.0,
name="Вперёд 3")),
            MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="Поворо
т 3"),
            RetryAction(MoveRelativeAction(dx=side, dy=0.0, dyaw_deg=0.0,
name="Вперёд 4")),
            MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0, name="Поворо
т 4"),
            LogAction("Миссия завершена успешно!"),
        ])
    ], required_successes=1)

class BtSquare(RobotDriver):
    def __init__(self, side=1.0):
        super().__init__('bt_square')
        self._bt_root = build_square_tree(side)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info(f"Дерево квадрата {side}x{side} м готово")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print(f"\nРобот готов к миссии 'Квадрат'.")
            input("Нажмите Enter для старта ... ")
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Дерево выполнено успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Дерево вернуло FAILURE. Аварийная
остановка.")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    import sys as _sys
    rclpy.init()
    # УМНЫЙ ОДНОСТРОЧНЫЙ ПАРСЕР:
    # Ищем первый аргумент, который НЕ начинается с дефиса и НЕ содержит
знака равенства (это и будет наш размер стороны)
    side_arg = [arg for arg in sys.argv[1:] if not arg.startswith('-')
and ':=' not in arg]


# Если такое число нашли — конвертируем его, иначе берем дефолтный 1.0
метр
```

```
    side = float(side_arg[0]) if side_arg else 1.0
    node = BtSquare(side)
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

Запуск:

```
python3 ~/ws/rbt/scripts/bt_square.py
# Или с другим размером:
python3 ~/ws/rbt/scripts/bt_square.py 0.5
```

2.3. Тестирование на полигоне

1. **Нормальный прогон:** робот проезжает квадрат и возвращается в начало.
2. **С препятствием:** поставьте коробку на пути второй стороны. Робот застревает, `RetryAction` пробует ещё 2 раза, затем дерево возвращает `FAILURE`.

Наблюдайте за логами: каждый узел пишет `[BT] имя_узла: ...`. Видно, какой узел выполняется, какой провалился.

3. Уровень «Углублённый»: Сравнение с последовательным кодом

(Что под капотом)

Два подхода — одна задача.

3.1. Сравнительная таблица

Критерий	Последовательный код (зан. 30)	Дерево поведения (зан. 34)
Строк кода	~80 (с двухфазным ожиданием)	~50 (сборка из готовых узлов)
Обработка ошибок	Ручная: `if status == ABORTED: stop()`	Встроена: `RetryAction`, `FAILURE` → `stop()`
Добавление манёвра	Добавить строку в список + вызов `wait_for_result`	Добавить узел в `Sequence`
Heartbeat	Ручной вызов в каждом цикле	`HeartbeatAction` в `Parallel`
Отладка	Читать логи, искать номер манёвра	Логи подписаны именем узла (`[BT] Вперёд 2: ...`)
Гонка команд	Возможна, нужна осторожность	Исключена конечным автоматом узла
Расширение (объезд)	Переписывать всю логику	Добавить `Fallback` с альтернативой

3.2. Что легче: отлаживать и расширять

Отладка: Дерево легче. Каждый узел пишет своё имя в лог. Вы сразу видите: `[BT] Вперёд 2: Ошибка: ABORTED_STUCK`. В последовательном коде нужно по номерам строк вычислять, где произошла ошибка.

Расширение: Дерево легче. Чтобы добавить объезд препятствия, достаточно заменить `MoveRelativeAction` на `Fallback([MoveRelativeAction(...), MoveRelativeAction(dy=0.5, ...)])`. В последовательном коде пришлось бы добавлять `if-else` и менять структуру циклов.

Вывод: Дерево поведения масштабируется лучше. Последовательный код проще для совсем маленьких задач (2–3 действия).

4. Уровень «Исследовательский»: Параметризация и точность

(Прорыв за границы)

Создаём универсальный построитель многоугольников.

4.1. Функция build_polygon(sides, length)

Модифицируйте `build_square_tree`, чтобы она принимала число сторон и угол:

```
def build_polygon(sides, length):
    """Создаёт дерево для правильного многоугольника."""
    angle = 180.0 * (sides - 2) / sides  # внутренний угол
    turn = 180.0 - angle                  # внешний угол (поворот)

    children = [LogAction(f"Старт: {sides}-угольник со стороной {length}
м")]
    for i in range(sides):
        children.append(
```

```
            RetryAction(MoveRelativeAction(dx=length, dy=0.0,
dyaw_deg=0.0, name=f"Сторона {i+1}"))
        )
        children.append(
            MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=turn, name=f"Повор
от {i+1}")
        )
    children.append(LogAction(f"{sides}-угольник завершён!"))

    return Parallel([
        HeartbeatAction(),
        Sequence(children)
    ], required_successes=1)
```

4.2. Мини-исследование: какой многоугольник точнее

Вопрос: Робот проезжает треугольник, квадрат и пятиугольник. Какой фигуры финиш ближе к старту?

Гипотеза: Квадрат точнее треугольника (меньше накопление угловой ошибки). Пятиугольник может быть ещё точнее (меньше угол поворота → меньше проскальзывание).

Проверьте: запустите `build_polygon(3, 0.5)`, `build_polygon(4, 0.5)`, `build_polygon(5, 0.5)`. Измерьте отклонение.

5. Выводы по занятию

1. **Дерево поведения — конструктор миссий.** Квадрат собирается из готовых узлов за 5 минут.
2. **Код стал короче и надёжнее.** Конечные автоматы внутри узлов исключают гонку команд и пропуск статуса.
3. **Отладка прозрачнее.** Каждый узел подписан, логи читаются как «история миссии».
4. **Расширение — добавление узлов.** Не нужно переписывать логику.
5. **Параметризация делает дерево универсальным.** `build_polygon(sides, length)` — любые правильные многоугольники.

Связь со следующим занятием: Мы освоили деревья. На следующем занятии — **итоговое испытание**: комплексные траектории (змейка, восьмёрка), нештатные ситуации и зачёт.

Домашнее задание:

- **Базовое**: Запустите `bt_square.py`. Измерьте отклонение от старта.
- **Продвинутое**: Добавьте `RetryAction` на повороты. Улучшилась ли точность?
- **Исследовательское**: Реализуйте `build_polygon` и сравните точность треугольника, квадрата и пятиугольника.

Занятие 35: Итоговое испытание: комплексная траектория на деревьях

Содержание занятия: Самостоятельная сборка миссии из готовых узлов. Зачёт

Цели занятия

- **Образовательная**: Закрепить навыки построения деревьев поведения на примере сложных траекторий (змейка, восьмёрка).
- **Развивающая**: Научиться выбирать подходящий тип действия (`MoveRelativeAction` vs `MoveTimeAction`) для разных участков траектории.
- **Воспитательная**: Провести зачёт в формате «реального испытания» с нештатными ситуациями.

План занятия (90 минут)

1. **Организационный момент и регламент зачёта (5 мин)**
2. **Уровень «База»: Сборка и испытание траекторий (50 мин)**
   - Змейка: `MoveRelativeAction` с чередованием `dx` и `dy` .
   - Восьмёрка: `MoveTimeAction` с дугами.
   - Испытание на полигоне с препятствием.
3. **Уровень «Углублённый»: Анализ ошибок (20 мин)**
   - Какое дерево упало, какое выполнилось.
   - Где уместнее `MoveRelativeAction` , а где `MoveTimeAction` .
4. **Уровень «Исследовательский»: Универсальный лаунчер (10 мин)**
   - Класс, принимающий готовое дерево и запускающий его.
5. **Подведение итогов блока (5 мин)**

1. Введение: Выпускной экзамен деревьев

Мы прошли путь от ручного `Twist` до сервисов автопилота и деревьев поведения. Сегодня — **итоговое испытание**.

Вы получите задание: собрать дерево для сложной траектории, запустить на полигоне и **обработать нештатную ситуацию** (преподаватель создаст препятствие).

Это не просто «проехать по программе». Это проверка того, что ваше дерево **умеет восстанавливаться после ошибок**.

2. Уровень «База»: Сборка и испытание траекторий

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем две сложные траектории и испытываем их.

2.1. Траектория «Змейка»

Змейка — это последовательность прямолинейных участков с чередованием направления:

```
Вперёд 0.5 м → Влево 0.5 м → Вперёд 0.5 м → Вправо 0.5 м → ...
```

Для змейки идеально подходит `MoveRelativeAction` с `dy` (крабовое движение).

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_snake.py
```

```
#!/usr/bin/env python3
"""Змейка на дереве поведения."""
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Parallel,
                        MoveRelativeAction, HeartbeatAction,
                        RetryAction, LogAction)

def build_snake_tree(segments, step):
    """Создаёт дерево для змейки.
    segments — количество зигзагов.
    step — длина одного сегмента, м.
    """
    children = [LogAction(f"Старт: Змейка из {segments} сегментов")]
    for i in range(segments):
        # Прямой участок (вперёд)
        children.append(
            RetryAction(MoveRelativeAction(dx=step, dy=0.0, dyaw_deg=0.0,
name=f"Вперёд {i+1}"))
        )
        # Боковой участок (влево/вправо по очереди)
        dy = step if i % 2 == 0 else -step
        children.append(
            MoveRelativeAction(dx=0.0, dy=dy, dyaw_deg=0.0, name=f"Бок
{i+1}")
        )
    children.append(LogAction("Змейка завершена!"))

    return Parallel([
        HeartbeatAction(),
        Sequence(children)
    ], required_successes=1)

class BtSnake(RobotDriver):
    def __init__(self, segments=4, step=0.5):
        super().__init__('bt_snake')
        self._bt_root = build_snake_tree(segments, step)
```

```
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info(f"Змейка готова: {segments} сегментов по {s
tep} м")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print(f"\nРобот готов к миссии 'Змейка'.")
            input("Нажмите Enter для старта ... ")
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Змейка выполнена успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Змейка провалена!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    rclpy.init()
    node = BtSnake()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

2.2. Траектория «Восьмёрка»

Восьмёрка — это две дуги. Для дуг нужен `MoveTimeAction` с одновременными `vx` и `omega`:

Создайте файл:

```
nano ~/ws/rbt/scripts/bt_eight.py
```

```
#!/usr/bin/env python3
"""Восьмёрка на дереве поведения."""
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Parallel,
                        MoveTimeAction, HeartbeatAction, LogAction)
```

```
class BtEight(RobotDriver):
    def __init__(self):
        super().__init__('bt_eight')
        self._bt_root = Parallel([
            HeartbeatAction(),
            Sequence([
                LogAction("Старт: Восьмёрка"),
                MoveTimeAction(vx=0.2, vy=0.0, omega=0.4, duration=8.0, na
me="Дуга влево"),
                MoveTimeAction(vx=0.2, vy=0.0, omega=-0.4, duration=8.0, n
ame="Дуга вправо"),
                LogAction("Восьмёрка завершена!"),
            ])
        ], required_successes=1)
        self._active = False
        self._ticker = self.create_timer(0.2, self._loop)
        self.get_logger().info("Восьмёрка готова")

    def _loop(self):
        if not self.is_connected():
            return
        if not self._active:
            print("\nРобот готов к миссии 'Восьмёрка'.")
            input("Нажмите Enter для старта ... ")
            self._active = True
            return
        status = self._bt_root.tick(self)
        if status == "SUCCESS":
            self.get_logger().info("Восьмёрка выполнена успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Восьмёрка провалена!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

def main():
    rclpy.init()
    node = BtEight()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        try:
            node.destroy_node()
        except Exception:
            pass
        if rclpy.ok():
            rclpy.shutdown()

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

2.3. Зачётное испытание

Задание:

1. Выберите одну из траекторий (змейка или восьмёрка) или придумайте свою.
2. Соберите дерево с `RetryAction` на критичных участках.
3. Запустите на полигоне.
4. **Преподаватель ставит препятствие** на пути робота.
5. Робот должен: застрять → `RetryAction` пробует 3 раза → если не вышло → `FAILURE` → `stop()`.

Критерии оценки:

- Дерево собрано корректно (все скобки, запятые).
- Heartbeat работает (нет потери связи).
- `RetryAction` срабатывает при застревании.
- После `FAILURE` робот останавливается.
- Ученик может объяснить, какой узел за что отвечает.

3. Уровень «Углублённый»: Анализ ошибок

(Что под капотом)

После испытания — разбор полётов.

3.1. Какое дерево упало, какое выполнилось

Запишите для каждой траектории:

- Сколько раз сработал `RetryAction`.
- На каком узле дерево вернуло `FAILURE`.
- Успел ли робот остановиться до столкновения.

3.2. MoveRelativeAction vs MoveTimeAction: где что уместнее

Участок траектории	Что лучше	Почему
Прямая на 1 м	`MoveRelativeAction`	Точная дистанция, контроль по одометрии
Поворот на 90°	`MoveRelativeAction(dyaw_deg=90)`	Точный угол
Дуга	`MoveTimeAction(vx, omega)`	Нельзя задать дугу через `dx/dy`
Отъезд «чуть-чуть»	`MoveTimeAction`	Не нужна точность, важна простота

Правило: если нужна **точность** — `MoveRelativeAction`. Если нужно **движение по времени** или **комбинированное** (дуга) — `MoveTimeAction`.

4. Уровень «Исследовательский»: Универсальный лаунчер

(Прорыв за границы)

Класс, который принимает любое дерево и запускает его.

4.1. Класс BTRunner

Идея: вынести общую логику (таймер, `_loop`, `is_connected`, `stop`) в отдельный класс, а дерево передавать параметром.

```
class BTRunner(RobotDriver):
    """
    Универсальный запускатель дерева поведения.
    Принимает корневой узел, тикает по таймеру с заданной частотой,
    обрабатывает SUCCESS/FAILURE, может выводить состояние дерева.
    """

    def __init__(self, root: BTNode, node_name: str = 'bt_runner', tick_ra
te: float = 5.0):
        super().__init__(node_name)
        self._bt_root = root
        self._active = False
        self._debug = False
        self._ticker = self.create_timer(1.0 / tick_rate, self._loop)
        self.get_logger().info(f"Дерево готово, тик {tick_rate} Гц")

    @property
    def debug(self) -> bool:
        return self._debug

    @debug.setter
    def debug(self, value: bool):
        self._debug = value

    def _loop(self):
        if not self.is_connected():
            return

        if not self._active:
            print("\nРобот готов к миссии.")
            input("Нажмите Enter для старта ... ")
            self.get_logger().info("Запуск дерева ... ")
            self._active = True
            return

        status = self._bt_root.tick(self)

        if self._debug:
            self._print_tree()
```

```
        if status == "SUCCESS":
            self.get_logger().info("Дерево выполнено успешно!")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())
        elif status == "FAILURE":
            self.get_logger().error("Дерево вернуло FAILURE. Аварийная
остановка.")
            self.stop()
            self.create_timer(0.5, lambda: rclpy.shutdown())

    def _print_tree(self, node=None, indent=0):
        if node is None:
            node = self._bt_root
        prefix = "  " * indent + ("→ " if indent > 0 else "")
        name = getattr(node, 'name', node.__class__.__name__)
        state = getattr(node, '_state', '')
        state_str = f" [{state}]" if state else ""
        self.get_logger().info(f"{prefix}{name}{state_str}")
        for child in getattr(node, 'children', []):
            self._print_tree(child, indent + 1)

    @staticmethod
    def run(node: 'BTRunner'):
        """Запустить ноду с корректной очисткой при выходе."""
        try:
            rclpy.spin(node)
        except KeyboardInterrupt:
            pass
        finally:
            try:
                node.destroy_node()
            except Exception:
                pass
            if rclpy.ok():
                rclpy.shutdown()
```

Использование:

```
def main():
    rclpy.init()
    tree = Parallel([
        HeartbeatAction(),
        Sequence([
            LogAction("Старт миссии"),
            MoveRelativeAction(dx=1.0, name="Вперёд"),
            MoveTimeAction(omega=0.5, duration=3.14, name="Поворот"),
        ]),
    ], required_successes=1)

    node = BTRunner(tree, 'my_mission', tick_rate=10)
    node.debug = True   # показать дерево в логах
    BTRunner.run(node)
```

Это избавляет от дублирования кода в каждом скрипте. Можно вынести `BTRunner` в отдельный модуль и переиспользовать...

Но! у меня для вас хорошая новость: нечто похожее уже реализовано в самом классе `RobotDriver`.

Пример использования:

```
#!/usr/bin/env python3
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, RetryAction,
                        MoveRelativeAction, LogAction)

class SquareMission(RobotDriver):
    def build_tree(self):
        side = 1.0
        return Sequence([
            LogAction(f"Квадрат {side}×{side}"),
            RetryAction(MoveRelativeAction(dx=side, name="Вперёд 1")),
            MoveRelativeAction(dyaw_deg=90.0, name="Поворот 1"),
            RetryAction(MoveRelativeAction(dx=side, name="Вперёд 2")),
            MoveRelativeAction(dyaw_deg=90.0, name="Поворот 2"),
            RetryAction(MoveRelativeAction(dx=side, name="Вперёд 3")),
            MoveRelativeAction(dyaw_deg=90.0, name="Поворот 3"),
            RetryAction(MoveRelativeAction(dx=side, name="Вперёд 4")),
            MoveRelativeAction(dyaw_deg=90.0, name="Поворот 4"),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(SquareMission('square', tick_rate=5))

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

Вот так просто можно проехать по квадрату на роботе с ROS2.

5. Подведение итогов блока «Программирование движения меканум робота с помощью Деревьев поведения»

Чему мы научились (занятия 26–35)

1. **Сервисы автопилота:** `go_time`, `go_relative`, `stop` — высокоуровневые команды.
2. **RobotDriver:** единый интерфейс к роботу из библиотеки.
3. **Деревья поведения:** модульный конструктор миссий.
4. **Базовые узлы:** `Sequence`, `Parallel`, `Fallback`, `Condition`, `RetryAction`.
5. **Действия:** `MoveRelativeAction`, `MoveTimeAction`, `StopAction`, `HeartbeatAction`, `LogAction`.
6. **Обработка ошибок:** `RetryAction` для восстановления, `FAILURE` для безопасной остановки.

Что дальше

Следующий большой блок — **«Сенсоры»**. Для чего они нужны.

Домашнее задание:

- **Базовое**: Повторите весь пройденный материал раздела.

Сенсоры: лидар и камера

Занятие 36: Зачем сенсоры? Лидар: устройство, первый запуск, визуализация

Содержание занятия: Проблема накопления ошибки одометрии. Лидар как источник обратной связи от среды

Цели занятия

- **Образовательная**: Понять фундаментальную проблему одометрии — накопление ошибки. Познакомиться с лидаром как средством получения объективных данных о среде.
- **Развивающая**: Научиться читать «сырые» данные лидара из топика `/scan` и визуализировать их в RViz.
- **Воспитательная**: Показать, что настоящая автономность начинается там, где робот **видит** мир, а не просто считает обороты колёс.

План занятия (90 минут)

1. **Организационный момент и введение (5 мин)**
2. **Уровень «База»: Проблема одометрии и лидар (40 мин)**
   - Диагностика: почему квадрат «разъезжается».
   - Обратная связь от среды — решение.
   - Лидар: что измеряет, как выглядит, параметры.
   - Топик `/scan`, ranges, angle_min, angle_increment.
   - Визуализация в RViz.
3. **Уровень «Углублённый»: Как работает лидар (25 мин)**
   - Принцип Time of Flight.
   - Почему лидар не видит стекло и чёрные поверхности.
   - Разрешение и нулевые значения.
4. **Уровень «Исследовательский»: Сравнение с ультразвуком и влияние цвета (15 мин)**
   - Лидар vs ультразвуковой дальномер.
   - Как цвет и текстура влияют на показания.
5. **Подведение итогов (5 мин)**

1. Введение: Почему робот не возвращается в начало

В блоке «Движение» мы проезжали квадрат с контролем по одометрии. Робот **думал**, что вернулся в точку (0, 0). Но рулетка показывала отклонение в 10–30 см.

Почему? Одометрия считает обороты колёс. Но:

- Колёса проскальзывают.
- Меканум-ролики уводят вбок.
- Неровности пола отклоняют траекторию.

Каждый метр ошибка накапливается. Через 10 метров робот может «думать», что он в точке А, а реально быть в точке Б.

Решение: дать роботу **«глаза»** — сенсоры, которые измеряют реальный мир. Главный из них — **лидар**.

2. Уровень «База»: Проблема одометрии и лидар

(Инструмент, навык, интерфейс)

Обязательный уровень. Знакомимся с лидаром и его данными.

2.1. Диагностика: квадрат разъезжается

Вспомните занятие 24 (квадрат по одометрии). Типичный результат:

- Одометрия: `x=0.00, y=0.00` (робот думает, что в начале).
- Рулетка: `x=0.15, y=-0.22` (реальное отклонение 27 см).

Это не ошибка программы. Это **фундаментальное ограничение** одометрии без внешних сенсоров.

Кибернетический принцип: для точного управления нужна **обратная связь от среды**. Не «я думаю, что я здесь», а «я вижу стену на расстоянии 1.2 м, значит я здесь».

2.2. Лидар: что это и как выглядит

Лидар (LIDAR — Light Detection and Ranging) — это лазерный дальномер, который сканирует пространство вокруг себя.

Как выглядит на роботе: небольшая вращающаяся башенка сверху. Найдите её.

Что измеряет: расстояние до ближайшего препятствия в каждом направлении. За один оборот (10 раз в секунду) делает сотни измерений.

Параметры нашего лидара:

- Дальность: до 12 м.
- Угол обзора: 360°.
- Частота: 10 Гц (10 оборотов в секунду).
- Точек на оборот: ~1000.

2.3. Топик /scan и его поля

Учитель уже запустил стек (лидар работает). Подключитесь по SSH и посмотрите данные:

```
ros2 topic echo /scan --once
```

Вы увидите сообщение `LaserScan` с полями:

- `angle_min` — начальный угол сканирования (рад).
- `angle_max` — конечный угол.
- `angle_increment` — шаг между измерениями (рад).
- `ranges` — массив расстояний (метры).

Практика:

```
# Сколько точек в одном скане?
ros2 topic echo /scan --once

# Частота публикации
ros2 topic hz /scan
```

2.4. Визуализация в RViz

Учитель запускает RViz на доске. В нём:

- Добавьте дисплей `LaserScan`.
- Укажите топик `/scan`.
- Вы увидите **облако точек** — красные точки вокруг робота. Это стены полигона, препятствия, люди.

Задание: Подойдите к роботу с разных сторон. Смотрите на RViz — точки меняются. Это лидар «видит» вас.

3. Уровень «Углублённый»: Как работает лидар

(Что под капотом)

Разбираем физику и ограничения.

3.1. Принцип Time of Flight (ToF)

Лидар работает как эхолот, но с лазером:

1. Лазерный диод испускает короткий инфракрасный импульс.
2. Импульс летит до препятствия, отражается и возвращается.
3. Высокоточный приёмник фиксирует отражённый сигнал.
4. Расстояние: `D = c × t / 2`, где `c` — скорость света (≈300 000 км/с).

Почему «/ 2»? Луч проходит расстояние туда и обратно.

Время полёта для объекта на расстоянии 1 м: `t = 2 × 1 / 300 000 000 ≈ 6.7 нс`. Лидар измеряет наносекунды!

3.2. Почему лидар не видит стекло и чёрные поверхности

Стекло: инфракрасный луч проходит сквозь стекло почти без отражения. Лидар «видит» то, что за стеклом, или не видит ничего.

Чёрная матовая поверхность: поглощает лазерный луч. Отражённый сигнал слишком слабый, приёмник его не фиксирует.

Вывод: лидар измеряет расстояние до **отражающих** поверхностей. Прозрачные и поглощающие объекты — «невидимки».

3.3. Разрешение и нулевые значения

Угловое разрешение: `angle_increment = 0.0061 рад ≈ 0.35°`. Это значит, что на 1 градус приходится ~2.8 точки.

Нулевые значения в ranges:

- `0.0` означает, что измерение не удалось (нет отражения, ошибка).
- В библиотеке `driver_lib` такие значения **отбрасываются** (фильтр `v > 0.01`).
- Но если несколько нулей подряд — это может быть реальное препятствие (чёрное), а не шум.

4. Уровень «Исследовательский»: Сравнение с ультразвуком и влияние цвета

(Прорыв за границы)

Эксперименты и анализ.

4.1. Лидар vs ультразвуковой дальномер

Характеристика	Лидар	Ультразвук
Принцип	Лазер (свет)	Звук
Дальность	До 40 м	До 4 м
Угловое разрешение	~0.3°	~15° (широкий конус)
Видит стекло?	Нет	Да
Видит чёрное?	Плохо	Да
Помехоустойчивость	Высокая	Низкая (эхо, перекрёстные помехи)

Вывод: лидар точнее, но ультразвук дополняет его для прозрачных объектов.

4.2. Мини-исследование: влияние цвета и текстуры

Эксперимент:

1. Поставьте перед роботом листы бумаги: белый, серый, чёрный.
2. Замерьте показания `get_lidar_front()` для каждого (через будущую миссию `SensorTest`).
3. Запишите расстояния.

Ожидаемый результат:

- Белый лист — чёткое расстояние.
- Чёрный лист — расстояние может быть завышено или `inf` (не видит).

Вывод: лидар «видит» отражающую способность, а не геометрию.

5. Выводы по занятию

1. **Одометрия накапливает ошибку.** Без сенсоров робот теряет своё положение.
2. **Лидар — лазерный дальномер 360°.** Измеряет расстояние до препятствий с высоким разрешением.
3. **Данные лидара доступны в `/scan`.** Массив `ranges` + угловые метаданные.
4. **Лидар не идеален.** Стекло и чёрные поверхности — его слабость.
5. **Визуализация в RViz** превращает массив чисел в понятную картину.

Связь со следующим занятием: Теперь мы знаем, какие данные даёт лидар. На следующем занятии мы будем **получать их из кода** через геттеры `RobotDriver`: `get_lidar_front()`, `get_lidar_sectors()` и выводить в лог.

Домашнее задание:

- **Базовое**: Изучите вывод `ros2 topic echo /scan --once`. Найдите `angle_min` и `angle_increment`.
- **Продвинутое**: Посчитайте, сколько точек в полном скане: `(angle_max - angle_min) / angle_increment`.
- **Исследовательское**: Проведите эксперимент с листами бумаги разного цвета. Запишите расстояния.

Занятие 37: Геттеры лидара в RobotDriver: первый тест без движения

Содержание занятия: Получение данных лидара из кода через готовые методы RobotDriver

Цели занятия

- **Образовательная**: Научиться использовать геттеры лидара в `RobotDriver`: `get_lidar_front()`, `get_lidar_right()`, `get_lidar_left()`, `get_lidar_back()`, `get_lidar_min_distance()`, `get_lidar_sectors()`.
- **Развивающая**: Написать первую миссию с `EvalAction` для вывода данных сенсора в лог.
- **Воспитательная**: Показать, что библиотека уже содержит всё необходимое — нужно только научиться этим пользоваться.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия SensorTest (45 мин)**
   - Обзор геттеров лидара и системы координат.
   - `EvalAction` — узел, вычисляющий и логирующий функцию.
   - `build_tree()` с `EvalAction` для вывода всех секторов.
   - Запуск, наблюдение за данными.
3. **Уровень «Углублённый»: Как работают геттеры (25 мин)**
   - `_lidar_index()`, `_lidar_min_in_range()`.
   - Расчёт направления, фильтрация нулей.
4. **Уровень «Исследовательский»: Визуализация сырых данных (10 мин)**
   - Идея: matplotlib + полярные координаты.
   - Шум в зависимости от расстояния.
5. **Подведение итогов (5 мин)**

1. Введение: От топика — к методам

На прошлом занятии мы смотрели сырые данные лидара через `ros2 topic echo /scan`. Но для миссий нам не нужен массив из 1000 чисел. Нам нужно знать:

- Что спереди? На каком расстоянии?
- Что справа? Слева? Сзади?

Библиотека `driver_lib` уже содержит **геттеры лидара** — методы, которые отвечают на эти вопросы. Сегодня мы научимся их использовать.

2. Уровень «База»: Миссия SensorTest

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем миссию, которая выводит данные лидара в лог.

2.1. Геттеры лидара в RobotDriver и система координат

Система координат ROS2:

- **X** — вперёд (угол 0°).
- **Y** — влево (угол 90° = +π/2).
- **-Y** — вправо (угол -90° = -π/2).
- **-X** — назад (угол ±180° = ±π).

Геттеры лидара и их секторы:

Метод	Сектор	Описание
`get_lidar_front(width_deg=30)`	0° ± 15°	Вперёд
`get_lidar_left(width_deg=20)`	90° ± 10° (π/2)	Влево
`get_lidar_right(width_deg=20)`	-90° ± 10° (-π/2)	Вправо
`get_lidar_back(width_deg=20)`	±180° ± 10° (±π)	Назад
`get_lidar_min_distance()`	360°	Глобальный минимум
`get_lidar_sectors(front_w, side_w)`	—	Словарь `{"front", "left", "right", "back"}`

Параметр `width_deg` — ширина сектора в градусах. Чем шире сектор, тем больше шансов заметить препятствие, но меньше точность определения его направления.

В коде:

```
# get_lidar_left ищет минимум в диапазоне π/2 ± 10° (влево)
def get_lidar_left(self, width_deg=20):
    w = math.radians(width_deg / 2)
    return self._lidar_min_in_range(math.pi/2 - w, math.pi/2 + w)

# get_lidar_right ищет минимум в диапазоне -π/2 ± 10° (вправо)
def get_lidar_right(self, width_deg=20):
    w = math.radians(width_deg / 2)
    return self._lidar_min_in_range(-math.pi/2 - w, -math.pi/2 + w)
```

2.2. EvalAction — узел для вывода произвольных данных

`EvalAction` — это узел, который принимает **функцию** и логирует её результат. Идеален для отладки сенсоров.

```
EvalAction(
    lambda d: f"Спереди: {d.get_lidar_front():.2f} м",
    name="Перед"
)
```

На каждом `tick` (пока не выполнен) он вызывает функцию и выводит результат в лог. После первого же тика возвращает `SUCCESS`.

2.3. Миссия SensorTest

Создайте файл:

```
nano ~/ws/rbt/scripts/sensor_test.py
```

```
#!/usr/bin/env python3
"""Миссия: вывод данных лидара без движения."""
import sys
import rclpy
from driver_lib import RobotDriver, Sequence, EvalAction

class SensorTest(RobotDriver):
    def build_tree(self):
        return Sequence([
            EvalAction(
                lambda d: f"Спереди (0°): {d.get_lidar_front():.2f} м",
                name="Перед"
            ),
            EvalAction(
                lambda d: f"Слева (90°): {d.get_lidar_left():.2f} м",
                name="Лево"
            ),
            EvalAction(
                lambda d: f"Справа (-90°): {d.get_lidar_right():.2f} м",
                name="Право"
            ),
            EvalAction(
                lambda d: f"Сзади (±180°): {d.get_lidar_back():.2f} м",
                name="Тыл"
            ),
            EvalAction(
                lambda d: f"Глобальный минимум: {d.get_lidar_min_distance(
):.2f} м",
                name="Минимум"
            ),
            EvalAction(
                lambda d: f"Все сектора: {d.get_lidar_sectors()}",
                name="Все"
            ),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(SensorTest('sensor_test', tick_rate=1))
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/sensor_test.py
```

Ожидаемый вывод:

```
[BT] Перед: Спереди (0°): 2.45 м
[BT] Лево: Слева (90°): 3.12 м
[BT] Право: Справа (-90°): 0.87 м
[BT] Тыл: Сзади (±180°): 1.55 м
[BT] Минимум: Глобальный минимум: 0.87 м
[BT] Все: Все сектора: {'front': 2.45, 'left': 3.12, 'right': 0.87,
'back': 1.55}
```

Эксперимент: подойдите к роботу с разных сторон. Перезапустите миссию. Значения изменятся.

2.4. Параметр tick_rate

Обратите внимание: `tick_rate=1` (1 Гц). Для отладочной миссии этого достаточно — данные лидара обновляются 10 Гц, но нам не нужно выводить их 10 раз в секунду.

В боевых миссиях (остановка перед препятствием) используется `tick_rate=10` для минимальной задержки реакции.

3. Уровень «Углублённый»: Как работают геттеры

(Что под капотом)

Заглядываем внутрь методов.

3.1. _lidar_index() и _lidar_min_in_range()

Геттеры используют два внутренних метода:

`_lidar_index(angle_rad)` — переводит угол в индекс массива `ranges`:

```
index = (angle_rad - angle_min) / angle_increment
```

`_lidar_min_in_range(start_rad, end_rad)` — находит минимум в заданном диапазоне углов, отбрасывая нули (`v > 0.01`).

Направление по индексу:

```
angle = angle_min + index × angle_increment
```

Индекс 0 соответствует углу `angle_min` (обычно -π, т.е. направление назад). Индекс 500 — углу `angle_min + 500 × angle_increment`.

3.2. Почему нулевые значения — не всегда шум

Фильтр отбрасывает значения `v ⩽ 0.01`. Но:

- **Одиночный 0** — вероятно, ошибка измерения.
- **Несколько 0 подряд** — возможно, чёрная матовая поверхность или стекло.
- **Все 0 в секторе** — сектор пуст (далеко до стены) или препятствие поглощает луч.

Геттер `_lidar_min_in_range` в случае всех нулей возвращает `float('inf')` — «бесконечно далеко».

4. Уровень «Исследовательский»: Визуализация сырых данных

(Прорыв за границы)

Идеи для самостоятельного расширения.

4.1. Визуализация в полярных координатах

Можно написать отдельную ноду, которая подписывается на `/scan` и строит полярный график через `matplotlib`. Это даст «вид сверху», похожий на RViz, но в коде.

```
import matplotlib.pyplot as plt
angles = [angle_min + i * angle_increment for i in range(len(ranges))]
plt.polar(angles, ranges, 'r.')
plt.show()
```

4.2. Мини-исследование: шум и расстояние

Вопрос: Как уровень шума (разброс значений `ranges`) зависит от расстояния до объекта?

Методика:

1. Поставьте робота на расстоянии 0.5, 1, 2, 5 м от стены.
2. Для каждого расстояния запишите 10 последовательных значений `get_lidar_front()`.
3. Вычислите стандартное отклонение.

Гипотеза: на больших расстояниях шум выше (сигнал слабее).

5. Выводы по занятию

1. **Геттеры лидара — простой интерфейс к сложным данным.** Вместо массива `ranges` — готовые ответы: «спереди 2.45 м», «слева 3.12 м».
2. **Система координат:** 0° — вперёд, 90° (π/2) — влево, -90° (-π/2) — вправо, ±180° (±π) — назад.
3. **`EvalAction` — отладочный узел.** Вызывает функцию и логирует результат. Идеален для первых тестов.

4. **Новый стиль миссий:** `build_tree()` → `Sequence` из узлов, запуск через `RobotDriver.spin()`.
5. **`tick_rate` управляет частотой.** Для отладки — 1 Гц, для движения — 5–10 Гц.

Связь со следующим занятием: Мы умеем получать данные лидара. Теперь превратим их в **условия** для дерева поведения: `Condition` с лямбдой «если спереди меньше 0.5 м». И добавим **сегментацию по секторам**.

Домашнее задание:

- **Базовое**: Запустите `sensor_test.py` в разных местах полигона. Запишите показания.
- **Продвинутое**: Добавьте `EvalAction` для вывода `get_lidar_front(60)` — расширенный сектор. Сравните с узким.
- **Исследовательское**: Проведите замер шума на разных расстояниях. Постройте график.

Занятие 38: Сегментация данных лидара по секторам и Condition-узлы

Содержание занятия: Превращение данных лидара в условия для дерева поведения

Цели занятия

- **Образовательная**: Научиться использовать `get_lidar_sectors()` и создавать `Condition`-узлы на основе геттеров лидара.
- **Развивающая**: Понять, как дерево поведения принимает решения на основе сенсорных данных.
- **Воспитательная**: Показать связку «сенсор → условие → действие» как основу автономного поведения.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Сектора и Condition (45 мин)**
   - `get_lidar_sectors()` — словарь четырёх направлений.
   - `Condition` с лямбдой: `lambda d: d.get_lidar_front() < 0.5`.
   - Миссия `ObstacleCheck`: робот стоит, выводит статус.
3. **Уровень «Углублённый»: Как устроены сектора (25 мин)**
   - Границы секторов в индексах массива `ranges`.
   - Почему передний сектор шире заднего.
   - Параметры `front_width` и `side_width`.
4. **Уровень «Исследовательский»: Динамические сектора и Blackboard (10 мин)**
   - Изменение ширины сектора от скорости.
   - Сколько секторов оптимально.

5. **Подведение итогов (5 мин)**

1. Введение: От данных — к решениям

На прошлом занятии мы выводили данные лидара в лог. Но роботу недостаточно «видеть» цифры — он должен **принимать решения** на их основе.

Сегодня мы свяжем два изученных инструмента:

- **Геттеры лидара** — источники данных о среде.
- **`Condition`-узлы** — механизм ветвления в дереве поведения.

Вместе они образуют **сенсорно-управляемое поведение**: «ЕСЛИ спереди меньше 0.5 м, ТО остановиться».

2. Уровень «База»: Сектора и Condition

(Инструмент, навык, интерфейс)

Обязательный уровень. Создаём условия на основе лидара.

2.1. get_lidar_sectors() — вся картина одним словарём

Метод `get_lidar_sectors(front_width=30, side_width=20)` возвращает словарь:

```
{
    "front": 2.45,   # мин. расстояние в секторе 0° ± 15°
    "left": 3.12,    # мин. расстояние в секторе 90° ± 10°
    "right": 0.87,   # мин. расстояние в секторе -90° ± 10°
    "back": 1.55     # мин. расстояние в секторе ±180° ± 10°
}
```

Параметры:

- `front_width=30` — сектор впереди шириной 30° (±15°).
- `side_width=20` — секторы сбоку и сзади шириной 20° (±10°).

Это удобно: одно обращение к методу — и у вас полная информация о ближайших препятствиях со всех четырёх сторон.

2.2. Condition с лямбдой — условие на основе лидара

`Condition` принимает функцию, которая получает `driver` и возвращает `True` / `False`:

```
# Препятствие спереди ближе 0.5 м?
Condition(lambda d: d.get_lidar_front() < 0.5)

# Справа больше метра свободно?
Condition(lambda d: d.get_lidar_right() > 1.0)
```

```
# Препятствие сзади ближе 0.3 м?
Condition(lambda d: d.get_lidar_back() < 0.3)
```

Как это работает: на каждом `tick` `Condition` вызывает лямбду. Если условие истинно — возвращает `SUCCESS`, иначе `FAILURE`.

