УЧЕБНОЕ ПОСОБИЕ ROS 1 • PX4 • GAZEBO



Клевер в Gazebo

Программирование квадрокоптера на ROS: Blockly и Python

От основ Python и ROS к автономной миссии в симуляторе

Сергей Андреевич Корягин

Редакция 3 • 2026

ROS 1 Noetic • PX4 SITL • Gazebo Classic
Клевер 4; перенос на Клевер 5

Фото на обложке: COEX, Клевер 5 КОД, документация COEX, CC BY-NC-SA 4.0.

Предисловие

Книга посвящена программированию автономного квадрокоптера Клевер (COEX) на основе Robot Operating System (ROS) и отработке алгоритмов в симуляторе Gazebo. Изложение идёт от основ к прикладным задачам: устройство системы и назначение ROS; терминал и язык Python; обмен данными между программами; системы координат; программирование полёта в Blockly и Python; маршруты и их математика; моделирование среды; навигация по маркерам ArUco; обработка изображений; планирование и анализ эксперимента; отладка. Предварительные знания Python и Linux не предполагаются: им посвящены отдельные главы. Все полётные примеры рассчитаны на **один дрон в симуляторе**.

Книга не является официальным изданием COEX. Команды и имена интерфейсов сверены с публичной документацией и исходниками Clover; объяснения, алгоритмы и задания разработаны автором. Полётные программы не выполнялись в работающем Gazebo: объём проведённых проверок указан в главе «Источники и атрибуция» и в файле `output/verification.txt`.

Структура глав

Каждая глава начинается с перечня целей и необходимых знаний и заканчивается итогами с вопросами для самопроверки и ответами. После каждого теоретического раздела приведено задание с критерием проверки. Рабочая тетрадь (глава 29) содержит дополнительные задачи по всем темам.

Порядок изучения

Последовательное чтение рекомендуется: каждая глава опирается на предыдущие. Читателю, знакомому с Python, можно пропустить главу 6; читателю, уже настроившему стенд, – главы 5 и 8, выполнив из последней только приёмку. Для аппаратного Клевера 5 сначала следует прочитать главу 28.

Материалы проекта

Вместе с книгой поставляются исходник `textbook.tex`, каталог `examples` с программами и моделями SDF, иллюстрации с атрибуцией и сценарий сборки; описание – в `README.md`. Код листингов запускается из одноимённых файлов каталога примеров; модуль `flight_tools.py` должен находиться рядом с полётными программами.

Обозначения и единицы

Рамка «Цели главы» – результаты обучения; «Задание» – действие с проверяемым результатом; «Практика» и «Лабораторная работа» – расширенные опыты; рамка с тёплым цветом указывает ограничение версии или условие эксперимента; «Итоги» – резюме и самопроверка. В ROS используются метры, секунды, м/с и радианы; интерфейс Blockly может показывать градусы. Команды shell выполняются в **терминале Ubuntu внутри стенда**, а

не в PowerShell Windows.

Иллюстрации и лицензия

Снимки COEX могут отличаться от установленного образа. Источники указаны в подписях и в `assets/sources.json`. © Сергей Андреевич Корягин, 2026. Книга распространяется по лицензии CC BY-NC-SA 4.0: при некоммерческом распространении сохраняются авторство и лицензия. Условия использования программ – в `examples/LICENSE.txt`.

Оглавление

Предисловие **i**

1 Введение: предмет книги **1**
1.1 Предмет и итоговая задача . . . 1
1.2 Назначение симулятора . . . 2
1.3 Две вычислительные части аппарата . . . 2
1.4 Структура книги . . . 3
1.5 Основные термины . . . 3

2 Что именно мы моделируем **5**
2.1 Слой за слоем: из чего сложен виртуальный дрон . . . 5
2.2 Какую версию мы изучаем и почему . . . 6
2.3 Что можно исследовать . . . 7

3 Зачем роботу ROS **9**
3.1 Начнём с задачи, а не с определения . . . 9
3.2 Как это устроено на настоящем Клевере . . . 10
3.3 Какие трудности ROS снимает на примере Клевера . . . 12
3.4 Преимущества ROS: практическая польза . . . 12
3.5 Что ROS сам по себе не решает . . . 13
3.6 ROS 1 и ROS 2: о какой версии речь . . . 13

4 Терминал, файлы и запуск программ **15**
4.1 Три места, которые легко перепутать . . . 15
4.2 Узнайте, где вы находитесь . . . 15
4.3 Как появляется программа . . . 16
4.4 Окружение ROS . . . 16
4.5 Один опыт – несколько терминалов . . . 17
4.6 Читайте ошибку снизу вверх . . . 18
4.7 Рабочая привычка: версия перед опытом . . . 19

5 Настройка рабочей среды **20**
5.1 Что нужно установить . . . 20
5.2 Два способа получить стенд . . . 20
5.3 Путь A: виртуальная машина COEX . . . 21
5.4 Путь B: ручная сборка на Ubuntu 20.04 . . . 22
5.5 Сохраните паспорт окружения . . . 23

6 Python с нуля: язык вашей миссии **25**
6.1 Данные, команда и программа . . . 25
6.2 Арифметика: пять действий, которые понадобятся . . . 26
6.3 Строки, логические значения и единицы . . . 27
6.4 Условия: программа выбирает действие . . . 27
6.5 Списки и цикл for . . . 29
6.6 Цикл while и остановка ожидания . . . 30
6.7 Функции: один алгоритм, разные значения . . . 30
6.8 Модули: подключаем готовую работу . . . 31
6.9 Объект и атрибут без сложной теории . . . 31
6.10 Ошибки и завершение: try, except, finally . . . 31
6.11 Как переводить блоки в Python . . . 32

7 Лаборатория ROS: связываем программы 34
7.1 Цель и условия работы . . . 34
7.2 Нода – программа с одной ролью . . . 35
7.3 ROS master – каталог участников . . . 35
7.4 Топик – именованный поток . . . 36
7.5 Первый издатель: искусственная высота . . . 36
7.6 Подписчик и callback . . . 38
7.7 Третья нода: скользящее среднее . . . 38
7.8 Параметр – настройка поведения . . . 40
7.9 Сервис – один запрос и один ответ . . . 40
7.10 Тип сообщения – договор о данных . . . 41
7.11 Имена, namespace и переназначение . . . 42
7.12 Пакет и launch: запускаем систему вместе . . . 43
7.13 Когда нужны топик, сервис и параметр . . . 44

8 Первый запуск и приёмка стенда 46
8.1 Запускаем весь стек . . . 46
8.2 Сначала проверьте связь . . . 47
8.3 Камера и оценка позиции . . . 48
8.4 Рабочий цикл одного опыта . . . 49

9 Конфигурация Клевера: clover.launch и aruco.launch 51
9.1 Откуда берётся поведение Клевера . . . 51
9.2 Где лежат файлы . . . 51
9.3 Как работают флаги launch-файла . . . 52
9.4 Подключение к дрону и редактирование файла . . . 52
9.5 Флаги clover.launch . . . 53
9.6 Флаги aruco.launch . . . 55
9.7 Типовые наборы флагов . . . 56
9.8 Как это соотносится с симулятором книги . . . 56

10 ROS и координаты без путаницы 58
10.1 Зачем нужна система координат . . . 58
10.2 Одна тройка чисел – разные цели . . . 58
10.3 Соглашения об осях: FLU, ENU и камера . . . 60
10.4 TF и время . . . 60

11 Первые миссии в Blockly 62
11.1 Открываем интерфейс . . . 62
11.2 Миссия A: взлёт – шаг – посадка . . . 63
11.3 Миссия B: квадрат с циклом . . . 65
11.4 Состояние, условия и LED . . . 66

12 Blockly: мастерская алгоритмов 67
12.1 От цепочки к алгоритму . . . 67
12.2 Миссия C. Высота как переменная . . . 67
12.3 Миссия D. Три ступени высоты . . . 68
12.4 Миссия E. «Туда и обратно» с циклом . . . 68
12.5 Миссия F. Треугольник без поворотов . . . 69
12.6 Миссия G. «Змейка» для осмотра . . . 69
12.7 Миссия H. Решение по измеренной высоте . . . 70
12.8 Миссия I. Одна карта – несколько станций . . . 70
12.9 Пауза и ожидание прибытия: контрольный опыт . . . 71
12.10 Как читать сгенерированный Python . . . 71

13 Python: надёжная первая программа 72
13.1 Управление Клевером: четыре сервиса . . . 72

13.2 Из чего состоит ROS-программа . . . 73
13.3 Первая миссия . . . 74
13.4 Почему ожидание важнее паузы . . . 74
13.5 Что этот модуль контролирует . . . 77

14 Разбираем полётную программу по шагам 79
14.1 Сначала наблюдаем . . . 79
14.2 От сервиса к помощнику Flight . . . 80
14.3 Приняли команду – ещё не прибыли . . . 80
14.4 Зачем нужен таймаут . . . 81
14.5 Сохраняем точку дома . . . 82
14.6 Контекст mission и посадка . . . 82
14.7 Создайте свою короткую миссию . . . 82
14.8 Проверка входных данных . . . 82

15 Маршруты, циклы и траектории 84
15.1 Квадрат в фиксированной системе . . . 85
15.2 Обследование «змейкой» . . . 85
15.3 Плавная окружность . . . 86

16 Миссии: фигуры, осмотр и доставка 88
16.1 Как работать с коллекцией . . . 88
16.2 Треугольник вокруг дома . . . 88
16.3 Змейка: маршрут осмотра поля . . . 89
16.4 Три высоты: исследуем вертикальный переход . . . 90
16.5 Курьер: список станций и события . . . 90
16.6 Квадрат с настройками без редактирования файла . . . 91
16.7 Сначала видим путь, потом летим . . . 92
16.8 Таблица проверки каждой миссии . . . 92

17 Математика движения и обратная связь 94
17.1 Расстояние: зачем нужен квадратный корень . . . 94
17.2 Градусы и радианы . . . 94
17.3 Как повернуть смещение body в map . . . 95
17.4 Генератор правильного многоугольника . . . 95
17.5 Разомкнутый и замкнутый алгоритм . . . 96
17.6 Пропорциональная идея и ограничение скорости . . . 97
17.7 Точность и допуск . . . 97

18 Настраиваем Gazebo под свою задачу 99
18.1 Launch, мир и модель – разные уровни . . . 99
18.2 Проверенные переключатели . . . 99
18.3 Локальный launcher для книги . . . 100
18.4 Собственный мир с сохранением физики . . . 101
18.5 Камера, масса и инерция . . . 102
18.6 Производительность . . . 103

19 Gazebo как лабораторный стенд 104
19.1 Что именно моделирует симулятор . . . 104
19.2 Gazebo, RViz и QGroundControl . . . 104
19.3 Сценарий 1. Чистая площадка для первых миссий . . . 106
19.4 Сценарий 2. Учебный склад . . . 106
19.5 Сценарий 3. Коридор вокруг коробки . . . 106
19.6 SDF: прочитайте свою модель . . . 107
19.7 Сценарий 4. Проверка камеры . . . 107
19.8 Сценарий 5. Нагрузка и реальное время . . . 108
19.9 Что можно изучать в этой книге . . . 109

19.10 Паспорт воспроизводимой сцены . . . 109

20 Навигация по ArUco 110
20.1 Что такое ArUco и зачем он нужен . . . 110
20.2 Как это устроено в Клевере . . . 111
20.3 Подготовка маркеров: размер, печать, наклейка . . . 112
20.4 Карта маркеров: формат файла . . . 113
20.5 Генерация карты командой genmap.py . . . 114
20.6 Как «зашить» карту в дрон . . . 114
20.7 Карта в симуляторе: согласованное поле . . . 117
20.8 Полёты по маркерам . . . 117
20.9 Если маркеры не распознаются или позиция «плывёт» . . . 119

21 Компьютерное зрение: от кадра к решению 122
21.1 Изображение как набор чисел . . . 122
21.2 Сначала детектор, потом управление . . . 122
21.3 Изображение ROS и OpenCV . . . 123
21.4 Полный детектор красной области . . . 123
21.5 Как устроено измерение . . . 124
21.6 Задержка кадра и команды полёта . . . 125

22 Камера: фотография, пиксели и качество детектора 127
22.1 Первый результат – сохранённый кадр . . . 127
22.2 Как устроены пиксели . . . 128
22.3 Офлайн-опыт: размер и средний цвет . . . 128
22.4 Пороговая маска . . . 129
22.5 Данные для проверки, а не один удачный кадр . . . 130
22.6 Как оценить результат . . . 130
22.7 От камеры к управлению: что ещё потребуется . . . 130

23 Телеметрия, логи и воспроизводимый опыт 133
23.1 Записывайте результат, а не впечатление . . . 133
23.2 Какие ошибки считать . . . 134
23.3 rosbag: запись сообщений . . . 135
23.4 Серия опытов . . . 136

24 Как искать ошибки и строить устойчивую миссию 137
24.1 Симптом, причина и проверка . . . 137
24.2 Проверяйте систему слоями . . . 137
24.3 Опыты с искусственными отказами . . . 138
24.4 Состояния миссии . . . 138
24.5 Сторож времени и свежесть данных . . . 139
24.6 Очередь и тяжёлая обработка . . . 140
24.7 Почему несколько миссий мешают друг другу . . . 140
24.8 Дневник отладки . . . 140

25 Если реальный дрон ведёт себя не так 142
25.1 Реальный дрон – не симулятор . . . 142
25.2 Предполётная проверка . . . 143
25.3 Главный приём диагностики: разделяй ручное и автономное управление . . . 144
25.4 Вибрации и тряска . . . 144
25.5 Таблица «симптом – причина – проверка» . . . 147
25.6 Алгоритм действий при нештатном поведении . . . 148

26 Проекты для задач с Клевером 150
26.1 Что считать завершённым проектом . . . 150
26.2 Автомат состояний миссии . . . 151

27 Практикум профессионального уровня **153**
27.1 Как устроены профессиональные состязания . . . 153
27.2 Как решать конкурсное задание . . . 154
27.3 Автосудья: тренируйтесь с известным счётом . . . 156
27.4 Модуль 1. Подготовка и диагностика аппарата . . . 158
27.5 Модуль 2. Обход маркеров с чтением QR-кода . . . 159
27.6 Модуль 3. Цветные зоны . . . 160
27.7 Модуль 4. Работа с отказом . . . 161
27.8 Модуль 5. Комплексная задача . . . 161
27.9 Чек-лист участника и типичные ошибки . . . 161

28 Клевер 4 и Клевер 5: перенос навыков **163**
28.1 Что подтверждено документацией . . . 163
28.2 Что переносится сразу, а что проверяется . . . 164
28.3 Проверка имеющегося комплекта . . . 164
28.4 Практичный путь переноса . . . 165

29 Рабочая тетрадь и подсказки к решениям **166**
29.1 Как выполнять задания . . . 166
29.2 Блок A. Архитектура и ROS . . . 166
29.3 Блок B. Python . . . 166
29.4 Блок C. Координаты . . . 167
29.5 Блок D. Логика миссии . . . 167
29.6 Блок E. Симулятор и эксперимент . . . 167
29.7 Блок F. Камера . . . 167
29.8 Подсказки и ответы: архитектура . . . 168
29.9 Подсказки и ответы: Python . . . 168
29.10 Подсказки и ответы: координаты . . . 168
29.11 Подсказки и ответы: миссия . . . 168
29.12 Подсказки и ответы: симулятор . . . 169
29.13 Подсказки и ответы: камера . . . 169
29.14 Итоговое задание: собственный учебный полигон . . . 169

A Справочник для рабочего стола **171**
A.1 API, который нужен в книге . . . 171
A.2 Быстрая диагностика . . . 171
A.3 Минимальная памятка терминала . . . 172
A.4 Глоссарий . . . 172
A.5 Что отложено за пределы основного содержания . . . 174
A.6 Локальный launcher: устройство генератора . . . 174

Источники и атрибуция **175**

ГЛАВА 1

Введение: предмет книги

Цели главы

- определить предмет книги и итоговую задачу, к которой ведёт изложение;
- описать, из каких двух вычислительных частей состоит беспилотный летательный аппарат (БПЛА) Клевер и чем они различаются;
- назвать и кратко определить основные термины, используемые далее;
- представить структуру книги по ступеням.

Нужно знать заранее: предварительных знаний не требуется.

1.1 Предмет и итоговая задача

Предмет книги – программирование автономного полёта квадрокоптера и инженерная работа с ним: постановка задачи, программная реализация, измерение результата. В качестве платформы используется Клевер – образовательный квадрокоптер COEX, управляемый средствами ROS, а в качестве среды отладки – симулятор Gazebo.

Итоговая задача, к которой ведёт изложение: виртуальный квадрокоптер взлетает, облетает заданный маршрут, находит по изображению красную площадку, садится на неё, а программа формирует отчёт о длине пройденного пути и точности возвращения. Задача разбивается на подзадачи, каждая из которых вводится отдельно и заканчивается проверяемым результатом.



Рис. 1.1. Квадрокоптер – летательный аппарат с четырьмя винтами. Управление таким аппаратом в книге выполняется в компьютерной модели.

Фото: Jason Mavrommatis, Unsplash (бесплатная лицензия Unsplash).

• Задание

Сформулируйте итоговую задачу собственного проекта одной фразой. **Проверка:** в формулировке есть действие аппарата и измеримый результат (например, «облететь квадрат со стороной 1 м и вернуться с ошибкой менее 0.2 м»).

1.2 Назначение симулятора

Симулятор – программа, которая вычисляет поведение аппарата в виртуальной среде: динамику движения, столкновения, показания датчиков, изображение камеры. Он даёт три преимущества.

1. **Безопасность.** Ошибка в программе приводит к падению модели, а не к повреждению аппарата.
2. **Повторяемость.** Один и тот же опыт можно воспроизводить многократно с одинаковыми условиями, что необходимо для сравнения алгоритмов.
3. **Доступность.** Для отладки достаточно компьютера.

Симулятор не заменяет полёт реального аппарата: модель упрощает физику, шум датчиков и задержки. Поэтому в книге результаты симуляции всегда отделяются от выводов, требующих проверки на реальном аппарате.

• Задание

Приведите пример ситуации, где проверка решения на модели предпочтительнее проверки на реальном объекте, и укажите, какую информацию модель дать не может. **Проверка:** названы риск реальной проверки и конкретное ограничение модели (упрощённая физика, шум датчиков, ветер).

1.3 Две вычислительные части аппарата

Реальный Клевер содержит два вычислителя с разными функциями.

Полётный контроллер – плата, которая с частотой порядка сотен герц считывает датчики ориентации и подбирает обороты четырёх моторов так, чтобы аппарат сохранял устойчивость. На контроллере работает программа-**автопилот** (в книге – PX4).

Бортовой компьютер – одноплатный компьютер Raspberry Pi под управлением Linux. Он принимает изображение с камеры, выполняет пользовательскую программу и определяет, *куда* лететь, но не управляет отдельными винтами.

Фото: Vishnu Mohanan, Unsplash (бесплатная лицензия Unsplash).

Разделение функций повышает надёжность: контроллер продолжает стабилизировать аппарат даже при сбое программы на бортовом компьютере. В симуляторе роль контроллера выполняет та же программа PX4, запущенная на обычном компьютере.

• Задание

Распределите четыре действия между вычислителями: 1) стабилизация наклона; 2) поиск красного пятна на кадре; 3) расчёт оборотов моторов; 4) выбор следующей точки маршрута. **Проверка:** контроллер – 1 и 3; бортовой компьютер – 2 и 4.

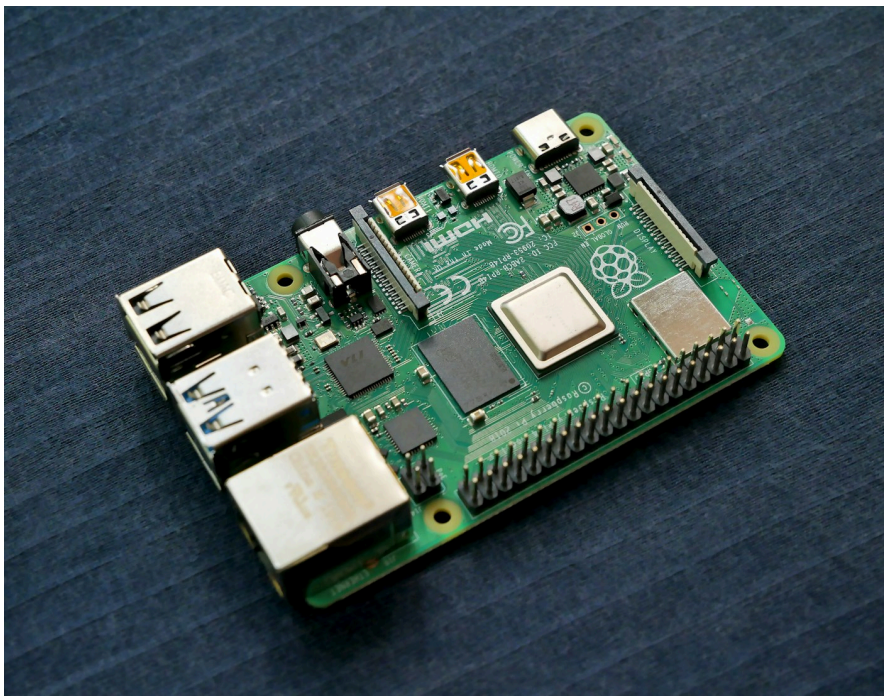


Рис. 1.2. Одноплатный компьютер Raspberry Pi – аппаратная основа бортового компьютера Клевера.

1.4 Структура книги

Изложение построено по восьми ступеням; каждая опирается на предыдущую.



Рис. 1.3. Структура книги: восемь ступеней от основных понятий до итогового проекта.

1.5 Основные термины

Термины ниже используются с первых глав. Здесь приведены краткие определения; полное изложение – в указанных главах.

Термин	Краткое определение	Подробно
Программа	Последовательность команд, выполняемых компьютером	гл. 6
Терминал	Окно для ввода текстовых команд операционной системе	гл. 4
ROS	Набор библиотек, инструментов и соглашений для взаимодействия программ робота	гл. 3
Нода	Запущенная программа, участвующая в обмене средствами ROS	гл. 7
Топик	Именованный поток сообщений: одни программы публикуют, другие подписываются	гл. 7
Сервис	Обмен «запрос–ответ» между двумя программами	гл. 7
Автопилот (PX4)	Программа полётного контроллера, стабилизирующая аппарат	гл. 2
Симулятор (Gazebo)	Среда, моделирующая физику, сенсоры и окружение	гл. 2
Телеметрия	Данные состояния аппарата: положение, скорость, режим	гл. 8
Система координат (фрейм)	Соглашение о начале и направлении осей измерения	гл. 10
Blockly	Визуальное программирование блоками с генерацией кода Python	гл. 11
Миссия	Программа, задающая последовательность целей и проверок	гл. 13

• Задание

Выберите три термина и сформулируйте их определения без обращения к таблице. **Проверка:** каждое определение указывает назначение термина, а не только его название.

Итоги и самопроверка

- Предмет книги – программирование автономного полёта Клевера на ROS и отладка в симуляторе Gazebo; итоговая задача – миссия с маршрутом, поиском цели и отчётом.
- Симулятор обеспечивает безопасность, повторяемость и доступность, но не заменяет реальный полёт.
- У аппарата два вычислителя: полётный контроллер стабилизирует, бортовой компьютер решает прикладную задачу.
- Изложение организовано в восемь ступеней; глава начинается с целей и заканчивается итогами.

Проверьте себя: 1) Зачем аппарату два вычислителя? 2) Назовите три преимущества симулятора. 3) Какой термин из таблицы остался неясным?
Ответы: 1) контроллер быстро и надёжно стабилизирует аппарат, бортовой компьютер выполняет прикладные и более медленные задачи (камера, маршрут); 2) безопасность, повторяемость, доступность; 3) индивидуальный ответ: следует перейти к главе, указанной в таблице.

ГЛАВА 2

Что именно мы моделируем

Цели главы

- перечислить пять слоёв, из которых состоит наш «виртуальный дрон», и сказать, что делает каждый;
- проследить путь одной команды от вашей программы до моторов и обратно;
- понять, какие версии программ использует книга и почему их нельзя менять наугад.

Нужно знать заранее: только введение. Все названия ниже объясняются по одному.

2.1 Слой за слоем: из чего сложен виртуальный дрон

Систему удобно рассматривать снизу вверх: каждый слой решает одну задачу и опирается на нижележащий. Слои описываются последовательно.

Слой 1. Мир и физика – Gazebo. Gazebo – программа, которая создаёт виртуальный мир: пол, стены, предметы, силу тяжести. Она считает, как под действием винтов движется модель дрона, и «рисует» то, что увидела бы камера. Если сравнить с компьютерной игрой, Gazebo – её движок. Версия, которую использует книга, называется Gazebo Classic (прежняя ветка Gazebo; новое поколение называется иначе и здесь не применяется).

Слой 2. Автопилот – PX4. PX4 – программа автопилота. Её задача: получив цель «держи высоту один метр», бесконечно быстро (сотни раз в секунду) подбирать обороты моторов, чтобы дрон не упал и не кренился. Для этого он по показаниям датчиков вычисляет положение и скорость аппарата (*оценка состояния*), а затем рассчитывает управление так, чтобы уменьшать ошибку между целью и текущим состоянием (*регуляторы*). В настоящем Клевере PX4 живёт на отдельной плате; в симуляторе он работает на вашем компьютере. Режим работы называется **PX4 SITL** – Software In The Loop, «программа вместо железа в контуре».

Слой 3. Переводчик – MAVROS. У автопилота и у ROS разные языки. Автопилот говорит на протоколе **MAVLink** (протокол – заранее согласованный формат обмена; здесь это короткие пакеты «положение», «режим», «команда»). MAVROS – программа-переводчик: она превращает пакеты MAVLink в понятные сообщения ROS и обратно.

Слой 4. Система связи – ROS. ROS (Robot Operating System) позволяет нескольким программам обмениваться данными: камера публикует кадры, автопилот – положение, ваша миссия – команды. Подробно – в главе 3; пока достаточно помнить: ROS – это «почтовая служба» между программами, у которой есть общие правила адресации и формат сообщений. Программа, участвующая в такой почтовой службе, называется *нодой*.

Слой 5. Удобные команды – Clover и simple_offboard. Clover – пакет (набор программ с общим назначением), который добавляет программы-помощники. Самая важная: `simple_offboard`. Она позволяет написать «лети в точку (1, 0, 1)», не думая о моторах и сообщениях MAVLink.

И наконец – ваша миссия. Это программа на Blockly или Python. Она ставит цели («взлети», «лети вперёд на метр», «сядь») и проверяет результат. Всё, что ниже, работает без вас; миссия

даёт смысл.

Запомните одним предложением

Миссия просит, Clover переводит просьбу в цель, MAVROS передаёт цель автопилоту, PX4 управляет моторами, Gazebo двигает модель и сообщает новые показания – и круг замыкается.

Рядом стоят два понятия, которые путают. **TF** – служба ROS, которая хранит, где какая система координат находится относительно другой (подробно – глава 10). **ArUco** – чёрно-белые квадратные метки, по которым камера узнаёт своё положение (глава 20). Эти два понятия здесь только названы; они понадобятся позднее и будут объяснены подробно.

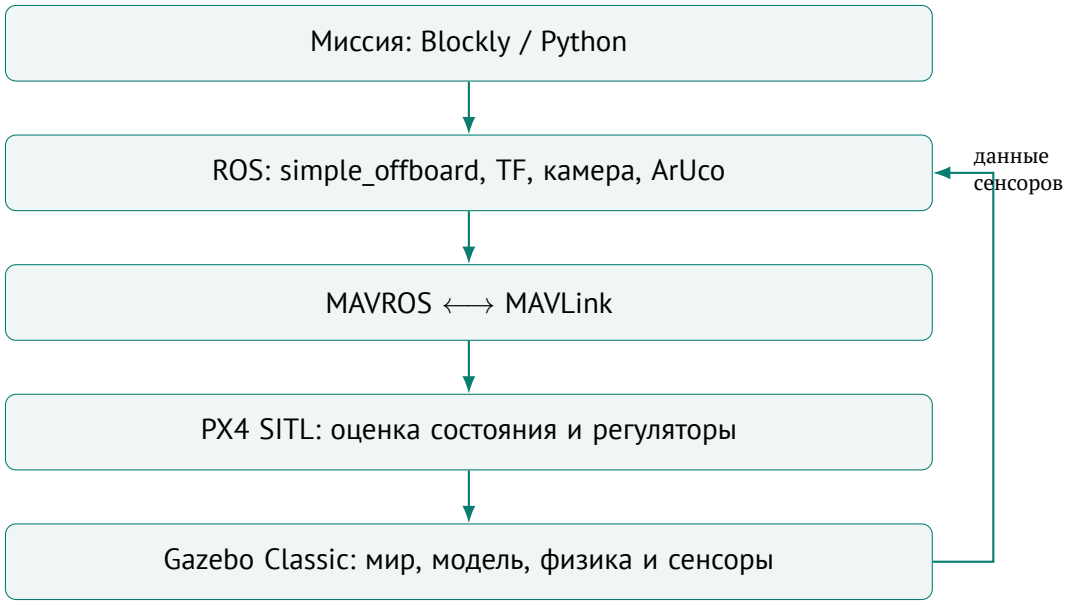


Рис. 2.1. Схема пути команды и обратной связи. Стрелки показывают, кто кому передаёт данные.

Источник: COEX, состав и особенности симулятора. Проверено 11.10.2026.

• Задание

Перерисуйте пять слоёв столбиком и над каждой стрелкой подпишите, что передаётся (цель, команда, данные). **Проверка:** на схеме есть стрелка обратной связи от Gazebo к миссии.

2.2 Какую версию мы изучаем и почему

Документация классической симуляции прямо указывает визуальную модель **Клевера 4**. Это хороший стенд для алгоритмов навигации, работы с координатами и зрением. Он не обещает точную аэродинамику каждой комплектации. В проверенном launch-файле (текстовом файле, который перечисляет, какие программы запустить и с какими настройками) симулятора параметр `use_clover_physics`, включающий физические параметры, подобранные под Клевер, по умолчанию отключён. Поэтому внешний вид модели не доказывает, что масса, моменты инерции (характеристики сопротивления вращению) и тяга совпадают с настоящим дроном.

Клевер 5 существует, и его нельзя считать просто новым названием той же модели. В обновлённой документации фигурируют новая модель бортового компьютера Raspberry Pi 5 и полётный контроллер MicoAirH743v2, а производитель описывает развитие платформы с

ROS 2 (следующее поколение ROS, несовместимое с ROS 1, которая используется в книге). Для учебных алгоритмов это близкая предметная область, для установки и программного интерфейса (API – набора доступных программе команд) – отдельная проверка. Глава 28 объясняет перенос.

Программный стек зафиксирован

Ubuntu 20.04 (версия операционной системы Linux), ROS 1 Noetic (Noetic – название выпуска ROS 1), Gazebo Classic 11 (версия симулятора) и совместимая сборка Clover/PX4. ROS Noetic завершил поддержку 31 мая 2025 года, Gazebo Classic 11 – 29 января 2025 года. В 2026 году такой набор удобен как воспроизводимый учебный стенд, но версии следует сохранять. Установка современного Gazebo поверх него не обновляет существующие плагины автоматически.

Источник: ROS PMC, завершение поддержки Noetic. Проверено 11.10.2026.

Источник: Open Robotics, жизненный цикл Gazebo Classic. Проверено 11.10.2026.

• Задание

Выпишите версии, которые не следует менять, и одной фразой объясните зачем. **Проверка:** названы Ubuntu 20.04, ROS Noetic, Gazebo Classic 11; причина – воспроизводимость.

2.3 Что можно исследовать

Задача	Что делает программа	Что измеряем
Полёт по точкам	Задаёт положение и ждёт прибытия	Ошибка, время, повторяемость
Маркерная навигация	Использует карту ArUco и TF	Видимость, доступность позиции
Обработка камеры	Выделяет объект, линию или QR-код	Задержка, ошибки распознавания
Облёт препятствия	Следует подготовленному маршруту	Просвет и столкновения
Обследование поля	Строит сетку или «змейку»	Покрытие и длина пути
Исследование отказа	Вносит управляемое изменение	Обнаружение и реакция

Обход препятствия сам по себе не появляется: основная команда Клевера `navigate` («лететь в точку», подробно – глава 13) ведёт аппарат к цели по прямой. Планировщик пути, обнаружение препятствий и логика возврата – задачи вашей программы. Виртуальная камера видит рендер сцены (изображение, построенное по описанию сцены), поэтому изображение должно содержать нужные признаки: маркеры, контраст пола или цветные объекты.

Практика. Опишите собственную миссию

Выберите один сценарий из таблицы. Запишите входы (камера, координаты, карта), выходы (точки, скорости, отчёт) и критерий успеха. Пример: «Облететь квадрат 1 м, вернуться к старту с ошибкой менее 0.2 м, приземлиться, повторить три раза». Числа в заданиях – учебные ориентиры, которые вы проверяете на своём стенде.

Итоги и самопроверка

- Пять слоёв: Gazebo (мир) → PX4 (автопилот) → MAVROS (переводчик) → ROS (связь) → Clover (удобные команды); сверху – ваша миссия.
- Ответ «команда принята» приходит раньше, чем дрон долетел: прибытие проверяют по данным.
- Версии зафиксированы (Ubuntu 20.04, ROS Noetic, Gazebo Classic 11); менять их наугад нельзя.

Проверьте себя: 1) Какой слой вычисляет обороты моторов? 2) Что делает MAVROS? 3) Почему симулятор не доказывает, что программа заработает на настоящем дроне? *Ответы:* 1) PX4; 2) переводит между MAVLink и ROS; 3) модель упрощает физику, камеру и задержки – параметры нужно проверить на реальном аппарате.

ГЛАВА 3

Зачем роботу ROS

Цели главы

- объяснить, какую проблему решает ROS, на примере задачи «найти красную площадку»;
- показать на схеме, что именно работает на бортовом компьютере реального Клевера и как оно связано с автопилотом;
- перечислить, что ROS даёт вам на практике и чего он не даёт;
- отличать ROS 1 от ROS 2.

Нужно знать заранее: введение и главу 2.

3.1 Начнём с задачи, а не с определения

Рассмотрим задачу: дрон должен подняться, найти красную площадку, долететь до неё и вернуться. Чтобы это получилось, одновременно должны работать пять «исполнителей». Камера снимает изображение. Программа ищет на снимке красный объект. Другая программа решает, куда двигаться. Автопилот удерживает аппарат в воздухе. Ещё одна программа записывает результат опыта. Все они работают *одновременно* и должны постоянно передавать друг другу данные.

Есть два плохих способа. Можно написать **одну огромную программу**: изменение камеры затронет код полёта, запись журнала окажется вплетена в распознавание, а ошибку будет трудно найти. Можно написать **несколько программ без общих правил**: придётся самим договариваться, в каком формате передавать данные, как соединяться, как называть данные и в каком порядке всё запускать. Именно эту организационную работу берёт на себя ROS. Отдельную работающую программу в ROS называют **нодой** (от англ. node, «узел»), а общие каналы передачи данных между нодами – **топиками**; оба понятия разобраны подробно в главе 7.

ROS, Robot Operating System, – набор библиотек, инструментов и правил для создания программ робота. Название переводится как «операционная система робота», но настоящей операционной системой остаётся Linux (в книге – Ubuntu): ROS работает *поверх* неё и помогает отдельным программам обмениваться данными.

Аналогия: коллектив исполнителей

Камера – наблюдатель, детектор – аналитик, миссия – координатор, автопилот – исполнитель. ROS задаёт общие правила обмена: у каждого канала есть название и тип сообщений, любой участник может подписаться на нужный канал, а капитан может позвонить конкретному исполнителю с просьбой. Аналогия объясняет роли; программы не «понимают» сообщения, как люди, они лишь соблюдают формат.

• Задание

Нарисуйте схему «дрон ищет красную площадку»: пять прямоугольников (камера, детектор, миссия, автопилот, журнал) и стрелки с подписью, *что* передаётся. **Проверка:** на схеме отдельно показаны цель полёта (от миссии к автопилоту) и измеренное положение (от автопилота обратно); у журнала нет прямой связи с камерой, если ему достаточно данных миссии.

3.2 Как это устроено на настоящем Клевере

Это главный раздел главы: после него симулятор перестаёт быть абстракцией. Реальный Клевер – это **два компьютера в одном аппарате** (введение, раздел 1.3), и ROS живёт на втором из них.

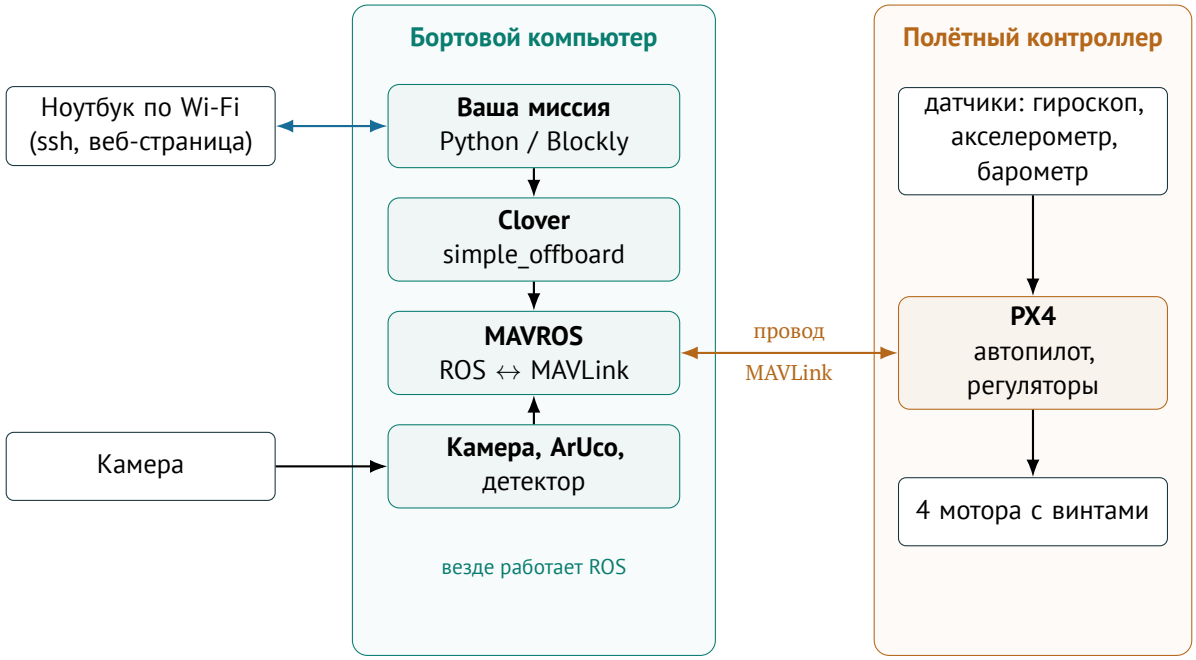


Рис. 3.1. Что где работает на реальном Клевере (упрощённая схема). ROS – только на бортовом компьютере; автопилот PX4 – на отдельной плате и связан с компьютером проводом по протоколу MAVLink.

Что происходит при включении. Бортовой компьютер загружает Linux. Готовый образ Клевера автоматически запускает основной launch-файл `clover.launch`. Launch-файл – текстовый файл, в котором перечислено, какие ноды запустить и с какими настройками. В результате поднимаются ROS-master (программа-справочник, через которую ноды находят друг друга), MAVROS, пакет Clover с `simple_offboard`, драйвер камеры, детектор ArUco (программа, которая находит на кадре ArUco-метки) и веб-инструменты. Пока вы ничего не написали, система уже работает: к ней можно подключиться и посмотреть, какие топики существуют и какие данные по ним идут.

Откуда берётся ваша программа. Ноутбук подключается к Wi-Fi дрона (по умолчанию дрон сам создаёт беспроводную сеть и открывает веб-страницу с инструментами). Для работы с командной строкой бортового компьютера используется `ssh` – защищённое удалённое подключение: окно терминала на ноутбуке, команды из которого выполняются на дроне. Скрипт Python или блоки Blockly *выполняются на самом бортовом компьютере*, а ноутбук служит экраном и клавиатурой. Поэтому дрон продолжает миссию, даже если ноутбук отключился: программа и ROS находятся на борту.

Как программа управляет моторами. Она моторами не управляет. Цепочка такая: ваш вызов `navigate` → сервис Clover → MAVROS → провод → PX4 → регуляторы → моторы. Важная деталь: режим автопилота OFFBOARD (внешнее управление: PX4 выполняет цели, которые присылает бортовой компьютер) требует *непрерывного потока* команд-целей (не реже нескольких раз в секунду). Если поток прекратился, автопилот считает, что связь потеряна, и переходит в аварийное поведение. Поэтому `simple_offboard` сам повторяет последнюю цель в фоне, а вам достаточно один раз сказать «лети туда». Побочный эффект: завершение вашего Python-скрипта *не* останавливает дрон – он продолжает удерживать последнюю цель (подробнее – глава 11, рамка про Stop и Land).

Кто следит за безопасностью. На реальном аппарате рядом всегда должен быть человек с пультом радиоуправления: переключатель режимов на пульте позволяет перехватить управление у программы. Это не формальность: программа может ошибиться, а винты опасны. В симуляторе пульт не нужен, но принцип «сначала определить способ остановки» применяется и в симуляции.

Схема обобщённая

Состав запускаемых программ, порты и адреса зависят от версии образа Клевера и комплектации. Точный список смотрите в файле `clover.launch` вашего образа и в документации COEX. Принцип – ROS на бортовом компьютере, PX4 на отдельной плате, связь по проводу – не меняется.

3.2.1 Чем симулятор отличается от настоящего дрона

Часть системы	На реальном Клевере	В симуляторе
Миссия, ROS, Clover, MAVROS	Те же программы на Raspberry Pi	Те же программы на вашем компьютере
Автопилот PX4	Отдельная плата, провод к Raspberry Pi	Программа PX4 SITL на вашем компьютере
Датчики и камера	Настоящие, с шумом и задержками	Модели в Gazebo, шум упрощён
Моторы и физика	Реальный воздух, батарея, ветер	Расчёт физики Gazebo
Безопасность	Пульт, зона полёта, проверка винтов	Перезапуск симуляции

Вывод для учёбы: **верхние слои** (ваша миссия, ROS, команды Clover) одинаковы, поэтому алгоритм, отлаженный в Gazebo, можно переносить на аппарат. **Нижние слои** отличаются, поэтому перед реальным полётом нужно перепроверить допуски, скорости и поведение камеры.

• Задание

Откройте схему и без подсказки ответьте письменно: 1) на каком из двух компьютеров работает ваш скрипт; 2) что произойдёт с дроном, если ноутбук отключится от Wi-Fi; 3) почему завершение скрипта не сажает дрон; 4) какие два слоя заменяет Gazebo в симуляторе. **Проверка:** ответы совпадают с рамкой «Итоги» в конце главы.

3.3 Какие трудности ROS снимает на примере Клевера

Без общей системы	С ROS	Что вы заметите
Каждую программу соединяют вручную	Подписка на топик по имени	Добавить наблюдателя можно, не трогая издателя
Свой формат для каждого сообщения	Типы сообщений (`Image` – изображение, `PointStamped` – точка с отметкой времени и системой координат)	Любой инструмент умеет показать изображение или точку
Для проверки пишут отладочный код	Команды `rostopic echo` (показать сообщения топика) и `rostopic hz` (измерить частоту сообщений), графическая оболочка `rqt`	Видно, что камера работает, а подписчик ошибся с именем
Данные опыта теряются	Инструмент `rosbag` записывает поток сообщений в файл	Распознавание можно отлаживать на записи без полёта
Каждая система координат – «в голове»	TF (служба, хранящая взаимное положение систем координат) хранит все системы координат	Положение маркера пересчитывается в положение дрона
Код полёта и код камеры в одном файле	Отдельные ноды	Детектор и маршрут разрабатываются и проверяются независимо

• Задание

Выберите одну строку таблицы и придумайте, как бы вы решали задачу без ROS. **Проверка:** описание содержит дополнительную работу, которую ROS берёт на себя.

3.4 Преимущества ROS: практическая польза

Разделение работы. Детектор, маршрут и журнал разрабатываются отдельно. Разработчики согласуют интерфейс («детектор публикует центр цели») и проверяют каждую часть независимо.

Повторное использование. Для многих камер, сообщений и инструментов есть готовые пакеты: вместо собственного формата изображения используют `sensor_msgs/Image`, вместо новой системы связи – издателя и подписчика. Совместимость пакета с вашей версией ROS всё равно проверяют.

Наблюдаемость. Все команды диагностики одинаковы для любого робота на ROS. Вы научитесь отвечать на вопросы «работает ли камера?», «с какой частотой идут данные?», «кто подписан на топик?» без вставки `print` в чужой код.

Запись и повтор данных. Rosbag – инструмент, который сохраняет поток сообщений в файл и затем воспроизводит его. Распознавание можно запускать на записи, пока симулятор выключен, и сравнивать алгоритмы на одном и том же наборе кадров.

Один набор программ для симулятора и реального дрона. Алгоритм маршрута использует те же сервисы в симуляции и на аппарате. Совпадение интерфейса не делает условия одинаковыми (таблица выше).

Распределённая работа. Ноды можно запустить на нескольких компьютерах в одной сети: тяжёлую обработку изображения – на ноутбуке, управление – на борту. В книге используется один компьютер.

• Задание

Выберите два преимущества из списка и приведите по одному примеру из вашего будущего проекта. Например: «Запись данных: я записываю камеру один раз и три раза настраиваю пороги детектора без полётов». **Проверка:** в каждом примере названо действие, которого без ROS пришлось бы делать больше или сложнее.