2.3. Миссия ObstacleCheck

Создайте файл:

```
nano ~/ws/rbt/scripts/obstacle_check.py
```

```
#!/usr/bin/env python3
"""Миссия: проверка препятствий по лидару."""
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence, Condition, LogAction, Fallb
ack)

class ObstacleCheck(RobotDriver):
    def build_tree(self):
        return Sequence([
            # Проверка спереди
            Fallback([
                Sequence([
                    Condition(lambda d: d.get_lidar_front() < 0.5),
                    LogAction("   Препятствие спереди!"),
                ]),
                LogAction("   Спереди свободно"),
            ]),
            # Проверка слева
            Fallback([
                Sequence([
                    Condition(lambda d: d.get_lidar_left() < 0.5),
                    LogAction("   Препятствие слева!"),
                ]),
                LogAction("   Слева свободно"),
            ]),
            # Проверка справа
            Fallback([
                Sequence([
                    Condition(lambda d: d.get_lidar_right() < 0.5),
                    LogAction("   Препятствие справа!"),
                ]),
                LogAction("   Справа свободно"),
            ]),
            # Все сектора разом
            Fallback([
                Sequence([
                    Condition(lambda d: d.get_lidar_front() < 0.5 or
                                        d.get_lidar_left() < 0.5 or
                                        d.get_lidar_right() < 0.5),
                    LogAction("   Есть препятствия рядом!"),
                ]),
                LogAction("   Всё чисто вокруг"),
```

```
            ]),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(ObstacleCheck('obstacle_check', tick_rate=1))

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

Запуск:

```
python3 ~/ws/rbt/scripts/obstacle_check.py
```

Эксперимент: подойдите к роботу спереди, слева, справа. Наблюдайте за выводом. `Fallback` + `Sequence` + `Condition` образуют «если ... то ... иначе ...».

2.4. Шаблон «если ... то ... иначе» на деревьях

```
Fallback([
    Sequence([Condition(проверка), LogAction("Да")]),  # пробуем это
    LogAction("Нет"),  # если первое провалилось — это
])
```

Если `Condition` возвращает `SUCCESS` → `Sequence` продолжается → `LogAction("Да")` → `SUCCESS` → `Fallback` возвращает `SUCCESS`.

Если `Condition` возвращает `FAILURE` → `Sequence` возвращает `FAILURE` → `Fallback` пробует второго ребёнка → `LogAction("Нет")` → `SUCCESS`.

3. Уровень «Углублённый»: Как устроены сектора

(Что под капотом)

Индексы, ширина, кластеризация.

3.1. Границы секторов в массиве ranges

Сектор «спереди» (0° ± 15°):

```
start_idx = _lidar_index(-15°)
end_idx = _lidar_index(+15°)
min(ranges[start_idx : end_idx+1])
```

Сектор «справа» (-90° ± 10°):

```
start_idx = _lidar_index(-100°)
end_idx = _lidar_index(-80°)
min(ranges[start_idx : end_idx+1])
```

Почему передний сектор шире (30°) чем задний (20°)? Робот обычно движется вперёд. Широкий передний сектор позволяет заметить препятствие на траектории заранее. Узкие боковые секторы дают более точное направление на препятствие.

3.2. Простейшая кластеризация: объединение соседних точек

Текущая реализация `_lidar_min_in_range` просто находит минимум. Это работает для обнаружения **ближайшего** препятствия в секторе.

Но для более сложных задач (например, «сколько объектов в секторе?») нужна **кластеризация** — объединение соседних точек, расстояние между которыми меньше порога. Это тема будущих занятий по OpenCV.

4. Уровень «Исследовательский»: Динамические сектора и Blackboard

(Прорыв за границы)

Идеи для продвинутых сценариев.

4.1. Динамическая ширина сектора от скорости

На высокой скорости нужно «смотреть дальше вперёд» — расширять передний сектор. На низкой — можно сузить.

Идея: хранить текущую скорость в `Blackboard` и использовать её в лямбде:

```
bb = Blackboard()
bb.set("speed", 0.2)
Condition(lambda d: d.get_lidar_front(width_deg=30 + bb.get("speed")*50)
< 0.5)
```

При скорости 0.2 м/с ширина сектора = 30 + 10 = 40°. При 0.5 м/с = 30 + 25 = 55°.

4.2. Мини-исследование: 4, 8 или 16 секторов

Вопрос: Сколько секторов оптимально для навигации в лабиринте?

Гипотеза: 4 сектора (front/left/right/back) достаточно для правила правой руки. 8 секторов (добавить диагонали) могут помочь при объезде. 16 — избыточно для учебного робота.

Проверка: напишите `get_lidar_sectors_8()` с диагоналями и сравните поведение в лабиринте.

5. Выводы по занятию

1. **`get_lidar_sectors()` — готовая картина мира.** Четыре направления, одно обращение.
2. **`Condition` + геттер — основа автономного поведения.** «Если спереди меньше 0.5 м → стоп».
3. **`Fallback` + `Sequence` + `Condition` = «если ... то ... иначе».** Универсальный шаблон ветвления.
4. **Передний сектор шире боковых** — робот движется вперёд.

Связь со следующим занятием: Мы умеем проверять условия. Теперь добавим **движение**: `StreamAction` будет ехать вперёд, а `Condition` в `Parallel` — следить за препятствием и вызывать `StopAction`. Первая миссия с сенсором и движением!

Домашнее задание:

- **Базовое**: Запустите `obstacle_check.py` с препятствиями с разных сторон.
- **Продвинутое**: Добавьте проверку «препятствие сзади ближе 0.3 м».
- **Исследовательское**: Напишите `get_lidar_sectors_8()` с диагоналями (45°, 135°, -45°, -135°).

Занятие 39: Остановка перед препятствием по лидару: первая BT-миссия с сенсором

Содержание занятия: Соединение движения и сенсора. Робот едет и сам останавливается перед стеной

Цели занятия

- **Образовательная**: Собрать первую миссию, в которой робот движется и одновременно следит за лидаром, останавливаясь перед препятствием.
- **Развивающая**: Освоить узел `StreamAction` для непрерывного движения и `WaitCondition` для ожидания сенсорного события.
- **Воспитательная**: Показать, что даже простая миссия «ехать до стены» требует правильной архитектуры (гистерезис, время реакции).

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия StopBeforeObstacle (45 мин)**
   - `StreamAction` для непрерывного движения.
   - `WaitCondition` для ожидания сенсорного события.
   - `Parallel`: движение + мониторинг.
   - Сборка дерева, запуск, замер точности.
3. **Уровень «Углублённый»: Почему WaitCondition, а не Condition (25 мин)**
   - Разница между мгновенной проверкой и ожиданием.
   - Два контура остановки: штатный и страховочный.
   - Частота тика и точность.
4. **Уровень «Исследовательский»: Тормозной путь и адаптивный порог (10 мин)**
   - Замер тормозного пути на разных скоростях.
   - Идея адаптивного порога.
5. **Подведение итогов (5 мин)**

1. Введение: Робот, который видит и реагирует

На прошлом занятии мы проверяли данные лидара, но робот стоял на месте. Сегодня мы соединим **движение** и **сенсор** в одной миссии.

Задача: робот едет вперёд и **сам** останавливается, когда лидар видит стену ближе 30 см. Никакого `sleep`, никаких «проехать 2 метра». Робот реагирует на **реальный мир**.

Для этого нам понадобятся два новых инструмента:

- **`StreamAction`** — отправляет команды движения **непрерывно**, пока не сработает условие.
- **`WaitCondition`** — ждёт, пока условие станет истинным (в отличие от `Condition`, который проверяет мгновенно).

2. Уровень «База»: Миссия StopBeforeObstacle

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем миссию и тестируем на полигоне.

2.1. StreamAction — непрерывное движение

`StreamAction` — это узел, который на каждом `tick` вызывает функцию `command_func(driver)` и отправляет результат как `go_time`. Он возвращает `RUNNING` до тех пор, пока `stop_condition` не вернёт `True`.

```
StreamAction(
    command_func=lambda d: (0.2, 0.0, 0.0, 1.0),  # vx, vy, omega,
duration
    stop_condition=lambda d: d.get_lidar_front() < 0.3,  # когда
остановиться
    name="Еду вперёд"
)
```

Параметры `command_func`: отправляет кортеж `(vx, vy, omega, duration)` — те же поля, что и в `go_time`. Duration = 1.0 для корректности должно быть больше периода тика. `StreamAction` вызывает `driver.move_time(vx, vy, omega, duration)` на каждом тике, и параметр `duration` — это длительность команды `go_time`, которая отправляется автопилоту.

Поскольку автопилот перезаписывает цель при повторном вызове, `duration` должно быть больше периода тика, чтобы команда не истекала между тиками. При `tick_rate=10` (период 0.1 с), `duration=1.0` даёт 10-кратный запас. При `tick_rate=5` (0.2 с) — 5-кратный.

`stop_condition`: вызывается на каждом тике. Когда возвращает `True`, `StreamAction` вызывает `stop()` и возвращает `SUCCESS`.

2.2. WaitCondition — ожидание события

`Condition` проверяет условие **один раз** и сразу возвращает `SUCCESS` или `FAILURE`. Но нам нужно **ждать**, пока расстояние не станет меньше 30 см.

`WaitCondition` делает именно это:

```
WaitCondition(
    check_func=lambda d: d.get_lidar_front() < 0.3,
    timeout=3000.0,  # большой таймаут, чтобы точно доехать
    name="Жду стену"
)
```

На каждом `tick` вызывает `check_func`. Пока `False` — возвращает `RUNNING`. Когда `True` — возвращает `SUCCESS`. Если за `timeout` секунд условие не выполнилось — возвращает `FAILURE`.

2.3. Архитектура: два контура остановки

```
Parallel (required_successes=1)
├── StreamAction                    ← штатная остановка по stop_conditior
└── Sequence                        ← страховочный контур
    ├── WaitCondition (ждёт стену)  ← ждёт, пока расстояние < 0.3 м
    ├── LogAction ("Стена!")
    └── StopAction                  ← экстренный стоп
```

Два контура:

1. **Штатный:** `StreamAction` сам проверяет `stop_condition` на каждом тике и останавливается.
2. **Страховочный:** `WaitCondition` + `StopAction`. Если `StreamAction` по какой-то причине не остановился, этот контур заметит стену и вызовет экстренный `stop`.

`required_successes=1` означает, что `Parallel` завершится, когда **любой** из детей вернёт `SUCCESS`. Первым это сделает либо `StreamAction` (штатно), либо `Sequence` (страховочно).

2.4. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/stop_before_obstacle.py
```

```
#!/usr/bin/env python3
"""Миссия: ехать вперёд, остановиться перед стеной."""
import sys
import rclpy
from driver_lib import (RobotDriver, Parallel, Sequence,
                        StreamAction, Condition, StopAction,
                        LogAction, WaitCondition)

class StopBeforeObstacle(RobotDriver):
    def build_tree(self):
        return Parallel([
            # Штатный контур: движение с остановкой по лидару
```

```
            StreamAction(
                command_func=lambda d: (0.2, 0.0, 0.0, 1.0),
                stop_condition=lambda d: d.get_lidar_front() < 0.3,
                name="Еду вперёд"
            ),
            # Страховочный контур: ждём стену и экстренно стоп
            Sequence([
                WaitCondition(
                    check_func=lambda d: d.get_lidar_front() < 0.3,
                    timeout=3000.0,
                    name="Жду стену"
                ),
                LogAction("   Стена! Останавливаюсь."),
                StopAction(name="Экстренный стоп"),
            ]),
        ], required_successes=1)

def main():
    rclpy.init()
    RobotDriver.spin(StopBeforeObstacle('stop_before_obstacle',
tick_rate=10))

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

Запуск:

```
python3 ~/ws/rbt/scripts/stop_before_obstacle.py
```

Тест: поставьте робота в метре от стены. Запустите миссию. Робот должен доехать и остановиться на расстоянии ~30 см.

2.5. Замер точности остановки

1. Замерьте рулеткой **реальное** расстояние от робота до стены после остановки.
2. Сравните с порогом 30 см.
3. Повторите 3 раза. Запишите разброс.

Сравнение со «слепым» движением: если бы мы использовали `MoveRelativeAction(dx=1)`, точность зависела бы от одометрии. Здесь точность зависит **только от лидара**, который меряет реальное расстояние.

3. Уровень «Углублённый»: Почему WaitCondition, а не Condition

(Что под капотом)

Разбираем ключевое отличие.

3.1. Condition — мгновенная проверка

```
Condition(lambda d: d.get_lidar_front() < 0.3)
```

На первом же `tick` вызывает лямбду. Если стена далеко (2 метра), условие **ложно** → `Condition` возвращает `FAILURE` → `Sequence` возвращает `FAILURE` → `Parallel` видит `FAILURE` → **миссия провалена мгновенно**, робот даже не начал ехать!

Вывод: `Condition` годится для проверок, которые должны сработать **сразу** (например, «батарея заряжена?»). Для ожидания **события в будущем** нужен `WaitCondition`.

3.2. WaitCondition — терпеливое ожидание

```
WaitCondition(
    check_func=lambda d: d.get_lidar_front() < 0.3,
    timeout=3000.0,
    name="Жду стену"
)
```

На каждом `tick` вызывает лямбду. Пока стена далеко — возвращает `RUNNING` (продолжаем ждать). Когда стена приблизится — возвращает `SUCCESS`.

Сравнение:

Характеристика	`Condition`	`WaitCondition`
Когда срабатывает	Мгновенно на первом тике	Когда условие станет истинным
Возвращает	`SUCCESS` или `FAILURE` сразу	`RUNNING` пока ждёт, потом `SUCCESS` / `FAILURE`
Для чего	Проверка «сейчас»	Ожидание события «в будущем»
Пример	«Батарея > 20%?»	«Доехать до стены»

3.3. Частота тика и точность остановки

При `tick_rate=10` (10 Гц) между проверками лидара проходит **100 мс**.

При скорости 0.2 м/с за 100 мс робот проезжает **2 см**. Это минимальная погрешность остановки из-за дискретизации.

Формула: `погрешность = скорость × (1 / tick_rate)`.

tick_rate	Погрешность при 0.2 м/с	Погрешность при 0.5 м/с
5 Гц	4 см	10 см
10 Гц	2 см	5 см
20 Гц	1 см	2.5 см

Вывод: чем выше `tick_rate`, тем точнее остановка, но выше нагрузка на CPU.

4. Уровень «Исследовательский»: Тормозной путь и адаптивный порог

(Прорыв за границы)

Эксперименты с динамическими параметрами.

4.1. Замер тормозного пути

Вопрос: Какое расстояние робот проезжает после того, как лидар увидел стену и сработал `stop`?

Методика:

1. Запустите миссию на разных скоростях (меняйте `vx` в `command_func`: 0.1, 0.2, 0.3).
2. Для каждой скорости замерьте реальное расстояние до стены после остановки.
3. Вычтите из него порог 0.3 м — это «перелёт».

Скорость, м/с	Порог, м	Реальное расстояние, м	Перелёт, см
0.1	0.3		
0.2	0.3		
0.3	0.3		

4.2. Идея адаптивного порога

Если на скорости 0.3 м/с перелёт составляет 8 см, значит, порог нужно **увеличить** на 8 см: `0.3 + 0.08 = 0.38 м`.

Продвинутая идея: хранить скорость в `Blackboard` и вычислять порог динамически:

```
stop_condition=lambda d: d.get_lidar_front() < 0.3 + bb.get("speed") * 0.4
```

При 0.2 м/с порог = 0.38 м, при 0.1 м/с = 0.34 м.

5. Выводы по занятию

1. **`StreamAction` — непрерывное движение под контролем сенсора.** Едет, пока `stop_condition` не сработает.
2. **`WaitCondition` ждёт событие.** В отличие от `Condition`, не проваливается мгновенно, а терпеливо ждёт.
3. **Два контура остановки — надёжность.** Штатный в `StreamAction` + страховочный через `WaitCondition` + `StopAction`.
4. **Частота тика влияет на точность.** 10 Гц → погрешность ~2 см. Выше частота — выше точность.

Связь со следующим занятием: Мы использовали лидар. Но у робота есть ещё один сенсор — **камера**. На следующем занятии мы познакомимся с её устройством.

Домашнее задание:

- **Базовое**: Запустите `stop_before_obstacle.py` 3 раза. Запишите расстояние остановки.
- **Продвинутое**: Измените скорость на 0.3 м/с и порог на 0.5 м. Сработала ли остановка?
- **Исследовательское**: Проведите замер перелёта для 3 скоростей. Постройте график.

Занятие 40: Камера: устройство, запуск, калибровка (на оригинальном роботе)

Содержание занятия: Второй глаз робота. Как работает камера и почему её нужно калибровать

Цели занятия

- **Образовательная**: Познакомиться с устройством камеры, её характеристиками. Научиться просматривать видеопоток и проводить калибровку.
- **Развивающая**: Понять природу дисторсии и необходимость калибровки для точных измерений.
- **Воспитательная**: Показать, что сенсоры требуют настройки — идеальных устройств не бывает.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Камера и её данные (40 мин)**
   - Устройство: матрица, разрешение, угол обзора, фокус.
   - Топик `/camera/image_raw`.
   - Просмотр через `http://172.22.17.30:8000/`.
   - Калибровка шахматной доской.
3. **Уровень «Углублённый»: Дисторсия и матрица камеры (25 мин)**
   - Бочка и подушка.
   - Матрица камеры и коэффициенты дисторсии.
   - Почему важна перекалибровка.
4. **Уровень «Исследовательский»: Качество калибровки (15 мин)**
   - Сколько кадров нужно.
   - Влияние освещения.
5. **Подведение итогов (5 мин)**

1. Введение: Лидар — не единственный глаз

Лидар отлично измеряет расстояния, но он не различает **смысл** объектов. Стена, человек, знак — для лидара это просто «препятствие на расстоянии X».

Камера дополняет лидар: она видит цвет, текстуру, форму. С помощью камеры робот может:

- Распознавать ArUco-маркеры для точного позиционирования.
- Следовать по цветной линии.
- Детектировать людей и знаки через нейросеть.

Но прежде чем использовать камеру, её нужно **откалибровать** — исправить искажения, которые вносит линза.

2. Уровень «База»: Камера и её данные

(Инструмент, навык, интерфейс)

Обязательный уровень. Знакомимся с железом, топиком и калибровкой.

2.1. Устройство камеры: ключевые характеристики

Найдите камеру на роботе. Обычно она закреплена спереди и направлена вперёд.

Основные параметры:

Параметр	Что означает	Наша камера
Матрица	Светочувствительный сенсор (CMOS)	Разрешение 1920×1080 (Full HD)
Разрешение	Количество пикселей по горизонтали × вертикали	2 мегапикселя
Угол обзора	Сколько пространства видно (по горизонтали)	~160° (не 360° как у лидара!)
Фокусное расстояние	Определяет угол обзора и масштаб	Чем меньше, тем шире угол
Частота кадров	Сколько кадров в секунду	30 FPS

Важно: камера видит только то, что **перед ней**. В отличие от лидара, у неё нет обзора 360°.

2.2. Топик /camera/image_raw и просмотр видео

Учитель запустил стек (камера работает). Данные публикуются в топик `/camera/image_raw`.

Проверьте:

```
ros2 topic hz /camera/image_raw
# Должно быть ~30 Гц

ros2 topic info /camera/image_raw
# Тип: sensor_msgs/msg/Image
```

Просмотр видеострима (учитель запускает на доске):

```
rqt_image_view /camera/image_raw
http://172.22.17.30:8000/
```

Вы увидите то, что «видит» робот. Помашите рукой перед камерой — изображение изменится.

Обратите внимание на края изображения: прямые линии (край стола, стена) могут быть **изогнуты**. Это дисторсия — с ней мы будем бороться.

2.3. Калибровка шахматной доской

Зачем: исправить искажения линзы, чтобы по пикселям можно было точно определить направление на объект.

Инструменты: шахматная доска (распечатанный лист с чёрно-белыми клетками), утилита `camera_calibrator`.

Порядок действий (под руководством учителя):

1. **Запуск калибратора:**

   ```
   ros2 run camera_packege camera_calibrator_pro_node -f 3
   ```

2. **Съёмка:** держите шахматную доску перед камерой, медленно перемещайте в разные положения по указанию скрипта:

   • Близко / далеко.

   • Слева / справа / сверху / снизу.

   • Под наклоном.

   • В каждом положении держите неподвижно.

3. **Окончание калибровки:** когда скрипт наберет достаточное количество данных он произведет калибровку.

2.4. Что даёт калибровка

После калибровки:

• Прямые линии на изображении становятся прямыми.

• Можно точно определить угол на объект по его пиксельным координатам.

• ArUco-маркеры распознаются точнее.

Проверка: запустите `rqt_image_view` с топиком `/camera/image_rect` (если драйвер публикует выпрямленное изображение после калибровки). Сравните с `/camera/image_raw`.

3. Уровень «Углублённый»: Дисторсия и матрица камеры

(Что под капотом)

Почему линза искажает и как калибровка это исправляет.

3.1. Бочка и подушка: два типа дисторсии

Линза камеры — не идеальная. Лучи света, проходящие через край линзы, преломляются иначе, чем через центр.

Два основных типа искажений:

Тип	Как выглядит	Когда возникает
Бочка (barrel)	Прямые линии выгнуты наружу	Широкоугольная линза, близкое расстояние
Подушка (pincushion)	Прямые линии вогнуты внутрь	Телеобъектив, большое расстояние

У нашей камеры, скорее всего, **бочкообразная дисторсия** — края изображения «закруглены».

3.2. Матрица камеры и коэффициенты дисторсии

Калибровка вычисляет два набора параметров:

1. Матрица камеры (Camera Matrix):

```
    [ fx   0   cx ]
K = [  0  fy   cy ]
    [  0   0    1 ]
```

- `fx`, `fy` — фокусное расстояние в пикселях (по горизонтали и вертикали).
- `cx`, `cy` — координаты оптического центра (обычно близки к центру изображения).

2. Коэффициенты дисторсии (Distortion Coefficients):

```
D = [k1, k2, p1, p2, k3]
```

- `k1, k2, k3` — радиальная дисторсия (бочка/подушка).
- `p1, p2` — тангенциальная дисторсия (линза не параллельна матрице).

Формула исправления (упрощённо):

```
x_испр = x × (1 + k1×r² + k2×r⁴ + k3×r⁶)
y_испр = y × (1 + k1×r² + k2×r⁴ + k3×r⁶)
```

где `r` — расстояние от центра изображения. Чем дальше пиксель от центра, тем сильнее коррекция.

3.3. Почему важно перекалибровать камеру

Калибровка — это не «сделал и забыл». Параметры меняются:

- **После удара** — линза сместилась относительно матрицы.
- **При смене фокуса** — если камера с ручной фокусировкой.
- **При изменении температуры** — тепловое расширение материалов.

Правило: после сборки робота, после транспортировки, раз в месяц — перекалибровывайте камеру.

4. Уровень «Исследовательский»: Качество калибровки

(Прорыв за границы)

Эксперименты с параметрами калибровки.

4.1. Сколько кадров нужно для хорошей калибровки

Эксперимент:

1. Сделайте калибровку с 5 кадрами. Запишите значения `fx`, `fy`, `k1`.
2. Повторите с 10 кадрами.
3. Повторите с 20 кадрами.
4. Сравните параметры. Насколько они различаются?

Ожидаемый результат: после 15–20 кадров параметры стабилизируются. 5 кадров — мало, ошибка до 10%. 20 — оптимально.

4.2. Мини-исследование: влияние освещения

Вопрос: Влияет ли яркость освещения на точность калибровки?

Проверка:

1. Проведите калибровку при ярком свете.
2. Проведите калибровку в полутьме (но чтобы доска была видна).
3. Сравните параметры.

Гипотеза: при плохом освещении шум матрицы выше → углы клеток определяются с меньшей точностью → калибровка хуже.

Вывод: калибруйте камеру при том освещении, в котором робот будет работать.

5. Выводы по занятию

1. **Камера дополняет лидар.** Лидар даёт геометрию, камера — цвет и текстуру.
2. **Камера требует калибровки.** Линза искажает изображение, калибровка исправляет.
3. **Процесс калибровки:** шахматная доска → 15–20 кадров → `cameracalibrator` → файл параметров.
4. **Параметры не вечны.** Удары, смена фокуса, температура — причины для перекалибровки.

Связь со следующим занятием: Мы откалибровали камеру. Теперь научимся **получать кадры из кода** через геттеры `RobotDriver`: `get_frame()` и `save_frame()`. И сделаем первую миссию с фотофиксацией по лидару.

Домашнее задание:

- **Базовое**: Посмотрите видеопоток через `rqt_image_view`. Найдите искажения по краям.
- **Продвинутое**: Проведите калибровку под руководством учителя. Запишите полученные параметры `fx`, `fy`.
- **Исследовательское**: Сравните калибровку с 5, 10 и 20 кадрами. Насколько отличаются параметры?

Занятие 41: Геттеры камеры в RobotDriver: фотофиксация по лидару

Содержание занятия: Получение кадров из кода и автоматическое сохранение при обнаружении препятствия

Цели занятия

- **Образовательная**: Научиться использовать геттеры камеры `get_frame()` и `save_frame()` в `RobotDriver`.
- **Развивающая**: Собрать миссию, которая автоматически фотографирует препятствие в момент обнаружения лидаром.
- **Воспитательная**: Показать, как робот может самостоятельно собирать датасет для обучения нейросетей.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия PhotoOnObstacle (45 мин)**
   - Геттеры `get_frame()`, `save_frame()`.
   - `EvalAction` для сохранения кадра по условию.
   - Сборка миссии: едем, видим стену, останавливаемся, фотографируем.
3. **Уровень «Углублённый»: Архитектура и синхронизация (25 мин)**
   - Почему фото после остановки, а не во время движения.
   - Разница частот: лидар 10 Гц, камера 30 Гц.
4. **Уровень «Исследовательский»: Разметка датасета (10 мин)**
   - Сохранение пары «расстояние + кадр».
   - Точность синхронизации.
5. **Подведение итогов (5 мин)**

1. Введение: Робот-фотограф

На прошлом занятии мы откалибровали камеру. Теперь научимся **получать кадры из кода**.

`RobotDriver` уже содержит готовые методы:

- `get_frame()` — возвращает последний кадр с камеры (numpy-массив).
- `save_frame(filename)` — сохраняет последний кадр в файл.

Сегодня мы соберём миссию, которая едет вперёд, обнаруживает препятствие лидаром, **останавливается** и **фотографирует** его. Это первый шаг к созданию **датасета** — набора изображений для обучения нейросетей.

2. Уровень «База»: Миссия PhotoOnObstacle

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем миссию с фотофиксацией.

2.1. Геттеры камеры в RobotDriver

Метод	Возвращает	Описание
`get_frame()`	`numpy.ndarray` или `None`	Последний кадр с камеры (BGR)
`save_frame(filename=None)`	`str` (имя файла)	Сохраняет кадр в файл. Если имя не задано — генерирует с датой/временем

Важно: `get_frame()` возвращает `None`, если камера ещё не прислала ни одного кадра. В нашей миссии камера успеет запуститься за время движения.

Вызов в лямбде:

```
EvalAction(
    lambda d: (d.save_frame(), "Кадр сохранён")[1]
              if d.get_frame() is not None
              else "Камера не готова",
    name="Сохраняю кадр"
)
```

Лямбда сохраняет кадр и возвращает строку для лога.

2.2. Архитектура: фото после остановки

```
Sequence
├── StreamAction (едет вперёд, пока не увидит стену)  ← SUCCESS когда доехал
├── LogAction ("Стена! Фотографирую.")
├── EvalAction (сохраняет кадр в файл)                ← вызов save_frame()
└── StopAction (экстренный стоп)
```

Почему не Parallel? На прошлом занятии мы использовали `Parallel` для одновременного движения и мониторинга. Здесь логика проще:

1. `StreamAction` сам следит за лидаром и останавливается по `stop_condition`.

2. **После** остановки делаем фото — кадр будет чётким, без смазывания от движения.

3. Затем подтверждаем остановку через `StopAction`.

`Sequence` гарантирует, что фото сделается **после** того, как робот доехал до стены, а не во время движения.

2.3. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/photo_on_obstacle.py
```

```
#!/usr/bin/env python3
"""Миссия: ехать вперёд, сфотографировать препятствие по лидару."""
import sys
import rclpy
from driver_lib import (RobotDriver, Sequence,
                        StreamAction, EvalAction, StopAction,
                        LogAction)

class PhotoOnObstacle(RobotDriver):
    def build_tree(self):
        return Sequence([
            # 1. Едем вперёд, пока лидар не увидит стену
            StreamAction(
                command_func=lambda d: (0.2, 0.0, 0.0, 1.0),
                stop_condition=lambda d: d.get_lidar_front() < 0.3,
                name="Еду вперёд"
            ),
            # 2. Фотографируем препятствие
            LogAction("   Стена! Фотографирую."),
            EvalAction(
                lambda d: (d.save_frame(), "Кадр сохранён")[1]
                          if d.get_frame() is not None
                          else "Камера не готова",
                name="Сохраняю кадр"
            ),
            # 3. Явный стоп
            StopAction(name="Стоп"),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(PhotoOnObstacle('photo_on_obstacle', tick_rate=10))

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

Запуск:

```
python3 ~/ws/rbt/scripts/photo_on_obstacle.py
```

После выполнения: в домашней папке появится файл `frame_20250622_143015.jpg` (с текущей датой). Это снимок препятствия, перед которым остановился робот.

2.4. Создание датасета

Повторите миссию 10 раз, поднося к роботу **разные** объекты:

- Коробку.
- Стул.
- Человека (одноклассника).
- Стену полигона.

Каждый раз сохраняется новый кадр (имя с датой/временем уникально). Соберите их в папку — это ваш первый **датасет** для обучения нейросети распознаванию препятствий.

3. Уровень «Углублённый»: Архитектура и синхронизация

(Что под капотом)

Почему фото после остановки и как согласованы сенсоры.

3.1. Почему Sequence, а не Parallel

В занятии 39 мы использовали `Parallel` для одновременного движения и мониторинга. Но там была **страховка**: если `StreamAction` не остановится, `WaitCondition` в параллельной ветке заметит стену и вызовет `StopAction`.

Здесь задача другая: нам нужно **сделать фото после остановки**. Если бы фото делалось в параллельной ветке, оно могло бы сработать **до** остановки — кадр был бы смазанным.

Правило:

- **Parallel** — когда действия должны происходить **одновременно** (движение + мониторинг).
- **Sequence** — когда действия идут **строго по порядку** (доехал → сфотографировал → стоп).

3.2. Разница частот лидара и камеры

- **Лидар** публикует `/scan` с частотой **10 Гц** (каждые 100 мс).
- **Камера** публикует `/camera/image_raw` с частотой **30 Гц** (каждые 33 мс).

Когда `StreamAction` обнаруживает стену (лидар показал `< 0.3 м`), он вызывает `stop()` и возвращает `SUCCESS`. Затем `Sequence` переходит к фото.

Кадр в `_last_frame` обновляется асинхронно с частотой 30 Гц. На момент вызова `save_frame()` кадр имеет давность **не более 33 мс** — за это время робот уже остановился, смазывания нет.

Вывод: архитектура «фото после остановки» не требует идеальной синхронизации сенсоров.

4. Уровень «Исследовательский»: Разметка датасета

(Прорыв за границы)

Автоматическое связывание расстояния и изображения.

4.1. Сохранение пары «расстояние + кадр»

Идея: сохранять не только кадр, но и расстояние до препятствия в имени файла.

```
EvalAction(
    lambda d: (
        d.save_frame(f"obstacle_{d.get_lidar_front():.2f}m.jpg"),
        f"Сохранено с дистанцией {d.get_lidar_front():.2f} м"
    )[1],
    name="Сохраняю с разметкой"
)
```

Теперь имя файла содержит расстояние: `obstacle_0.28m.jpg`. Это **размеченный датасет**: по имени файла можно узнать, на каком расстоянии был объект. Для обучения нейросети это готовая разметка.

4.2. Мини-исследование: точность разметки

Вопрос: Насколько точно имя файла отражает реальное расстояние?

Методика:

1. Поставьте объект на расстоянии 0.5 м от старта робота.
2. Запустите миссию 5 раз.
3. Сравните расстояние в имени файла с измеренным рулеткой.

Ожидание: погрешность 2–5 см (точность лидара + время реакции).

5. Выводы по занятию

1. **`get_frame()` и `save_frame()` — готовые геттеры камеры.** Не нужно работать с топиками напрямую.
2. **`Sequence` для строгого порядка: доехал → сфотографировал → стоп.** `Parallel` не нужен.
3. **Фото после остановки — чёткий кадр.** Нет смазывания от движения.
4. **Разметка расстоянием — шаг к машинному обучению.** Имя файла содержит дистанцию.

Связь со следующим занятием: Мы умеем видеть препятствия лидаром и фотографировать их. Теперь научимся **двигаться вдоль стены**, используя лидар для удержания дистанции.

Домашнее задание:

- **Базовое**: Запустите `photo_on_obstacle.py` с разными объектами. Проверьте сохранённые кадры.

- **Продвинутое**: Модифицируйте `EvalAction`, чтобы в имя файла добавлялось расстояние.
- **Исследовательское**: Сделайте 5 кадров одного объекта, сравните расстояние в имени файла с рулеткой.

Занятие 42: Движение вдоль стены по лидару

Содержание занятия: Непрерывное управление с обратной связью. Эволюция от П к ПИД с антивиндапом

Цели занятия

- **Образовательная**: Собрать три версии регулятора для движения вдоль стены: П, ПИД без защиты, ПИД с антивиндапом. Сравнить их поведение.
- **Развивающая**: Понять проблемы интегрального насыщения и способы его устранения.
- **Воспитательная**: Показать эволюцию инженерного решения: простая идея → проблема → улучшение.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: П-регулятор (30 мин)**
    - Алгоритм, код, запуск, анализ.
3. **Уровень «Углублённый»: ПИД без защиты (25 мин)**
    - Добавление интеграла и производной.
    - Проблема интегрального насыщения на логах.
4. **Уровень «Исследовательский»: ПИД с антивиндапом (20 мин)**
    - Ограничение интеграла.
    - Сравнение трёх версий.
5. **Подведение итогов (10 мин)**

1. Введение: Три шага к идеальному движению

Мы хотим, чтобы робот ехал вдоль стены на расстоянии 40 см. Сегодня мы пройдём эволюцию регулятора:

1. **П-регулятор** — простейшая обратная связь: ошибка × коэффициент.
2. **ПИД-регулятор** — добавляем память (интеграл) и предвидение (производная).
3. **ПИД с антивиндапом** — чиним проблему интегрального насыщения.

Каждая версия — отдельная программа. Запустите все три и сравните логи.

2. Уровень «База»: П-регулятор

(Инструмент, навык, интерфейс)

Простейшая обратная связь без памяти.

2.1. Алгоритм

```
vy = (target - current) × Kp
```

Плюсы: простота, быстрая реакция. **Минусы:** может рыскать из-за инерции, не устраняет постоянную ошибку.

2.2. Код: follow_wall_p.py

```
nano ~/ws/rbt/scripts/follow_wall_p.py
```

```
#!/usr/bin/env python3
"""Миссия: движение вдоль стены. П-регулятор."""
import sys
import rclpy
from driver_lib import RobotDriver, StreamAction

class FollowWallP(RobotDriver):
    def build_tree(self):
        target = 0.4   # целевое расстояние, м
        Kp = 0.5       # коэффициент усиления

        def command(d):
            current = d.get_lidar_right()
            if current == float('inf'):
                return (0.2, 0.0, 0.0, 1.0)
            error = target - current
            vy = error * Kp
            if vy > 0.3: vy = 0.3
            elif vy < -0.3: vy = -0.3
            d.get_logger().info(
                f'Справа: {current:.2f} м, ош: {error:+.2f}, vy: {vy:
+.2f}',
                throttle_duration_sec=1.0
            )
            return (0.2, vy, 0.0, 1.0)

        return StreamAction(
            command_func=command,
            stop_condition=lambda d: d.get_lidar_front() < 0.3,
            name="П-регулятор"
        )

def main():
    rclpy.init()
    RobotDriver.spin(FollowWallP('follow_wall_p', tick_rate=10))
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/follow_wall_p.py
```

2.3. Пример лога

```
Справа: 0.35 м, ош: +0.05, vy: -0.02
Справа: 0.23 м, ош: +0.17, vy: -0.08
Справа: 0.39 м, ош: +0.01, vy: -0.01
Справа: 0.43 м, ош: -0.03, vy: +0.01
```

Видно, что `vy` зависит только от текущей ошибки. При ошибке 0.05 `vy = -0.02`, при 0.17 `vy = -0.08`. Пропорционально.

3. Уровень «Углублённый»: ПИД без защиты

(Что под капотом)

Добавляем память и предвидение.

3.1. Алгоритм

```
vy = (Kp × error + Ki × integral + Kd × derivative)
```

Интеграл (`integral += error`) — сумма всех ошибок. Устраняет постоянное смещение. **Производная** (`error - prev_error`) — скорость изменения ошибки. Гасит рыскание.

Проблема: интеграл неограниченно растёт → интегральное насыщение (windup).

3.2. Код: follow_wall_pid.py

```
nano ~/ws/rbt/scripts/follow_wall_pid.py
```

```
#!/usr/bin/env python3
"""Миссия: движение вдоль стены. ПИД-регулятор без защиты."""
import sys
import rclpy
from driver_lib import RobotDriver, StreamAction

def make_pid_controller(target, Kp, Ki, Kd):
    """Создаёт ПИД-регулятор с памятью (интеграл и предыдущая ошибка)."""
    integral = 0.0
    prev_error = 0.0

    def control(d):
        nonlocal integral, prev_error
        current = d.get_lidar_right()
        if current == float('inf'):
            return (0.2, 0.0, 0.0, 1.0)
```

```
        error = target - current
        integral += error
        derivative = error - prev_error
        prev_error = error

        vy = (Kp * error + Ki * integral + Kd * derivative)

        if vy > 0.3: vy = 0.3
        elif vy < -0.3: vy = -0.3

        d.get_logger().info(
            f'Справа: {current:.2f} м, ош: {error:+.2f}, '
            f'инт: {integral:+.2f}, диф: {derivative:+.2f}, vy: {vy:
+.2f}',
            throttle_duration_sec=1.0
        )
        return (0.2, vy, 0.0, 1.0)

    return control

class FollowWallPID(RobotDriver):
    def build_tree(self):
        command = make_pid_controller(target=0.4, Kp=0.5, Ki=0.01, Kd=0.1)

        return StreamAction(
            command_func=command,
            stop_condition=lambda d: d.get_lidar_front() < 0.3,
            name="ПИД (без защиты)"
        )

def main():
    rclpy.init()
    RobotDriver.spin(FollowWallPID('follow_wall_pid', tick_rate=10))

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

Запуск:

```
python3 ~/ws/rbt/scripts/follow_wall_pid.py
```

3.3. Пример лога и проблема

```
Справа: 0.47 м, ош: -0.07, инт: -0.07, диф: -0.07, vy: +0.04
Справа: 0.38 м, ош: +0.02, инт: -0.69, диф: +0.02, vy: -0.00
Справа: 0.25 м, ош: +0.15, инт: +0.33, диф: +0.01, vy: -0.08
Справа: 0.29 м, ош: +0.11, инт: +1.76, диф: -0.01, vy: -0.07
Справа: 0.36 м, ош: +0.04, инт: +2.57, диф: -0.00, vy: -0.05
Справа: 0.39 м, ош: +0.01, инт: +2.79, диф: -0.00, vy: -0.03
```

Проблема: интеграл (`инт`) неограниченно растёт. За 6 секунд он достиг +2.79. Если робот отъедет от стены, интеграл будет **долго** возвращаться к нулю, вызывая неадекватную реакцию. Это **интегральное насыщение (integral windup)**.

4. Уровень «Исследовательский»: ПИД с антивиндапом

(Прорыв за границы)

Ограничиваем интеграл для предотвращения насыщения.

4.1. Алгоритм антивиндапа

Добавляем **ограничение интеграла** (`integral_limit`):

```
    integral += error
    if integral > limit: integral = limit
    elif integral < -limit: integral = -limit
```

Это не даёт интегралу «разогнаться» до больших значений. Когда ошибка меняет знак, интеграл быстро возвращается в норму.

Выбор предела: `integral_limit` должен быть таким, чтобы `Ki × limit` давало заметный, но не огромный вклад в `vy` . Для `Ki = 0.01` и предела 1.0 вклад = 0.01 м/с. Лучше взять предел 2.0–5.0 для более заметного эффекта.

4.2. Код: follow_wall_antiwindup.py

```
nano ~/ws/rbt/scripts/follow_wall_antiwindup.py
```

```
#!/usr/bin/env python3
"""Миссия: движение вдоль стены. ПИД с антивиндапом."""
import sys
import rclpy
from driver_lib import RobotDriver, StreamAction

def make_pid_antiwindup(target, Kp, Ki, Kd, integral_limit=3.0):
    """Создаёт ПИД-регулятор с защитой от интегрального насыщения."""
    integral = 0.0
    prev_error = 0.0

    def control(d):
        nonlocal integral, prev_error
        current = d.get_lidar_right()
        if current == float('inf'):
            return (0.2, 0.0, 0.0, 1.0)

        error = target - current
        integral += error

        # Антивиндап: ограничение интеграла
        if integral > integral_limit:
            integral = integral_limit
        elif integral < -integral_limit:
            integral = -integral_limit

        derivative = error - prev_error
        prev_error = error
```

```
        vy = (Kp * error + Ki * integral + Kd * derivative)

        if vy > 0.3: vy = 0.3
        elif vy < -0.3: vy = -0.3

        d.get_logger().info(
            f'Справа: {current:.2f} м, ош: {error:+.2f}, '
            f'инт: {integral:+.2f}, диф: {derivative:+.2f}, vy: {vy:
+.2f}',
            throttle_duration_sec=1.0
        )
        return (0.2, vy, 0.0, 1.0)

    return control

class FollowWallAntiwindup(RobotDriver):
    def build_tree(self):
        command = make_pid_antiwindup(target=0.4, Kp=0.5, Ki=0.01,
Kd=0.1, integral_limit=3.0)

        return StreamAction(
            command_func=command,
            stop_condition=lambda d: d.get_lidar_front() < 0.3,
            name="ПИД (с защитой)"
        )

def main():
    rclpy.init()
    RobotDriver.spin(FollowWallAntiwindup('follow_wall_aw', tick_rate=10))

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

Запуск:

```
python3 ~/ws/rbt/scripts/follow_wall_antiwindup.py
```

4.3. Сравнение трёх версий

Запустите все три скрипта по очереди на одном и том же участке стены. Сравните логи:

Характеристика	П	ПИД без защиты	ПИД с защитой
Сколько параметров	1 (Kp)	3 (Kp, Ki, Kd)	4 (+ предел)
Интеграл	Нет	Растёт неограниченно	Ограничен
Устойчивость	Может рыскать	Страдает от windup	Стабильнее
Когда использовать	Быстрое прототипирование	Не использовать	Реальная эксплуатация

5. Выводы по занятию

1. **П-регулятор** — прост, но без памяти. Реагирует только на текущую ошибку.

2. **ПИД-регулятор** добавляет память (интеграл) и предвидение (производная). Но интеграл нужно ограничивать.
3. **Антивиндап** — обязательная защита для реальных систем. Без неё ПИД может стать хуже П.
4. **Эволюция регулятора** — пример инженерного подхода: простая идея → проблема → улучшение.

Связь со следующим занятием: Мы научились следовать вдоль стены. Теперь используем это для **выхода из лабиринта по правилу правой руки**.

Домашнее задание:

- **Базовое**: Запустите П-регулятор. Запишите 10 строк лога.
- **Продвинутое**: Запустите ПИД без защиты. Найдите в логах момент, где интеграл превысил 2.0.
- **Исследовательское**: Подберите `integral_limit` так, чтобы интеграл не превышал 1.0. Как изменилось поведение?

Занятие 43: Выход из лабиринта по правилу правой руки

Содержание занятия: Реализация классического алгоритма на деревьях поведения

Цели занятия

- **Образовательная**: Реализовать правило правой руки для выхода из лабиринта, используя `RepeatUntil`, `Fallback`, `Sequence` и `Condition`.
- **Развивающая**: Понять, почему дерево поведения удобнее конечного автомата для перебора вариантов.
- **Воспитательная**: Показать, что классические алгоритмы можно элегантно выразить через деревья поведения.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия MazeMission (45 мин)**
   - Правило правой руки: логика.
   - Дерево: `RepeatUntil` с `Fallback` из трёх `Sequence`.
   - Запуск в простом лабиринте.
3. **Уровень «Углублённый»: Почему дерево, а не автомат (25 мин)**
   - `Fallback` сам перебирает варианты.
   - Обработка тупика: разворот на 180°.
   - Ограничения правила правой руки.
4. **Уровень «Исследовательский»: Непрерывное управление как альтернатива (10 мин)**
   - Три луча и приоритеты вместо дерева.

- Сравнение двух подходов.

5. **Подведение итогов (5 мин)**

1. Введение: Лабиринт как тест на интеллект

Мы умеем следовать вдоль стены и останавливаться перед препятствием. Теперь соберём это в **алгоритм выхода из лабиринта**.

Правило правой руки: всегда держись правой стены. Если есть проход направо — поверни. Если нельзя направо, но можно вперёд — иди вперёд. Если нельзя ни направо, ни вперёд — поверни налево.

Это **не** самый быстрый алгоритм, но он **гарантированно** выводит из любого односвязного лабиринта. И реализуется очень элегантно на деревьях поведения.

2. Уровень «База»: Миссия MazeMission

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем дерево и тестируем в лабиринте.

2.1. Правило правой руки в виде блок-схемы

На каждом шаге робот проверяет три условия **в строгом порядке**:

```
1. Справа свободно (> 0.5 м)?
   └─ Да → повернуть направо (dyaw_deg = -90°), выйти
   └─ Нет → проверяем дальше

2. Впереди свободно (> 0.5 м)?
   └─ Да → проехать вперёд 0.5 м, выйти
   └─ Нет → проверяем дальше

3. (Оба ложны) → тупик → повернуть налево (dyaw_deg = +90°), выйти
```

Знаки углов: в ROS2 положительный угол — поворот влево (против часовой стрелки). Поэтому:

- Поворот направо: `dyaw_deg = -90°`.
- Поворот налево: `dyaw_deg = +90°`.

2.2. Дерево: Fallback перебирает варианты

```
RepeatUntil (stop_condition = тупик со всех сторон)
└── Fallback
    ├── Sequence 1: Condition(справа > 0.5) → MoveRelativeAction(поворот
-90°)
    ├── Sequence 2: Condition(впереди > 0.5) → MoveRelativeAction(вперёд
0.5 м)
    └── Sequence 3: MoveRelativeAction(поворот +90°)
```

Как это работает:

1. `Fallback` пробует `Sequence 1`: если справа свободно → поворот направо. `SUCCESS`.
2. Если `Condition` в `Sequence 1` вернул `FAILURE` → `Fallback` пробует `Sequence 2`.
3. Если `Condition` в `Sequence 2` тоже `FAILURE` → `Sequence 3` (поворот налево) выполняется всегда.
4. `RepeatUntil` повторяет весь `Fallback`, пока `stop_condition` не станет истиной.

Красота дерева: мы не пишем `if-elif-else`. `Fallback` сам перебирает варианты, пока один не сработает.

2.3. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/maze_right_hand.py
```

```
#!/usr/bin/env python3
"""Миссия: выход из лабиринта по правилу правой руки."""
import sys
import rclpy
from driver_lib import (RobotDriver, RepeatUntil, Fallback, Sequence,
                        Condition, MoveRelativeAction, LogAction)

class MazeRightHand(RobotDriver):
    def build_tree(self):
        return RepeatUntil(
            child=Fallback([
                # Приоритет 1: справа свободно → поворот направо (-90°)
                Sequence([
                    Condition(lambda d: d.get_lidar_right() > 0.5),
                    LogAction("↱ Поворот направо"),
                    MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=-90.0, nam
e="Направо"),
                ]),
                # Приоритет 2: впереди свободно → вперёд 0.5 м
                Sequence([
                    Condition(lambda d: d.get_lidar_front() > 0.5),
                    LogAction("↑ Вперёд"),
                    MoveRelativeAction(dx=0.5, dy=0.0, dyaw_deg=0.0,
name="Вперёд"),
                ]),
                # Приоритет 3: тупик → разворот налево (+90°)
                Sequence([
                    LogAction("↰ Тупик! Поворот налево"),
                    MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=90.0,
name="Налево"),
                ]),
            ]),
            stop_condition=lambda d: (
                d.get_lidar_front() < 0.2 and
                d.get_lidar_right() < 0.2 and
                d.get_lidar_left() < 0.2
```

```
            ),
            name="Правило правой руки"
        )

def main():
    rclpy.init()
    RobotDriver.spin(MazeRightHand('maze_right_hand', tick_rate=5))

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

Запуск:

```
python3 ~/ws/rbt/scripts/maze_right_hand.py
```

`stop_condition`: миссия завершится, когда робот окажется в тупике со всех сторон (спереди, справа, слева < 0.2 м).

Тест: постройте простой лабиринт из коробок. Запустите робота. Он должен, держась правой стены, найти выход.

2.4. Почему tick_rate=5

Для лабиринта не нужна высокая частота. Робот делает дискретные шаги (поворот или 0.5 м вперёд). Между шагами `MoveRelativeAction` ждёт `SUCCESS` от автопилота. 5 Гц достаточно для проверки условий между манёврами.

3. Уровень «Углублённый»: Почему дерево, а не автомат

(Что под капотом)

Сравнение с классическим подходом.

3.1. Дерево vs конечный автомат

Конечный автомат потребовал бы:

- Переменную `state` (IDLE, TURN_RIGHT, GO_FORWARD, TURN_LEFT).
- `if-elif-else` для переходов между состояниями.
- Ручной сброс состояния после каждого действия.

Дерево поведения делает то же самое декларативно:

- `Fallback` — это таблица переходов.
- `Condition` — условия перехода.
- `MoveRelativeAction` — действия в состоянии.
- `RepeatUntil` — цикл.

Преимущество дерева: добавление нового варианта (например, «разворот на 180° при застревании») — это просто ещё один `Sequence` в `Fallback`.

3.2. Обработка тупика и ограничения правила

Что если все три условия ложны, а разворот налево тоже упирается в стену?

Это **полный тупик**. Нужно развернуться на 180°:

```
# Добавляем 4-й вариант перед разворотом налево:
Sequence([
    LogAction("↶ Полный тупик! Разворот на 180°"),
    MoveRelativeAction(dx=0.0, dy=0.0, dyaw_deg=180.0, name="Разворот"),
]),
```

Ограничения правила правой руки:

- Работает только в **односвязных** лабиринтах (без замкнутых петель).
- Не находит кратчайший путь.
- Застревает, если выход не на внешней стене.
- Дискретные шаги (90°) не работают для не-прямоугольных коридоров.

4. Уровень «Исследовательский»: Непрерывное управление как альтернатива

(Прорыв за границы)

Когда дискретные шаги не справляются — на помощь приходит непрерывное управление.

4.1. Три луча и приоритеты вместо дерева

Если коридоры не прямоугольны (углы не 90°), дискретные шаги не работают. Решение — **непрерывное управление** с обратной связью по трём лучам лидара:

```python
#!/usr/bin/env python3
"""Миссия: выход из лабиринта — непрерывное следование вдоль правой
стены."""
import sys
import rclpy
import math
from driver_lib import RobotDriver, StreamAction

class MazeFollowRight(RobotDriver):
    def build_tree(self):
        target = 0.5   # целевое расстояние до правой стены, м
        Kp = 0.5       # коэффициент усиления по расстоянию
        Kyaw = 0.8     # коэффициент усиления по углу

        def command(d):
            r_front = d._lidar_distance_at(-math.radians(45))
            r_side  = d._lidar_distance_at(-math.radians(90))
            r_back  = d._lidar_distance_at(-math.radians(135))
            front = d.get_lidar_front()

            if r_front == float('inf') or r_side == float('inf') or
r_back == float('inf') or front == float('inf'):
                return (0.2, 0.0, 0.0, 1.0)

            avg_right = (r_front + r_side + r_back) / 3.0
            error_dist = target - avg_right
            vy = error_dist * Kp
            if vy > 0.3: vy = 0.3
            elif vy < -0.3: vy = -0.3

            error_angle = r_back - r_front
            omega = error_angle * Kyaw
            if omega > 0.5: omega = 0.5
            elif omega < -0.5: omega = -0.5

            if front < 0.5:
                d.get_logger().info('   Стена впереди! Разворот налево')
                return (0.0, 0.0, 0.5, 1.0)

            if r_side > 1.0:
                d.get_logger().info('↱ Проход справа! Поворот направо')
                return (0.0, 0.0, -0.5, 1.0)

            d.get_logger().info(
                f'Справа: {r_front:.2f} | {r_side:.2f} | {r_back:.2f} м, '
                f'vy: {vy:+.2f}, ω: {omega:+.2f}',
                throttle_duration_sec=1.0
            )
            return (0.2, vy, omega, 1.0)

        return StreamAction(
            command_func=command,
            stop_condition=lambda d: (
                d.get_lidar_front() < 0.2 and
                d.get_lidar_right() < 0.2 and
```

```
                d.get_lidar_left() < 0.2
            ),
            name="Правило правой руки"
        )

def main():
    rclpy.init()
    RobotDriver.spin(MazeFollowRight('maze_follow_right', tick_rate=10))

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

4.2. Сравнение двух подходов

Характеристика	Дискретное дерево	Непрерывное управление
Сложность кода	4 узла, 3 уровня	1 функция, 40 строк
Повороты	Только 90° (не адаптируются)	Плавные, любой угол
Выравнивание	Отсутствует	По трём лучам
Подходит для	Прямоугольных лабиринтов	Лабиринтов любой геометрии
Отказоустойчивость	RetryAction, StopAction	Ручная обработка в if

Вывод: для прямоугольных лабиринтов дискретное дерево проще и нагляднее. Для реальных помещений с нестандартными углами непрерывное управление надёжнее.

5. Выводы по занятию

1. **Правило правой руки** — классический алгоритм для лабиринтов. Реализуется тремя `Sequence` в `Fallback`.
2. **`Fallback` перебирает варианты** — декларативная замена `if-elif-else`.
3. **`RepeatUntil` зацикливает** поведение до достижения цели.
4. **Дискретные шаги (90°) работают только в прямоугольных лабиринтах.** Для реальных помещений лучше непрерывное управление.

Связь со следующим занятием: Мы освоили лидар для навигации. На следующем занятии — **зачёт по сенсорам**: комплексная миссия с объездом препятствия, фотофиксацией и возвратом на маршрут.

Домашнее задание:

- **Базовое**: Запустите `maze_right_hand.py` в лабиринте из 3 коробок.
- **Продвинутое**: Добавьте обработку полного тупика с разворотом на 180°.
- **Исследовательское**: Запустите непрерывную версию `maze_follow_right.py` и сравните с дискретной в одном лабиринте.

Занятие 44: Зачёт по сенсорам

Содержание занятия: Комплексная миссия — движение к цели с обнаружением препятствий, объездом и фотофиксацией

Цели занятия

- **Образовательная**: Собрать миссию, объединяющую все навыки работы с сенсорами: непрерывное движение, обнаружение препятствия лидаром, поиск просвета, объезд, продолжение движения.
- **Развивающая**: Понять архитектуру «цикл с обработкой нештатной ситуации»: `RepeatUntil` с `StreamAction` для движения и `Sequence` для объезда.
- **Воспитательная**: Провести зачёт в формате «реальной миссии» — робот должен автономно дойти до цели, несмотря на препятствия.

План занятия (90 минут)

1. **Организационный момент и регламент зачёта (5 мин)**
2. **Уровень «База»: Разбор миссии DriveToGoal (50 мин)**
   - Архитектура: `RepeatUntil` + `StreamAction` + `Sequence`.
   - Поиск просвета смещением влево.
   - Движение после объезда с контролем датчиков.
   - Запуск и демонстрация.
3. **Уровень «Углублённый»: Анализ архитектуры (20 мин)**
   - Почему `StreamAction` для движения, а `MoveRelativeAction` для манёвров.
   - `RepeatUntil` для поиска просвета.
   - Относительный подсчёт дистанции.
4. **Уровень «Исследовательский»: Умный выбор стороны объезда (10 мин)**
   - Сравнение `get_lidar_left()` и `get_lidar_right()`.
   - Адаптивный выбор направления смещения.
5. **Подведение итогов блока «Сенсоры» (5 мин)**

1. Введение: Выпускной экзамен блока «Сенсоры»

Мы прошли большой путь:

- **Лидар:** получение данных, сектора, остановка перед препятствием, следование вдоль стены, лабиринт.
- **Камера:** калибровка, получение кадров, фотофиксация.
- **Деревья поведения:** `Sequence`, `Fallback`, `Parallel`, `RepeatUntil`, `StreamAction`.

Сегодня — **зачёт**. Вы запустите готовую миссию, которая объединяет всё:

1. Робот непрерывно едет вперёд.
2. Обнаруживает препятствие лидаром.
3. Ищет просвет, смещаясь влево.
4. Проезжает вперёд мимо препятствия.
5. Продолжает движение к цели (5 метров).

Это **цикл с обработкой нештатной ситуации** — стандартный паттерн в промышленной робототехнике.

2. Уровень «База»: Разбор миссии DriveToGoal

(Инструмент, навык, интерфейс)

Обязательный уровень. Разбираем архитектуру и запускаем на полигоне.

2.1. Архитектура миссии

```
Sequence
  └── RepeatUntil (stop = прошли 5 метров)
        └── Sequence
            ├── StreamAction (едем вперёд, стоп по препятствию)
            └── Sequence (объезд)
                ├── MoveRelativeAction (назад 0.2 м)
                ├── RepeatUntil (stop = просвет найден)
                │   └── MoveRelativeAction (смещение влево 0.2 м)
                └── StreamAction (движение вперёд после объезда)
```

Ключевые моменты:

- **Внешний `RepeatUntil`** — главный цикл: «ехать → если препятствие → объехать → продолжать».
- **`StreamAction` для движения** — непрерывное движение с проверкой лидара 10 раз в секунду.
- **`MoveRelativeAction` для манёвров** — дискретные шаги (отъезд, смещение).
- **Внутренний `RepeatUntil`** — поиск просвета: смещаемся влево, пока передний луч не покажет > 0.9 м.

2.2. Как работает цикл

1. **`StreamAction("DriveForward")`** — робот едет вперёд 0.15 м/с. На каждом тике проверяет `obstacle_ahead`: если фронт < 0.4 м — останавливается.
2. **Объезд:**
   - Отъезд назад 0.2 м (`MoveRelativeAction`).
   - **Поиск просвета:** смещение влево по 0.2 м, пока `front_is_clear` не покажет > 0.9 м.
   - **Движение вперёд** (`StreamAction("Proceed")`) с контролем датчиков: стоп, если впереди < 0.5 м или справа < 0.3 м.

3. **Проверка цели:** `goal_reached` считает пройденное расстояние от старта. Если ≥ 5 м — миссия завершена.

2.3. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/drive_5m_avoid.py
```

```
#!/usr/bin/env python3
"""Финальная версия: цикл 'еду вперёд → препятствие → объезд → снова
вперёд' до 5 метров."""
import sys
import rclpy
from driver_lib import (RobotDriver, RepeatUntil, Fallback, Sequence,
                        Condition, MoveRelativeAction, StreamAction,
                        EvalAction, LogAction, Parallel, HeartbeatAction)

class DriveToGoalWithObstacleAvoidance(RobotDriver):
    def __init__(self):
        super().__init__('drive_5m_avoid', tick_rate=10)

    def build_tree(self):
        start_x = [None]
        FRONT_WIDTH = 26

        def log_status(driver):
            x, y, yaw = driver.get_pose()
            front = driver.get_lidar_front(FRONT_WIDTH)
            right = driver.get_lidar_right(20)
            left = driver.get_lidar_left(20)
            if start_x[0] is None:
                start_x[0] = x
            progress = x - start_x[0]
            remaining = max(0, 5.0 - progress)
            return f"  ({x:.2f},{y:.2f}) | прошли {progress:.2f}м,
осталось {remaining:.2f}м |    п={front:.2f} пр={right:.2f} л={left:.2f}"

        def goal_reached(driver):
            if start_x[0] is None:
                return False
            x, y, yaw = driver.get_pose()
            if (x - start_x[0]) ≥ 5.0:
                driver.get_logger().info("   Цель достигнута: 5 метров!")
                return True
            return False

        def obstacle_ahead(driver):
            front = driver.get_lidar_front(FRONT_WIDTH)
            if front ≤ 0.4:
                driver.get_logger().info(f"    Препятствие на {front:.2f}м
!")
                return True
            return False
```

```
        def front_is_clear(driver):
            front = driver.get_lidar_front(FRONT_WIDTH)
            if front > 0.9:
                driver.get_logger().info(f"   Просвет! front={front:.2f}
м")
                return True
            return False

        def path_blocked(driver):
            front = driver.get_lidar_front(FRONT_WIDTH)
            right = driver.get_lidar_right(20)
            if front ⩽ 0.5:
                driver.get_logger().info(f"   Впереди препятствие:
{front:.2f}м")
                return True
            if right ⩽ 0.3:
                driver.get_logger().info(f"    Справа близко: {right:.2f}м
")
                return True
            return False

        def drive_forward(driver):
            return (0.15, 0.0, 0.0, 0.3)

        def proceed(driver):
            front = driver.get_lidar_front(FRONT_WIDTH)
            right = driver.get_lidar_right(20)
            driver.get_logger().info(
                f"    Еду | п={front:.2f}м пр={right:.2f}м",
                throttle_duration_sec=0.5
            )
            return (0.15, 0.0, 0.0, 0.3)

        return Sequence([
                LogAction("=" * 60),
                LogAction("   Миссия: проехать 5 метров с объездом
препятствий"),
                EvalAction(log_status, "Start"),
                LogAction("=" * 60),

                RepeatUntil(
                    child=Sequence([
                        # Попытка ехать вперёд
                        StreamAction(
                            command_func=drive_forward,
                            stop_condition=obstacle_ahead,
                            name="DriveForward"
                        ),

                        # Если остановились (препятствие) — объезд
                        LogAction("=" * 60),
                        LogAction("   НАЧИНАЮ ОБЪЕЗД ... "),
                        EvalAction(log_status, "BeforeAvoid"),

                        # Отъезд назад
                        LogAction("    Отъезд назад 0.2м"),
```

```
                        MoveRelativeAction(dx=-0.2, dy=0.0, dyaw_deg=0.0,
                                           name="BackOff", exec_timeout=5.0)
,

                        # Поиск просвета: смещаемся влево
                        LogAction("    Ищу просвет слева ... "),
                        RepeatUntil(
                            child=Sequence([
                                MoveRelativeAction(dx=0.0, dy=0.2, dyaw_de
g=0.0,
                                                   name="ShiftLeft", exec_ti
meout=5.0),
                                EvalAction(log_status, "Shift"),
                            ]),
                            stop_condition=front_is_clear,
                            name="FindGap"
                        ),

                        LogAction("   Просвет найден! Еду вперёд ... "),

                        # Движение вперёд с контролем датчиков
                        StreamAction(
                            command_func=proceed,
                            stop_condition=path_blocked,
                            name="Proceed"
                        ),

                        EvalAction(log_status, "AfterProceed"),
                        LogAction("=" * 60),
                    ]),
                    stop_condition=goal_reached,
                    name="MainLoop"
                ),

                LogAction("=" * 60),
                LogAction("   5 МЕТРОВ ПРОЙДЕНЫ! МИССИЯ ВЫПОЛНЕНА!"),
                EvalAction(log_status, "Final"),
                LogAction("=" * 60),
            ])

def main():
    rclpy.init()
    RobotDriver.spin(DriveToGoalWithObstacleAvoidance())

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

Запуск:

```
python3 ~/ws/rbt/scripts/drive_5m_avoid.py
```

Тест: поставьте коробку на пути робота. Запустите миссию. Робот должен доехать до коробки, остановиться, сместиться влево до просвета, проехать вперёд и продолжить движение.

3. Уровень «Углублённый»: Анализ архитектуры

(Что под капотом)

Почему выбраны именно такие узлы.

3.1. StreamAction для движения, MoveRelativeAction для манёвров

Действие	Узел	Почему
Движение вперёд	`StreamAction`	Непрерывная проверка лидара 10 Гц. Остановка по условию.
Отъезд назад	`MoveRelativeAction`	Точная дистанция, один раз.
Смещение влево	`MoveRelativeAction`	Дискретный шаг, контролируемое смещение.
Поиск просвета	`RepeatUntil` + `MoveRelativeAction`	Цикл смещений до выполнения условия.
Движение после объезда	`StreamAction`	Снова непрерывное, но с другими условиями остановки.

3.2. Относительный подсчёт дистанции

Вместо абсолютных координат (которые могут быть любыми) используется **относительное** расстояние:

```
start_x = [None]  # список для мутабельности в замыкании
if start_x[0] is None:
    start_x[0] = x     # запомнили стартовую позицию
progress = x - start_x[0]  # сколько прошли
```

Это работает независимо от начальных показаний одометрии.

4. Уровень «Исследовательский»: Умный выбор стороны объезда

(Прорыв за границы)

Адаптивный объезд.

4.1. Выбор стороны по данным лидара

В текущей версии робот всегда смещается **влево**. Но если слева стена, а справа свободно — нужно смещаться **вправо**.

Модификация: перед поиском просвета сравнить `get_lidar_left()` и `get_lidar_right()`:

```
if driver.get_lidar_left() > driver.get_lidar_right():
    dy = 0.2   # смещаться влево
else:
    dy = -0.2  # смещаться вправо
```

Это требует динамического построения узла `MoveRelativeAction`, что в текущей архитектуре делается через `lambda` отдельную функцию.

4.2. Мини-исследование: форма препятствия

Эксперимент: поставьте препятствия разной формы (коробка, стул, мяч). Для каждого запустите миссию 3 раза. Запишите:

- Сколько смещений влево потребовалось для поиска просвета.
- На сколько сантиметров робот отклонился от маршрута после объезда.

Гипотеза: широкие препятствия требуют большего бокового смещения.

5. Подведение итогов блока «Сенсоры»

Чему мы научились (занятия 36–44)

1. **Лидар:** чтение данных, геттеры, сектора, `Condition`-узлы, остановка перед препятствием, следование вдоль стены, лабиринт.
2. **Камера:** калибровка, получение кадров, фотофиксация.
3. **Интеграция:** миссии, объединяющие оба сенсора.
4. **Архитектурные паттерны:** цикл с обработкой нештатной ситуации, поиск просвета, относительная одометрия.

Что дальше

Следующий большой блок — **«Введение в OpenCV»**. Мы научимся обрабатывать изображения: выделять линии, трекать цветные объекты, распознавать ArUco-маркеры. Это следующий уровень компьютерного зрения.

Домашнее задание:

- **Базовое**: Запустите `drive_5m_avoid.py` с препятствием.
- **Продвинутое**: Модифицируйте код для выбора стороны смещения по данным лидара.
- **Исследовательское**: Проведите эксперимент с препятствиями разной формы. Запишите количество смещений.

Введение в OpenCV

Занятие 45: Проверка установки OpenCV, загрузка изображений, цветовые пространства

Содержание занятия: Знакомство с библиотекой компьютерного зрения. Первое изображение с камеры робота

Цели занятия

- **Образовательная**: Проверить установку OpenCV, научиться получать кадры с камеры робота, преобразовывать цветовые пространства (BGR, RGB, HSV), разделять каналы.
- **Развивающая**: Понять разницу между BGR и RGB, преимущества HSV для цветового поиска. Освоить многопоточную обработку с защитой от повторного запуска.
- **Воспитательная**: Показать, что изображение — это массив чисел, с которым можно работать программно.

План занятия (90 минут)

1. **Организационный момент и введение (5 мин)**
2. **Уровень «База»: Первый кадр и разбор каналов (45 мин)**
   - Проверка установки OpenCV.
   - Миссия `FirstFrame`: получение кадра через `Blackboard`.
   - Миссия `ColorChannels`: разбор на B, G, R и HSV.
   - Архитектура: `Blackboard` + многопоточность + защита от повторного запуска.
3. **Уровень «Углублённый»: BGR vs RGB и HSV (25 мин)**
   - Исторические причины BGR.
   - Hue, Saturation, Value.
   - Почему HSV удобнее для цветового поиска.
4. **Уровень «Исследовательский»: Гистограммы и освещение (10 мин)**
   - Построение гистограммы.
   - Влияние освещения на HSV.
5. **Подведение итогов (5 мин)**

1. Введение: Компьютерное зрение — это матрицы

Мы использовали камеру для сохранения кадров. Теперь мы будем **обрабатывать** эти кадры: искать линии, объекты, маркеры.

OpenCV (Open Source Computer Vision Library) — самая популярная библиотека компьютерного зрения. Для неё изображение — это **numpy-массив** размером (высота × ширина × 3), где каждый пиксель — три числа: синий, зелёный, красный (BGR).

В новой версии `driver_lib` используется **потокобезопасный Blackboard** и **многопоточный исполнитель**. Тяжёлая обработка кадров выносится в отдельный узел (`Node`), который работает параллельно с деревом поведения. Для безопасности добавлен флаг `_busy`, защищающий от повторного входа в обработчик.

2. Уровень «База»: Первый кадр и разбор каналов

(Инструмент, навык, интерфейс)

Обязательный уровень. Получаем кадр с камеры и разбираем на каналы.

2.1. Проверка установки OpenCV

OpenCV уже установлен на роботе (требуется для `driver_lib`). Проверим версию:

```
python3 -c "import cv2; print(cv2.__version__)"
```

Если выводится версия (например, `4.10.0`) — всё готово.

2.2. Миссия FirstFrame: сохранение кадра

Создайте файл:

```
nano ~/ws/rbt/scripts/first_frame.py
```

```
#!/usr/bin/env python3
"""Миссия: получить кадр с камеры, обработать в отдельном потоке,
сохранить."""
import sys, os
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction


class CameraProcessorNode(Node):
    """Узел обработки камеры. Защита от повторного запуска."""

    def __init__(self, driver):
        super().__init__('camera_processor')
        self.driver = driver
        self._busy = False
        self._timer = self.create_timer(0.1, self._process)
        self.get_logger().info("Camera processor готов")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
```

```
            if frame is None:
                return

            h, w, c = frame.shape

            self.driver.bb.update(
                frame_width=w,
                frame_height=h,
                frame_channels=c,
                frame_dtype=str(frame.dtype),
            )

            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/original.jpg'),
frame)
            self.driver.bb.set('frame_saved', True)
        finally:
            self._busy = False


class FirstFrame(RobotDriver):

    def __init__(self):
        self._processor = CameraProcessorNode(self)
        super().__init__('first_frame', tick_rate=1)
        self.add_node(self._processor)

    def build_tree(self):

        def check_frame(driver):
            if driver.bb.get('frame_saved', False):
                w = driver.bb.get('frame_width', 0)
                h = driver.bb.get('frame_height', 0)
                c = driver.bb.get('frame_channels', 0)
                dtype = driver.bb.get('frame_dtype', '?')
                return f"   Кадр сохранён! {w}×{h}, каналов: {c}, dtype:
{dtype}"
            return "   Ожидаю кадр ... "

        return Sequence([
                EvalAction(lambda d: "   Камера работает ... ", "Start"),
                EvalAction(check_frame, "CheckFrame"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(FirstFrame())


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

Запуск:

```
python3 ~/ws/rbt/scripts/first_frame.py
```

После запуска в `~/ws/rbt/logs/` появится файл `original.jpg`.

2.3. Миссия ColorChannels: разбор каналов

Создайте файл:

```
nano ~/ws/rbt/scripts/color_channels.py
```

```
#!/usr/bin/env python3
"""Миссия: разобрать кадр на каналы и сохранить. Защита от повторного
запуска."""
import sys, os
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction


class ColorChannelsProcessor(Node):
    """Узел разбора кадра на каналы. Защита от повторного запуска."""

    def __init__(self, driver):
        super().__init__('color_channels_processor')
        self.driver = driver
        self._busy = False
        self._timer = self.create_timer(0.1, self._process)
        self.get_logger().info("Color channels processor готов")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
            if frame is None:
                return

            b, g, r = cv2.split(frame)

            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/
channel_b.jpg'), b)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/
channel_g.jpg'), g)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/
channel_r.jpg'), r)

            hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/hsv.jpg'), hsv)

            self.driver.bb.update(
                b_mean=float(b.mean()),
                g_mean=float(g.mean()),
                r_mean=float(r.mean()),
```

```
                channels_saved=True,
            )
        finally:
            self._busy = False


class ColorChannels(RobotDriver):

    def __init__(self):
        self._processor = ColorChannelsProcessor(self)
        super().__init__('color_channels', tick_rate=1)
        self.add_node(self._processor)

    def build_tree(self):

        def check_channels(driver):
            if driver.bb.get('channels_saved', False):
                b = driver.bb.get('b_mean', 0)
                g = driver.bb.get('g_mean', 0)
                r = driver.bb.get('r_mean', 0)
                return f"   Каналы сохранены! B:{b:.0f} G:{g:.0f} R:
{r:.0f}"
            return "   Ожидаю обработку каналов ... "

        return Sequence([
                EvalAction(lambda d: "   Разбор каналов ... ", "Start"),
                EvalAction(check_channels, "CheckChannels"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(ColorChannels())


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

Запуск:

```
python3 ~/ws/rbt/scripts/color_channels.py
```

После запуска в папке `logs` появятся 4 файла: `channel_b.jpg`, `channel_g.jpg`, `channel_r.jpg`, `hsv.jpg`.

2.4. Архитектура: Blackboard, многопоточность и защита от повторного запуска

Как это работает:

1. `CameraProcessorNode` / `ColorChannelsProcessor` — отдельные `Node` с таймером. `MultiThreadedExecutor` вызывает `_process()` в отдельном потоке.
2. **Флаг `_busy`** защищает от повторного входа: если предыдущая обработка ещё не завершилась, новый вызов игнорируется. Это важно, потому что таймер может сработать быстрее, чем закончится обработка.

3. Результаты записываются в `driver.bb` (потокобезопасный `Blackboard`).
4. Дерево поведения читает `driver.bb` через `EvalAction` и выводит результат.
5. `driver.add_node( ... )` регистрирует узел в пуле исполнителя.

Преимущество: тяжёлая обработка (сохранение файлов, разбор каналов) не блокирует дерево поведения и heartbeat.

2.5. Цветовые пространства: BGR → RGB → HSV

OpenCV по умолчанию работает в **BGR** (Blue-Green-Red), а не RGB.

Преобразования:

```
import cv2
import numpy as np

# Создадим тестовое изображение (синий квадрат в BGR)
img_bgr = np.zeros((100, 100, 3), dtype=np.uint8)
img_bgr[:, :] = (255, 0, 0)  # B=255, G=0, R=0 → синий

# BGR → RGB
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

# BGR → HSV
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)

print(f"BGR: {img_bgr[0,0]}")   # [255, 0, 0]
print(f"RGB: {img_rgb[0,0]}")   # [0, 0, 255]
print(f"HSV: {img_hsv[0,0]}")   # [120, 255, 255]
```

Разбор каналов — уже сделан в `ColorChannelsProcessor` через `cv2.split(frame)`.

3. Уровень «Углублённый»: BGR vs RGB и HSV

(Что под капотом)

Почему каналы перепутаны и зачем HSV.

3.1. Почему OpenCV использует BGR, а не RGB

Это историческая причина. OpenCV начинался в Intel, где железо (камеры, чипы) работало с BGR-форматом. Переделывать на RGB было дорого, так и осталось.

Следствие: когда вы работаете с `cv2.imread()`, получаете BGR. Когда отображаете через `cv2.imshow()` — тоже BGR. Если передаёте кадр в другую библиотеку (например, `matplotlib`), нужно преобразовать в RGB:

```
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
```

3.2. HSV: Hue, Saturation, Value

HSV — это представление цвета, интуитивно понятное человеку:

- **Hue (оттенок):** угол от 0 до 179 в OpenCV (0 — красный, 60 — жёлтый, 120 — зелёный, 180 — красный снова).
- **Saturation (насыщенность):** 0 — серый, 255 — чистый цвет.
- **Value (яркость):** 0 — чёрный, 255 — максимально яркий.

Почему HSV удобнее для цветового поиска: в BGR/RGB цвет зависит от освещения. Красный мяч в тени и на свету — очень разные значения (R, G, B). В HSV у него будет близкий **Hue** (оттенок), а освещение влияет только на **Value** (яркость).

Пример: чтобы найти красный объект в HSV:

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv, (0, 100, 100), (10, 255, 255))  # красный диапазон
```

В BGR для этого потребовалось бы сложное условие с тремя каналами.

4. Уровень «Исследовательский»: Гистограммы и освещение

(Прорыв за границы)

Анализ распределения цветов.

4.1. Построение гистограммы каналов

Гистограмма показывает, сколько пикселей имеют каждое значение яркости:

```
import matplotlib.pyplot as plt

frame = driver.get_frame()
color = ('b', 'g', 'r')
for i, col in enumerate(color):
    hist = cv2.calcHist([frame], [i], None, [256], [0, 256])
    plt.plot(hist, color=col)
plt.savefig(os.path.expanduser('~/ws/rbt/logs/histogram.png'))
```

Это покажет распределение яркостей по каждому каналу.

4.2. Мини-исследование: влияние освещения на HSV

Вопрос: Как меняется HSV-представление одного и того же объекта при разном освещении?

Методика:

1. Сфотографируйте цветной объект при ярком свете.
2. Сфотографируйте тот же объект в полутьме.
3. Преобразуйте оба кадра в HSV.

4. Сравните значения Hue для одного и того же пикселя объекта.

Ожидаемый результат: Hue остаётся почти неизменным (±5–10°), Saturation и Value меняются сильнее. Именно поэтому HSV предпочитают для цветового поиска.

5. Выводы по занятию

1. **OpenCV работает с numpy-массивами.** Изображение = (H × W × 3) чисел.
2. **BGR — стандарт OpenCV.** Преобразование `cvtColor` для перехода в RGB или HSV.
3. **HSV удобен для цветового поиска.** Hue не зависит от освещения.
4. **Blackboard + многопоточность** позволяют обрабатывать кадры параллельно с деревом поведения.
5. **Флаг `_busy`** защищает от повторного входа в обработчик.

Связь со следующим занятием: Мы научились получать и разбирать кадры. На следующем занятии — **фильтрация**: размытие, выделение границ (Canny).

Домашнее задание:

- **Базовое**: Запустите `first_frame.py` и `color_channels.py`. Посмотрите сохранённые файлы.
- **Продвинутое**: Сфотографируйте красный объект и найдите его в HSV-пространстве. Запишите диапазон Hue.
- **Исследовательское**: Постройте гистограмму кадров при ярком и тусклом свете. Сравните.

Занятие 46: Фильтрация: размытие, выделение границ Canny

Содержание занятия: Предварительная обработка изображений. Размытие шумов и выделение границ

Цели занятия

- **Образовательная**: Освоить базовые фильтры OpenCV: `GaussianBlur`, `medianBlur`, `Canny`. Научиться подбирать параметры фильтров.
- **Развивающая**: Понять, как работает свёртка и почему медианный фильтр сохраняет края лучше гауссова.
- **Воспитательная**: Показать, что предварительная обработка — обязательный этап перед поиском линий, контуров и объектов.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Размытие и Canny (45 мин)**
    - `GaussianBlur` и `medianBlur`.

- `Canny` с двумя порогами.
- Миссия `FilterTest`: обработка кадра с камеры.

3. **Уровень «Углублённый»: Как работают фильтры (25 мин)**
   - Ядро свёртки.
   - Медианный фильтр и края.
   - Двойной порог Canny и гистерезис.
4. **Уровень «Исследовательский»: Подбор порогов (10 мин)**
   - Оптимальные пороги для разного освещения.
   - Влияние размера ядра.
5. **Подведение итогов (5 мин)**

1. Введение: Зачем нужна предобработка

Сырой кадр с камеры содержит **шумы**: случайные пиксели, неравномерное освещение, мелкие блики. Прежде чем искать линии или объекты, нужно:

1. **Размыть** шумы (но сохранить края).
2. **Выделить границы** — пиксели, где яркость резко меняется.

Сегодня мы освоим три главных инструмента: `GaussianBlur`, `medianBlur` и `Canny`.

2. Уровень «База»: Размытие и Canny

(Инструмент, навык, интерфейс)

Обязательный уровень. Запускаем фильтры на кадре с камеры.

2.1. GaussianBlur и medianBlur

Гауссово размытие (`GaussianBlur`) — усредняет каждый пиксель с соседями по «колоколу» (гауссовой кривой). Чем дальше сосед, тем меньше его вес.

```
blurred = cv2.GaussianBlur(frame, (5, 5), 0)
# (5, 5) — размер ядра (только нечётные!)
# 0 — стандартное отклонение (авто)
```

Медианное размытие (`medianBlur`) — заменяет каждый пиксель на **медиану** (среднее по порядку) значений в окне. Лучше сохраняет края, но медленнее.

```
blurred = cv2.medianBlur(frame, 5)
# 5 — размер окна (только нечётные!)
```

Правило: для удаления мелкого шума перед Canny используют `GaussianBlur 5×5`. Для сохранения тонких линий — `medianBlur`.

2.2. Canny — выделение границ

Алгоритм Canny находит пиксели, где яркость резко меняется:

```
edges = cv2.Canny(frame, 50, 150)
# 50  — нижний порог (слабые границы)
# 150 — верхний порог (сильные границы)
```

Два порога:

- Пиксели с градиентом > 150 — **точно граница** (сильная).
- Пиксели с градиентом < 50 — **точно не граница** (отбрасываются).
- Пиксели между 50 и 150 — **слабые границы**: сохраняются, только если соединены с сильной границей (гистерезис).

Стандартный порядок: `GaussianBlur → Canny`.

2.3. Миссия FilterTest

Создайте файл:

```
nano ~/ws/rbt/scripts/filter_test.py
```

```
#!/usr/bin/env python3
"""Миссия: тестирование фильтров на кадре с камеры."""
import sys, os
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction


class FilterProcessor(Node):
    """Узел обработки: размытие и Canny. Защита от повторного запуска."""

    def __init__(self, driver):
        super().__init__('filter_processor')
        self.driver = driver
        self._busy = False
        self._timer = self.create_timer(0.2, self._process)
        self.get_logger().info("Filter processor готов")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
            if frame is None:
                return

            # 1. Размытие по Гауссу
```

```
            gauss = cv2.GaussianBlur(frame, (5, 5), 0)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/gauss_blur.jpg')
, gauss)

            # 2. Медианное размытие
            median = cv2.medianBlur(frame, 5)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/
median_blur.jpg'), median)

            # 3. Canny (на размытом по Гауссу)
            edges = cv2.Canny(gauss, 50, 150)
            cv2.imwrite(os.path.expanduser('~/ws/rbt/logs/
canny_edges.jpg'), edges)

            # Статистика
            total_pixels = edges.size
            edge_pixels = cv2.countNonZero(edges)
            edge_percent = 100.0 * edge_pixels / total_pixels

            self.driver.bb.update(
                edge_percent=round(edge_percent, 1),
                edge_pixels=edge_pixels,
                filters_saved=True,
            )
        finally:
            self._busy = False


class FilterTest(RobotDriver):

    def __init__(self):
        self._processor = FilterProcessor(self)
        super().__init__('filter_test', tick_rate=1)
        self.add_node(self._processor)

    def build_tree(self):
        def check_filters(driver):
            if driver.bb.get('filters_saved', False):
                pct = driver.bb.get('edge_percent', 0)
                cnt = driver.bb.get('edge_pixels', 0)
                return f"   Фильтры сохранены! Границ: {cnt} пикселей ({pc
t}%)"
            return "   Ожидаю обработку ..."

        return Sequence([
                EvalAction(lambda d: "   Фильтрация кадра ...", "Start"),
                EvalAction(check_filters, "CheckFilters"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(FilterTest())
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/filter_test.py
```

После запуска в `~/ws/rbt/logs/` появятся файлы: `gauss_blur.jpg`, `median_blur.jpg`, `canny_edges.jpg`.

2.4. Эксперименты с параметрами

Измените пороги Canny и посмотрите на результат:

- `(30, 90)` — больше границ, но больше шума.
- `(100, 200)` — только самые сильные границы.
- `(50, 150)` — стандартный компромисс.

Измените размер ядра `GaussianBlur`:

- `(3, 3)` — меньше размытия, больше шума на Canny.
- `(7, 7)` — сильнее размытие, меньше границ.
- `(5, 5)` — стандарт.

3. Уровень «Углублённый»: Как работают фильтры

(Что под капотом)

Математика и логика.

3.1. Ядро свёртки: как GaussianBlur усредняет пиксели

Фильтр «проезжает» по изображению **окном** (ядром). Каждый пиксель заменяется на взвешенную сумму соседей.

Пример ядра 3×3 для гауссова размытия:

```
1/16 × [1 2 1]
       [2 4 2]
       [1 2 1]
```

Центральный пиксель получает вес 4/16, соседи по сторонам — 2/16, по углам — 1/16. Это создаёт плавное размытие.

3.2. Почему medianBlur лучше сохраняет края

Гауссово размытие усредняет **все** пиксели в окне, включая пиксели края. Край «размывается», становится менее чётким.

Медианное размытие берёт **медиану** (среднее по порядку, не по значению!). Если в окне 9 пикселей: `[10, 12, 12, 13, 15, 16, 18, 20, 200]`, медиана = 15. Выброс 200 (шум) отбрасывается, а край (200) не размывается на соседей.

Вывод: для удаления шума перед поиском тонких линий лучше `medianBlur`. Для общего сглаживания — `GaussianBlur`.

3.3. Двойной порог Canny и гистерезис

Градиент — мера изменения яркости. На границе объекта градиент большой.

Canny использует **два порога**:

- `threshold2` (150) — выше этого → **точно граница**.
- `threshold1` (50) — ниже этого → **точно не граница**.
- Между ними — **слабый кандидат**: становится границей, только если **примыкает** к сильной границе.

Гистерезис — это «залипание»: слабый пиксель, соединённый с сильным, считается границей. Так Canny строит непрерывные линии, а не разрозненные точки.

Правило подбора порогов: `threshold2 ≈ 2–3 × threshold1`. Для яркого изображения — выше, для тёмного — ниже.

4. Уровень «Исследовательский»: Подбор порогов

(Прорыв за границы)

Эксперименты с параметрами.

4.1. Оптимальные пороги Canny для разного освещения

Эксперимент:

1. Сделайте кадр при ярком свете. Подберите пороги, при которых границы видны, но шума мало.
2. Сделайте кадр в полутьме. Подберите пороги.
3. Запишите найденные пары `(t1, t2)`.

Ожидаемый результат: для тёмного кадра пороги должны быть **ниже**, потому что градиенты слабее.

4.2. Мини-исследование: размер ядра и количество границ

Вопрос: Как размер ядра `GaussianBlur` влияет на количество найденных границ?

Методика:

1. Запустите `FilterProcessor` с ядром 3×3. Запишите `edge_percent`.
2. Повторите с 5×5, 7×7, 9×9.
3. Постройте график: размер ядра → процент граничных пикселей.

Гипотеза: чем больше ядро, тем сильнее размытие → меньше границ.

5. Выводы по занятию

1. **`GaussianBlur`** — стандартное размытие перед Canny. Ядро 5×5.
2. **`medianBlur`** — лучше сохраняет края, но медленнее.
3. **Canny** — выделяет границы. Два порога: слабый (50) и сильный (150).
4. **Гистерезис** соединяет слабые границы с сильными в непрерывные линии.

Связь со следующим занятием: Мы научились выделять границы. Теперь превратим их в **бинарное изображение** (чёрно-белое) и очистим от шума морфологическими операциями.

Домашнее задание:

- **Базовое**: Запустите `filter_test.py`. Посмотрите сохранённые изображения.
- **Продвинутое**: Подберите пороги Canny для яркого и тёмного кадра.
- **Исследовательское**: Проведите эксперимент с размером ядра (3, 5, 7, 9) и постройте график.

Занятие 47: Бинаризация и морфологические операции

Содержание занятия: Превращение изображения в чёрно-белое и очистка от шумов

Цели занятия

- **Образовательная**: Освоить бинаризацию (`threshold`, `adaptiveThreshold`) и морфологические операции (`erode`, `dilate`, `MORPH_OPEN`, `MORPH_CLOSE`).
- **Развивающая**: Понять, как структурный элемент влияет на результат и почему адаптивный порог лучше при неравномерном освещении.
- **Воспитательная**: Показать, что бинарное изображение — основа для поиска контуров, линий и объектов.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Бинаризация и морфология (45 мин)**
   - `threshold` — глобальный порог.
   - `adaptiveThreshold` — адаптивный порог.
   - Морфология: `erode`, `dilate`, `morphologyEx`.
   - Миссия `MorphologyTest`.
3. **Уровень «Углублённый»: Структурный элемент и адаптивность (25 мин)**
   - `kernel` — прямоугольник, эллипс, крест.
   - Открытие и закрытие.

- Почему `adaptiveThreshold` лучше.

4. **Уровень «Исследовательский»: Сравнение методов (10 мин)**
   - Бинаризация разными методами на одном кадре.
   - Влияние размера `kernel`.
5. **Подведение итогов (5 мин)**

1. Введение: От серого — к чёрно-белому

На прошлом занятии мы выделили границы (Canny). Но для поиска контуров и объектов нужно **бинарное изображение** — только чёрные (0) и белые (255) пиксели.

Бинаризация — это превращение серого изображения в чёрно-белое по порогу. Всё, что темнее порога → чёрное, что светлее → белое (или наоборот).

Морфология — это операции над формой объектов: удалить мелкий шум (открытие), заполнить дыры (закрытие), истончить или утолщить линии.

2. Уровень «База»: Бинаризация и морфология

(Инструмент, навык, интерфейс)

Обязательный уровень. Обрабатываем кадр с камеры.

2.1. threshold и adaptiveThreshold

Глобальный порог (`threshold`):

```
ret, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
# 127 — порог
# 255 — значение для белых пикселей
# THRESH_BINARY — всё что > 127 → 255, остальное → 0
```

Проблема: если левая часть кадра темнее правой, глобальный порог не подходит.

Адаптивный порог (`adaptiveThreshold`):

```
binary = cv2.adaptiveThreshold(gray, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY, 11, 2)
# 11 — размер окна (только нечётные!)
# 2 — константа, вычитаемая из среднего
```

Порог вычисляется **для каждого пикселя** отдельно по его окрестности. Левая тёмная область получит низкий порог, правая светлая — высокий.

2.2. Морфологические операции

Все операции используют **структурный элемент** (`kernel`) — маленькую матрицу, которая «ездит» по изображению.