3.5 Что ROS сам по себе не решает

ROS не придумывает алгоритм поиска площадки, не исправляет неверные координаты и не делает программу «реального времени»: ошибка в Python, задержка сети или неправильный фрейм остаются задачами разработчика. ROS также не отвечает за безопасность полёта – её обеспечивают автопилот, пульт и ваша аккуратность. Для программы, которая считает площадь треугольника, ROS не нужен вообще. Его преимущества проявляются в системе, где несколько программ работают вместе. В книге есть оба типа примеров: обычные Python-файлы изучают алгоритм, ROS-программы подключают его к системе робота.

• Задание

Назовите три задачи проекта «красная площадка», которые ROS за вас extbfне решит. **Проверка:** например: алгоритм поиска, правильные координаты, безопасность полёта.

3.6 ROS 1 и ROS 2: о какой версии речь

Название ROS объединяет два поколения. Наш стенд основан на **ROS 1 Noetic**: Python-библиотека `rospy` (набор функций для написания нод на Python), запуск набора нод командой `roslaunch` по launch-файлу, сборка пакетов системой catkin. ROS 2 использует другие интерфейсы (например, библиотеку `rclpy` вместо `rospy`), и простая замена номера версии в инструкции программу не переносит. Noetic и Gazebo Classic завершили поддержку (глава 2); в книге фиксируется образ и документацию этого стека. Перед выбором нового комплекта выясните, какая платформа в нём установлена (глава 28).

• Задание

Прочитайте в терминале стенда команду `echo $ROS_DISTRO` (глава 4) и запишите результат. Затем найдите в тексте главы три слова, которые относятся *только* к ROS 1 (подсказка: библиотека, команда запуска, система сборки). **Проверка:** вывод `noetic`; слова – `rospy`, `roslaunch`, catkin.

Итоги и самопроверка

- ROS – правила и инструменты поверх Linux, чтобы несколько программ робота обменивались данными; он не заменяет ОС и не придумывает алгоритмы.
- На реальном Клевере ROS работает на бортовом компьютере; автопилот PX4 – на отдельной плате, связь по проводу (MAVLink через MAVROS).
- Ваш скрипт выполняется на борту; ноутбук по Wi-Fi лишь показывает и запускает. OFFBOARD требует непрерывного потока целей: его повторяет `simple_offboard`.

- Рядом с настоящим дроном нужен человек с пультом.
- Верхние слои одинаковы в симуляторе и на дроне; нижние (датчики, моторы, ветер) – разные.

Проверьте себя: 1) Можно ли заменить Ubuntu библиотекой ROS? 2) Кто удерживает горизонтальное положение дрона? 3) Что удобнее передавать из детектора: его внутренний код или сообщение с результатом? 4) Где ROS помогает сильнее: в одном вычислении или в обмене между программами? 5) Что произойдёт с полётом, если на реальном дроне отключить ноутбук от Wi-Fi?
Ответы: 1) нет, ROS работает поверх ОС; 2) автопилот PX4 (не ROS и не ваш цикл); 3) сообщение с результатом; 4) в обмене между программами; 5) ничего: миссия работает на бортовом компьютере; пропадёт только возможность наблюдать и вмешаться с ноутбука, поэтому нужен пульт.

Источник: COEX, ROS в архитектуре Клевера. Проверено 11.10.2026.

ГЛАВА 4

Терминал, файлы и запуск программ

Цели главы

- ориентироваться в каталогах командами pwd, ls, cd;
- создать и запустить свой Python-файл;
- понимать, что такое окружение ROS и зачем нужна команда source;
- принять обозначение терминалов A, B, C, которое используется во всей книге;
- читать сообщение об ошибке снизу вверх.

Нужно знать заранее: ничего специального. Слова «ROS», «симулятор» и «миссия» в этой главе только упоминаются; они объясняются в главах 2 и 3.

4.1 Три места, которые легко перепутать

Все программы этой книги запускаются в операционной системе **Ubuntu** – одном из вариантов операционной системы Linux. Чаще всего компьютер читателя работает под Windows, поэтому Ubuntu запускают внутри Windows как **виртуальную машину** (VM): это программа, которая в обычном окне имитирует «второй компьютер» со своей операционной системой, своими файлами и своими программами. В результате получаются три вложенных места: Windows → окно виртуальной машины с Ubuntu → терминалы внутри Ubuntu. Файл, лежащий на рабочем столе Windows, не появляется в домашнем каталоге Ubuntu автоматически: это файлы разных компьютеров.

Терминал – окно, в котором команды вводят текстом вместо щелчков мышью. **Shell** (оболочка) – программа внутри терминала, которая читает введённую строку и выполняет её. В Ubuntu обычно используется оболочка Bash, в Windows – PowerShell. Команды этих оболочек различаются, поэтому все команды книги выполняются *только* в терминале Ubuntu.

Строка вроде `clover@ubuntu:~$` – приглашение оболочки: она показывает, что можно вводить команду. В книге приводится только сама команда, приглашение копировать не нужно. Знак `#` в начале строки означает комментарий – пояснение для человека, которое оболочка не выполняет.

• Задание

Нарисуйте вложенные окна: Windows → виртуальная машина Ubuntu → терминал. Отметьте, где будут вводиться команды из этой книги. **Проверка:** команды вводятся в терминале внутри Ubuntu, а не в Windows.

4.2 Узнайте, где вы находитесь

Файлы хранятся в **каталогах** (папках). В каждый момент терминал «находится» в одном каталоге – текущем.

```
pwd
ls
cd "$HOME"
mkdir -p clover_course_files
cd clover_course_files
pwd
```

`pwd` (print working directory) показывает текущий каталог. `ls` показывает его содержимое. `cd` (change directory) меняет каталог. `mkdir -p` создаёт папку и не сообщает об ошибке, если она уже существует. Команда выполняется после нажатия Enter; следующую вводят, когда снова появилось приглашение.

`$HOME` – домашний каталог пользователя (имя переменной, в которой оболочка хранит этот путь). Точка `.` означает текущий каталог, две точки `..` – родительский. **Абсолютный путь** начинается от корня файловой системы, например `/home/clover/clover_course_files`. **Относительный путь** отсчитывается от текущего каталога. Кавычки сохраняют путь с пробелами как один аргумент команды (*аргумент* – то, что записано после имени команды и уточняет её действие).

Каталог опыта

Создайте в `clover_course_files` папку `lesson01`. Перейдите в неё и выведите `pwd`. Затем вернитесь на уровень выше командой `cd ..`. **Проверка:** путь в выводе изменился ровно на один каталог.

4.3 Как появляется программа

Программа – текстовый файл с командами. Откройте текстовый редактор Ubuntu, создайте файл `hello.py` в рабочем каталоге и введите:

```
print('Hello, Clover!')
print(2 + 3)
```

Сохраните файл. В терминале перейдите в тот же каталог и запустите его интерпретатором Python (программой, которая читает файл и выполняет команды):

```
python3 hello.py
```

Ожидаемый вывод – строка `Hello, Clover!` и число 5. Python прочитал файл, выполнил команды сверху вниз и завершился.

Если появляется `can't open file`, проверьте `pwd`, `ls` и имя. Linux различает большие и маленькие буквы: `Hello.py` и `hello.py` – разные имена. Расширение должно быть `.py`, а не `.py.txt`. Кнопка Save сохраняет файл, команда `python3` запускает его; это разные действия.

• Задание

Создайте файл `hello.py`, который печатает ваше имя, и запустите его. **Проверка:** в терминале появилось ваше имя.

4.4 Окружение ROS

ROS – набор программ и библиотек для роботов (глава 3). Чтобы терминал мог их найти, ему нужны настройки – **переменные окружения**: именованные значения, например список

каталогов, где искать программы. Команда `source` читает файл настроек и применяет его к *текущему* терминалу. Следующие четыре команды выполняются после установки стенда (глава 5); здесь описано, что они делают.

```
source /opt/ros/noetic/setup.bash
source "$HOME/catkin_ws/devel/setup.bash"
echo "$ROS_DISTRO"
rospack find clover
```

Первая строка подключает базовую установку ROS (Noetic – название выпуска, который использует книга). Вторая подключает **рабочее пространство** (workspace) – каталог `catkin_ws`, в котором хранятся и собираются пакеты ROS, в том числе пакет `clover`. **Пакет** – набор программ и файлов с общим назначением. Команда `echo` печатает значение переменной; для `ROS_DISTRO` ожидается `noetic`. Команда `rospack find` показывает, в каком каталоге лежит пакет. Если рабочее пространство расположено иначе, используйте путь из паспорта стенда (глава 5).

Настройки действуют только в том терминале, где выполнена `source`. Новый терминал начинается «с чистого листа» – отсюда частая ситуация «в одном окне работает, в другом не находится». В готовой виртуальной машине эти команды могут выполняться автоматически: они записаны в файл `.bashrc`, который оболочка читает при открытии каждого нового терминала. Поэтому в новом терминале стоит проверить окружение командой `echo "$ROS_DISTRO"`.

• Задание

После установки стенда выполните `echo $ROS_DISTRO` в новом терминале до и после команды `source`. **Проверка:** до `source` – пусто (если окружение не подключается автоматически), после – `noetic`.

4.5 Один опыт – несколько терминалов

Многие опыты книги требуют одновременной работы нескольких программ. Каждая программа занимает свой терминал: пока она работает, в этом окне нельзя вводить новые команды, потому что приглашение оболочки не вернулось. Поэтому открывают несколько терминалов и называют их буквами. **Эти обозначения используются во всей книге.**

Окно	Что в нём работает	Пример
A	Основная долгая программа, которую нельзя прерывать до конца опыта: чаще всего *симулятор* (программа, которая моделирует виртуальный дрон; глава 2), в лаборатории ROS – `roscore` (глава 7)	Окно с непрерывно идущим выводом запуска
B	Ваша программа, *миссия* – программа, которая командует дроном (или учебная программа главы)	`python3 hello.py`
C	Наблюдение и проверка: команды диагностики, *телеметрия* (данные о состоянии дрона), запись журнала	Команды вида `rostopic echo`

Если в главе сказано «в окне B», это значит: откройте (или используйте уже открытый) второй терминал и введите команду там. Не вводите новую команду в окно A, пока в нём работает долгая программа.

Сочетание `Ctrl+C` отправляет работающей в терминале программе просьбу завершиться. Для обычной программы это завершение цикла. Для полётной программы поведение зависит от её кода: дрон может остаться в воздухе. Поэтому после опыта проверяют, что аппарат сел и его моторы выключены: в данных дрона поле `armed` (моторы запущены, аппарат «взведён» к полёту) равно `false`. Обратное состояние называется disarmed.

• Задание

Опишите, что запускается в окнах A, B и C для опыта «квадрат» (аппарат облетает квадрат и садится). **Проверка:** A – симулятор, B – миссия, C – наблюдение за телеметрией или запись журнала.

4.6 Читайте ошибку снизу вверх

Если программа завершилась ошибкой, Python печатает *traceback* – цепочку вызовов, приведшую к ошибке:

```
Traceback (most recent call last):
  File "hello.py", line 2, in <module>
    print(height)
NameError: name 'height' is not defined
```

Последняя строка называет ошибку: переменная height не определена. Строка выше показывает файл и номер строки. Откройте именно этот файл, проверьте написание имени и место, где оно получает значение. Не исправляйте всё подряд: сформулируйте одну возможную причину и проверьте её.

Сообщение	Первое действие
`SyntaxError`	Проверьте скобки, кавычки и двоеточие в указанной строке
`IndentationError`	Проверьте отступы; используйте пробелы одинаковой ширины
`ModuleNotFoundError`	Python не нашёл подключаемый модуль: проверьте, что запущен системный `python3`, подключено окружение ROS и верно написано имя
`Permission denied`	Нет прав на файл или каталог: проверьте путь; не добавляйте `sudo` (запуск с правами администратора) без понимания причины
`Service unavailable`	Служба ROS не отвечает: проверьте, что нужная программа запущена, и точное имя службы (глава 7)

• Задание

В программе написано `print(heigth)`. Найдите тип ошибки и причину, не запуская. **Проверка:** NameError: имя heigth нигде не определено (опечатка).

4.7 Рабочая привычка: версия перед опытом

Сохраните исходную работающую программу под отдельным именем. В копии измените одну величину, например скорость. В имени журнала укажите номер опыта. Тогда можно связать код и результат: `square_slow.py`, `square_slow_01.csv` (CSV – текстовый файл-таблица, значения разделены запятыми). Для командной работы существуют системы контроля версий, например Git, но на начальном этапе достаточно понятной структуры файлов и записи изменений.

Самостоятельный запуск

Создайте `hello.py`, выполните его и намеренно сделайте опечатку в имени переменной. Найдите номер строки в сообщении об ошибке и исправьте код. Сохраните короткую памятку: где лежат файлы, чем запускается Python и какие команды `source` нужны в вашей VM.

Итоги и самопроверка

- Терминал – окно ввода команд, shell – программа, которая их выполняет; все команды книги вводятся в терминале Ubuntu.
- `source` подключает настройки ROS только в текущий терминал.
- Окна A, B и C – обозначения терминалов во всей книге: A – долгая основная программа, B – ваша программа, C – наблюдение.
- Ошибку читают снизу вверх: тип, затем файл и строка.

Проверьте себя: 1) Чем абсолютный путь отличается от относительного? 2) Почему в новом терминале ROS может «пропасть»? 3) Что означает «запустите в окне B»?
Ответы: 1) абсолютный начинается от корня, относительный – от текущего каталога; 2) `source` действует только в том окне, где выполнен; 3) ввести команду во втором терминале, не прерывая программу в окне A.

ГЛАВА 5

Настройка рабочей среды

Цели главы

- выбрать между готовой виртуальной машиной и нативной Ubuntu;
- получить рабочий стенд и проверить его командами;
- составить «паспорт окружения» – запись версий, без которой опыт нельзя повторить.

Нужно знать заранее: главы 4 (терминал, каталоги, source).

5.1 Что нужно установить

Стенд – набор программ на одном компьютере, на котором можно запускать опыты этой книги. Он состоит из следующих частей (каждая подробно описана в главе 2).

Часть	Назначение
Ubuntu 20.04	Операционная система (вариант Linux), в которой работает всё остальное
ROS 1 Noetic	Система обмена данными между программами робота
Gazebo	Симулятор: виртуальный мир и физика
PX4 (SITL)	Автопилот, работающий как обычная программа
MAVROS	Переводчик между ROS и автопилотом
Clover	Пакет с готовыми командами для управления Клевером

Все части можно получить двумя способами: готовым образом виртуальной машины или установкой с нуля.

5.2 Два способа получить стенд

Для первого знакомства проще импортировать **готовый образ виртуальной машины** COEX (файл, содержащий уже установленную и настроенную Ubuntu со всеми частями стенда). Для собственной разработки и лучшей производительности подходит **нативная Ubuntu** – Ubuntu, установленная прямо на компьютер, без виртуальной машины. Windows здесь служит системой-хозяином (операционной системой, внутри которой запускается виртуальная машина): ROS, PX4 и симулятор работают внутри Ubuntu. WSL2 (встроенная в Windows подсистема для запуска Linux) с графикой можно исследовать отдельно, но она не является проверенной базой этой книги.

Вариант	Преимущество	Что проверить
Готовая VM	Меньше шагов, веб-инструменты рядом	Версия образа, GPU, ресурсы VM
Нативная Ubuntu	Меньше накладных расходов графики	Версии пакетов, PX4, веб-сервер

• Задание

Выберите способ получения стенда и запишите причину выбора. **Проверка:** названы преимущество выбранного варианта и то, что нужно проверить.

5.3 Путь A: виртуальная машина COEX

Откройте релизы clover_vm и скачайте образ с указанной там страницы. При проверке релиз v1.4 отмечен как Latest; в описании указаны PX4 v1.13.3 и обновления Clover v0.24. Не подменяйте PX4 в этом образе версией из инструкции ручной сборки. Документация образа описывает Ubuntu 20.04 с XFCE и предустановленными инструментами.

1. Включите аппаратную виртуализацию (режим процессора, без которого виртуальная машина работает крайне медленно) в настройках BIOS/UEFI – программы, которая запускается при включении компьютера до загрузки Windows; если режим уже включён, шаг пропускается.
2. Импортируйте файл образа (формат OVA – «упакованная» виртуальная машина) в программу виртуализации (гипервизор), например VMware или VirtualBox. COEX исторически рекомендует VMware при проблемах совместимости VirtualBox; точное поведение зависит от версии гипервизора и драйвера.
3. Для начала попробуйте 4 виртуальных ядра и 6–8 ГБ RAM, если компьютер позволяет. Это рабочая рекомендация автора, а не минимальные требования производителя. Оставьте ресурсы Windows.
4. Включите доступное 3D-ускорение и запустите образ. В документации приведены стандартные логин и пароль `clover / clover`; проверьте описание вашего релиза.
5. Для работы только в симуляторе в настройках сети виртуальной машины выберите режим NAT (виртуальная машина выходит в сеть через хозяина и не видна другим компьютерам). Режим «мост» (виртуальная машина получает собственный адрес в сети) потребуется лишь для отдельных сетевых сценариев; браузер внутри VM открывает её `localhost`.
6. Создайте снимок VM «чистый рабочий стенд» (сохранённое состояние виртуальной машины, к которому можно вернуться) после проверки первого запуска.

Источник: COEX, установка виртуальной машины. Проверено 11.10.2026.

Источник: CopterExpress, релизы виртуальной машины. Проверено 11.10.2026.

• Задание

Выполните шаги 1–6 и сделайте снимок виртуальной машины. **Проверка:** VM запускается, терминал открывается, снимок виден в списке снимков.

5.4 Путь B: ручная сборка на Ubuntu 20.04

Этот путь рассчитан на компьютер с установленной Ubuntu 20.04 и уже установленным **ROS Noetic Desktop или Desktop Full** (варианты установки ROS с графическими инструментами). Установка ROS описана в официальной инструкции, на которую ссылается COEX. На системе другого поколения сначала создайте отдельную Ubuntu 20.04; не выполняйте эти команды в Ubuntu 24.04 с ROS 2. Репозитории и способы получения старых пакетов могут меняться после окончания поддержки.

5.4.1 Рабочее пространство и пакеты

Ниже по очереди выполняются три действия: устанавливаются системные программы (`apt-get` – менеджер программ Ubuntu; `pip` – менеджер библиотек Python), создаётся рабочее пространство (каталог `catkin_ws`, где собираются пакеты ROS; `src` – подкаталог с исходными текстами пакетов) и копируются исходные тексты пакетов командой `git clone` (Git – система хранения версий кода; `clone` копирует репозиторий с сайта GitHub). `rosdep` – программа, которая устанавливает недостающие зависимости пакетов.

```
source /opt/ros/noetic/setup.bash
sudo apt-get update
sudo apt-get install build-essential git python3-pip python3-rosdep
mkdir -p "$HOME/catkin_ws/src"
cd "$HOME/catkin_ws/src"
git clone https://github.com/CopterExpress/clover.git
git clone https://github.com/CopterExpress/ros_led.git
git clone https://github.com/ethz-asl/mav_comm.git
cd "$HOME/catkin_ws"
sudo rosdep init
rosdep update --include-eol-distros
rosdep install --from-paths ./src --ignore-src --rosdistro noetic -y
sudo /usr/bin/python3 -m pip install -r src/clover/clover/requirements.txt
```

Если `rosdep init` сообщает, что конфигурация уже существует, второй раз инициализировать её не нужно. Если какой-либо пакет не находится, сначала зафиксируйте полную ошибку и проверьте источник пакетов; добавление случайного репозитория другой Ubuntu ухудшит воспроизводимость. Установка Python-зависимостей выше следует историческому рецепту COEX именно для Ubuntu 20.04.

5.4.2 PX4: совместимая версия и подмодули

PX4 – программа автопилота (глава 2); её исходные тексты копируются в отдельный каталог и подключаются к рабочему пространству символическими ссылками (`ln -s` создаёт ссылку на каталог, не копируя его). Подмодули (`--recursive`) – вложенные репозитории, которые нужны PX4. Команда `catkin_make` собирает пакеты рабочего пространства (переводит исходный код в исполняемые программы). Инструкция нативной сборки COEX указывает PX4 `v1.12.3`. Это другая исходная конфигурация, чем готовый образ v1.4. Используйте весь рецепт одной ветки установки целиком.

```
git clone --branch v1.12.3 --recursive \
  https://github.com/PX4/PX4-Autopilot.git "$HOME/PX4-Autopilot"
ln -s "$HOME/PX4-Autopilot" "$HOME/catkin_ws/src/PX4-Autopilot"
ln -s "$HOME/PX4-Autopilot/Tools/sitl_gazebo" \
  "$HOME/catkin_ws/src/sitl_gazebo"
ln -s "$HOME/PX4-Autopilot/mavlink" "$HOME/catkin_ws/src/mavlink"
cd "$HOME/PX4-Autopilot/Tools/setup"
sudo ./ubuntu.sh --no-nuttx
/usr/bin/python3 -m pip install --user toml
ln -s "$HOME"/catkin_ws/src/clover/clover_simulation/airframes/* \
  "$HOME/PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes/"
sudo /opt/ros/noetic/lib/mavros/install_geographiclib_datasets.sh
cd "$HOME/catkin_ws"
catkin_make -j2
source devel/setup.bash
```

Флаг `--no-nuttx` подходит для SITL: сборка прошивки настоящего контроллера в книге не требуется. Сообщение `File exists` для символической ссылки означает, что её уже создали; проверьте путь, а не удаляйте каталог вслепую. Если скрипт geographiclib отсутствует, проверьте результат установки MAVROS через rosdep.

После успешной сборки добавьте в файл `~/.bashrc` (файл, который читает оболочка при открытии каждого терминала; глава 4) две строки `source`: для `/opt/ros/noetic/setup.bash` и для `~/catkin_ws/devel/setup.bash`. Порядок важен: сначала ROS, затем workspace. Откройте новый терминал и проверьте `rospack find clover`.

Источник: COEX, нативная сборка: пакеты, PX4 и зависимости. Проверено 11.10.2026.

• Задание

Откройте новый терминал и выполните `rospack find clover`. **Проверка:** выводится путь к пакету; если «not found», проверьте строки source в `.bashrc`.

5.5 Сохраните паспорт окружения

```
mkdir -p "$HOME/clover_runs"
lsb_release -ds
rosversion -d
gazebo --version
cat /etc/clover_vm_version
git -C "$HOME/catkin_ws/src/clover" rev-parse HEAD
git -C "$HOME/PX4-Autopilot" describe --tags --always
```

Команды выводят: версию Ubuntu (`lsb_release`), версию ROS (`rosversion`), версию Gazebo, версию образа виртуальной машины (файл `/etc/clover_vm_version`), а также идентификатор версии пакета Clover и PX4 в системе Git (`git rev-parse`, `git describe`) – по ним можно точно воспроизвести стенд. Некоторые пути существуют только в одном варианте установки. Отсутствие `/etc/clover_vm_version` на нативной Ubuntu нормально. Сохраните вывод доступных команд, характеристики компьютера и настройки VM в текстовый файл. При совместной работе используйте одну проверенную VM и одинаковый каталог примеров.

Практика. Проверка установки

Найдите пути пакетов `clover` (команды Клевера), `clover_simulation` (описание симулятора Клевера), `gazebo_ros` (связь Gazebo с ROS) и `mavros` (переводчик между ROS и автопилотом) командой `rospack find`. Укажите версию ROS и способ установки. Успех: все четыре пакета доступны из нового терминала, а паспорт окружения сохранён.

Итоги и самопроверка

- Готовая VM быстрее в запуске; нативная Ubuntu даёт больше производительности и контроля.
- Версии (Ubuntu 20.04, ROS Noetic, PX4 из выбранной инструкции) смешивать нельзя.
- Паспорт окружения – файл с версиями и параметрами VM.

Проверьте себя: 1) Почему нельзя ставить команды из этой главы в Ubuntu 24.04? 2) Зачем делать снимок VM? 3) Что входит в паспорт окружения?
Ответы: 1) там другое поколение ROS и пакетов; 2) чтобы вернуться к чистому рабочему состоянию после ошибки; 3) версии ОС, ROS, Gazebo, Clover, PX4 и параметры VM.

ГЛАВА 6

Python с нуля: язык вашей миссии

Цели главы

- хранить данные в переменных и считать с ними;
- выбирать действие условием, повторять циклом;
- оформлять функции и читать ошибки.

Нужно знать заранее: главы 4 (запуск файла).

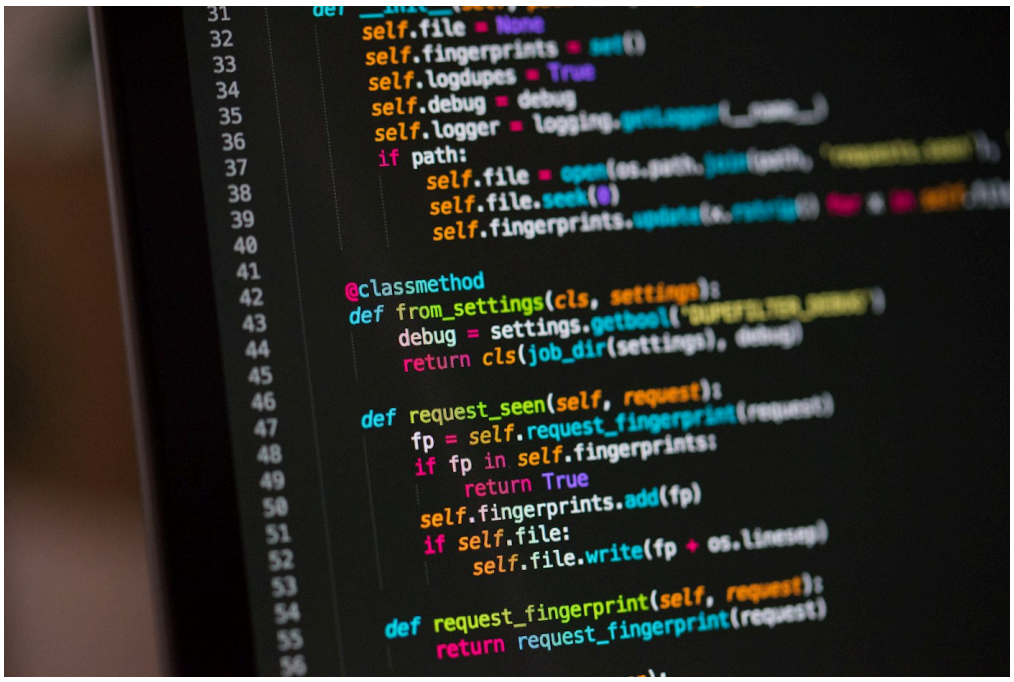


Рис. 6.1. Программа – это текст, который компьютер выполняет строка за строкой. Миссия дрона записывается тем же способом.

Фото: Chris Ried, Unsplash (бесплатная лицензия Unsplash).

6.1 Данные, команда и программа

Программа – последовательность инструкций и решений. Все примеры книги лежат в каталоге `examples`, который поставляется вместе с книгой; в Ubuntu его копируют в `~/clover_course_files/examples`. Чтобы выполнить пример, в терминале переходят в этот каталог и запускают файл: `cd "$HOME/clover_course_files/examples"`, затем `python3 python_basics.py` (глава 4). В первом примере дрон не управляется: программа только считает, сколько времени примерно понадобится на перемещение. Это позволяет изучить язык без ROS и без симулятора.

```
#!/usr/bin/env python3
"""Offline: numbers, variables, units and formatted output."""
height = 1.2
speed = 0.4
distance = 2.0
estimated_time = distance / speed
print(f'Height: {height:.1f} m')
print(f'Speed: {speed:.1f} m/s')
print(f'Ideal travel time: {estimated_time:.1f} s')
print('This estimate excludes acceleration and stabilization.')
```

Ожидаемый вывод этой программы (его получили настоящим запуском):

```
Height: 1.2 m
Speed: 0.4 m/s
Ideal travel time: 5.0 s
This estimate excludes acceleration and stabilization.
```

Если у вас то же – вы всё сделали правильно. Если нет, сравните строку за строкой: ошибка почти всегда в одном символе.

`height = 1.2` создаёт имя height и связывает его с числом. Знак `=` означает присваивание. Справа вычисляется выражение, результат сохраняется слева. `distance / speed` – деление; `print` выводит результат. Число с десятичной точкой – тип `float`; целое, например 4, – `int`.

Математическая оценка $t = s/v$ даёт 5 секунд для 2 м при 0.4 м/с. Коптер разгоняется, тормозит и стабилизируется, поэтому реальное время миссии может быть больше. Хорошая программа не выдаёт приближение за точное измерение.

• Задание

Измените расстояние и скорость, предскажите время, затем запустите. **Проверка:** время равно расстоянию, делённому на скорость.

6.2 Арифметика: пять действий, которые понадобятся

Python выполняет арифметические действия, причём для деления существуют три варианта. Результат каждой строки указан в комментарии:

```
print(7 + 2)    # 9    сложение
print(7 - 2)    # 5    вычитание
print(7 * 2)    # 14   умножение
print(7 / 2)    # 3.5  обычное деление, результат всегда с точкой
print(7 // 2)   # 3    целочисленное деление: целая часть
print(7 % 2)    # 1    остаток от деления
print(2 ** 3)   # 8    степень: 2 в степени 3
```

Всё после символа `#` – комментарий: Python его не выполняет, он нужен человеку. Остаток `%` полезен для чередования: `row % 2 == 0` верно для чётных строк змейки (глава 15). Порядок действий, как в математике: сначала степень, затем умножение и деление, затем сложение и вычитание; скобки меняют порядок.

Предскажите и проверьте

Не запуская программу, запишите, что напечатают `print(10 % 3)`, `print(2 + 3 * 4)` и `print((2 + 3) * 4)`. Затем проверьте. Расхождение означает, что порядок действий нужно повторить. Ответы: 1, 14, 20.

6.3 Строки, логические значения и единицы

`'body'` – строка, тип `str` (последовательность символов). Кавычки показывают, что это текст, а не имя переменной. В дальнейшем так записываются названия систем координат, например `'body'` (система, связанная с дроном) и `'map'` (система координат поля); подробно – в главе 10. `True` и `False` – логические значения («истина» и «ложь»), тип `bool`. Они нужны для ответов «да/нет»: например, «есть ли связь с дроном?» или параметр `auto_arm` команд полёта (разрешение автоматически включить моторы перед взлётом; он разбирается в главе 13). В Python их пишут с большой буквы; в launch-файлах (текстовых файлах запуска, глава 7) действуют другие правила записи.

Имя `height` не сообщает Python, что число измеряется в метрах. Единицы определяются интерфейсом и вашей договорённостью. Ошибка «передать 100 вместо 1, потому что думали в сантиметрах» вполне возможна. В отчёте подписывайте единицы, а в коде используйте понятные имена: `height_m`, `speed_mps`, если это помогает избежать путаницы.

```
height_m = 1.0
height_cm = height_m * 100
connected = True
frame = 'map'
print(type(height_m), type(connected), type(frame))
print(f'Height: {height_cm:.0f} cm')
```

Вывод:

```
<class 'float'> <class 'bool'> <class 'str'>
Height: 100 cm
```

Строка с буквой `f` перед кавычкой называется f-строкой: в неё можно вставлять значения переменных в фигурных скобках. Запись `:.0f` выводит число без знаков после точки, `:.2f` – с двумя. Форматирование изменяет отображение, а не точность самой переменной.

• Задание

Объявите `height_m = 1.5` и выведите высоту в сантиметрах. **Проверка:** выведено 150.

6.4 Условия: программа выбирает действие

Условия отвечают на вопросы вида: есть ли связь с дроном (поступают ли от него данные), получены ли координаты, нужно ли продолжать? Оператор сравнения == проверяет равенство; = присваивает. Операторы <, >, <=, >= сравнивают числа.

```
height = 0.6
if height < 0.8:
    print('Below the selected band')
elif height > 1.2:
    print('Above the selected band')
else:
    print('Inside the selected band')
```

После условия стоит двоеточие. Отступ показывает, какие строки относятся к ветке. Используйте четыре пробела. `elif` проверяется, если предыдущая ветка не подошла. `else` выполняется, если не подошло ни одно условие.

`and` требует выполнения обеих проверок, `or` – хотя бы одной, `not` меняет логическое значение. Например, `connected and position_valid` означает «есть связь (`connected`) и есть корректная позиция (`position_valid`)»; обе переменные в этом примере принимают значения `True` или `False`. Такой смысл легко объяснить словами перед написанием кода.

6.4.1 Почему неизвестная высота – отдельный случай

В вычислениях может появиться `NaN`, not a number. Это не нулевая высота. Сравнения NaN с обычными числами не ведут себя как проверка реального положения. Для координат сначала используют `math.isfinite`: оно отбрасывает NaN и бесконечности. В файле ниже строка `import math` подключает стандартную библиотеку математических функций (подробнее о подключении – в разделе «Модули»).

```
#!/usr/bin/env python3
"""Offline: classify artificial observations; no flight commands."""
import math


def classify(height, connected):
    if not connected:
        return 'NO_LINK'
    if not math.isfinite(height):
        return 'NO_POSITION'
    if height < 0.8:
        return 'BELOW_BAND'
    if height > 1.2:
        return 'ABOVE_BAND'
    return 'IN_BAND'


if __name__ == '__main__':
    samples = [(0.6, True), (1.0, True), (1.4, True),
               (float('nan'), True), (1.0, False)]
    for height, connected in samples:
        print(height, connected, '->', classify(height, connected))
```

Ожидаемый вывод файла `conditions.py`:

```
0.6 True -> BELOW_BAND
1.0 True -> IN_BAND
1.4 True -> ABOVE_BAND
nan True -> NO_POSITION
1.0 False -> NO_LINK
```

Запустите файл и сопоставьте каждую строку вывода с веткой функции. Сначала проверяется связь, затем корректность числа, затем диапазон. Такой порядок предотвращает «успешное» решение на недостоверных данных.

Условие и граница

Предскажите результаты для 0.8, 1.2 и 1.21 м. Затем замените диапазон на 0.9–1.1 м. Проверьте обе границы и NaN. Для дополнительного задания объясните, почему проверка `height != 0` не подтверждает правильность координаты.

6.5 Списки и цикл for

Когда точек много, хранить каждую в отдельной переменной неудобно. **Список** хранит последовательность значений. **Кортеж** объединяет значения одной точки; здесь это пара X и Y.

```
points = [(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]
for x, y in points:
    print('Next point:', x, y)
```

Вывод (пять строк, по одной на точку):

```
Next point: 0 0
Next point: 1 0
Next point: 1 1
Next point: 0 1
Next point: 0 0
```

Цикл берёт по одному элементу. `x, y` распаковывает пару. Порядок элементов определяет порядок посещения точек. Скобки `[]` обозначают список, `()` – кортеж; для первых программ важнее понимать назначение этих структур, чем запоминать все их различия.

`range(4)` выдаёт индексы 0, 1, 2, 3. Правая граница не включается. Если нужен и номер, и точка, используйте `enumerate`:

```
for index, (x, y) in enumerate(points):
    print(index, x, y)
```

Вывод: номер, затем координаты. Заметьте, что номер первой точки – 0.

```
0 0 0
1 1 0
2 1 1
3 0 1
4 0 0
```

Нумерация с нуля станет полезной для журнала: вы сможете увидеть, на каком переходе возникла ошибка.

• Задание

Составьте список из трёх вершин треугольника и выведите их с номерами. **Проверка:** три строки с номерами 0, 1, 2.

6.6 Цикл while и остановка ожидания

`for` удобен, когда известна последовательность. `while` повторяет действия, пока условие истинно. Он нужен для ожидания цели, но бесконечный цикл не должен стать скрытой ошибкой.

```
import time
deadline = time.monotonic() + 3
while time.monotonic() < deadline:
    print('Waiting...')
    time.sleep(0.5)
print('Finished')
```

`time.sleep` делает паузу. `time.monotonic` даёт часы для измерения прошедшего времени; нас интересует разность показаний. Мы заранее рассчитали крайний срок и на каждой итерации его проверяем. `break` немедленно завершает ближайший цикл, `continue` переходит к следующей итерации.

Позднее, в программах для ROS, к условию цикла добавляют `not rospy.is_shutdown()`. Здесь `rospy` – библиотека Python для ROS (глава 7), а `is_shutdown()` возвращает `True`, когда система просит программу завершиться; цикл тогда прекращается. В этой главе ROS не используется.

• Задание

Измените срок ожидания на 2 секунды. Сколько раз выведется `Waiting...`? **Проверка:** примерно 4 раза (по одному на каждые 0.5 с).

6.7 Функции: один алгоритм, разные значения

```
def travel_time(distance, speed):
    if speed <= 0:
        raise ValueError('Speed must be positive')
    return distance / speed

print(travel_time(2.0, 0.4))
print(travel_time(distance=1.2, speed=0.3))
```

Вывод: `5.0` и `4.0`. Вызов `travel_time(2.0, 0.4)` подставил 2.0 вместо `distance` и 0.4 вместо `speed`.

`def` создаёт функцию. В скобках – её параметры. `return` возвращает результат и завершает функцию. При вызове можно передавать значения по порядку или по именам. Именованные аргументы помогают читать полётный код: `z=1, frame='body'` яснее, чем несколько чисел без пояснения.

`raise` сообщает об ошибке. Здесь деление при нулевой скорости не имеет смысла, поэтому мы останавливаем расчёт с понятным текстом. Лучше обнаружить неверный параметр до запуска миссии, чем получить странный результат в воздухе.

• Задание

Напишите функцию `speed(distance, time)`. Чему равен результат `speed(2, 5)`? **Проверка:** 0.4.

6.8 Модули: подключаем готовую работу

Файл с функциями можно использовать в другой программе. `import math` подключает стандартную библиотеку математических функций. `from route_math import snake` подключает функцию из соседнего файла `route_math.py`.

Конструкция `if __name__ == '__main__':` отделяет демонстрацию от функций. Когда вы напрямую запускаете файл, демонстрация выполняется. Когда импортируете его как модуль, функции становятся доступны, а демонстрация не запускается. Так генератор маршрута можно проверять отдельно и использовать в полётной миссии.

• Задание

Импортируйте `polygon` и `length` из `route_math` и выведите длину треугольника. **Проверка:** 3.118 (с округлением до тысячных).

6.9 Объект и атрибут без сложной теории

Объект – переменная, которая хранит несколько связанных значений и умеет выполнять действия. Значения называются *полями* (атрибутами), действия – *методами*. Точка означает «взять поле или метод этого объекта». Пример из следующих глав: *телеметрия* (данные о состоянии дрона) приходит в программу как объект `state`; у него есть поля `state.x`, `state.z` (координаты) и `state.armed` (включены ли моторы). Вызов `flight.go(...)` запускает метод `go` объекта `flight` – это команда «лететь в точку».

Класс – описание того, как создать объекты определённого вида и какие действия они умеют. Класс `Flight` (его написал автор книги, он разбирается в главе 14) описывает объект-помощник для полётных команд. Для запуска миссии собственный класс писать не требуется. Сначала следует научиться читать его интерфейс: какие аргументы передать, что вернётся и какие ошибки возможны. Затем можно разобрать реализацию.

• Задание

Пусть `state` – результат телеметрии. Запишите, как получить высоту и признак armed. **Проверка:** `state.z` и `state.armed`.

6.10 Ошибки и завершение: try, except, finally

```
try:
    result = travel_time(1, 0)
except ValueError as error:
    print('Invalid input:', error)
finally:
    print('Calculation attempt ended')
```

Вывод (две строки; обратите внимание, что `finally` выполнился тоже):

```
Invalid input: Speed must be positive
Calculation attempt ended
```

`try` содержит работу, которая может не получиться. `except` обрабатывает выбранный тип ошибки. `finally` выполняется при обычном выходе и при исключении из блока. В полётных помощниках этот механизм используют для попытки посадки после миссии. Однако исчезнувший сервис ROS (программа, отвечающая на запросы дрона; глава 7) не восстановится только благодаря слову finally.

• Задание

Вызовите `travel_time(1, 0)` без `try` и прочитайте сообщение снизу вверх. **Проверка:** последняя строка – `ValueError: Speed must be positive`.

6.11 Как переводить блоки в Python

Blockly (визуальное программирование блоками, глава 11)	Python	Смысл
Число и переменная	`height = 1.0`	Сохранить настройку
Если / иначе	`if / else`	Выбрать ветку
Повторить 4 раза	`for i in range(4)`	Повторить действие
Повторять пока	`while condition`	Ждать или наблюдать
Список точек	`points = [...]`	Хранить маршрут
Процедура	`def visit(...)`	Переиспользовать действия

Blockly помогает увидеть структуру. Python позволяет подробнее контролировать ошибки, данные и ожидание. Сначала соберите понятный алгоритм словами или блоками, затем перенесите его в код. Строка с командой движения – лишь часть программы; проверки и завершение тоже входят в алгоритм.

Мини-проект без дрона

Сделайте программу, которая хранит пять точек, печатает их с номерами и считает время пути при выбранной скорости. Начальный уровень: все отрезки по 1 м. Дополнительный уровень: расстояния считайте по координатам. Проверка: изменение скорости вдвое должно менять идеальную оценку времени вдвое, а нулевая скорость должна давать понятную ошибку.

Итоги и самопроверка

- Переменная – имя значения; = присваивает, == сравнивает.
- if выбирает ветку, for перебирает последовательность, while ждёт условия.
- Функции избавляют от повторов; исключения сообщают об ошибках.

Проверьте себя: 1) Что напечатает `print(7 // 2)`? 2) Чем for отличается от while? *Ответы:* 1) 3; 2) for перебирает известную последовательность, while повторяет, пока условие истинно.

ГЛАВА 7

Лаборатория ROS: связываем программы

Цели главы

- запустить издателя, подписчика, фильтр и сервис и объяснить, что делает каждый;
- различать топик, сервис и параметр и выбирать подходящий механизм;
- наблюдать работу системы командами rostopic, rosservice, rosparam;
- запустить несколько нод одним launch-файлом.

Нужно знать заранее: главы 4 (терминал, окна A, B, C, команда `source`), 5 (установленный стенд) и 6 (функции Python).

7.1 Цель и условия работы

В этой главе дрон не взлетает и симулятор не нужен. Создаётся небольшая система из нескольких программ: искусственный датчик публикует «высоту» (просто числа, меняющиеся по формуле), наблюдатель печатает её, фильтр усредняет значения. Отдельно создаётся сервис «виртуальной лампы». Так основные понятия ROS становятся действующими программами.

Четыре понятия ROS, которые вводятся в этой главе, – коротко:

Понятие	Смысл
Нода	Одна работающая программа, которая участвует в обмене средствами ROS
Топик	Именованный поток сообщений: одни ноды отправляют, другие получают
Сервис	Вопрос и ответ: одна нода просит другую выполнить действие и получает результат
Параметр	Настройка, значение которой хранится в ROS и которое нода может прочитать

Сообщение – порция данных с заранее определённой структурой (например, одно число или одно изображение). Набор имён, по которым программы находят друг друга (имя топика, имя сервиса), называют *интерфейсом* ноды.

Условия работы. Стенд установлен (глава 5); в каждом используемом терминале выполнены команды `source` (глава 4); каталог примеров скопирован в `~/clover_course_files/examples`. Терминалы названы A, B, C, как в главе 4: **окно A** – для долгой программы, которую нельзя прерывать (здесь – `roscore`); **окно B** – для программы, которую вы запускаете в данный момент (издатель, подписчик, фильтр, сервер); **окно C** – для команд наблюдения. Если нужно запустить ещё одну программу, открывают ещё один терминал и называют его следующей буквой (D, E).

У участников лаборатории следующие интерфейсы. Датчик публикует топик `/course/height` с сообщениями типа `std_msgs/Float64` (одно число с плавающей точкой). Фильтр читает этот поток и публикует результат в топик `/course/height_mean`. Лампа предоставляет сервис `/course/set_lamp`. Префикс `/course` отличает учебные имена от имён реальных программ

дрона.

• Задание

Выпишите три учебных интерфейса лаборатории и тип каждого (топик или сервис). **Проверка:** /course/height и /course/height_mean – топики; /course/set_lamp – сервис.

7.2 Нода – программа с одной ролью

Нода – работающий процесс (запущенная программа), который участвует в ROS. Обычный Python-файл становится нодой после вызова `rospy.init_node` (`rospy` – библиотека Python для ROS 1). Имя ноды нужно для регистрации и диагностики. Файл и нода могут называться по-разному: файл `ros_talker.py`, имя ноды `course_talker`.

Две ноды с одним именем конфликтуют: новый запуск может вытеснить прежний. Для нескольких одинаковых наблюдателей используют `anonymous=True`: ROS добавит к имени уникальную часть. Для программы управления дроном стабильное имя помогает найти её в списке. Уникальное имя не разрешает запускать несколько управляющих программ одновременно.

• Задание

Назовите имя файла и имя ноды издателя. **Проверка:** файл ros_talker.py, нода course_talker – имена не обязаны совпадать.

7.3 ROS master – каталог участников

Ноды должны как-то находить друг друга. В ROS 1 для этого есть специальная программа – **master**. Она ведёт «справочник»: какая нода какие топики и сервисы предоставляет. Команда `roscore` запускает master и вспомогательные службы. Когда издатель и подписчик нашли друг друга, сообщения идут между ними напрямую: master не пересылает каждое сообщение.

В окне A подключите окружение и запустите master:

```
source /opt/ros/noetic/setup.bash
source "$HOME/catkin_ws/devel/setup.bash"
roscore
```

Окно A остаётся занятым: в нём идёт вывод master. В остальных окнах выполните те же две команды `source`. Если master уже запущен (например, симулятором, который запускает его сам), второй `roscore` не нужен.