```
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))

# Эрозия — «съедает» белые пиксели по краям, удаляет мелкий шум
eroded = cv2.erode(binary, kernel, iterations=1)

# Дилатация — «раздувает» белые области, заполняет дыры
dilated = cv2.dilate(binary, kernel, iterations=1)

# Открытие = эрозия + дилатация — удаляет мелкий белый шум
opened = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)

# Закрытие = дилатация + эрозия — заполняет мелкие чёрные дыры
closed = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)
```

Правило: для удаления шума перед поиском контуров — `MORPH_OPEN`.

2.3. Миссия MorphologyTest

Создайте файл:

```
nano ~/ws/rbt/scripts/morphology_test.py
```

```
#!/usr/bin/env python3
"""Миссия: бинаризация и морфология на кадре с камеры."""
import sys, os
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction


class MorphologyProcessor(Node):
    """Узел бинаризации и морфологии. Защита от повторного запуска."""

    def __init__(self, driver):
        super().__init__('morphology_processor')
        self.driver = driver
        self._busy = False
        self._timer = self.create_timer(0.2, self._process)
        self.get_logger().info("Morphology processor готов")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
            if frame is None:
                return

            log_dir = os.path.expanduser('~/ws/rbt/logs')
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
            # 1. Глобальный порог
            _, global_bin = cv2.threshold(gray, 127, 255, cv2.THRESH_BINAR
Y)
            cv2.imwrite(f'{log_dir}/global_binary.jpg', global_bin)

            # 2. Адаптивный порог
            adaptive_bin = cv2.adaptiveThreshold(gray, 255,
                cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                cv2.THRESH_BINARY, 11, 2)
            cv2.imwrite(f'{log_dir}/adaptive_binary.jpg', adaptive_bin)

            # 3. Морфология: открытие на адаптивной бинаризации
            kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
            opened = cv2.morphologyEx(adaptive_bin, cv2.MORPH_OPEN,
kernel)
            cv2.imwrite(f'{log_dir}/morph_opened.jpg', opened)

            # 4. Закрытие
            closed = cv2.morphologyEx(adaptive_bin, cv2.MORPH_CLOSE, kerne
l)
            cv2.imwrite(f'{log_dir}/morph_closed.jpg', closed)

            # Статистика: сколько белых пикселей
            total = adaptive_bin.size
            white_global = cv2.countNonZero(global_bin)
            white_adaptive = cv2.countNonZero(adaptive_bin)
            white_opened = cv2.countNonZero(opened)

            self.driver.bb.update(
                white_global_pct=round(100.0 * white_global / total, 1),
                white_adaptive_pct=round(100.0 * white_adaptive / total,
1),
                white_opened_pct=round(100.0 * white_opened / total, 1),
                morphology_saved=True,
            )
        finally:
            self._busy = False


class MorphologyTest(RobotDriver):

    def __init__(self):
        self._processor = MorphologyProcessor(self)
        super().__init__('morphology_test', tick_rate=1)
        self.add_node(self._processor)

    def build_tree(self):
        def check(driver):
            if driver.bb.get('morphology_saved', False):
                g = driver.bb.get('white_global_pct', 0)
                a = driver.bb.get('white_adaptive_pct', 0)
                o = driver.bb.get('white_opened_pct', 0)
                return f"   Белых пикселей — глоб: {g}%, адапт: {a}%,
открытие: {o}%"
            return "   Ожидаю обработку ..."
```

```
        return Sequence([
                EvalAction(lambda d: "   Бинаризация и морфология ...", "St
art"),
                EvalAction(check, "CheckMorphology"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(MorphologyTest())


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

Запуск:

```
python3 ~/ws/rbt/scripts/morphology_test.py
```

После запуска в `~/ws/rbt/logs/` появятся файлы: `global_binary.jpg`, `adaptive_binary.jpg`, `morph_opened.jpg`, `morph_closed.jpg`.

2.4. Эксперименты с параметрами

Порог adaptiveThreshold:

- Размер окна `11` → меньшие блоки, больше деталей, но и шума.
- Размер окна `21` → большие блоки, меньше шума, но и деталей.
- Константа `2` → порог = среднее − 2. Увеличьте до 5 — меньше белого.

Размер kernel:

- `(2, 2)` — минимальное воздействие.
- `(3, 3)` — стандарт.
- `(5, 5)` — агрессивная очистка, могут исчезнуть тонкие линии.

3. Уровень «Углублённый»: Структурный элемент и адаптивность

(Что под капотом)

Форма и логика.

3.1. Структурный элемент (kernel)

```
cv2.getStructuringElement(форма, (ширина, высота))
```

Форма	Константа	Для чего
Прямоугольник	`MORPH_RECT`	Универсальный
Эллипс	`MORPH_ELLIPSE`	Округлые объекты
Крест	`MORPH_CROSS`	Тонкие линии

Для линии разметки лучше `MORPH_RECT` или `MORPH_CROSS`.

3.2. Открытие и закрытие: что и когда

Открытие (MORPH_OPEN): эрозия → дилатация. Удаляет **мелкие белые точки** (шум), не меняя размер крупных объектов.

Закрытие (MORPH_CLOSE): дилатация → эрозия. Заполняет **мелкие чёрные дыры** внутри объектов, не меняя их форму.

Порядок для линии: `GaussianBlur → adaptiveThreshold → MORPH_OPEN`.

3.3. Почему adaptiveThreshold лучше при неравномерном освещении

Глобальный порог берёт **одно число** (127) для всего кадра. Если лампа освещает левую часть, а правая в тени:

- В тёмной части многие пиксели < 127 → чёрные (потеря информации).
- В светлой части многие > 127 → белые (засветка).

Адаптивный порог вычисляет порог **для каждого пикселя** по его окрестности (11×11). Тёмная область получит низкий порог, светлая — высокий. Линия будет видна везде.

4. Уровень «Исследовательский»: Сравнение методов

(Прорыв за границы)

Эксперименты.

4.1. Сравнение бинаризации разными методами

Эксперимент:

1. Сделайте кадр полигона с линией разметки.
2. Примените `threshold` с порогами 100, 127, 150.
3. Примените `adaptiveThreshold` с окнами 5, 11, 21.
4. Визуально сравните, на каком изображении линия видна лучше всего.

Ожидание: для линии на полу adaptiveThreshold с окном 11–21 работает лучше.

4.2. Мини-исследование: размер kernel и качество очистки линии

Вопрос: При каком размере `kernel` линия очищается от шума, но не исчезает?

Методика:

1. Бинаризуйте кадр с линией.
2. Примените `MORPH_OPEN` с kernel 2×2, 3×3, 5×5, 7×7.
3. Для каждого измерьте количество белых пикселей и визуально оцените целостность линии.

Гипотеза: kernel 3×3 оптимален. 5×5 может разорвать тонкую линию.

5. Выводы по занятию

1. **Бинаризация** превращает серое изображение в чёрно-белое. Глобальный порог прост, но не работает при неравномерном освещении.
2. **adaptiveThreshold** вычисляет порог для каждого пикселя — лучше для реальных условий.
3. **Морфология** чистит бинарное изображение: открытие удаляет шум, закрытие заполняет дыры.
4. **kernel 3×3** — стандарт для очистки линии разметки.

Связь со следующим занятием: Теперь у нас есть чистое бинарное изображение. На следующем занятии — **поиск контуров** и определение геометрических примитивов (треугольники, квадраты, круги).

Домашнее задание:

- **Базовое**: Запустите `morphology_test.py`. Сравните `global_binary.jpg` и `adaptive_binary.jpg`.
- **Продвинутое**: Подберите оптимальные параметры `adaptiveThreshold` для линии на вашем полигоне.
- **Исследовательское**: Проведите эксперимент с размером kernel. Постройте график зависимости количества белых пикселей от размера kernel.

Занятие 48: Поиск контуров, геометрические примитивы

Содержание занятия: Выделение объектов на бинарном изображении и определение их формы

Цели занятия

- **Образовательная**: Освоить поиск контуров (`findContours`), их фильтрацию по площади и периметру, аппроксимацию (`approxPolyDP`) и детектирование геометрических примитивов.
- **Развивающая**: Понять, как работает алгоритм Раммера-Дугласа-Пеккера и почему круг определяется через соотношение площадь/периметр.
- **Воспитательная**: Показать, что форма объекта — это количество вершин аппроксимации.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Контуры и примитивы (45 мин)**
   - `findContours` и `drawContours`.
   - Фильтрация: `contourArea`, `arcLength`.
   - `approxPolyDP` — аппроксимация.
   - Детектирование фигур по количеству вершин.
   - Миссия `ShapeDetector`.
3. **Уровень «Углублённый»: Иерархия и алгоритмы (25 мин)**
   - Иерархия контуров.
   - Метод Раммера-Дугласа-Пеккера.
   - Круг через площадь и периметр.
4. **Уровень «Исследовательский»: Форма и поворот (10 мин)**
   - Детектирование независимо от размера.
   - Влияние угла поворота.
5. **Подведение итогов (5 мин)**

1. Введение: От пикселей — к объектам

У нас есть бинарное изображение (чёрно-белое). Теперь нужно найти на нём **объекты** — непрерывные белые области — и определить их форму.

Контур — это граница объекта, цепочка точек (x, y). OpenCV находит все контуры, а мы фильтруем их и определяем геометрию.

2. Уровень «База»: Контуры и примитивы

(Инструмент, навык, интерфейс)

Обязательный уровень. Обрабатываем кадр и находим фигуры.

2.1. Поиск и отрисовка контуров

```
contours, hierarchy = cv2.findContours(binary, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)
# RETR_EXTERNAL — только внешние контуры
# CHAIN_APPROX_SIMPLE — сжимает прямые участки (меньше точек)

# Отрисовка на цветном кадре
cv2.drawContours(frame, contours, -1, (0, 255, 0), 2)
# -1 — рисовать все контуры
# (0, 255, 0) — зелёный цвет (BGR)
# 2 — толщина линии
```

2.2. Фильтрация по площади и периметру

Мелкие контуры (шум) нужно отбросить:

```
min_area = 500  # минимальная площадь в пикселях
for cnt in contours:
    area = cv2.contourArea(cnt)
    if area < min_area:
        continue  # пропускаем мелкий шум
    perimeter = cv2.arcLength(cnt, True)  # True — замкнутый контур
```

2.3. Аппроксимация и определение формы

`approxPolyDP` заменяет контур на многоугольник с заданной точностью:

```
epsilon = 0.02 * perimeter  # 2% от периметра — точность
approx = cv2.approxPolyDP(cnt, epsilon, True)
vertices = len(approx)  # количество вершин

if vertices == 3:
    shape = "треугольник"
elif vertices == 4:
    shape = "квадрат/прямоугольник"
elif vertices >= 8:
    shape = "круг"
else:
    shape = f"{vertices}-угольник"
```

Для круга дополнительная проверка: `4π × площадь / периметр² ≈ 1` (для идеального круга = 1).

2.4. Миссия ShapeDetector

Создайте файл:

```
nano ~/ws/rbt/scripts/shape_detector.py
```

```
#!/usr/bin/env python3
"""Миссия: поиск контуров и определение фигур на кадре."""
import sys, os, math
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction


class ShapeProcessor(Node):
    """Узел поиска фигур. Защита от повторного запуска."""

    def __init__(self, driver):
        super().__init__('shape_processor')
        self.driver = driver
        self._busy = False
```

```
        self._timer = self.create_timer(0.2, self._process)
        self.get_logger().info("Shape processor готов")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
            if frame is None:
                return

            log_dir = os.path.expanduser('~/ws/rbt/logs')
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            binary = cv2.adaptiveThreshold(gray, 255,
                cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                cv2.THRESH_BINARY, 11, 2)

            # Морфологическая очистка
            kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
            cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)

            # Поиск контуров
            contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                           cv2.CHAIN_APPROX_SIMPLE)

            shapes_found = []
            output = frame.copy()

            for cnt in contours:
                area = cv2.contourArea(cnt)
                if area < 500:  # фильтр шума
                    continue

                perimeter = cv2.arcLength(cnt, True)
                epsilon = 0.02 * perimeter
                approx = cv2.approxPolyDP(cnt, epsilon, True)
                vertices = len(approx)

                # Определение формы
                if vertices == 3:
                    shape = "треугольник"
                elif vertices == 4:
                    shape = "четырёхугольник"
                elif vertices >= 8:
                    shape = "круг"
                else:
                    shape = f"{vertices}-угольник"

                shapes_found.append(shape)
                cv2.drawContours(output, [approx], -1, (0, 255, 0), 2)

            cv2.imwrite(f'{log_dir}/shapes_detected.jpg', output)

            self.driver.bb.update(
```

```
                shapes_count=len(shapes_found),
                shapes_list=shapes_found,
                shapes_saved=True,
            )
        finally:
            self._busy = False


class ShapeDetector(RobotDriver):

    def __init__(self):
        self._processor = ShapeProcessor(self)
        super().__init__('shape_detector', tick_rate=1)
        self.add_node(self._processor)

    def build_tree(self):
        def check(driver):
            if driver.bb.get('shapes_saved', False):
                count = driver.bb.get('shapes_count', 0)
                shapes = driver.bb.get('shapes_list', [])
                return f"   Найдено фигур: {count} — {', '.join(shapes)
if shapes else 'нет'}"
            return "   Ожидаю обработку ..."

        return Sequence([
                EvalAction(lambda d: "   Поиск фигур ...", "Start"),
                EvalAction(check, "CheckShapes"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(ShapeDetector())


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

Запуск:

```
python3 ~/ws/rbt/scripts/shape_detector.py
```

После запуска в `~/ws/rbt/logs/` появится `shapes_detected.jpg` с обведёнными фигурами.

2.5. Эксперименты с параметрами

- `min_area`: увеличивайте, чтобы отсеять больше шума.
- `epsilon = 0.02 * perimeter`: уменьшайте до 0.01 для более точной аппроксимации (больше вершин), увеличивайте до 0.05 для грубой (меньше вершин).

3. Уровень «Углублённый»: Иерархия и алгоритмы

(Что под капотом)

Как это работает внутри.

3.1. Иерархия контуров

`cv2.RETR_EXTERNAL` возвращает только **внешние** контуры. Но есть и другие режимы:

- `RETR_LIST` — все контуры без иерархии.
- `RETR_TREE` — полное дерево: внешние → внутренние → вложенные.

Для поиска объектов на полу обычно достаточно `RETR_EXTERNAL`.

3.2. Алгоритм Раммера-Дугласа-Пеккера (approxPolyDP)

Это рекурсивный алгоритм, который заменяет кривую на ломаную с меньшим числом точек:

1. Соединяет первую и последнюю точки контура прямой.
2. Находит точку, максимально удалённую от прямой.
3. Если расстояние > `epsilon` — разбивает контур в этой точке и повторяет для двух половин.
4. Если расстояние <= `epsilon` — все промежуточные точки отбрасываются.

Результат: контур из сотен точек превращается в многоугольник из 3–10 вершин.

3.3. Почему круг определяется через площадь/периметр

Для круга: `площадь = π × r²`, `периметр = 2 × π × r`.

Подставляем:
`4π × площадь / периметр² = 4π × (πr²) / (2πr)² = 4π²r² / 4π²r² = 1`.

Для квадрата со стороной `a`: `4π × a² / (4a)² = 4π / 16 ≈ 0.785`.

Правило: если `4π × area / perimeter² > 0.85` — это круг. Эта проверка надёжнее, чем считать вершины (у круга их может быть 4–8 после аппроксимации).

4. Уровень «Исследовательский»: Форма и поворот

(Прорыв за границы)

Эксперименты.

4.1. Детектирование объектов по форме независимо от размера

Форма определяется **количеством вершин**, которое не зависит от размера. Треугольник остаётся треугольником, будь он 100 или 1000 пикселей.

Проверка: поднесите треугольник ближе к камере (стал больше) и дальше (стал меньше). `approxPolyDP` должна вернуть 3 вершины в обоих случаях.

4.2. Мини-исследование: угол поворота и точность

Вопрос: Как угол поворота квадрата влияет на количество вершин после `approxPolyDP`?

Методика:

1. Положите квадрат перед камерой. Запишите количество вершин.
2. Поверните на 15°, 30°, 45°. Запишите.

Гипотеза: при повороте на 45° квадрат может быть принят за ромб, но `approxPolyDP` всё равно вернёт 4 вершины, потому что углы не важны — важно количество сторон.

5. Выводы по занятию

1. **`findContours`** находит все замкнутые цепочки белых пикселей.
2. **Фильтрация по площади** отсеивает шум.
3. **`approxPolyDP`** превращает контур в многоугольник, число вершин → форма.
4. **Круг** определяется по соотношению `4π × площадь / периметр² ≈ 1`.

Связь со следующим занятием: Мы обрабатываем отдельные кадры. Но камера выдаёт **поток** 30 кадров/с. На следующем занятии — **CvBridge** и обработка видеострима в реальном времени.

Домашнее задание:

- **Базовое**: Запустите `shape_detector.py` с треугольником, квадратом и кругом в кадре.
- **Продвинутое**: Добавьте проверку круга через соотношение площадь/периметр.
- **Исследовательское**: Поворачивайте квадрат и измеряйте количество вершин. Запишите результаты.

Занятие 49: CvBridge, обработка видеострима в реальном времени

Содержание занятия: Преобразование ROS-сообщений в формат OpenCV и непрерывная обработка кадров

Цели занятия

- **Образовательная**: Понять, как `CvBridge` преобразует `sensor_msgs/Image` в numpy-массив OpenCV. Научиться обрабатывать видеопоток в реальном времени с замером FPS.
- **Развивающая**: Разобраться, почему при медленной обработке кадры пропускаются и как это исправить.
- **Воспитательная**: Показать, что реальный видеопоток — это баланс между качеством обработки и скоростью.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**

2. **Уровень «База»: CvBridge и FPS (45 мин)**
    - Как работает CvBridge в `driver_lib`.
    - Миссия `StreamProcessor`: непрерывная обработка с замером FPS.
    - Сравнение FPS с частотой камеры.
3. **Уровень «Углублённый»: Преобразование и очередь (25 мин)**
    - ROS Image → numpy.
    - Пропуск кадров при медленной обработке.
    - Синхронизация частот.
4. **Уровень «Исследовательский»: Буфер и разрешение (10 мин)**
    - Буфер кадров.
    - Влияние разрешения на FPS.
5. **Подведение итогов (5 мин)**

1. Введение: Мост между мирами

ROS2 передаёт изображения в формате `sensor_msgs/Image` — это специальное сообщение с заголовком, временной меткой и сырыми байтами. OpenCV работает с numpy-массивами.

CvBridge — это «мост», который преобразует одно в другое. В `driver_lib` он уже встроен в метод `_on_camera`:

```
frame = self._cv_bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
```

Мы используем его с первого занятия с камерой, даже не задумываясь. Сегодня разберёмся, как он работает и как мерить скорость обработки.

2. Уровень «База»: CvBridge и FPS

(Инструмент, навык, интерфейс)

Обязательный уровень. Обрабатываем поток и меряем скорость.

2.1. Как CvBridge уже работает в driver_lib

Метод `_on_camera` вызывается каждый раз, когда приходит новое сообщение `/camera/image_raw`:

```
def _on_camera(self, msg):
    frame = self._cv_bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    self.bb.update(last_frame=frame, camera_ready=True)
```

Кадр сохраняется в `Blackboard`. Наши `ProcessorNode` читают его через `driver.get_frame()`.

Частота камеры: обычно 30 FPS (кадров в секунду). `_on_camera` вызывается 30 раз в секунду. **Частота обработки:** наш `ProcessorNode` с таймером 0.1 с обрабатывает 10 FPS.

Вывод: мы обрабатываем только каждый 3-й кадр. Остальные пропускаются — это нормально для медленной обработки.

2.2. Измерение FPS обработки через время ROS2

Вместо `time.time()` используем `self.get_clock().now()` — это даёт время ROS2, которое можно ускорять/замедлять в симуляции.

2.3. Миссия StreamProcessor

Создайте файл:

```
nano ~/ws/rbt/scripts/stream_processor.py
```

```
#!/usr/bin/env python3
"""Миссия: обработка видеопотока с замером FPS (время ROS2)."""
import sys, os
import rclpy
import cv2
from rclpy.node import Node
from driver_lib import RobotDriver, Sequence, EvalAction, Parallel, Heartb
eatAction
from driver_lib import RepeatUntil

class StreamProcessor(Node):
    """Узел непрерывной обработки с замером FPS через время ROS2."""

    def __init__(self, driver):
        super().__init__('stream_processor')
        self.driver = driver
        self._busy = False
        self._frame_count = 0
        self._start_time = self.get_clock().now()
        self._timer = self.create_timer(0.1, self._process)
        self.get_logger().info("Stream processor готов (10 Гц)")

    def _process(self):
        if self._busy:
            return

        self._busy = True
        try:
            frame = self.driver.get_frame()
            if frame is None:
                return

            self._frame_count += 1

            # Лёгкая обработка: контуры на лету
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            binary = cv2.adaptiveThreshold(gray, 255,
                cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                cv2.THRESH_BINARY, 11, 2)
            contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
```

```
                                                cv2.CHAIN_APPROX_SIMPLE)

            # Считаем FPS раз в секунду
            now = self.get_clock().now()
            elapsed = (now - self._start_time).nanoseconds / 1e9
            if elapsed >= 1.0:
                fps = self._frame_count / elapsed
                self.driver.bb.update(
                    current_fps=round(fps, 1),
                    contours_count=len(contours),
                )
                self._frame_count = 0
                self._start_time = now

        finally:
            self._busy = False


class StreamProcessorMission(RobotDriver):

    def __init__(self):
        self._processor = StreamProcessor(self)
        super().__init__('stream_processor_mission', tick_rate=2)
        self.add_node(self._processor)

    def build_tree(self):
        def check_fps(driver):
            fps = driver.bb.get('current_fps', 0)
            cnt = driver.bb.get('contours_count', 0)
            driver.get_logger().info(f"   FPS: {fps}, контуров: {cnt}", th
rottle_duration_sec=1)
            return f"FPS: {fps}, контуров: {cnt}"

        return Sequence([
            EvalAction(lambda d: "   Обработка потока ... ", "Start"),
            RepeatUntil(
                child=EvalAction(check_fps, "CheckFPS"),
                stop_condition=lambda d: False,  # Бесконечно (прервать
Ctrl+C)
                name="Мониторинг FPS"
            ),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(StreamProcessorMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/stream_processor.py
```

В логах будет выводиться текущий FPS обработки и количество найденных контуров.

2.4. Наблюдение

Типичный вывод:

```
FPS обработки: 9.5, контуров: 3
FPS обработки: 9.8, контуров: 5
```

FPS обработки < 10, хотя таймер стоит на 10 Гц. Почему? Обработка занимает время, и `_busy` иногда блокирует вызов. Это нормально.

Частота камеры 30 FPS, обработка 10 FPS: каждый 3-й кадр обрабатывается, 2 из 3 пропускаются.

3. Уровень «Углублённый»: Преобразование и очередь

(Что под капотом)

Как CvBridge работает и почему кадры пропускаются.

3.1. ROS Image → numpy-массив

```python
from cv_bridge import CvBridge
bridge = CvBridge()

# Прямое преобразование: ROS → OpenCV
frame = bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
# frame — numpy.ndarray, shape=(H, W, 3), dtype=uint8

# Обратное: OpenCV → ROS
ros_msg = bridge.cv2_to_imgmsg(frame, encoding='bgr8')
```

desired_encoding='bgr8' означает: 8 бит на канал, порядок BGR. Можно указать `'mono8'` для ч/б, `'rgb8'` для RGB.

3.2. Пропуск кадров при медленной обработке

Камера публикует 30 кадров/с. Подписчик ROS2 имеет **очередь** (queue_size=10). Если обработчик `_on_camera` не успевает, кадры накапливаются в очереди. Когда очередь переполняется — старые кадры выбрасываются.

Мы читаем `driver.get_frame()` с частотой 10 Гц — получаем **последний** кадр из `Blackboard`. Предыдущие 2 кадра, пришедшие между нашими чтениями, перезаписали `last_frame`. Они потеряны.

Это нормально для большинства задач. Для точной синхронизации (лидар + камера) нужно сравнивать временные метки (`header.stamp`).

3.3. Синхронизация частоты обработки с частотой камеры

Частота обработки	Что происходит
1 Гц	Каждый 30-й кадр. Большая задержка, но низкая нагрузка.
10 Гц	Каждый 3-й кадр. Хороший баланс.
30 Гц	Каждый кадр. Максимальная точность, но процессор должен успевать.

Для линии на скорости 0.2 м/с: при 10 Гц между кадрами робот проезжает 2 см — погрешность допустима. При 1 Гц — 20 см, можно сойти с линии.

4. Уровень «Исследовательский»: Буфер и разрешение

(Прорыв за границы)

Эксперименты с производительностью.

4.1. Буфер кадров для стабильной обработки

Вместо одного `last_frame` можно хранить **кольцевой буфер** последних N кадров. Тогда обработчик сможет взять кадр, ближайший по времени к данным лидара.

Это усложняет код, но улучшает синхронизацию. В `driver_lib` пока используется один `last_frame`.

4.2. Мини-исследование: разрешение и FPS

Вопрос: При каком разрешении кадра FPS обработки падает ниже 10?

Методика:

1. Запустите `StreamProcessor` с полным разрешением (1920×1080). Запишите FPS.
2. Измените в `_process` уменьшение кадра: `frame = cv2.resize(frame, (960, 540))`.
3. Повторите с (640, 360) и (320, 180).
4. Постройте график: разрешение → FPS.

Ожидание: FPS растёт обратно пропорционально количеству пикселей.

5. Выводы по занятию

1. **CvBridge** преобразует `sensor_msgs/Image` в numpy-массив и обратно. В `driver_lib` это уже встроено.
2. **FPS обработки** измеряется подсчётом обработанных кадров за секунду через `get_clock().now()`.
3. **Кадры пропускаются** из-за очереди и перезаписи `last_frame`. Это нормально при частоте обработки < частоты камеры.

4. **Скорость обработки зависит от разрешения** и сложности алгоритмов.

Связь со следующим занятием: Мы научились обрабатывать поток. Теперь решим конкретную задачу — **детектирование линии разметки**: найдём контур линии, вычислим смещение и угол.

Домашнее задание:

- **Базовое**: Запустите `stream_processor.py`. Запишите FPS.
- **Продвинутое**: Добавьте отрисовку контуров на кадре и сохраняйте каждый 10-й обработанный кадр.
- **Исследовательское**: Проведите замер FPS для 4 разрешений. Постройте график.

Занятие 50: Детектирование линии разметки

Содержание занятия: Поиск линии, вычисление смещения и угла, HTTP-стрим для визуализации

Цели занятия

- **Образовательная**: Написать класс `LineDetectorProcessor` (наследник `ImageProcessorNode`) для поиска линии разметки с визуализацией через `HttpStreamerNode`.
- **Развивающая**: Понять, как работает `moments` и `atan2`. Освоить архитектуру с несколькими параллельными обработчиками.
- **Воспитательная**: Показать полный цикл: сенсор → обработка → визуализация → управление.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: LineDetectorProcessor + HttpStreamerNode (45 мин)**
   - Наследование от `ImageProcessorNode`.
   - Конвейер обработки в `process_frame()`.
   - `HttpStreamerNode` — стрим в браузер.
   - Миссия `LineFollowMission`: следование по линии до препятствия.
3. **Уровень «Углублённый»: moments и atan2 (25 мин)**
   - `moments` — центр масс контура.
   - `atan2` — угол отклонения.
   - Перекрёстки и потеря линии.
4. **Уровень «Исследовательский»: Перекрёстки и высота камеры (10 мин)**
   - Детектирование перекрёстка.
   - Влияние высоты камеры на ширину линии.
5. **Подведение итогов (5 мин)**

1. Введение: Полный цикл

Мы умеем находить контуры. Теперь выделим из них **линию разметки** — и не просто выведем цифры в лог, а **увидим** результат в браузере через HTTP-стрим.

В новой версии `driver_lib` добавлены:

- **`ImageProcessorNode`** — базовый класс для обработки кадров (защита от повторного входа, таймер, `process_frame`).
- **`HttpStreamerNode`** — готовая нода HTTP-стримера, берёт JPEG из `Blackboard` по ключу.

Нам остаётся только унаследоваться от `ImageProcessorNode` и переопределить `process_frame()`.

2. Уровень «База»: LineDetectorProcessor + HttpStreamerNode

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем детектор линии с визуализацией.

2.1. Конвейер обработки

```
Кадр (BGR) → gray → adaptiveThreshold → MORPH_OPEN
  → findContours → max area contour
  → moments → центр масс (cx, cy)
  → atan2 → угол отклонения от вертикали
  → отрисовка на копии кадра (контур, центр, направление, метка центра)
  → JPEG в Blackboard ('stream_jpeg')
  → driver.set_line_info(offset, angle)
```

`HttpStreamerNode` забирает `stream_jpeg` и раздаёт по HTTP.

2.2. Код миссии LineFollowMission

Создайте файл:

```
nano ~/ws/rbt/scripts/line_follow_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: детектирование линии + HTTP стример + движение до
препятствия."""
import math
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, Parallel,
                        HeartbeatAction, StreamAction, HttpStreamerNode,
                        ImageProcessorNode)
```

```
# ═══ Детектор линии (только обработка!) ═══

class LineDetectorProcessor(ImageProcessorNode):

    def process_frame(self, frame):
        h, w = frame.shape[:2]
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
            cv2.THRESH_BINARY, 11, 2)

        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)

        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        display = frame.copy()

        if not contours:
            self.driver.set_line_info(0.0, 0.0, detected=False)
            cv2.putText(display, "NO LINE", (10, 30),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
        else:
            best = max(contours, key=cv2.contourArea)
            area = cv2.contourArea(best)

            if area < 500:
                self.driver.set_line_info(0.0, 0.0, detected=False)
                cv2.putText(display, "LINE TOO SMALL", (10, 30),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
            else:
                M = cv2.moments(best)
                if M['m00'] == 0:
                    self.driver.set_line_info(0.0, 0.0, detected=False)
                else:
                    cx = int(M['m10'] / M['m00'])
                    cy = int(M['m01'] / M['m00'])
                    offset = cx - w / 2
                    topmost = tuple(best[best[:, :, 1].argmin()][0])
                    angle = math.atan2(topmost[0] - cx, cy - topmost[1])

                    self.driver.set_line_info(offset, angle,
detected=True)
                    self.driver.bb.update(
                        line_area=area,
                        line_cx=cx,
                        line_cy=cy,
                    )

                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2)
                    cv2.circle(display, (cx, cy), 5, (0, 0, 255), -1)
                    cv2.line(display, (cx, cy), (topmost[0], topmost[1]),
(255, 0, 0), 2)
                    cv2.drawMarker(display, (w // 2, h // 2), (255, 255, 2
55),
```

```
                                    cv2.MARKER_CROSS, 20, 2)
                    cv2.putText(display, f"Off:{offset:.0f} Ang:{math.degr
ees(angle):.0f}", (10, 30),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255,
0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('stream_jpeg', jpeg.tobytes())


# ═══════ Миссия ═══════

class LineFollowMission(RobotDriver):

    def __init__(self):
        super().__init__('line_follow_mission', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='stream_jpeg', port=80
80))
        self.add_node(LineDetectorProcessor(self, fps=30))

    def build_tree(self):

        def obstacle_ahead(driver):
            return driver.get_lidar_front(26) < 0.4

        def should_stop(driver):
            if obstacle_ahead(driver):
                driver.get_logger().info("   Препятствие!")
                return True
            return False

        def drive_with_line(driver):
            offset, angle, detected = driver.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            return (vx, 0.0, omega, 0.3)

        return Sequence([
                EvalAction(lambda d: "   Следование по линии + HTTP стрим
на :8080/stream", "Start"),
                StreamAction(
                    command_func=drive_with_line,
                    stop_condition=should_stop,
                    name="LineFollow"
                ),
                EvalAction(lambda d: "   Остановлен перед препятствием.",
"Finish"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(LineFollowMission())
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/line_follow_mission.py
```

Просмотр стрима: откройте браузер на `http://<ip_робота>:8080/stream`.

2.3. Архитектура с параллельными нодами

1. **`LineDetectorProcessor(ImageProcessorNode)`** — наследует готовый скелет: таймер, защита `_busy`, вызов `process_frame()`. Мы только переопределяем обработку.
2. **`HttpStreamerNode`** — готовая нода, принимает `bb`, `key` (имя поля в Blackboard) и `port`.
3. **`LineFollowMission`** — дерево поведения, управляет движением.

Все три работают параллельно в `MultiThreadedExecutor`. Можно добавить ещё обработчиков — например, детектор ArUco параллельно с детектором линии.

2.4. Управление по линии

Функция `drive_with_line`:

```
omega = -offset / 200.0
```

Если линия смещена вправо (`offset > 0`), `omega` отрицательная → робот поворачивает вправо, возвращаясь к линии. Это простейший **П-регулятор** по смещению.

Коэффициент `1/200` означает: при смещении 200 пикселей `omega = ±0.3` рад/с (максимальная скорость поворота).

3. Уровень «Углублённый»: moments и atan2

(Что под капотом)

Математика детектирования.

3.1. moments — центр масс контура

Моменты — статистические характеристики контура:

- `m00` — площадь.
- `m10` — сумма x-координат.
- `m01` — сумма y-координат.

Центр масс: `cx = m10 / m00`, `cy = m01 / m00`. **Смещение:** `offset = cx - w/2` — на сколько пикселей центр линии правее/левее центра кадра.

3.2. atan2 — угол отклонения

Вектор от центра масс к самой верхней точке контура. `atan2` даёт угол:

- `≈ 0` — вертикально.
- `> 0` — отклонена вправо.
- `< 0` — отклонена влево.

3.3. Почему линия теряется на перекрёстках

Контур раздваивается или сливается с поперечной линией. Алгоритм «самый большой контур» может выбрать не то. Решение: детектировать перекрёсток и временно ехать прямо.

4. Уровень «Исследовательский»: Перекрёстки и высота камеры

(Прорыв за границы)

Эксперименты.

4.1. Детектирование перекрёстка

Идея: если `findContours` нашёл **два** больших контура — это перекрёсток. Запишите флаг в `Blackboard` и временно игнорируйте угол.

4.2. Мини-исследование: высота камеры и ширина линии

Вопрос: Как высота камеры влияет на ширину линии в пикселях?

Методика: измерьте ширину линии на стриме при разной высоте камеры (поднимите/опустите на 5 см).

Ожидание: чем выше камера, тем тоньше линия.

5. Выводы по занятию

1. **`ImageProcessorNode`** — готовый скелет для обработки кадров. Только переопределите `process_frame()`.
2. **`HttpStreamerNode`** — стрим результата в браузер одной строкой.
3. **Конвейер линии:** `gray → adaptiveThreshold → MORPH_OPEN → findContours → max area → moments → offset & angle`.
4. **Управление:** `omega = -offset / 200` — П-регулятор по смещению.

Связь со следующим занятием: Мы следуем по линии. Теперь добавим **трекинг цветного объекта** по HSV — найдём мяч и будем за ним следовать.

Домашнее задание:

- **Базовое**: Запустите миссию и посмотрите стрим в браузере.

- **Продвинутое**: Подберите коэффициент `200` для вашей камеры.
- **Исследовательское**: Измерьте ширину линии в пикселях на разной высоте камеры.

Занятие 51: Трекинг объекта по цвету в HSV

Содержание занятия: Поиск цветного объекта через HSV-маску, морфологию и моменты. Адаптивный ROI

Цели занятия

- **Образовательная**: Написать `ColorTrackerProcessor` (наследник `ImageProcessorNode`) для поиска объекта по цвету в HSV. Освоить подбор диапазона HSV и адаптивный ROI.
- **Развивающая**: Понять, почему HSV устойчивее BGR к освещению, и как ROI ускоряет поиск.
- **Воспитательная**: Показать, что трекинг — это детектирование + память о предыдущем положении.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: ColorTrackerProcessor (45 мин)**
   - `inRange` по HSV.
   - Морфологическая очистка.
   - `moments` → центр масс.
   - `driver.set_object_position(x, y)`.
   - Миссия с HTTP-стримом: робот стоит, в кадре мяч, выводит координаты.
3. **Уровень «Углублённый»: HSV и ROI (25 мин)**
   - Подбор диапазона HSV.
   - Устойчивость HSV к освещению.
   - Адаптивный ROI.
4. **Уровень «Исследовательский»: Многоцветный трекинг и скорость ROI (10 мин)**
   - Список трекеров для разных цветов.
   - Влияние размера ROI на скорость.
5. **Подведение итогов (5 мин)**

1. Введение: От линии — к цвету

Линия — это контраст яркости. Но многие объекты (мячи, знаки, одежда) отличаются **цветом**. HSV-пространство позволяет искать их, задав диапазон оттенка (Hue).

Сегодня мы напишем `ColorTrackerProcessor` — детектор цветного объекта с визуализацией через HTTP-стрим.

2. Уровень «База»: ColorTrackerProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем трекер и смотрим в браузере.

2.1. Конвейер обработки

```
Кадр (BGR)
  → cvtColor(HSV)
  → inRange(lower_hsv, upper_hsv) → маска
  → morphologyEx(MORPH_OPEN) → очистка
  → findContours → max area contour
  → moments → центр масс (cx, cy)
  → driver.set_object_position(cx, cy)
  → отрисовка: контур, центр, bounding box
  → JPEG в Blackboard ('tracker_jpeg')
```

2.2. Подбор диапазона HSV

Для красного мяча (два диапазона, т.к. красный «обёрнут» вокруг 0°):

```
lower_hsv = (0, 100, 100)    # H: 0-10
upper_hsv = (10, 255, 255)
# ИЛИ
lower_hsv = (160, 100, 100)  # H: 160-179
upper_hsv = (179, 255, 255)
```

Для зелёного:

```
lower_hsv = (40, 100, 100)
upper_hsv = (80, 255, 255)
```

Для синего:

```
lower_hsv = (100, 100, 100)
upper_hsv = (130, 255, 255)
```

2.3. Код миссии ColorTrackerMission

Создайте файл:

```
nano ~/ws/rbt/scripts/color_tracker_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: трекинг цветного объекта + HTTP стрим."""
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, Parallel,
                        HeartbeatAction, HttpStreamerNode,
                        ImageProcessorNode)
```

```python
class ColorTrackerProcessor(ImageProcessorNode):
    """Трекер цветного объекта в HSV."""

    def __init__(self, driver, fps=30,
                 lower_hsv=(0, 100, 100), upper_hsv=(10, 255, 255)):
        super().__init__(driver, fps=fps)
        self.lower_hsv = lower_hsv
        self.upper_hsv = upper_hsv
        self._last_roi = None  # (x, y, w, h) последнего положения

    def process_frame(self, frame):
        h, w = frame.shape[:2]
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

        # Если знаем где был объект — ищем только в ROI
        if self._last_roi is not None:
            x, y, rw, rh = self._last_roi

            # Жестко страхуем координаты среза под текущий кадр
            x1 = max(0, min(x, w - 1))
            y1 = max(0, min(y, h - 1))
            x2 = max(1, min(x + rw, w))
            y2 = max(1, min(y + rh, h))

            roi = hsv[y1:y2, x1:x2]

            # Если срез все-таки схлопнулся в 0, ищем по всему кадру
            if roi.size > 0:
                mask_roi = cv2.inRange(roi, self.lower_hsv,
self.upper_hsv)
                mask = np.zeros((h, w), dtype=np.uint8)
                mask[y1:y2, x1:x2] = mask_roi
            else:
                mask = cv2.inRange(hsv, self.lower_hsv, self.upper_hsv)
                self._last_roi = None
        else:
            mask = cv2.inRange(hsv, self.lower_hsv, self.upper_hsv)

        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        display = frame.copy()

        if contours:
            best = max(contours, key=cv2.contourArea)
            area = cv2.contourArea(best)
            if area > 200:
                M = cv2.moments(best)
                if M['m00'] > 0:
                    cx = int(M['m10'] / M['m00'])
                    cy = int(M['m01'] / M['m00'])
```

```
                    self.driver.set_object_position(cx, cy, detected=True)
                    self.driver.bb.update(obj_cx=cx, obj_cy=cy, obj_area=a
rea)

                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2)
                    cv2.circle(display, (cx, cy), 7, (0, 0, 255), -1)
                    cv2.putText(display, f"({cx},{cy}) A:{area:.0f}",
(cx+10, cy),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255,
0), 2)

                    # Считаем новый ROI через абсолютные координаты углов
                    margin = 40
                    nx1 = max(0, cx - margin)
                    ny1 = max(0, cy - margin)
                    nx2 = max(0, min(cx + margin, w))
                    ny2 = max(0, min(cy + margin, h))

                    # Сохраняем обратно в формате (x, y, w, h)
                    self._last_roi = (nx1, ny1, nx2 - nx1, ny2 - ny1)
                else:
                    self.driver.set_object_position(0, 0, detected=False)
            else:
                self.driver.set_object_position(0, 0, detected=False)
                self._last_roi = None
        else:
            self.driver.set_object_position(0, 0, detected=False)
            self._last_roi = None

        # Рисуем рамку ROI на экране отладки
        if self._last_roi is not None:
            x, y, rw, rh = self._last_roi
            cv2.rectangle(display, (x, y), (x+rw, y+rh), (255, 255, 0), 1)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('tracker_jpeg', jpeg.tobytes())


class ColorTrackerMission(RobotDriver):

    def __init__(self):
        super().__init__('color_tracker_mission', tick_rate=2)
        self.add_node(HttpStreamerNode(self.bb, key='tracker_jpeg', port=8
080))
        # Трекинг красного объекта
        self.add_node(ColorTrackerProcessor(self, fps=30,
            lower_hsv=(0, 100, 100), upper_hsv=(10, 255, 255)))

    def build_tree(self):
        def check(driver):
            ox, oy, detected = driver.get_object_position()
            if detected:
                return f"   Объект: ({ox}, {oy})"
            return "   Ищу объект ... "
```

```
        return Sequence([
                EvalAction(lambda d: "   Трекинг красного объекта + стрим
:8080/stream", "Start"),
                EvalAction(check, "CheckObject"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(ColorTrackerMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/color_tracker_mission.py
```

Тест: поместите красный мяч в кадр. В браузере `http://<ip>:8080/stream` увидите обводку и координаты.

2.4. Адаптивный ROI

Когда объект найден, `_last_roi` запоминает прямоугольник вокруг него с запасом 40 пикселей. На следующем кадре `inRange` применяется **только к этой области** — это ускоряет обработку в 4–10 раз.

Если объект потерян — `_last_roi = None`, и маска строится по всему кадру.

3. Уровень «Углублённый»: HSV и ROI

(Что под капотом)

Почему HSV и зачем ROI.

3.1. Почему HSV устойчивее BGR

В BGR красный мяч в тени и на свету — это разные тройки (B, G, R). В HSV у них будет **близкий Hue** (0–10°), а освещение повлияет только на Value. `inRange` по Hue стабильно находит объект при разном освещении.

3.2. Адаптивный ROI: расширение при потере

Если объект не найден в текущем ROI, можно расширить ROI на 50%, а не сбрасывать сразу на весь кадр. Это даёт второй шанс найти объект рядом с последним положением.

4. Уровень «Исследовательский»: Многоцветный трекинг и скорость ROI

(Прорыв за границы)

Эксперименты.

4.1. Трекинг нескольких объектов

Добавьте в `__init__` миссии второй `ColorTrackerProcessor` для зелёного мяча с другим ключом `tracker_jpeg` и другим портом (8081). Получите два стрима одновременно.

4.2. Мини-исследование: размер ROI и скорость

Вопрос: Во сколько раз ускоряется обработка при использовании ROI?

Методика:

1. Замерьте FPS без ROI (сбросьте `_last_roi = None`).
2. Замерьте FPS с ROI (объект в кадре).
3. Сравните.

Ожидание: ускорение пропорционально отношению площади ROI к площади кадра.

5. Выводы по занятию

1. **HSV-маска** (`inRange`) — основной инструмент цветового поиска.
2. **Морфология** очищает маску от шума.
3. `moments` даёт центр масс объекта.
4. **Адаптивный ROI** ускоряет поиск в 4–10 раз.

Связь со следующим занятием: Мы ищем объекты на всём кадре. Но иногда нужно искать **только в определённой зоне** — например, где лидар увидел препятствие. Это **ROI по лидару** — тема следующего занятия.

Домашнее задание:

- **Базовое**: Запустите миссию с красным мячом. Посмотрите стрим.
- **Продвинутое**: Подберите HSV-диапазон для зелёного или синего объекта.
- **Исследовательское**: Замерьте FPS с ROI и без. Вычислите ускорение.

Занятие 52: Выделение зон интереса (ROI)

Содержание занятия: Ограничение области поиска для ускорения обработки. Связка лидара и камеры

Цели занятия

- **Образовательная**: Научиться задавать ROI (Region of Interest) в `ImageProcessorNode` и связывать зону поиска с данными лидара.

- **Развивающая**: Понять, как преобразовать угол лидара в пиксельные координаты камеры и почему возникает параллакс.
- **Воспитательная**: Показать, что сенсоры работают лучше вместе: лидар указывает **где искать**, камера — **что искать**.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: ROI в процессоре и связка с лидаром (45 мин)**
   - Добавление ROI в `process_frame()`.
   - Установка ROI из дерева поведения через `Blackboard`.
   - Миссия: при препятствии спереди — ROI на переднюю область, поиск линии.
3. **Уровень «Углублённый»: Лидар → пиксели и параллакс (25 мин)**
   - Преобразование угла в координаты кадра.
   - Параллакс и калибровка.
4. **Уровень «Исследовательский»: Авто-ROI и ускорение (10 мин)**
   - Несколько ROI по секторам лидара.
   - Замер ускорения.
5. **Подведение итогов (5 мин)**

1. Введение: Не весь кадр одинаково полезен

Камера видит 70° по горизонтали. Но если лидар показывает препятствие **только спереди**, зачем искать линию по всему кадру? Можно ограничить поиск **передней областью** — это ускорит обработку.

Сегодня мы научимся задавать **ROI (Region of Interest)** — прямоугольник в кадре, где ведётся поиск. И свяжем ROI с данными лидара.

2. Уровень «База»: ROI в процессоре и связка с лидаром

(Инструмент, навык, интерфейс)

Обязательный уровень. Модифицируем `LineDetectorProcessor` и добавляем логику ROI.

2.1. ROI в process_frame()

Если в `Blackboard` задан `roi_x, roi_y, roi_w, roi_h`, обработка идёт только в этом прямоугольнике. Иначе — по всему кадру.

```
def process_frame(self, frame):
    h, w = frame.shape[:2]
    # Читаем ROI из Blackboard
    rx = self.driver.bb.get('roi_x', 0)
    ry = self.driver.bb.get('roi_y', 0)
    rw = self.driver.bb.get('roi_w', w)
    rh = self.driver.bb.get('roi_h', h)
```

```
    # Обрезаем кадр
    roi_frame = frame[ry:ry+rh, rx:rx+rw]
    #  ... обработка roi_frame  ...
    # Смещение центра линии корректируем на rx, ry
    cx += rx
    cy += ry
```

2.2. Связка с лидаром: установка ROI из дерева

Когда лидар видит препятствие спереди, устанавливаем ROI на нижнюю половину кадра (там линия ближе к роботу):

```
Condition(lambda d: d.get_lidar_front() < 0.5),
EvalAction(lambda d: (
    d.bb.update(roi_x=0, roi_y=d.get_frame().shape[0]//2,
                roi_w=d.get_frame().shape[1],
roi_h=d.get_frame().shape[0]//2),
    "ROI: передняя область"
)[1], name="SetROI"),
```

2.3. Код миссии LineFollowWithROI

Создайте файл:

```
nano ~/ws/rbt/scripts/line_follow_roi.py
```

```
#!/usr/bin/env python3
"""Миссия: следование по линии с ROI от лидара + HTTP стрим."""
import math
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, Parallel,
                        HeartbeatAction, StreamAction, Condition,
                        HttpStreamerNode, ImageProcessorNode)


class LineDetectorWithROI(ImageProcessorNode):
    """Детектор линии с поддержкой ROI."""

    def process_frame(self, frame):
        h, w = frame.shape[:2]
        # Читаем ROI из Blackboard (если не задан — весь кадр)
        rx = self.driver.bb.get('roi_x', 0)
        ry = self.driver.bb.get('roi_y', 0)
        rw = self.driver.bb.get('roi_w', w)
        rh = self.driver.bb.get('roi_h', h)


        # ИСПРАВЛЕНИЕ: Ограничиваем координаты, чтобы они физически не
вылезали за кадр
        rx = max(0, min(rx, w - 1))
        ry = max(0, min(ry, h - 1))
```

```
        rw = max(1, min(rw, w - rx))
        rh = max(1, min(rh, h - ry))
        roi_frame = frame[ry:ry+rh, rx:rx+rw]


        # Защитная проверка: если ROI всё равно пустой, выходим без
падения
        if roi_frame.size == 0:
            self.driver.set_line_info(0.0, 0.0, detected=False)
            return
        gray = cv2.cvtColor(roi_frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        display = frame.copy()
        cv2.rectangle(display, (rx, ry), (rx+rw, ry+rh), (255, 255, 0), 2)

        if contours:
            best = max(contours, key=cv2.contourArea)
            area = cv2.contourArea(best)
            if area > 500:
                M = cv2.moments(best)
                if M['m00'] > 0:
                    cx = int(M['m10'] / M['m00']) + rx
                    cy = int(M['m01'] / M['m00']) + ry
                    offset = cx - w / 2
                    topmost = tuple(best[best[:, :, 1].argmin()][0])
                    angle = math.atan2(topmost[0] + rx - cx, cy -
topmost[1] - ry)
                    self.driver.set_line_info(offset, angle,
detected=True)
                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2,
offset=(rx, ry))
                    cv2.circle(display, (cx, cy), 5, (0, 0, 255), -1)
                    cv2.putText(display, f"Off:{offset:.0f}", (10, 30),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255,
0), 2)
                else:
                    self.driver.set_line_info(0.0, 0.0, detected=False)
            else:
                self.driver.set_line_info(0.0, 0.0, detected=False)
        else:
            self.driver.set_line_info(0.0, 0.0, detected=False)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('stream_jpeg', jpeg.tobytes())


class LineFollowWithROI(RobotDriver):

    def __init__(self):
```

```
        super().__init__('line_follow_roi', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='stream_jpeg', port=80
80))
        self.add_node(LineDetectorWithROI(self, fps=30))

    def build_tree(self):
        def obstacle_ahead(d):
            return d.get_lidar_front(26) < 4.0

        def set_front_roi(d):
            frame = d.get_frame()
            if frame is not None:
                h, w = frame.shape[:2]
                d.bb.update(roi_x=0, roi_y=h//3, roi_w=w, roi_h=2*h//3)
            return "ROI: передняя область"

        def clear_roi(d):
            d.bb.update(roi_x=0, roi_y=0, roi_w=0, roi_h=0)
            return "ROI: весь кадр"

        def drive_with_line(d):
            offset, angle, detected = d.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            return (vx, 0.0, omega, 0.3)

        return Sequence([
                EvalAction(lambda d: "   Следование по линии + ROI от
лидара", "Start"),
                Condition(obstacle_ahead),
                EvalAction(set_front_roi, "SetFrontROI"),
                StreamAction(
                    command_func=drive_with_line,
                    stop_condition=obstacle_ahead,
                    name="LineFollow"
                ),
                EvalAction(clear_roi, "ClearROI"),
                EvalAction(lambda d: "   Остановлен.", "Finish"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(LineFollowWithROI())


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

Запуск:

```
python3 ~/ws/rbt/scripts/line_follow_roi.py
```

На стриме будет видна жёлтая рамка ROI.

2.4. Наблюдение

Когда препятствие далеко — ROI не задан (поиск по всему кадру). Когда лидар видит стену ближе 40 см — ROI сужается до нижних 2/3 кадра, где видна ближайшая часть линии.

3. Уровень «Углублённый»: Лидар → пиксели и параллакс

(Что под капотом)

Геометрия преобразования.

3.1. Преобразование угла лидара в пиксельные координаты

Грубая формула:

```
px = (угол_лидара_рад / угол_обзора_камеры_рад) * ширина_кадра + ширина_кадра/2
```

Для лидара 0° (вперёд) → центр кадра. Для -15° (вправо) → правая половина.

Но это без учёта **параллакса** — смещения между лидаром и камерой.

3.2. Параллакс и калибровка

Лидар стоит сверху, камера — спереди. Один и тот же объект будет под разными углами для них. Для близких объектов ошибка больше.

Решение: один раз измерить смещение (линейкой) и вычесть его при расчёте ROI. Для учебных целей можно пренебречь.

4. Уровень «Исследовательский»: Авто-ROI и ускорение

(Прорыв за границы)

Эксперименты.

4.1. Несколько ROI по секторам лидара

`get_lidar_sectors()` возвращает расстояния по 4 направлениям. Можно создать 4 маленьких ROI, соответствующих этим направлениям, и запустить поиск объектов в каждом параллельно (разные `ImageProcessorNode`).

4.2. Мини-исследование: ускорение от ROI

Вопрос: Во сколько раз ускоряется обработка при использовании ROI?

Методика:

1. Замерьте FPS без ROI.

2. Замерьте FPS с ROI размером 1/3 кадра.
3. Сравните.

Ожидание: ускорение ≈ 3× (пропорционально уменьшению площади).

5. Выводы по занятию

1. **ROI** ограничивает область поиска и ускоряет обработку.
2. **Связка лидар → ROI:** лидар говорит «где препятствие», камера ищет линию именно там.
3. **Параллакс** — смещение из-за разного положения сенсоров. Для близких объектов критично.

Связь со следующим занятием: Мы научились искать линию и объекты. Теперь оценим **движение** самой камеры — через оптический поток.

Домашнее задание:

- **Базовое**: Запустите миссию с ROI. Посмотрите стрим.
- **Продвинутое**: Добавьте ROI не только по фронту, но и по правому сектору.
- **Исследовательское**: Замерьте ускорение FPS с ROI и без.

Занятие 53: Оптический поток (Лукас-Канаде)

Содержание занятия: Отслеживание движения камеры по характерным точкам. Сравнение с одометрией

Цели занятия

- **Образовательная**: Написать `OpticalFlowProcessor` для вычисления смещения камеры методом Лукаса-Канаде с накоплением и сравнением с одометрией.
- **Развивающая**: Понять принцип оптического потока и как вычислить масштабный коэффициент «пиксели → метры».
- **Воспитательная**: Показать, что движение робота можно измерять не только энкодерами, но и камерой.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: OpticalFlowProcessor (45 мин)**
    - Инициализация точек и отслеживание.
    - Накопление смещения (`_flow_cum_x` , `_flow_cum_y`).
    - Накопление одометрии.
    - Масштабный коэффициент px → метры.
    - Миссия с крабовым движением и HTTP-стримом.

3. **Уровень «Углублённый»: Как работает Лукас-Канаде (25 мин)**
   • Постоянство яркости, пирамида, потеря точек.
   • Периодическая переинициализация каждые 30 кадров.
4. **Уровень «Исследовательский»: Точность на разных поверхностях (10 мин)**
   • Сравнение с одометрией.
   • Влияние текстуры пола.
5. **Подведение итогов (5 мин)**

1. Введение: Камера как датчик движения

Энкодеры меряют обороты колёс. Но колёса проскальзывают. Камера видит **реальное** смещение относительно пола. Если взять характерные точки на первом кадре и отследить их на следующем, можно вычислить, куда и насколько сместился робот.

Сегодня мы реализуем оптический поток методом Лукаса-Канаде и **сравним его с одометрией** — получим масштабный коэффициент «пиксели → метры».

2. Уровень «База»: OpticalFlowProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем трекер потока с накоплением.

2.1. Алгоритм с накоплением

```
Первый кадр:
  → gray → goodFeaturesToTrack → массив точек p0

Последующие кадры:
  → gray → calcOpticalFlowPyrLK(prev_gray, gray, p0) → p1
  → среднее (p1 - p0) → dx, dy
  → НАКАПЛИВАЕМ: flow_cum_x += dx, flow_cum_y += dy
  → НАКАПЛИВАЕМ одометрию: odom_cum_x, odom_cum_y
  → Масштаб: px_to_m = |odom_cum / flow_cum|
  → p0 = p1, prev_gray = gray
  → Каждые 30 кадров — переинициализация точек
```

Зачем переинициализация: точки постепенно теряются (уходят из кадра, теряют качество). Периодический сброс поддерживает стабильное количество точек.

2.2. Код миссии OpticalFlowMission

Создайте файл:

```
nano ~/ws/rbt/scripts/optical_flow_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: оптический поток + крабовое движение + HTTP стрим."""
import sys
```

```
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, Parallel,
                        HeartbeatAction, StreamAction,
                        HttpStreamerNode, ImageProcessorNode)


class OpticalFlowProcessor(ImageProcessorNode):
    """Оптический поток методом Лукаса-Канаде + сравнение с одометрией."""

    def __init__(self, driver, fps=30):
        super().__init__(driver, fps=fps)
        self._prev_gray = None
        self._points = None
        self._lk_params = dict(
            winSize=(21, 21),
            maxLevel=3,
            criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
30, 0.01)
        )
        self._feature_params = dict(
            maxCorners=100,
            qualityLevel=0.3,
            minDistance=15,
            blockSize=7
        )
        self._prev_odom_x = 0.0
        self._prev_odom_y = 0.0
        self._odom_initialized = False
        self._flow_cum_x = 0.0
        self._flow_cum_y = 0.0
        self._odom_cum_x = 0.0
        self._odom_cum_y = 0.0
        self._frame_count = 0

    def process_frame(self, frame):
        self._frame_count += 1
        status_msg = ""
        h, w = frame.shape[:2]
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        display = frame.copy()

        odom_x, odom_y, _ = self.driver.get_pose()
        if not self._odom_initialized:
            self._prev_odom_x = odom_x
            self._prev_odom_y = odom_y
            self._odom_initialized = True

        need_init = (self._prev_gray is None or
                     self._points is None or
                     len(self._points) < 10 or
                     self._frame_count % 30 == 0)

        if need_init:
            self._prev_gray = gray.copy()
```

```
            self._points = cv2.goodFeaturesToTrack(gray, mask=None,
**self._feature_params)
            if self._points is None:
                self._stream_display(display, "No features found")
                return
            self._points = self._points.reshape(-1, 2)
            self.driver.set_optical_flow(0.0, 0.0)
            for pt in self._points:
                cv2.circle(display, (int(pt[0]), int(pt[1])), 3, (0, 255,
0), -1)
            self._stream_display(display, f"Init: {len(self._points)}
points")
            return

        new_points, status, _ = cv2.calcOpticalFlowPyrLK(
            self._prev_gray, gray, self._points, None, **self._lk_params
        )

        if new_points is not None and len(new_points) > 0:
            good_new = new_points[status.flatten() == 1]
            good_old = self._points[status.flatten() == 1]

            if len(good_new) > 5:
                dx_px = np.mean(good_new[:, 0] - good_old[:, 0])
                dy_px = np.mean(good_new[:, 1] - good_old[:, 1])
                self.driver.set_optical_flow(dx_px, dy_px)
                self._flow_cum_x += dx_px
                self._flow_cum_y += dy_px

                for old, new in zip(good_old, good_new):
                    cv2.arrowedLine(display,
                                    (int(old[0]), int(old[1])),
                                    (int(new[0]), int(new[1])),
                                    (0, 255, 0), 1, tipLength=0.3)

                status_msg = f"Points: {len(good_new)}"
                self._points = good_new.reshape(-1, 2)
                self._prev_gray = gray.copy()
            else:
                status_msg = f"Low points: {len(good_new)}"
                self._points = None
                self._prev_gray = None
        else:
            status_msg = "Tracking lost"
            self._points = None
            self._prev_gray = None

        self._odom_cum_x += odom_x - self._prev_odom_x
        self._odom_cum_y += odom_y - self._prev_odom_y
        self._prev_odom_x = odom_x
        self._prev_odom_y = odom_y

        px_to_m_x = abs(self._odom_cum_x / self._flow_cum_x) if abs(self._
flow_cum_x) > 1 else 0
        px_to_m_y = abs(self._odom_cum_y / self._flow_cum_y) if abs(self._
flow_cum_y) > 1 else 0
```

```
        self.driver.bb.update(
            flow_cum_x=self._flow_cum_x,
            flow_cum_y=self._flow_cum_y,
            odom_cum_x=self._odom_cum_x,
            odom_cum_y=self._odom_cum_y,
            px_to_m_x=px_to_m_x,
            px_to_m_y=px_to_m_y,
        )

        self._stream_display(display, status_msg)

    def _stream_display(self, display, status=""):
        h, w = display.shape[:2]
        cv2.putText(display, f"Flow:  ({self._flow_cum_x:+.0f}, {self._flo
w_cum_y:+.0f}) px",
                    (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
2)
        cv2.putText(display, f"Odom:  ({self._odom_cum_x:+.3f}, {self._odo
m_cum_y:+.3f}) m",
                    (10, 50), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255,
0), 2)
        scale_x = self.driver.bb.get('px_to_m_x', 0)
        scale_y = self.driver.bb.get('px_to_m_y', 0)
        cv2.putText(display, f"Scale: x={scale_x:.4f} y={scale_y:.4f} m/
px",
                    (10, 70), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0,
255), 2)
        if status:
            cv2.putText(display, status, (10, h - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200, 200, 200), 1)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('flow_jpeg', jpeg.tobytes())


class OpticalFlowMission(RobotDriver):

    def __init__(self):
        super().__init__('optical_flow_mission', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='flow_jpeg',
port=8080))
        self.add_node(OpticalFlowProcessor(self, fps=15))

    def build_tree(self):
        def drive_sideways(d):
            return (0.0, -0.15, 0.0, 0.3)

        def check_flow(d):
            dx, dy, ready = d.get_optical_flow()
            x, y, _ = d.get_pose()
            return f"Поток: dx={dx:.1f} dy={dy:.1f} px | Одо: x={x:.2f}
y={y:.2f} м"

        return Sequence([
```

```
                EvalAction(lambda d: "   Оптический поток + крабовое
движение", "Start"),
                StreamAction(
                    command_func=drive_sideways,
                    stop_condition=lambda d: d.get_pose()[1] < -0.5,
                    name="Sideways"
                ),
                EvalAction(check_flow, "Result"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(OpticalFlowMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/optical_flow_mission.py
```

Тест: робот поедет боком на 0.5 м. На стриме будут видны зелёные стрелки, накопленный поток в пикселях, накопленная одометрия в метрах и масштабный коэффициент.

2.3. Масштабный коэффициент: пиксели → метры

```
px_to_m_x = abs(odom_cum_x / flow_cum_x)
```

Если робот проехал 0.5 м по одометрии, а поток накопил 200 пикселей, то `px_to_m = 0.5 / 200 = 0.0025 м/px`. Это значит, что 1 пиксель смещения ≈ 2.5 мм на полу.

Этот коэффициент зависит от высоты камеры. Его можно использовать для пересчёта оптического потока в метры без одометрии.

3. Уровень «Углублённый»: Как работает Лукас-Канаде

(Что под капотом)

Три предположения и пирамида.

3.1. Три предположения метода

1. **Постоянство яркости:** точка на объекте не меняет яркость при смещении.
2. **Малые смещения:** между соседними кадрами точка смещается на несколько пикселей.
3. **Пространственная когерентность:** соседние точки движутся одинаково.

3.2. Пирамида и переинициализация

Пирамида (`maxLevel=3`) позволяет отслеживать смещения до 20–30 пикселей: кадр уменьшается, смещение ищется грубо на мелком изображении, затем уточняется.

Переинициализация каждые 30 кадров решает проблему потери точек: точки уходят из кадра, текстура меняется. Сброс поддерживает стабильное количество.

4. Уровень «Исследовательский»: Точность на разных поверхностях

(Прорыв за границы)

Эксперименты.

4.1. Сравнение с одометрией

На стриме видны две строки: `Flow` (пиксели) и `Odom` (метры). Если колёса не проскальзывают, они пропорциональны. Если проскальзывают — `Odom` будет завышать пройденный путь, а `Flow` покажет реальное смещение.

4.2. Мини-исследование: на какой поверхности поток точнее

Вопрос: На какой поверхности масштабный коэффициент `px_to_m` наиболее стабилен?

Методика:

1. Запустите крабовое движение на линолеуме, ковре, плитке.
2. Запишите `px_to_m_x` и `px_to_m_y` для каждой.
3. Сравните разброс.

Ожидание: на текстурированной поверхности (плитка, ковёр) поток стабильнее, чем на гладком линолеуме.

5. Выводы по занятию

1. **Оптический поток** отслеживает характерные точки между кадрами.
2. **Накопление** `flow_cum` и `odom_cum` позволяет вычислить масштабный коэффициент px → метры.
3. **Периодическая переинициализация** (каждые 30 кадров) поддерживает стабильность.
4. **Поток дополняет одометрию** — не зависит от проскальзывания колёс.

Связь со следующим занятием: Мы отслеживаем движение камеры. Теперь научимся детектировать **движущиеся объекты** на статичном фоне — через вычитание фона (MOG2).

Домашнее задание:

- **Базовое**: Запустите миссию. Посмотрите на масштабный коэффициент в стриме.
- **Продвинутое**: Измените период переинициализации с 30 на 60 кадров. Как изменилась стабильность?
- **Исследовательское**: Сравните `px_to_m` на разных покрытиях. Постройте график.

Занятие 54: Детектирование препятствий: BackgroundSubtractor MOG2

Содержание занятия: Выделение движущихся объектов на статичном фоне

Цели занятия

- **Образовательная**: Написать `MotionDetectorProcessor` (наследник `ImageProcessorNode`) на основе `BackgroundSubtractorMOG2` для детектирования движущихся объектов.
- **Развивающая**: Понять, как MOG2 моделирует фон смесью гауссиан и почему статические объекты не детектируются.
- **Воспитательная**: Показать, что камера и лидар дополняют друг друга: одна видит движение, другой — геометрию.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: MotionDetectorProcessor (45 мин)**
   - `BackgroundSubtractorMOG2`.
   - Накопление фона (первые ~20 кадров).
   - Выделение переднего плана, поиск контуров, bounding box.
   - Миссия: робот стоит, в кадре проходит человек.
3. **Уровень «Углублённый»: MOG2 и статика (25 мин)**
   - Смесь гауссиан для каждого пикселя.
   - Адаптация к освещению.
   - Почему статические препятствия не видны.
4. **Уровень «Исследовательский»: Камера vs лидар (10 мин)**
   - Когда камера замечает раньше.
   - Комбинированное детектирование.
5. **Подведение итогов (5 мин)**

1. Введение: Движение — ключ к обнаружению

Лидар видит все препятствия, но не отличает движущиеся объекты от статических. Камера, наоборот, может выделить **движение** на статичном фоне.

MOG2 (Mixture of Gaussians) — алгоритм, который строит модель фона для каждого пикселя. Всё, что не вписывается в модель — передний план (движущийся объект).

2. Уровень «База»: MotionDetectorProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем детектор движения.

2.1. Алгоритм

```
Кадр → BackgroundSubtractorMOG2.apply(frame) → маска переднего плана
  → morphologyEx(MORPH_OPEN) → очистка шума
  → findContours → boundingRect для каждого контура
  → фильтр по площади (> 500)
  → запись bounding boxes в Blackboard (motion_boxes)
  → отрисовка прямоугольников на кадре
```

Накопление фона: первые ~20 кадров MOG2 «учится» — строит модель фона. В это время движущиеся объекты могут оставлять «призраков». После обучения фон стабилен.

2.2. Код миссии MotionDetectorMission

Создайте файл:

```
nano ~/ws/rbt/scripts/motion_detector_mission.py
```

```python
#!/usr/bin/env python3
"""Миссия: детектирование движения + HTTP стрим."""
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, Parallel,
                        HeartbeatAction, HttpStreamerNode,
                        ImageProcessorNode)


class MotionDetectorProcessor(ImageProcessorNode):
    """Детектор движения на основе MOG2."""

    def __init__(self, driver, fps=30):
        super().__init__(driver, fps=fps)
        self._bg_subtractor = cv2.createBackgroundSubtractorMOG2(
            history=500, varThreshold=36, detectShadows=False
        )
        self._frame_count = 0
```

```
    def process_frame(self, frame):
        self._frame_count += 1
        h, w = frame.shape[:2]

        # Применяем вычитание фона
        fg_mask = self._bg_subtractor.apply(frame)

        # Морфологическая очистка
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
        fg_mask = cv2.morphologyEx(fg_mask, cv2.MORPH_OPEN, kernel)
        fg_mask = cv2.morphologyEx(fg_mask, cv2.MORPH_CLOSE, kernel)

        # Поиск контуров движущихся объектов
        contours, _ = cv2.findContours(fg_mask, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        display = frame.copy()
        boxes = []

        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area < 500:
                continue
            x, y, bw, bh = cv2.boundingRect(cnt)
            boxes.append((x, y, bw, bh, area))
            cv2.rectangle(display, (x, y), (x+bw, y+bh), (0, 0, 255), 2)
            cv2.putText(display, f"MOTION {area:.0f}", (x, y-5),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)

        self.driver.bb.update(
            motion_detected=len(boxes) > 0,
            motion_boxes=boxes,
            motion_frame_count=self._frame_count,
        )

        status = f"Frames: {self._frame_count}"
        if len(boxes) > 0:
            status += f" | Motion: {len(boxes)} objects"
        cv2.putText(display, status, (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('motion_jpeg', jpeg.tobytes())


class MotionDetectorMission(RobotDriver):

    def __init__(self):
        super().__init__('motion_detector_mission', tick_rate=2)
        self.add_node(HttpStreamerNode(self.bb, key='motion_jpeg', port=80
80))
        self.add_node(MotionDetectorProcessor(self, fps=30))

    def build_tree(self):
```

```
        def check_motion(d):
            if d.bb.get('motion_detected', False):
                boxes = d.bb.get('motion_boxes', [])
                return f"   Движение обнаружено! Объектов: {len(boxes)}"
            return "   Движения нет ... "

        return Sequence([
                EvalAction(lambda d: "   Детектор движения + стрим :8080/
stream", "Start"),
                EvalAction(check_motion, "CheckMotion"),
            ])


def main():
    rclpy.init()
    RobotDriver.spin(MotionDetectorMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/motion_detector_mission.py
```

Тест: робот стоит. Пройдите перед камерой. На стриме появятся красные прямоугольники вокруг движущихся объектов.

2.3. Наблюдение

- **Первые секунды:** MOG2 накапливает фон. Возможны ложные срабатывания.
- **После 20–30 кадров:** фон стабилизируется. Движущиеся объекты чётко выделяются.
- **Если объект замер:** через несколько секунд он «врастает» в фон и перестаёт детектироваться.

3. Уровень «Углублённый»: MOG2 и статика

(Что под капотом)

Как работает модель фона.

3.1. Смесь гауссиан для каждого пикселя

MOG2 моделирует каждый пиксель **несколькими** гауссианами (обычно 3–5). Например, пиксель на полу может иметь две моды: «светло-серый когда никого нет» и «тёмный когда наступили». Алгоритм сам определяет, сколько мод нужно для каждого пикселя.

Параметры:

- `history=500` — сколько кадров помнить.
- `varThreshold=36` — порог чувствительности (меньше → больше ложных срабатываний).

- `detectShadows=False` — не выделять тени (они серые, не нужны).

3.2. Почему статические препятствия не детектируются

MOG2 детектирует **изменения** относительно фона. Если коробка стояла до запуска — она стала частью фона. Если коробку поставили после запуска — она будет детектироваться как движение, но через несколько секунд «врастёт» в фон.

Решение для статики: использовать **лидар**. Он видит геометрию независимо от движения. Камера (MOG2) — для движущихся, лидар — для статических.

4. Уровень «Исследовательский»: Камера vs лидар

(Прорыв за границы)

Когда какой сенсор лучше.

4.1. Когда камера замечает раньше лидара

- **Боковое зрение:** лидар смотрит вперёд узким лучом. Камера видит широкий угол (70°). Человек, подходящий сбоку, попадёт в кадр раньше, чем в луч лидара.
- **Прозрачные объекты:** стекло, решётка — лидар не видит, камера — да.
- **Дальние объекты:** разрешение камеры позволяет заметить маленький объект на большом расстоянии.

4.2. Мини-исследование: совместное детектирование

Идея: объединить `MotionDetectorProcessor` и данные лидара. Если MOG2 нашёл движение **и** лидар показывает `< 1.0 м` в том же направлении — это точно препятствие. Запишите флаг `confirmed_obstacle` в `Blackboard`.

5. Выводы по занятию

1. **MOG2** моделирует фон смесью гауссиан и выделяет передний план.
2. **Первые 20 кадров** — обучение фона, возможны ложные срабатывания.
3. **Статические объекты не детектируются** — для них нужен лидар.
4. **Камера и лидар дополняют друг друга** — движение vs геометрия.

Связь со следующим занятием: Мы детектируем движение. Теперь научимся распознавать **ArUco-маркеры** — специальные метки для точного позиционирования.

Домашнее задание:

- **Базовое**: Запустите миссию. Пройдите перед камерой.
- **Продвинутое**: Подберите `varThreshold` (10, 36, 100). Как меняется чувствительность?

- **Исследовательское**: Сравните время обнаружения человека камерой и лидаром. Кто первый?

Занятие 55: ArUco-маркеры

Содержание занятия: Распознавание маркеров дополненной реальности для точного позиционирования

Цели занятия

- **Образовательная**: Написать `ArucoDetectorProcessor` (наследник `ImageProcessorNode`) для распознавания ArUco-маркеров и определения их позы.
- **Развивающая**: Понять, как устроен маркер, как работает PnP-алгоритм и почему важна калибровка камеры.
- **Воспитательная**: Показать, что маркеры — это «QR-коды для роботов»: дешёвое и точное средство навигации.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: ArucoDetectorProcessor (45 мин)**
   - `cv2.aruco.detectMarkers()`.
   - `cv2.aruco.estimatePoseSingleMarkers()`.
   - Сохранение через `driver.set_marker_info()`.
   - Миссия: робот стоит перед маркером, вывод id, расстояния, угла.
3. **Уровень «Углублённый»: Устройство маркера и PnP (25 мин)**
   - Матрица, коды коррекции.
   - Алгоритм PnP.
   - Словари маркеров.
4. **Уровень «Исследовательский»: Карта маркеров и точность (10 мин)**
   - Навигация по маркерам.
   - Зависимость точности от расстояния.
5. **Подведение итогов (5 мин)**

1. Введение: QR-коды для роботов

ArUco-маркер — это квадратная чёрно-белая матрица. Камера робота видит маркер и определяет:

- **id** маркера (какой именно знак).
- **Расстояние** до него (в метрах).
- **Угол** поворота маркера относительно камеры.

Это позволяет создавать **навигационные знаки** на полигоне.

2. Уровень «База»: ArucoDetectorProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем детектор маркеров.

2.1. Алгоритм

```
Кадр → cv2.aruco.ArucoDetector.detectMarkers() → (corners, ids)
  → если ids не пусто:
      → estimatePoseSingleMarkers → tvecs, rvecs
      → poses = {id: {'distance': ..., 'angle': ...}}
      → driver.set_marker_info(ids, poses)
  → отрисовка: drawDetectedMarkers
```

Все результаты сохраняются через `driver.set_marker_info()` — не нужно дублировать в `bb.update`.

2.2. Код миссии ArucoDetectorMission

Создайте файл:

```
nano ~/ws/rbt/scripts/aruco_detector_mission.py
```

```python
#!/usr/bin/env python3
"""Миссия: детектирование ArUco-маркеров + HTTP стрим."""
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        HttpStreamerNode, ImageProcessorNode)


class ArucoDetectorProcessor(ImageProcessorNode):
    """Детектор ArUco-маркеров с оценкой позы."""

    def __init__(self, driver, fps=30,
                 marker_size=0.1,
                 dictionary=cv2.aruco.DICT_4X4_50):
        super().__init__(driver, fps=fps)
        self._marker_size = marker_size
        self._aruco_dict = cv2.aruco.getPredefinedDictionary(dictionary)
        self._aruco_params = cv2.aruco.DetectorParameters()
        self._aruco_detector = cv2.aruco.ArucoDetector(
            self._aruco_dict, self._aruco_params
        )
        self._K = np.array([[600, 0, 320], [0, 600, 240], [0, 0, 1]],
                           dtype=np.float32)
        self._D = np.zeros((1, 5), dtype=np.float32)

    def process_frame(self, frame):
        display = frame.copy()
        corners, ids, _ = self._aruco_detector.detectMarkers(frame)
```

```
        if ids is not None and len(ids) > 0:
            rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
                corners, self._marker_size, self._K, self._D
            )

            poses = {}
            for i, marker_id in enumerate(ids.flatten()):
                tvec = tvecs[i][0]
                rvec = rvecs[i][0]
                poses[int(marker_id)] = {
                    'distance': float(np.linalg.norm(tvec)),
                    'angle': float(np.linalg.norm(rvec)),
                }

            # Единый метод сохранения
            self.driver.set_marker_info(ids.flatten().tolist(), poses)

            cv2.aruco.drawDetectedMarkers(display, corners, ids)
            for i, marker_id in enumerate(ids.flatten()):
                dist = poses[int(marker_id)]['distance']
                c = corners[i][0]
                cx, cy = int(c[:, 0].mean()), int(c[:, 1].mean())
                cv2.putText(display, f"ID:{marker_id} {dist:.2f}m",
                            (cx, cy-10), cv2.FONT_HERSHEY_SIMPLEX,
                            0.5, (0, 255, 255), 2)
        else:
            self.driver.set_marker_info([], {})

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('aruco_jpeg', jpeg.tobytes())


class ArucoDetectorMission(RobotDriver):

    def __init__(self):
        super().__init__('aruco_detector_mission', tick_rate=2)
        self.add_node(HttpStreamerNode(self.bb, key='aruco_jpeg', port=808
0))
        self.add_node(ArucoDetectorProcessor(self, fps=30,
marker_size=0.1))

    def build_tree(self):
        def check_markers(d):
            ids, poses, detected = d.get_marker_info()
            if detected:
                info = ", ".join(
                    f"ID:{mid} {poses[mid]['distance']:.2f}м"
                    for mid in ids
                )
                return f"   Маркеры: {info}"
            return "   Ищу маркеры ... "

        return Sequence([
            EvalAction(lambda d: "   Детектор ArUco + стрим :8080/
```

```
stream", "Start"),
            EvalAction(check_markers, "CheckMarkers"),
        ])


def main():
    rclpy.init()
    RobotDriver.spin(ArucoDetectorMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/aruco_detector_mission.py
```

Тест: распечатайте ArUco-маркер 4×4 (id=0), положите перед роботом.

2.3. Методы драйвера для маркеров

`driver.set_marker_info(ids, poses)` сохраняет всё в `Blackboard`. Для чтения:

- `driver.get_marker_info()` → `(ids, poses, detected)`.
- `driver.get_marker_id()` → первый найденный id (или `None`).

Не нужно дублировать данные через `bb.update` — `set_marker_info` уже делает это.

3. Уровень «Углублённый»: Устройство маркера и PnP

(Что под капотом)

Как маркер устроен и как определяется поза.

3.1. Как устроен ArUco-маркер

Маркер — квадрат из чёрных и белых клеток (4×4, 5×5, 6×6). Внешняя рамка всегда чёрная. Внутренние клетки кодируют id. Часть битов — коррекция ошибок (коды Рида-Соломона).