В ROS есть служба часов. По умолчанию нода берёт время из часов компьютера. Если же параметр `/use_sim_time` равен `true`, время берётся из топика `/clock`, который публикует симулятор: так время симуляции может идти медленнее или быстрее реального. В этой главе симулятора нет, поэтому параметр должен быть `false`; иначе функции, которые ждут по часам (в том числе `rospy.Rate` ниже), будут ждать часы, которых нет. Параметры читают и меняют командой `rosparam`; `rosnode list` показывает список работающих нод.

```
rosparam set /use_sim_time false
rosnode list
```

Эти команды вводятся в окне C. Если до этой главы запускался симулятор, остановите его (Ctrl+C в его окне) и перед полётными опытами убедитесь, что симулятор снова включил `/use_sim_time=true`.

• Задание

Запустите roscore (окно A) и выполните `rosnode list` в окне C, затем запустите издатель (следующий раздел) и повторите. **Проверка:** после запуска издателя в списке появилась его нода.

7.4 Топик – именованный поток

Топик похож на радиоканал с названием: одна нода (*издатель*, publisher) отправляет в него сообщения, другие (*подписчики*, subscriber) получают. Все сообщения топика имеют один тип. Подписчиков может быть несколько: наблюдатель, фильтр и программа записи потока (rosbag, глава 23) могут читать один и тот же топик.



Рис. 7.1. Схема обмена. Добавление второго подписчика не требует изменения издателя.

Топик не хранит таблицу последних значений. Сообщения приходят во времени; подписчик, подключившийся позже, не получает прежние сообщения автоматически. Для истории нужна отдельная запись. Если нужно дождаться одного следующего сообщения, в `rospy` есть функция `wait_for_message`.

• Задание

Нарисуйте издателя и двух подписчиков одного топика. **Проверка:** у издателя одна исходящая связь с топиком; подписчики не знают друг о друге.

7.5 Первый издатель: искусственная высота

В окне B перейдите в каталог примеров и запустите файл (окно B – это терминал, в котором работает запускаемая сейчас программа):

```
cd "$HOME/clover_course_files/examples"
python3 ros_talker.py
```

```
#!/usr/bin/env python3
"""ROS 1 laboratory: artificial altitude on /course/height."""
import math
import rospy
from std_msgs.msg import Float64

rospy.init_node('course_talker')
pub = rospy.Publisher('/course/height', Float64, queue_size=10)
rate = rospy.Rate(5)
index = 0
try:
    while not rospy.is_shutdown():
        height = 1.0 + 0.3 * math.sin(index * 0.2)
        pub.publish(Float64(data=height))
        rospy.loginfo('Artificial height: %.3f m', height)
        index += 1
        rate.sleep()
except rospy.ROSInterruptException:
    pass
```

`Publisher` создаёт издателя; ему передают имя топика, тип сообщений и размер очереди. `queue_size=10` – ограниченная очередь отправки: если подписчик не успевает, старые сообщения отбрасываются, а не накапливаются бесконечно. `Rate(5)` создаёт «метроном» с частотой 5 Гц (герц, Гц – число повторений в секунду): пять итераций цикла в секунду. `rate.sleep()` ждёт до следующего такта.

Формула $1 + 0.3\sin(0.2i)$ даёт плавно меняющиеся значения от 0.7 до 1.3. Это искусственные числа, а не показания дрона. `Float64(data=height)` упаковывает число в сообщение согласованного типа. `rospy.loginfo` печатает текст в терминал и в журнал ROS.

В окне C выполните команды наблюдения:

```
rostopic list
rostopic type /course/height
rosmsg show std_msgs/Float64
rostopic echo -n 3 /course/height
rostopic hz /course/height
```

Команда	Что делает
`rostopic list`	Показывает список всех существующих топиков
`rostopic type /course/height`	Показывает тип сообщений топика
`rosmsg show std_msgs/Float64`	Показывает структуру сообщения этого типа (список полей)
`rostopic echo -n 3 /course/height`	Печатает сообщения топика; ключ `-n 3` означает «только три сообщения и остановиться»
`rostopic hz /course/height`	Измеряет, сколько сообщений в секунду приходит по топику; прекращается по Ctrl+C

Ожидается, что в сообщении есть поле `data` и что частота около 5 Гц (без точного совпадения до всех знаков). Если вывода нет, проверьте, что издатель запущен, и точное имя топика.

• Задание

Измените частоту издателя с 5 на 2 Гц и проверьте `rostopic hz`. **Проверка:** частота около 2 Гц.

7.6 Подписчик и callback

```
#!/usr/bin/env python3
"""ROS 1 laboratory: read artificial data, no aircraft connection."""
import rospy
from std_msgs.msg import Float64


def on_height(message):
    rospy.loginfo('Received %.3f m', message.data)


rospy.init_node('course_listener', anonymous=True)
rospy.Subscriber('/course/height', Float64, on_height, queue_size=10)
rospy.spin()
```

Запустите `python3 ros_listener.py` в ещё одном терминале. `Subscriber` связывает входящий поток с функцией `on_height`. Такая функция называется **callback** («обратный вызов»): библиотека сама вызывает её при получении каждого сообщения. Вы передаёте *саму функцию*, поэтому в аргументе `Subscriber` нет скобок: `on_height`, а не `on_height()`.

`message.data` — число внутри сообщения. `rospy.spin()` оставляет программу работающей и обрабатывающей события. Без него программа закончилась бы сразу после создания подписчика. `spin` не публикует данные и не запускает полёт: он только поддерживает работу callback.

Добавьте наблюдателя

Запустите второй listener в новом терминале. Убедитесь, что оба получают данные, а в выводе `rostopic info /course/height` (команда показывает издателей и подписчиков топика) видны оба подписчика. Остановите один listener (Ctrl+C). **Проверка:** издатель и второй listener продолжают работать. Объясните, какое преимущество ROS показал этот опыт.

7.7 Третья нода: скользящее среднее

Фильтр хранит последние несколько значений. Среднее этих значений сглаживает быстрые изменения; это полезно для шумных датчиков, но добавляет запаздывание. На искусственной синусоиде видно, как работает этот компромисс.

```
#!/usr/bin/env python3
"""ROS 1 laboratory: a third node can reuse the same topic."""
from collections import deque
import math
import rospy
from std_msgs.msg import Float64

rospy.init_node('course_filter')
window_size = int(rospy.get_param('~window', 5))
if not 1 <= window_size <= 100:
    raise ValueError('window must be between 1 and 100')
values = deque(maxlen=window_size)
pub = rospy.Publisher('/course/height_mean', Float64, queue_size=10)


def on_height(message):
    if not math.isfinite(message.data):
        rospy.logwarn('Non-finite sample ignored')
        return
    values.append(message.data)
    pub.publish(Float64(data=sum(values) / len(values)))


rospy.Subscriber('/course/height', Float64, on_height, queue_size=10)
rospy.spin()
```

`deque(maxlen=5)` – «окно»: структура, которая сама удаляет самое старое значение при добавлении нового, если значений стало больше пяти. `sum(values) / len(values)` вычисляет среднее. В начале окно ещё не заполнено, поэтому программа делит на фактическое количество значений. Проверка `isfinite` не даёт значению NaN («не число») испортить всё среднее.

Размер окна задаётся при запуске записью `_window:=5` (подробнее – в следующем разделе о параметрах):

```
python3 ros_filter.py _window:=5
rostopic echo /course/height_mean
```

Первая команда запускается в окне B, вторая – в окне C. Запустите фильтр с окнами 1 и 10 (перед новым запуском останавливайте предыдущий фильтр). При окне 1 среднее равно входу. При большом окне изменение становится плавнее, но отстаёт от входного сигнала. Увидеть две кривые одновременно позволяет `rqt_plot` – программа, которая рисует графики значений топиков:

```
rosrun rqt_plot rqt_plot /course/height/data /course/height_mean/data
```

Команда `rosrun` *`пакет программа`* запускает программу из пакета ROS; здесь программа получает названия двух значений для отображения. Если `rqt_plot` не установлен, для первого опыта достаточно `echo`.

• Задание

Запустите фильтр с окном 1, затем 10 и опишите, как меняется результат. **Проверка:** окно 1 повторяет вход; окно 10 сглаживает, но отстаёт.

7.8 Параметр – настройка поведения

Параметр – именованное значение, хранящееся в ROS (в справочнике master). Приватный параметр ноды записывается с тильдой: `~window` означает «параметр window именно этой ноды». Для ноды `course_filter` его полное имя – `/course_filter/window`. Запись `_window:=5` в командной строке при запуске Python-программы с `rospy` задаёт значение приватного параметра.

```
rosparam get /course_filter/window
rosparam list
```

`rosparam get` показывает значение параметра, `rosparam list` – список всех параметров. В этом примере параметр читается один раз при старте. Изменение через `rosparam set` не перестроит уже созданное окно: ноду нужно перезапустить. Параметры – не непрерывные команды управления; программа сама определяет, когда их перечитывать.

• Задание

Найдите путь параметра окна и объясните, почему `rosparam set` после старта ничего не изменит. **Проверка:** /course_filter/window; параметр читается один раз при старте.

7.9 Сервис – один запрос и один ответ

Камера выдаёт поток, а включение лампы естественнее выразить одной операцией: «установи состояние и ответь, получилось ли». Это сервис: нода-клиент отправляет запрос, нода-сервер выполняет действие и возвращает ответ. Запрос содержит входные поля, ответ – результат. В этой работе лампа виртуальная: её состояние хранится в переменной и печатается в терминал.

```
#!/usr/bin/env python3
"""ROS 1 laboratory: virtual lamp; no GPIO and no drone commands."""
import rospy
from std_srvs.srv import SetBool, SetBoolResponse

lamp_on = False


def set_lamp(request):
    global lamp_on
    lamp_on = request.data
    text = 'ON' if lamp_on else 'OFF'
    rospy.loginfo('Virtual lamp: %s', text)
    return SetBoolResponse(success=True, message=text)


rospy.init_node('course_lamp')
rospy.Service('/course/set_lamp', SetBool, set_lamp)
rospy.spin()
```

Тип `std_srvs/SetBool` задаёт логическое поле `data` в запросе и поля `success`, `message` в ответе. Функция `set_lamp` (callback, как и у подписчика) получает запрос и возвращает `SetBoolResponse`. Слово `global` разрешает функции изменить переменную, объявленную снаружи; в большом проекте состояние лучше хранить внутри объекта.

Запустите сервер в окне B, затем в окне C выполните:

```
rosservice list
rosservice type /course/set_lamp
rossrv show std_srvs/SetBool
rosservice call /course/set_lamp "data: true"
rosservice call /course/set_lamp "data: false"
```

`rosservice list` – список сервисов, `rosservice type` – тип сервиса, `rossrv show` – поля запроса и ответа, `rosservice call` – вызвать сервис с заданными значениями полей. Каждый вызов получает ответ. Сервис подходит для разовой операции, но не хранит историю включений для нового наблюдателя. Если нужно постоянно наблюдать состояние лампы, добавляют отдельный топик состояния.

7.9.1 Клиент сервиса на Python

```
#!/usr/bin/env python3
"""Call the virtual lamp service once."""
import rospy
from std_srvs.srv import SetBool

rospy.init_node('course_lamp_client')
rospy.wait_for_service('/course/set_lamp', timeout=5)
set_lamp = rospy.ServiceProxy('/course/set_lamp', SetBool)
result = set_lamp(data=True)
if not result.success:
    raise RuntimeError(result.message)
print('Service replied:', result.message)
```

`wait_for_service` ждёт появления сервера не дольше пяти секунд. `ServiceProxy` создаёт объект, с помощью которого вызывается удалённая функция; это не сам сервер и не доказательство того, что операция выполнена. `result = set_lamp(data=True)` отправляет запрос и получает ответ.

Остановите сервер и повторите запуск клиента: ожидается сообщение об ошибке ожидания, а не бесконечное молчание. Поэтому в полётных программах сначала ждут сервис, а затем проверяют `result.success`. Таймаут ожидания доступности не ограничивает длительность каждого последующего вызова.

• Задание

Вызовите сервис лампы со значениями true и false. **Проверка:** каждый вызов вернул ответ, в терминале сервера видно состояние.

7.10 Тип сообщения – договор о данных

Издатель и подписчик должны согласовать тип сообщения. Строка `'1.2'` и число `1.2` имеют разный смысл для программы. Тип `Float64` содержит число, `String` – текст, `Image` – изображение с описанием (размер, кодировка). Типы позволяют инструментам автоматически показывать структуру сообщения.

Для координат одного числа недостаточно: нужны хотя бы X, Y, Z, система координат и время измерения. Для этого в сообщениях есть `Header` (заголовок: время и название системы

координат), например в типе `PointStamped` – «точка с отметкой». Запустите следующий пример и посмотрите поля сообщения:

```
#!/usr/bin/env python3
"""ROS 1 laboratory: artificial PointStamped with explicit frame/time."""
import rospy
from geometry_msgs.msg import PointStamped

rospy.init_node('course_stamped_talker')
pub = rospy.Publisher('/course/point', PointStamped, queue_size=10)
rate = rospy.Rate(2)
try:
    while not rospy.is_shutdown():
        message = PointStamped()
        message.header.stamp = rospy.Time.now()
        message.header.frame_id = 'course_demo'
        message.point.x = 1.0
        message.point.y = 0.5
        message.point.z = 1.2
        pub.publish(message)
        rate.sleep()
except rospy.ROSInterruptException:
    pass
```

```
rostopic echo -n 1 /course/point
rosmsg show geometry_msgs/PointStamped
```

Название `course_demo` в поле системы координат – просто подпись нашего учебного сообщения. Она не создаёт преобразование между системами координат (TF, глава 10). Если другой программе нужен пересчёт координат, преобразование публикуют отдельно.

• Задание

Выведите `rosmsg show geometry_msgs/PointStamped` и перечислите поля заголовка. **Проверка:** seq, stamp, frame_id.

7.11 Имена, namespace и переназначение

Namespace (пространство имён) – префикс, который группирует имена нод, топиков и параметров, подобно каталогам для файлов.

Запись	Как понимается имя	Пример
`/course/height`	Абсолютное имя от корня	Всегда этот топик
`height`	Относительно namespace ноды	В /demo станет /demo/height
`~debug`	Относительно имени ноды	/course_red_detector/debug

Поэтому сокращённое имя из кода может выглядеть иначе в выводе `rostopic list`: при сомнении надёжнее смотреть список работающей системы. **Переназначение** (remapping) заменяет имя при запуске без редактирования программы:

```
python3 ros_listener.py /course/height:=/course/height_mean
```

Запись `старое:=новое` означает «вместо старого имени используй новое». Теперь тот же наблюдатель читает результат фильтра: интерфейс прежний, источник новый.

• Задание

Нода работает в пространстве имён `/demo` и подписывается на `height`. Какой топик она прочитает? **Проверка:** `/demo/height`.

7.12 Пакет и launch: запускаем систему вместе

Запускать каждую программу в своём терминале неудобно. В ROS для этого есть **launch-файл** – текстовый файл (формат XML: текст с тегами в угловых скобках), в котором перечислено, какие ноды запустить и с какими параметрами. Launch-файл хранится в **пакете** – каталоге с программами, launch-файлами и описанием зависимостей. Пакеты собираются в **рабочем пространстве** (workspace) – каталоге `catkin_ws`, подготовленном в главе 5. **catkin** – система сборки пакетов ROS 1.

Для следующего опыта создаётся пакет `clover_course` и в него копируются три программы и launch-файл:

```
cd "$HOME/catkin_ws/src"
catkin_create_pkg clover_course rospy clover std_msgs
mkdir -p clover_course/scripts clover_course/launch
cp "$HOME/clover_course_files/examples/ros_talker.py" clover_course/scripts/
cp "$HOME/clover_course_files/examples/ros_listener.py" clover_course/scripts/
cp "$HOME/clover_course_files/examples/ros_filter.py" clover_course/scripts/
cp "$HOME/clover_course_files/examples/ros_lab.launch" clover_course/launch/
```

`catkin_create_pkg` создаёт каркас пакета; после имени пакета перечислены его *зависимости* – другие пакеты, которые ему нужны (`rospy`, `clover`, `std_msgs`). `mkdir -p` создаёт каталоги, `cp` копирует файлы. Если пакет уже создан, повторять первую строку не нужно.

Добавьте в начало *каждого из трёх скопированных скриптов* строку `#!/usr/bin/env python3`, если её ещё нет. Эта первая строка (её называют shebang) сообщает Linux, какой программой запускать файл: без неё `roslaunch` не сможет его выполнить. Затем:

```
chmod +x clover_course/scripts/ros_*.py
cd "$HOME/catkin_ws"
catkin_make -j2
source devel/setup.bash
roslaunch clover_course ros_lab.launch window:=7
```

`chmod +x` разрешает запускать файл как программу. `catkin_make` собирает рабочее пространство (ключ `-j2` – использовать два потока). `source devel/setup.bash` подключает собранное. `roslaunch` *`пакет файл`* запускает launch-файл; запись `window:=7` задаёт его аргумент. Перед последней командой остановите отдельно запущенные talker, filter и listener. Если пакет уже существовал, проверьте в его файле `package.xml` (описании пакета), что среди зависимостей указаны `rospy` и `std_msgs`.

```
<launch>
  <arg name="window" default="5"/>
  <node pkg="clover_course" type="ros_talker.py" name="course_talker" output="screen"/>
  <node pkg="clover_course" type="ros_listener.py" name="course_listener" output="screen"/>
  <node pkg="clover_course" type="ros_filter.py" name="course_filter" output="screen">
    <param name="window" value="$(arg window)" type="int"/>
  </node>
</launch>
```

Тег `node` указывает пакет, исполняемый файл и имя ноды. `param` внутри `node` создаёт приватный параметр. `arg` задаёт аргумент launch-файла, который затем подставляется в `param`. Это разные уровни: аргумент помогает организовать запуск, параметр доступен самой ноде.

• Задание

Запустите систему одним launch-файлом с окном 7 и проверьте параметр. **Проверка:** три ноды работают, параметр окна равен 7.

7.13 Когда нужны топик, сервис и параметр

Задача	Механизм	Причина
Поступают кадры камеры	Топик	Непрерывный поток
Назначить следующую цель	Сервис	Отдельный запрос с ответом
Выбрать размер окна фильтра	Параметр	Настройка алгоритма
Передавать события миссии	Топик	Несколько наблюдателей
Долгая задача с обратной связью и отменой	Action	Специальный шаблон длительной операции

Action (действие) – в ROS шаблон для долгих операций с промежуточными отчётами и возможностью отмены. Полётный интерфейс Clover в этой книге использует сервисы, поэтому завершение перехода отслеживается отдельно по данным о положении. Механизм выбирают по характеру данных и доступному интерфейсу.

Итоговая проверка лабораторной работы

Запустите три ноды, покажите тип входного сообщения, частоту и параметр фильтра. Подключите наблюдатель к другому топику через remapping. Вызовите сервис лампы. Объясните, чем поток высоты отличается от одного ответа лампы. Дополнительное задание: добавьте ноду, печатающую «выше 1.2 м», не меняя издателя.

Источник: ROS, официальные учебные примеры rospy: публикация, сервисы и параметры. Проверено 11.10.2026.

Переход к полётным опытам. Остановите учебные ноды и launch (Ctrl+C). Для полётных опытов запускается симулятор (глава 8); он сам включает `/use_sim_time=true` и запускает master, поэтому отдельный `roscore` должен быть остановлен.

Итоги и самопроверка

- Топик – поток; сервис – запрос и ответ; параметр – настройка.
- Подписчик получает данные через callback; `rospy.spin()` не даёт программе завер-

шиться.
- Имена бывают абсолютные и относительные; remapping заменяет имя без правки кода.
- Launch-файл запускает несколько нод с параметрами одной командой.

Проверьте себя: 1) Что выбрать для кадров камеры: топик, сервис или параметр? 2) Для чего нужен rospy.spin()? 3) Что делает команда rostopic hz?
Ответы: 1) топик; 2) не даёт программе завершиться и обрабатывает callback; 3) измеряет частоту сообщений топика.

ГЛАВА 8

Первый запуск и приёмка стенда

Цели главы

- запустить симулятор и понять, из каких программ он состоит;
- проверить командами ROS, что система готова к полёту (приёмка стенда);
- объяснить, что означает каждый показатель приёмки: связь, часы, позиция, камера;
- знать порядок одного опыта и почему независимые опыты начинают с перезапуска.

Нужно знать заранее: главы 4 (терминал, окна A, B, C), 5 (установленный стенд) и 7 (ноды, топики, сервисы, команды `rostopic`, `rosservice`, `rosparam`).

8.1 Запускаем весь стек

Стек – набор программ, работающих вместе (глава 2). Для его запуска используется launch-файл (глава 7) пакета `clover_simulation`. В **окне A** (глава 4) выполните:

```
source /opt/ros/noetic/setup.bash
source "$HOME/catkin_ws/devel/setup.bash"
roslaunch clover_simulation simulator.launch
```

`roslaunch` запускает launch-файл `simulator.launch`: он поднимает master, симулятор Gazebo, автопилот PX4 в режиме SITL, переводчик MAVROS и пакет Clover. Окно A остаётся занятым: оно показывает сообщения всех этих программ. В некоторых сборках в нём доступна и консоль PX4 (командная строка автопилота). Остальные команды главы вводятся в **окне C** (диагностика) и **окне B** (Python). Если для запуска используется ярлык виртуальной машины «Gazebo PX4» или «Gazebo Clover», он уже запускает launch-файл: второй экземпляр не запускайте.

Первый запуск может занять до минуты. Когда симулятор готов, появляется окно Gazebo. Его основные элементы: центральная 3D-сцена с моделью дрона и площадкой; панель инструментов сверху; внизу – кнопка паузы/пуска (Play/Pause), счётчик времени симуляции и показатель **RTF** (Real Time Factor) – отношение скорости течения времени в симуляции к реальному времени. RTF = 1 означает, что симуляция идёт в реальном темпе; меньше 1 – медленнее, потому что компьютеру не хватает мощности.

Источник: COEX, запуск симулятора. Проверено 11.10.2026.

• Задание

Запустите симулятор и найдите в окне Gazebo модель дрона, кнопку паузы и значение RTF. **Проверка:** в окне A нет строк с ERROR, в Gazebo видна модель на площадке, время симуляции идёт.

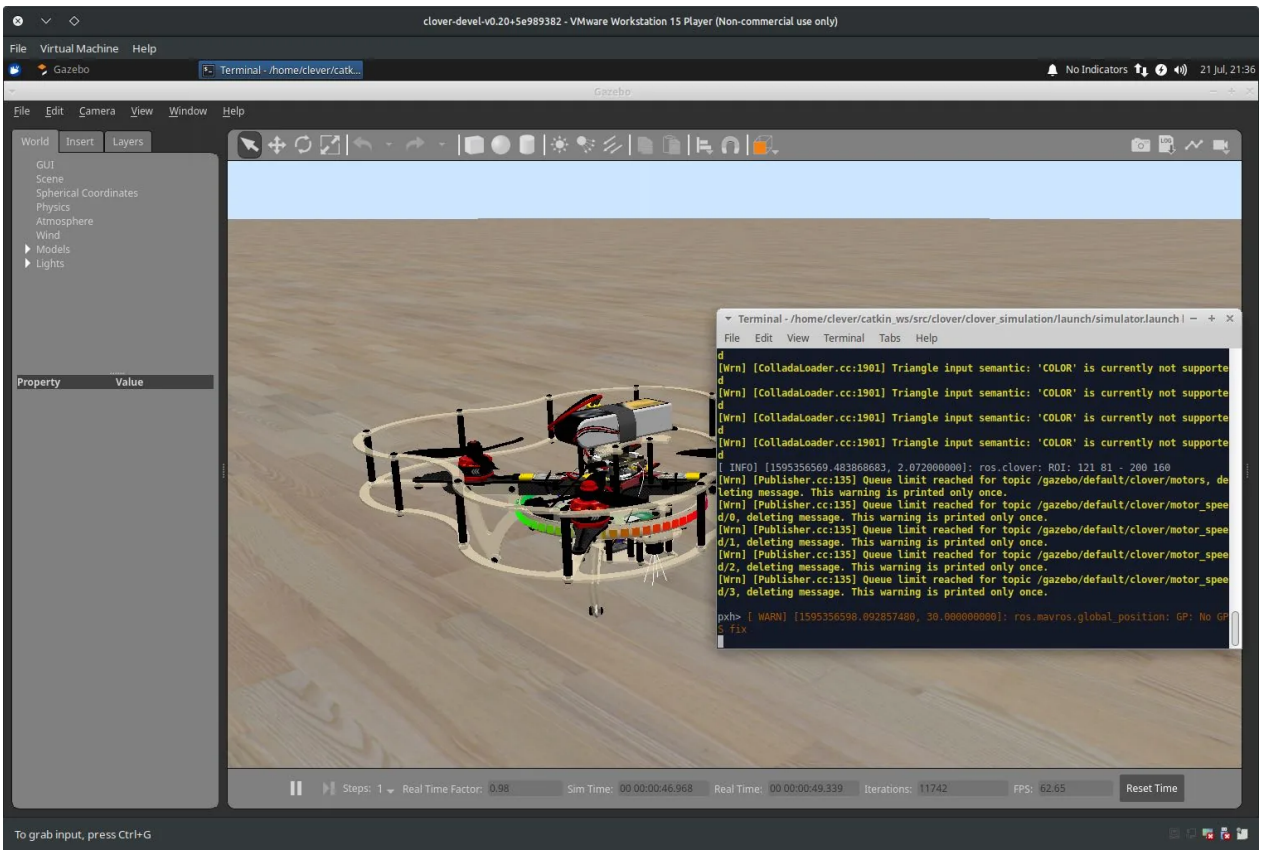


Рис. 8.1. Классический симулятор Клевера. Снимок COEX из раздела «Использование симулятора», CC BY-NC-SA 4.0. Внешний вид вашего образа может отличаться.

8.2 Сначала проверьте связь

Приёмка стенда – проверка того, что все программы стека работают и готовы к полёту. В **окне C** выполните команды (перед ними там тоже нужны две команды `source`, глава 4):

```
rosnode list
rosservice list
rosservice info /navigate
rosservice call /get_telemetry "{frame_id: 'map'}"
rosparam get /use_sim_time
rostopic echo -n 1 /clock
rostopic echo -n 1 /mavros/state
```

Эти команды разобраны в главе 7; здесь важно, *что именно мы спрашиваем у системы*.

Команда	Вопрос к системе	Хороший ответ
`rosnode list`	Какие ноды запущены?	Длинный список; есть `/mavros`, `/gazebo`
`rosservice list`	Какие сервисы доступны?	В списке есть `/navigate` (лететь в точку), `/get_telemetry` (получить данные о состоянии), `/land` (посадка)
`rosservice info /navigate`	Какая нода отвечает на сервис и какие у него аргументы?	Указана нода Clover и список аргументов
`rosservice call /get_telemetry "{frame_id: 'map'}"`	Где дрон в системе координат `map` (система координат поля, глава 10)?	Числа `x, y, z`, `connected: True`
`rosparam get /use_sim_time`	Берётся ли время из часов симулятора?	`true`
`rostopic echo -n 1 /clock`	Идёт ли время симуляции?	Одно сообщение с временем; если ответа нет – нажмите Play в Gazebo
`rostopic echo -n 1 /mavros/state`	Связан ли автопилот с ROS?	`connected: True`

Результат сервиса `get_telemetry` называется **телеметрией**: это данные о состоянии дрона. Основные поля: `connected` – есть ли связь с автопилотом (должно быть `true`); `armed` – включены ли моторы (перед полётом обычно `false`); `mode` – режим автопилота (перед стартом может быть любым); `x, y, z` – координаты дрона в выбранной системе координат (метры), которые должны быть конечными числами. Значение `NaN` («не число») в координате означает, что данных нет.

Что считается готовностью

Окно Gazebo и красивая модель доказывают только то, что работает графика. Для полёта нужны: работающий PX4, связь MAVROS с ним, сервис `navigate`, доступная локальная позиция (координаты не `NaN`) и идущие часы. Подставлять вместо `NaN` ноль нельзя: это означало бы, что дрон лежит в начале координат.

• Задание

Выполните команды из таблицы и заполните собственную таблицу: команда, что увидели, «норма / не норма». **Проверка:** `connected` равно true, `use_sim_time` равно true, координаты – конечные числа.

8.3 Камера и оценка позиции

Бортовая камера дрона публикует кадры в топик. Проверка камеры и самопроверка Клевера (окно C):

```
rostopic list
rostopic hz /main_camera/image_raw
rostopic echo -n 1 /main_camera/camera_info
rosrun rqt_image_view rqt_image_view
rosrun clover selfcheck.py
```

`rostopic list` показывает все топики; нужен топик `/main_camera/image_raw`: `main_camera` – имя основной камеры Клевера, `image_raw` – «сырые» кадры без обработки. `rostopic hz` измеряет частоту кадров: она должна быть заметно больше нуля. `camera_info` содержит параметры камеры – размер кадра, фокусное расстояние. `rqt_image_view` открывает окно, в котором выбирается топик с изображением и показывается картинка. `selfcheck.py` – готовая программа самопроверки Клевера: она проверяет основные системы и сообщает, что нашла.

В `rqt_image_view` выберите нужный топик из выпадающего списка. Если камера работает в другом пространстве имён (глава 7), перед именем появится префикс; точное название определяют по `rostopic list`. `selfcheck.py` может сообщить о периферии, которой нет в симуляторе; внимание следует уделить сообщениям о связи и о позиции.

Позиция дрона вычисляется автопилотом по показаниям датчиков (в симуляторе – виртуальных); этот процесс называют *оценкой позиции*. Пока оценка не готова, координаты в телеметрии равны `NaN`.

• Задание

Сохраните скриншот окна `rqt_image_view` и запишите частоту из `rostopic hz`. **Проверка:** частота больше нуля, изображение не чёрное.

8.4 Рабочий цикл одного опыта

1. Запустите стенд и дождитесь готовности (приёмка выше). Сохраните исходную конфигурацию.
2. Если опыт требует измерений, запустите запись телеметрии или rosbag (инструмент записи сообщений в файл, глава 23).
3. Выполните ровно одну программу (окно B), наблюдайте поведение в Gazebo.
4. Убедитесь, что посадка завершилась и аппарат disarmed (моторы выключены, `armed=false`).
5. Сохраните результаты и остановите запись.
6. Перед новым независимым опытом перезапустите весь launch-стек (Ctrl+C в окне A и повторный запуск).

В меню Gazebo есть команды Reset World и Reset Simulation (вернуть мир или всю симуляцию к начальному состоянию). Если выполнить их во время полёта, часы, положение в Gazebo и оценка положения в PX4 могут рассогласоваться. Поэтому для независимых повторов перезапускают launch. Объекты, добавленные в работающий мир командой `spawn` (добавить модель в симулятор), после перезапуска исчезнут; их описывают в файле мира или в файле модели формата SDF (текстовое описание объекта для Gazebo, глава 18).

Практика. Приёмочный протокол

Сохраните телеметрию, значение `/use_sim_time`, кадр камеры и версию Clover. Дважды пройдите цикл запуска и остановки. **Проверка:** связь, камера и позиция доступны до взлёта в обоих повторах.

Итоги и самопроверка

- Симулятор – набор программ: Gazebo, PX4, MAVROS, Clover; запускается одним launch-файлом в окне A.
- Красивое окно Gazebo ещё не означает готовность: нужны PX4, MAVROS, сервис navigate, позиция и идущие часы.
- NaN значит «данных нет»; заменять его нулём нельзя.
- Независимые опыты начинают с перезапуска всего launch.

Проверьте себя: 1) Назовите четыре условия готовности к полёту. 2) Что означает NaN в координате? 3) Почему нельзя «сбросить мир» посреди полёта? 4) Что показывает RTF? *Ответы:* 1) PX4 работает, связь MAVROS, сервис navigate доступен, позиция конечна и часы идут; 2) данные недоступны; 3) рассогласуются часы, Gazebo и оценка PX4; 4) во сколько раз время симуляции идёт быстрее или медленнее реального.

ГЛАВА 9

Конфигурация Клевера: clover.launch и aruco.launch

Цели главы

- объяснить, как запускается программное обеспечение Клевера и где хранятся его настройки;
- назвать значение по умолчанию и назначение каждого флага в `clover.launch` и `aruco.launch`, в том числе почему Blockly и ArUco по умолчанию выключены;
- безопасно изменить флаг на реальном дроне и перезапустить службу;
- подобрать набор флагов под задачу: Blockly, распознавание маркеров, навигация по карте, маркеры на потолке.

Нужно знать заранее: главы 4 (терминал), 7 (launch-файлы, ноды) и 8 (приёмка стенда).

9.1 Откуда берётся поведение Клевера

Когда Raspberry Pi на Клевере загружается, готовый образ (глава 3) автоматически запускает пакет Clover. Он оформлен как **служба** (service) операционной системы Linux с именем `clover` – программа, которая запускается при старте компьютера и работает в фоне. Служба выполняет launch-файл `clover.launch`, а тот в свою очередь запускает вложенные launch-файлы (MAVROS, камера, ArUco, светодиодная лента) и отдельные ноды (`simple_offboard`, веб-сервисы). Поэтому **набор работающих программ определяется содержимым launch-файлов**: если в них нужная часть выключена, соответствующей ноды в системе просто нет, и никакая команда её не заменит.

Именно поэтому у начинающих часто возникает ситуация «страница Blockly не открывается» или «ArUco-маркеры не распознаются» при полностью исправном дроне: в поставляемой конфигурации ряд подсистем отключён и включается правкой одного флага.

• Задание

Объясните своими словами, почему при отсутствии ноды `clover_blocks` страница Blockly не работает, хотя дрон исправен. **Проверка:** нужная нода не запущена, потому что флаг `blocks` в launch-файле выключен.

9.2 Где лежат файлы

На образе Клевера все файлы пакетов лежат в рабочем пространстве пользователя `pi` (глава 4):

Файл (в каталоге ~/catkin_ws/src/clover/)	Назначение
clover/launch/clover.launch	Главный launch-файл: что запускать
clover/launch/aruco.launch	Настройки распознавания маркеров и карты
clover/launch/main_camera.launch	Драйвер камеры: разрешение, частота, экспозиция
clover/launch/led.launch	Светодиодная лента
aruco_pose/map/	Каталог файлов карт маркеров
clover_blocks/programs/	Сохранённые программы Blockly (файлы XML)

В симуляторе те же файлы находятся в вашем рабочем пространстве `catkin_ws`; запуск симулятора обёрнут собственным launch-файлом (глава 18), поэтому настройки симулятора передают аргументами этого файла.

9.3 Как работают флаги launch-файла

В launch-файле флаг – это **аргумент**: строка вида

```
<arg name="blocks" default="false"/>
```

Здесь `name` – имя флага, `default` – значение по умолчанию (используется, если при запуске значение не указано). Булевы флаги принимают значения `true` и `false` (в нижнем регистре: это XML, а не Python). Дальше в файле значение используется условием, например:

```
<node name="clover_blocks" pkg="clover_blocks" type="clover_blocks"
      output="screen" if="$(arg blocks)"/>
```

Атрибут `if="$(arg blocks)"` означает: «запусти эту ноду, только если флаг `blocks` равен `true`». Значение флага можно задать двумя способами.

1. **Навсегда**: изменить `default` в файле и перезапустить службу.
2. **На один запуск**: передать значение в командной строке, например `roslaunch clover clover.launch blocks:=true` (так книга включает Blockly в симуляторе, глава 11). На работающем дроне служба `clover` уже запущена автоматически, поэтому там меняют файл.

9.4 Подключение к дрону и редактирование файла

Файл правят на самом бортовом компьютере. По документации COEX образ Клевера по умолчанию раздаёт сеть Wi-Fi с именем `clover-xxxx` (четыре цифры – случайные, назначаются при первом включении) и паролем `cloverwifi`; бортовой компьютер доступен по адресу `192.168.11.1`. Эти данные относятся к стандартному образу и могут быть изменены; проверяйте их по документации вашей версии.

1. Подключите ноутбук к сети Wi-Fi дрона.
2. Откройте терминал дрона: либо командой `ssh pi@192.168.11.1` (пользователь `pi`, пароль по умолчанию в стандартном образе `raspberry`; `ssh` – защищённое удалённое подключение, глава 3), либо веб-терминалом на странице `http://192.168.11.1`, кнопка «Open web terminal».
3. Сделайте резервную копию файла и откройте его в текстовом редакторе `nano`:

```
cd ~/catkin_ws/src/clover/clover/launch
cp clover.launch clover.launch.bak
nano clover.launch
```

В `nano` стрелками переходят к нужной строке, меняют значение, сохраняют сочетанием Ctrl+O (затем Enter) и выходят Ctrl+X. После правки службу перезапускают:

```
sudo systemctl restart clover
sudo systemctl status clover
```

Первая команда перезапускает пакет Clover (`sudo` – выполнить с правами администратора), вторая показывает состояние службы: ожидается `active (running)`. Перезапуск занимает порядка десяти секунд, затем нужно повторить проверки (`selfcheck.py`, глава 8). Если после правки служба не запускается, чаще всего ошибка в синтаксисе XML (потерянная кавычка или угловая скобка): верните резервную копию командой `cp clover.launch.bak clover.launch` и повторите правку.

Перед правкой на реальном дроне

Правку и перезапуск выполняйте только при выключенных моторах, без пропеллеров или с отключённой батареей двигателей, на земле. Меняйте один флаг за раз и после каждого изменения проверяйте результат.

Практика. Резервная копия и откат

На стенде (или на копии файла в симуляторе) создайте `clover.launch.bak`, измените значение одного флага, перезапустите, убедитесь в изменении поведения (например, появилась или исчезла нода в `rosnode list`), затем верните копию и повторите проверку. **Проверка:** после отката список нод совпадает с исходным.

9.5 Флаги clover.launch

Ниже перечислены аргументы файла `clover.launch` из исходного кода Clover (проверено по репозиторию COEX; в вашем образе значения по умолчанию могут отличаться, поэтому проверьте свой файл командой `less`).

Флаг	По умолч.	За что отвечает и когда менять
`fcu_conn`	usb	Как бортовой компьютер соединён с полётным контроллером: по USB-кабелю или по UART (последовательному порту). Меняют, если контроллер подключён иначе
`fcu_ip`	127.0.0.1	Адрес автопилота при сетевом соединении; в симуляторе автопилот работает на том же компьютере
`fcu_sys_id`	1	Номер системы в протоколе MAVLink; нужен, если в сети несколько аппаратов

Флаг	По умолч.	За что отвечает и когда менять
`gcs_bridge`	tcp	Как QGroundControl подключается к автопилоту через бортовой компьютер по Wi-Fi: `tcp` (по умолчанию), `udp`, `udp-b` (UDP с автоматическим подключением)
`web_video_server`	true	Веб-сервер изображений: позволяет смотреть топики с видео в браузере. Выключают для экономии ресурсов
`rosbridge`	true	«Мост» между ROS и браузером; нужен веб-инструментам, в том числе визуализации TF. Без него веб-страницы Клевера не получают данные
`main_camera`	true	Драйвер основной камеры. Выключен – нет кадров, нет ArUco и оптического потока
`optical_flow`	true	Определение смещения по изображению пола (оптический поток). Требует камеры, направленной вниз, и дальномера. При камере, смотрящей вверх (маркеры на потолке), выключают
`aruco`	**false**	Включает подсистему ArUco (`aruco.launch`). **По умолчанию выключена**: без неё нет фреймов `aruco_*` и `aruco_map`. Для любых задач с маркерами ставят `true`
`rangefinder_vl53l1x`	true	Лазерный дальномер VL53L1X (высота над полом). В симуляторе не используется (там своя модель)
`led`	true	Светодиодная лента
`blocks`	**false**	Включает Clover Blocks (программирование блоками). **По умолчанию выключен**: чтобы страница `/clover_blocks/` заработала, ставят `true`
`rc`	**false**	Включает бэкенд управления дроном со смартфона (страница веб-пульта)
`force_init`	true	Заставляет оценщик положения автопилота инициализироваться, отправляя нулевое положение, когда внешних данных ещё нет
`simulator`	false	Признак работы в симуляторе; в нём вместо реальной камеры, дальномера и ленты работают их модели

Важное следствие: **Blockly (`blocks`), ArUco (`aruco`) и веб-пульт (`rc`) в поставляемом `clover.launch` выключены**, потому что они нагружают бортовой компьютер и нужны не в каждой задаче. Остальные подсистемы (камера, оптический поток, дальномер, лента, веб-видео, `rosbridge`) включены.

• Задание

Откройте свой `clover.launch` и заполните таблицу: для флагов `aruco`, `blocks`, `rc`, `optical_flow` запишите текущее значение и нужное для задачи «полёт по карте маркеров с Blockly». **Проверка:** нужны `aruco=true`, `blocks=true`, `optical_flow=true` (для маркеров на полу), `rc` безразличен.

9.6 Флаги aruco.launch

Файл `aruco.launch` подключается из `clover.launch`, когда `aruco` равен `true`. Его аргументы:

Флаг	По умолч.	За что отвечает
`aruco_detect`	true	Нода распознавания отдельных маркеров: публикует их положение и фреймы `aruco_N`
`aruco_map`	**false**	Нода карты маркеров: по нескольким маркерам вычисляет положение камеры и публикует фрейм `aruco_map`. Для полёта по координатам поля ставят `true`
`aruco_vpe`	**false**	Передача вычисленного положения автопилоту (VPE, глава 10), чтобы он использовал маркеры как источник позиции и удерживал её по ним. Ставят `true` вместе с `aruco_map`
`placement`	floor	Где наклеены маркеры: `floor` (все на полу), `ceiling` (все на потолке); для смешанных случаев строку удаляют
`length`	0.22	Размер стороны чёрной части маркера в метрах (без белой рамки) для маркеров, не описанных в карте
`map`	map.txt	Имя файла карты в каталоге `aruco_pose/map/`
`force_init`	false	Служебный флаг: инициализация оценщика нулевым положением
`disable`	false	Служебный флаг: выключить распознавание, оставив только инициализацию

Словарь маркеров в файле зафиксирован: `DICT_4X4_250` (параметр `dictionary=2` ноды `aruco_detect`). Поэтому печатать нужно маркеры именно этого словаря с номерами от 0 до 249 (глава 20). Параметры точности детектора (например, минимальный размер маркера в кадре `minMarkerPerimeterRate`) заданы в том же файле; для кадра 320×240 значение 0.075, для 640×480 – 0.0375.

9.7 Типовые наборы флагов

Задача	Что включить	Файл
Программирование блоками	`blocks=true`	clover
Распознавание отдельных маркеров, полёт над маркером `aruco_N`	`aruco=true`, `aruco_detect=true`, `placement`, `length`	clover и aruco
Полёт по координатам карты маркеров	Предыдущее и `aruco_map=true`, `map=ваш_файл.txt`	aruco
Удержание позиции автопилотом по маркерам (VPE)	Предыдущее и `aruco_vpe=true` (и настройки PX4, глава 20)	aruco
Маркеры на потолке	`placement=ceiling`; `optical_flow=false`; камера смотрит вверх	aruco и clover
Освободить ресурсы Raspberry Pi	Выключить ненужное: `web_video_server`, `rosbridge`, `led`	clover

9.8 Как это соотносится с симулятором книги

В симуляторе нет бортового компьютера с автозапуском, поэтому те же флаги передают аргументами запуска. Наш launch-файл `course.launch` (глава 18) принимает аргументы `course_blocks` и `course_aruco`; внутри он передаёт их в штатные флаги `blocks` и `aruco`. Правило одно: **чтобы подсистема заработала, её флаг должен быть `true` там, где вы запускаете систему**.

Практика. Подберите конфигурацию

Для каждой из трёх задач выпишите значения флагов (файл, флаг, значение) и порядок действий: а) показать на встрече демонстрацию программы из блоков; б) пролететь над маркером 5 на высоте 1 м; в) провести полёт по карте из 8 маркеров с удержанием позиции по маркерам. **Проверка:** в задаче б) нужны `aruco=true` и `aruco_detect=true`, `aruco_map` не требуется; в задаче в) добавлены `aruco_map=true`, `aruco_vpe=true` и свой файл карты; в задаче а) – только `blocks=true`. После каждой правки – перезапуск службы и проверка.

Источник: CopterExpress, clover.launch. Проверено 11.10.2026.

Источник: CopterExpress, aruco.launch. Проверено 11.10.2026.

Итоги и самопроверка

- Поведение Клевера задаётся launch-файлами; пакет Clover запускается автоматически как служба `clover`.
- Флаг – это `arg` с значением `default`; булевы значения `true`/`false` в нижнем регистре.
- В поставляемом `clover.launch` выключены `aruco`, `blocks` и `rc`; в `aruco.launch` выключены `aruco_map` и `aruco_vpe`.

- Правка файла: резервная копия, изменение одного флага, `sudo systemctl restart clover`, проверка.

Проверьте себя: 1) Какой флаг включает Blockly и в каком файле он находится? 2) Чем `aruco_map` отличается от `aruco_vpe`? 3) Что делать, если после правки служба не запускается?
Ответы: 1) `blocks` в `clover.launch`; 2) `aruco_map` вычисляет положение по карте маркеров, `aruco_vpe` передаёт его автопилоту как источник позиции; 3) проверить синтаксис XML, при неудаче вернуть резервную копию и перезапустить.

ГЛАВА 10

ROS и координаты без путаницы

Цели главы

- объяснить, почему тройка чисел бессмысленна без системы координат;
- отличать системы координат map, body и navigate_target и знать, что описывает каждая;
- предсказывать результат команды в разных системах координат;
- понимать назначение TF и читать типичные ошибки TF.

Нужно знать заранее: главу 7 (нода, топик, сервис) и главу 8 (телеметрия).

10.1 Зачем нужна система координат

Положение точки задаётся числами, но числа имеют смысл только относительно выбранной точки отсчёта и направлений осей. **Система координат** (в ROS – *фрейм*, от англ. frame) определяется началом отсчёта и тремя взаимно перпендикулярными осями X, Y, Z; расстояния измеряются в метрах. Фрейм имеет имя, например `map`. Одна и та же точка пространства в разных фреймах имеет разные координаты.

Для летающего аппарата кроме положения важна *ориентация* – в какую сторону он повёрнут. Поворот описывают тремя углами: **yaw** (рыскание, поворот вокруг вертикальной оси: куда «смотрит» дрон), **pitch** (тангаж, наклон вперёд-назад) и **roll** (крен, наклон набок). В ROS углы измеряют в радианах (глава 17); положительный yaw – поворот против часовой стрелки при взгляде сверху.

Прежде чем разбирать фреймы Клевера, нужно назвать две основные команды, в которых они встречаются. Это сервисы Clover (глава 7): `navigate` («лететь в точку»; аргументы – координаты x, y, z, скорость и имя фрейма, в котором заданы координаты) и `get_telemetry` («сообщить состояние дрона» в выбранном фрейме; её вы уже вызывали в главе 8). Подробно они разбираются в главе 13. Когда Python вызывает `navigate`, он не управляет оборотами каждого мотора, а просит сервис назначить цель. Ответ `success=true` означает только принятие команды, а не завершение движения: положение нужно наблюдать отдельно. По этой же причине в Blockly есть флажок ожидания (глава 11).

• Задание

Объясните, чем отличаются положение и ориентация дрона, и назовите три угла ориентации. **Проверка:** положение – где дрон находится (x, y, z), ориентация – куда он повёрнут; углы yaw, pitch, roll.

10.2 Одна тройка чисел – разные цели

Команда `x=1, y=0, z=0` неполна без указания фрейма. Два главных фрейма Клевера:

- `map` – «карта»: система координат, неподвижная относительно поля; её начало – точка, в

которой система оценки положения дрона начала работу, оси не поворачиваются вместе с дроном;
- `body` – система координат, привязанная к дрону: начало в центре аппарата, ось X направлена вперёд, Y влево, Z вверх; она повёрнута вместе с дроном по yaw, но не наклоняется вместе с креном и тангажем.

В `body` команда `x=1, y=0, z=0` означает «на метр вперёд относительно текущего положения». В `map` та же тройка – абсолютная точка поля с высотой 0: аппарат может начать снижение к полу. Поэтому фрейм всегда указывают явно, а цель сначала рисуют на бумаге.

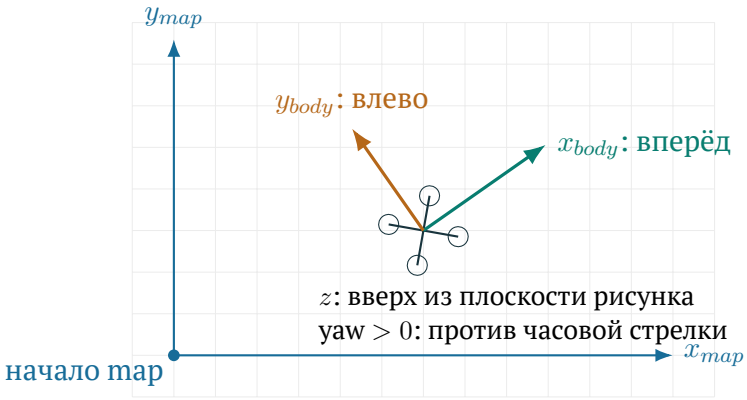


Рис. 10.1. Вид сверху. После поворота коптера оси body поворачиваются вместе с ним, оси map остаются на месте.

Кроме map и body в Клевере используются другие фреймы:

Фрейм	С чем связан	Применение
`map`	Локальное начало оценки позиции	Квадрат, сетка, траектория
`body`	Текущее положение и yaw дрона, без крена и тангажа	Относительный взлёт, шаг вперёд
`navigate_target`	Последняя назначенная цель команды `navigate`	Остаток до цели, цепочка шагов
`aruco_map`	Карта визуальных маркеров ArUco (глава 20)	Полёт по координатам поля
`aruco_N`	Одиночный маркер с номером N	Точка над известным маркером
`base_link`	Корпус дрона с полной ориентацией, включая крен и тангаж	Положение датчиков и камеры

Источник: COEX, системы координат и TF. Проверено 11.10.2026.

• Задание

Дрон в map в точке $(1, 2, 1)$ направлен вдоль $+X$. Куда он придёт после команды $(2, 0, 0)$ в body? А после той же команды в map? **Проверка:** в body: $(3, 2, 1)$; в map: $(2, 0, 0)$ – возле пола, это ошибка.

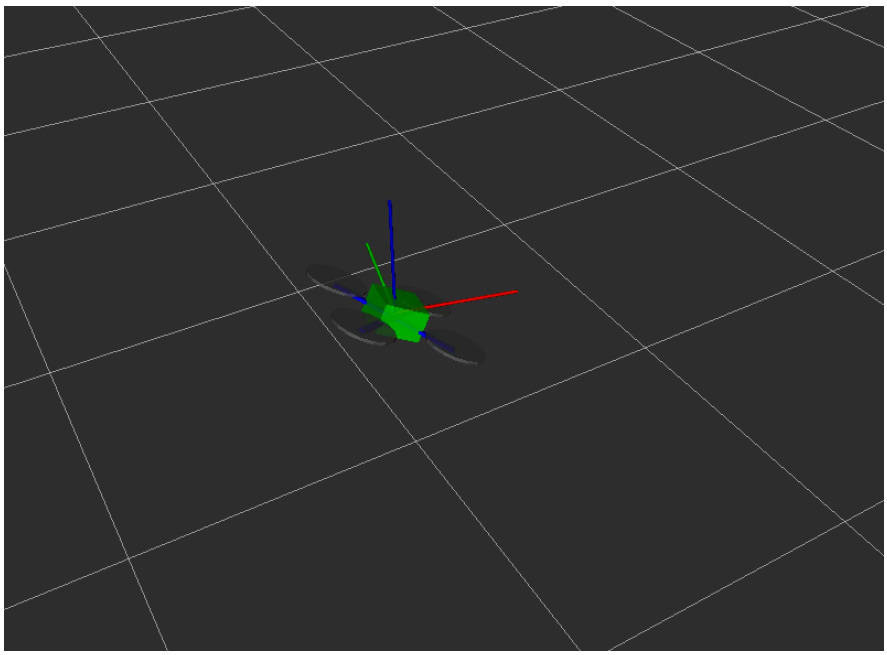


Рис. 10.2. Системы координат Клевера из документации COEX, CC BY-NC-SA 4.0: у аппарата и у поля есть свои оси, и положения связываются преобразованиями TF.

10.3 Соглашения об осях: FLU, ENU и камера

У осей есть общепринятые соглашения. В ROS системы координат корпуса задаются так: X вперёд (Forward), Y влево (Left), Z вверх (Up) – сокращённо **FLU**. Система, привязанная к сторонам света, называется **ENU**: X на восток (East), Y на север (North), Z вверх (Up). Но `map` без привязки к сторонам света не следует считать точным компасом: её ось X направлена туда, куда дрон был ориентирован при включении. Автопилот PX4 внутри использует другое соглашение (NED: север, восток, вниз); MAVROS выполняет пересчёт. Поэтому, отправляя команды через API Clover (набор его команд), не нужно менять знак осей «потому что PX4 использует NED».

У камеры своё соглашение (оптический фрейм): X вправо по изображению, Y вниз, Z вдоль оптической оси (в ту сторону, куда смотрит объектив). Поэтому ошибка в пикселях изображения не равна метрам в `body`. Чтобы управлять по камере, нужны *калибровка* (определение параметров объектива), известная геометрия установки камеры и пересчёт координат. В главе 21 сначала строится детектор, затем обсуждается управление.

• Задание

Нарисуйте оси body и оси оптического фрейма камеры рядом. **Проверка:** у body Y влево и Z вверх; у камеры Y вниз и Z вдоль оптической оси.

10.4 TF и время

Ответ на вопрос «где точка из одного фрейма в другом?» даёт **TF** – служба ROS, которая хранит взаимное положение фреймов и его изменение во времени (как топик, TF публикуется всё время). Благодаря ей, например, положение маркера, увиденного камерой, можно пересчитать в `map`. Типичные ошибки: `Unknown frame` – запрошенного фрейма нет (никто его не публикует); `Extrapolation` – нужное преобразование известно только для другого момента времени (часто из-за рассогласования часов). Поэтому при ошибке TF проверяют, идёт ли

`/clock` (часы симулятора), публикуется ли `/tf` (топик с преобразованиями) и не пропала ли карта маркеров из поля зрения камеры.

```
rosrun tf tf_echo map base_link
rosrun tf tf_echo map aruco_map
rostopic echo -n 1 /tf
```

`tf_echo` *A B* непрерывно печатает положение фрейма *B* в фрейме *A*. Первая команда показывает, где дрон относительно `map`; вторая – где карта маркеров, но только пока она видна камере. Название доступного фрейма уточняют в установленной конфигурации: при использовании маркеров в качестве внешнего источника позиции (по-английски VPE, Vision Position Estimate – «оценка положения по зрению»; глава 20) документация описывает также фрейм `aruco_map_detected`. Отсутствие вывода второй команды до включения ArUco нормально.

Для наглядного просмотра фреймов используется **RViz** – программа трёхмерной визуализации данных ROS: в ней фреймы отображаются осями и стрелками.

Практика. Предскажите положение

Дрон находится в map $(2, 1, 1)$ и направлен вдоль положительной оси X карты. После команды body $(1, 0, 0)$ ожидается $(3, 1, 1)$. После поворота yaw на $+90°$ тот же шаг даёт примерно $(2, 2, 1)$, если начать из исходной точки. Объясните, почему команда map $(1, 0, 0)$ в обоих случаях означает одну и ту же точку. Сверьте направление в RViz.

Итоги и самопроверка

- Фрейм отвечает на вопрос «от чего и куда мерим»; одна точка в разных фреймах имеет разные координаты.
- body – относительно дрона и его направления, map – абсолютно в поле.
- TF хранит положения систем координат во времени; ошибка Unknown frame означает отсутствие фрейма.

Проверьте себя: 1) Что значит x=1 в body? 2) Куда полетит дрон, если дать map (1, 0, 0) с высоты 1 м? 3) Какая ошибка TF означает отсутствие фрейма?
Ответы: 1) один метр вперёд относительно текущего положения; 2) к точке у пола в начале карты – дрон может начать снижаться; 3) Unknown frame.

ГЛАВА 11

Первые миссии в Blockly

Цели главы

- собрать и запустить первую миссию «взлёт – шаг – посадка»;
- объяснить, что делает флажок wait;
- отличать Stop от Land.

Нужно знать заранее: главы 8 (приёмка стенда).

11.1 Открываем интерфейс

Blockly – среда визуального программирования: программа собирается из цветных блоков, как из конструктора, а система автоматически строит по ним код на Python, который можно посмотреть. Среда Clover Blocks работает как веб-страница: её открывают в браузере, а выполняет программу сам стенд. Для этого нужен *веб-сервер* – программа, которая отдаёт браузеру страницы; в готовой VM он уже настроен. **В браузере внутри Ubuntu VM** откройте `http://localhost/clover_blocks/`. Адрес `localhost` означает «этот же компьютер»: если открыть его в браузере Windows, а не в браузере внутри VM, запрос пойдёт на Windows, а не на Ubuntu. Если страница не открывается, проверьте, что веб-сервер работает и установлены веб-инструменты Клевера.

Адрес `http://192.168.11.1/clover_blocks/` в документации относится к настоящему Клеверу, подключённому по Wi-Fi. Для симулятора используйте адрес компьютера, на котором работает веб-сервер. Если браузер запущен в Windows (на «хосте»), понадобится адрес виртуальной машины («гостя») или проброс порта (настройка гипервизора, перенаправляющая обращения к порту хоста на гостя).

Для Blockly одновременно нужны: веб-страница; *backend* – нода ROS `clover_blocks`, которая принимает программу от страницы и выполняет её; и связь с ROS. Backend включается флагом (параметром запуска) `blocks` в launch-файле `clover.launch`; в поставляемом файле этот флаг по умолчанию выключен (`false`), все флаги описаны в главе 9. На реальном дроне его включают правкой файла и перезапуском службы. Поэтому книга использует собственный launch-файл `course.launch`: он запускает штатный launch Клевера с нужными флагами; создаёт его скрипт `make_course_launch.py`, устройство которого описано в главе 18. На нативной Ubuntu веб-сервер настраивают отдельно по документации COEX.

11.1.1 Включаем backend перед первым полётом

Пакет `clover_course` уже создан в главе 7. Если она пропущена, сначала создайте пакет по инструкции главы 18. Завершите текущую миссию, убедитесь, что дрон disarmed (моторы выключены, `armed=false`), и остановите ранее запущенный симулятор (Ctrl+C в окне A). Затем в окне A (терминал Ubuntu с выполненными командами `source`):

```
cd "$HOME/clover_course_files/examples"
python3 make_course_launch.py
roslaunch clover_course course.launch \
  course_blocks:=true course_aruco:=false
```

Первая команда запускает скрипт (создаёт файл `course.launch` в пакете `clover_course`); вторая запускает симулятор с включённым backend блоков. Аргумент `course_aruco:=false` отключает навигацию по маркерам ArUco, которая подключается в отдельной главе. Повторите приёмку (глава 8) и откройте страницу. Если скрипт не узнаёт структуру установленного launch, используйте разбор из главы 18.

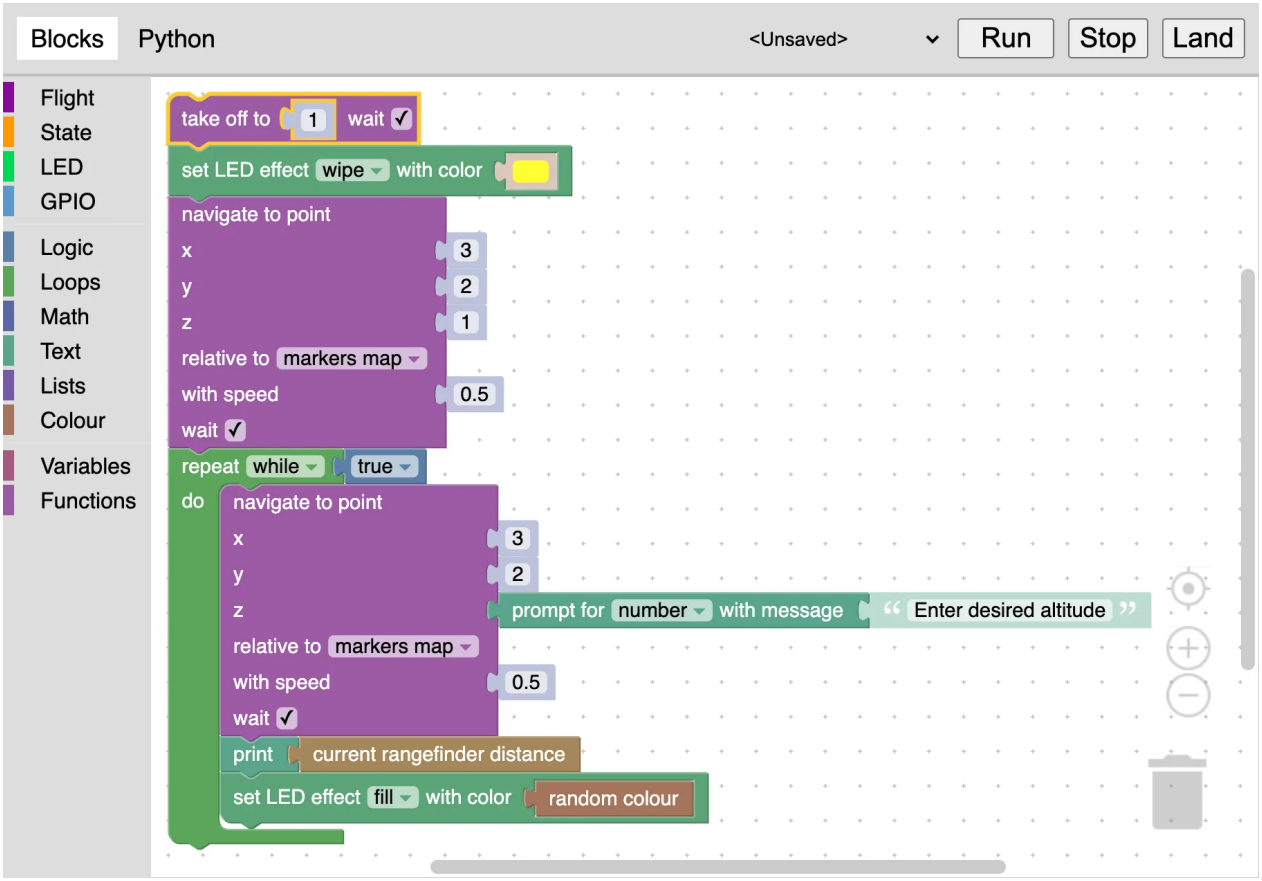


Рис. 11.1. Интерфейс Clover Blocks из документации COEX, CC BY-NC-SA 4.0. Вкладка Python показывает код собранной программы.

Источник: COEX, настройка и управление Blockly. Проверено 11.10.2026.

• Задание

Откройте страницу блоков и найдите вкладки Blocks и Python, кнопки Run, Stop и Land. **Проверка:** вы можете сказать, какая кнопка сажает дрон (Land).

11.2 Миссия A: взлёт – шаг – посадка

Соберите следующую программу. Ниже приведена **учебная схема блоков**, а не снимок вашей программы и не файл для импорта. Названия полей ориентированы на англоязычный интерфейс COEX.

Вот как эти блоки выглядят в настоящем интерфейсе Clover Blocks. Сравните их с нашей схемой: они устроены так же, но у каждого есть поля для чисел и флажок.

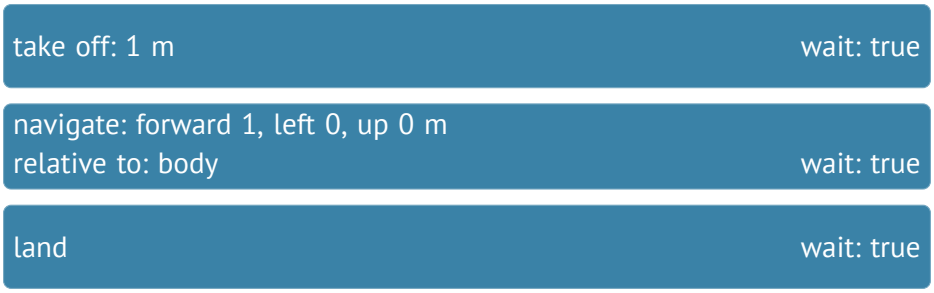
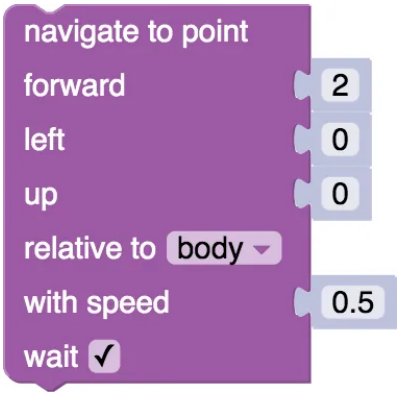


Рис. 11.2. Схема первой программы Blockly. Движение по body задаётся относительно положения на момент команды.



take off to – взлететь на заданную высоту в метрах. Флажок **wait** означает «не переходить к следующему блоку, пока дрон не достигнет высоты».



navigate to point – лететь в точку. Поля **forward, left, up** – на сколько метров вперёд, влево, вверх. Поле **relative to** выбирает систему координат (мы разбирали её в главе 10): body – от дрона, map – от начала поля. **with speed** – скорость в м/с.



land – посадка; с флажком wait ждёт, пока дрон приземлится.



wait arrival – отдельный блок «дождаться прибытия», если флажок wait у перемещения был выключен.

Рис. 11.3. Основные блоки полёта. Изображения из документации COEX, CC BY-NC-SA 4.0.

1. Проверьте связь и позиционирование по главе 8. Уберите другие запущенные миссии.
2. В разделе Flight выберите блок взлёта, задайте 1 м, включите wait.
3. Добавьте перемещение на 1 м вперёд в body и снова включите wait.
4. Добавьте land с ожиданием. Нажмите Run и наблюдайте последовательность.
5. После посадки сохраните программу через Save с именем `first_flight`.

Флажок wait предотвращает мгновенный переход к следующей команде. Если выключить его у перемещения и сразу вызвать land, дрон может начать посадку раньше, чем достигнет точки. Блок wait на фиксированное число секунд удобен для паузы, но не подтверждает прибытие.

Stop и Land – разные действия

Документация COEX описывает Stop как остановку программы, а Land – как остановку программы с посадкой. Нельзя считать, что закрытие вкладки или прекращение Python автоматически сажает аппарат: `simple_offboard` (программа Clover, которая отправляет автопилоту команды) продолжает удерживать последнюю заданную цель (по-английски setpoint – «заданное значение»), пока не получит новую. Для штатного завершения используйте явную посадку.

• Задание

До запуска запишите прогноз: что дрон сделает и в какой момент. После запуска сравните. **Проверка:** прогноз совпал, либо расхождение объяснено (например, флажком wait).

11.3 Миссия B: квадрат с циклом

Для квадрата без поворота (без изменения yaw) используйте перемещения относительно **last navigate target** – последней заданной цели: система координат, начало которой находится в предыдущей цели, а не в текущем положении дрона (глава 10, фрейм `navigate_target`). После взлёта повторите четыре шага, включив ожидание у каждого:

```
take off 1 m, wait
navigate +1 forward,  0 left, 0 up; last navigate target, wait
navigate  0 forward, +1 left, 0 up; last navigate target, wait
navigate -1 forward,  0 left, 0 up; last navigate target, wait
navigate  0 forward, -1 left, 0 up; last navigate target, wait
land, wait
```

Эта форма избегает накопления ошибки «от текущего неточного положения»: следующая цель отсчитывается от прошлой *заданной* цели. Более короткий вариант – цикл «повторить 4 раза», шаг вперёд и поворот на $+90°$ с ожиданием. Он полезен как отдельный опыт, но требует правильно выбрать систему координат поворота и дождаться ориентации.

Откройте вкладку Python. Найдите вызов `navigate` и код ожидания. Сравните, где указан `auto_arm=True` (разрешение автоматически включить моторы и перейти в нужный режим перед взлётом), какой фрейм передан и как переводятся углы (в Python углы в радианах, в блоках – в градусах). Не редактируйте этот код с ожиданием, что он автоматически перестроит блоки: вкладка служит представлением генерируемой программы, а возможность редактирования зависит от версии.

• Задание

Соберите квадрат и запустите его один раз. **Проверка:** дрон вернулся близко к старту и после посадки disarmed.

11.4 Состояние, условия и LED

Блоки State дают координаты и другие параметры. Условие позволяет решить: продолжать маршрут или завершить миссию. LED можно использовать как визуальное состояние: один цвет перед стартом, другой при достижении точки. LED – светодиодная лента на корпусе дрона; она симулируется в поддерживаемой конфигурации. Выводы GPIO (управляемые контакты платы Raspberry Pi, к которым подключают, например, захват) и электрическое действие захвата в Gazebo автоматически не возникают.

В учебном цикле ожидания обязательно подумайте, что произойдёт при пропаже данных. Если Blockly вашей версии не предоставляет таймаут, первый эксперимент делайте коротким и завершайте через Land при зависании. Реализацию ограниченного ожидания изучим на Python.

Практика. Блоки и их Python

Сохраните две программы: `first_flight` и `square`. Для каждой приложите снимок блоков и сгенерированный Python. В отчёте объясните wait, body и last navigate target. Успех: последовательность видна в Gazebo, а после land телеметрия показывает `armed=false`.

Итоги и самопроверка

- Блок – наглядная запись команды; под ним генерируется Python.
- wait подтверждает прибытие, пауза в секундах – нет.
- Stop останавливает программу, но не сажает дрон: для посадки нужен Land.

Проверьте себя: 1) Что случится, если выключить wait у перемещения и сразу вызвать land? 2) Чем last navigate target удобнее body в квадрате?
Ответы: 1) посадка начнётся до прибытия в точку; 2) следующая цель отсчитывается от прошлой заданной цели, ошибка не накапливается.

ГЛАВА 12

Blockly: мастерская алгоритмов

Цели главы

- использовать переменные, циклы и условия в блоках;
- сравнить блоки и сгенерированный Python;
- собрать набор из пяти миссий.

Нужно знать заранее: главы 11.

12.1 От цепочки к алгоритму

Первая миссия показала, как расположить команды по порядку. Теперь цель – научиться изменять программу осмысленно. Каждое задание начинается с прогноза: куда полетит аппарат, когда начнётся следующая команда и что подтвердит завершение. Затем вы запускаете короткий опыт и сравниваете результат с прогнозом.

Все схемы в этой главе – описание сборки в интерфейсе, а не файлы XML для импорта. Категории Flight, State, LED и стандартные блоки переменных, логики, математики и циклов могут иметь разные переводы. Ориентируйтесь на смысл и поля установленной версии. Сохраняйте каждый вариант отдельно.



Рис. 12.1. Меню сохранения и кнопки Run, Stop, Land в интерфейсе COEX, CC BY-NC-SA 4.0. Сохраняйте вариант до изменения алгоритма.

• Задание

Выберите одну из миссий ниже и запишите прогноз: траектория, момент следующей команды, подтверждение завершения. **Проверка:** прогноз содержит все три пункта.

12.2 Миссия C. Высота как переменная

Создайте переменную `height`, присвойте ей 1.0 и вставьте её в поле take off. Добавьте wait 1 s, затем land с ожиданием. Порядок:

```
set height = 1.0
take off height m; wait = true
wait 1 s
land; wait = true
```

Запустите два опыта: height 0.8 и 1.2 м. Перед каждым дрон должен стоять на полу и быть disarmed. В Python переменной соответствует присваивание, а поле блока передаёт значение как аргумент функции.

Нельзя заключить, что время ожидания одна секунда гарантирует полную неподвижность. Сначала take off с wait подтверждает прибытие по логике блока, затем пауза добавляет время наблюдения. Для точной съёмки позже потребуется критерий скорости.

Проверка переменной

Измените только присваивание height и покажите, что число в блоке взлёта не приходится исправлять отдельно. В отчёте подпишите единицы. Дополнительное задание: создайте переменную side для всех сторон квадрата.

12.3 Миссия D. Три ступени высоты

После взлёта на 0.8 м добавьте два относительных подъёма по 0.2 м в `last navigate target`, с ожиданием между ними. После каждого подъёма сделайте короткую паузу для наблюдения.

```
take off 0.8 m; wait
navigate 0 forward, 0 left, +0.2 up; last navigate target; wait
wait 1 s
navigate 0 forward, 0 left, +0.2 up; last navigate target; wait
wait 1 s
land; wait
```

Прогноз высот над стартовым уровнем: 0.8, 1.0, 1.2 м. Относительное перемещение 0.2 – не команда «занять абсолютную высоту 0.2». Сравните программу с файлом `heights.py` (Python-версия из главы 16): он задаёт абсолютные уровни в map относительно измеренного старта.

• Задание

Предскажите высоты после каждого шага и сравните с наблюдением. **Проверка:** 0.8, 1.0 и 1.2 м.

12.4 Миссия E. «Туда и обратно» с циклом

После взлёта повторите два раза пару перемещений: 0.6 м вперёд и 0.6 м назад, оба в last navigate target. Не меняйте yaw. Включите wait у каждого перехода, затем завершите посадкой.

```
take off 1 m; wait
repeat 2 times:
    navigate +0.6 forward, 0 left, 0 up; last navigate target; wait
    navigate -0.6 forward, 0 left, 0 up; last navigate target; wait
land; wait
```

Четыре команды движения теперь описаны двумя блоками внутри цикла. Это хороший первый пример сокращения повторений. Увеличение количества повторов удлиняет миссию, но не меняет геометрию одного цикла.

Следите за тем, что повторяется

Если поместить land внутрь цикла, посадка начнётся после первого прохода. Если поместить take off внутрь цикла, смысл следующего взлёта зависит от текущего состояния. Сначала выделите на бумаге повторяемую часть, затем охватите её блоком repeat.

• Задание

Увеличьте число повторов до 3. Сколько команд движения будет выполнено? **Проверка:** шесть.

12.5 Миссия F. Треугольник без поворотов

Для равностороннего треугольника со стороной 0.8 м используйте смещения в last navigate target при неизменном yaw: $(0.8, 0)$, $(-0.4, 0.693)$, $(-0.4, -0.693)$. Сумма X равна нулю, сумма Y равна нулю, поэтому маршрут замыкается в заданных координатах.

```
take off 1 m; wait
navigate +0.8 forward,     0 left, 0 up; last navigate target; wait
navigate -0.4 forward, +0.693 left, 0 up; last navigate target; wait
navigate -0.4 forward, -0.693 left, 0 up; last navigate target; wait
land; wait
```

Число 0.693 – приближение $0.8\sqrt{3}/2$. Реальный путь будет сглажен динамикой аппарата. Замыкание списка заданных точек и точное возвращение модели – разные результаты; второй нужно измерять.

• Задание

Проверьте сумму смещений по X и по Y треугольника. **Проверка:** $0.8 - 0.4 - 0.4 = 0$ и $0 + 0.693 - 0.693 = 0$ – маршрут замыкается.

12.6 Миссия G. «Змейка» для осмотра

Змейка покрывает несколько параллельных строк. После взлёта: вперёд 1.2 м, влево 0.4 м, назад 1.2 м, влево 0.4 м, вперёд 1.2 м. Все перемещения – last navigate target с wait, yaw неизменный. Для возврата добавьте отдельные переходы по свободному коридору: последний пункт змейки не является домом.

На бумаге нарисуйте сетку с шагом 0.4 м и пронумеруйте цели. В Python список для такой змейки создаёт `snake.py`. Сначала убедитесь, что пространство свободно по всему пути, включая возврат. Одна конечная точка не описывает обход препятствий.

• Задание

Нарисуйте змейку на сетке 0.4 м и найдите длину пути. **Проверка:** $1.2 + 0.4 + 1.2 + 0.4 + 1.2 = 4.4$ м.

12.7 Миссия H. Решение по измеренной высоте

После взлёта прочитайте высоту через блок State/get position в выбранном фрейме. Запишите её в переменную `measured`. Если число ниже выбранной границы, выведите один сигнал состояния, иначе другой. Для первого опыта достаточно сообщения, если оно доступно в стандартных блоках, или двух цветов LED при включённой модели.

Сравнение корректно только при известном начале координат. Высота в map не всегда равна расстоянию над полом. После проверки состояния в любом случае выполните land. Ветвление здесь выбирает сигнал, а не отменяет завершение миссии.



Рис. 12.2. Блок состояния **current x relative to …** (коричневый) отдаёт число – координату; блок **set LED effect** (зелёный) включает эффект светодиодной ленты нужного цвета. COEX, CC BY-NC-SA 4.0.

```
take off 1 m; wait
set measured = get_position(z, selected frame)
if measured < chosen_threshold:
    indicate LOW
else:
    indicate HIGH
land; wait
```

Это учебный алгоритм: точные подписи блока состояния и доступность проверки некорректного числа уточните во вкладке Python. Если блоки вашей версии не позволяют надёжно обработать отсутствие позиции, такую проверку перенесите в Python, где есть `math.isfinite`.

• Задание

Запишите, какую проверку выгоднее выполнить в Python, а не в блоках. **Проверка:** проверку на NaN (`math.isfinite`).

12.8 Миссия I. Одна карта – несколько станций

После установки согласованной карты ArUco (глава 20) задайте точки в системе координат `markers map` (в Python – фрейм `aruco_map`: система координат карты маркеров): $(0, 0, 1.2)$, $(1, 0, 1.2)$, $(1, 1, 1.2)$, $(0, 1, 1.2)$ и обратно $(0, 0, 1.2)$. Включите wait на каждом переходе. Сначала выполните проверку карты из главы 20; до неё этот опыт не запускайте.

Это координаты карты, а не четыре относительных смещения. При одинаковых абсолютных координатах маршрут привязан к разметке, а не к неточному месту старта. Если переставить

модель поля, карта и TF должны остаться согласованными. Иначе красивый рисунок маршрута не соответствует реальным целям.

• Задание

Запишите пять точек миссии I списком. **Проверка:** $(0, 0, 1.2)$, $(1, 0, 1.2)$, $(1, 1, 1.2)$, $(0, 1, 1.2)$, $(0, 0, 1.2)$.

12.9 Пауза и ожидание прибытия: контрольный опыт

На свободном стенде выполните одну программу с wait true на перемещении. Во второй программе выключите этот wait, добавьте обычный wait 0.2 s и затем land. Скорость небольшая, путь 0.6 м. Наблюдайте, почему короткая фиксированная пауза не заменяет проверку цели.

Цель опыта – увидеть различие, а не сделать длинный полёт с конфликтующими командами. Не меняйте одновременно скорость, длину и ожидание: тогда будет непонятно, что вызвало результат. После каждого опыта проверьте disarm.

• Задание

Запишите, что вы ожидаете увидеть в каждом из двух опытов, до запуска. **Проверка:** ожидание: в первом дрон прилетает и затем садится; во втором посадка может начаться раньше прибытия.

12.10 Как читать сгенерированный Python

Откройте вкладку Python у самой короткой программы. Найдите четыре места: инициализация ноды, создание ServiceProxy, вызов navigate, ожидание или land. Затем найдите числовые значения из ваших блоков. Отметьте, где блок высоты превращается в аргумент z, где выбран frame_id и где стоят True/False.

Сгенерированный код может включать функции-помощники, которых не видно в блоковой цепочке. Это нормально: один визуальный блок может соответствовать нескольким строкам. Но не считайте, что любой код из интернета автоматически превратится обратно в Blockly. Для переноса алгоритма важен смысл конструкции.

Набор миссий Blockly

Сохраните не менее пяти миссий этой главы. Для каждой приложите схему пути, снимок блоков, прогноз и наблюдение. Выберите одну и объясните её Python: где переменная, цикл, условие и ожидание. Дополнительное задание: найдите повторяющуюся часть и оформите её процедурой, если такой блок есть в вашей версии.

Итоги и самопроверка

- Переменная избавляет от правки числа в нескольких местах.
- Цикл повторяет только нужную часть миссии.
- Ветвление выбирает сигнал, но посадка выполняется в любом случае.

Проверьте себя: 1) Что произойдёт, если land поставить внутрь цикла? 2) Почему относительный подъём на 0.2 – не абсолютная высота?
Ответы: 1) посадка начнётся после первого прохода; 2) он прибавляется к текущей высоте.

ГЛАВА 13

Python: надёжная первая программа

Цели главы

- назвать основные сервисы управления Клевером и их аргументы;
- прочитать полётную программу целиком и объяснить каждую часть;
- отличить «команду приняли» от «долетели»;
- запустить телеметрию и первую миссию.

Нужно знать заранее: главы 6 (основы Python), 7 (ноды и сервисы), 8 (приёмка стенда, телеметрия) и 10 (фреймы).

13.1 Управление Клевером: четыре сервиса

Пакет Clover (глава 2) предоставляет программе управления набор *сервисов* ROS (запрос–ответ, глава 7). В книге используются четыре.

Сервис	Что делает	Основные аргументы / результат
`get_telemetry`	Возвращает состояние дрона	Аргумент `frame_id` (в каком фрейме дать координаты). Результат: `connected`, `armed`, `mode`, `x`, `y`, `z` и др.
`navigate`	«Лететь в точку»: по прямой с заданной скоростью	`x`, `y`, `z` – цель в метрах; `frame_id` – фрейм цели; `speed` – скорость, м/с; `yaw` – желаемый поворот в радианах (`NaN` – не менять); `auto_arm` – включить моторы автоматически. Результат: `success` и `message`
`set_position`	Задаёт цель так же, как `navigate`, но без плавного разгона; используется для потока целей (окружность, глава 15)	Те же координаты и фрейм
`land`	Посадка	Без аргументов. Результат: `success` и `message`

Несколько слов, без которых сервисы непонятны.

- **armed** (от англ. arm, «взвести, привести в боевую готовность») – моторы включены и аппарат готов к полёту; **disarm** – выключение моторов. Перед взлётом моторы должны быть включены: это делает флаг `auto_arm=True`.
- **Режим полёта** (`mode`) – состояние автопилота, определяющее, кто им управляет. Режим **OFFBOARD** означает «внешнее управление»: автопилот выполняет цели, которые присылает бортовой компьютер (то есть команды Clover). Если режим сменился (например,

пилот переключил его с пульта), автономная миссия прекращается.
- **Цель** – точка (и, возможно, поворот), к которой автопилот ведёт аппарат; программа ставит цель, а регуляторы автопилота (глава 2) решают, как её достичь.
- Ответ `success=true` сервиса `navigate` означает только то, что цель *принята*; дрон ещё не прилетел (глава 10). Прибытие проверяют отдельно.

Источник: COEX, сервисы simple_offboard. Проверено 11.10.2026.

• Задание

Какие аргументы `navigate` нужно указать, чтобы лететь на 1 м вперёд от дрона со скоростью 0.3 м/с? **Проверка:** `x=1, y=0, z=0, frame_id='body', speed=0.3`.

13.2 Из чего состоит ROS-программа

Любая программа управления выполняет три действия: инициализирует ноду, дожидается нужных сервисов и создаёт для них *прокси* – Python-объекты, вызов которых отправляет запрос сервису (`ServiceProxy`, глава 7). Типы сервисов Clover (например, `srv.Navigate`) создаются при сборке пакета Clover системой catkin (глава 7); установка похожей библиотеки через `pip` их не заменяет.

Каталог `examples` из комплекта книги копируют в Ubuntu в `~/clover_course_files/examples` (внутри VM это удобно делать через общую папку гипервизора или обычным копированием); во всех командах ниже предполагается именно этот путь. Симулятор должен быть запущен в окне A (глава 8), а следующие команды вводятся в окне B:

```
source /opt/ros/noetic/setup.bash
source "$HOME/catkin_ws/devel/setup.bash"
cd "$HOME/clover_course_files/examples"
python3 inspect_state.py
```

Не называйте собственные файлы `rospy.py`, `clover.py` или `math.py`: Python может подключить их вместо настоящих библиотек. Запускайте системный Python Ubuntu с ROS, а не отдельный Python Windows или изолированное окружение (virtualenv), в котором ROS не виден.

13.2.1 Телеметрия перед полётом

```
#!/usr/bin/env python3
import math
import rospy
from clover import srv

rospy.init_node('course_inspect')
rospy.wait_for_service('get_telemetry', timeout=15)
telemetry = rospy.ServiceProxy('get_telemetry', srv.GetTelemetry)
state = telemetry(frame_id='map')
print('Connected:', state.connected, 'Armed:', state.armed)
print('Mode:', state.mode, 'XYZ:', state.x, state.y, state.z)
print('Finite XYZ:', all(math.isfinite(v) for v in
                         (state.x, state.y, state.z)))
print('Simulation clock:', rospy.get_param('/use_sim_time', False))
```

Программа вызывает `get_telemetry` и печатает ответ. Получение телеметрии не включает моторы. Убедитесь, что вывод содержит признак связи, конечные координаты (проверка

`math.isfinite`) и значение параметра симуляционного времени (глава 7). Только после этого переходят к полёту.

Источник: COEX, типы сервисов и телеметрия. Проверено 11.10.2026.

• Задание

Запустите `inspect_state.py` и найдите в выводе связь и координаты. **Проверка:** connected равно True, XYZ – числа.

13.3 Первая миссия

Миссия – программа, которая командует полётом. Первая миссия использует вспомогательный модуль `flight_tools.py`, написанный для книги (его текст приведён ниже и разбирается построчно в главе 14). Модуль определяет класс `Flight` с методом `go` («лететь и дождаться прибытия») и контекстом `mission()`, который гарантирует посадку в конце.

```
#!/usr/bin/env python3
import rospy
from flight_tools import Flight

rospy.init_node('course_first_flight')
flight = Flight()
with flight.mission():
    flight.go(z=1, frame='body', auto_arm=True)
    flight.go(x=1, z=0, frame='body')
```

Запуск – `python3 first_flight.py`. Конструкция `with flight.mission():` выполняет вложенные команды и затем обязательно вызывает посадку, даже если произошла ошибка (механизм `try/finally` из главы 6).

Первая команда `go(z=1, frame='body', auto_arm=True)` поднимает дрон на 1 м относительно текущего уровня: в системе `body` (глава 10) цель смещена на метр вверх; моторы включаются автоматически. Вторая команда `go(x=1, z=0, frame='body')` смещает цель на метр вперёд, оставляя высоту прежней. `auto_arm=True` достаточно указать только в первой команде. В системе `map` высоту пришлось бы передавать как абсолютную координату.

• Задание

Измените высоту взлёта на 0.8 м и запустите программу. **Проверка:** высота взлёта ниже прежней, после посадки armed равно false.

13.4 Почему ожидание важнее паузы

Фиксированная пауза («подождать пять секунд») не доказывает, что дрон прилетел: он может лететь дольше или совсем не лететь. Правильно *измерять расстояние до цели*. В Клевере для этого служит фрейм `navigate_target` – система координат, связанная с последней поставленной целью: координаты дрона в ней и есть остаток пути.

$$d = \sqrt{x^2 + y^2 + z^2},$$

где x, y, z – координаты коптера в системе целевой точки. Когда расстояние d меньше допуска (допустимого отклонения) в нескольких измерениях подряд, переход считается за-

вершённым. Это проверка положения, а не доказательство нулевой скорости: для точной съёмки можно дополнительно ограничить скорость и добавить время стабилизации.

Модуль использует `time.monotonic()` (монотонные часы, не зависящие от симулятора) для таймаута. Эти часы идут даже при паузе Gazebo, тогда как `rospy.get_time()` следует часам симуляции. Допуск 0.15 м и три измерения подряд – параметры учебного опыта; на медленном стенде таймаут увеличивают, так как он измеряется в реальных секундах.

13.4.1 Модуль помощников: полный листинг

```
#!/usr/bin/env python3
"""ROS 1 helpers for the textbook; simulation only, SI units."""
import math
import time
from contextlib import contextmanager

import rospy
from clover import srv
from std_srvs.srv import Trigger


class Flight:
    def __init__(self, simulation_only=True):
        # This guard is a classroom convenience, not a security boundary.
        # A real aircraft needs simulation_only=False and a pilot with a
        # radio transmitter ready to take over (see the real-drone chapter).
        if simulation_only and not rospy.get_param('/use_sim_time', False):
            raise RuntimeError('Expected Gazebo /use_sim_time=true')
        for name in ('get_telemetry', 'navigate', 'land', 'set_position'):
            rospy.wait_for_service(name, timeout=15)
        self.telemetry = rospy.ServiceProxy('get_telemetry', srv.GetTelemetry)
        self.navigate = rospy.ServiceProxy('navigate', srv.Navigate)
        self.land = rospy.ServiceProxy('land', Trigger)
        self.set_position = rospy.ServiceProxy('set_position', srv.SetPosition)
        self.check('map')

    def check(self, frame='map'):
        state = self.telemetry(frame_id=frame)
        if not state.connected:
            raise RuntimeError('FCU disconnected')
        if not all(math.isfinite(v) for v in (state.x, state.y, state.z)):
            raise RuntimeError('Position or TF unavailable: ' + frame)
        return state

    @staticmethod
    def accepted(response):
        if not response.success:
            raise RuntimeError(response.message)

    def go(self, x=0, y=0, z=0, frame='map', speed=0.4,
           auto_arm=False, timeout=30, tolerance=0.15):
        if not all(math.isfinite(v) for v in (x, y, z, speed, timeout, tolerance)):
            raise ValueError('Coordinates and motion settings must be finite')
        if speed <= 0 or timeout <= 0 or tolerance <= 0:
            raise ValueError('speed, timeout and tolerance must be positive')
        self.accepted(self.navigate(
            x=x, y=y, z=z, frame_id=frame, speed=speed,
            yaw=float('nan'), auto_arm=auto_arm))
        deadline = time.monotonic() + timeout
        start = time.monotonic()
        stable = 0
        while not rospy.is_shutdown():
            state = self.check('navigate_target')
            if time.monotonic() - start > 2:
```

```
                if not state.armed or state.mode != 'OFFBOARD':
                    raise RuntimeError('Flight left OFFBOARD or disarmed')
            distance = math.sqrt(state.x**2 + state.y**2 + state.z**2)
            stable = stable + 1 if distance < tolerance else 0
            if stable >= 3 and state.armed and state.mode == 'OFFBOARD':
                return
            if time.monotonic() >= deadline:
                raise TimeoutError('Arrival timeout')
            # Wall time keeps deadlines effective when Gazebo is paused.
            time.sleep(0.1)
        raise RuntimeError('ROS shutdown before arrival')

    def land_wait(self, timeout=40):
        self.accepted(self.land())
        deadline = time.monotonic() + timeout
        while not rospy.is_shutdown():
            state = self.telemetry(frame_id='map')
            if not state.connected:
                raise RuntimeError('FCU disconnected during landing')
            if not state.armed:
                return
            if time.monotonic() >= deadline:
                raise TimeoutError('Landing/disarm timeout; inspect Gazebo')
            time.sleep(0.1)
        raise RuntimeError('ROS shutdown during landing')

    @contextmanager
    def mission(self):
        try:
            yield self
        finally:
            # Land both after completion and after a Python exception.
            # ROS shutdown or failed transport can still prevent this call.
            self.land_wait()
```

• Задание

Остаток до цели: 0.1 м по X, 0.1 м по Y, 0.05 м по Z. Вычислите расстояние и сравните с допуском 0.15 м. **Проверка:** $\sqrt{0.0225} = 0.15$ м – ровно на границе допуска.