Словари:

Словарь	Размер	Маркеров
`DICT_4X4_50`	4×4	50
`DICT_5X5_100`	5×5	100
`DICT_6X6_250`	6×6	250

3.2. Алгоритм PnP (Perspective-n-Point)

По четырём углам маркера на изображении (2D) и известному размеру (3D) вычисляется положение маркера в пространстве:

- `tvec` — вектор смещения (x, y, z).
- `rvec` — вектор вращения.

Требования: маркер должен быть плоским (не мятым).

4. Уровень «Исследовательский»: Карта маркеров и точность

(Прорыв за границы)

Эксперименты.

4.1. Карта полигона из маркеров

Разместите несколько маркеров с известными координатами. Сохраните карту в `Blackboard`:

```
d.bb.set('marker_map', {0: (0.0, 0.0), 1: (1.0, 0.0), 2: (0.0, 1.0)})
```

Когда робот видит маркер, он определяет своё положение по карте.

4.2. Мини-исследование: точность от расстояния

Вопрос: Как расстояние до маркера влияет на точность?

Методика: измерьте расстояние на 0.3, 0.5, 1.0, 1.5, 2.0 м. Вычислите ошибку.

Ожидание: ошибка растёт с расстоянием (меньше пикселей на маркер).

5. Выводы по занятию

1. **ArUco-маркеры** — пассивные навигационные знаки.
2. **`detectMarkers`** находит, **`estimatePoseSingleMarkers`** вычисляет положение.
3. **PnP** по четырём углам определяет позу маркера.
4. **`set_marker_info`** — единый метод сохранения, не нужно дублировать.

Связь со следующим занятием: Объединяем линию и маркеры — **навигация по перекрёсткам**.

Домашнее задание:

- **Базовое**: Запустите миссию с маркером.
- **Продвинутое**: Подставьте реальную матрицу камеры.
- **Исследовательское**: Измерьте точность на разных расстояниях.

Занятие 56: Объединение линии и ArUco-маркеров

Содержание занятия: Навигация по полигону: следование по линии с остановкой на маркерах и выполнением команд

Цели занятия

- **Образовательная**: Собрать миссию, объединяющую `LineDetectorProcessor` и `ArucoDetectorProcessor`: робот следует по линии, при обнаружении маркера выполняет заданное действие.
- **Развивающая**: Понять, как организовать приоритеты: линия — основное поведение, маркер — прерывание.
- **Воспитательная**: Показать, что реальная навигация — это композиция простых поведений.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия LineWithMarkers (50 мин)**
   - Два процессора: `LineDetectorProcessor` + `ArucoDetectorProcessor`.
   - `RepeatUntil` + `StreamAction` для следования по линии.
   - `stop_condition` по маркеру.
   - Словарь действий: `{id: (dx, dy, dyaw)}`.
   - Выполнение `MoveRelativeAction` по маркеру, возврат на линию.
3. **Уровень «Углублённый»: Приоритеты и отказоустойчивость (20 мин)**
   - Линия или маркер — что важнее?
   - Что если маркер не читается — `RetryAction`.
   - Возврат на линию после манёвра.
4. **Уровень «Исследовательский»: Маркеры на скорости (10 мин)**
   - Надёжность распознавания при движении.
   - Программируемая логистика.
5. **Подведение итогов (5 мин)**

1. Введение: Дорожные знаки для робота

Линия — это дорога. Маркеры — это **дорожные знаки** на перекрёстках. Робот едет по линии, видит маркер, считывает его id и выполняет команду: «поверни направо», «остановись», «сдай назад».

Сегодня мы объединим два процессора в одной миссии.

2. Уровень «База»: Миссия LineWithMarkers

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем миссию с двумя процессорами.

2.1. Архитектура

```
RepeatUntil (stop_condition = миссия завершена)
└── Sequence
    ├── StreamAction (следование по линии, стоп по маркеру)
    └── Sequence (манёвр по маркеру)
        ├── EvalAction (вывод id маркера)
        └── MoveRelativeAction (действие из словаря)
```

Два процессора работают параллельно в фоне:

- `LineDetectorProcessor` — обновляет `driver.bb['line_offset']`.
- `ArucoDetectorProcessor` — обновляет `driver.bb['marker_ids']`.

Дерево поведения читает эти данные и принимает решения.

2.2. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/line_with_markers.py
```

```
#!/usr/bin/env python3
"""Миссия: следование по линии + действия по ArUco-маркерам + стрим."""
import math
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        StreamAction, MoveRelativeAction,
                        RepeatUntil, LogAction,
                        HttpStreamerNode, ImageProcessorNode)


# ═══ Процессор линии (копия из занятия 50) ═══

class LineDetectorProcessor(ImageProcessorNode):
    def process_frame(self, frame):
        h, w = frame.shape[:2]
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)
        display = frame.copy()
        if contours:
            best = max(contours, key=cv2.contourArea)
            if cv2.contourArea(best) > 500:
                M = cv2.moments(best)
```

```
                if M['m00'] > 0:
                    cx = int(M['m10'] / M['m00'])
                    offset = cx - w / 2
                    self.driver.set_line_info(offset, 0.0, detected=True)
                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2)
                    cv2.circle(display, (cx, int(M['m01']/M['m00'])), 5,
(0, 0, 255), -1)
                    return self._send(display)
        self.driver.set_line_info(0.0, 0.0, detected=False)
        self._send(display)

    def _send(self, display):
        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('line_jpeg', jpeg.tobytes())


# ═══ Процессор маркеров (копия из занятия 55) ═══

class ArucoDetectorProcessor(ImageProcessorNode):
    def __init__(self, driver, fps=30, marker_size=0.1):
        super().__init__(driver, fps=fps)
        self._marker_size = marker_size
        self._aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DIC
T_4X4_50)
        self._aruco_detector = cv2.aruco.ArucoDetector(
            self._aruco_dict, cv2.aruco.DetectorParameters())
        self._K = np.array([[600, 0, 320], [0, 600, 240], [0, 0, 1]], dtyp
e=np.float32)
        self._D = np.zeros((1, 5), dtype=np.float32)

    def process_frame(self, frame):
        display = frame.copy()
        corners, ids, _ = self._aruco_detector.detectMarkers(frame)
        if ids is not None and len(ids) > 0:
            rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
                corners, self._marker_size, self._K, self._D)
            poses = {}
            for i, mid in enumerate(ids.flatten()):
                poses[int(mid)] = {'distance':
float(np.linalg.norm(tvecs[i][0]))}
            self.driver.set_marker_info(ids.flatten().tolist(), poses)
            cv2.aruco.drawDetectedMarkers(display, corners, ids)
        else:
            self.driver.set_marker_info([], {})
        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('aruco_jpeg', jpeg.tobytes())


# ═══ Миссия ═══

class LineWithMarkers(RobotDriver):
    def __init__(self):
        super().__init__('line_with_markers', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='line_jpeg',
```

```
port=8080))
        self.add_node(HttpStreamerNode(self.bb, key='aruco_jpeg', port=808
1))
        self.add_node(LineDetectorProcessor(self, fps=30))
        self.add_node(ArucoDetectorProcessor(self, fps=30,
marker_size=0.1))
        # Словарь действий: id маркера → (dx, dy, dyaw)
        self.bb.set('marker_actions', {
            0: (0.0, 0.0, 90.0),   # маркер 0 → поворот налево 90°
            1: (0.0, 0.0, -90.0),  # маркер 1 → поворот направо 90°
            2: (0.5, 0.0, 0.0),    # маркер 2 → вперёд 0.5 м
            3: (0.0, 0.0, 0.0),    # маркер 3 → стоп (завершение)
        })

    def build_tree(self):
        def drive_with_line(d):
            offset, angle, detected = d.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            return (vx, 0.0, omega, 0.3)

        def marker_detected(d):
            return d.get_marker_id() is not None

        def do_marker_action(d):
            mid = d.get_marker_id()
            actions = d.bb.get('marker_actions', {})
            if mid in actions:
                dx, dy, dyaw = actions[mid]
                d.bb.update(action_dx=dx, action_dy=dy, action_dyaw=dyaw)
                return f"Маркер {mid}: dx={dx}, dy={dy}, dyaw={dyaw}°"
            return f"Маркер {mid}: нет действия"

        return RepeatUntil(
            child=Sequence([
                StreamAction(
                    command_func=drive_with_line,
                    stop_condition=marker_detected,
                    name="FollowLine"
                ),
                EvalAction(do_marker_action, "MarkerAction"),
                MoveRelativeAction(
                    dx=lambda d: d.bb.get('action_dx', 0),
                    dy=lambda d: d.bb.get('action_dy', 0),
                    dyaw_deg=lambda d: d.bb.get('action_dyaw', 0),
                    name="ExecuteAction"
                ),
                LogAction("   Действие выполнено, продолжаю по линии"),
            ]),
            stop_condition=lambda d: d.get_marker_id() == 3,
            name="MainLoop"
        )


def main():
```

```
    rclpy.init()
    RobotDriver.spin(LineWithMarkers())


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

Запуск:

```
python3 ~/ws/rbt/scripts/line_with_markers.py
```

Два стрима: линия на порту 8080, маркеры на порту 8081.

2.3. Как это работает

1. `StreamAction` едет по линии, пока `get_marker_id()` не станет `not None`.
2. При обнаружении маркера `EvalAction` читает его id, ищет действие в словаре `marker_actions` и сохраняет `(dx, dy, dyaw)` в `Blackboard`.
3. `MoveRelativeAction` выполняет действие (поворот, движение).
4. `RepeatUntil` возвращает робота к следованию по линии.
5. Если маркер id=3 — миссия завершается.

2.4. Динамические параметры в MoveRelativeAction

`MoveRelativeAction` поддерживает **вызываемые** (callable) параметры:

```
dx=lambda d: d.bb.get('action_dx', 0)
```

Лямбда вычисляется в момент `tick`, а не в момент создания дерева. Это позволяет менять действие динамически — по данным маркера.

3. Уровень «Углублённый»: Приоритеты и отказоустойчивость

(Что под капотом)

Тонкости навигации.

3.1. Линия или маркер — что важнее?

В текущей реализации маркер имеет **приоритет**: как только он обнаружен, `stop_condition` срабатывает, и робот останавливается для выполнения действия. Это правильно для перекрёстков — не нужно проезжать маркер на скорости.

Но если маркер находится **на** линии (не на перекрёстке), робот может остановиться посреди пути. Решение: проверять **расстояние** до маркера (`poses[mid]['distance']`) и реагировать только если маркер близко.

3.2. Отказоустойчивость: RetryAction

Если маркер обнаружен, но **не распознан** (id не в словаре), можно обернуть `MoveRelativeAction` в `RetryAction`:

```
RetryAction(MoveRelativeAction( ... ), max_attempts=2)
```

Или добавить `Condition` с проверкой `mid in actions` перед `MoveRelativeAction`.

4. Уровень «Исследовательский»: Маркеры на скорости

(Прорыв за границы)

Эксперименты.

4.1. Надёжность распознавания при движении

Вопрос: На какой скорости робот перестаёт надёжно распознавать маркеры?

Методика: увеличивайте `vx` в `drive_with_line` с 0.1 до 0.3 м/с. Записывайте, сколько маркеров из 5 распознано.

Ожидание: при 0.3 м/с камера может смазывать кадр → маркер не распознаётся.

4.2. Программируемая логистика

Идея: маркеры задают пункты назначения. Словарь `marker_actions` можно загружать из JSON-файла:

```
import json
with open('route.json') as f:
    self.bb.set('marker_actions', json.load(f))
```

Это позволяет менять маршрут без переписывания кода.

5. Выводы по занятию

1. **Два процессора** работают параллельно: линия и маркеры.
2. **`RepeatUntil`** зацикливает следование по линии с остановками на маркерах.
3. **Словарь `marker_actions`** связывает id маркера с действием.
4. **Динамические параметры** `MoveRelativeAction` позволяют менять действие по данным маркера.

Связь со следующим занятием: Мы запускаем детекторы как отдельные ноды. На следующем занятии объединим их в **один `VisionAction`-узел** для удобства.

Домашнее задание:

- **Базовое**: Запустите миссию с маркером на линии.

- **Продвинутое**: Добавьте проверку расстояния до маркера (реагировать только если < 0.3 м).
- **Исследовательское**: Проверьте надёжность распознавания на разных скоростях.

Занятие 57: Интеграция детекторов в единый Action-узел

Содержание занятия: Объединение всех детекторов в одном `VisionProcessor`. Параллельная обработка в дереве

Цели занятия

- **Образовательная**: Написать `VisionProcessor` — единый `ImageProcessorNode`, который за один вызов `process_frame` запускает линию, ArUco и трекинг цвета.
- **Развивающая**: Понять, почему детекторы запускаются последовательно (GIL), и как измерить время каждого.
- **Воспитательная**: Показать, что интеграция упрощает архитектуру: один процессор вместо трёх.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: VisionProcessor (45 мин)**
   - Единый `process_frame`: линия + ArUco + цвет.
   - Один HTTP-стрим со всеми отрисовками.
   - Миссия с `StreamAction` и `EvalAction`.
3. **Уровень «Углублённый»: GIL и замеры времени (25 мин)**
   - Почему детекторы последовательно.
   - Замер времени каждого детектора.
4. **Уровень «Исследовательский»: Динамическое включение (10 мин)**
   - Флаги `use_line`, `use_aruco` в Blackboard.
   - Снижение FPS от числа детекторов.
5. **Подведение итогов (5 мин)**

1. Введение: Один процессор — три детектора

В занятии 56 мы запускали `LineDetectorProcessor` и `ArucoDetectorProcessor` как **отдельные ноды**. Это работало, но:

- Каждый процессор имел свой таймер и свой темп обработки.
- Один и тот же кадр обрабатывался дважды (конвертация в серый, бинаризация).

Сегодня мы объединим всё в **один** `VisionProcessor`, который за один кадр делает всё.

2. Уровень «База»: VisionProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем единый процессор.

2.1. Архитектура

```
VisionProcessor.process_frame(frame):
  1. gray = cvtColor(frame)
  2. binary = adaptiveThreshold(gray)  ← один раз!
  3. contours = findContours(binary)
  4. line = detect_line(contours)      → set_line_info()
  5. markers = detect_aruco(frame)     → set_marker_info()
  6. object = detect_color(hsv)        → set_object_position()
  7. отрисовка всего на display
  8. JPEG в Blackboard
```

Оптимизация: серая картинка и бинарное изображение считаются **один раз** и используются всеми детекторами.

2.2. Код миссии VisionMission

Создайте файл:

```
nano ~/ws/rbt/scripts/vision_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: единый VisionProcessor (линия + ArUco + цвет) + стрим."""
import math, time
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        StreamAction, HttpStreamerNode,
                        ImageProcessorNode)


class VisionProcessor(ImageProcessorNode):
    """Единый процессор: линия + ArUco + цвет."""

    def __init__(self, driver, fps=30):
        super().__init__(driver, fps=fps)
        # ArUco
        self._aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DIC
T_4X4_50)
        self._aruco_detector = cv2.aruco.ArucoDetector(
            self._aruco_dict, cv2.aruco.DetectorParameters())
        self._K = np.array([[600, 0, 320], [0, 600, 240], [0, 0, 1]], dtyp
e=np.float32)
        self._D = np.zeros((1, 5), dtype=np.float32)
        # Цвет (красный)
        self._lower_hsv = (0, 100, 100)
```

```
        self._upper_hsv = (10, 255, 255)
        # Замеры времени
        self._timings = {}

    def process_frame(self, frame):
        h, w = frame.shape[:2]
        t_start = time.time()
        display = frame.copy()

        # ═══ Общая предобработка (один раз!) ═══
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        # ═══ 1. Линия ═══
        t0 = time.time()
        if contours:
            best = max(contours, key=cv2.contourArea)
            if cv2.contourArea(best) > 500:
                M = cv2.moments(best)
                if M['m00'] > 0:
                    cx = int(M['m10'] / M['m00'])
                    cy = int(M['m01'] / M['m00'])
                    offset = cx - w / 2
                    topmost = tuple(best[best[:, :, 1].argmin()][0])
                    angle = math.atan2(topmost[0] - cx, cy - topmost[1])
                    self.driver.set_line_info(offset, angle,
detected=True)
                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2)
                    cv2.circle(display, (cx, cy), 5, (0, 0, 255), -1)
                    cv2.putText(display, f"Line: off={offset:.0f}", (10, 3
0),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255,
0), 1)
                else:
                    self.driver.set_line_info(0.0, 0.0, detected=False)
            else:
                self.driver.set_line_info(0.0, 0.0, detected=False)
        else:
            self.driver.set_line_info(0.0, 0.0, detected=False)
        self._timings['line'] = (time.time() - t0) * 1000

        # ═══ 2. ArUco ═══
        t0 = time.time()
        corners, ids, _ = self._aruco_detector.detectMarkers(frame)
        if ids is not None and len(ids) > 0:
            poses = {}
            for i, mid in enumerate(ids.flatten()):
                poses[int(mid)] = {'distance': 0.0}
            self.driver.set_marker_info(ids.flatten().tolist(), poses)
            cv2.aruco.drawDetectedMarkers(display, corners, ids)
        else:
```

```
            self.driver.set_marker_info([], {})
        self._timings['aruco'] = (time.time() - t0) * 1000

        # ═══ 3. Цвет ═══
        t0 = time.time()
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, self._lower_hsv, self._upper_hsv)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        color_contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                             cv2.CHAIN_APPROX_SIMPLE)
        if color_contours:
            best_c = max(color_contours, key=cv2.contourArea)
            if cv2.contourArea(best_c) > 200:
                Mc = cv2.moments(best_c)
                if Mc['m00'] > 0:
                    obj_cx = int(Mc['m10'] / Mc['m00'])
                    obj_cy = int(Mc['m01'] / Mc['m00'])
                    self.driver.set_object_position(obj_cx, obj_cy, detect
ed=True)
                    cv2.circle(display, (obj_cx, obj_cy), 10, (255, 0,
0), 2)
                    cv2.putText(display, f"Obj: ({obj_cx},{obj_cy})",
                                (obj_cx+15, obj_cy), cv2.FONT_HERSHEY_SIMPL
EX,
                                0.5, (255, 0, 0), 1)
                else:
                    self.driver.set_object_position(0, 0, detected=False)
            else:
                self.driver.set_object_position(0, 0, detected=False)
        else:
            self.driver.set_object_position(0, 0, detected=False)
        self._timings['color'] = (time.time() - t0) * 1000

        # ═══ Вывод замеров ═══
        total_ms = (time.time() - t_start) * 1000
        y = 50
        for name, ms in self._timings.items():
            cv2.putText(display, f"{name}: {ms:.1f}ms", (10, y),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.4, (200, 200, 200), 1)
            y += 18
        cv2.putText(display, f"Total: {total_ms:.1f}ms", (10, y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 0), 1)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('vision_jpeg', jpeg.tobytes())


class VisionMission(RobotDriver):

    def __init__(self):
        super().__init__('vision_mission', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='vision_jpeg', port=80
80))
        self.add_node(VisionProcessor(self, fps=30))
```

```
    def build_tree(self):
        def drive_with_line(d):
            offset, angle, detected = d.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            return (vx, 0.0, omega, 0.3)

        def check_all(d):
            _, _, line_ok = d.get_line_position()
            _, _, markers_ok = d.get_marker_info()
            _, _, obj_ok = d.get_object_position()
            return (f"Линия: {'  ' if line_ok else '  '}, "
                    f"Маркеры: {'  ' if markers_ok else '  '}, "
                    f"Объект: {'  ' if obj_ok else '  '}")

        return Sequence([
            EvalAction(lambda d: "   VisionProcessor: линия + ArUco +
цвет", "Start"),
            StreamAction(
                command_func=drive_with_line,
                stop_condition=lambda d: d.get_marker_id() is not None,
                name="FollowLine"
            ),
            EvalAction(check_all, "Results"),
        ])


def main():
    rclpy.init()
    RobotDriver.spin(VisionMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/vision_mission.py
```

На стриме: линия (зелёный), маркеры (разноцветные рамки), цветной объект (синяя окружность) и замеры времени (мс) в левом верхнем углу.

2.3. Замеры времени

В левом верхнем углу стрима выводятся миллисекунды на каждый детектор:

```
line: 2.3ms
aruco: 8.1ms
color: 1.5ms
Total: 15.2ms
```

ArUco — самый тяжёлый (поиск маркеров). **Цвет — самый лёгкий** (простая пороговая маска). **Линия — средний**.

3. Уровень «Углублённый»: GIL и замеры

(Что под капотом)

Почему детекторы последовательно.

3.1. GIL в Python

GIL (Global Interpreter Lock) — блокировка, которая разрешает только одному потоку выполнять Python-код одновременно. Даже в `MultiThreadedExecutor` потоки не ускоряют **вычислительные** задачи (OpenCV), только **ожидание** (I/O, таймеры).

Поэтому детекторы в `process_frame` выполняются **последовательно** — это правильно. Попытка распараллелить их через `threading` не даст выигрыша из-за GIL.

3.2. Анализ замеров

Если `Total > 33ms` (30 FPS), процессор не успевает за камерой — кадры будут пропускаться. Решения:

- Уменьшить разрешение кадра (`cv2.resize`).
- Отключить самый тяжёлый детектор (ArUco).
- Уменьшить частоту процессора (`fps=15`).

4. Уровень «Исследовательский»: Динамическое включение

(Прорыв за границы)

Управление детекторами через Blackboard.

4.1. Флаги в Blackboard

Добавьте проверку флагов перед запуском детектора:

```
if self.driver.bb.get('use_line', True):
    # ... detect line ...
if self.driver.bb.get('use_aruco', True):
    # ... detect aruco ...
if self.driver.bb.get('use_color', True):
    # ... detect color ...
```

Флагами можно управлять из дерева поведения: `EvalAction` → `d.bb.set('use_aruco', False)` .

4.2. Мини-исследование: снижение FPS

Вопрос: На сколько падает FPS при добавлении каждого нового детектора?

Методика:

1. Запустите только линию. Запишите Total ms.
2. Добавьте ArUco. Запишите.
3. Добавьте цвет. Запишите.

Ожидание: время растёт линейно с числом детекторов.

5. Выводы по занятию

1. **VisionProcessor** объединяет линию, ArUco и цвет в одном `process_frame`.
2. **Общая предобработка** (серый, бинарный) экономит время.
3. **GIL** не позволяет ускорить детекторы через потоки — они и так последовательные.
4. **Замеры времени** показывают, какой детектор самый затратный.

Связь со следующим занятием: Мы освоили все детекторы. На следующем занятии — **зачёт по OpenCV**: комплексная миссия с линией, маркерами, цветом и оптическим потоком.

Домашнее задание:

- **Базовое**: Запустите `vision_mission.py`. Посмотрите замеры времени.
- **Продвинутое**: Добавьте флаги `use_line`, `use_aruco` в Blackboard и управляйте ими.
- **Исследовательское**: Измерьте Total ms для 1, 2 и 3 детекторов. Постройте график.

Занятие 58: Зачёт по OpenCV

Содержание занятия: Комплексная миссия — линия, ArUco-маркеры, цветной объект, оптический поток

Цели занятия

- **Образовательная**: Собрать миссию, объединяющую все детекторы: следование по линии, остановка на маркерах, трекинг объекта, оценка дрейфа.
- **Развивающая**: Научиться комбинировать детекторы в одном `VisionProcessor` и выбирать правильную архитектуру дерева.
- **Воспитательная**: Провести зачёт — демонстрация комплексной миссии с анализом ошибок.

План занятия (90 минут)

1. **Организационный момент и регламент зачёта (5 мин)**
2. **Уровень «База»: Комплексная миссия ExamMission (50 мин)**
   - `VisionProcessor`: линия + ArUco + цвет + оптический поток.
   - `RepeatUntil` + `StreamAction` + действия по маркерам.
   - Трекинг объекта и приближение.

- Оценка дрейфа.

3. **Уровень «Углублённый»: Анализ ошибок (20 мин)**
   - Почему выбран такой порядок детекторов.
   - Разбор типичных ошибок.
4. **Уровень «Исследовательский»: Тюнинг под условия (10 мин)**
   - Освещение, ширина линии.
   - Самый затратный детектор.
5. **Подведение итогов блока OpenCV (5 мин)**

1. Введение: Выпускной экзамен блока OpenCV

Мы прошли 13 занятий по компьютерному зрению:

- **OpenCV:** установка, цветовые пространства, фильтрация, бинаризация, морфология.
- **Детекторы:** линия, цветной объект, ArUco-маркеры, оптический поток, движение (MOG2).
- **Интеграция:** `VisionProcessor`, HTTP-стрим, многопоточность.

Сегодня — **зачёт**. Вы соберёте комплексную миссию с четырьмя детекторами в одном процессоре.

2. Уровень «База»: Комплексная миссия ExamMission

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем и запускаем.

2.1. Требования к зачётной миссии

1. Следование по линии (`LineDetector`).
2. Остановка на маркерах и выполнение действий (`ArucoDetector` + словарь).
3. При обнаружении цветного объекта — приближение к нему (`ColorTracker`).
4. Вывод оценки дрейфа через оптический поток (`OpticalFlow`).
5. Один HTTP-стрим со всеми отрисовками.
6. Запуск через `RobotDriver.spin()`.

2.2. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/exam_mission.py
```

```
#!/usr/bin/env python3
"""Зачётная миссия: линия + ArUco + цвет + оптический поток."""
import math, time
import sys
import rclpy
```

```
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        StreamAction, MoveRelativeAction,
                        RepeatUntil, LogAction,
                        HttpStreamerNode, ImageProcessorNode)


class ExamVisionProcessor(ImageProcessorNode):
    """Зачётный процессор: линия + ArUco + цвет + оптический поток."""

    def __init__(self, driver, fps=30):
        super().__init__(driver, fps=fps)
        # ArUco
        self._aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DIC
T_4X4_50)
        self._aruco_detector = cv2.aruco.ArucoDetector(
            self._aruco_dict, cv2.aruco.DetectorParameters())
        self._K = np.array([[600, 0, 320], [0, 600, 240], [0, 0, 1]], dtyp
e=np.float32)
        # Цвет (красный)
        self._lower_hsv = (0, 100, 100)
        self._upper_hsv = (10, 255, 255)
        # Оптический поток
        self._prev_gray = None
        self._flow_points = None
        self._lk_params = dict(winSize=(21,21), maxLevel=3,
            criteria=(cv2.TERM_CRITERIA_EPS|cv2.TERM_CRITERIA_COUNT, 30, 0
.01))
        self._feature_params = dict(maxCorners=50, qualityLevel=0.3,
                                    minDistance=15, blockSize=7)
        self._frame_count = 0

    def process_frame(self, frame):
        self._frame_count += 1
        h, w = frame.shape[:2]
        display = frame.copy()
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        # ═══ 1. Линия ═══
        if contours:
            best = max(contours, key=cv2.contourArea)
            if cv2.contourArea(best) > 500:
                M = cv2.moments(best)
                if M['m00'] > 0:
                    cx = int(M['m10']/M['m00'])
                    cy = int(M['m01']/M['m00'])
                    self.driver.set_line_info(cx - w/2, 0.0,
detected=True)
                    cv2.drawContours(display, [best], -1, (0, 255, 0), 2)
```

```
                    cv2.circle(display, (cx, cy), 5, (0, 0, 255), -1)
                else:
                    self.driver.set_line_info(0.0, 0.0, detected=False)
            else:
                self.driver.set_line_info(0.0, 0.0, detected=False)
        else:
            self.driver.set_line_info(0.0, 0.0, detected=False)

        # ═══ 2. ArUco ═══
        corners, ids, _ = self._aruco_detector.detectMarkers(frame)
        if ids is not None and len(ids) > 0:
            poses = {int(mid): {'distance': 0.0} for mid in ids.flatten()}
            self.driver.set_marker_info(ids.flatten().tolist(), poses)
            cv2.aruco.drawDetectedMarkers(display, corners, ids)
        else:
            self.driver.set_marker_info([], {})

        # ═══ 3. Цвет ═══
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, self._lower_hsv, self._upper_hsv)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        color_contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                             cv2.CHAIN_APPROX_SIMPLE)
        if color_contours:
            best_c = max(color_contours, key=cv2.contourArea)
            if cv2.contourArea(best_c) > 200:
                Mc = cv2.moments(best_c)
                if Mc['m00'] > 0:
                    obj_cx = int(Mc['m10']/Mc['m00'])
                    obj_cy = int(Mc['m01']/Mc['m00'])
                    self.driver.set_object_position(obj_cx, obj_cy, detect
ed=True)
                    cv2.circle(display, (obj_cx, obj_cy), 10, (255, 0,
0), 2)
                else:
                    self.driver.set_object_position(0, 0, detected=False)
            else:
                self.driver.set_object_position(0, 0, detected=False)
        else:
            self.driver.set_object_position(0, 0, detected=False)

        # ═══ 4. Оптический поток ═══
        need_init = (self._prev_gray is None or self._flow_points is None
                     or len(self._flow_points) < 10 or self._frame_count
% 30 == 0)
        if need_init:
            self._prev_gray = gray.copy()
            pts = cv2.goodFeaturesToTrack(gray, mask=None, **self._feature
_params)
            self._flow_points = pts.reshape(-1, 2) if pts is not None
else None
            self.driver.set_optical_flow(0.0, 0.0)
        elif self._flow_points is not None:
            new_pts, status, _ = cv2.calcOpticalFlowPyrLK(
                self._prev_gray, gray, self._flow_points, None, **self._lk
_params)
```

```
            if new_pts is not None and len(new_pts) > 0:
                good_new = new_pts[status.flatten()==1]
                good_old = self._flow_points[status.flatten()==1]
                if len(good_new) > 5:
                    dx = np.mean(good_new[:,0] - good_old[:,0])
                    dy = np.mean(good_new[:,1] - good_old[:,1])
                    self.driver.set_optical_flow(dx, dy)
                    for old, new in zip(good_old, good_new):
                        cv2.arrowedLine(display,
                            (int(old[0]),int(old[1])), (int(new[0]),int(ne
w[1])),
                            (255, 255, 0), 1, tipLength=0.3)
                    self._flow_points = good_new.reshape(-1, 2)
                    self._prev_gray = gray.copy()

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('exam_jpeg', jpeg.tobytes())


class ExamMission(RobotDriver):

    def __init__(self):
        super().__init__('exam_mission', tick_rate=10)
        self.add_node(HttpStreamerNode(self.bb, key='exam_jpeg',
port=8080))
        self.add_node(ExamVisionProcessor(self, fps=30))
        self.bb.set('marker_actions', {
            0: (0.0, 0.0, 90.0),
            1: (0.0, 0.0, -90.0),
            2: (0.5, 0.0, 0.0),
            3: (0.0, 0.0, 0.0),
        })

    def build_tree(self):
        def drive_with_line(d):
            offset, angle, detected = d.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            # Если видим объект — приближаемся медленнее
            _, _, obj_detected = d.get_object_position()
            if obj_detected:
                vx = 0.05
            return (vx, 0.0, omega, 0.3)

        def marker_or_object(d):
            return d.get_marker_id() is not None or d.get_object_position(
)[2]

        def handle_marker(d):
            mid = d.get_marker_id()
            if mid is not None and mid in d.bb.get('marker_actions', {}):
                dx, dy, dyaw = d.bb.get('marker_actions')[mid]
                d.bb.update(action_dx=dx, action_dy=dy, action_dyaw=dyaw)
                return f"Маркер {mid}: dx={dx}, dy={dy}, dyaw={dyaw}°"
```

```
            return "Нет действия"

        def check_flow(d):
            dx, dy, ready = d.get_optical_flow()
            return f"Дрейф: dx={dx:.1f}, dy={dy:.1f} px"

        return RepeatUntil(
            child=Sequence([
                StreamAction(
                    command_func=drive_with_line,
                    stop_condition=marker_or_object,
                    name="FollowLine"
                ),
                EvalAction(handle_marker, "HandleMarker"),
                MoveRelativeAction(
                    dx=lambda d: d.bb.get('action_dx', 0),
                    dy=lambda d: d.bb.get('action_dy', 0),
                    dyaw_deg=lambda d: d.bb.get('action_dyaw', 0),
                    name="ExecuteAction"
                ),
                EvalAction(check_flow, "CheckFlow"),
                LogAction("   Действие выполнено"),
            ]),
            stop_condition=lambda d: d.get_marker_id() == 3,
            name="MainLoop"
        )


def main():
    rclpy.init()
    RobotDriver.spin(ExamMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/exam_mission.py
```

2.3. Критерии зачёта

1. Робот следует по линии.
2. Останавливается на маркере и выполняет действие.
3. При появлении красного объекта — замедляется.
4. На стриме видны все четыре отрисовки.
5. Ученик может объяснить архитектуру дерева.

3. Уровень «Углублённый»: Анализ ошибок

(Что под капотом)

Разбор типичных проблем.

3.1. Порядок детекторов в process_frame

Порядок не влияет на результат (все читают один `frame`), но влияет на задержку: если ArUco самый медленный, его можно поставить последним — тогда линия обновится раньше. Но для замеров времени порядок не важен.

3.2. Типичные ошибки

1. **Линия не найдена:** `adaptiveThreshold` с неправильным размером окна. Решение: увеличить до 21.
2. **Маркер не распознан:** плохое освещение, мятый маркер. Решение: яркий свет, ровная печать.
3. **Ложный цвет:** в кадре есть красный объект помимо целевого. Решение: сузить HSV-диапазон.
4. **Поток теряет точки:** однородный пол. Решение: уменьшить `minDistance`, увеличить `maxCorners`.

4. Уровень «Исследовательский»: Тюнинг под условия

(Прорыв за границы)

Улучшения.

4.1. Слабое освещение и широкая линия

Задание: предложите улучшение для случая, когда линия шире обычной (5 см вместо 2 см). Какие параметры изменить?

Ответ: увеличить размер окна `adaptiveThreshold` и `kernel` для морфологии, уменьшить `min_area`.

4.2. Мини-исследование: самый затратный детектор

Вопрос: Какой из четырёх детекторов потребляет больше всего CPU?

Методика: добавьте замеры времени для каждого (как в занятии 57).

Ожидание: ArUco > оптический поток > линия > цвет.

5. Подведение итогов блока «OpenCV»

Чему мы научились (занятия 45–58)

1. **OpenCV:** цветовые пространства, фильтрация, бинаризация, морфология, контуры.
2. **Детекторы:** линия, цветной объект (HSV), ArUco-маркеры, оптический поток, движение (MOG2).

3. **Интеграция:** `ImageProcessorNode`, `HttpStreamerNode`, `VisionProcessor`, многопоточность.
4. **Архитектура:** `RepeatUntil` + `StreamAction` + словари действий.

Что дальше

Следующий блок — **«Нейросети: PyTorch и Hailo-8»**. Мы перейдём от классического компьютерного зрения к машинному обучению: YOLO, инференс на NPU, детектирование людей и объектов.

Домашнее задание:

- **Базовое**: Запустите зачётную миссию.
- **Продвинутое**: Добавьте замеры времени детекторов.
- **Исследовательское**: Предложите улучшение для слабого освещения или широкой линии.

Введение в PyTorch и нейропроцессор Hailo-8

Занятие 59: Нейросети: базовые понятия, CNN, установка PyTorch

Содержание занятия: Как работает искусственный нейрон. Свёрточные сети для изображений

Цели занятия

- **Образовательная**: Понять, что такое нейрон, слой, свёртка, пулинг. Установить PyTorch и посмотреть структуру CNN в коде.
- **Развивающая**: Связать знакомые операции OpenCV (свёртка = фильтр) с нейросетевыми слоями.
- **Воспитательная**: Показать, что нейросеть — это не магия, а цепочка математических операций.

План занятия (90 минут)

1. **Организационный момент и введение (5 мин)**
2. **Уровень «База»: Нейрон, слой, CNN (45 мин)**
   - Нейрон: входы, веса, сумма, активация.
   - Слой нейронов. Полносвязный vs свёрточный.
   - CNN: Conv2d, ReLU, MaxPool, Linear.
   - Установка PyTorch, просмотр модели в коде.
3. **Уровень «Углублённый»: Свёртка и пулинг (25 мин)**
   - Ядро свёртки — аналог фильтра OpenCV.
   - Карты признаков. Пулинг — сжатие.
   - Backpropagation — кратко.
4. **Уровень «Исследовательский»: Изменение архитектуры (10 мин)**
   - Добавить/убрать слой.
   - Влияние на время и точность.
5. **Подведение итогов (5 мин)**

1. Введение: От классического зрения — к машинному

Мы научились искать линии, цвета, маркеры с помощью OpenCV. Но есть задачи, где ручная настройка алгоритмов не помогает: распознавание человека, автомобиля, знака.

Нейросети решают это, **обучаясь** на примерах. Сегодня мы начнём с основ: что такое нейрон, как из нейронов собирается CNN, и установим PyTorch.

2. Уровень «База»: Нейрон, слой, CNN

(Инструмент, навык, интерфейс)

Обязательный уровень. От нейрона к первой CNN.

2.1. Искусственный нейрон

Нейрон — это математическая функция:

```
y = f(w1×x1 + w2×x2 + ... + wn×xn + b)
```

- $x_1 \ldots x_n$ — входные сигналы (числа).
- $w_1 \ldots w_n$ — веса (настраиваемые параметры).
- **b** — смещение (bias).
- **f** — функция активации (например, ReLU: `max(0, x)`).

Аналогия: нейрон — это взвешенное голосование. Каждый вход «голосует» с силой w, результат пропускается через порог (активацию).

2.2. Слой нейронов и полносвязная сеть

Слой — это много нейронов, каждый со своими весами. **Полносвязный слой (Linear)** соединяет каждый вход с каждым нейроном.

Для изображения 64×64 = 4096 входов и 1000 нейронов — это 4 миллиона весов. Очень много! Для изображений придумали **свёртку**.

2.3. Свёрточная нейросеть (CNN)

CNN состоит из трёх типов слоёв:

Слой	Что делает	Аналог в OpenCV
Conv2d	Свёртка — выделяет признаки (края, углы, текстуры)	`filter2D`
ReLU	Активация — обнуляет отрицательные значения	`max(0, x)`
MaxPool	Пулинг — сжимает изображение, оставляя максимум в окне	Уменьшение размера
Linear	Полносвязный — принимает решение (класс объекта)	—

Типичная структура: `Conv2d → ReLU → MaxPool → Conv2d → ReLU → MaxPool → Linear`.

2.4. Установка PyTorch и просмотр модели

Проверим установку PyTorch:

```
python3 -c "import torch; print(torch.__version__)"
```

Создайте тестовый скрипт для просмотра структуры CNN:

```
nano ~/ws/rbt/scripts/cnn_demo.py
```

```
#!/usr/bin/env python3
"""Демонстрация структуры простой CNN на PyTorch."""
import torch
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Свёрточные слои
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        # Пулинг
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Активация
        self.relu = nn.ReLU()
        # Полносвязный
        self.fc = nn.Linear(32 * 16 * 16, 10)  # 10 классов

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))  # Conv1 + ReLU + Pool
        x = self.pool(self.relu(self.conv2(x)))  # Conv2 + ReLU + Pool
        x = x.view(x.size(0), -1)                # «Вытягиваем» в строку
        x = self.fc(x)                           # Полносвязный
        return x

model = SimpleCNN()
print(model)
print(f"\nПараметров: {sum(p.numel() for p in model.parameters()):,}")

# Подадим случайный тензор (batch=1, 3 канала, 64×64)
x = torch.randn(1, 3, 64, 64)
y = model(x)
print(f"Вход: {x.shape} → Выход: {y.shape}")
```

Запуск:

```
python3 ~/ws/rbt/scripts/cnn_demo.py
```

Вывод:

```
SimpleCNN(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2)
  (relu): ReLU()
  (fc): Linear(in_features=8192, out_features=10, bias=True)
)
Параметров: 98,890
Вход: torch.Size([1, 3, 64, 64]) → Выход: torch.Size([1, 10])
```

Что мы видим:

- `Conv2d(3, 16, 3)` — свёртка: 3 канала (RGB) → 16 карт признаков, ядро 3×3.

- `MaxPool2d(2)` — пулинг вдвое уменьшает размер: 64×64 → 32×32 → 16×16.
- `Linear(8192, 10)` — полносвязный слой: 8192 входа → 10 выходов (классы).
- **98 890 параметров** — именно они настраиваются при обучении.

3. Уровень «Углублённый»: Свёртка, пулинг и обратное распространение

(Что под капотом)

Как это работает внутри.

3.1. Свёртка: ядро скользит по изображению

Ядро (kernel) — это маленькая матрица весов (3×3, 5×5). Она «скользит» по изображению, умножая свои значения на пиксели и суммируя. Результат — **карта признаков**.

Это то же самое, что `cv2.filter2D`! Разница: в OpenCV ядро задано вручную (Cobel, Гаусс), в CNN веса ядра **обучаются**.

Пример: ядро `[[-1,-1,-1],[-1,8,-1],[-1,-1,-1]]` выделяет края. CNN сама подберёт такие ядра для своей задачи.

3.2. Пулинг: сжатие информации

`MaxPool2d(2)` берёт окно 2×2 и оставляет только максимальное значение. Картинка 64×64 → 32×32. Зачем:

- Уменьшает объём вычислений.
- Делает детектирование **инвариантным к сдвигу** — объект, смещённый на пару пикселей, всё равно будет найден.

3.3. Обратное распространение ошибки (backpropagation)

Обучение — это подбор весов. Алгоритм:

1. Прямой проход: подаём картинку, получаем предсказание.
2. Сравниваем с правильным ответом → **ошибка**.
3. **Обратный проход:** ошибка «растекается» от выхода к входу, обновляя веса так, чтобы в следующий раз ошибиться меньше.

Мы не будем обучать сети с нуля (это требует GPU и часов времени). Мы будем использовать **предобученные** модели.

4. Уровень «Исследовательский»: Изменение архитектуры

(Прорыв за границы)

Эксперименты.

4.1. Добавить/убрать слой

В `SimpleCNN` добавьте третий свёрточный слой `conv3` с 64 фильтрами. Посмотрите, как изменилось число параметров.

Или уберите `conv2` — параметров станет меньше, но и точность (на реальной задаче) упадёт.

4.2. Мини-исследование: слои и параметры

Вопрос: Как количество свёрточных слоёв влияет на число параметров?

Методика: посчитайте параметры для 1, 2, 3 свёрточных слоёв.

Ожидание: каждый новый слой добавляет тысячи параметров. Больше слоёв → больше способность к обучению, но и больше времени.

5. Выводы по занятию

1. **Нейрон** = взвешенная сумма + активация.
2. **CNN** = Conv2d (свёртка) + ReLU (активация) + MaxPool (сжатие) + Linear (решение).
3. **Свёртка** — это обучаемый фильтр, аналог `cv2.filter2D`.
4. **PyTorch** позволяет создать модель в 20 строк.

Связь со следующим занятием: Мы создали CNN с нуля. Но для реальных задач используют **предобученные** модели. На следующем занятии — **YOLO**: детектирование объектов на кадре с камеры.

Домашнее задание:

- **Базовое**: Запустите `cnn_demo.py`. Посмотрите структуру модели.
- **Продвинутое**: Добавьте третий свёрточный слой. Сколько стало параметров?
- **Исследовательское**: Измените размер ядра с 3 на 5. Как изменилось число параметров?

Занятие 60: Предобученные модели: YOLO, инференс на CPU

Содержание занятия: Детектирование объектов с помощью готовой нейросети без обучения

Цели занятия

- **Образовательная**: Загрузить предобученную модель YOLO из torch hub и запустить инференс на кадре с камеры робота.
- **Развивающая**: Понять разницу между классификацией (MobileNet) и детектированием (YOLO), узнать принцип работы YOLO.

- **Воспитательная**: Показать, что использовать нейросети можно уже сегодня, без обучения — достаточно загрузить готовую модель.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: YOLO на CPU (45 мин)**
   - Классификация vs детектирование.
   - Загрузка YOLO из `torch.hub`.
   - Инференс на кадре с камеры.
   - Миссия с `EvalAction` для вывода результатов.
3. **Уровень «Углублённый»: Как работает YOLO (25 мин)**
   - Сетка, bounding boxes, классы.
   - One-stage vs two-stage детекторы.
4. **Уровень «Исследовательский»: Версии YOLO (10 мин)**
   - Сравнение YOLOv5, v8, v11.
   - Компромисс скорость/точность.
5. **Подведение итогов (5 мин)**

1. Введение: Глаза, которые знают, что видят

OpenCV умеет находить линии и цвета, но не знает, **что** это за объект. Нейросеть, обученная на миллионах картинок, может сказать: «это человек», «это стул», «это машина».

YOLO (You Only Look Once) — одна из лучших моделей для детектирования объектов в реальном времени. Мы загрузим готовую версию и запустим на роботе.

2. Уровень «База»: YOLO на CPU

(Инструмент, навык, интерфейс)

Обязательный уровень. Загружаем модель и запускаем на кадре с камеры.

2.1. Классификация vs Детектирование

Задача	Что делает	Пример модели
Классификация	Отвечает «что на картинке?» (один объект)	MobileNet, ResNet
Детектирование	Находит **все** объекты и их координаты	YOLO, SSD, R-CNN

Для робота нужно **детектирование** — он должен знать не только «в кадре человек», но и **где именно**.

2.2. Загрузка YOLO из torch hub

PyTorch Hub — это репозиторий готовых моделей. YOLOv5 загружается одной строкой:

```
self._model = YOLO('yolov5nu.pt')
```

`yolov5nu` — оптимальна для Raspberry Pi. Есть `n` (nano), `m` (medium), `l` (large).

2.3. Миссия YOLOTestMission

Создайте файл:

```
nano ~/ws/rbt/scripts/yolo_test_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: тест YOLO на кадре с камеры + HTTP стрим."""
import sys, time
import rclpy
import cv2
import torch
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        HttpStreamerNode, ImageProcessorNode,
RepeatUntil)
from ultralytics import YOLO

# Заставляем PyTorch использовать все 4 ядра Raspberry Pi 5
torch.set_num_threads(4)
torch.set_num_interop_threads(4)


class YOLOProcessor(ImageProcessorNode):
    """Процессор YOLO: загрузка модели и инференс."""

    def __init__(self, driver, fps=10):
        super().__init__(driver, fps=fps)
        self.get_logger().info("Загрузка YOLOv5n ... ")
        # Загружаем оптимизированную для RPi5 Nano-версию
        self._model = YOLO('yolov5nu.pt')
        self.get_logger().info("YOLOv5n загружена")

    def process_frame(self, frame):
        display = frame.copy()
        t0 = time.time()

        results = self._model.predict(frame, verbose=False)
        boxes = results[0].boxes

        count = 0
        if boxes is not None and len(boxes) > 0:
            for box in boxes:
                x1, y1, x2, y2 = map(int, box.xyxy[0].tolist())
                conf = float(box.conf[0])
                cls = int(box.cls[0])
                name = self._model.names[cls]

                if conf < 0.5:
                    continue
                count += 1
                cv2.rectangle(display, (x1, y1), (x2, y2), (0, 255, 0), 2)
```

```
                label = f"{name} {conf:.2f}"
                cv2.putText(display, label, (x1, y1-5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        dt = (time.time() - t0) * 1000

        self.driver.bb.update(
            yolo_count=count,
            yolo_time_ms=round(dt, 1),
        )

        cv2.putText(display, f"YOLO: {count} obj, {dt:.0f}ms", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('yolo_jpeg', jpeg.tobytes())

class YOLOTestMission(RobotDriver):

    def __init__(self):
        super().__init__('yolo_test_mission', tick_rate=2)
        self.add_node(HttpStreamerNode(self.bb, key='yolo_jpeg',
port=8080))
        self.add_node(YOLOProcessor(self, fps=10))

    def build_tree(self):
        def check_yolo(d):
            cnt = d.bb.get('yolo_count', 0)
            ms = d.bb.get('yolo_time_ms', 0)
            d.get_logger().info(f"   Объектов: {cnt}, инференс: {ms} мс",
throttle_duration_sec=1)
            return f"Объектов: {cnt}"

        def minute_passed(d):
            return (time.time() - d.bb.get('start_time', time.time())) >=
60.0

        return Sequence([
            EvalAction(lambda d: d.bb.set('start_time', time.time()) or "
   Старт", "Init"),
            RepeatUntil(
                child=EvalAction(check_yolo, "CheckYOLO"),
                stop_condition=minute_passed,
                name="YOLO 60 сек"
            ),
            EvalAction(lambda d: "   Готово", "Finish"),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(YOLOTestMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/yolo_test_mission.py
```

Внимание: первый запуск скачает модель (~14 MB). На Raspberry Pi инференс на CPU — **500–1500 мс на кадр**.

2.4. Наблюдение

На стриме будут зелёные рамки вокруг объектов с подписями: `person 0.85`, `chair 0.72`, `bottle 0.63`. Время инференса — сотни миллисекунд на CPU.

3. Уровень «Углублённый»: Как работает YOLO

(Что под капотом)

Сетка и предсказания.

3.1. Разбиение на сетку

YOLO делит изображение на сетку (например, 13×13). Каждая ячейка предсказывает:

- **x1, y1, x2, y2 = map(int, box.xyxy[0].tolist())** — границы объекта.
- **conf = float(box.conf[0])** — насколько модель уверена, что в ячейке есть объект.
- **cls = int(box.cls[0])** — что это за объект.
- **name = self._model.names[cls]** — имя класса (person, car, ...).

Это делается **за один проход** (one-stage) — поэтому YOLO быстрый.

3.2. One-stage vs two-stage детекторы

Тип	Пример	Как работает	Скорость
Two-stage	R-CNN, Fast R-CNN	Сначала находит регионы, потом классифицирует	Медленно
One-stage	YOLO, SSD	Сразу предсказывает и boxes, и классы	Быстро

Для робота важна скорость → YOLO.

3.2. Требования на 26.04 (установка только для симулятора на машине с симулятором, на RPI5 все уже установлено)

Так как на Raspberry Pi 5 стоит новая Ubuntu 26.04 с Python 3.14, базовый PyTorch нужно ставить из системных репозиториев:

```
sudo apt update
sudo apt install python3-torch python3-venv git build-essential libjpeg-dev zlib1g-dev libpng-dev -y
```

Создаем изолированную среду для проекта, но разрешаем ей использовать системный `torch`:

```
# 1. Создаем окружение в домашней папке (или папке воркспейса)
python3 -m venv ~/.venv

# 2. Включаем доступ к системным пакетам (чтобы подцепился python3-torch)
sed -i 's/include-system-site-packages = false/include-system-site-
packages = true/g' ~/.venv/pyvenv.cfg

# 3. Активируем окружение
source ~/.venv/bin/activate
```

Ставим остальные пакеты через `pip`. Флаг `--no-deps` для `torchvision` обязателен, чтобы менеджер пакетов не попытался скачать тяжелую CUDA-версию:

```
# Обновляем инсталлятор
pip install --upgrade pip

# Устанавливаем движок YOLO
pip install ultralytics pandas polars pyarrow

#Сносим только torch и torchvision, которые притянул pip. Всё остальное
(мелкие зависимости вроде tqdm и matplotlib) останется нетронутым
pip uninstall torch torchvision -y

# Устанавливаем совместимый torchvision без лишних зависимостей
pip install torchvision==0.24.0 --no-deps --force-reinstall
```

Поскольку C++ библиотеки torchvision для Python 3.14 в pip собраны «криво», отключаем проверку сломанного оператора (на CPU он все равно заменяется стандартным Python-кодом):

```
sed -i 's/@torch.library.register_fake("torchvision::nms")/#
@torch.library.register_fake("torchvision::nms")/' ~/.venv/lib/python3.14/
site-packages/torchvision/_meta_registrations.py
```

Проверяем результат:

```
python3 -c "import ultralytics; import torchvision; print('   ИДЕАЛЬНО.
Всё импортируется без ошибок.')"
```

4. Уровень «Исследовательский»: Версии YOLO

(Прорыв за границы)

Сравнение.

4.1. Сравнение YOLOv5, v8, v11

Замените `yolov5s` на `yolov8s` (нужен `ultralytics` пакет) или `yolo11s`. Сравните время инференса и количество найденных объектов на одном кадре.

4.2. Мини-исследование: скорость vs точность

Вопрос: Насколько отличается время инференса у `yolov5n` (nano), `yolov5s` (small), `yolov5m` (medium)?

Методика: замерьте время для трёх версий на одном кадре.

Ожидание: nano — быстрее всех, но меньше точность. medium — точнее, но медленнее.

5. Выводы по занятию

1. **YOLO** — детектор объектов за один проход. Загружается из torch hub.
2. **Инференс на CPU** медленный (500+ мс) — для реального времени нужен ускоритель.
3. **YOLO делит изображение на сетку** и предсказывает boxes + классы для каждой ячейки.

Связь со следующим занятием: Мы получили сырые результаты YOLO. Теперь нужно их **обработать**: отфильтровать по confidence, убрать дубли через NMS, сохранить в `driver.set_detections()`.

Домашнее задание:

- **Базовое**: Запустите миссию. Посмотрите на детекции в стриме.
- **Продвинутое**: Замерьте время инференса для `yolov5n` и `yolov5s`.
- **Исследовательское**: Сравните количество детекций и время для трёх версий YOLO.

Занятие 61: Обработка результатов YOLO: boxes, классы, confidence, NMS

Содержание занятия: Фильтрация и сохранение детекций. Подготовка к использованию Hailo-8

Цели занятия

- **Образовательная**: Научиться обрабатывать результаты YOLO: фильтровать по confidence, понимать NMS, сохранять в `driver.set_detections()`.
- **Развивающая**: Понять, как работает IoU и зачем нужен Non-Maximum Suppression.
- **Воспитательная**: Показать, что сырой вывод нейросети требует постобработки перед использованием.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Обработка и сохранение (45 мин)**
   - Структура результатов: x1, y1, x2, y2, confidence, name.
   - Фильтрация по порогу.
   - Сохранение в `driver.set_detections()`.

- Миссия с выводом детекций.

3. **Уровень «Углублённый»: NMS и IoU (25 мин)**
   - Зачем нужен NMS.
   - IoU — метрика перекрытия.
4. **Уровень «Исследовательский»: Подбор порогов (10 мин)**
   - Confidence и IoU для полигона.
   - Ложные и пропущенные детекции.
5. **Подведение итогов (5 мин)**

1. Введение: Сырые данные → полезная информация

YOLO возвращает **все** потенциальные объекты — в том числе с низкой уверенностью и дубликаты. Прежде чем использовать результаты в дереве поведения, их нужно:

1. **Отфильтровать** по confidence (уверенность).
2. **Убрать дубли** через NMS (Non-Maximum Suppression).
3. **Сохранить** в `driver.set_detections()` для использования в `Condition` и `EvalAction`.

2. Уровень «База»: Обработка и сохранение

(Инструмент, навык, интерфейс)

Обязательный уровень. Обрабатываем результаты YOLO.

2.1. Структура результатов YOLO

После `results = self._model.predict(frame, verbose=False)` получаем `boxes = results[0].boxes`, `for box in boxes`:

- **x1, y1, x2, y2 = map(int, box.xyxy[0].tolist())** — границы объекта.
- **conf = float(box.conf[0])** — насколько модель уверена, что в ячейке есть объект.
- **cls = int(box.cls[0])** — что это за объект.
- **name = self._model.names[cls]** — имя класса (person, car, ...).

2.2. NMS — удаление дубликатов

YOLO может нарисовать несколько рамок вокруг одного объекта. NMS оставляет только одну — с максимальным confidence, подавляя остальные, если они сильно перекрываются (IoU > порога).

YOLO из `ultralytics` делает NMS **автоматически** внутри `predict()`. Но мы всё равно фильтруем по confidence.

2.3. Миссия YOLODetectorMission

Создайте файл:

```
nano ~/ws/rbt/scripts/yolo_detector_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: YOLO с обработкой результатов + driver.set_detections() +
стрим."""
import sys, time
import rclpy
import cv2
import torch
from ultralytics import YOLO
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        HttpStreamerNode, ImageProcessorNode,RepeatUntil)

torch.set_num_threads(4)
torch.set_num_interop_threads(4)


class YOLODetectorProcessor(ImageProcessorNode):
    """YOLO детектор с сохранением в driver.set_detections()."""

    def __init__(self, driver, fps=10):
        super().__init__(driver, fps=fps)
        self.get_logger().info("Загрузка YOLOv5n ... ")
        self._model = YOLO('yolov5nu.pt')
        self.get_logger().info("YOLOv5n загружена")

    def process_frame(self, frame):
        display = frame.copy()
        t0 = time.time()

        results = self._model.predict(frame, verbose=False)
        boxes = results[0].boxes
        dt = (time.time() - t0) * 1000

        det_list = []
        count = 0
        if boxes is not None and len(boxes) > 0:
            for box in boxes:
                conf = float(box.conf[0])
                if conf < 0.5:
                    continue
                count += 1
                x1, y1, x2, y2 = map(int, box.xyxy[0].tolist())
                cls = int(box.cls[0])
                name = self._model.names[cls]

                det_list.append((name, conf, x1, y1, x2, y2))
                cv2.rectangle(display, (x1, y1), (x2, y2), (0, 255, 0), 2)
                cv2.putText(display, f"{name} {conf:.2f}", (x1, y1-5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        self.driver.set_detections(det_list)
```

```
        self.driver.bb.update(
            yolo_count=count,
            yolo_time_ms=round(dt, 1),
        )

        cv2.putText(display, f"YOLO: {count} obj, {dt:.0f}ms", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('yolo_jpeg', jpeg.tobytes())

class YOLODetectorMission(RobotDriver):

    def __init__(self):
        super().__init__('yolo_detector_mission', tick_rate=2)
        self.add_node(HttpStreamerNode(self.bb, key='yolo_jpeg',
port=8080))
        self.add_node(YOLODetectorProcessor(self, fps=10))

    def build_tree(self):
        def check_yolo(d):
            cnt = d.bb.get('yolo_count', 0)
            ms = d.bb.get('yolo_time_ms', 0)
            if d.has_detections():
                classes = set(det[0] for det in d.get_detections())
                return f"   {cnt} объектов: {', '.join(classes)}, {ms} мс"
            return f"   Объектов нет, {ms} мс"

        def minute_passed(d):
            return (time.time() - d.bb.get('start_time', time.time())) >=
60.0

        return Sequence([
            EvalAction(lambda d: d.bb.set('start_time', time.time()) or "
   Старт на 60 сек", "Init"),
            RepeatUntil(
                child=EvalAction(check_yolo, "CheckYOLO"),
                stop_condition=minute_passed,
                name="YOLO 60 сек"
            ),
            EvalAction(lambda d: "   Минута прошла", "Finish"),
        ])

def main():
    rclpy.init()
    RobotDriver.spin(YOLODetectorMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/yolo_detector_mission.py
```

2.4. Методы драйвера для детекций

- `driver.set_detections(det_list)` — сохраняет список кортежей `(class_name, confidence, x1, y1, x2, y2)`.
- `driver.get_detections()` → список детекций.
- `driver.has_detections()` → `True` если есть объекты.
- `driver.is_class_detected("person")` → `True` если есть человек.
- `driver.is_person_detected()` → сокращение для `is_class_detected("person")`.
- `driver.get_detections_by_class("person")` → только детекции этого класса.

3. Уровень «Углублённый»: NMS и IoU

(Что под капотом)

Как работает подавление дубликатов.

3.1. Как работает NMS

1. Отсортировать все рамки по убыванию confidence.
2. Взять первую (с максимальной уверенностью) — это точно объект.
3. Сравнить её со всеми остальными: если **IoU > порога** (например, 0.5) — удалить как дубликат.
4. Повторить для следующей оставшейся рамки.

3.2. IoU (Intersection over Union)

```
IoU = Площадь_пересечения / Площадь_объединения
```

Две рамки, полностью совпадающие → IoU = 1.0. Две рамки, не перекрывающиеся → IoU = 0.0. Порог 0.5 означает: если рамки перекрываются больше чем наполовину — это дубликат.

Без NMS один объект может быть обведён 5–10 рамками — YOLO предсказывает несколько boxes для одной и той же области.

4. Уровень «Исследовательский»: Подбор порогов

(Прорыв за границы)

Эксперименты.

4.1. Оптимальный порог confidence

Увеличьте порог до 0.7 — ложных детекций меньше, но реальные объекты могут пропускаться. Уменьшите до 0.3 — больше объектов, но много шума.

Задание: найдите оптимальный порог для вашего полигона.

4.2. Мини-исследование: порог NMS и детекции

Вопрос: Как порог IoU (в NMS) влияет на количество дубликатов?

Методика: в YOLO можно задать `model.predict(frame, iou=0.3)` и `iou=0.7`. Сравните количество детекций одного объекта.

Ожидание: при низком IoU дубликатов меньше, но разные объекты могут сливаться.

5. Выводы по занятию

1. **YOLO возвращает данные** с информацией `x1, y1, x2, y2, confidence, name`.
2. **Фильтрация по confidence** (обычно > 0.5) убирает слабые детекции.
3. **NMS** (встроен в `ultralytics.YOLO`) удаляет дубликаты по IoU.
4. **`driver.set_detections()`** сохраняет информацию `(class_name, confidence, x1, y1, x2, y2)`.

Связь со следующим занятием: На CPU инференс медленный. На следующем занятии — **Hailo-8**: нейропроцессор, который ускоряет YOLO в 10–30 раз.

Домашнее задание:

- **Базовое**: Запустите миссию. Посмотрите детекции в стриме.
- **Продвинутое**: Подберите порог confidence для вашего полигона.
- **Исследовательское**: Сравните количество детекций при `iou=0.3` и `iou=0.7`.

Занятие 62: Hailo-8: архитектура, HailoRT, конвертация модели

Содержание занятия: Нейропроцессор для робота. Как модель превращается в Hailo Executable File

Цели занятия

- **Образовательная**: Узнать, что такое Hailo-8, проверить его наличие на роботе, понять этапы конвертации модели.
- **Развивающая**: Понять, чем архитектура NPU отличается от CPU/GPU и почему Hailo-8 эффективнее для реальных задач.
- **Воспитательная**: Показать, что современная робототехника — это симбиоз механики, электроники и специализированного ИИ-железа.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**

2. **Уровень «База»: Знакомство с Hailo-8 (45 мин)**
    • Что такое NPU и зачем он роботу.
    • Проверка Hailo-8: `hailortcli fw-control identify`.
    • Этапы конвертации: PyTorch → ONNX → HEF.
    • Запуск конвертации, получение `.hef`.
3. **Уровень «Углублённый»: Архитектура Hailo-8 (25 мин)**
    • Потоковый процессор.
    • Сравнение с CPU и GPU.
    • ONNX как промежуточный формат.
4. **Уровень «Исследовательский»: Hailo vs Coral (10 мин)**
    • Сравнение с Google Coral Edge TPU.
    • Неподдерживаемые слои.
5. **Подведение итогов (5 мин)**

1. Введение: Железо для искусственного интеллекта

CPU робота (Raspberry Pi) справляется с YOLO за 200–1500 мс. Для движения на скорости 0.2 м/с это 4–30 см пройденного пути — слишком много для безопасной остановки перед человеком.

Hailo-8 — это **NPU (Neural Processing Unit)**, специализированный чип для выполнения нейросетей. Он делает инференс за 5–15 мс — в 20–100 раз быстрее CPU.

Сегодня мы проверим, что Hailo-8 работает на роботе, и научимся конвертировать модель для него.

2. Уровень «База»: Знакомство с Hailo-8

(Инструмент, навык, интерфейс)

Обязательный уровень. Проверяем устройство и конвертируем модель.

2.1. Что такое NPU и зачем он роботу

Процессор	Универсальность	Скорость на нейросетях	Энергопотребление
CPU (RPi)	Универсальный	Медленно (200–1500 мс)	Среднее
GPU	Графика	Быстро (20–50 мс)	Высокое (50–200 Вт)
NPU (Hailo-8)	Только нейросети	Очень быстро (5–15 мс)	Низкое (2–5 Вт)

NPU оптимизирован для операций нейросетей: умножение матриц, свёртки, активации. Он не умеет работать с файлами или сетью — только считать нейросети.

2.2. Проверка Hailo-8 на роботе

Выполните в терминале:

```
hailortcli fw-control identify
```

Если Hailo-8 подключён, вы увидите информацию об устройстве: версию прошивки, идентификатор. Если нет — сообщите преподавателю.

2.3. Этапы конвертации модели

Hailo-8 не понимает PyTorch-модели напрямую. Нужна конвертация:

```
PyTorch (.pt) → ONNX (.onnx) → Hailo Executable File (.hef)
```

1. **PyTorch → ONNX:** экспорт обученной модели в открытый формат ONNX.

```
python3 -c "from ultralytics import YOLO;
YOLO('yolov5nu.pt').export(format='onnx')"
```

2. **ONNX → HEF:** компиляция под архитектуру Hailo-8 утилитой `hailomz`.

```
hailomz compile yolov5nu.onnx \
  --ckpt yolov5nu.pt \
  --hw-arch hailo8l \
  --classes 80 \
  --performance best
```

Результат — файл `yolov5nu.hef`, готовый для инференса на Hailo-8.

2.4. Что такое HEF

HEF (Hailo Executable File) — это бинарный файл, содержащий:

- Скомпилированную нейросеть.
- Параметры (веса).
- Инструкции для потокового процессора Hailo-8.

Он загружается в HailoRT (Hailo Runtime) и выполняется аппаратно.

3. Уровень «Углублённый»: Архитектура Hailo-8

(Что под капотом)

Почему он быстрее.

3.1. Потоковый процессор

Hailo-8 построен по **dataflow**-архитектуре. Данные (пиксели) «протекают» через конвейер вычислительных блоков:

1. Каждый блок выполняет свою операцию (свёртку, активацию, пулинг).
2. Результат сразу передаётся следующему блоку.
3. **Все блоки работают одновременно** (параллельно).

В CPU команды выполняются последовательно. В Hailo-8 — как на заводском конвейере.

3.2. Почему Hailo эффективнее GPU для ИИ задач

GPU (NVIDIA Jetson Nano):

- Гибкий, но энергозатратный (5–10 Вт).
- Требует активного охлаждения.
- Хорош для обучения, избыточен для инференса.

NPU (Hailo-8):

- Специализированные блоки только для операций нейросетей.
- Очень низкое энергопотребление (2–5 Вт).
- Не требует вентилятора.
- Идеален для **инференса** на мобильном роботе.

3.3. ONNX — промежуточный формат

ONNX (Open Neural Network Exchange) — открытый формат для нейросетей. Он позволяет:

- Обучить модель в PyTorch.
- Запустить на Hailo-8 (через компилятор).
- Или на Google Coral, или на Intel NCS — без изменений.

Это как PDF для документов: создал в одном приложении, открыл в другом.

3.4. Если обновилось ядро и драйвера Hailo-8 слетели

Если после обновления системы команда hailortcli scan или hailortcli fw-control identify выдает пустой вывод или ошибку, проверьте статус модуля ядра:

```
sudo dmesg | grep -i hailo
```

Если вывод пустой — ядро потеряло драйвер при обновлении.

Заходим в ваш скачанный Git-репозиторий драйверов, обновляем его код и компилируем строго под версию 4.24.0 (которая умеет работать с новыми ядрами 7.x):

```
# 1. Переходим в папку репозитория драйверов
git clone https://github.com/hailo-ai/hailort-drivers.git
cd hailort-drivers

# 2. Сбрасываем старые файлы и качаем свежие исправления
git reset --hard
git pull

# 3. Переключаемся на нужную ветку для Hailo-8L
git checkout hailo8
```

```
# 4. Компилируем модуль ядра
cd linux/pcie
make clean
make all

# 5. Устанавливаем драйвер в папки нового ядра
sudo make install

# 6. Обновляем зависимости модулей системы
sudo depmod -a

# Файл прошивки должен лежать строго по этому пути и с этим именем:
sudo cp /путь/к/скачанному/hailo8_fw.4.24.0.bin /lib/firmware/hailo/
hailo8_fw.bin

sudo modprobe hailo_pci

# Проверяем логи ядра — они должны показать успешную загрузку FW за ~85 ms
sudo dmesg | grep -i hailo
```

Если выскочит ошибка несовпадения версий (например, библиотека осталась 4.24, а драйвер стал 4.25), пересоберите основной софт hailortcli из его репозитория Git с флагом совместимости CMake для Ubuntu 26.04:

```
sudo cmake --uninstall

# 1. Переходим в репозиторий основного софта
git clone https://github.com/hailo-ai/hailort.git
cd hailort

# 2. Переключаем код на версию 4.25.0
git checkout v4.25.0

# 3. Удаляем старую конфигурацию, если она была
rm -rf build

# 4. Конфигурируем CMake с флагами Python и политикой совместимости
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -DCMAKE_POLICY_VERSION_MINI
MUM=3.5 -DHAILO_BUILD_PYTHON=ON
cmake -S . -B build \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_POLICY_VERSION_MINIMUM=3.5 \
  -DHAILO_BUILD_PYTHON=ON \
  -DPYTHON_EXECUTABLE=$(which python3.14)

# 5. Собираем и устанавливаем в систему
cmake --build build --config Release
sudo cmake --install build

#python3.14 -m pip install wheel setuptools

# Переходим к скрипту сборки пакета
cd ~/ws/git/hailort/hailort/libhailort/bindings/python/platform

# Собираем wheel-пакет под Python 3.14
```

```
python3.14 setup.py bdist_wheel

#pip install dist/hailort-*.whl

# правильная установка в окружение
sudo -i

python3.14 -m venv /opt/hailo_env

apt install -y libjpeg-dev zlib1g-dev libpng-dev python3-torch

sed -i 's/include-system-site-packages = false/include-system-site-
packages = true/g' /opt/hailo_env/pyvenv.cfg

source /opt/hailo_env/bin/activate

mkdir -p /opt/hailo_tmp

TMPDIR=/opt/hailo_tmp pip install  torchvision==0.24.0 --no-deps --force-
reinstall

TMPDIR=/opt/hailo_tmp pip install --no-cache-dir ultralytics  pandas polar
s pyarrow

pip uninstall torch torchvision -y

TMPDIR=/opt/hailo_tmp pip install  torchvision==0.24.0 --no-deps --force-
reinstall

rm -rf /opt/hailo_tmp

sed -i 's/@torch.library.register_fake("torchvision::nms")/#
@torch.library.register_fake("torchvision::nms")/' /opt/hailo_env/lib/
python3.14/site-packages/torchvision/_meta_registrations.py

pip install /home/pi/ws/git/hailort/hailort/libhailort/bindings/python/
platform/dist/hailort-*.whl

chown -R root:root /opt/hailo_env
find /opt/hailo_env -type d -exec chmod 755 {} +
find /opt/hailo_env -type f -exec chmod 644 {} +
chmod -R 755 /opt/hailo_env/bin

deactivate

exit

source /opt/hailo_env/bin/activate

python3 -c "import ultralytics; import torchvision; from hailo_platform
import HEF; print('   ИДЕАЛЬНО. Всё импортируется без ошибок.')"

deactivate
```

4. Уровень «Исследовательский»: Hailo vs Coral

(Прорыв за границы)

Сравнение с альтернативами.

4.1. Сравнение с Google Coral Edge TPU

Характеристика	Hailo-8	Google Coral
Производительность	26 TOPS	4 TOPS
Энергопотребление	2.5–5 Вт	2 Вт
Конвертация	PyTorch → ONNX → HEF	TensorFlow Lite → Edge TPU
Поддержка PyTorch	Да (через ONNX)	Нет (только TFLite)

4.2. Мини-исследование: неподдерживаемые слои

Вопрос: Все ли слои PyTorch поддерживаются Hailo-8?

Ответ: не все. Например, `Upsample` (увеличение размера) может не поддерживаться. При конвертации `hailomz` сообщит об ошибке и предложит замену.

Задание: при конвертации YOLOv5s посмотрите логи `hailomz compile`. Есть ли предупреждения о замене слоёв?

5. Выводы по занятию

1. **Hailo-8** — NPU, ускоряет нейросети в 20–100 раз по сравнению с CPU.
2. **Конвертация:** PyTorch → ONNX → HEF через `hailomz compile`.
3. **Потоковая архитектура** обеспечивает параллелизм.
4. **ONNX** — универсальный формат для переноса моделей между платформами.

Связь со следующим занятием: Мы сконвертировали модель. На следующем занятии — **сравнение CPU vs Hailo-8**: замер FPS и latency на одном роботе.

Домашнее задание:

- **Базовое**: Проверьте Hailo-8 через `hailortcli fw-control identify`.
- **Продвинутое**: Сконвертируйте `yolov5nu.pt` в `.hef`.
- **Исследовательское**: Сравните характеристики Hailo-8 и Google Coral. Какой лучше для YOLO?

Занятие 63: Сравнение CPU vs Hailo-8: FPS, latency

Содержание занятия: Количественное сравнение производительности. Почему Hailo-8 критичен для робота

Цели занятия

- **Образовательная**: Написать `HailoDetectorProcessor` для инференса на Hailo-8, сравнить FPS и latency с CPU-версией.
- **Развивающая**: Понять, почему для движущегося робота важнее **latency**, а не просто FPS.
- **Воспитательная**: Показать, что выбор железа — инженерное решение, основанное на цифрах.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: HailoDetectorProcessor (45 мин)**
   - Загрузка `.hef` через HailoRT.
   - Сравнительная миссия: CPU vs Hailo-8.
3. **Уровень «Углублённый»: Latency vs FPS (25 мин)**
   - Почему latency критична.
   - Расчёт тормозного пути.
4. **Уровень «Исследовательский»: Энергопотребление (10 мин)**
   - Влияние на время работы от батареи.
5. **Подведение итогов (5 мин)**

1. Введение: Цифры решают всё

Мы знаем, что Hailo-8 быстрее CPU. Но **насколько** быстрее? И почему это важно?

Сегодня мы **измерим** разницу в скорости на одном и том же кадре. И поймём, почему 100 мс задержки — это разница между «остановился перед человеком» и «врезался в человека».

2. Уровень «База»: HailoDetectorProcessor

(Инструмент, навык, интерфейс)

Обязательный уровень. Пишем процессор для Hailo-8 и сравниваем.

2.1. HailoDetectorProcessor

Создайте файл:

```
nano ~/ws/rbt/scripts/hailo_compare_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: сравнение CPU (YOLOv5n) vs Hailo-8 на одном кадре."""
import sys, time
import rclpy, threading
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction, TreeDashboardNo
de, HailoProcessorNode,
                        HttpStreamerNode, ImageProcessorNode,
RepeatUntil)

# ── CPU версия ──
import torch
from ultralytics import YOLO
#torch.set_num_threads(4)
#torch.set_num_interop_threads(4)

class CPUYOLOProcessor(ImageProcessorNode):
    def __init__(self, driver, fps=10):
        super().__init__(driver, fps=fps, node_name="CPU_YOLOv5n")
        self._model = YOLO('yolov5nu.pt')
        self.get_logger().info("CPU YOLOv5n загружена")

    def process_frame(self, frame):
        t0 = time.time()
        results = self._model.predict(frame, verbose=False)
        dt = (time.time() - t0) * 1000

        boxes = results[0].boxes
        count = 0
        if boxes is not None and len(boxes) > 0:
            for box in boxes:
                conf = float(box.conf[0])
                if conf > 0.5:
                    count += 1

        self.driver.bb.update(cpu_ms=round(dt, 1), cpu_count=count)
        # не шлём JPEG (стрим только с Hailo)

# ── Hailo-8 версия убедитесь, что установлены модули в предыдущем
занятии──
import cv2
import numpy as np

class HailoDetectorProcessor(HailoProcessorNode):
    def __init__(self, driver, hef_path, fps=30):
        # Инициализируем наш правильный базовый класс HailoProcessorNode
        super().__init__(driver, hef_path, fps=fps, node_name="hailo_yolov
5s_detector")

        # Порог уверенности для отсечения ложных срабатываний
        self.conf_threshold = 0.4

    def handle_hailo_result(self, output, display, dt_ms):
        """
```

```
        Финальный, устойчивый к багам pyhailort парсинг структурированного
массива NMS.
        Работает напрямую со встроенными типами через принудительную
индексацию.
        """
        nms_layer = list(output.keys())[0]
# Извлекаем чистую строку имени слоя

        # Получаем кастомный структурированный массив из чипа
        raw_data = output[nms_layer]

        # Извлекаем батч (нулевой элемент) обычным питоновским
индексированием,

# полностью отказываясь от np.squeeze(), который приводил к ошибке
inhomogeneous shape!
        detections_by_class = raw_data[0]

        img_h, img_w = display.shape[:2]
        det_list = []
        count = 0

        # Рабочий порог уверенности (изменим на 0.30)
        CONF_THRESH = 0.40

        # Параметры летербокса  базового класса
        pad_x, pad_y, scale = self._pad_info

        # Безопасно перебираем все 80 классов датасета COCO
        num_classes = len(detections_by_class)
        for class_id in range(num_classes):
            class_matrix = detections_by_class[class_id]

            # Проверяем наличие объектов для этого класса строго через
len()
            if class_matrix is None or len(class_matrix) == 0:
                continue

            # Итерируемся по строкам таблицы детекций конкретного класса
            for j in range(len(class_matrix)):
                row = class_matrix[j]

                # Защита: если структура строки внезапно вернулась пустой
или побилась
                if row is None or len(row) < 5:
                    continue

                # Извлекаем 5 аппаратных параметров из строки
структурированного массива.
                # Порядок в официальном NMS YOLOv5s: ymin, xmin, ymax,
xmax, confidence
                ymin, xmin, ymax, xmax, confidence = row[:5]

                # Фильтруем пустые заглушки чипа по порогу уверенности
                if float(confidence) > CONF_THRESH:
                    confidence = float(confidence)
```

```
                    # 1. Переводим нормализованные координаты (0..1) в
пиксели холста (640x640)
                    x_box = xmin * self._w_in
                    y_box = ymin * self._h_in
                    w_box = (xmax - xmin) * self._w_in
                    h_box = (ymax - ymin) * self._h_in

                    # 2. Вычитаем серые полосы letterbox и масштабируем
под исходную камеру робота
                    x = int((x_box - pad_x) / scale)
                    y = int((y_box - pad_y) / scale)
                    w = int(w_box / scale)
                    h = int(h_box / scale)

                    # Ограничиваем рамки внутри физического кадра вашей
камеры
                    x = max(0, x)
                    y = max(0, y)
                    w = min(img_w - x, w)
                    h = min(img_h - y, h)

                    if w > 0 and h > 0:
                        count += 1
                        # Записываем в стандартный формат вашего робота:
[class_id, confidence, [x, y, w, h]]
                        det_list.append([int(class_id), confidence, [x,
y, w, h]])

                        # Рисуем рамку детекции и текст на кадре для
видеопотока
                        cv2.rectangle(display, (x, y), (x + w, y + h),
(0, 255, 0), 2)
                        cv2.putText(display, f"ID:{class_id} {confidence:.
2f}", (x, max(y - 5, 10)),
                                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
255, 0), 2)

        # Отправка результатов и телеметрии во внешний драйвер вашего
робота
        self.driver.set_detections(det_list)
        self.driver.bb.update(hailo_ms=round(dt_ms, 1), hailo_count=count)

        # Вывод общего времени работы ИИ на экране
        cv2.putText(display, f"Hailo YOLOv5s: {dt_ms:.1f}ms Objects: {coun
t}", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

        # Сжимаем обработанный кадр в JPEG для стриминг-буфера робота
        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('hailo_jpeg', jpeg.tobytes())


class CompareMission(RobotDriver):
    def __init__(self):
```

```
        super().__init__('compare_mission', tick_rate=2)
        self.add_node(TreeDashboardNode(self, port=8079))
        self.add_node(HttpStreamerNode(self.bb, key='hailo_jpeg', port=808
0))
        #self.add_node(HailoDetectorProcessor(self, 'yolov11x.hef',
fps=30))
        self.add_node(CPUYOLOProcessor(self, fps=5))
        # Раскомментировать, когда Hailo-8 готов:
        self.add_node(HailoDetectorProcessor(self, 'yolov5nu.hef', fps=30)
)

    def build_tree(self):
        def check_compare(d):
            cpu_ms = d.bb.get('cpu_ms', 0)
            hailo_ms = d.bb.get('hailo_ms', 0)
            cpu_cnt = d.bb.get('cpu_count', 0)
            hailo_cnt = d.bb.get('hailo_count', 0)
            speedup = f"{cpu_ms/hailo_ms:.0f}x" if isinstance(hailo_ms, (i
nt, float)) and hailo_ms > 0 else "—"
            return (f"CPU: {cpu_ms}ms ({cpu_cnt} obj) | "
                    f"Hailo: {hailo_ms}ms ({hailo_cnt} obj) | "
                    f"Ускорение: {speedup}")

        def minute_passed(d):
            return (time.time() - d.bb.get('start_time', time.time())) ≥
60.0

        return Sequence([
            EvalAction(lambda d: d.bb.set('start_time', time.time()) or "
   Старт на 60 сек", "Init"),
            RepeatUntil(
                child=EvalAction(check_compare, "Compare"),
                stop_condition=minute_passed,
                name="Сравнение 60 сек"
            ),
            EvalAction(lambda d: "   Минута прошла", "Finish"),
        ])


def main():
    rclpy.init()
    RobotDriver.spin(CompareMission())


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

Запуск:

```
python3 ~/ws/rbt/scripts/hailo_compare_mission.py
```

Ожидаемый вывод:

```
   CPU: 320ms (3 obj) | Hailo: 14ms (3 obj) | Ускорение: 23x
```

2.2. Построение графика сравнения

Запишите 10 замеров CPU и Hailo-8. Постройте столбчатую диаграмму в Excel или Python:

```
import matplotlib.pyplot as plt
plt.bar(['CPU', 'Hailo-8'], [320, 14])
plt.ylabel('Время инференса, мс')
plt.savefig('compare.png')
```

3. Уровень «Углублённый»: Latency vs FPS

(Что под капотом)

Почему миллисекунды решают.

3.1. Почему latency критична

FPS (Frames Per Second) — сколько кадров обрабатывается за секунду. Важно для видео.

Latency (задержка) — время от захвата кадра до получения результата. Критично для управления.

Пример:

- Робот едет 0.2 м/с.
- CPU: latency 320 мс → робот проехал **6.4 см** прежде чем узнал о препятствии.
- Hailo-8: latency 14 мс → робот проехал **0.28 см**.

Разница в 6 см может быть критичной при остановке перед человеком.

3.2. Расчёт тормозного пути

```
S = v × t_latency + v² / (2 × a)
где v = 0.2 м/с, a = 2 м/с²

CPU (t=0.32c): S = 0.2×0.32 + 0.04/4 = 0.064 + 0.01 = 7.4 см
Hailo (t=0.014c): S = 0.2×0.014 + 0.01 = 0.3 + 0.01 = 1.3 см
```

Hailo-8 останавливает робота в 5 раз быстрее.

4. Уровень «Исследовательский»: Энергопотребление

(Прорыв за границы)

Автономность.

4.1. Влияние на время работы от батареи

Вопрос: На сколько минут хватит батареи при непрерывном инференсе?

- CPU (RPi4): ~8 Вт при нагрузке. Аккумулятор 5000 мА·ч / 12 В = 60 Вт·ч. Время: 60 / 8 ≈ **7.5 часов**.

- CPU + Hailo-8: 8 + 3 = 11 Вт. Время: 60 / 11 ≈ **5.5 часов**.

Но: Hailo-8 делает инференс в 20 раз быстрее и может **спать** между кадрами. При 10 FPS и 14 мс/кадр он работает только 14% времени. Реальное потребление: 8 + 3×0.14 = **8.4 Вт** — почти как без Hailo.