13.5 Что этот модуль контролирует

- Перед началом проверяет, что включено симуляционное время, что нужные сервисы доступны и что координаты конечны. Флаг времени не доказывает, что аппарат виртуальный.
- Прерывает миссию, если сервис вернул `success=false`.
- Во время полёта контролирует связь, координаты, признак `armed` и режим OFFBOARD; на переход в режим отводит две секунды.
- Контекст `mission()` вызывает посадку и ждёт disarm. Автоматическое выключение моторов после посадки зависит от параметра автопилота `COM_DISARM_LAND` (настройка PX4, которую можно посмотреть в QGroundControl – программе мониторинга и настройки автопилота).

Источник: COEX, идея ожидания цели и завершения посадки. Проверено 11.10.2026.

Границы обработки ошибок

Циклы ожидания имеют таймаут, но отдельный синхронный вызов сервиса может зависнуть при проблеме передачи данных. Потеря связи, завершение ROS, остановленный PX4 или аварийное завершение процесса могут помешать посадке из блока `finally`. Модуль не заменяет внешнюю программу-надзирателя (supervisor, отслеживающую состояние дрона и принимающую решения вне миссии) и не гарантирует посадку при любом отказе. После ошибки проверяйте фактическое состояние стенда.

Если ожидание посадки (`land_wait`) истекло, опыт нельзя считать успешным только потому, что модель стоит на полу: проверьте `armed`, режим и настройку автоматического disarm в QGroundControl. Принудительный disarm в воздухе мгновенно прекращает тягу и не является штатной посадкой.

Практика. Отказ должен быть заметен

После успешной посадки повторите опыт с несуществующим фреймом `no_such_frame`. **Проверка:** программа сообщает отказ, прекращает маршрут и пытается сесть. Сохраните ответ сервиса, затем верните правильный фрейм.

Итоги и самопроверка

- Управление Клевером – четыре сервиса: `get_telemetry`, `navigate`, `set_position`, `land`; ответ `success` означает принятие команды.
- Типы сервисов Clover создаются при сборке catkin; pip их не заменяет.
- Ожидание цели – проверка расстояния и таймаут, а не пауза.
- Модуль flight_tools контролирует связь, режим и посадку, но не гарантирует её при любом отказе.

Проверьте себя: 1) Зачем таймаут в цикле ожидания? 2) Какие аргументы означают «подняться на метр от текущего уровня»? 3) Что означает режим OFFBOARD?
Ответы: 1) без него отказ превращается в бесконечное ожидание; 2) frame=body, z=1; 3) автопилот выполняет цели, присланные внешним (бортовым) компьютером.

ГЛАВА 14

Разбираем полётную программу по шагам

Цели главы

- прочитать код помощника Flight строка за строкой;
- понимать роль таймаута и контекста mission;
- написать свою короткую миссию.

Нужно знать заранее: главы 13.

14.1 Сначала наблюдаем

Прежде чем менять положение, научимся читать состояние. Файл `telemetry_watch.py` десять реальных секунд печатает XYZ, armed и режим. Он не отправляет полётных команд.

```
#!/usr/bin/env python3
"""Read-only telemetry for ten wall-clock seconds."""
import math
import time
import rospy
from clover import srv

rospy.init_node('course_watch')
rospy.wait_for_service('get_telemetry', timeout=10)
get_state = rospy.ServiceProxy('get_telemetry', srv.GetTelemetry)
deadline = time.monotonic() + 10
while not rospy.is_shutdown() and time.monotonic() < deadline:
    state = get_state(frame_id='map')
    valid = state.connected and all(math.isfinite(v)
                                    for v in (state.x, state.y, state.z))
    print(f'valid={valid} armed={state.armed} mode={state.mode} '
          f'xyz=({state.x:.2f}, {state.y:.2f}, {state.z:.2f})')
    time.sleep(0.5)
```

`from clover import srv` подключает сгенерированные типы сервисов. `srv.GetTelemetry` описывает структуру запроса и ответа. `get_state` – созданный нами прокси; его имя не обязано совпадать с названием сервиса. `frame_id='map'` сообщает, в какой системе координат нужны XYZ.

Поле connected показывает связь с автопилотом. Armed показывает, включены ли моторы (глава 13), с точки зрения автопилота. Mode – текущий режим. Конечные XYZ подтверждают, что числовое положение доступно в выбранном фрейме. Ни одна из этих проверок по отдельности не заменяет остальные.

Наблюдение без команд

Запустите наблюдатель до полёта и запишите одну строку. Затем наблюдайте первую короткую миссию в отдельном терминале. Найдите изменения armed и mode. Не запускайте вторую миссию: наблюдатель только читает данные, а управление должно оставаться у одной программы.

14.2 От сервиса к помощнику Flight

В миссии `flight = Flight()` создаёт объект, в котором уже настроены прокси. Его метод `go` делает четыре вещи: отправляет сервису navigate запрос, проверяет ответ, наблюдает расстояние до цели и ограничивает ожидание. Поэтому одна строка миссии соответствует небольшому алгоритму.

```
flight.go(z=1.0, frame='body', auto_arm=True)
```

`z=1.0` – целевое относительное смещение вверх. `frame='body'` – система координат. `auto_arm=True` разрешает сервису самостоятельно включить моторы (arm) и перейти в режим OFFBOARD для взлёта. Значения x и y остаются нулевыми по умолчанию. Метод не возвращается сразу после принятия команды: он ждёт выполнения своего критерия прибытия.

Следующая строка `flight.go(x=1.0, z=0, frame='body')` задаёт движение вперёд без относительного изменения высоты. Это не значит «лететь на высоте ноль»: в body z описывает смещение. Дайте прогноз положения на бумаге до запуска.

• Задание

Откройте `flight_tools.py` и найдите, где создаётся `ServiceProxy`. **Проверка:** вы назвали номера строк.

14.3 Приняли команду – ещё не прибыли

Один вызов navigate возвращает объект ответа. Сначала проверяют поле success. Если оно false, дальнейший маршрут не имеет смысла. Если true, требуется ждать результата движения. Рассмотрим чтение остатка до цели отдельно.

```
#!/usr/bin/env python3
"""Read-only residual-distance monitor after a separately issued goal."""
import math
import time
import rospy
from clover import srv

rospy.init_node('course_arrival_monitor')
rospy.wait_for_service('get_telemetry', timeout=10)
get_state = rospy.ServiceProxy('get_telemetry', srv.GetTelemetry)
deadline = time.monotonic() + 20
stable = 0
while not rospy.is_shutdown():
    state = get_state(frame_id='navigate_target')
    values = (state.x, state.y, state.z)
```

```
    if not state.connected or not all(math.isfinite(v) for v in values):
        raise RuntimeError('Residual coordinates unavailable')
    distance = math.sqrt(sum(v * v for v in values))
    print(f'Residual: {distance:.3f} m')
    stable = stable + 1 if distance < 0.15 else 0
    if stable >= 3:
        print('Position within tolerance; this monitor does not land the drone.')
        break
    if time.monotonic() >= deadline:
        raise TimeoutError('Monitor deadline exceeded')
    time.sleep(0.1)
```

Этот файл только наблюдает уже назначенную цель. Его можно запустить в окне C во время короткой миссии из главы 13. Сам он не задаёт точку и не сажает аппарат. `navigate_target` появляется после назначения цели; до этого чтение может дать некорректные координаты, и монитор сообщит об ошибке.

Расстояние $d = \sqrt{x^2 + y^2 + z^2}$ сравнивается с допуском 0.15 м. Счётчик stable увеличивается при подходящем измерении и сбрасывается при выходе из допуска. Три последовательных измерения уменьшают вероятность завершения по одному случайному попаданию. Это всё ещё позиционный критерий, а не проверка полной остановки.

• Задание

Объясните разницу «команду приняли» и «долетели» на примере печати документа. **Проверка:** принтер принял задание, но бумага ещё не вышла.

14.4 Зачем нужен таймаут

Причины отсутствия прибытия могут быть разными: неверный фрейм, недоступная цель, потеря данных, пауза Gazebo. Без крайнего срока программа может ждать бесконечно. Таймаут превращает неопределённое ожидание в явный результат «не удалось завершить за отведённое время».

Время ожидания выбирайте по длине пути и скорости, затем добавляйте запас на разгон и стабилизацию. Если RTF=0.5 (симуляция идёт вдвое медленнее реального времени, глава 8), путь, занимающий 5 симуляционных секунд, займёт около 10 реальных. В наших помощниках крайний срок идёт по реальным часам. Нельзя делать таймаут очень коротким только ради быстрого сообщения об ошибке.

Две шкалы времени

ROS-время в симуляторе описывает события виртуального мира. Реальные монотонные часы измеряют, сколько вы ждёте за компьютером. Первая шкала подходит для траектории и сравнения временных меток сообщений (timestamp – момент, когда сообщение создано). Вторая помогает ограничивать ожидание даже при паузе симулятора.

• Задание

Что произойдёт без таймаута, если цель недостижима? **Проверка:** программа будет ждать бесконечно.

14.5 Сохраняем точку дома

После взлёта строка `home = flight.check('map')` получает координаты. Это снимок состояния в этот момент, а не постоянно обновляющаяся переменная. Если дрон потом полетит, сохранённые `home.x` и `home.y` останутся числами из того снимка.

Цель `x=home.x+0.8, y=home.y, z=home.z` расположена на 0.8 м вдоль X карты от сохранённого дома. Она не поворачивается вместе с новым yaw аппарата. Мы явно задаём абсолютные координаты в map и тем самым делаем геометрию маршрута понятной.

• Задание

Объясните, зачем координаты старта запоминают после взлёта, а не задают числами. **Проверка:** старт может быть не в нуле карты; относительные цели считаются от измеренной точки.

14.6 Контекст mission и посадка

```
with flight.mission():
    flight.go(z=1.0, frame='body', auto_arm=True)
    home = flight.check('map')
    flight.go(x=home.x + 0.8, y=home.y, z=home.z, frame='map')
    flight.go(x=home.x, y=home.y, z=home.z, frame='map')
```

Отступ определяет границы блока with. После выхода помощник пытается вызвать land и дождаться disarm. Внутри используются try/finally, поэтому попытка выполняется и при исключении в маршруте. Это удобная привычка структурирования, но потеря связи может помешать самой попытке.

Если посадка тоже завершилась ошибкой, Python может показать и исходную, и новую ошибку. Сохраните весь traceback: последняя строка описывает последнюю проблему, а более ранний контекст объясняет причину прерывания маршрута. Не скрывайте ошибку пустым `except: pass`.

• Задание

Что произойдёт при ошибке внутри блока `with flight.mission()`? **Проверка:** будет сделана попытка посадки и ожидание disarm.

14.7 Создайте свою короткую миссию

Скопируйте `first_flight.py` в `my_mission.py`. Сначала измените только расстояние шага на 0.6 м. Затем добавьте сохранение дома и возврат через map. Запустите один опыт, подтвердите посадку и сохраните рабочую версию. После этого добавьте вторую точку.

Записывайте каждый вызов go в таблицу: frame, x, y, z и ожидаемое положение. В body запишите смещение, в map – координату. Если таблица не складывается, исправьте её до полёта. Это быстрее, чем разбираться с неожиданным движением в графике.

14.8 Проверка входных данных

Полётной программе нельзя принимать любое число из аргумента. Скорость должна быть конечной и положительной. Координаты должны соответствовать рабочему пространству,

высота – свободному уровню. Допуск слишком большой допускает завершение далеко от цели; слишком маленький может сделать ожидание бесконечным до таймаута.

```
import math
speed = 0.3
if not math.isfinite(speed) or not 0.1 <= speed <= 0.6:
    raise ValueError('Speed outside the selected experiment limits')
```

Границы 0.1–0.6 м/с – параметры нашего короткого учебного опыта, а не универсальные пределы Клевера. Рабочее пространство и модель должны позволять маршрут. Программа проверяет выбранную договорённость до начала управления.

Разбор без запуска

Возьмите `courier.py`. Подчеркните создание ноды, создание помощника, включение моторов, сохранение точки дома, цикл и возврат. Объясните, в каком месте начинается посадка. Дополнительное задание: предскажите, что произойдёт, если список станций пустой, и проверьте это на чтении кода.

Итоги и самопроверка

- Наблюдение безопасно: чтение телеметрии не включает моторы.
- Ожидание = расстояние + допуск + несколько подряд измерений.
- mission() сажает дрон при ошибке.

Проверьте себя: 1) Почему допуск проверяют несколько раз подряд? 2) Что делает finally?
Ответы: 1) один случайный отсчёт не доказывает остановку; 2) выполняется и при ошибке – там пытаются посадить.

ГЛАВА 15

Маршруты, циклы и траектории

Цели главы

- построить квадрат, змейку и окружность;
- понять разницу между последовательностью точек и непрерывно меняющейся целью;
- рассчитать шаг змейки по углу обзора камеры.

Нужно знать заранее: глав 13 и 6 (списки и циклы).

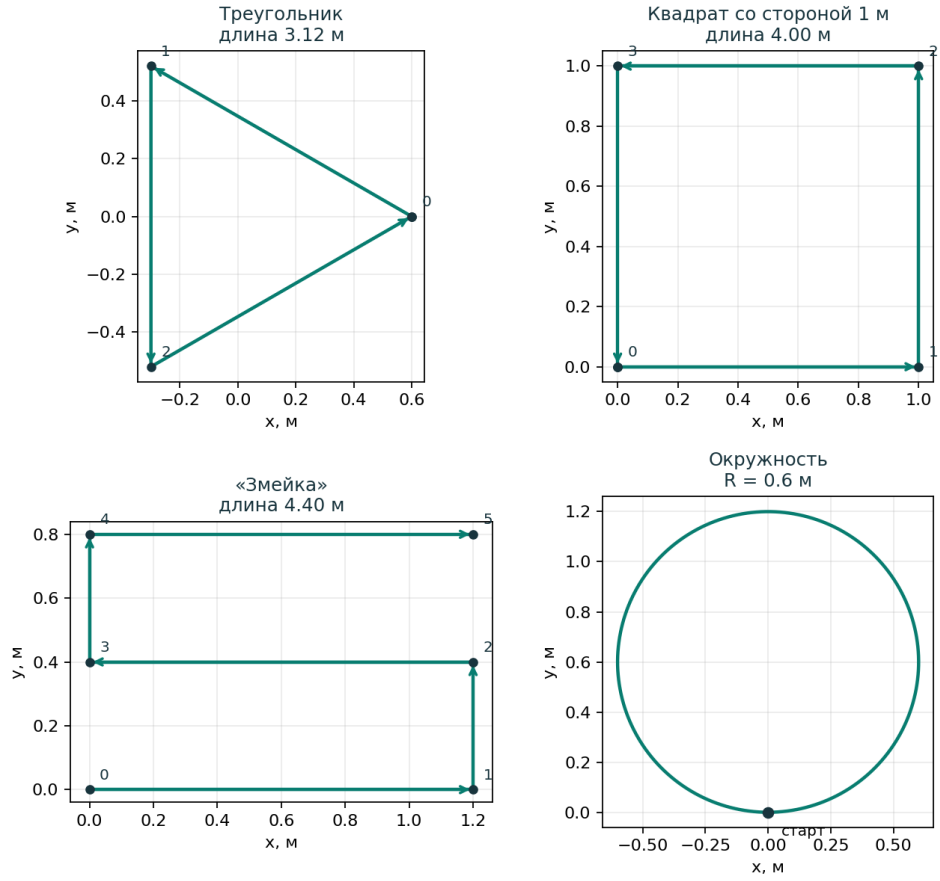


Рис. 15.1. Четыре маршрута главы, построенные по тем же формулам и тому же коду, что и в полётных программах. Цифры – порядок точек, стрелки – направление, указана длина пути. Рисунок построен программой `tools/make_figures.py`.

Перед тем как читать код, изучите рисунок: у каждого маршрута своя «форма» и своя длина. Длину пути можно посчитать до полёта – это первая метрика для сравнения.

15.1 Квадрат в фиксированной системе

Вместо четырёх относительных шагов запомним положение после взлёта и зададим абсолютные точки в `map`. Это удобно для сравнения нескольких повторов. Не предполагаем, что старт всегда находится в $(0, 0, 1)$: исходную высоту берём из телеметрии.

```
#!/usr/bin/env python3
import rospy
from flight_tools import Flight

rospy.init_node('course_square')
flight = Flight()
with flight.mission():
    flight.go(z=1, frame='body', auto_arm=True)
    origin = flight.check('map')
    for dx, dy in [(1, 0), (1, 1), (0, 1), (0, 0)]:
        flight.go(x=origin.x + dx, y=origin.y + dy,
                  z=origin.z, frame='map')
```

Сторона квадрата – 1 м. Первую цель назначаем как смещение от измеренного старта, следующие – как смещения от того же старта. Эта программа сохраняет yaw: помощник передаёт в `navigate` значение `yaw=NaN`, которое означает «не менять поворот» (глава 13). Поэтому коптер не обязан «смотреть по направлению движения».

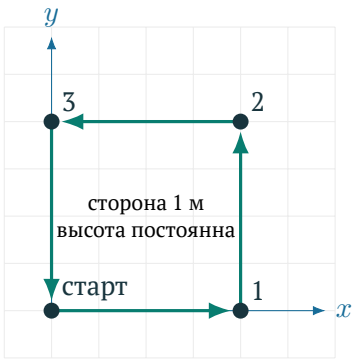


Рис. 15.2. Схема целей квадрата; начало показано относительно измеренного старта.

• Задание

Рассчитайте длину пути квадрата со стороной 1.5 м и время при скорости 0.3 м/с без учёта разгона. **Проверка:** 6 м и 20 с.

15.2 Обследование «змейкой»

«Змейка» – обход поля параллельными строками: по первой строке вперёд, по второй назад и так далее. Для учебного поля создайте списки координат X и Y. В чётной строке (номер 0, 2, …) X проходится слева направо, в нечётной – справа налево. Передайте каждую точку в `flight.go`. Это сокращает пустые возвраты, но ещё не гарантирует полное покрытие камерой.

```
# Fragment inside an existing flight.mission(), after takeoff.
origin = flight.check('map')
xs = [0.0, 0.5, 1.0, 1.5]
ys = [0.0, 0.5, 1.0]
for row, dy in enumerate(ys):
    scan = xs if row % 2 == 0 else list(reversed(xs))
    for dx in scan:
        flight.go(x=origin.x + dx, y=origin.y + dy,
                  z=origin.z, frame='map')
        # Capture/report only after a separate settling check if needed.
```

Если камера смотрит вниз, ширина видимой на полу полосы зависит от высоты и *угла обзора* (угла, в пределах которого объектив видит сцену). В простом приближении для плоского пола:

$$W = 2h\tan(\alpha/2), \qquad \Delta \le W(1-q),$$

где h – расстояние до пола, α – угол обзора по выбранной оси, q – желаемая доля перекрытия. Например, при $h = 1.5$ м, $\alpha = 60^\circ$ и $q = 0.3$ получается $W \approx 1.73$ м и шаг не более 1.21 м. Это расчётная модель, а не утверждение об объективе вашего комплекта: параметры камеры берите из конфигурации и калибровки.

• Задание

Рассчитайте ширину обзора и максимальный шаг змейки для $h = 1$ м, угла 60° и $q = 0.3$. **Проверка:** $W = 2\tan 30^\circ = 1.155$ м; шаг около 0.81 м.

15.3 Плавная окружность

`navigate` хорош для отдельных переходов, а `set_position` – для потока целей. Возьмём радиус $R = 0.6$ м и угловую скорость $\omega = 0.35$ рад/с (угол, на который поворачивается направление на дрон из центра окружности за секунду; радианы – в главе 17). Скорость вдоль окружности равна $v = R\omega = 0.21$ м/с; центростремительное ускорение (ускорение, направленное к центру и удерживающее движение по окружности) $a = R\omega^2 \approx 0.074$ м/с^2.

Чтобы первая цель совпала с текущей позицией, центр окружности сместим по Y:

$$x = x_0 + R\sin(\omega t), \qquad y = y_0 + R[1-\cos(\omega t)], \qquad z = z_0.$$

```
#!/usr/bin/env python3
import math
import time
import rospy
from flight_tools import Flight

rospy.init_node('course_circle')
flight = Flight()
radius, omega = 0.6, 0.35
with flight.mission():
    flight.go(z=1.2, frame='body', auto_arm=True)
    origin = flight.check('map')
    started = rospy.get_time()
    wall_deadline = time.monotonic() + 90
    while not rospy.is_shutdown():
        if time.monotonic() > wall_deadline:
```

```
            raise TimeoutError('Circle timed out or simulation paused')
        angle = omega * (rospy.get_time() - started)
        if angle < 0:
            raise RuntimeError('Simulation time moved backwards')
        if angle >= 2 * math.pi:
            break
        state = flight.check('map')
        if not state.armed or state.mode != 'OFFBOARD':
            raise RuntimeError('Flight left OFFBOARD')
        # Starts exactly at origin; centre is (origin.x, origin.y+radius).
        flight.accepted(flight.set_position(
            x=origin.x + radius * math.sin(angle),
            y=origin.y + radius * (1 - math.cos(angle)),
            z=origin.z, yaw=float('nan'), frame_id='map'))
        time.sleep(0.05)
    flight.go(x=origin.x, y=origin.y, z=origin.z)
```

Источник: COEX, назначение set_position для сложных траекторий. Проверено 11.10.2026.

Угловая скорость определяет движение цели, а не частоту сервисных вызовов. В примере угол вычисляется по симуляционному времени, а вызовы происходят примерно каждые 0.05 реальной секунды. На замедленном стенде это больше вызовов на симуляционную секунду; на ускоренном – меньше. Для серьёзного эксперимента обеспечьте частоту в часах ROS и отдельно контролируйте зависания. Повторную отправку текущей цели автопилоту (setpoint – «заданное значение») выполняет сам simple_offboard (глава 3).

Начало движения цели непрерывно по позиции, но скорость в начале скачком меняется с нуля до $R\omega$. Для более плавного входа потребуется разгон угловой скорости. Это дополнительное задание, когда базовая окружность работает.

Практика. Сравнение маршрутов

Повторите квадрат и окружность по три раза. Измерьте время, максимальное отклонение высоты и ошибку возвращения к старту. Отдельно объясните различие между последовательностью точек и непрерывно меняющейся целью. Усложнение: измените сторону квадрата или радиус без изменения логики программы.

Итоги и самопроверка

- Квадрат – четыре точки; окружность – поток целей по времени.
- Шаг змейки зависит от высоты, угла обзора и перекрытия.
- Замкнутый список точек и точное возвращение модели – разные результаты.

Проверьте себя: 1) Чему равна скорость на окружности R=0.6 м при ω=0.35 рад/с? 2) Для чего нужно перекрытие строк змейки?
Ответы: 1) $v = R\omega = 0.21$ м/с; 2) чтобы между проходами камеры не осталось непросмотренной полосы.

ГЛАВА 16

Миссии: фигуры, осмотр и доставка

Цели главы

- изменять готовые миссии осмысленно;
- запускать миссию с параметрами без правки файла;
- просматривать маршрут до полёта.

Нужно знать заранее: глав 15 и 14.

16.1 Как работать с коллекцией

Для каждой программы сначала прочитайте алгоритм, нарисуйте точки и проверьте свободное место в мире. Затем запустите один опыт, дождитесь disarm и сохраните журнал. Новый пример не требует нового автопилота: меняется логика вашей миссии, а интерфейс go остаётся тем же.

Все файлы лежат рядом с `flight_tools.py` в каталоге examples. Числа подобраны для небольшого свободного участка; этот участок нужно подтвердить в вашей сцене. Непрерывное видео удобно для показа, но для оценки точности используйте телеметрию.

• Задание

Выберите одну миссию коллекции и заполните строку таблицы проверки: ожидание, запуск, наблюдение. **Проверка:** в строке есть ожидаемый результат до запуска.

16.2 Треугольник вокруг дома

В `triangle.py` используется генератор `polygon` – функция из файла `route_math.py` (он разбирается в главе 17): по числу вершин и радиусу она возвращает список точек правильного многоугольника. Она создаёт три вершины на окружности радиуса 0.6 м и повторяет первую в конце, чтобы замкнуть маршрут. Дрон сначала подлетает к первой вершине, проходит треугольник, затем возвращается в центр.

```
#!/usr/bin/env python3
"""Closed triangle around the takeoff origin, then return home."""
import rospy
from flight_tools import Flight
from route_math import polygon

rospy.init_node('course_triangle')
flight = Flight()
with flight.mission():
    flight.go(z=1.0, frame='body', auto_arm=True)
    home = flight.check('map')
    for dx, dy in polygon(n=3, radius=0.6):
        flight.go(x=home.x + dx, y=home.y + dy, z=home.z,
```

```
                  frame='map', speed=0.3)
    flight.go(x=home.x, y=home.y, z=home.z, frame='map')
```

Это отличается от Blockly-треугольника, который начинался в одной из вершин. Здесь home – центр фигуры. При n=4 тот же генератор даст квадрат, расположенный вершинами на осях карты. Если хотите квадрат с горизонтальными сторонами, понадобится поворот фигуры или другой список координат.

Изменение геометрии

Сравните n=3 и n=4 при одинаковом радиусе. Измерьте длину стороны по координатам. Объясните, почему одинаковый радиус не означает одинаковую сторону. Дополнительное задание: найдите формулу стороны правильного n-угольника.

16.3 Змейка: маршрут осмотра поля

На каждой строке аппарат проходит от одного края к другому, затем смещается на следующую строку. Генератор чередует порядок концов: чётная строка идёт от 0 к width, нечётная – обратно. Это исключает ненужный возврат в начало каждой строки.

```
#!/usr/bin/env python3
"""Three-row survey; generate offsets before starting the flight."""
import rospy
from flight_tools import Flight
from route_math import snake

points = snake(rows=3, width=1.2, step=0.4)
rospy.init_node('course_snake')
flight = Flight()
with flight.mission():
    flight.go(z=1.0, frame='body', auto_arm=True)
    home = flight.check('map')
    for index, (dx, dy) in enumerate(points):
        rospy.loginfo('Survey point %d', index)
        flight.go(x=home.x + dx, y=home.y + dy, z=home.z,
                  frame='map', speed=0.3)
    flight.go(x=home.x, y=home.y, z=home.z, frame='map')
```

Первый пункт совпадает с home. В этой миссии возврат из последней точки задан прямым переходом. Поэтому свободным должен быть и диагональный коридор возврата. Если там находится препятствие, добавьте промежуточные цели. Генератор точек сам по себе не планирует обход.

Начальная ширина 1.2 м, шаг между строками 0.4 м. Шаг выбран для демонстрации, а не рассчитан по обзору камеры. Для настоящего осмотра нужно определить площадь кадра на полу при рабочей высоте, желаемое перекрытие и направление камеры.

• Задание

Измените в `route_math.snake` число строк на 4. Какова длина пути при ширине 1.2 м и шаге 0.4 м? **Проверка:** $4 \cdot 1.2 + 3 \cdot 0.4 = 6.0$ м.

16.4 Три высоты: исследуем вертикальный переход

```
#!/usr/bin/env python3
"""Three absolute heights above the preflight map origin."""
import time
import rospy
from flight_tools import Flight

rospy.init_node('course_heights')
flight = Flight()
ground = flight.check('map')
with flight.mission():
    for index, height in enumerate([0.8, 1.0, 1.2]):
        flight.go(x=ground.x, y=ground.y, z=ground.z + height,
                  frame='map', auto_arm=(index == 0), speed=0.3)
        for _ in range(10):
            state = flight.check('map')
            if not state.armed or state.mode != 'OFFBOARD':
                raise RuntimeError('Unexpected flight state')
            time.sleep(0.1)
```

До включения моторов программа сохраняет координаты точки на земле (в коде – `ground`). Каждая высота прибавляется к `ground.z`. Значения 0.8, 1.0 и 1.2 м относятся к одному стартовому уровню, поэтому подъёмы не суммируются как 0.8+1.0+1.2.

На каждом уровне выполняется десять коротких наблюдений. Проверяется режим и armed. Эти наблюдения занимают примерно одну реальную секунду; это не точное время виртуальной стабилизации. Для эксперимента запишите CSV и сравните среднюю высоту на выбранном установившемся интервале (участке, где колебания после подъёма уже затухли). CSV – текстовый файл-таблица: значения одной строки разделены запятыми.

Сравнение двух способов

Сопоставьте `heights.py` и ступени в Blockly. В одном случае перечислены абсолютные уровни от ground, в другом – относительные прибавки от заданной цели. Составьте таблицу ожидаемых высот и докажите, что геометрические цели совпадают при одинаковом начале.

16.5 Курьер: список станций и события

```
#!/usr/bin/env python3
"""Delivery is simulated by a log message; there is no gripper."""
import rospy
from flight_tools import Flight

rospy.init_node('course_courier')
flight = Flight()
with flight.mission():
    flight.go(z=1.0, frame='body', auto_arm=True)
    home = flight.check('map')
    jobs = [('A', 0.8, 0.0), ('B', 0.8, 0.8), ('C', 0.0, 0.8)]
    for name, dx, dy in jobs:
        flight.go(x=home.x + dx, y=home.y + dy, z=home.z, frame='map')
        rospy.loginfo('SIMULATED delivery at station %s', name)
```

```
    flight.go(x=home.x, y=home.y, z=home.z, frame='map')
```

Станция хранится как тройка: имя и два смещения. Цикл распаковывает их. После прибытия программа пишет событие доставки. Это модель задания: груз не меняет массу, захват не двигается, выводы GPIO платы (глава 11) не выполняют действие. Для первого проекта достаточно проверить посещение станций и журнал.

Измените порядок станций и сравните длину маршрута. При одинаковом наборе точек путь может стать длиннее. Более сложная задача – подобрать порядок посещения по общей длине; сначала решите её офлайн на координатах и лишь затем перенесите в миссию.

• Задание

Добавьте четвёртую станцию D в список заданий. Что нужно изменить в коде? **Проверка:** одну строку списка заданий с именем и смещениями.

16.6 Квадрат с настройками без редактирования файла

Программа `route_parameters.py` читает приватные параметры (глава 7) `side` (сторона квадрата), `height` (высота) и `speed` (скорость). Она проверяет выбранные пределы до создания миссии.

```
#!/usr/bin/env python3
"""A configurable square: ROS private parameters, validated preflight."""
import math
import rospy
from flight_tools import Flight

rospy.init_node('course_route')
side = float(rospy.get_param('~side', 0.8))
height = float(rospy.get_param('~height', 1.0))
speed = float(rospy.get_param('~speed', 0.3))
limits = [(side, 0.2, 1.5), (height, 0.6, 1.5), (speed, 0.1, 0.6)]
if not all(math.isfinite(v) and low <= v <= high for v, low, high in limits):
    raise ValueError('Parameters outside the classroom experiment limits')
flight = Flight()
with flight.mission():
    flight.go(z=height, frame='body', speed=speed, auto_arm=True)
    home = flight.check('map')
    for dx, dy in [(side, 0), (side, side), (0, side), (0, 0)]:
        flight.go(x=home.x + dx, y=home.y + dy, z=home.z,
                  frame='map', speed=speed)
```

```
python3 route_parameters.py _side:=0.6 _height:=1.0 _speed:=0.2
```

После завершения и посадки выполните отдельный опыт со speed 0.4. Не запускайте оба одновременно. Получаются два воспроизводимых режима одного файла; параметры можно записать в отчёт как команду запуска.

• Задание

Какие значения параметров `side`, `height`, `speed` программа примет, а какие отклонит? **Проверка:** принимает side 0.2–1.5, height 0.6–1.5, speed 0.1–0.6; остальные отклоняет.

16.7 Сначала видим путь, потом летим

Файл `route_preview.py` создаёт рисунок змейки в формате SVG (векторное изображение, записанное текстом) средствами стандартного Python. Он работает без ROS и без OpenCV (библиотеки обработки изображений, глава 21). Откройте `route.svg` в браузере после запуска.

```
#!/usr/bin/env python3
"""Offline SVG plan: Python standard library only."""
from pathlib import Path
from route_math import snake, length

points = snake()
scale = 180
margin = 45
width = max(p[0] for p in points) * scale + 2 * margin
height = max(p[1] for p in points) * scale + 2 * margin
pixels = [(margin + x * scale, height - margin - y * scale)
          for x, y in points]
polyline = ' '.join(f'{x},{y}' for x, y in pixels)
svg = [f'<svg xmlns="http://www.w3.org/2000/svg" '
       f'width="{width}" height="{height}" viewBox="0 0 {width} {height}">',
       '<rect width="100%" height="100%" fill="white"/>',
       f'<polyline points="{polyline}" fill="none" '
       'stroke="#087d70" stroke-width="3"/>']
for i, (x, y) in enumerate(pixels):
    svg.append(f'<circle cx="{x}" cy="{y}" r="5" fill="#176c98"/>')
    svg.append(f'<text x="{x+8}" y="{y-8}" font-size="14">{i}</text>')
svg.append('</svg>')
Path('route.svg').write_text('\n'.join(svg), encoding='utf-8')
print('Created route.svg; horizontal route length:', length(points), 'm')
```

В изображении Y направлен вниз, поэтому формула для пиксельного Y вычитает координату карты. Масштаб 180 означает 180 пикселей на метр рисунка. Это масштаб плана, не калибровка камеры. Номера вершин позволяют связать схему и журнал полёта.

• Задание

Запустите `route_preview.py` и откройте созданный `route.svg`. **Проверка:** на рисунке шесть пронумерованных точек, длина пути 4.4 м.

16.8 Таблица проверки каждой миссии

Миссия	До запуска	После запуска
Треугольник	Радиус, центр, свободные рёбра	Замкнутый путь и возврат в home
Змейка	Строки, шаг, коридор возврата	Все точки по порядку
Высоты	Один ground, свободный объём вверх	Три измеренных уровня
Курьер	Станции и порядок	Событие после прибытия
Параметрический квадрат	Проверенные значения параметров	Лог связан с командой запуска

Собственная миссия из двух частей

Объедините осмотр двух строк и посещение одной станции. Сначала создайте список точек офлайн и план SVG, затем используйте go для исполнения. Начальный уровень: все точки на одной высоте. Дополнительный: событие в каждой точке и отдельный анализ длительности переходов. Успех: опыт воспроизводится независимо по вашему файлу и команде запуска.

Итоги и самопроверка

- Параметры отделяют настройку от алгоритма.
- Предпросмотр SVG показывает ошибку маршрута до полёта.

Проверьте себя: 1) Зачем предпросмотр маршрута? 2) Для чего нужна таблица проверки миссии?
Ответы: 1) дешевле найти ошибку на чертеже, чем в полёте; 2) фиксирует ожидаемое и фактическое.

ГЛАВА 17

Математика движения и обратная связь

Цели главы

- считать расстояние, углы и повороты;
- различать разомкнутый и замкнутый алгоритм;
- понимать идею пропорционального управления.

Нужно знать заранее: главы 15.

17.1 Расстояние: зачем нужен квадратный корень

Если остаток до цели равен 0.3 м по X и 0.4 м по Y, общее расстояние в горизонтальной плоскости не 0.7, а $\sqrt{0.3^2 + 0.4^2} = 0.5$ м. Это теорема Пифагора: компоненты направлены перпендикулярно. В пространстве добавляется квадрат остатка по Z.

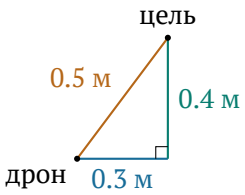


Рис. 17.1. Схема расстояния по двум компонентам.

В Python для двух координат удобно `math.hypot(dx, dy)`. Для XYZ можно использовать квадратный корень суммы квадратов. Допуск 0.15 м описывает шар вокруг целевой точки; попадание в шар ещё не означает нулевую скорость.

• Задание

Остатки до цели 0.6 м по X и 0.8 м по Y. Найдите расстояние. **Проверка:** 1.0 м.

17.2 Градусы и радианы

Радиан – угол, дуга которого на окружности равна её радиусу; это принятая в математике и ROS единица измерения углов. Полный круг – $360°$ или 2π радиан. Поэтому $90° = \pi/2$, $180° = \pi$. API Python Clover принимает углы в радианах; поля Blockly могут показывать градусы. Используйте явный перевод:

```
import math
angle = math.radians(90)
print(angle)               # approximately 1.5708
print(math.degrees(angle)) # approximately 90
```

Передача числа 90 в поле yaw на Python – не то же самое, что поворот на 90 градусов. В показанных миссиях помощник передаёт yaw=NaN для сохранения ориентации; повороты изучайте отдельно с проверкой фактического yaw.

• Задание

Переведите $45°$, $180°$ и $270°$ в радианы. **Проверка:** 0.785; 3.142; 4.712.

17.3 Как повернуть смещение body в map

Пусть yaw равен ψ, а желаемое горизонтальное смещение в body – (b_x, b_y). Тогда соответствующее смещение в map при согласованной ориентации:

$$\Delta x = b_x \cos\psi - b_y \sin\psi, \qquad \Delta y = b_x \sin\psi + b_y \cos\psi.$$

Если yaw=0, шаг вперёд даёт положительное X. Если yaw=$\pi/2$, тот же шаг даёт положительное Y. Для цели прибавьте смещение к текущим координатам: $x_t = x_0 + \Delta x$, $y_t = y_0 + \Delta y$.

Эти формулы помогают предсказать движение, но при работе через фреймы Clover преобразование выполняет система. Не делайте поворот вручную, а затем ещё раз передавайте полученное смещение как body: получится двойное преобразование. Либо задавайте исходное смещение в body, либо рассчитанную абсолютную цель в map.

Бумажная навигация

Дрон в map $(1, 2, 1)$, yaw=$90°$. Найдите цель body $(0.5, 0.2, 0)$. Подсказка: X карты уменьшается на 0.2, Y увеличивается на 0.5. Проверьте также yaw=$180°$ и отрицательное смещение вперёд.

17.4 Генератор правильного многоугольника

На окружности радиуса R вершины можно распределить с одинаковым угловым шагом $2\pi/n$. Каждая вершина имеет координаты $(R\cos\theta_i, R\sin\theta_i)$. Так одна функция строит треугольник, квадрат и другие фигуры.

```
#!/usr/bin/env python3
"""Offline route generation. Coordinates are offsets in metres."""
import math


def polygon(n=3, radius=0.6):
    if n < 3 or not math.isfinite(radius) or radius <= 0:
        raise ValueError('n >= 3 and finite positive radius required')
    points = [(radius * math.cos(2 * math.pi * i / n),
               radius * math.sin(2 * math.pi * i / n))
              for i in range(n)]
    return points + [points[0]]


def snake(rows=3, width=1.2, step=0.4):
    if rows < 1 or not all(math.isfinite(v) and v > 0
                           for v in (width, step)):
        raise ValueError('rows >= 1; finite positive dimensions required')
    points = []
    for row in range(rows):
        ends = (0, width) if row % 2 == 0 else (width, 0)
        points.extend((x, row * step) for x in ends)
```

```
    return points


def length(points):
    return sum(math.hypot(b[0] - a[0], b[1] - a[1])
               for a, b in zip(points, points[1:]))


if __name__ == '__main__':
    for name, points in [('triangle', polygon()), ('snake', snake())]:
        print(name, 'length:', round(length(points), 3), 'm')
        for x, y in points:
            print(f'  ({x:.3f}, {y:.3f})')
```

Функция length складывает расстояния между соседними точками. Она не включает путь от центра до первой вершины, если центр не добавлен в список. Поэтому длина списка polygon и длина всей миссии triangle различаются: миссия ещё подлетает к фигуре и возвращается домой.

• Задание

Сколько точек вернёт `polygon(5, 0.6)`? **Проверка:** шесть: пять вершин и замыкающая точка.

17.5 Разомкнутый и замкнутый алгоритм

Разомкнутый алгоритм отправил действие и дальше не смотрит на результат. Пример: «подождать пять секунд и считать, что долетели». Замкнутый алгоритм получает измерение и сравнивает с целью. В нашем go это остаточное расстояние и состояние полёта.

Обратная связь делает решение зависимым от результата, но качество измерения всё равно важно. Если датчик выдаёт устаревшие координаты, цикл может принять неверное решение. Поэтому пригодность данных включает не только число, но и время, фрейм и источник.

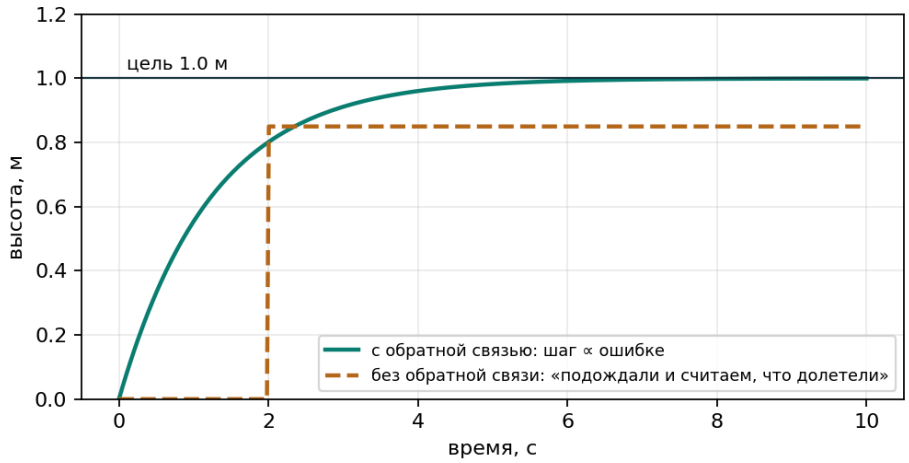


Рис. 17.2. Схема (не измерение): с обратной связью шаг пропорционален остатку до цели, поэтому дрон плавно приближается к цели; без неё программа «считает, что долетела» по времени и может остановиться раньше.

• Задание

Приведите бытовой пример действия без обратной связи и добавьте обратную связь. **Проверка:** пример содержит измерение результата и коррекцию.

17.6 Пропорциональная идея и ограничение скорости

Регулятор – алгоритм, который по ошибке (разности между целью и измеренным значением) вычисляет управляющее действие. Простейший вариант – пропорциональный. Для одной оси его идея $v = k(x_t - x)$: чем дальше цель, тем больше команда скорости. Коэффициент k имеет размерность $1/\text{с}$, если расстояние в метрах, а скорость в м/с. Вблизи цели команда уменьшается. Но при большой ошибке результат может стать слишком большим, поэтому вводят ограничение:

$$v_{cmd} = \max(-v_{max}, \min(v_{max}, ke)).$$

Это учебная формула, а не готовый регулятор для запуска. На устойчивость влияют задержка, динамика, частота и источник позиции. В примерах с navigate большую часть работы выполняют уже имеющиеся компоненты; нам важно понимать, зачем контролировать вход и наблюдать результат.

Для наведения по изображению дополнительно нужен правильный знак и преобразование осей. Если объект справа на кадре, нельзя без геометрии решить, что нужно лететь вправо в body: направление камеры может быть другим. Начните с маленьких диагностических перемещений и записи кадров.

• Задание

При $k = 0.8$ и $v_{max} = 0.4$ м/с найдите скорость для ошибок 0.2 м и 1.0 м. **Проверка:** 0.16 м/с и 0.4 м/с (ограничение).

17.7 Точность и допуск

Допуск отражает цель опыта. Для обзорного посещения станции 0.15 м может подходить, а для небольшой посадочной площадки – нет. Уменьшая допуск, вы увеличиваете требования к позиционированию и можете получить частые таймауты. Поэтому сначала измерьте разброс повторов.

Не сравнивайте координаты float на точное равенство: `x == target_x` почти никогда не является хорошим критерием прибытия. Используйте расстояние, допуск и, при необходимости, скорость и длительность удержания в пределах допуска.

Гипотеза про допуск

На одном коротком переходе сравните допуски 0.10 и 0.20 м при одинаковой скорости. Сделайте по три повтора. Запишите время завершения и расстояние в этот момент. До опыта сформулируйте гипотезу; после него объясните, поддерживают ли её измерения.

Итоги и самопроверка

- Расстояние – корень суммы квадратов компонент.
- API принимает радианы.

- Обратная связь корректирует действие по измеренной ошибке.

Проверьте себя: 1) Чему равно расстояние при смещениях 0.3 и 0.4 м? 2) Сколько радиан в $90°$?
Ответы: 1) 0.5 м; 2) $\pi/2 \approx 1.571$.

ГЛАВА 18

Настраиваем Gazebo под свою задачу

Цели главы

- различать launch, world и model;
- создать свой мир с препятствием без потери физики;
- оценить нагрузку симулятора.

Нужно знать заранее: глав 8 и 4.

18.1 Launch, мир и модель – разные уровни

При настройке симулятора важно различать три уровня. **Launch**-файл (глава 7) запускает процессы и передаёт настройки. **World** («мир») – файл с расширением `.world` в формате SDF (текстовый формат Gazebo с тегами в угловых скобках, *Simulation Description Format*); он описывает сцену, освещение, физику и объекты. **Модель дрона** задаёт корпус, массу, сенсоры и плагины (подключаемые программные модули, которые, например, имитируют работу камеры или моторов). Если нужно добавить коробку, меняйте мир. Если нужна другая камера – описание сенсора. Если нужно выключить графическое окно – аргумент launch.

Сначала смотрите свою установленную версию, поскольку документация может описывать другой релиз:

```
less "$(rospack find clover_simulation)/launch/simulator.launch"
less "$(rospack find clover_description)/launch/spawn_drone.launch"
```

Команда `less` открывает файл для чтения (выход – клавиша q). Ориентируйтесь на объявления `<arg>` (аргументы launch-файла: имя и значение по умолчанию) в самом файле. Наличие аргумента у вложенного launch не делает его автоматически доступным на верхнем уровне.