5. Выводы по занятию

1. **Hailo-8 ускоряет YOLO в 20–30 раз** по сравнению с CPU Raspberry Pi.
2. **Latency (14 мс vs 320 мс)** — ключевой параметр для безопасности.
3. **Энергопотребление** Hailo-8 минимально при покадровом инференсе.

Связь со следующим занятием: Мы протестировали Hailo-8 отдельно. Теперь **интегрируем его в миссию на дереве поведения**: детектор работает в фоне, а дерево реагирует на появление человека.

Домашнее задание:

- **Базовое**: Запустите сравнение CPU vs Hailo-8. Запишите результаты.
- **Продвинутое**: Постройте график сравнения времени инференса.
- **Исследовательское**: Рассчитайте тормозной путь для скорости 0.5 м/с на CPU и Hailo-8.

Занятие 64: Интеграция детектора в миссию на дереве поведения

Содержание занятия: HailoDetectorProcessor как часть дерева. Робот детектирует объекты в реальном времени

Цели занятия

- **Образовательная**: Подключить `HailoDetectorProcessor` к миссии с деревом поведения. Научиться читать результаты детекций через `driver.is_person_detected()` в `Condition`.
- **Развивающая**: Понять архитектуру: процессор пишет в `Blackboard`, дерево читает и реагирует.
- **Воспитательная**: Показать, что нейросеть — это просто ещё один сенсор в дереве поведения.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия с Hailo-8 (45 мин)**
   - `HailoDetectorProcessor` как `add_node()`.
   - Чтение `driver.has_detections()` в `EvalAction`.
   - `RepeatUntil` на 60 секунд для сбора статистики.
   - HTTP-стрим с детекциями.
   - Робот стоит, детектирует, выводит в лог.

3. **Уровень «Углублённый»: Предобработка и постобработка (25 мин)**
   • Леттербокс, нормализация.
   • Декодирование выхода Hailo-8.
   • Задержка от кадра до результата.
4. **Уровень «Исследовательский»: Буферизация и разрешение (10 мин)**
   • Буфер кадров.
   • Влияние разрешения на точность.
5. **Подведение итогов (5 мин)**

1. Введение: Нейросеть как сенсор

Мы протестировали Hailo-8 отдельно. Теперь **интегрируем** его в миссию: процессор детектирует людей в фоне, а дерево поведения читает результаты через `Blackboard` и реагирует.

`HailoDetectorProcessor` — это такой же `ImageProcessorNode`, как `LineDetectorProcessor`. Он пишет результаты в `driver.set_detections()`, а дерево читает их через `driver.has_detections()`, `driver.is_person_detected()`.

2. Уровень «База»: Миссия с Hailo-8

(Инструмент, навык, интерфейс)

Обязательный уровень. Подключаем Hailo-8 к миссии на 60 секунд.

2.1. Архитектура

```
DetectorInMission (RobotDriver)
├── HailoDetectorProcessor (HailoProcessorNode) — пишет в bb: detections,
hailo_ms
├── HttpStreamerNode — стрим с Hailo
├── TreeDashboardNode — дашборд дерева
└── Дерево поведения:
    └── Sequence
        ├── EvalAction: запомнить время старта
        ├── RepeatUntil (60 сек)
        │   └── EvalAction: читает bb и выводит статистику
        └── EvalAction: финиш
```

Ключевой момент: процессоры и дерево работают **параллельно** в `MultiThreadedExecutor`. Процессоры пишут в `Blackboard`, дерево читает.

2.2. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/detector_in_mission.py
```

```
#!/usr/bin/env python3
"""Миссия: Hailo-8 + стрим + статистика в дереве 60 сек."""
import sys
import rclpy
import cv2
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        TreeDashboardNode, HailoProcessorNode,
                        HttpStreamerNode, RepeatUntil)


class HailoDetectorProcessor(HailoProcessorNode):
    """Детектор на Hailo-8 с парсингом NMS."""

    def __init__(self, driver, hef_path, fps=30):
        super().__init__(driver, hef_path, fps=fps,
                         node_name="hailo_detector")

    def handle_hailo_result(self, output, display, dt_ms):
        nms_layer = list(output.keys())[0]
        raw_data = output[nms_layer]
        detections_by_class = raw_data[0]
        img_h, img_w = display.shape[:2]
        pad_x, pad_y, scale = self._pad_info

        det_list = []
        count = 0

        for class_id in range(len(detections_by_class)):
            class_matrix = detections_by_class[class_id]
            if class_matrix is None or len(class_matrix) == 0:
                continue
            for j in range(len(class_matrix)):
                row = class_matrix[j]
                if row is None or len(row) < 5:
                    continue
                ymin, xmin, ymax, xmax, confidence = row[:5]
                if float(confidence) > 0.4:
                    confidence = float(confidence)
                    x_box = xmin * self._w_in
                    y_box = ymin * self._h_in
                    w_box = (xmax - xmin) * self._w_in
                    h_box = (ymax - ymin) * self._h_in
                    x = int((x_box - pad_x) / scale)
                    y = int((y_box - pad_y) / scale)
                    w = int(w_box / scale)
                    h = int(h_box / scale)
                    x = max(0, x)
                    y = max(0, y)
                    w = min(img_w - x, w)
                    h = min(img_h - y, h)
                    if w > 0 and h > 0:
                        count += 1
                        det_list.append([int(class_id), confidence, [x,
y, w, h]])
                        cv2.rectangle(display, (x, y), (x+w, y+h), (0,
```

```
255, 0), 2)

        self.driver.set_detections(det_list)
        self.driver.bb.update(hailo_ms=round(dt_ms, 1), hailo_count=count)

        cv2.putText(display, f"Hailo: {dt_ms:.0f}ms, {count} obj", (10,
30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('hailo_jpeg', jpeg.tobytes())


class DetectorInMission(RobotDriver):

    def __init__(self):
        super().__init__('detector_in_mission', tick_rate=2)
        self.add_node(TreeDashboardNode(self, port=8079))
        self.add_node(HttpStreamerNode(self.bb, key='hailo_jpeg', port=808
0))
        self.add_node(HailoDetectorProcessor(self, 'yolov11x.hef', fps=30)
)

    def build_tree(self):
        def check_detections(d):
            ms = d.bb.get('hailo_ms', 0)
            cnt = d.bb.get('hailo_count', 0)
            if d.has_detections():
                if d.is_person_detected():
                    return f"   Человек! {cnt} объектов, {ms} мс"
                classes = set(str(det[0]) for det in d.get_detections())
                return f"   {cnt} объектов: {', '.join(classes)}, {ms} мс"
            return f"   Объектов нет, {ms} мс"

        def minute_passed(d):
            elapsed = d.get_clock().now().nanoseconds / 1e9 - d.bb.get('st
art_time', 0)
            return elapsed >= 60.0

        return Sequence([
            EvalAction(lambda d: d.bb.set('start_time', d.get_clock().now(
).nanoseconds / 1e9) or "   Старт на 60 сек", "Init"),
            RepeatUntil(
                child=EvalAction(check_detections, "CheckDetections"),
                stop_condition=minute_passed,
                name="60 сек"
            ),
            EvalAction(lambda d: "   Минута прошла", "Finish"),
        ])


def main():
    rclpy.init()
    RobotDriver.spin(DetectorInMission())
```

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

Запуск:

```
python3 ~/ws/rbt/scripts/detector_in_mission.py
```

Стрим: `http://<ip>:8080/stream` — кадры с bounding boxes. **Дашборд:** `http://<ip>:8079` — состояние дерева.

2.3. Как это работает

1. `HailoDetectorProcessor` (30 FPS) получает кадр, прогоняет через Hailo-8, парсит NMS-вывод, сохраняет `det_list` через `driver.set_detections()`.
2. `RepeatUntil` в течение 60 секунд на каждом тике (2 Гц) читает `driver.has_detections()` и `driver.is_person_detected()`.
3. `EvalAction` выводит статистику в лог.
4. По истечении минуты миссия завершается.
5. `HttpStreamerNode` раздаёт обработанный JPEG.

Разделение частот: детектор — 30 FPS (быстро), дерево — 2 Гц (экономия CPU).

3. Уровень «Углублённый»: Предобработка и постобработка

(Что под капотом)

Что делает `HailoProcessorNode`.

3.1. Предобработка: леттербокс и нормализация

`HailoProcessorNode` автоматически:

1. Ресайзит кадр под входной размер модели с сохранением пропорций (леттербокс).
2. Добавляет серые полосы.
3. Преобразует BGR→RGB.
4. Нормализует в [0, 1].
5. Сохраняет `_pad_info` (pad_x, pad_y, scale) и `_w_in`, `_h_in`.

В `handle_hailo_result` нужно **обратно пересчитать** координаты из нормализованных в пиксели исходного кадра, используя `_pad_info`.

3.2. Постобработка: декодирование NMS

Hailo-8 возвращает структурированный массив: 80 классов, для каждого — матрица детекций `[ymin, xmin, ymax, xmax, confidence]`. Координаты нормализованы в [0, 1] относительно холста 640×640.

Формула пересчёта:

```
x_px = (x_norm * w_in - pad_x) / scale
y_px = (y_norm * h_in - pad_y) / scale
w_px = w_norm * w_in / scale
h_px = h_norm * h_in / scale
```

4. Уровень «Исследовательский»: Буферизация и разрешение

(Прорыв за границы)

Эксперименты.

4.1. Буферизация кадров

Если детектор не успевает за камерой (30 FPS), кадры теряются. Можно хранить **кольцевой буфер** последних N кадров и обрабатывать их с той скоростью, на которую способен NPU.

4.2. Мини-исследование: разрешение и точность

Вопрос: Как разрешение входного кадра влияет на точность детектирования?

Методика: сравните количество найденных объектов при ресайзе в 640×640, 320×320 и 160×160.

Ожидание: на 160×160 мелкие объекты теряются, но скорость растёт.

5. Выводы по занятию

1. **HailoDetectorProcessor** — `HailoProcessorNode`, работающий параллельно с деревом.
2. **Данные передаются через** `Blackboard`: процессор пишет `set_detections()`, дерево читает `has_detections()`.
3. `RepeatUntil` позволяет собирать статистику в течение заданного времени.
4. **Леттербокс и NMS** обрабатываются внутри `HailoProcessorNode`.

Связь со следующим занятием: Мы детектируем объекты. Теперь **реагируем** на них: робот едет и останавливается при обнаружении человека.

Домашнее задание:

- **Базовое**: Запустите миссию. Посмотрите стрим и дашборд.
- **Продвинутое**: Добавьте `CPUYOLOProcessor` и сравните с Hailo-8.
- **Исследовательское**: Сравните точность на разных разрешениях входного кадра.

Занятие 65: Реакция робота на детектирование

Содержание занятия: Движение с остановкой при обнаружении человека. Приоритеты в дереве поведения

Цели занятия

- **Образовательная**: Собрать миссию, где робот едет вперёд и останавливается при обнаружении человека через Hailo-8. Всё в одном дереве.
- **Развивающая**: Понять, как организовать передачу флагов через `Blackboard` и приоритеты через `RepeatUntil`.
- **Воспитательная**: Показать, что нейросеть — не просто демонстрация, а реальный инструмент безопасности.

План занятия (90 минут)

1. **Организационный момент и проверка ДЗ (5 мин)**
2. **Уровень «База»: Миссия StopOnPerson (45 мин)**
   - `HailoDetectorProcessor` пишет `person_detected` в `Blackboard`.
   - Дерево: `RepeatUntil` с `StreamAction` + `StopAction` + `WaitCondition`.
   - HTTP-стрим и дашборд.
3. **Уровень «Углублённый»: Приоритеты и зоны (25 мин)**
   - `Fallback` для выбора приоритета.
   - Составное условие: человек И близко.
4. **Уровень «Исследовательский»: Следование за человеком (10 мин)**
   - Потоковое управление по координатам.
   - Задержка и безопасная дистанция.
5. **Подведение итогов (5 мин)**

1. Введение: Нейросеть как стоп-сигнал

В занятии 64 мы детектировали объекты и выводили статистику. Теперь закроем петлю обратной связи: робот **едет** и **реагирует** на детекции.

Задача: робот движется вперёд. Если Hailo-8 видит человека — экстренная остановка. Когда человек уходит — продолжает движение.

Это классический паттерн безопасности в робототехнике.

2. Уровень «База»: Миссия StopOnPerson

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем миссию с остановкой.

2.1. Архитектура

```
HailoDetectorProcessor (30 FPS) → пишет person_detected в Blackboard
        ↓
Дерево поведения (10 Гц):
└── RepeatUntil (бесконечно)
    └── Sequence
        ├── StreamAction (едет вперёд, stop_condition = person_detected)
        ├── LogAction ("Человек!")
        ├── StopAction
        └── WaitCondition (ждёт person_gone)
```

Ключевой момент: `HailoDetectorProcessor` определяет `person_detected` по `class_id == 0` (person в COCO) и сохраняет флаг в `driver.bb.set('person_detected', ...)`. Дерево читает этот флаг через `d.bb.get('person_detected')`.

2.2. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/stop_on_person.py
```

```python
#!/usr/bin/env python3
"""Миссия: движение вперёд, остановка при обнаружении человека."""
import time
import sys
import rclpy
import cv2
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        StreamAction, StopAction, LogAction,
                        WaitCondition, RepeatUntil,
                        TreeDashboardNode, HailoProcessorNode,
                        HttpStreamerNode)


class HailoDetectorProcessor(HailoProcessorNode):
    """Детектор на Hailo-8."""

    def __init__(self, driver, hef_path, fps=30):
        super().__init__(driver, hef_path, fps=fps,
                         node_name="hailo_detector")

    def handle_hailo_result(self, output, display, dt_ms):
        nms_layer = list(output.keys())[0]
        raw_data = output[nms_layer]
        detections_by_class = raw_data[0]
        img_h, img_w = display.shape[:2]
        pad_x, pad_y, scale = self._pad_info

        det_list = []
        count = 0
        person_detected = False
```

```
        for class_id in range(len(detections_by_class)):
            class_matrix = detections_by_class[class_id]
            if class_matrix is None or len(class_matrix) == 0:
                continue
            for j in range(len(class_matrix)):
                row = class_matrix[j]
                if row is None or len(row) < 5:
                    continue
                ymin, xmin, ymax, xmax, confidence = row[:5]
                if float(confidence) > 0.4:
                    confidence = float(confidence)
                    x_box = xmin * self._w_in
                    y_box = ymin * self._h_in
                    w_box = (xmax - xmin) * self._w_in
                    h_box = (ymax - ymin) * self._h_in
                    x = int((x_box - pad_x) / scale)
                    y = int((y_box - pad_y) / scale)
                    w = int(w_box / scale)
                    h = int(h_box / scale)
                    x = max(0, x)
                    y = max(0, y)
                    w = min(img_w - x, w)
                    h = min(img_h - y, h)
                    if w > 0 and h > 0:
                        count += 1
                        det_list.append([int(class_id), confidence, [x,
y, w, h]])

                        cv2.rectangle(display, (x, y), (x+w, y+h), (0,
255, 0), 2)
                        if int(class_id) == 0:  # 0 = person в COCO
                            person_detected = True

        self.driver.set_detections(det_list)
        # Ключевое: пишем в bb для stop_condition
        self.driver.bb.set('person_detected', person_detected)

        status = "   ЧЕЛОВЕК!" if person_detected else "   OK"
        cv2.putText(display, f"{status} {dt_ms:.0f}ms", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7,
                    (0, 0, 255) if person_detected else (0, 255, 0), 2)

        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('hailo_jpeg', jpeg.tobytes())


class StopOnPerson(RobotDriver):

    def __init__(self):
        super().__init__('stop_on_person', tick_rate=10)
        self.add_node(TreeDashboardNode(self, port=8079))
        self.add_node(HttpStreamerNode(self.bb, key='hailo_jpeg', port=808
0))
        self.add_node(HailoDetectorProcessor(self, 'yolov11x.hef', fps=30)
)
```

```
    def build_tree(self):
        def drive_forward(d):
            return (0.15, 0.0, 0.0, 0.3)

        def person_detected(d):
            return d.bb.get('person_detected', False)

        def person_gone(d):
            return not d.bb.get('person_detected', True)

        return RepeatUntil(
            child=Sequence([
                StreamAction(
                    command_func=drive_forward,
                    stop_condition=person_detected,
                    name="Движение вперёд"
                ),
                LogAction("   Обнаружен человек! Остановка."),
                StopAction(name="Экстренный стоп"),
                WaitCondition(
                    check_func=person_gone,
                    timeout=300.0,
                    name="Жду ухода"
                ),
                LogAction("   Человек ушёл, продолжаю."),
            ]),
            stop_condition=lambda d: False,
            name="Патруль"
        )


def main():
    rclpy.init()
    RobotDriver.spin(StopOnPerson())


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

Запуск:

```
python3 ~/ws/rbt/scripts/stop_on_person.py
```

Стрим: `http://<ip>:8080/stream`. **Дашборд:** `http://<ip>:8079`.

2.3. Как это работает

1. `HailoDetectorProcessor` (30 FPS) детектирует, определяет `class_id == 0` как человека, сохраняет `person_detected` в `Blackboard`.
2. `StreamAction` (10 Гц) едет вперёд, на каждом тике проверяет `person_detected(d)` → `d.bb.get('person_detected')`.
3. Как только флаг стал `True` — `StreamAction` вызывает `stop()` и возвращает `SUCCESS`.

4. `StopAction` подтверждает остановку.
5. `WaitCondition` ждёт, пока `person_gone(d)` не станет `True` (флаг `person_detected` стал `False`).
6. `RepeatUntil` сбрасывает `Sequence`, робот снова едет.

Почему `person_detected` пишется в `bb`, а не через `is_person_detected()`? Потому что `stop_condition` в `StreamAction` вызывается 10 раз в секунду, а `set_detections()` может обновляться асинхронно. Прямой флаг в `bb` — самый быстрый способ.

3. Уровень «Углублённый»: Приоритеты и зоны

(Что под капотом)

Обработка нескольких условий.

3.1. Приоритеты через Fallback

Что если нужно реагировать и на человека, и на знак? `person_detected` — только один флаг. Можно завести несколько флагов и использовать `Fallback`:

```
Fallback([
    Sequence([
        Condition(lambda d: d.bb.get('person_detected')),
        StopAction(),
        LogAction("Человек!")
    ]),
    Sequence([
        Condition(lambda d: d.bb.get('stop_sign_detected')),
        StopAction(),
        LogAction("Знак стоп!")
    ]),
])
```

`Fallback` пробует первый вариант, если флаг не активен — второй.

3.2. Составное условие: человек И близко

Тормозить при появлении человека на горизонте — избыточно. Добавляем проверку лидара:

```
def person_close(d):
    return d.bb.get('person_detected', False) and d.get_lidar_front() < 1.
0
```

Это объединяет данные нейросети и лидара в одном условии.

4. Уровень «Исследовательский»: Следование за человеком

(Прорыв за границы)

Активное поведение.

4.1. Следование на безопасной дистанции

Вместо остановки можно **следовать** за человеком:

```
def follow_person(d):
    persons = d.get_detections_by_class("person")
    if persons:
        _, _, (x, y, w, h) = persons[0]
        vx = 0.1 if h < 300 else -0.1
        return (vx, 0.0, 0.0, 0.3)
    return (0.0, 0.0, 0.0, 0.3)
```

4.2. Мини-исследование: задержка и безопасная дистанция

Вопрос: Насколько нужно увеличить безопасную дистанцию с учётом задержки инференса?

Расчёт: при скорости 0.15 м/с и задержке 15 мс (Hailo-8) робот проезжает 2.25 мм. При задержке 300 мс (CPU) — 4.5 см. Разница в 20 раз.

Вывод: с Hailo-8 безопасная дистанция может быть минимальной.

5. Выводы по занятию

1. **Флаг `person_detected` в Blackboard** — быстрый способ передать состояние от процессора к дереву.
2. `RepeatUntil` позволяет роботу бесконечно патрулировать, останавливаясь перед людьми.
3. `Fallback` организует приоритеты между несколькими флагами.
4. **Составные условия** объединяют нейросеть и лидар.

Связь со следующим занятием: Мы интегрировали Hailo-8 в движение. На следующем занятии — **зачёт по PyTorch и Hailo-8**: комплексная миссия с линией, лидаром и детектором.

Домашнее задание:

- **Базовое**: Запустите миссию. Проверьте остановку перед человеком.
- **Продвинутое**: Добавьте условие «человек И лидар < 1.0 м».
- **Исследовательское**: Реализуйте следование за человеком.

Занятие 66: Зачёт по PyTorch и Hailo-8

Содержание занятия: Комплексная миссия — линия, Hailo-8, лидар. Три подсистемы в одном дереве

Цели занятия

- **Образовательная**: Собрать зачётную миссию, объединяющую три подсистемы: следование по линии, детектирование людей через Hailo-8, объезд препятствий по лидару.

- **Развивающая**: Понять, как `Parallel` и `Fallback` разрешают конфликты между подсистемами.
- **Воспитательная**: Провести зачёт — демонстрация комплексной миссии с анализом взаимодействия.

План занятия (90 минут)

1. **Организационный момент и регламент зачёта (5 мин)**
2. **Уровень «База»: Комплексная миссия ExamMission (50 мин)**
   - `LineDetectorProcessor` + `HailoDetectorProcessor`.
   - Дерево: `RepeatUntil` с `StreamAction` + `Fallback` (человек > препятствие > линия).
   - HTTP-стрим и дашборд.
3. **Уровень «Углублённый»: Взаимодействие подсистем (20 мин)**
   - Как `Blackboard` связывает процессоры и дерево.
   - `Parallel` и `Fallback` для разрешения конфликтов.
4. **Уровень «Исследовательский»: Робот-курьер (10 мин)**
   - Сценарий с ArUco-маркером.
   - Задержка от обнаружения до остановки.
5. **Подведение итогов блока (5 мин)**

1. Введение: Выпускной экзамен блока PyTorch и Hailo-8

Мы прошли 8 занятий по нейросетям:

- **Основы:** нейрон, CNN, PyTorch.
- **Детектирование:** YOLO на CPU и Hailo-8.
- **Интеграция:** `HailoProcessorNode`, `Blackboard`, дерево поведения.

Сегодня — **зачёт**. Вы соберёте миссию, где робот **следует по линии**, **останавливается перед людьми** (Hailo-8) и **объезжает статические препятствия** (лидар).

2. Уровень «База»: Комплексная миссия ExamMission

(Инструмент, навык, интерфейс)

Обязательный уровень. Собираем и запускаем.

2.1. Архитектура

```
ExamMission
├── LineDetectorProcessor (ImageProcessorNode) → line_offset в bb
├── HailoDetectorProcessor (HailoProcessorNode) → person_detected в bb
├── HttpStreamerNode → стрим
├── TreeDashboardNode → дашборд
└── Дерево:
    └── RepeatUntil (бесконечно)
```

```
└── Sequence
    ├── StreamAction (следует по линии,
    │   stop_condition = человек ИЛИ препятствие)
    ├── Fallback (реакция на остановку)
    │   ├── Sequence (человек → стоп → ждать)
    │   └── Sequence (препятствие → объезд)
    └── (продолжаем)
```

Приоритеты: человек > препятствие > линия. Если обнаружен человек — останавливаемся и ждём. Если препятствие — объезжаем. Иначе — следуем по линии.

2.2. Код миссии

Создайте файл:

```
nano ~/ws/rbt/scripts/exam_nn_mission.py
```

```
#!/usr/bin/env python3
"""Зачёт: линия + Hailo-8 + лидар в одном дереве."""
import math
import sys
import rclpy
import cv2
import numpy as np
from driver_lib import (RobotDriver, Sequence, EvalAction,
                        StreamAction, StopAction, LogAction,
                        WaitCondition, RepeatUntil, Fallback,
                        MoveRelativeAction,
                        TreeDashboardNode, HailoProcessorNode,
                        HttpStreamerNode, ImageProcessorNode)


# ═══ Детектор линии ═══
class LineDetectorProcessor(ImageProcessorNode):
    def process_frame(self, frame):
        h, w = frame.shape[:2]
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        binary = cv2.adaptiveThreshold(gray, 255,
            cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
        cleaned = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
        contours, _ = cv2.findContours(cleaned, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)
        if contours:
            best = max(contours, key=cv2.contourArea)
            if cv2.contourArea(best) > 500:
                M = cv2.moments(best)
                if M['m00'] > 0:
                    offset = int(M['m10']/M['m00']) - w//2
                    self.driver.set_line_info(offset, 0.0, detected=True)
                    return
        self.driver.set_line_info(0.0, 0.0, detected=False)


# ═══ Детектор Hailo-8 ═══
```

```
class HailoDetectorProcessor(HailoProcessorNode):
    def __init__(self, driver, hef_path, fps=30):
        super().__init__(driver, hef_path, fps=fps,
                         node_name="hailo_detector")

    def handle_hailo_result(self, output, display, dt_ms):
        nms_layer = list(output.keys())[0]
        raw_data = output[nms_layer]
        detections_by_class = raw_data[0]
        img_h, img_w = display.shape[:2]
        pad_x, pad_y, scale = self._pad_info
        det_list = []
        person_detected = False
        for class_id in range(len(detections_by_class)):
            class_matrix = detections_by_class[class_id]
            if class_matrix is None or len(class_matrix) == 0:
                continue
            for j in range(len(class_matrix)):
                row = class_matrix[j]
                if row is None or len(row) < 5:
                    continue
                ymin, xmin, ymax, xmax, conf = row[:5]
                if float(conf) > 0.4:
                    conf = float(conf)
                    x = int((xmin*self._w_in - pad_x) / scale)
                    y = int((ymin*self._h_in - pad_y) / scale)
                    w = int(((xmax-xmin)*self._w_in) / scale)
                    h = int(((ymax-ymin)*self._h_in) / scale)
                    x, y = max(0,x), max(0,y)
                    w, h = min(img_w-x,w), min(img_h-y,h)
                    if w>0 and h>0:
                        det_list.append([int(class_id), conf, [x,y,w,h]])
                        if int(class_id) == 0:
                            person_detected = True
        self.driver.set_detections(det_list)
        self.driver.bb.set('person_detected', person_detected)
        cv2.putText(display, f"{'  ' if person_detected else '  '}
{dt_ms:.0f}ms",
                    (10,30), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
                    (0,0,255) if person_detected else (0,255,0), 2)
        _, jpeg = cv2.imencode('.jpg', display,
[cv2.IMWRITE_JPEG_QUALITY, 80])
        self.driver.bb.set('exam_jpeg', jpeg.tobytes())


# ═══ Миссия ═══
class ExamMission(RobotDriver):
    def __init__(self):
        super().__init__('exam_mission', tick_rate=10)
        self.add_node(TreeDashboardNode(self, port=8079))
        self.add_node(HttpStreamerNode(self.bb, key='exam_jpeg',
port=8080))
        self.add_node(LineDetectorProcessor(self, fps=30))
        self.add_node(HailoDetectorProcessor(self, 'yolov11x.hef', fps=30)
)
```

```
    def build_tree(self):
        def drive_with_line(d):
            offset, _, detected = d.get_line_position()
            vx = 0.1
            omega = -offset / 200.0 if detected else 0.0
            omega = max(-0.3, min(0.3, omega))
            return (vx, 0.0, omega, 0.3)

        def should_stop(d):
            return d.bb.get('person_detected', False) or
d.get_lidar_front() < 0.3

        def is_person(d):
            return d.bb.get('person_detected', False)

        def is_obstacle(d):
            return d.get_lidar_front() < 0.3

        def person_gone(d):
            return not d.bb.get('person_detected', True)

        return RepeatUntil(
            child=Sequence([
                StreamAction(
                    command_func=drive_with_line,
                    stop_condition=should_stop,
                    name="Следование по линии"
                ),
                Fallback([
                    # Приоритет 1: человек
                    Sequence([
                        EvalAction(lambda d: "   Человек! Жду ... ", "Person
"),
                        StopAction(name="Стоп"),
                        WaitCondition(person_gone, timeout=300.0, name="Жд
у ухода"),
                        LogAction("   Ушёл"),
                    ]),
                    # Приоритет 2: препятствие
                    Sequence([
                        LogAction("   Препятствие! Объезд"),
                        MoveRelativeAction(dx=-0.2, name="Назад"),
                        MoveRelativeAction(dy=0.3, name="Влево"),
                        MoveRelativeAction(dx=0.4, name="Вперёд"),
                        MoveRelativeAction(dy=-0.3, name="Вправо"),
                        LogAction("   Объехал"),
                    ]),
                ]),
            ]),
            stop_condition=lambda d: False,
            name="Главный цикл"
        )


def main():
    rclpy.init()
```

```
    RobotDriver.spin(ExamMission())

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

Запуск:

```
python3 ~/ws/rbt/scripts/exam_nn_mission.py
```

2.3. Критерии зачёта

1. Робот следует по линии.
2. При обнаружении человека — остановка и ожидание.
3. При статическом препятствии — объезд.
4. После ухода человека / объезда — продолжение движения.
5. Ученик может объяснить, как `Fallback` выбирает приоритет.

3. Уровень «Углублённый»: Взаимодействие подсистем

(Что под капотом)

Как три подсистемы уживаются в одном дереве.

3.1. Blackboard как шина данных

```
LineDetectorProcessor → set_line_info() → bb['line_offset']
HailoDetectorProcessor → bb['person_detected']
Лидар (встроен в driver) → get_lidar_front()
        ↓
Дерево читает все три источника в should_stop():
  person_detected OR front < 0.3
```

Правило: процессоры **пишут** в `Blackboard` (или через `set_*` методы). Дерево **читает**.

3.2. Fallback для разрешения конфликтов

Что если одновременно и человек, и препятствие? `Fallback` пробует **первый** вариант:

1. `Sequence` с человеком — если `person_detected == True`, выполнится и вернёт `SUCCESS`.
2. Если `person_detected == False`, `Condition` неявно провалится? Нет, в нашей реализации `EvalAction` всегда `SUCCESS`. Поэтому `Fallback` всегда выберет первый вариант, если человек обнаружен.

Улучшение: добавить `Condition(is_person)` перед `StopAction`, чтобы при отсутствии человека первый `Sequence` возвращал `FAILURE` и `Fallback` переходил ко второму.

4. Уровень «Исследовательский»: Робот-курьер

(Прорыв за границы)

Сценарий с ArUco-маркером.

4.1. Доставка к маркеру с остановками перед людьми

Добавьте `ArucoDetectorProcessor` и словарь `marker_actions`. Робот следует по линии, при маркере id=42 выполняет действие, но **перед каждым действием** проверяет `person_detected` и ждёт.

4.2. Мини-исследование: суммарная задержка

Вопрос: Сколько времени проходит от появления человека в кадре до остановки робота?

Цепочка: кадр → Hailo-8 (15 мс) → `set_detections` → `bb.set` → следующий тик дерева (100 мс при 10 Гц) → `StreamAction.stop_condition` → `stop()`.

Суммарно: ~115 мс. При 0.1 м/с робот проедет ~1.2 см — безопасно.

5. Подведение итогов блока «PyTorch и Hailo-8»

Чему мы научились (занятия 59–66)

1. **Нейросети:** нейрон, CNN, PyTorch, YOLO.
2. **Hailo-8:** архитектура NPU, конвертация модели, инференс.
3. **Интеграция:** `HailoProcessorNode`, `Blackboard`, дерево поведения, приоритеты.

Что дальше

Вы завершили этот лабораторный практикум. Теперь у вас есть фундамент: вы умеете управлять роботом, обрабатывать данные с сенсоров, строить деревья поведения, запускать нейросети на NPU. Этих знаний достаточно, чтобы спроектировать и реализовать собственную автономную миссию.

Но самое главное — вы научились учиться на реальном железе: подключаться, запускать, ошибаться, анализировать, отлаживать, снова запускать. Этот навык останется с вами независимо от того, с каким роботом или технологией вы столкнётесь в будущем.

Следующее учебное пособие будет посвящено нейросетям — глубокому обучению для задач восприятия и управления. Вы перейдёте от использования готовых моделей (YOLO) к самостоятельному обучению нейросетей: датасеты, тренировка, fine-tuning под конкретные задачи робота.

А пока — творите свои миссии. Робот ждёт ваших команд.

Домашнее задание:

- **Базовое**: Запустите зачётную миссию.

• **Продвинутое**: Добавьте `Condition` перед `StopAction` для корректного `Fallback`.

• **Исследовательское**: Измерьте суммарную задержку от кадра до остановки.

Приложение: Работа с RViz2

Тема: Визуализация данных робота в реальном времени

О чём это приложение

RViz2 — стандартный инструмент визуализации в ROS2. Он позволяет **видеть** данные с датчиков, положение робота, карту, видеопоток с камеры и многое другое. Для отладки автономных систем это незаменимо.

В этом приложении мы разберём:

- Как запустить RViz2 для нашего робота.
- Как добавить дисплеи: `LaserScan`, `RobotModel`, `Odometry`, `TF`, `Map`, `Image`.
- Как построить карту через SLAM.
- Как сохранить и загрузить конфигурацию.
- Полезные приёмы для отладки.

Важно: вся работа в RViz ведётся во фрейме `odom` — системе координат, связанной с одометрией. Это позволяет видеть траекторию робота относительно стартовой точки.

1. Запуск RViz2

Предварительно: учитель уже запустил основной стек робота:

```
ros2 launch mecanum_bringup bridge_ekf_lidar.launch.py
```

Запуск RViz2 (на компьютере с графическим интерфейсом, проекция на доску):

```
rviz2
```

Откроется окно с серым фоном и сеткой. В левой части — панель **Displays** (список слоёв визуализации). В центре — 3D-вид.

Первое, что нужно сделать: задать **Fixed Frame** (опорную систему координат). В панели `Displays` → `Global Options` → `Fixed Frame` введите `odom`. Это базовая система отсчёта, связанная с одометрией робота.

Почему `odom`, а не `map`:

- `odom` — система координат, которая дрейфует вместе с накоплением ошибки одометрии. Но она **всегда доступна** без SLAM и карты.
- `map` — глобальная система координат, требует запущенного SLAM и карты. Для наших учебных задач `odom` достаточно.

- Траектория робота во фрейме `odom` показывает, как робот «думает», что он движется. Это идеально для отладки.

2. Визуализация LaserScan (лидар)

Добавим слой для отображения данных лидара.

1. Нажмите кнопку **Add** в нижней части панели Displays.
2. В открывшемся окне выберите вкладку **By topic**.
3. Найдите топик `/scan` типа `sensor_msgs/msg/LaserScan`.
4. Нажмите **OK**.

Вы увидите **облако точек** вокруг робота:

- Красные, оранжевые, жёлтые точки — это препятствия.
- Цвет зависит от интенсивности отражённого сигнала.
- Расстояние до точек соответствует реальным стенам, людям, предметам.

Проверка: подойдите к роботу с разных сторон. Точки в RViz должны обновляться — лидар «видит» вас.

Настройки слоя LaserScan (в панели Displays, раскройте список):

Параметр	Значение	Что меняет
`Size (m)`	0.03	Размер точки
`Style`	Flat Squares / Points	Вид точек
`Decay Time`	0	Через сколько сек гаснут старые точки (0 — не гаснут)
`Color Transformer`	Intensity	Цвет по интенсивности сигнала

Полезный приём: если точек слишком много и RViz тормозит, уменьшите `Size` или включите `Decay Time = 1.0`.

3. Визуализация RobotModel (3D-модель робота)

Чтобы видеть сам робот в RViz, добавим его модель.

1. Нажмите **Add** → вкладка **By display type**.
2. Выберите `RobotModel`.
3. Нажмите **OK**.

Теперь в центре облака точек появится 3D-модель робота:

- Видно шасси, колёса, лидар, камеру.
- Модель движется вместе с реальным роботом.
- Это помогает понять, где робот и куда он «смотрит».

Настройки RobotModel:

Параметр	Значение
`Description Topic`	`/robot_description`
`Visual Enabled`	`true` (показывать модель)
`Collision Enabled`	`false` (не показывать коллизионные объёмы)

Если модель не отображается: проверьте, что топик `/robot_description` публикуется:

```
ros2 topic info /robot_description
```

Он публикуется один раз при запуске (`latched`), поэтому `ros2 topic echo` может не показать данные — это нормально. RViz читает latched-топики автоматически.

4. Визуализация одометрии

Одометрия показывает, как робот «думает», что он движется. Это полезно для отладки накопления ошибки.

1. Нажмите **Add** → вкладка **By topic**.
2. Найдите топик `/odom` (или `/odom_fused` — одометрия, объединённая с IMU через EKF).
3. Нажмите **OK**.

Появится слой `Odometry` . Он рисует **стрелки** — направление и положение робота по данным одометрии.

Полезное сравнение для отладки:

- Запустите проезд квадрата.
- В RViz смотрите на слой `Odometry` .
- Когда робот физически вернулся в начало (рулетка), одометрия может показывать смещение — это и есть накопленная ошибка.
- Траектория во фрейме `odom` визуализирует эту ошибку: робот «думает», что он в начале, а на самом деле смещён.

Настройки Odometry:

Параметр	Значение	
`Keep`	100	Сколько последних положений показывать
`Length`	0.3	Длина стрелки направления
`Color`	Голубой	Цвет траектории

5. Визуализация TF (систем координат)

TF (Transform) показывает связь между системами координат: `odom` → `base_link` → `lidar_link`.

1. Нажмите **Add** → вкладка **By display type**.
2. Выберите `TF`.
3. Нажмите **OK**.

Вы увидите **цветные оси координат** для каждого фрейма:

- Красная ось — X (вперёд).
- Зелёная ось — Y (влево).
- Синяя ось — Z (вверх).

Это помогает понять:

- Где находится лидар относительно центра робота.
- Куда направлена камера.
- Правильно ли откалиброваны датчики.

Фильтр фреймов: если осей слишком много, в настройках `TF` можно включить `Only Show Enabled Frames` и отметить только нужные:

- `odom` — система одометрии.
- `base_link` — центр робота.
- `lidar_link` — положение лидара.

Остальные фреймы (колёса, IMU) можно скрыть для чистоты.

6. Визуализация видеопотока с камеры (Image)

RViz может показывать видеопоток с камеры робота в отдельном окне.

1. Нажмите **Add** → вкладка **By topic**.
2. Найдите топик `/camera/image_raw` типа `sensor_msgs/msg/Image`.
3. Нажмите **OK**.

В нижней части RViz появится окно с видеопотоком. Вы увидите то, что «видит» робот.

Настройки Image:

Параметр	Значение
`Topic`	`/camera/image_raw`
`Transport Hint`	`raw` (без сжатия)

Полезно: можно одновременно видеть 3D-сцену (лидар, модель) и изображение с камеры. Это помогает сопоставить облако точек с реальной картинкой.

Если изображения нет:

1. Проверьте топик: `ros2 topic hz /camera/image_raw`.
2. Убедитесь, что камера подключена и драйвер запущен (в составе launch-файла).

7. Визуализация карты

RViz может показывать карту

В RViz:

1. Нажмите **Add** → вкладка **By topic**.
2. Найдите топик `/map` типа `nav_msgs/msg/OccupancyGrid`.
3. Нажмите **OK**.

8. Сохранение и загрузка конфигурации RViz

Чтобы не настраивать все слои заново при каждом запуске:

Сохранить конфигурацию:

1. Настройте все нужные дисплеи (LaserScan, RobotModel, Odometry, TF, Image, Map).
2. File → Save Config As → `~/ws/rbt/config/my_rviz.rviz`.

Загрузить конфигурацию при запуске:

```
rviz2 -d ~/ws/rbt/config/my_rviz.rviz
```

Полезно: создайте конфигурации для разных задач:

- `navigation.rviz` — LaserScan + RobotModel + Map.
- `debug_odom.rviz` — Odometry + TF.
- `camera.rviz` — Image (видеопоток с камеры).

9. Полезные приёмы

9.1. Измерение расстояния

Кнопка **Measure** (линейка в верхней панели) позволяет измерить расстояние между двумя точками прямо в RViz. Полезно для проверки: «правильно ли лидар видит стену на расстоянии 1 метр?».

9.2. 2D вид сверху

В правой части окна нажмите кнопку **TopDownOrtho** (вид сверху). Это удобно для навигации — видно облако точек, карту и робота как на чертеже.

9.3. Следование камеры за роботом

В панели `Views` (справа) выберите `ThirdPersonFollower` и укажите `Target Frame = base_link`. Камера будет следовать за роботом.

9.4. Публикация цели для навигации (если настроен Nav2)

Кнопка **2D Goal Pose** (флажок в верхней панели) позволяет указать точку на карте, куда робот должен приехать. Работает только с настроенным Nav2.

9.5. Отладка зависаний

Если RViz перестал обновляться:

1. Проверьте `Fixed Frame` — должен быть `odom`.
2. Проверьте частоту топика: `ros2 topic hz /scan`.
3. Уменьшите `Decay Time` у LaserScan, чтобы старые точки не накапливались.
4. Проверьте, что топик `/tf` публикуется: `ros2 topic hz /tf`.

10. Типичные проблемы и решения

Проблема	Причина	Решение
Пустой экран, нет точек	Неправильный `Fixed Frame`	Установите `odom`
Робот не отображается	Нет `/robot_description`	Проверьте launch-файл
Точки лидара «уплывают»	Нет трансформации `odom` → `base_link`	Проверьте EKF-узел (`/odom_fused`)
RViz тормозит	Слишком много точек, большое `Decay Time`	Уменьшите `Size`, включите `Decay Time = 0.5`
TF показывает ошибку	Нет связи между фреймами	Проверьте `ros2 run tf2_tools view_frames`
Карта не обновляется	SLAM не запущен или робот стоит	Запустите `slam_toolbox`, подвигайте робота
Нет изображения с камеры	Камера не подключена или драйвер не запущен	Проверьте `ros2 topic hz /camera/image_raw`

Главное правило отладки в RViz: если слой не отображается — проверьте, публикуется ли соответствующий топик:

```
ros2 topic list | grep -E "scan|odom|tf|map|image"
```

Если топик есть, но слой пуст — проверьте `Fixed Frame`.