• Задание

Распределите по уровням: файл с запуском нод; файл, описывающий площадку и свет; описание коробки-препятствия. **Проверка:** launch; world; model.

18.2 Проверенные переключатели

В просмотренном `simulator.launch` объявлены следующие аргументы. Значения относятся к проверенному исходнику, а не ко всем выпущенным образам.

Аргумент	Default	Назначение
`gui`	true	Окно клиента Gazebo (графический интерфейс)
`main_camera`	true	Симуляция основной камеры
`rangefinder`	true	Дальномер вниз (датчик расстояния до пола)
`led`	true	Светодиодная (LED) лента
`gps`	false	Модуль спутниковой навигации GPS
`maintain_camera_rate`	false	Замедление ради частоты камеры
`use_clover_physics`	false	Физические параметры (масса, инерция), полученные из CAD-модели (чертежа аппарата)
`vehicle`	clover	Конфигурация автопилота PX4
`est`	ekf2	Алгоритм оценки положения PX4 (EKF – расширенный фильтр Калмана: объединяет показания датчиков в одну оценку положения)

```
roslaunch clover_simulation simulator.launch gui:=false
roslaunch clover_simulation simulator.launch gps:=true
roslaunch clover_simulation simulator.launch maintain_camera_rate:=true
```

Это **три альтернативных запуска**, а не три команды для одновременного выполнения. `gui:=false` отключает клиентское окно, но виртуальной камере всё равно нужен работающий рендер: режим без окна (headless) не становится автоматически независимым от GPU/дисплея.

Источник: CopterExpress, аргументы simulator.launch. Проверено 11.10.2026.

Включение `gps:=true` добавляет сенсор, но само по себе не гарантирует его использование алгоритмом оценки положения (EKF). Проверьте позицию и параметры оценивателя в QGroundControl (программе настройки и мониторинга автопилота). Не выключайте камеру в конфигурации для помещения, если она обеспечивает optical flow (оптический поток – определение смещения аппарата по смещению изображения пола): сначала подтвердите другой источник позиционирования.

• Задание

Найдите в launch-файле симулятора параметр `use_clover_physics` и запишите значение по умолчанию. **Проверка:** значение – false (см. главу 2).

18.3 Локальный launcher для книги

В проверенном файле название мира передано *внутрь* `gazebo_ros/empty_world.launch`, а верхнего аргумента `world_name` нет. Чтобы выбирать мир и включать Blockly и ArUco, создадим отдельный пакет `clover_course` с собственным launch-файлом (*launcher* – запускающая обёртка над штатным launch Клевера). Скрипт книги читает установленный launcher и добавляет три аргумента, сохраняя остальные настройки конкретной версии.

```
cd "$HOME/catkin_ws/src"
# Only if clover_course has not been created earlier (ROS laboratory chapter):
catkin_create_pkg clover_course rospy clover std_msgs
cd "$HOME/catkin_ws"
catkin_make -j2
source devel/setup.bash
cd "$HOME/clover_course_files/examples"
python3 make_course_launch.py
roslaunch clover_course course.launch
```

Если пакет уже создан в главе 7, пропустите строку `catkin_create_pkg`; создавать его повторно не нужно. Остановите предыдущий симулятор до последней команды. Если пакет ещё не виден в `rospack`, повторно активируйте workspace. Скрипт прекращает работу при неожиданной структуре исходного launcher: просмотрите его вручную, а не игнорируйте проверку.

В новом launcher доступны `course_world`, `course_aruco` и `course_blocks`. По умолчанию выбран штатный `clover_aruco.world`, а два флага включены. Это **аргументы учебного пакета**, а не обещание наличия таких флагов в официальном simulator.launch.

• Задание

Запустите генератор из главы и откройте полученный `course.launch`. **Проверка:** файл создан, оригинальный launch не изменён.

18.4 Собственный мир с сохранением физики

Не создавайте сцену с пустой физикой по памяти. Возьмите штатный мир как основу: документация требует шаг 0.004 с и частоту обновления 250 Гц для используемых плагинов.

```
mkdir -p "$HOME/clover_worlds"
cp "$(rospack find clover_simulation)/resources/worlds/clover_aruco.world" \
  "$HOME/clover_worlds/course.world"
roslaunch clover_course course.launch \
  course_world:="$HOME/clover_worlds/course.world"
```

Чтобы сохранить совместимость, оставьте раздел `physics`, магнитное поле, пол и штатные включения моделей из исходника. Меняйте по одному элементу сцены. Раздел `physics` задаёт параметры физического расчёта (шаг по времени и т. п.). Для постоянного препятствия вставьте модель внутрь тега `<world>`, не заменяя ею весь файл мира.

Источник: COEX, ограничения физики и изменение мира. Проверено 11.10.2026.

18.4.1 Визуальное и физическое препятствие

В SDF `visual` рисует объект, а `collision` участвует в контактах. Для неподвижной коробки с основанием на полу задайте центр на половине высоты. Модель `obstacle.sdf` имеет размер $0.4 \times 0.4 \times 0.8$ м, поэтому при добавлении командой `spawn_model` (она добавляет модель из файла в уже работающий мир) ей нужен Z=0.4.

```
<?xml version="1.0"?>
<sdf version="1.6">
  <model name="course_obstacle">
    <static>true</static>
    <link name="box">
      <collision name="collision">
        <geometry><box><size>0.4 0.4 0.8</size></box></geometry>
      </collision>
      <visual name="visual">
        <geometry><box><size>0.4 0.4 0.8</size></box></geometry>
        <material><ambient>0.15 0.4 0.7 1</ambient>
          <diffuse>0.15 0.4 0.7 1</diffuse></material>
      </visual>
    </link>
  </model>
</sdf>
```

Пока симулятор работает, в другом терминале:

```
rosrun gazebo_ros spawn_model -sdf \
  -file "$HOME/clover_course_files/examples/obstacle.sdf" \
  -model course_obstacle -x 0.8 -y 0.8 -z 0.4
```

Не ставьте коробку на стартовую точку. Если нужен постоянный объект, перенесите содержимое `<model>` в `course.world` и добавьте `<pose>0.8 0.8 0.4 0 0 0</pose>`. Модель, добавленная только через spawn, исчезнет после перезапуска.

• Задание

Скопируйте `obstacle.sdf`, измените положение (pose) на 1 м по X и добавьте в мир. **Проверка:** в Gazebo виден объект в новом месте, физика дрона не изменилась.

18.5 Камера, масса и инерция

Описание дрона написано на языке URDF/xacro (URDF – описание робота как набора тел и соединений, xacro – «шаблонный» вариант URDF с подстановками). В проверенной структуре оно находится в `clover_description/urdf/clover/clover4.xacro`. Перед экспериментом сохраните копию или коммит. Камере задают положение, ориентацию, разрешение и частоту; после изменения перезапускают симуляцию. Смещение камеры требует согласованного TF, иначе изображение выглядит правильно, но оценка позы оказывается неверной.

Смена цвета и размеров visual не меняет физическую модель. Для физики нужны масса, центр масс и корректная матрица инерции. Крупная collision-геометрия может давать ранние столкновения. Не подбирайте коэффициенты регулятора (PID – пропорционально-интегрально-дифференциальный регулятор автопилота) под случайно изменённую массу, если цель задания – программирование маршрута.

• Задание

Объясните своими словами, зачем модели нужны масса и инерция. **Проверка:** без них физика ведёт себя неправдоподобно: объект «не знает», насколько тяжёл и как вращается.

18.6 Производительность

Смотрите Real Time Factor (RTF) на нижней панели Gazebo. Если RTF=0.4, десять симуляционных секунд занимают примерно 25 реальных. Уменьшите разрешение/нагрузку камеры и сложность сцены, выделите VM достаточно ядер. Режим `maintain_camera_rate` может замедлять симуляцию, чтобы кадры не выпадали.

```
PX4_SIM_SPEED_FACTOR=0.4 roslaunch clover_course course.launch \
  maintain_camera_rate:=true
```

Число 0.4 здесь – пример для измеренного RTF около 0.4. Не копируйте его без измерения. Переменная помогает SITL учитывать замедление; она не заставляет слабый компьютер работать быстрее.

Практика. Сцена с препятствием

Создайте локальный launcher и копию мира. Добавьте коробку, затем проложите маршрут через промежуточные точки с просветом не менее 0.3 м от геометрии корпуса до коробки. Зафиксируйте RTF и отсутствие контактов. Объясните, почему простой `navigate` в конечную точку не делает автоматический обход.

Итоги и самопроверка

- Launch запускает программы, world описывает среду, model – объект в ней.
- Visual – что видно, collision – что сталкивается; это разные описания.
- Производительность измеряйте (Real Time Factor), а не угадывайте.

Проверьте себя: 1) Где описывают положение препятствия? 2) Чем collision отличается от visual?
Ответы: 1) в world/SDF-файле; 2) collision участвует в физике, visual только отображается.

ГЛАВА 19

Gazebo как лабораторный стенд

Цели главы

- оценить, что именно моделирует Gazebo;
- создать несколько сценариев-лабораторий;
- измерять нагрузку и реальное время.

Нужно знать заранее: глав 18.

19.1 Что именно моделирует симулятор

Симулятор вычисляет изменения виртуального мира шаг за шагом. У объектов есть положение, масса, форма и взаимодействия. Сенсорная модель создаёт данные, похожие по интерфейсу на данные камеры или дальномера. Автопилот получает эти данные и управляет виртуальным аппаратом.

Так можно исследовать алгоритм без подготовки физического полигона для каждого опыта. Вы быстро меняете расположение станций, сохраняете мир и повторяете маршрут. Но результат относится к конкретной модели. Если модель камеры не включает определённый шум или выдержку, опыт не показывает влияние этих факторов на настоящий сенсор.

• Задание

Назовите три вещи, которые симулятор моделирует упрощённо. **Проверка:** например: шум датчиков, ветер, батарея, задержки связи.

19.2 Gazebo, RViz и QGroundControl

Эти окна показывают разные стороны системы. Gazebo рисует физический мир и вычисляет его состояние. RViz отображает данные ROS: фреймы, изображения, маркеры и другие сообщения. QGroundControl работает с автопилотом, его режимами и параметрами.

Вопрос	Где смотреть	Почему
Стоит ли коробка на полу?	Gazebo	Геометрия мира и контакты
Как повернуты оси фрейма?	RViz / TF	Представление данных ROS
Какой режим PX4?	Телеметрия / QGC	Состояние автопилота
Есть ли кадры?	`rqt_image_view` (окно просмотра изображений топика, глава 8)	Реальный поток изображений
Почему команда отвергнута?	Ответ сервиса / терминал	Сообщение компонента

Модель может красиво двигаться в Gazebo, а TF выбранного сенсора отсутствовать в ROS. Наоборот, RViz может показывать старые или записанные данные без работающего физического мира. Всегда связывайте наблюдение с его источником.

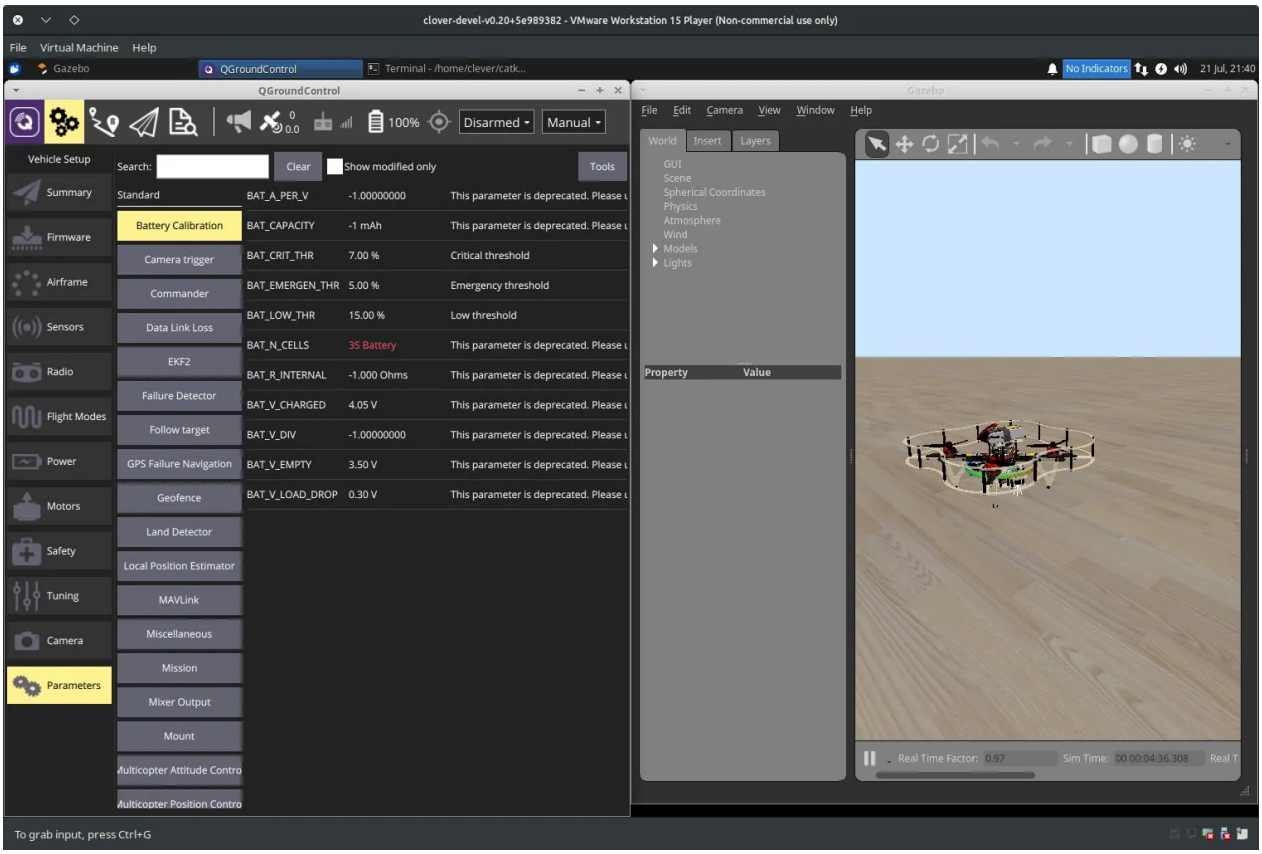


Рис. 19.1. Gazebo и QGroundControl рядом: справа физическая сцена, слева настройки автопилота. Снимок COEX, CC BY-NC-SA 4.0; версии окон могут отличаться.

• Задание

Что показывает каждое из окон: Gazebo, RViz, QGroundControl? **Проверка:** Gazebo – мир; RViz – данные ROS и TF; QGroundControl – состояние автопилота.

19.3 Сценарий 1. Чистая площадка для первых миссий

Начните со штатного мира (мира из комплекта Клевера) и не меняйте физику. Найдите стартовую модель, направления осей и свободную область. Выполните короткий взлёт и посадку, потом шаг и возврат. Только после приёмки добавляйте препятствия.

Запишите размеры свободного участка в системе мира Gazebo и выясните, как он соотносится с map. Для этого сравните положение модели и телеметрию в одной известной точке. Мир Gazebo имеет своё начало, а map может быть связан с инициализацией оценки позиции. Совпадение нулей нельзя предполагать без проверки.

• Задание

Составьте паспорт чистой площадки: размер, положение старта, версия мира. **Проверка:** паспорт позволяет независимо воссоздать сцену.

19.4 Сценарий 2. Учебный склад

Добавьте три станции A, B и C как визуальные площадки. Сохраните их координаты и выберите высоту осмотра. Миссия `courier` (глава 16) посещает соответствующие точки в системе `map` и пишет события. «Станция» здесь – просто названная точка маршрута, отмеченная площадкой. На первом этапе сами определите преобразование координат мира в координаты задания.

В каталоге есть `red_pad.sdf`. Для нескольких станций можно использовать тот же файл с разными именами модели и положениями. Каждое имя при spawn должно быть уникальным. Для постоянного склада перенесите модели в копию world; иначе после перезапуска добавленные во время работы объекты исчезнут.

```
rosrun gazebo_ros spawn_model -sdf \
  -file "$HOME/clover_course_files/examples/red_pad.sdf" \
  -model station_a -x 0.8 -y 0 -z 0.02
rosrun gazebo_ros spawn_model -sdf \
  -file "$HOME/clover_course_files/examples/red_pad.sdf" \
  -model station_b -x 0.8 -y 0.8 -z 0.02
```

Команды используют координаты мира, а не body. Перед размещением убедитесь, что они не перекрывают старт и необходимые маркеры. Третью станцию разместите самостоятельно и подпишите в плане.

• Задание

Нарисуйте план склада: размеры, стеллажи, свободные коридоры. **Проверка:** коридор шире аппарата с запасом, указаны координаты препятствий.

19.5 Сценарий 3. Коридор вокруг коробки

Один navigate не выполняет автоматический поиск пути. Для первого обхода нарисуйте промежуточные точки по свободному коридору. Учтите размеры корпуса и запас: проверяется не только математическая линия через центр дрона.

Например, коробка находится в середине прямого пути. Добавьте точку сбоку от неё, затем точку за ней, затем конечную станцию. Все соседние отрезки должны проходить свободно.

Аппарат может сглаживать переход, поэтому запас нужен и в углах. Сначала изучите путь на малой скорости.

План обхода

Нарисуйте прямоугольник коробки и расширьте его на радиус корпуса плюс выбранный запас. Постройте маршрут вне расширенной области. Сверьте прогноз в Gazebo. Дополнительное задание: объясните, почему отсутствие контакта при одном быстром проходе не доказывает надёжность всего маршрута.

19.6 SDF: прочитайте свою модель

SDF – текстовое описание сцены и моделей в формате XML (теги в угловых скобках, вложенные друг в друга). Тег model содержит объект, link – его физическую часть, visual – внешний вид, collision – форму контакта. Geometry задаёт геометрическую примитивную форму, material – отображение, pose – положение и ориентацию.

```
<pose>0.8 0.8 0.4 0 0 0</pose>
<geometry>
  <box><size>0.4 0.4 0.8</size></box>
</geometry>
```

В pose первые три числа – X, Y, Z в метрах, затем roll, pitch, yaw в радианах относительно родительского элемента. В size – размеры по трём осям. Для коробки высотой 0.8 м центр Z=0.4 помещает основание на Z=0. Это не правило для каждой модели: начало её локальных координат может быть выбрано иначе.

`<static>true</static>` делает объект неподвижным в смысле физической модели. Для учебного препятствия это удобно. Для падающего груза нужна другая модель с массой, инерцией и контактами. Замена static на false сама по себе не даёт физически достоверный груз.

• Задание

Найдите в SDF-файле теги `visual` и `collision`. **Проверка:** оба тега есть; объясните, чем они отличаются.

19.7 Сценарий 4. Проверка камеры

Разместите одну цветную площадку, сохраните кадр, затем переместите площадку и повторите. Сопоставьте направление её смещения на изображении и в мире. Отдельно измените освещение или высоту, сохраняя остальные настройки.

Если цель исчезла, выясните: она вне поля зрения, слишком мала для порога или не проходит по цвету? Снимок кадра отвечает на первый вопрос, *маска* (изображение, где отмечены подходящие по цвету пиксели, глава 21) – на второй и третий. Весь процесс остаётся задачей наблюдения до подключения контроллера.

• Задание

Поставьте красную площадку и сохраните кадр камеры. **Проверка:** красная область видна на кадре.

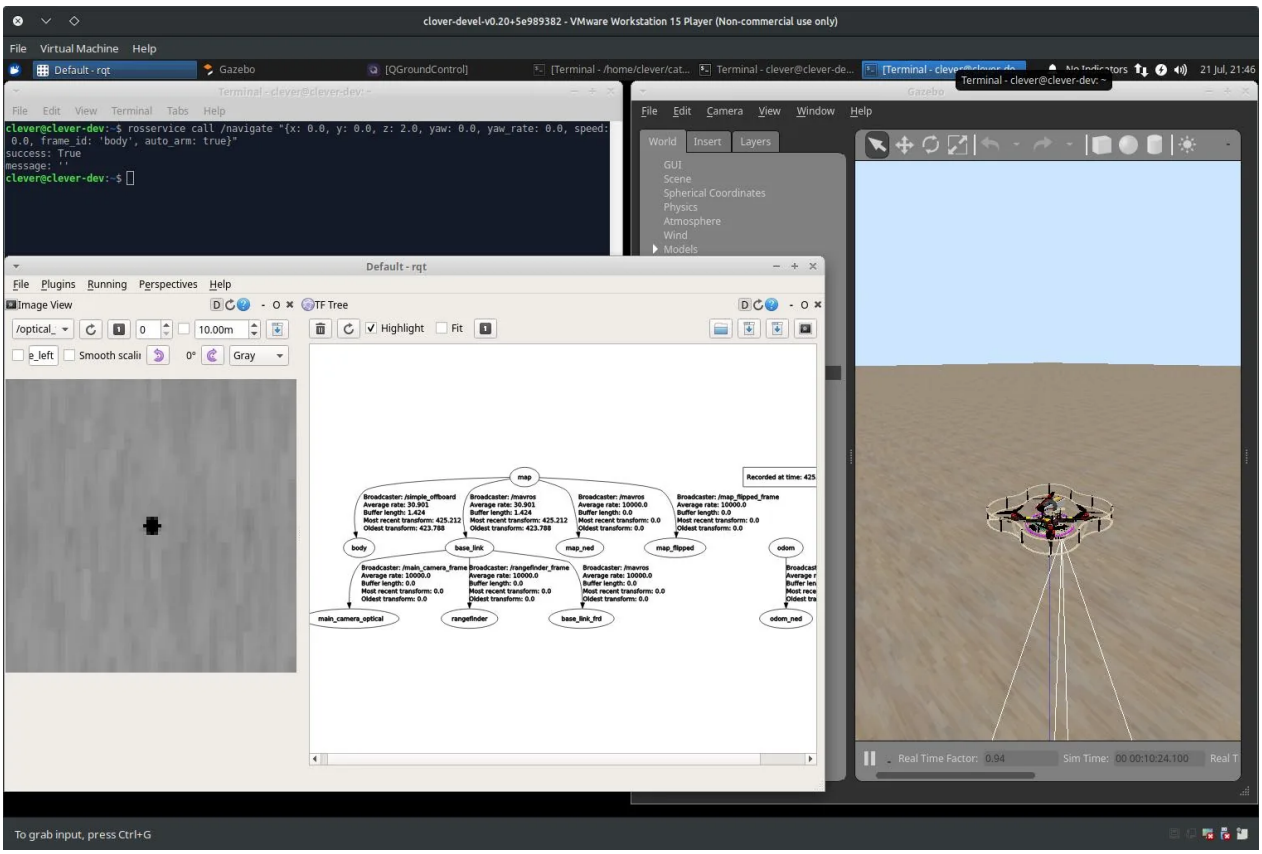


Рис. 19.2. Rqt показывает изображение и дерево TF, Gazebo – виртуальный аппарат. Исторический снимок COEX, CC BY-NC-SA 4.0; команду из снимка не используйте как готовую миссию.

19.8 Сценарий 5. Нагрузка и реальное время

RTF – отношение прошедшего времени симуляции к реальному. При RTF=1 виртуальный мир идёт примерно в реальном темпе. При RTF=0.25 одна виртуальная секунда занимает около четырёх реальных. Высокое значение FPS (число кадров в секунду, которое рисует окно Gazebo) не заменяет измерение RTF физики.

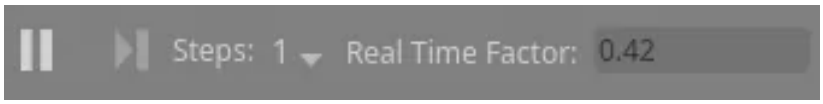


Рис. 19.3. Индикатор Real Time Factor в Gazebo: пример значения 0.42. COEX, CC BY-NC-SA 4.0.

Сравните штатный мир и мир с дополнительными объектами при одинаковых настройках. Запишите RTF и частоту камеры. Затем уменьшите нагрузку одним способом, например отключением клиента gui, и повторите. Не меняйте шаг физики без понимания плагинов: в главе 18 приведены ограничения классического Clover.

Таблица производительности

Для трёх режимов запишите: world, gui, частоту кадров, RTF и время короткой миссии. Сравнивайте реальное и симуляционное время отдельно. Дополнительное задание: покажите, почему медленный стенд может дать таймаут даже при правильном движении.

19.9 Что можно изучать в этой книге

Маршруты и геометрию, работу ROS-интерфейсов, обработку изображений, согласование карты ArUco, организацию миссии, обработку ошибок, планирование коротких переходов и методику эксперимента. Можно проверять, как ваше решение меняется при другой скорости, задержке обработки или расположении станций.

Для задач захвата груза, контакта с поверхностью, сложной аэродинамики или точной оценки энергопотребления нужно отдельно проверить наличие и достоверность соответствующей модели. В штатной учебной сцене команда «доставка завершена» остаётся событием программы. Это нормальный результат первого проекта, если вы точно описываете его смысл.

• Задание

Выберите один вопрос для исследования в симуляторе и назовите измеряемую величину. **Проверка:** вопрос сформулирован так, что ответ – число.

19.10 Паспорт воспроизводимой сцены

Сохраните world, все добавленные SDF, карту маркеров, команду запуска и версии пакетов. Отметьте, какие объекты добавлены через spawn и поэтому не содержатся в world. Запишите начальную позицию, ориентацию, RTF и источник позиционирования. Такой набор позволяет независимо повторить опыт.

Воспроизведение сцены

Подготовьте склад с двумя станциями и одной коробкой. Составьте файлы и инструкцию так, чтобы сцену можно было воспроизвести без устных пояснений: по ним должно получаться запустить сцену, найти станции и выполнить маршрут. Любой вопрос, на который пришлось отвечать голосом, превращайте в улучшение инструкции.

Источник: Gazebo Classic, компоненты симулятора. Проверено 11.10.2026.

Итоги и самопроверка

- RViz показывает данные ROS, Gazebo – мир, QGroundControl – состояние автопилота.
- Паспорт сцены делает опыт воспроизводимым.

Проверьте себя: 1) Что показывает Real Time Factor? 2) Зачем паспорт сцены?
Ответы: 1) во сколько раз симуляция идёт медленнее или быстрее реального времени; 2) чтобы другая пара воспроизвела сцену.

ГЛАВА 20

Навигация по ArUco

Цели главы

- объяснить, что такое ArUco-маркер, как по нему определяют положение камеры и где это применяют;
- создать и распечатать собственные маркеры и сгенерировать карту маркеров командой `genmap.py`;
- загрузить карту на дрон, включить нужные флаги и проверить, что система видит поле;
- выполнить полёт над маркером и по координатам карты, выполнить точную посадку на маркер;
- найти причину, если маркеры не распознаются или позиция «плывёт».

Нужно знать заранее: главы 10 (фреймы, TF), 13 (сервисы управления), 9 (флаги `aruco` и `aruco_map`) и 8 (приёмка).

20.1 Что такое ArUco и зачем он нужен

ArUco-маркер – квадратная метка из чёрных и белых клеток, окружённая белой рамкой. Рисунок клеток кодирует целое число – номер (ID) маркера, а чёрная окантовка позволяет программе найти четыре угла квадрата на кадре. Набор допустимых рисунков называется *словарём*. В Клевере используется словарь `DICT_4X4_250`: сетка 4×4 клетки, 250 разных номеров (0–249).

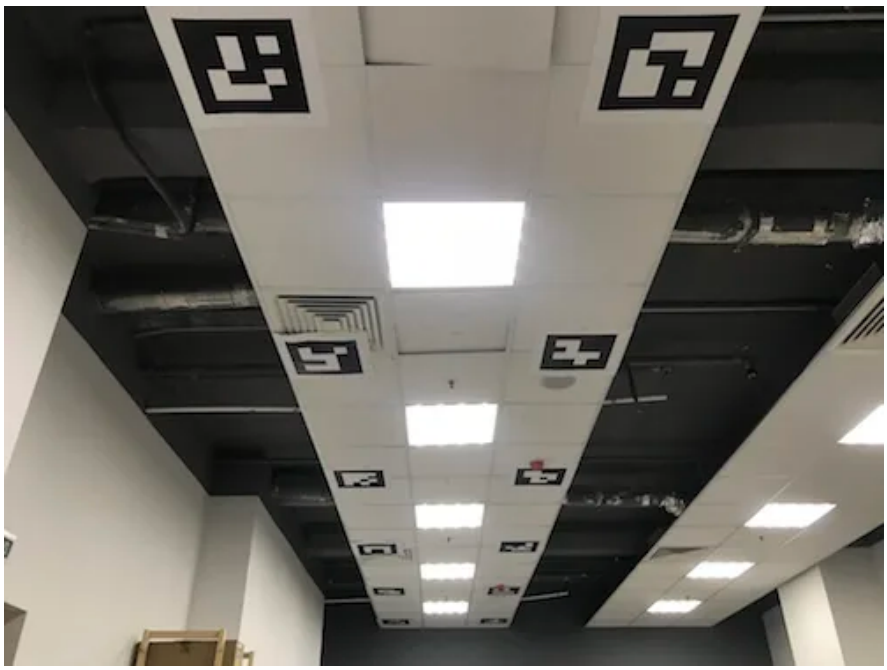


Рис. 20.1. Настоящие ArUco-метки на потолке лаборатории: у каждой свой рисунок и, следовательно, свой номер. Фото COEX, CC BY-NC-SA 4.0.

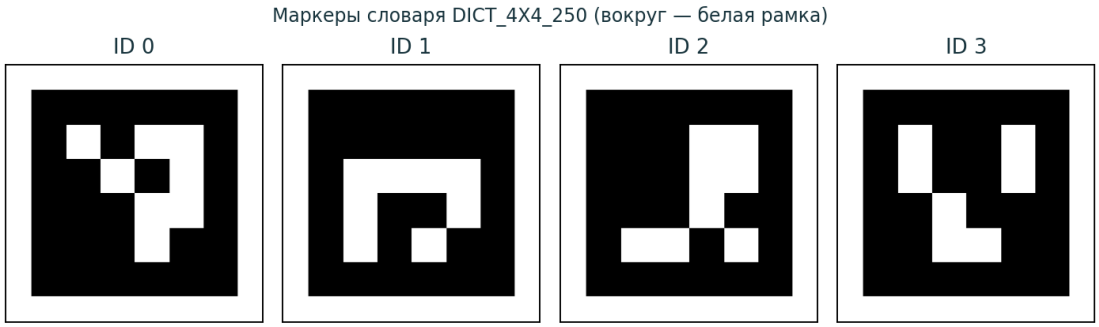


Рис. 20.2. Четыре маркера словаря DICT_4X4_250 с номерами 0–3, сгенерированные программой `make_markers.py`.

Как маркер определяет положение. Если известны реальный размер стороны маркера и параметры объектива (калибровка камеры), то по положению четырёх углов на кадре можно вычислить, где находится маркер относительно камеры: расстояние, направление и поворот. Это задача *оценки позы*. Один маркер даёт положение дрона относительно этого маркера. Несколько маркеров с известными координатами на поле образуют **карту**: по ней дрон вычисляет своё положение в системе координат поля (`aruco_map`), даже если видит только часть маркеров.

Где это применяют.

- *Навигация в помещении*, где нет спутниковой навигации (GPS): ангары, залы, лаборатории, состязательные полигоны. Маркеры на полу или потолке дают привязку к координатам.
- *Точная посадка*: маркер на посадочной площадке позволяет сесть с точностью в единицы сантиметров.
- *Контрольные точки маршрута*: «пролети над маркером 3, затем над маркером 7» – удобное задание для соревнований и учебных миссий.
- *Привязка* камеры, дорожных объектов, ворот и станций к известным координатам; калибровка и проверка систем позиционирования.
- *Внешний источник позиции для автопилота* (механизм VPE, глава 10): автопилот удерживает позицию по маркерам, как по GPS.

Достоинство метода – дешевизна (маркеры печатают на бумаге) и независимость от внешней инфраструктуры. Ограничения: маркеры должны быть видны камере, хорошо освещены и иметь точно известный размер.

• Задание

Приведите два примера задач, где ArUco удобнее GPS, и одно ограничение метода. **Проверка:** примеры – помещение без GPS, точная посадка; ограничение – нужна видимость маркеров, освещение, известный размер.

20.2 Как это устроено в Клевере

Подсистема ArUco состоит из трёх нод; их включают флагами из главы 9.

Нода	Что делает	Что публикует
`aruco_detect`	Находит маркеры на кадре камеры и оценивает положение каждого	Топик `aruco_detect/markers`; фреймы TF `aruco_N`; отладочное изображение `aruco_detect/debug`
`aruco_map`	По карте и найденным маркерам вычисляет положение камеры в системе поля	Топик `aruco_map/pose`; фрейм `aruco_map`; изображения `aruco_map/debug`, `aruco_map/image`
`vpe_publisher`	Передаёт положение автопилоту (при `aruco_vpe=true`)	Топик `mavros/vision_pose/pose`; фрейм `aruco_map_detected`

Система координат маркера: ось X направлена вправо по маркеру, Y – к верхней стороне маркера, Z – от плоскости маркера. В карте используется система ENU: X вправо по полю, Y вверх по полю, Z от плоскости поля.

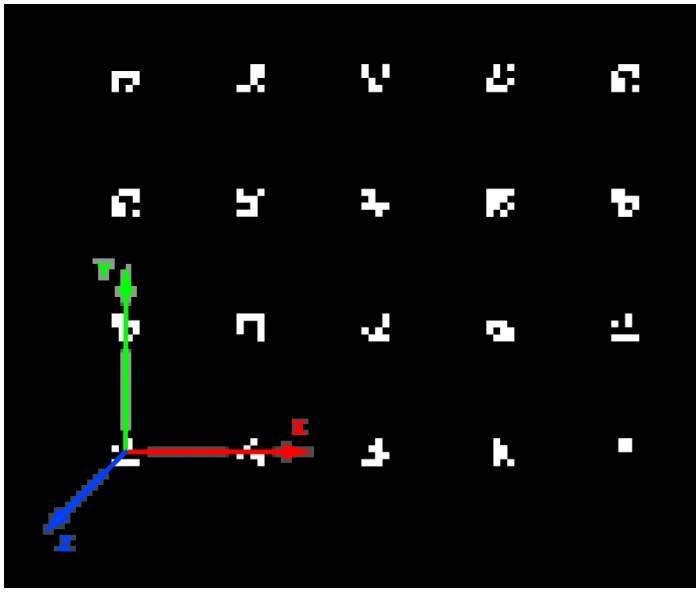


Рис. 20.3. Система координат карты маркеров: оси X и Y лежат в плоскости поля. COEX, CC BY-NC-SA 4.0.

20.3 Подготовка маркеров: размер, печать, наклейка

Генерация изображений. Маркеры можно получить готовыми генераторами или программой книги `make_markers.py`, которая использует тот же словарь, что и детектор:

```
python3 make_markers.py --ids 0 1 2 3 --side-px 800 --out markers
```

Команда создаёт в каталоге `markers` файлы `aruco_000.png`, `aruco_001.png` и т. д.: чёрно-белые квадраты со стороной 800 пикселей и белой рамкой. Ключ `--ids` – список номеров, `--side-px` – размер стороны в пикселях, `--out` – каталог. Программа требует OpenCV с модулем `aruco` и не нуждается в ROS.

Печать. Правила, от которых зависит точность:

1. Печатайте *без масштабирования* (100% или «реальный размер», не «по размеру страницы»).
2. После печати *измерьте линейкой* сторону чёрного квадрата (без белой рамки): это число – длина маркера, которая записывается в карту (например, 0.20 м). Ошибка размера пропорционально искажает все расстояния.
3. Белая рамка вокруг маркера обязательна (не менее одной клетки), иначе детектор не найдёт границу.
4. Бумага матовая (глянец даёт блики), маркер наклеен ровно, без складок, на жёсткую плоскую основу.
5. Маркеры разного номера отличают по рисунку; для карты нужно столько *разных* номеров, сколько маркеров в поле.

Какого размера делать. Чем выше летит дрон и чем ниже разрешение кадра, тем крупнее должен быть маркер. Ориентир: маркер должен занимать на кадре не менее нескольких десятков пикселей по стороне (для кадра 320×240 минимальный размер контролируется параметром `minMarkerPerimeterRate`, глава 9). Для полёта на 1–2 м типичны маркеры 15–33 см. В примере документации COEX используются маркеры 0.33 м.

• Задание

Сгенерируйте и распечатайте маркеры с номерами 10, 11, 12 и 13. Измерьте сторону чёрного квадрата. **Проверка:** файлы `aruco_010.png`…`aruco_013.png` созданы; измеренный размер записан с точностью до миллиметра.

20.4 Карта маркеров: формат файла

Карта – текстовый файл. Каждая строка описывает один маркер:

```
# id    length    x    y    z    rot_z    rot_y    rot_x
1    0.33    0    0    0    0    0    0
2    0.33    1    0    0    0    0    0
3    0.33    0    1    0    0    0    0
4    0.33    1    1    0    0    0    0
```

Столбцы разделены пробелами или табуляцией: `id` – номер маркера; `length` – сторона чёрного квадрата в метрах; `x`, `y`, `z` – координаты *центра* маркера в системе поля, метры; `rot_z`, `rot_y`, `rot_x` – повороты маркера вокруг осей в радианах (для лежащего ровно маркера нули). Строка, начинающаяся с `#`, – комментарий. Начало отсчёта поля $(0, 0, 0)$ вы выбираете сами, но все маркеры должны быть заданы в этой единой системе.

Правило точности. Положение дрона определяется по карте, поэтому ошибка в координатах маркеров прямо превращается в ошибку полёта. Расстояния между центрами маркеров измеряйте рулеткой, а маркеры наклеивайте по разметке.

20.5 Генерация карты командой genmap.py

Для прямоугольной сетки маркеров в Clover есть утилита `genmap.py`. Её запускают на бортовом компьютере (или в симуляторе):

```
rosrun aruco_pose genmap.py length x y dist_x dist_y first -o test_map.txt
```

Аргументы: `length` – размер маркера, м; `x` – число маркеров по оси X; `y` – число маркеров по оси Y; `dist_x`, `dist_y` – расстояния между центрами соседних маркеров по осям, м; `first` – номер первого маркера; `-o` – имя файла (он создаётся в подкаталоге `map` пакета `aruco_pose`). Дополнительные необязательные аргументы: координаты первого маркера `x0`, `y0` (по умолчанию 0) и ключ `--bottom-left`, который начинает нумерацию с левого нижнего угла (по умолчанию – с левого верхнего). Справка: `rosrun aruco_pose genmap.py -h`.

Пример из документации: 8 маркеров по 0.33 м, 2 в ширину и 4 в высоту, шаг 1 м, нумерация с 0:

```
rosrun aruco_pose genmap.py 0.33 2 4 1 1 0 -o test_map.txt
```

Результат (его легко проверить вручную):

```
0    0.33    0    3    0    0    0    0
1    0.33    1    3    0    0    0    0
2    0.33    0    2    0    0    0    0
3    0.33    1    2    0    0    0    0
4    0.33    0    1    0    0    0    0
5    0.33    1    1    0    0    0    0
6    0.33    0    0    0    0    0    0
7    0.33    1    0    0    0    0    0
```

Маркер 0 расположен в левом верхнем углу сетки, номера идут вправо и затем вниз.

Если сетка нерегулярная, файл пишут вручную по формату выше. Перед использованием файл проверяют офлайн программой `check_map.py` (она не требует ROS):

```
python3 check_map.py lab_map.txt
```

Программа проверяет: по 8 полей в строке, повторы номеров, номера в диапазоне 0–249, положительные размеры, перекрытие маркеров – и печатает сводку (число маркеров, границы поля). При ошибках выводит строки `PROBLEM` и завершается с ненулевым кодом.

• Задание

Составьте команду `genmap.py` для карты из 12 маркеров (3 столбца, 4 ряда) размером 0.20 м с шагом 0.8 м по обеим осям, нумерацией с 20. Нарисуйте результат. **Проверка:** `rosrun aruco_pose genmap.py 0.2 3 4 0.8 0.8 20 -o m12.txt`; маркер 20 – слева сверху в точке $(0; 2.4)$, маркер 31 – справа снизу в точке $(1.6; 0)$.

20.6 Как «зашить» карту в дрон

Важно понимать, *где* хранится карта. Это **не** память полётного контроллера: карту читает нода `aruco_map` на бортовом компьютере Raspberry Pi из текстового файла на SD-карте. Автопилот получает от неё только готовые положения (если включён `aruco_vpe`). Поэтому «загрузить карту в дрон» означает три действия: положить файл на Raspberry Pi, указать его в `aruco.launch` и перезапустить службу.

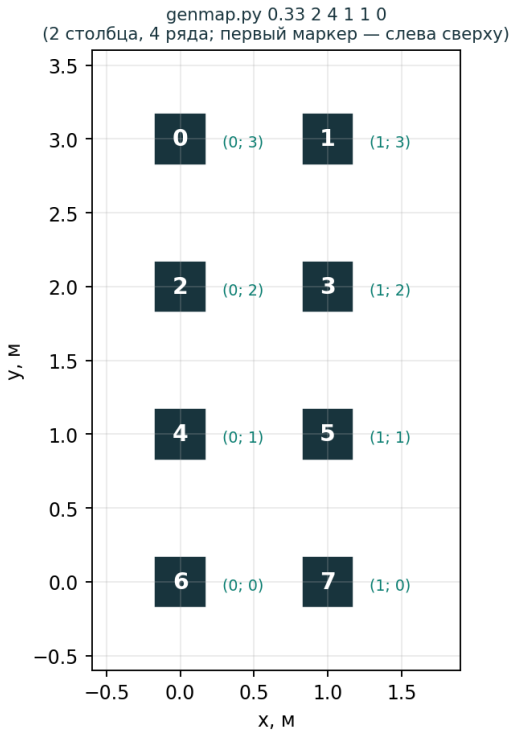


Рис. 20.4. Карта, созданная командой `genmap.py 0.33 2 4 1 1 0`: номера и координаты центров маркеров. Рисунок построен по формулам исходного кода утилиты.

1. **Скопировать файл карты.** Если карта создана на ноутбуке, её копируют на дрон командой `scp` (копирование по ssh) из терминала ноутбука при подключении к Wi-Fi дрона; если карту сгенерировал `genmap.py` на самом дроне, этот шаг не нужен:

```
scp my_map.txt pi@192.168.11.1:~/catkin_ws/src/clover/aruco_pose/map/
```

2. **Указать файл карты и включить подсистему.** В `clover.launch` установить `aruco` равным `true`; в `aruco.launch` – `aruco_detect` и `aruco_map` равными `true`, `map` равным `my_map.txt`, `placement` (`floor` или `ceiling`), `length` (размер маркеров, не вошедших в карту). Для удержания позиции автопилотом по маркерам – `aruco_vpe` равным `true`. Порядок правки – как в главе 9:

```
cd ~/catkin_ws/src/clover/clover/launch
nano aruco.launch
```

Нужные строки после правки:

```
<arg name="aruco_detect" default="true"/>
<arg name="aruco_map" default="true"/>
<arg name="aruco_vpe" default="true"/>
<arg name="placement" default="floor"/>
<arg name="length" default="0.33"/>
<arg name="map" default="my_map.txt"/>
```

3. **Перезапустить службу и проверить.**

```
sudo systemctl restart clover
rosrun clover selfcheck.py
```

Для проверки карты откройте в браузере отладочные изображения (адреса даны для стандартного образа): http://192.168.11.1:8080/stream_viewer?topic=/aruco_map/debug – кадр с найденными маркерами и осями карты; http://192.168.11.1:8080/snapshot?topic=/aruco_map/image – изображение карты, по которой дрон сейчас ориентируется. Фрейм `aruco_map` появляется, когда камера видит достаточно маркеров карты; проверка в терминале: `rosrun tf tf_echo map aruco_map`.

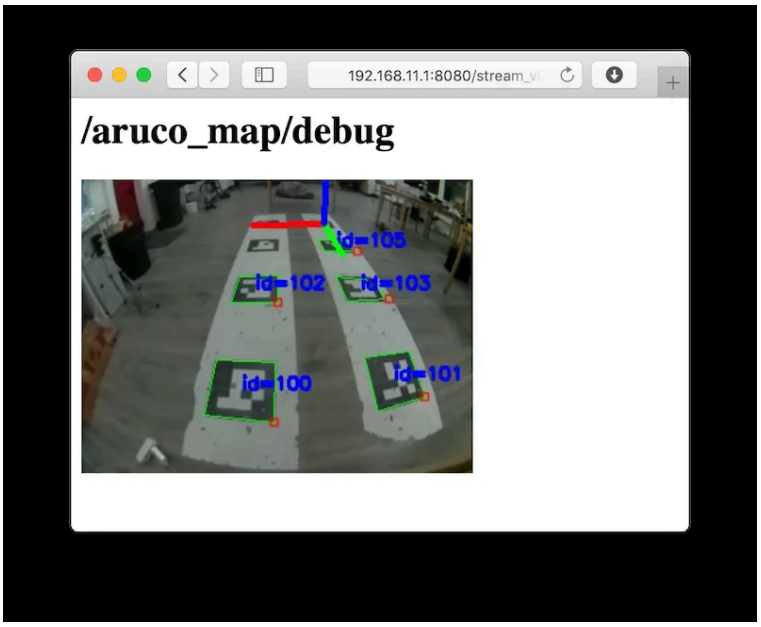


Рис. 20.5. Отладочное изображение карты маркеров `/aruco_map/debug`: найденные маркеры подписаны номерами, по ним вычислено положение камеры. COEX, CC BY-NC-SA 4.0.

Настройки автопилота для VPE

Чтобы автопилот использовал положение по маркерам, в PX4 (через QGroundControl, раздел параметров) должны быть включены соответствующие параметры. По документации COEX при оценщике EKF2: в `EKF2_AID_MASK` включены `vision position fusion` и `vision yaw fusion`, `EKF2_EVA_NOISE`=0.1 рад, `EKF2_EVP_NOISE`=0.1 м, `EKF2_EV_DELAY`=0; при оценщике LPE – `LPE_FUSION` с флажками `vision position` и `land detector`, `ATT_EXT_HDG_M`=1, `LPE_VIS_XY`=`LPE_VIS_Z`=0.1. Для специальной сборки PX4 для Клевера нужные параметры применяются автоматически. Названия и смысл параметров меняются между версиями PX4: сверяйте с документацией своей версии и проверяйте итог командой `selfcheck.py`.

Практика. Загрузка собственной карты

Создайте карту 2×2 (маркеры 0–3, размер 0.20 м, шаг 1 м) файлом `lab_map.txt`, проверьте `check_map.py`, поместите в каталог карт (в симуляторе – командой из следующего раздела), включите флаги, перезапустите и добейтесь появления фрейма `aruco_map`. **Проверка:** `tf_echo map aruco_map` печатает положение; на отладочном изображении подписаны номера всех видимых маркеров.

20.7 Карта в симуляторе: согласованное поле

В Gazebo нужно, чтобы **физическое поле** (модели маркеров в мире) совпадало с **картой**, которую читает `aruco_map`. Смена одного файла без другого создаёт противоречивую геометрию. Порядок:

1. Скопируйте `examples/lab_map.txt` в каталог карт пакета `aruco_pose`.
2. Создайте мир из штатного `clover.world` генератором `aruco_gen` (программа из пакета Clover, которая по файлу карты добавляет модели маркеров в файл мира).
3. В `aruco.launch` (после резервной копии) задайте аргумент `map` равным `lab_map.txt`. Это имя файла для ноды распознавания, а не имя мира.
4. Перезапустите симулятор с новым миром и проверьте отладочное изображение и TF.

```
cp "$HOME/clover_course_files/examples/lab_map.txt" \
  "$(rospack find aruco_pose)/map/lab_map.txt"
rosrun clover_simulation aruco_gen --single-model \
  --source-world="$(rospack find clover_simulation)/resources/worlds/clover.world" \
  "$(rospack find aruco_pose)/map/lab_map.txt" \
  > "$HOME/clover_worlds/lab_aruco.world"
roslaunch clover_course course.launch \
  course_world:="$HOME/clover_worlds/lab_aruco.world"
```

Каталог `clover_worlds` создаётся в главе 18. После генерации убедитесь, что файл мира не пуст и команда завершилась без ошибки. Если ваш генератор имеет другой интерфейс, изучите `rosrun clover_simulation aruco_gen --help`. Для возврата к штатному миру верните штатную карту.

• Задание

В файле карты сдвиньте один маркер на 0.5 м по X. Что нужно изменить в Gazebo, чтобы данные согласовались? **Проверка:** сдвинуть модель этого маркера в мире на те же 0.5 м (или заново сгенерировать мир).

20.8 Полёты по маркерам

Над отдельным маркером. Фрейм `aruco_N` связан с маркером номер N; цель в нём задаётся смещением от центра маркера. Полёт на высоту 1 м над маркером 5:

```
navigate(frame_id='aruco_5', x=0, y=0, z=1)
```

Если маркер не виден дольше примерно полусекунды, дрон продолжает выполнять предыдущую команду. Обращение к `get_telemetry(frame_id='aruco_3')` при невидимом маркере возвращает `NaN` в координатах – такой результат обязательно проверяют (глава 6).

По координатам карты. После взлёта, когда камера увидела карту, можно задавать точки поля во фрейме `aruco_map`. Если карта задана, можно также лететь к маркеру через фрейм `aruco_N`, даже когда он сейчас не виден: положение известно по карте. Маршрут по углам нашей учебной карты:

```
#!/usr/bin/env python3
import math
import time
import rospy
from flight_tools import Flight

rospy.init_node('course_aruco_route')
flight = Flight()
with flight.mission():
    flight.go(z=1.2, frame='body', auto_arm=True)
    deadline = time.monotonic() + 10
    while True:
        try:
            flight.check('aruco_map')
            break
        except (RuntimeError, rospy.ServiceException):
            if time.monotonic() >= deadline:
                raise TimeoutError('ArUco map unavailable')
            time.sleep(0.1)
    # The supplied lab map has marker centres at these coordinates.
    for x, y in [(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]:
        flight.go(x=x, y=y, z=1.2, frame='aruco_map')
```

Программа взлетает во фрейме `body`, ограниченное время ждёт появления `aruco_map`, затем обходит квадрат. Высота 1.2 м отсчитывается от плоскости карты на полу; координаты рассчитаны на наш `lab_map.txt`.

Обход маркеров по списку. Типовая задача соревнований: посетить маркеры в заданном порядке и зависнуть над каждым:

```
#!/usr/bin/env python3
"""Visit ArUco markers in the given order: hover above each, then land.

Run with the marker map active (aruco, aruco_map, aruco_vpe flags on).
Parameters:  _ids:="[1,2,4]"  _height:=1.2  _hold:=2.0
For a real aircraft pass _real:=true; a pilot with a transmitter must be ready.
"""
import math
import time
import rospy
from flight_tools import Flight

rospy.init_node('course_aruco_tour')
ids = [int(i) for i in rospy.get_param('~ids', [1, 2, 4])]
height = float(rospy.get_param('~height', 1.2))
hold = float(rospy.get_param('~hold', 2.0))
real = bool(rospy.get_param('~real', False))
if not ids or not math.isfinite(height) or not 0.5 <= height <= 3.0:
    raise ValueError('ids must be non-empty and height within 0.5..3.0 m')

flight = Flight(simulation_only=not real)
with flight.mission():
    flight.go(z=height, frame='body', auto_arm=True)
    time.sleep(2)            # let the camera see enough of the map
    for marker_id in ids:
        frame = f'aruco_{marker_id}'
```

```
        flight.go(x=0, y=0, z=height, frame=frame, timeout=40)
        rospy.loginfo('Above marker %d', marker_id)
        time.sleep(hold)
```

Номера и высота задаются параметрами запуска, например: `python3 aruco_tour.py _ids:="[1,2,4]" _height:=1.2 _hold:=2.0`.

Точная посадка на маркер. Чтобы сесть на маркер, дрон подлетает над ним, снижается ступенями и каждый раз проверяет, что маркер виден:

```
#!/usr/bin/env python3
"""Precision landing on a marker: go above it, descend in steps, land.

Parameters:  _id:=3  _real:=false
Each step checks that the marker frame is still available (not NaN).
"""
import math
import rospy
from flight_tools import Flight

rospy.init_node('course_aruco_landing')
marker_id = int(rospy.get_param('~id', 3))
real = bool(rospy.get_param('~real', False))
frame = f'aruco_{marker_id}'
steps = [1.0, 0.7, 0.4]          # metres above the marker; the last step is before landing

flight = Flight(simulation_only=not real)
with flight.mission():
    flight.go(z=steps[0], frame='body', auto_arm=True)
    for z in steps:
        state = flight.telemetry(frame_id=frame)
        if not all(math.isfinite(v) for v in (state.x, state.y, state.z)):
            raise RuntimeError(f'Marker {marker_id} is not visible; aborting')
        flight.go(x=0, y=0, z=z, frame=frame, speed=0.3, timeout=40)
    # leaving the with-block lands the aircraft
```

Если маркер потерян (координаты `NaN`), программа прекращает снижение и контекст миссии выполняет посадку на месте. Для реального аппарата передают `_real:=true`: без этого помощник откажется работать вне симулятора (глава 25).

Маркеры на потолке. Камера должна смотреть вверх, в `aruco.launch` указывают `placement=ceiling`, а оптический поток выключают (`optical_flow=false`), потому что он не работает при таком положении камеры. Фрейм карты оказывается перевёрнутым, поэтому высота в `aruco_map` отсчитывается вниз от потолка.

• Задание

Запишите вызовы `navigate`: а) на 1.5 м над маркером 7; б) в точку $(2; 1)$ поля на высоте 1.2 м; в) на метр левее маркера 3 на высоте 1 м. **Проверка:** а) `frame_id='aruco_7', x=0, y=0, z=1.5`; б) `frame_id='aruco_map', x=2, y=1, z=1.2`; в) `frame_id='aruco_3', x=-1, y=0, z=1` (ось X маркера направлена вправо, поэтому «левее» – отрицательное x).

20.9 Если маркеры не распознаются или позиция «плывёт»

Проверяйте по слоям: от камеры к автопилоту.

Симптом	Возможная причина	Проверка
Нет топика `aruco_detect/markers`, нет фреймов	Флаг `aruco` выключен или служба не перезапущена	`rosnode list`, значение флага, `systemctl status clover`
Маркеры на отладочном кадре не отмечены	Другой словарь или номер вне 0–249; маркер слишком мал в кадре; блики; смаз при движении	Печать из `DICT_4X4_250`; увеличить маркер или снизить высоту; матовая бумага; сначала висение
Маркеры найдены, но расстояния неверны	Размер в `length` или карте не равен реальному; камера не откалибрована	Измерить сторону линейкой, поправить `length`; проверить `camera_info`
Фрейма `aruco_map` нет	Не включён `aruco_map`; маркеры карты не попадают в кадр; неверный файл карты	Флаги, `check_map.py`, высота полёта, отладочное изображение
Дрон летит «не туда» относительно поля	Карта не совпадает с реальной расстановкой; неверное направление осей; не тот `placement`	Сверить рулеткой с картой; правило осей из главы 10
Позиция скачет, дрон «качается» по маркерам	Плохое освещение, редкие маркеры, плохая калибровка, неверные параметры VPE, вибрации	Больше маркеров в кадре; параметры PX4; вибрации (глава 25)
Всё работает, но координаты не обновляются	Устаревшие данные: маркер вышел из кадра, а TF остался в буфере	Проверять метку времени и свежесть (глава 24)

Практика. Проверка согласованности карты

Поднимитесь над первым маркером, сравните координаты в `map` и `aruco_map`. Затем выполните маршрут и повторите при другой высоте. **Проверка:** номера и геометрия на отладочном кадре совпадают с файлом карты; маршрут проходит над ожидаемыми точками. Отчёт содержит карту, файл мира и настройки распознавания.

Задание в формате соревнований: «Обход маркеров»

Поле: карта 2×4 из команды `genmap.py 0.33 2 4 1 1 0`. Задача: взлететь, пролететь над маркерами 0, 3, 6, 5 в этом порядке на высоте 1.2 м, зависнуть над каждым 2 с и сесть на маркер 7. Время ограничено: 3 минуты. Критерии: каждая посещённая точка – 10 баллов (ошибка по горизонтали не более 0.2 м), посадка в круге радиусом 0.2 м от центра маркера – 20 баллов, нарушение порядка – минус 10, касание препятствий – минус 20. **Проверка:** программа не содержит абсолютных координат (только фреймы `aruco_N`), завершается посадкой, повторяется три раза подряд. Расширенный вариант: порядок маркеров считывается из QR-кода (глава 27).

Источник: COEX, навигация по картам ArUco. Проверено 11.10.2026.

Источник: COEX, распознавание ArUco-маркеров. Проверено 11.10.2026.

Итоги и самопроверка

- ArUco-маркер несёт номер и позволяет вычислить положение камеры относительно маркера; карта задаёт координаты нескольких маркеров в системе поля.
- Маркеры печатают из словаря DICT_4X4_250 без масштабирования, измеряют сторону и записывают её в карту.
- Карту создаёт `genmap.py` или пишут вручную (8 полей: `id length x y z rot_z rot_y rot_x`); она хранится на Raspberry Pi, а не в полётном контроллере.
- Чтобы подсистема работала, нужны флаги `aruco`, `aruco_detect`, `aruco_map` (и `aruco_vpe` для удержания позиции автопилотом) и перезапуск службы.
- Если маркер не виден, фрейм даёт NaN: это нужно проверять перед использованием.

Проверьте себя: 1) Где хранится карта маркеров? 2) Что означают столбцы `length`, `x`, `y`? 3) Что произойдёт при вызове `get_telemetry` для невидимого маркера? 4) Какой флаг отвечает за передачу позиции автопилоту?
Ответы: 1) в текстовом файле на Raspberry Pi (каталог `aruco_pose/map`); 2) сторона чёрного квадрата в метрах и координаты центра маркера в системе поля; 3) координаты вернутся как NaN; 4) `aruco_vpe`.

ГЛАВА 21

Компьютерное зрение: от кадра к решению

Цели главы

- отделить вопрос «что видно» от «где в пространстве»;
- найти красную область и её центр;
- понять, почему ошибка в пикселях – не метры.

Нужно знать заранее: главы 7 (топики и сообщения), 6 (Python) и 10 (системы координат). Понятия «пиксель», «HSV», «маска» вводятся в начале этой главы; подробный разбор кадра – в главе 22.

21.1 Изображение как набор чисел

Цифровое изображение – прямоугольная таблица из **пикселей** (точек изображения). Размер кадра задают ширина W и высота H в пикселях, например 640×480. Положение пикселя определяют два числа: номер столбца u (слева направо) и номер строки v (*сверху вниз*); начало отсчёта $(0, 0)$ – левый верхний угол. Каждый пиксель хранит яркости нескольких *каналов*: в цветном кадре обычно три значения от 0 до 255. В библиотеке OpenCV (программная библиотека обработки изображений и компьютерного зрения) порядок каналов – B, G, R (синий, зелёный, красный), поэтому чисто красный пиксель записывается как $(0, 0, 255)$.

Для поиска цвета удобнее другое описание цвета – **HSV**: H (Hue, «оттенок», положение на цветовом круге), S (Saturation, «насыщенность»: от серого до яркого цвета) и V (Value, «яркость»). Красный, зелёный и синий отличаются прежде всего оттенком H, а серый, белый и чёрный – малой насыщенностью или яркостью. **Маска** – изображение, в котором пиксели, прошедшие проверку (например, «оттенок красный и достаточно насыщенный»), отмечены белым, остальные – чёрным. Если по белым пикселям вычислить их среднее положение (физики назвали бы это центром масс), получится центр найденного объекта. Программа, которая выполняет такой поиск, называется **детектором**.

21.2 Сначала детектор, потом управление

В работе со зрением полезно разделить три вопроса. **Что видно?** Это задача детектора. **Где объект в пространстве?** Это задача геометрии и калибровки. **Как лететь к нему?** Это задача контроллера (программы, вычисляющей управляющие команды по ошибке, глава 17). Если объединить всё в один длинный callback, сложнее найти причину ошибки и проконтролировать задержку.

Для первого опыта разместим на полу красную площадку и найдём её центр на изображении. В проекте есть `red_pad.sdf`. Он содержит только visual: это цветная учебная разметка, а не физическая платформа с проверенной прочностью.

```
rosrun gazebo_ros spawn_model -sdf \
  -file "$HOME/clover_course_files/examples/red_pad.sdf" \
  -model course_red_pad -x 0.5 -y 0 -z 0.02
```

Точка расположена в **системе мира Gazebo**; она не обязана точно совпадать с `map`. Площадка не должна перекрывать все ArUco-маркеры или лишать текстуры пол, по которому работает оптический поток (определение смещения по изображению пола, глава 18). Если в кадре её нет, сначала измените расположение объекта и подтвердите ориентацию камеры.

• Задание

Для задачи «красная площадка» назовите, какая часть системы отвечает на вопросы «что видно?», «где в пространстве?», «как лететь?». **Проверка:** детектор; геометрия с калибровкой; контроллер полёта.

21.3 Изображение ROS и OpenCV

Кадр в ROS – сообщение типа `sensor_msgs/Image` (глава 7) с размером, кодировкой, данными и временной меткой (timestamp – момент съёмки). `CvBridge` – библиотека, которая преобразует его в массив OpenCV (Python-таблицу чисел, с которой работают функции библиотеки). Кодировка `bgr8` означает три 8-битных канала B, G, R. В HSV цвет выделять удобнее, чем жёстким порогом красного канала: яркость и насыщенность отделены от цветового тона.

В 8-битном HSV OpenCV H лежит в диапазоне 0–179. Красный находится возле начала и конца шкалы, поэтому используем две маски. Пороги S и V отсекают серые и очень тёмные области. Эти значения – стартовая настройка, которую надо проверять на вашей сцене.

• Задание

Сколько байт занимает кадр 640×480 в формате bgr8? **Проверка:** $640 \cdot 480 \cdot 3 = 921600$ байт.

21.4 Полный детектор красной области

```
#!/usr/bin/env python3
"""Publish a red-object centroid; this example does not command flight."""
import cv2
import numpy as np
import rospy
from clover import long_callback
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from std_msgs.msg import Float32MultiArray

rospy.init_node('course_red_detector')
bridge = CvBridge()
debug = rospy.Publisher('~debug', Image, queue_size=1)
target = rospy.Publisher('~target', Float32MultiArray, queue_size=1)

@long_callback
def on_image(message):
    try:
        frame = bridge.imgmsg_to_cv2(message, 'bgr8')
```

```
    except CvBridgeError as error:
        rospy.logwarn_throttle(2, str(error))
        return
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, np.array([0, 100, 80]),
                       np.array([10, 255, 255]))
    mask |= cv2.inRange(hsv, np.array([170, 100, 80]),
                        np.array([179, 255, 255]))
    moments = cv2.moments(mask, binaryImage=True)
    area = moments['m00']
    h, w = mask.shape
    if area >= 150:
        u = moments['m10'] / area
        v = moments['m01'] / area
        ex, ey = (u - w / 2) / (w / 2), (v - h / 2) / (h / 2)
        # stamp, normalized error x/y, area fraction
        values = [message.header.stamp.to_sec(), ex, ey, area/(w*h)]
        cv2.circle(frame, (int(u), int(v)), 8, (0, 255, 0), 2)
    else:
        values = [message.header.stamp.to_sec(), float('nan'),
                  float('nan'), 0.0]
    target.publish(Float32MultiArray(data=values))
    output = bridge.cv2_to_imgmsg(frame, 'bgr8')
    output.header = message.header
    debug.publish(output)

rospy.Subscriber('main_camera/image_raw_throttled', Image, on_image,
                 queue_size=1, buff_size=2**24)
rospy.spin()
```

Источник: COEX, подписка на камеру, CvBridge и long_callback. Проверено 11.10.2026.

Запустите `python3 color_detector.py`. В rqt выберите `/course_red_detector/debug`; центр найденной области будет отмечен зелёным. Топик `/course_red_detector/target` содержит время кадра, нормированные ошибки по X и Y изображения и долю красной области. Если объект не найден, ошибки равны `NaN`, а доля – нулю.

```
rostopic echo /course_red_detector/target
rosrun rqt_image_view rqt_image_view
```

Посмотрите на рисунок слева направо. Синий квадрат и зелёный круг – это «помехи»: они не красные, поэтому маска их не включила. Площадь красной области 3721 пиксель, центр $(220;\ 120)$ – именно эти числа программа публикует в топик `target`.

• Задание

Запустите детектор и откройте его debug-изображение. **Проверка:** на красной площадке виден зелёный круг центра.

21.5 Как устроено измерение

Центр маски вычисляется по *моментам* – суммам, которые считает OpenCV по белым пикселям (m_{00} – число белых пикселей, то есть площадь; m_{10} и m_{01} – суммы их координат u и

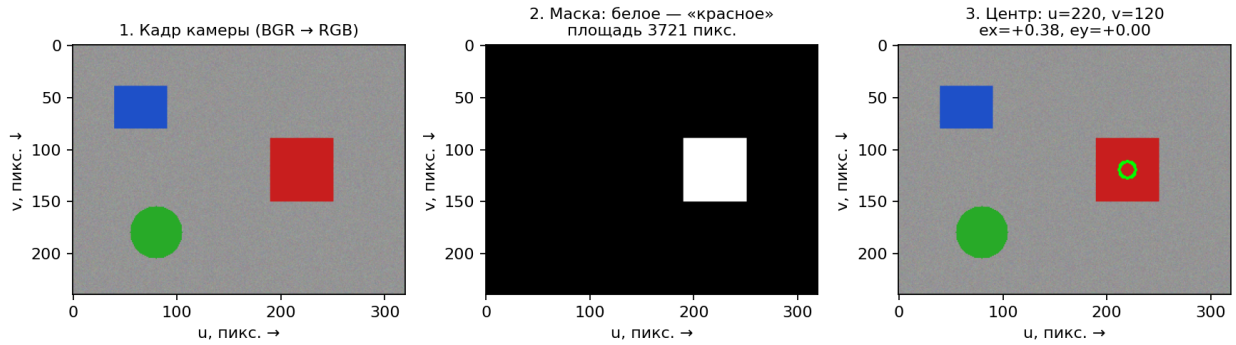


Рис. 21.1. Три шага детектора на учебном кадре 320×240: исходный кадр с красной площадкой и двумя «помехами», маска порога (белое – пиксели, прошедшие проверку) и найденный центр. Рисунок получен запуском тех же порогов HSV, что и в программе `color_detector.py`. Кадр искусственный, не снимок Gazebo.

v):

$$u_c = \frac{m_{10}}{m_{00}}, \qquad v_c = \frac{m_{01}}{m_{00}}.$$

Нормированные ошибки относительно центра кадра:

$$e_x = \frac{u_c - W/2}{W/2}, \qquad e_y = \frac{v_c - H/2}{H/2}.$$

Числа приблизительно лежат в пределах $[-1, 1]$. Они **безразмерны**. Это ещё не смещение в метрах. Порог площади 150 пикселей подавляет мелкий шум, но при другой высоте и разрешении его нужно пересмотреть.

Если в кадре два красных объекта, центр объединённой маски может попасть между ними. Следующий уровень – найти контуры, выбрать крупнейший объект, проверить форму и отсеять неверные цели. Упражнение особенно полезно для понимания того, что «цвет найден» и «нужная площадка найдена» – разные условия.

• Задание

Вычислите e_x и e_y для центра $(u, v) = (480, 120)$ в кадре 640×480. **Проверка:** $e_x = 0.5$, $e_y = -0.5$.

21.6 Задержка кадра и команды полёта

В примере используется `image_raw_throttled`, короткая очередь и `long_callback`, рекомендованный COEX для длительной обработки. Это уменьшает риск обработки большого количества устаревших кадров. Проверяйте реальную задержку как разность текущего ROS-времени и timestamp кадра; публикация с высокой частотой сама по себе не гарантирует свежесть.

Для контроллера наведения нужны отдельные ограничения: максимальная скорость, минимальная площадь, интервал свежести наблюдения и обработка пропажи цели. Сначала экспериментально определите знак команд: при маленьком движении вперёд куда сдвигается объект на кадре? Так проверяют установленную геометрию камеры. Не копируйте формулу `vx=-k*ex` без этого опыта.

Учебная логика наведения

Получить свежий кадр → проверить цель → преобразовать ошибку в нужный фрейм → ограничить скорость → отправить команду. При устаревшем наблюдении перейти к заранее определённому состоянию: удержание по достоверной локальной позиции, затем посадка при невозможности продолжения. Удержание требует рабочего позиционирования и не гарантируется при потере всех источников позиции.

При использовании `set_velocity` после исчезновения цели нужно явно изменить прежнюю команду: последняя скорость не становится автоматически нулевой. Центрирование по двум пиксельным координатам не даёт надёжную высоту над площадкой; для снижения используйте отдельное измерение расстояния и условия завершения.

Практика. Красная площадка

Разместите площадку в трёх положениях. Запишите ошибки и debug-кадры, проверьте смену знаков. Добавьте второй красный объект и покажите ошибку центра общей маски. Успех: публикация отсутствующей цели явно отличается от нулевой ошибки найденной цели; код ещё не управляет полётом.

Итоги и самопроверка

- Маска отмечает подходящие пиксели; центр считают по моментам.
- Красный в HSV лежит на обоих краях шкалы H – нужны две маски.
- Нормированные ошибки e_x, e_y безразмерны.

Проверьте себя: 1) Зачем порог площади? 2) Почему нельзя сразу превратить e_x в метры?
Ответы: 1) подавляет мелкий шум; 2) нужны высота, калибровка и геометрия установки камеры.

ГЛАВА 22

Камера: фотография, пиксели и качество детектора

Цели главы

- сохранить кадр и понять, из чего он состоит;
- построить пороговую маску;
- оценить качество детектора на нескольких сценах.

Нужно знать заранее: главы 21.

22.1 Первый результат – сохранённый кадр

Изображение полезно даже без распознавания: оно показывает, куда направлена камера и какие объекты попали в обзор. До настройки HSV сохраните один кадр и проверьте его глазами. Файл `take_photo.py` читает следующее сообщение камеры, конвертирует его и сохраняет PNG.

```
#!/usr/bin/env python3
"""Read one camera frame and save it; no flight commands."""
import argparse
import cv2
import rospy
from cv_bridge import CvBridge
from sensor_msgs.msg import Image

parser = argparse.ArgumentParser()
parser.add_argument('--output', default='frame.png')
args = parser.parse_args(rospy.myargv()[1:])
rospy.init_node('course_photo')
message = rospy.wait_for_message('/main_camera/image_raw_throttled',
                                 Image, timeout=10)
frame = CvBridge().imgmsg_to_cv2(message, desired_encoding='bgr8')
if not cv2.imwrite(args.output, frame):
    raise RuntimeError('Could not write image')
print('Saved:', args.output, 'pixels:', frame.shape[:2],
      'stamp:', message.header.stamp.to_sec())
```

```
python3 take_photo.py --output pad_01.png
```

Команда выполняется при работающей камере в Gazebo. Сохранение не управляет дроном; фотографировать можно со стартовой точки, если площадка видна. При желании снимка с высоты используйте отдельную короткую миссию и наблюдателя, не добавляя второй источник управления.

`wait_for_message` получает один следующий кадр и имеет таймаут. `CvBridge` переводит ROS-формат в массив. `cv2.imwrite` возвращает признак успеха; программа его проверяет. PNG сохраняется в текущий каталог, если вы не передали другой путь.

• Задание

Сохраните кадр программой `take_photo.py` и откройте файл. **Проверка:** файл открывается, на нём видна сцена.

22.2 Как устроены пиксели

Пиксель – элемент изображения с числовыми значениями каналов. Изображение шириной 640 и высотой 480 содержит 307200 пикселей. В bgr8 у каждого три канала от 0 до 255. В памяти OpenCV первый индекс выбирает строку, второй – столбец: `frame[v, u]`. Координата u растёт вправо, v вниз.

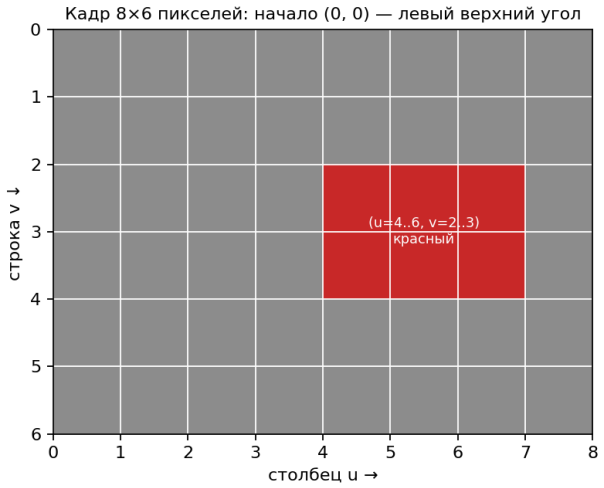


Рис. 22.1. Кадр 8×6 пикселей. Пиксель задаётся двумя числами: номером столбца u (слева направо) и номером строки v (*сверху вниз*). Начало отсчёта – левый верхний угол, поэтому ось v идёт вниз.

Если перепутать ширину и высоту, центр и нормированные ошибки будут неверными. `frame.shape[:2]` возвращает сначала высоту, затем ширину. Цвет $(0, 0, 255)$ в BGR – красный, а не синий. В других библиотеках может использоваться RGB, поэтому всегда проверяйте соглашение.

• Задание

Какой пиксель находится в центре кадра 640×480? **Проверка:** $u = 320$, $v = 240$.

22.3 Офлайн-опыт: размер и средний цвет

```
#!/usr/bin/env python3
"""Offline image statistics, OpenCV required."""
import argparse
import cv2

parser = argparse.ArgumentParser()
parser.add_argument('image')
args = parser.parse_args()
frame = cv2.imread(args.image)
if frame is None:
    raise ValueError('Image cannot be read: ' + args.image)
height, width = frame.shape[:2]
blue, green, red, _ = cv2.mean(frame)
print('Width:', width, 'height:', height)
print(f'Mean BGR: {blue:.1f}, {green:.1f}, {red:.1f}')
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
if not cv2.imwrite('gray.png', gray):
    raise RuntimeError('Could not save gray.png')
```

```
python3 image_stats.py pad_01.png
```

Программа требует OpenCV, но не требует ROS. Она печатает размеры и средние каналы, затем создаёт `gray.png`. Средний цвет всего изображения не говорит, где площадка: маленький красный объект может почти не изменить среднее. Это помогает понять, зачем нужен поиск отдельных областей.

Пиксели и данные

Сохраните кадры с площадкой и без неё. Сравните размер и средний красный канал. Объясните, почему одного среднего недостаточно для координаты цели. Дополнительное задание: вычислите средний цвет центральной четверти кадра и сравните с полным изображением.

22.4 Пороговая маска

Маска – изображение, где подходящие пиксели отмечены, а остальные выключены. В нашем детекторе HSV-пороги выделяют достаточно насыщенный и яркий красный цвет. Маска превращает вопрос «какой цвет?» в набор точек, по которым можно вычислить площадь и центр.

Центр объединённой маски – среднее положение всех выбранных пикселей. Если слева и справа есть два красных пятна, центр может оказаться на пустом месте между ними. Алгоритм при этом выполнил свою формулу правильно, но формула не соответствует задаче «найти конкретную площадку».

• Задание

Пройдёт ли тёмно-красный пиксель с насыщенностью 50 порог $S \geq 100$? **Проверка:** нет: пиксель слишком «серый».

22.5 Данные для проверки, а не один удачный кадр

Создайте небольшой набор: площадка в центре, у края, далеко, частично закрыта, два красных объекта и кадр без цели. Сначала определите, какой объект считается правильной целью, затем проверяйте детектор на всех кадрах. Не подбирайте порог только под один красивый пример.

Сцена	Что ожидаем	Возможная проблема
Одна площадка	Центр внутри объекта	Порог слишком строгий
Объект у края	Ненулевая ошибка правильного знака	Перепутаны оси
Два объекта	Выбор нужной цели по правилу	Центр маски между объектами
Нет площадки	Явное отсутствие измерения	Ложное красное пятно
Маленькая цель	Решение по минимальной площади	Порог не подходит высоте

• Задание

Составьте набор из пяти сцен для проверки детектора. **Проверка:** есть сцены с площадкой и без, с помехами и разным освещением.

22.6 Как оценить результат

Для начала вручную отметьте центр нужной площадки в нескольких сохранённых изображениях. Сравните с найденным центром. Ошибка в пикселях:

$$e_{px} = \sqrt{(u_{found} - u_{true})^2 + (v_{found} - v_{true})^2}.$$

Отдельно считайте случаи, когда цель была, но алгоритм её пропустил, и случаи ложного обнаружения. Их нельзя оценить только средним расстоянием: в пропущенном кадре корректного найденного центра вообще нет.

При сравнении разных разрешений нормированная ошибка удобнее пиксельной. Но нормирование не переводит результат в метры и не исправляет геометрию камеры. Одинаковые 20 пикселей при разных высотах могут соответствовать разным смещениям на полу.

• Задание

Из 10 кадров с площадкой найдено 8; из 10 без площадки ложно найдено 1. Найдите долю верных и долю ложных срабатываний. **Проверка:** 80% и 10%.

22.7 От камеры к управлению: что ещё потребуется

22.7.1 Ещё одна нода читает результат

Запустите `color_detector.py`, затем в другом терминале `target_watch.py`. Детектор менять не нужно: наблюдатель подключается к его уже существующему выходу. Это та же архитектура, что в главе 7.

```
#!/usr/bin/env python3
"""Observe the red detector result; never send flight commands."""
import math
import rospy
from std_msgs.msg import Float32MultiArray


def on_target(message):
    if len(message.data) != 4:
        rospy.logwarn('Expected stamp, ex, ey, area')
        return
    stamp, ex, ey, area = message.data
    age = rospy.get_time() - stamp
    if not math.isfinite(age) or not 0 <= age <= 0.5:
        rospy.logwarn('Observation timestamp unsuitable: age=%s', age)
        return
    if not all(math.isfinite(v) for v in (ex, ey, area)) or area <= 0:
        rospy.loginfo('Target MISSING')
        return
    state = 'CENTERED' if math.hypot(ex, ey) < 0.1 else 'OFFSET'
    rospy.loginfo('%s ex=%.3f ey=%.3f area=%.3f age=%.3f',
                  state, ex, ey, area, age)


rospy.init_node('course_target_watch')
rospy.Subscriber('/course_red_detector/target', Float32MultiArray,
                 on_target, queue_size=1)
rospy.spin()
```

Наблюдатель отличает найденную цель в центре, смещённую цель и отсутствие измерения. Порог 0.1 – безразмерный допуск нашего опыта. Проверка возраста 0.5 с относится к ROS-времени кадра. Она выявляет устаревшее поступившее сообщение, но не сообщает о полном прекращении потока: для этого нужен отдельный сторож последнего получения. На медленном стенде сначала измерьте задержку и обоснуйте порог.

22.7.2 План следующей ступени

Нужны проверенная ориентация камеры, калибровка, оценка расстояния, преобразование осей, ограничения скорости, свежесть наблюдения и реакция на потерю цели. Начните с детектора и журнала. Потом проверьте знак реакции маленькими движениями. Только после этого проектируйте замкнутый алгоритм.

При расширении системы полезно вывести явное сообщение результата вместо массива: stamp, found, error_x, error_y, area. В учебном `Float32MultiArray` эти поля договорены текстом, поэтому нужно помнить их порядок. Собственный тип ROS делает договор видимым, но потребует генерации сообщения через catkin. Это отдельный шаг расширения.

Мини-проект «качество зрения»

Соберите не менее шести сцен, сохраните кадры и таблицу результата. Улучшите одно правило детектора: например, выбирайте крупнейший контур вместо центра общей маски. Сравните исходный и изменённый алгоритм на одном наборе. Успех: вы можете показать и удачный случай, и оставшееся ограничение.

Источник: OpenCV, цветовые пространства и выделение цвета. Проверено 11.10.2026.

Итоги и самопроверка

- Кадр – массив пикселей; у каждого три канала.
- Качество оценивают на наборе кадров, а не на одном удачном.

Проверьте себя: 1) Сколько пикселей в кадре 640×480? 2) Что вернёт frame.shape[:2]?
Ответы: 1) 307200; 2) сначала высоту, затем ширину.

ГЛАВА 23

Телеметрия, логи и воспроизводимый опыт

Цели главы

- записывать результат опыта в виде данных;
- считать ошибки и сравнивать серии;
- записывать и воспроизводить поток rosbag.

Нужно знать заранее: главы 13.

23.1 Записывайте результат, а не впечатление

Фраза «летает нормально» не позволяет сравнить два алгоритма. До запуска выберите **метрику** (числовую величину, по которой сравнивают результаты) и измерительный интервал (отрезок времени, в котором её считают). Для квадрата это ошибка возвращения; для висения – ошибка высоты и разброс координат; для детектора – точность и задержка; для маршрута – время и столкновения.

В проекте есть простой логгер (программа, записывающая данные в файл) в формате CSV (текстовая таблица, значения в строке разделены запятыми). Он работает **в отдельном терминале параллельно миссии**: миссия – в окне B, логгер – в окне C (глава 4); логгер не меняет команды полёта. Запустите его до миссии, остановите Ctrl+C после завершения.

```
cd "$HOME/clover_course_files/examples"
python3 telemetry_logger.py --frame map --output square_01.csv
```

```
#!/usr/bin/env python3
"""Run in a separate terminal before a mission. Stop with Ctrl+C."""
import argparse
import csv
import time
import rospy
from clover import srv

parser = argparse.ArgumentParser()
parser.add_argument('--output', default='telemetry.csv')
parser.add_argument('--frame', default='map')
args = parser.parse_args(rospy.myargv()[1:])
rospy.init_node('course_logger')
rospy.wait_for_service('get_telemetry', timeout=15)
get_state = rospy.ServiceProxy('get_telemetry', srv.GetTelemetry)
fields = ['wall_time', 'sim_time', 'frame', 'connected', 'armed',
          'mode', 'x', 'y', 'z', 'vx', 'vy', 'vz', 'yaw']
with open(args.output, 'w', newline='', encoding='utf-8') as output:
    writer = csv.writer(output)
    writer.writerow(fields)
    while not rospy.is_shutdown():
```

```
        try:
            s = get_state(frame_id=args.frame)
            writer.writerow([time.monotonic(), rospy.get_time(), args.frame,
                             s.connected, s.armed, s.mode, s.x, s.y, s.z,
                             s.vx, s.vy, s.vz, s.yaw])
            output.flush()
        except rospy.ServiceException as error:
            rospy.logerr(str(error))
            break
        time.sleep(0.1)
```

Одна строка содержит время компьютера, время симуляции, фрейм, признак связи, armed (моторы включены), режим автопилота, координаты и скорость. Частота примерно 10 Гц по реальному времени. При RTF< 1 это больше выборок на симуляционную секунду. Синхронный сервис телеметрии добавляет нагрузку; для исследований с высокой частотой лучше записывать соответствующие топики.

• Задание

Придумайте числовую метрику для задачи «вернуться к старту»: что измерять и в каких единицах. **Проверка:** метрика числовая, с единицей, ясно, как её получить из журнала.

23.2 Какие ошибки считать

Ошибка – расстояние от измеренного положения до цели. Для целевой точки p^* и измерений p_i ($i = 1, \dots, N$):

$$e_i = \|p_i - p^*\|_2, \qquad \text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2}.$$

Здесь $\|\cdot\|_2$ – обычная длина вектора (корень из суммы квадратов компонент), а RMSE (root mean square error, «среднеквадратичная ошибка») – корень из среднего значения квадратов ошибок: единая числовая оценка разброса. Для высоты используют $e_i = z_i - z^*$. В интервал оценки висения не следует включать сам взлёт, посадку и некорректные значения. Выберите устойчивый участок, например с 8-й до 15-й симуляционной секунды, и запишите выбранные границы в отчёт.

Код анализа использует только стандартную библиотеку Python и подходит для обработки CSV вне ROS:

```
python3 analyze_log.py square_01.csv --height 1.0 --start 8 --end 15
```

```
#!/usr/bin/env python3
"""Offline height RMSE for armed OFFBOARD samples; standard library only."""
import argparse
import csv
import math

parser = argparse.ArgumentParser()
parser.add_argument('csv_file')
parser.add_argument('--height', required=True, type=float)
parser.add_argument('--start', type=float, default=0,
                    help='seconds from first logged simulation sample')
```

```
parser.add_argument('--end', type=float, default=float('inf'))
args = parser.parse_args()
with open(args.csv_file, encoding='utf-8') as source:
    rows = list(csv.DictReader(source))
if not rows:
    raise SystemExit('No samples')
t0 = float(rows[0]['sim_time'])
errors = []
for row in rows:
    elapsed = float(row['sim_time']) - t0
    z = float(row['z'])
    if (args.start <= elapsed <= args.end and math.isfinite(z)
            and row['connected'] == 'True' and row['armed'] == 'True'
            and row['mode'] == 'OFFBOARD'):
        errors.append(z - args.height)
if not errors:
    raise SystemExit('No valid OFFBOARD samples in interval')
print('N:', len(errors))
print('Height RMSE, m:', math.sqrt(sum(e*e for e in errors)/len(errors)))
print('Max abs height error, m:', max(abs(e) for e in errors))
```

Целевая высота должна быть задана **в том же фрейме**, что и CSV. После относительного взлёта body на метр действительное значение `z` в map может отличаться от 1.0: возьмите фактическую назначенную высоту из логики миссии. `--height 1.0` выше лишь пример для опыта с таким абсолютным заданием.

• Задание

Ошибки возврата в трёх опытах: 0.10, 0.20 и 0.15 м. Найдите среднее и максимум. **Проверка:** 0.15 м и 0.20 м.

23.3 rosbag: запись сообщений

Rosbag – инструмент ROS, записывающий поток сообщений выбранных топиков в файл (bag-файл) и позволяющий воспроизвести его позже (глава 3). Это позволяет повторно анализировать опыт без нового полёта. Сначала уточните доступные топики (`rostopic list`), затем запишите минимальный набор. Ключ `-O` задаёт имя файла, далее перечислены топики: часы, преобразования TF (`/tf` – меняющиеся, `/tf_static` – неизменные), состояние автопилота, положение и кадры камеры.

```
mkdir -p "$HOME/clover_runs/run_01"
cd "$HOME/clover_runs/run_01"
rosbag record -O flight.bag /clock /tf /tf_static \
  /mavros/state /mavros/local_position/pose \
  /main_camera/image_raw_throttled
```

После Ctrl+C выполните `rosbag info flight.bag`. Кадры занимают много места. Сервисы navigate и land не записываются как топики: цели сохраняйте отдельно в CSV или публикуйте в топик.

Bag воспроизводит сообщения для анализа. Физический полёт Gazebo не повторяется. Перед проигрыванием **остановите симулятор и управляющие программы**:

```
# Terminal 1: isolated analysis session, simulator stopped
roscore
# Terminal 2
rosparam set /use_sim_time true
```

При `--clock` исключите записанный `/clock`, чтобы не создавать два источника времени:

```
rosbag play --clock flight.bag --topics /tf /tf_static \
  /mavros/state /mavros/local_position/pose \
  /main_camera/image_raw_throttled
```

Флаг `--clock` создаёт часы воспроизведения. Убедитесь, что нужные статические преобразования действительно содержатся в bag.

• Задание

Запишите 10 секунд любого топика командой rosbag record и посмотрите `rosbag info`. **Проверка:** в информации указана длительность около 10 с и число сообщений.

23.4 Серия опытов

Зафиксируйте world, карту, версии Clover/PX4, RTF и программу. Меняйте один параметр. Для каждой настройки сделайте минимум три повтора из одинакового состояния: это помогает заметить разброс, но ещё не даёт строгого статистического доказательства.

Запуск	Скорость	RTF	Ошибка	Результат
01	0.3 м/с	измерить	измерить	посадка / отказ
02	0.3 м/с	измерить	измерить	посадка / отказ
03	0.5 м/с	измерить	измерить	посадка / отказ

Это шаблон протокола, не результаты выполненных испытаний. Паузы и сбросы часов отмечайте явно; стандартный анализатор предполагает, что симуляционное время не перемещалось назад.

Практика. Сравнение скоростей

Выполните по три квадрата со скоростями 0.3 и 0.5 м/с. Сравните время и ошибку возврата, приложите CSV и паспорт стенда. Укажите интервал расчёта метрик так, чтобы опыт можно было независимо повторить.

Итоги и самопроверка

- Один опыт – одна переменная.
- Метрику определяют до опыта.
- Данные хранят вместе с версией кода и настроек.

Проверьте себя: 1) Почему нужно несколько повторов? 2) Что хранить рядом с лог-файлом?
Ответы: 1) одно измерение может быть случайным; 2) версию программы, параметры и условия.

ГЛАВА 24

Как искать ошибки и строить устойчивую миссию

Цели главы

- находить ошибку по слоям;
- искусственно вызывать отказ и видеть реакцию;
- вести дневник отладки.

Нужно знать заранее: глав 13 и 23.

24.1 Симптом, причина и проверка

Отладка – поиск и устранение причины ошибки в программе или системе. Симптом – то, что вы наблюдаете: «не летит», «нет картинки», «ожидание закончилось». Причина – механизм: неверное имя сервиса, отсутствие TF, неподходящий режим, устаревший кадр. Проверка – действие, которое подтверждает или отвергает одну гипотезу.

Полезная запись: «Вызов navigate вернул false с сообщением об отсутствующем фрейме. Проверю список TF и точное имя». Неполезная: «ROS сломался». Чем точнее наблюдение, тем меньше лишних изменений понадобится.

• Задание

Дрон не взлетает. Составьте таблицу: три возможные причины и проверка для каждой.
Проверка: у каждой причины есть своя проверка.

24.2 Проверяйте систему слоями

1. Есть ли процесс и ROS master? Посмотрите терминал запуска и `rosnode list`.
2. Есть ли нужный интерфейс? Посмотрите `rostopic list` или `rosservice list`.
3. Совпадает ли тип? Используйте `rostopic info`, `rosservice type`.
4. Приходят ли данные? Выполните `rostopic echo`, проверьте частоту и временную метку сообщений (timestamp).
5. Корректны ли значения и фрейм? Проверьте конечные числа и TF.
6. Принимается ли действие? Сохраните success и message.
7. Меняется ли состояние после действия? Наблюдайте телеметрию.

Так вы двигаетесь от наличия системы к смыслу данных. Нет смысла настраивать HSV, если камера вообще не публикует кадры. Нет смысла менять скорость, если navigate отвергнут из-за неизвестного фрейма.

• Задание

Расставьте проверки от нижнего слоя к верхнему: сервис navigate; Gazebo запущен; MAVROS connected; камера публикует кадры. **Проверка:** Gazebo, MAVROS, навигационный сервис, камера.

24.3 Опыты с искусственными отказами

Исследование отказа – обычный учебный опыт: заранее известна внесённая ошибка и ожидаемая реакция. Начинайте с вариантов, которые не отправляют управление. Например, поменяйте имя топика listener или выключите сервер виртуальной лампы. Затем переходите к коротким полётным опытам в свободной сцене.

Внесённая ошибка	Ожидаемый симптом	Проверка
Неправильный топик listener	Нет сообщений	Info правильного и ошибочного имени
Сервер лампы остановлен	Таймаут клиента	Rosservice list
Window равен 0	Явная ошибка параметра	Перезапуск с window 5
Несуществующий фрейм	Отказ или некорректная позиция	Ответ сервиса и TF
Очень короткий таймаут	Цель не успела завершиться	Журнал времени и расстояния
Площадка вне кадра	NaN в измерении детектора	Debug-изображение

В каждом опыте меняйте одну причину и потом возвращайте рабочую настройку. Не собирайте пять отказов одновременно: будет трудно понять реакцию.

• Задание

Выберите один отказ (неверный фрейм) и заранее опишите ожидаемую реакцию программы. **Проверка:** ожидание: сообщение об отказе, остановка маршрута, посадка.

24.4 Состояния миссии

Длинная миссия удобнее читается как последовательность состояний: CHECK, TAKEOFF, ROUTE, LAND, DONE. У каждого состояния есть входные условия, действие и событие завершения. Ошибка переводит в предусмотренную ветку завершения, если посадка ещё возможна.

Конечный автомат – модель, в которой система в каждый момент находится в одном из состояний, а события переводят её из одного состояния в другое. Пример без ROS показывает только решения, не движение:

```
#!/usr/bin/env python3
"""Offline event machine: simulate decisions, not flight dynamics."""


def next_state(state, event):
    if state == 'DONE':
        return 'DONE'
    if event == 'error':
        return 'LAND'
    transitions = {('CHECK', 'ready'): 'TAKEOFF',
                   ('TAKEOFF', 'arrived'): 'ROUTE',
                   ('ROUTE', 'finished'): 'LAND',
                   ('LAND', 'disarmed'): 'DONE'}
    return transitions.get((state, event), state)


if __name__ == '__main__':
    state = 'CHECK'
    for event in ['ready', 'arrived', 'waiting', 'error', 'disarmed']:
        new_state = next_state(state, event)
        print(state, '+', event, '->', new_state)
        state = new_state
```

Запустите `python3 event_machine.py`. Событие waiting не меняет состояние. Событие error переводит в LAND. После disarmed получается DONE. Обратите внимание: само слово LAND не выполняет посадку. В реальной программе действию состояния должен соответствовать вызов land и наблюдение результата.

Сделайте таблицу переходов

Добавьте состояние PHOTO между ROUTE и LAND. Событие finished переводит из ROUTE в PHOTO, событие saved – из PHOTO в LAND. Проверьте последовательность обычного завершения и отказ при сохранении кадра. Дополнительное задание: объясните, почему неожиданное событие не должно само собой переводить в DONE.

24.5 Сторож времени и свежесть данных

Наличие топика не означает, что сообщения продолжают поступать. Для наблюдения нужно хранить время последнего получения и проверять возраст. Время кадра внутри сообщения может отличаться от времени получения; первое важно для геометрии движения, второе помогает понять, что связь не молчит.

При сравнении времени используйте одну шкалу. ROS stamp сравнивайте с ROS-временем, а `time.monotonic` – с другим показанием monotonic. Вычитание timestamp камеры из часов monotonic не даёт осмысленной задержки.

После reset Gazebo время может пойти назад. Не сбрасывайте мир в середине работающей миссии или записи, рассчитывая на корректность старых данных. Корректная процедура: завершить опыт, перезапустить нужные ноды и начать новый журнал. Отдельно можно изучить обработку скачка часов как продвинутую тему.

• Задание

Выберите границу свежести данных для камеры 10 Гц и объясните выбор. **Проверка:** граница больше 0.1 с (периода кадра) и меньше времени, за которое дрон заметно смещается.

24.6 Очередь и тяжёлая обработка

Если камера даёт десять кадров в секунду, а детектор обрабатывает только два, длинная очередь начинает накапливать прошлое. Изображение может быть правильным, но уже устаревшим. Для управления свежесть важнее обработки каждого кадра; для офлайн-анализа могут быть другие требования.

В учебном детекторе очередь короткая, а длительная обработка вынесена через механизм long_callback. Но это не освобождает от измерения задержки. Проверьте частоту входа, выхода и возраст кадра. Ускорение алгоритма измеряйте отдельно от улучшения распознавания.

• Задание

Почему при тяжёлой обработке полезна очередь размером 1? **Проверка:** обрабатывается свежий кадр, а не накопленные старые.

24.7 Почему несколько миссий мешают друг другу

Две программы могут обращаться к одному navigate. Одна задаст первую точку, другая заменит её. Оба процесса останутся работающими, но каждый будет наблюдать цель, которую мог изменить сосед. Список нод с разными именами не предотвращает такой конфликт.

Управлять аппаратом должна ровно одна программа. Наблюдатели и логгеры могут работать параллельно, потому что не назначают движения. Если нужен совместный проект с несколькими источниками команд, потребуется явная система выбора владельца управления; в этой книге используется одна миссия за раз.

• Задание

Что произойдёт, если одновременно запущены два скрипта с `navigate`? **Проверка:** цели конфликтуют: дрон выполняет то одну, то другую.

24.8 Дневник отладки

Для каждого отказа сохраняйте: команду запуска, версию файла, точное сообщение, состояние перед отказом и одно изменение после проверки. Пример:

```
Experiment 07
Command: python3 triangle.py
Symptom: arrival timeout at point 2
Observation: RTF approximately 0.35; distance still decreases
Hypothesis: wall-clock timeout is too short for this stand
Next change: increase timeout only; keep route and speed
```

Если расстояние не уменьшается и режим вышел из OFFBOARD, увеличение таймаута вряд ли исправит причину. Запись наблюдения отличает эти случаи. Код «заработал после случайных правок» труднее повторить и объяснить.

Разбор чужой ошибки

Внесите в рабочий пример одну ошибку и сохраните только traceback или журнал. По этим данным, не заглядывая в код, назовите симптом, предложите проверку и укажите, что должно измениться при верной гипотезе. Затем сравните вывод с реально внесённой ошибкой.

Итоги и самопроверка

- Симптом – причина – проверка; меняем одну вещь за раз.
- Сторож времени ловит «зависшие» данные.
- Две управляющие миссии одновременно мешают друг другу.

Проверьте себя: 1) Что такое свежесть данных? 2) Почему нельзя запускать две миссии?
Ответы: 1) возраст последнего сообщения; 2) обе отправляют цели автопилоту.

ГЛАВА 25

Если реальный дрон ведёт себя не так

Цели главы

- соблюдать правила безопасности при работе с реальным аппаратом и знать порядок действий при нештатном поведении;
- провести предполётную проверку по списку;
- отличать проблему железа и настройки автопилота от проблемы в программе миссии, разделяя ручной и автономный режимы;
- находить причину вибраций (тряски) и устранять её;
- по симптому определять вероятную причину и порядок проверки.

Нужно знать заранее: главы 3 (устройство реального Клевера), 13 (режимы, armed, OFFBOARD), 9 (флаги), 24 (отладка по слоям).

25.1 Реальный дрон – не симулятор

В симуляторе ошибка стоит минуты перезапуска. На реальном аппарате вращающиеся винты способны нанести травму, а падение – повредить дорогую электронику. Кроме того, у реального аппарата есть то, чего нет в модели: механические вибрации, неидеальные датчики, просадка напряжения батареи, ветер, радиопомехи. Поэтому перед любым полётом необходимы правила безопасности, а при странном поведении – чёткий порядок действий вместо «попробуем ещё раз».

Базовые правила безопасности

По инструкции COEX: пропеллеры устанавливают только непосредственно перед вылетом; перед подключением батареи включают пульт управления и ставят левый стик (газ) в нулевое положение; батарею подключают только перед взлётом; в момент взлёта находятся на расстоянии не менее 3 м от аппарата, а зрители стоят за спиной пилота; летают только в обозначенной зоне, скорость не выше скорости идущего человека; ремонт и пайку выполняют на отключённой батарее. **Рядом с аппаратом при любом автономном полёте должен быть пилот с пультом**, готовый перехватить управление или аварийно отключить моторы.

25.1.1 Аварийная и штатная посадка

1. Прекратить полёт, посадить аппарат; левый стик (газ) – в минимум.
2. Выполнить disarm (по инструкции COEX: левый стик влево-вниз и удерживать около трёх секунд).
3. Отключить батарею.
4. Выключить пульт.
5. Осмотреть аппарат; при жёсткой посадке или ударе – обязательно, до следующего включения.

Для аварийного отключения моторов в воздухе на пульте назначают *kill switch* – переключатель, мгновенно выключающий моторы (в документации COEX – SwA). Им пользуются только если аппарат опасен для людей: моторы остановятся, аппарат упадёт.

• Задание

Составьте для своего аппарата карточку «Действия при нештатной ситуации» из пяти пунктов и повесьте её на рабочем месте. **Проверка:** в карточке есть: перехват управления пультом, посадка, disarm, отключение батареи, осмотр.

25.2 Предполётная проверка

Большинство «странностей» поведения – следствие пропущенного пункта этого списка.

Узел	Что проверить
Механика	Затяжка гаек пропеллеров; отсутствие трещин и сколов на лопастях; пропеллеры не погнуты и правильного типа (по и против часовой стрелки на своих моторах); моторы вращаются свободно, вал не погнут; стойки и деки не болтаются; защита винтов цела
Полётный контроллер	Закреплён *на виброразвязке*, а не жёстко (см. раздел о вибрациях); плата ориентирована правильно (стрелка вперёд, либо верно задана ориентация в настройках); провода не натянуты и не касаются платы, не стучат при потряхивании
Питание	Батарея заряжена, напряжение в норме; разъёмы надёжны; светодиодная лента питается от стабильного преобразователя, а не напрямую от Raspberry Pi (перегрузка бортового компьютера)
Калибровки	Акселерометр, гироскоп, при использовании – компас и уровень горизонта (QGroundControl, раздел Sensors); при смене положения контроллера – заново
Пульт и режимы	Пульт откалиброван; переключатель режимов даёт нужные режимы (Stabilized, Altitude, Position); назначены kill switch и реакция на потерю связи с пультом (failsafe: посадка или отключение моторов, таймаут около 2 с)
Бортовой компьютер	Камера в фокусе и кадры идут; `selfcheck.py` без предупреждений уровня WARN по связи, позиции и оценке скоростей; флаги включены (глава 9)
Позиционирование	Под камерой текстурный пол (для оптического потока); дальномер видит пол; маркеры видны и карта верна (глава 20)
Зона	Зона полёта обозначена, расстояние до препятствий не менее 3 м, нет людей в зоне взлёта

Если контроллер COEX Pix установлен серворазъёмами назад, в калибровках (Autopilot Orientation) указывают ориентацию `ROTATION_ROLL_180_YAW_90`; иначе автопилот неправильно воспринимает наклоны, и аппарат при взлёте ведёт себя непредсказуемо.

• Задание

Проведите предполётную проверку по таблице на своём аппарате и заполните лист: пункт, результат, подпись. Одну неисправность найдите намеренно (например, незатянутая гайка), опишите, чем она опасна. **Проверка:** в листе отмечены все восемь узлов; для найденной неисправности указаны последствия.

25.3 Главный приём диагностики: разделяй ручное и автономное управление

У реального Клевера две «половины» (глава 3): полётный контроллер с автопилотом (механика, датчики, настройка PID, режимы) и бортовой компьютер с вашей программой. Чтобы понять, какая половина виновата, сравните поведение в разных режимах.

Наблюдение	Вывод
Аппарат трясёт или раскачивает *и в ручном режиме* (Stabilized, Altitude)	Проблема в железе или настройках автопилота: вибрации, балансировка, PID, калибровки, питание. Программа миссии ни при чём
В ручном режиме летит хорошо, а в режиме с удержанием позиции (Position) – дёргается или дрейфует	Проблема источника позиции: оптический поток, дальномер, маркеры (VPE), компас
В ручном режиме и в Position нормально, а в OFFBOARD по вашей программе – плохо	Проблема программы миссии, её целей и скоростей или связи бортового компьютера с автопилотом
Аппарат не армится никаким способом	Проверки автопилота перед полётом (preflight checks), калибровки, батарея, переключатель безопасности; читайте сообщение в QGroundControl

Такое разделение экономит часы: нельзя настраивать код, если дрожит механика, и нельзя менять PID, если неверна программа.

• Задание

Дрон ровно висит в режиме Position, но при запуске программы по маркерам сильно «качается» над полем. В каком режиме и какую проверку выполнить первой? **Проверка:** сначала убедиться, что в Position висит хорошо (исключить железо); затем проверять источник позиции по маркерам (карта, свет, `aruco_vpe`) и свежесть данных.

25.4 Вибрации и тряска

Вибрация – быстрые колебания корпуса, которые передаются датчикам полётного контроллера. Автопилот определяет наклон по гироскопу и акселерометру. Если на их показания накладываются сильные колебания от моторов и винтов, автопилот «видит» то, чего нет, и пытается это компенсировать: моторы дёргаются, аппарат дрожит, качается, плохо держит высоту и позицию, а оценка положения может сбиться. Вибрации опасны ещё тем, что *не*

лечатся настройками в коде.

Основные причины:

1. **Жёсткое крепление полётного контроллера.** Плата намертво закреплена на раме, и вибрации моторов идут напрямую на датчики.
2. Неуравновешенные, погнутые, треснувшие или разные по типу пропеллеры.
3. Изношенный подшипник или погнутый вал мотора, плохо закреплённый мотор.
4. Ослабленные стойки и деки, резонанс рамы; провода, натянутые на плату контроллера.
5. Слишком агрессивные коэффициенты регулятора (раскачка из-за настройки, а не вибраций).

Как контроллер должен быть закреплён. По инструкции сборки Клевера 4 (COEX): COEX Pix устанавливают на демпферные стойки 6 мм и крепят нейлоновыми гайками; Pixracer приклеивают к малой деке на **3–4 слоя двустороннего скотча** (мягкая виброразвязка). Идея одна: между платой и рамой должен быть мягкий слой, который гасит вибрации моторов. Если плату разобрать и вмонтировать *жёстко* – например, закрепить винтами на стойках без мягкого слоя, – вибрации передаются напрямую и аппарат трясёт, в том числе в автономных режимах.

Разбор типичного случая

Симптом. При полёте по автопилоту (удержание позиции, автономная миссия) аппарат заметно трясёт. **Что проверили.** В ручном режиме тряска тоже присутствует, значит, виновата не программа и не маркеры. Винты сбалансированы и целы, моторы вращаются ровно. **Причина.** Контроллер Pixracer при сборке был вынут из штатного крепления и жёстко закреплён на стойке, а не посажен на несколько слоёв мягкого двустороннего скотча, как требует инструкция. **Решение.** Вернуть контроллер на 3–4 слоя двустороннего скотча (мягкая виброразвязка), убедиться, что провода не натянуты, повторить калибровку акселерометра и контрольный полёт в ручном режиме, затем в Position, затем в автономном. **Вывод.** Не меняйте конструкцию крепления без понимания её назначения: виброразвязка – такая же часть системы, как мотор.

25.4.1 Как измерить вибрации

Измерение даёт числа вместо ощущений и позволяет сравнить «до» и «после».

- В QGroundControl (программа настройки и мониторинга автопилота) можно просматривать данные датчиков и графики во время работы; разброс (шум) показаний акселерометра в покое на земле и при малых оборотах – первый признак проблемы.
- После полёта журнал автопилота (лог на SD-карте) можно открыть в программах анализа логов PX4 и посмотреть графики вибраций и предупреждения оценщика положения.
- Программой из комплекта книги: `imu_vibration.py` читает необработанные данные инерциального датчика (`/mavros/imu/data_raw`) и печатает разброс (стандартное отклонение) ускорений и угловых скоростей по трём осям:

```
python3 imu_vibration.py --seconds 10
python3 imu_vibration.py --selftest
```

Режим `--selftest` работает без дрона на искусственных сигналах и показывает, как отличается «жёстко закреплённый» аппарат от «мягко закреплённого» (в примере коэффициент различия около 150; это демонстрация метода, а не измерение реального аппарата). Абсолютные числа зависят от датчика и режима, поэтому измеряйте на одном и том же аппарате

в одинаковых условиях и сравнивайте *до и после* изменения, либо с заведомо исправным аппаратом.

```
#!/usr/bin/env python3
"""Estimate airframe vibration from raw IMU data (accelerometer + gyro).

  python3 imu_vibration.py --selftest     offline demo on synthetic signals
  python3 imu_vibration.py --seconds 10   ROS: reads /mavros/imu/data_raw

Run it with the motors armed on the ground at low throttle (propellers
off in the workshop test) or in a steady hover and compare the numbers
with a known-good aircraft or with your own earlier measurement.
The absolute numbers depend on the sensor; the *change* is what matters.
"""
import argparse
import math
import statistics


def spread(values):
    """Standard deviation of the signal after removing its mean (the DC part)."""
    return statistics.pstdev(values) if len(values) > 1 else 0.0


def report(ax, ay, az, gx, gy, gz):
    acc = [spread(ax), spread(ay), spread(az)]
    gyro = [spread(gx), spread(gy), spread(gz)]
    print('accelerometer std, m/s^2 (x, y, z): %.3f %.3f %.3f' % tuple(acc))
    print('gyro std, rad/s       (x, y, z): %.3f %.3f %.3f' % tuple(gyro))
    return max(acc), max(gyro)


def selftest():
    # 100 Hz, 5 s: a calm airframe vs. one with a 60 Hz vibration on the accelerometer
    n = 500
    quiet = [0.02 * math.sin(0.3 * i) for i in range(n)]
    shaky = [3.0 * math.sin(2 * math.pi * 60 * i / 100.0 + 0.3 * i) for i in range(n)]
    print('rigid-mounted airframe (synthetic):')
    a1, _ = report(shaky, shaky, shaky, quiet, quiet, quiet)
    print('soft-mounted airframe (synthetic):')
    a2, _ = report(quiet, quiet, quiet, quiet, quiet, quiet)
    assert a1 > 10 * a2, 'vibration metric must separate the two cases'
    print('selftest OK: vibration ratio %.0f' % (a1 / max(a2, 1e-9)))


def run_ros(seconds):
    import time
    import rospy
    from sensor_msgs.msg import Imu

    rospy.init_node('course_imu_vibration')
    data = {k: [] for k in ('ax', 'ay', 'az', 'gx', 'gy', 'gz')}

    def on_imu(m):
        data['ax'].append(m.linear_acceleration.x)
        data['ay'].append(m.linear_acceleration.y)
```

```
        data['az'].append(m.linear_acceleration.z)
        data['gx'].append(m.angular_velocity.x)
        data['gy'].append(m.angular_velocity.y)
        data['gz'].append(m.angular_velocity.z)

    rospy.Subscriber('/mavros/imu/data_raw', Imu, on_imu, queue_size=200)
    end = time.monotonic() + seconds
    while time.monotonic() < end and not rospy.is_shutdown():
        time.sleep(0.05)
    if len(data['ax']) < 20:
        raise SystemExit('Too few IMU messages: is MAVROS connected?')
    print('samples:', len(data['ax']))
    report(*(data[k] for k in ('ax', 'ay', 'az', 'gx', 'gy', 'gz')))


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--selftest', action='store_true')
    parser.add_argument('--seconds', type=float, default=10)
    args = parser.parse_args()
    selftest() if args.selftest else run_ros(args.seconds)
```

Практика. Измерение вибраций до и после

Закрепите аппарат (пропеллеры сняты, батарея подключена только для данных контроллера; мотор не запускать) или выполните измерение в висении с пилотом. Запишите разброс ускорений. Затем внесите одно изменение (замена пропеллера, затяжка деки, перемонтаж контроллера на мягкий слой) и повторите. **Проверка:** в отчёте – таблица «до/после» с одним изменённым фактором, вывод о причине.

25.5 Таблица «симптом – причина – проверка»

Это диагностические гипотезы: проверяйте факт, а не меняйте всё подряд.

Симптом	Возможная причина	Что проверить и сделать
Тряска, дрожание	Вибрации (жёсткий контроллер, винты, моторы)	Раздел о вибрациях; ручной режим; балансировка винтов; мягкое крепление контроллера
Раскачка по крену или тангажу, «ходит»	Агрессивные коэффициенты регулятора, неверный центр тяжести, вибрации	Сначала исключить вибрации; центровка батареи; затем настройка коэффициентов по документации PX4
Дрейф в висении (в Position)	Оптический поток (нет текстуры пола, плохой свет, камера смотрит не вниз), дальномер, компас, VPE	`selfcheck.py`; изображение оптического потока; поверхность и освещение; калибровка
Не армится	Предполётные проверки, режим, калибровки, батарея, kill switch	Сообщение в QGroundControl; калибровки; положение переключателей

При взлёте переворачивается или улетает в сторону	Неверное направление вращения моторов или порядок, пропеллеры не на своих моторах, неверная ориентация контроллера	Схема вращения; тип пропеллеров по и против часовой стрелки; `Autopilot Orientation`; пробный запуск без пропеллеров
Не входит или выходит из OFFBOARD	Нет потока целей, нет позиции, потеря связи, переключатель режимов пульта	Телеметрия: `mode`, `armed`; `rostopic hz` состояния MAVROS; связь по проводу
Слабая тяга, просадка высоты, внезапное снижение	Батарея разряжена или просаживается под нагрузкой; перегружен аппарат; неверный газ висения	Напряжение под нагрузкой; масса; параметр газа висения по документации
Дальномер «скачет», высота неверна	Тёмный, блестящий или прозрачный пол; наклон; перекрыт датчик	Показания дальномера над разными поверхностями; поле зрения датчика
Raspberry Pi перезагружается, пропадает Wi-Fi	Просадка питания, перегрев, перегрузка светодиодной лентой	Питание через преобразователь, нагрузка, температура (`selfcheck.py`, строка RPi health)
Камера не даёт кадров	Шлейф, выключен флаг камеры, нет питания	Флаг `main_camera`; утилита `raspistill` при остановленной службе (по документации COEX); шлейф
Маркеры не распознаются	Словарь, размер, свет, флаг, карта	Глава 20
Страница Blockly не открывается или программа не запускается	Флаг `blocks` выключен, служба не перезапущена	Глава 9
После прошивки или правки всё сломалось	Ошибка в launch-файле, смена параметров	Резервная копия файла; вернуть последнюю рабочую конфигурацию

25.6 Алгоритм действий при нештатном поведении

1. **Безопасность.** Если аппарат опасен – перехватить управление пультом и посадить; при угрозе людям – kill switch. Не пытаться «досмотреть» полёт.
2. **Disarm и отключение батареи** после посадки.
3. **Запись симптомов** (что, в каком режиме, на какой фазе полёта, уровень батареи, что менялось в последний раз). Сохранить лог автопилота и журнал программы.
4. **Осмотр механики**: винты, моторы, крепления, контроллер, провода.
5. **Разделение**: ручной режим, Position, OFFBOARD (раздел выше). Определить половину системы.
6. **Одно изменение за раз**, после каждого – контрольная проверка (сначала на земле, по-

том короткий низкий полёт).

7. **Документирование**: что было, что сделали, результат. Это основа повторяемости.

Разбор инцидента

Опишите по шаблону из семи пунктов реальный или придуманный случай: «после замены пропеллера аппарат стал уходить влево при взлёте». Укажите гипотезы (минимум три), проверку каждой и порядок проверок. **Проверка:** гипотезы включают установку винтов не по типу, повреждение мотора или винта, ориентацию контроллера; каждая имеет безопасную проверку на земле до полёта.

Задание профессионального уровня: «Найди неисправность»

Преподавательская или соревновательная постановка: на аппарате намеренно создана одна неисправность из списка (жёсткое крепление контроллера, один пропеллер не затянут, не откалиброван акселерометр, неверная ориентация контроллера, выключен флаг `aruco`). Участник за 20 минут должен: провести предполётную проверку, назвать вероятную причину, подтвердить её измерением или осмотром, устранить и подтвердить устранение контрольным полётом. **Критерии:** безопасность (перехват, disarm) – 30%; обоснованность диагноза – 30%; устранение – 20%; отчёт – 20%.

Источник: COEX, инструкция по безопасности. Проверено 11.10.2026.

Источник: COEX, калибровка датчиков. Проверено 11.10.2026.

Источник: COEX, сборка Клевера 4: установка полётного контроллера. Проверено 11.10.2026.

Итоги и самопроверка

- Реальный аппарат требует правил безопасности: пилот с пультом, disarm, kill switch, осмотр после жёсткой посадки.
- Предполётная проверка охватывает механику, контроллер, питание, калибровки, пульт, бортовой компьютер, позиционирование и зону.
- Разделение ручного, позиционного и автономного режимов показывает, чья половина системы виновата.
- Вибрации – частая причина тряски; контроллер должен быть на мягкой виброразвязке (демпферные стойки или 3–4 слоя двустороннего скотча), а не жёстко.
- Меняйте одну вещь за раз и документируйте.

Проверьте себя: 1) Аппарат трясёт и в ручном режиме. Чья это половина системы? 2) Как должен быть закреплён Pixracer на Клевере 4? 3) Что делать первым делом при опасном поведении в полёте? 4) Зачем измерять вибрации до и после изменения?
Ответы: 1) полётного контроллера и железа, не программы; 2) на 3–4 слоя двустороннего скотча (мягко), а не жёстко; 3) перехватить управление пультом и посадить (при угрозе – kill switch); 4) чтобы убедиться, что именно изменение дало эффект.

ГЛАВА 26

Проекты для задач с Клевером

Цели главы

- выбрать проект и описать критерии успеха;
- разбить миссию на состояния;
- подготовить повторяемый запуск.

Нужно знать заранее: глав 15–23.

26.1 Что считать завершённым проектом

Для каждого проекта нужны сценарий, программа, world/карта, измеримые критерии и описание неудачных случаев. В Gazebo можно проверить логику и часть геометрии. Захват груза, тепловизор, реальная задержка сети или специфическая новая камера требуют дополнительных моделей. Если модуль только имитируется программным флагом, явно укажите это в отчёте.

26.1.1 Проект A. «Курьер между маркерами»

Задача: стартовать над маркером 0, посетить 1 и 2, вернуться к 0 и сесть. В каждой точке удерживаться 2 симуляционные секунды и записывать время события.

База: `lab_map.txt`, `aruco_route.py`, логгер. Добавьте список заданий `(x, y, z, hold)` и запись события после успешного прибытия. Наличие маркера в карте и его непосредственная видимость – разные условия; решите, какое требуется по сценарию.

Критерий: все точки посещены в правильном порядке; ошибка до цели менее 0.2 м; программа завершает посадку; три успешных повтора. «Доставка» без физической модели груза означает логическое событие, а не проверку механики захвата.

26.1.2 Проект B. «Осмотр учебного склада»

Задача: обследовать сетку 2×2 м, сохраняя изображение и координаты в заданных узлах. Между узлами стоят коробки. Сначала спроектируйте свободный коридор, затем генерируйте точки.

База: маршрут «змейка», SDF коробок, камера и логи. Геометрия планируется в фиксированном фрейме; карта Gazebo должна быть согласована с ним. Для захвата кадра по запросу можно использовать `rospy.wait_for_message` с таймаутом, но для непрерывного зрения предпочтительна подписка.

Критерий: не менее заданного покрытия, кадры не устарели, траектория не касается препятствий, папка отчёта содержит снимки с координатами и timestamp. Уточните, нужен ли каждый кадр именно после полной стабилизации.

26.1.3 Проект C. «Найти красную площадку»

Задача: найти одну площадку среди нескольких объектов и сообщить положение; второй этап – центрировать коптер над ней, третий – снижаться при подтверждённых условиях.

База: `color_detector.py`. Добавьте выбор контура и оценку достоверности. Отдельная нода контроллера читает свежую цель, проверяет timestamps и выдаёт ограниченные команды. После длительной потери объекта миссия прекращает поиск по заранее заданному условию.

Критерий: детектор не путает маленький шум и площадку, реагирует на пропажу цели, а центровка укладывается в установленную пиксельную ошибку. Не объявляйте точную посадку выполненной без отдельной проверки высоты и положения на земле.

26.1.4 Проект D. «Коридор и обход»

Задача: пройти от старта к цели вокруг прямоугольного препятствия. Базовый вариант – вручную задать промежуточные точки. Расширенный – построить сетку занятости (карту, где каждая клетка помечена «свободна» или «занята») и найти путь алгоритмом A* (алгоритм поиска кратчайшего пути на сетке).

База: статическая карта препятствий, `flight.go`. Для планирования увеличьте препятствие на радиус/полугабарит дрона плюс запас. Внутренняя траектория должна проходить по свободным ячейкам; диагональный проход между касающимися препятствиями тоже проверяется.

Критерий: путь существует в допустимых границах, нет столкновений, высота выдержана, недостижимая цель вызывает явный отказ. Коробка должна содержать `collision`, иначе проверяется только визуальная близость.

26.1.5 Проект E. «Устойчивость к потере наблюдения»

Задача: после успешной базовой миссии частично перекрыть маркерное поле и оценить, как программа обнаруживает устаревшие данные.

База: временная метка последнего валидного наблюдения карты и состояние автомата. Изменяйте видимость именно маркеров, сохраняя остальную систему настолько неизменной, насколько возможно. Не используйте отсутствие NaN как единственный критерий свежести.

Критерий: событие потери наблюдения обнаружено в установленный интервал, дальнейшая реакция записана в лог, программа не продолжает маршрут бесконечно. Это опыт с определённым отказом, а не универсальная проверка failsafe автопилота.

• Задание

Выберите проект и заполните карточку: цель, вход, выход, критерий успеха, число повторов. **Проверка:** критерий успеха содержит число и единицу, а повторов не меньше трёх.

26.2 Автомат состояний миссии

Сложную миссию лучше описать конечным автоматом. Каждое состояние имеет условия входа, действие, условие завершения и таймаут. Пример учебного автомата:

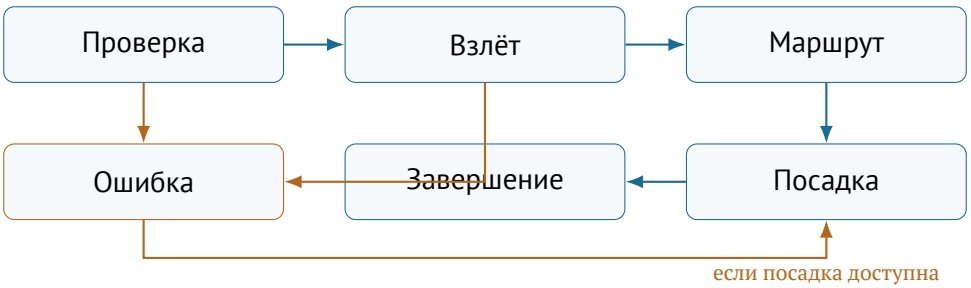


Рис. 26.1. Реакция на ошибку зависит от доступности связи и позиции; переход к посадке не гарантирует её исполнение.

Практика. Выберите итоговый проект

Запишите входы, последовательность, поле, ограничения, три критерия успеха и два предусмотренных отказа. Соберите минимальную рабочую версию, затем добавляйте усложнения.

Итоги и самопроверка

- Завершённый проект имеет критерий успеха, повторяемость и отчёт.
- Автомат состояний делает миссию читаемой.

Проверьте себя: 1) Как проверить, что проект повторяется? 2) Зачем явное состояние «посадка»?
Ответы: 1) запустить несколько раз с теми же настройками и сравнить; 2) завершение должно быть состоянием, а не побочным эффектом.

ГЛАВА 27

Практикум профессионального уровня

Цели главы

- понимать, как устроены задания профессиональных состязаний по эксплуатации и программированию беспилотных систем (модули, время, оценка, штрафы);
- строить решение конкурсного задания: разбивать на подзадачи, составлять минимальную рабочую программу, управлять временем и риском;
- решать типовые модули: подготовка и проверка аппарата, обход маркеров, чтение QR-кодов, цветные зоны, обследование поля, безопасное завершение при отказе;
- самостоятельно оценивать результат по критериям и вести отчётную документацию.

Нужно знать заранее: главы 13, 16, 20, 21, 24, 9 и 25.

27.1 Как устроены профессиональные состязания

На чемпионатах профессионального мастерства (в России это, в частности, чемпионаты серии «Профессионалы» и отраслевые соревнования по беспилотным авиационным системам) участник получает *конкурсное задание*, состоящее из *модулей*. Каждый модуль имеет условие, ограничение по времени, оборудование, критерии оценки и штрафы. Работа участника оценивается экспертами по заранее опубликованной схеме: часть критериев *измеряется* (время, точность, число выполненных точек, наличие столкновений), часть оценивается экспертом (безопасность, порядок работы, качество отчёта).

Важное замечание о материалах главы

Задания этой главы разработаны автором *в формате* конкурсных (модули, время, критерии, штрафы) для тренировки с Клевером. Это не официальные задания и не официальные критерии конкретного чемпионата. Регламент, конкурсное задание и схему оценки каждого соревнования публикуют его организаторы, и они могут отличаться от приведённых здесь; перед состязанием обязательно изучите официальные документы и инфраструктурный лист (перечень оборудования и условий).

27.1.1 Типовые модули

Модуль	Что требуется	Время (ориентир)
Подготовка и диагностика аппарата	Проверить, настроить, найти и устранить неисправность	20–30 мин
Полёт по меткам	Посетить ArUco-маркеры в заданном порядке, посадка на маркер	3–5 мин на попытку
Распознавание	Прочитать QR-код, найти цветные зоны, сообщить результат	3–5 мин
Обследование поля	Облететь поле «змейкой», зафиксировать объекты, составить отчёт	5–10 мин
Работа с отказом	Безопасно завершить миссию при потере данных	3–5 мин
Комплексная задача	Объединение предыдущих модулей на одном полигоне	8–12 мин

Времена – ориентиры для тренировки; в конкурсе они задаются регламентом.

27.1.2 Принципы оценки

- **Измеримые критерии**: каждая посещённая точка, точность посадки, время, число столкновений. Для них удобно написать собственный «автосудья» (программа `score_demo.py`, раздел 27.3) и тренироваться, зная счёт заранее.
- **Штрафы**: нарушение порядка, выход из зоны, касание препятствий, превышение времени – всегда дороже, чем потеря одной точки. Стратегия: надёжно сделать часть, чем рискованно – всё.
- **Безопасность** оценивается отдельно и может обнулить результат: аппарат с пультом пилота, disarm, отсутствие людей в зоне.
- **Воспроизводимость**: программа должна давать результат при повторе; нестабильное решение хуже стабильного, но чуть менее быстрого.

• Задание

Какой из двух вариантов выгоднее при схеме «+10 за точку, -10 за нарушение порядка»: облететь четыре точки быстро, рискуя перепутать порядок, или три надёжно? **Проверка:** вариант А – четыре точки по +10, при ошибке порядка теряется 20 баллов (потерянные +10 и штраф -10) с вероятностью p: среднее $40-20p$; вариант Б – три точки надёжно: 30. Равенство при $p = 0.5$: если вероятность ошибки ниже, выгоднее А, если выше – Б. Вывод: оценивайте вероятность ошибки по тренировочным прогонам.

27.2 Как решать конкурсное задание

1. **Прочитайте условие дважды** и выпишите требования отдельным списком: что должно произойти, в каком порядке, за какое время, какие ограничения, что оценивается. Отметьте то, что легко пропустить: формат отчёта, имена файлов, зона, высота.
2. **Разбейте задание на подзадачи** и расставьте по важности: сначала то, что даёт баллы наверняка (взлёт, одна точка, посадка), затем остальное.
3. **Минимальная рабочая программа**: самый короткий вариант, который взлетает, делает одну вещь и безопасно садится. Только после его проверки добавляйте возможности.

Каждый шаг – сохранённая рабочая версия (глава 4).

4. **Каркас с конечным автоматом**: состояния CHECK, TAKEOFF, ROUTE, LAND (глава 24); таймаут на каждое состояние; общий бюджет времени; запись событий в журнал.
5. **Безопасное завершение всегда**: любая ошибка или исчерпание времени ведёт к посадке.
6. **Проверка на симуляторе и на земле**, затем короткий полёт, затем полное задание. Не запускайте на аппарате непроверенный код.
7. **Отчёт**: что выполнено, что нет, почему; журнал полёта; настройки.

Каркас такой программы приведён в файле `pro_mission_skeleton.py`: общий бюджет времени, журнал событий в CSV и безопасное завершение. Он обходит маркеры списком и пишет события, которые читает автосудья.

```
#!/usr/bin/env python3
"""Competition-style mission skeleton: states, time budget, event log.

States run in order; any exception or time budget overrun leads to the LAND
state.  Every event is written to a CSV file that the offline judge
(score_demo.py) can read.  Fill in the body of each state for your task.

Parameters:  _real:=false  _budget:=170  _log:=flight_log.csv
"""
import csv
import math
import time
import rospy
from flight_tools import Flight

rospy.init_node('course_pro_mission')
real = bool(rospy.get_param('~real', False))
budget = float(rospy.get_param('~budget', 170.0))      # seconds for the whole task
log_path = rospy.get_param('~log', 'flight_log.csv')
started = time.monotonic()


def elapsed():
    return time.monotonic() - started


def out_of_time():
    return elapsed() > budget


class Log:
    def __init__(self, path):
        self.handle = open(path, 'w', newline='', encoding='utf-8')
        self.writer = csv.writer(self.handle)
        self.writer.writerow(['event', 'marker', 'time_s', 'error_m'])

    def event(self, name, marker=-1, error=float('nan')):
        self.writer.writerow([name, marker, f'{elapsed():.1f}', f'{error:.2f}'])
        self.handle.flush()
        rospy.loginfo('%s %s %.1fs', name, marker, elapsed())
```

```
def visit(flight, log, marker, height):
    flight.go(x=0, y=0, z=height, frame=f'aruco_{marker}', timeout=30)
    state = flight.telemetry(frame_id=f'aruco_{marker}')
    error = math.hypot(state.x, state.y)
    log.event('visit', marker, error)


def main():
    log = Log(log_path)
    flight = Flight(simulation_only=not real)
    with flight.mission():                    # always ends with a landing
        flight.go(z=1.2, frame='body', auto_arm=True)
        time.sleep(2)
        for marker in [0, 3, 6, 5]:           # state ROUTE
            if out_of_time():
                rospy.logwarn('Time budget exhausted: skipping the rest')
                break
            visit(flight, log, marker, 1.2)
        # state LAND_ON_MARKER would go here; leaving the block lands anyway
    log.event('finished')


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

• Задание

Допишите в каркас состояние посадки на маркер 7 (используйте логику `aruco_landing.py`, глава 20) и запись события `land` с ошибкой в метрах. **Проверка:** в журнале есть строка `land,7,время,ошибка`; при потере маркера выполняется посадка на месте.

27.3 Автосудья: тренируйтесь с известным счётом

Чтобы не гадать, сколько баллов получила программа, напишите «судью», который считает счёт по журналу событий и правилам. Файл `score_demo.py` (стандартная библиотека Python, без ROS) читает CSV `event,marker,time_s,error_m` и считает баллы по правилам модуля «обход маркеров» из главы 20: +10 за посещение в нужном порядке с ошибкой не более 0.2 м, +5 при ошибке до 0.4 м, −10 за нарушение порядка, +20 за посадку в круге 0.2 м (или +10 до 0.4 м), −20 за столкновение; события после 180 с не оцениваются.

```
#!/usr/bin/env python3
"""Offline judge for the 'marker tour' training task. Standard library only.

Input CSV columns:  event,marker,time_s,error_m
  event = visit | land | collision
Rules (training rules of this book, not an official regulation):
  visit in the required order: 10 points if error <= 0.2 m, 5 points if <= 0.4 m
  visit out of order: -10;   landing: 20 points if <= 0.2 m, 10 if <= 0.4 m
  collision: -20 each;       events after the time limit (180 s) score nothing
"""
import csv
import sys
```

```
REQUIRED = [0, 3, 6, 5]          # order of the visits
LAND_MARKER = 7
TIME_LIMIT = 180.0


def points_for_error(error, full, half):
    if error <= 0.2:
        return full
    if error <= 0.4:
        return half
    return 0


def judge(rows):
    score, expected, log = 0, 0, []
    for event, marker, time_s, error in rows:
        if time_s > TIME_LIMIT:
            log.append(f'{event} {marker}: after time limit, 0')
            continue
        if event == 'visit':
            if expected < len(REQUIRED) and marker == REQUIRED[expected]:
                p = points_for_error(error, 10, 5)
                expected += 1
            else:
                p = -10
            log.append(f'visit {marker}: {p:+d}')
            score += p
        elif event == 'land':
            p = points_for_error(error, 20, 10) if marker == LAND_MARKER else 0
            log.append(f'land {marker}: {p:+d}')
            score += p
        elif event == 'collision':
            score -= 20
            log.append('collision: -20')
        else:
            raise ValueError('unknown event ' + event)
    return score, log


def load(path):
    with open(path, newline='', encoding='utf-8') as handle:
        reader = csv.DictReader(handle)
        return [(r['event'], int(r['marker']), float(r['time_s']), float(r['error_m']))
                for r in reader]


if __name__ == '__main__':
    path = sys.argv[1] if len(sys.argv) > 1 else 'sample_flight_log.csv'
    total, lines = judge(load(path))
    print('\n'.join(lines))
    print('TOTAL:', total)
```

Пример журнала `sample_flight_log.csv`:

```
event,marker,time_s,error_m
visit,0,22.5,0.12
visit,3,48.0,0.18
visit,5,71.2,0.09
visit,6,95.4,0.35
land,7,121.0,0.15
```

Результат запуска `python3 score_demo.py`:

```
visit 0: +10
visit 3: +10
visit 5: -10
visit 6: +5
land 7: +20
TOTAL: 35
```

Маркер 5 посещён раньше маркера 6, хотя нужный порядок 0, 3, 6, 5: нарушение порядка стоит -10, а следующее посещение засчитано по своей очереди.

Практика. Свой автосудья

Измените правила в `score_demo.py` под собственное задание (другой порядок, другие баллы, штраф за время). Составьте три журнала: идеальный, с нарушением порядка и с превышением времени, и убедитесь, что счёт совпадает с ручным расчётом. **Проверка:** для идеального журнала счёт максимален; для остальных – ожидаемо ниже; правила записаны в шапке файла.

27.4 Модуль 1. Подготовка и диагностика аппарата

Условие. Дан реальный (или предварительно настроенный учебный) аппарат с одной скрытой неисправностью. За 25 минут необходимо: провести предполётную проверку (глава 25); включить подсистемы для задания «полёт по карте маркеров» (флаги `aruco`, `aruco_map`, `map`, глава 9); загрузить выданную карту; подтвердить работоспособность (`selfcheck.py`, фрейм `aruco_map`, отладочное изображение); устранить неисправность; выполнить контрольный взлёт и посадку.

Оценка (100 баллов). Предполётная проверка по листу – 20; безопасность (пульт, disarm, батарея) – 20; корректная конфигурация (по файлам) – 25; диагностика и устранение неисправности – 25; контрольный полёт – 10. Штрафы: полёт с непроверенным аппаратом – обнуление модуля; невосстановленная резервная копия при ошибке файла – -10.

Решение. Порядок: лист проверки $\rightarrow$ резервные копии файлов $\rightarrow$ правка флагов $\rightarrow$ перезапуск службы $\rightarrow$ проверка `selfcheck.py` $\rightarrow$ карта и отладочное изображение $\rightarrow$ контрольный полёт. Неисправность ищите разделением ручного и автономного режимов.

• Задание

Напишите пошаговый протокол выполнения модуля 1 с контрольными точками времени (например, через 5, 10, 15, 20 минут). **Проверка:** в протоколе есть резервная копия до правки, проверка после перезапуска и итоговый полёт; ни одно действие не требует полёта с непроверенным аппаратом.

27.5 Модуль 2. Обход маркеров с чтением QR-кода

Условие. В зоне наклеены маркеры карты 2×4 (`genmap.py 0.33 2 4 1 1 0`). На стойке в начале поля закреплён QR-код с текстом вида `0-3-6-5`: порядок посещения маркеров. Задача: взлететь, прочитать QR-код, посетить маркеры в прочитанном порядке (зависание 2 с над каждым), сесть на маркер 7, сохранить журнал.

Оценка (100 баллов). Корректное чтение кода – 15; каждая точка в нужном порядке с ошибкой до 0.2 м – по 15 (до 0.4 м – по 7); посадка до 0.2 м – 25; журнал событий – 10. Штрафы: нарушение порядка -10; столкновение -20; превышение 4 минут – события не оцениваются.

Ход решения.

1. Проверьте распознавание QR-кода *офлайн* на фото программой `qr_reader.py --image`. Пример: файл с кодом «ZONE-B» читается как `QR: ZONE-B`.
2. В ROS-режиме эта же программа подписывается на камеру и печатает каждый новый текст один раз.
3. Разберите строку `0-3-6-5` в список целых и передайте его в обход маркеров (`aruco_tour.py`).
4. Обработайте ошибки: код не прочитан за 10 секунд – используйте запасной порядок и запишите событие.

```
#!/usr/bin/env python3
"""Read QR codes. Two modes:
  python3 qr_reader.py --image photo.png      (offline, OpenCV only)
  python3 qr_reader.py                        (ROS: subscribes to the main camera)
Prints each new text once together with the time of the frame.
"""
import argparse
import cv2


def decode(frame, detector):
    text, points, _ = detector.detectAndDecode(frame)
    return text if text else None


def run_image(path):
    frame = cv2.imread(path)
    if frame is None:
        raise SystemExit('Could not read ' + path)
    text = decode(frame, cv2.QRCodeDetector())
    print('QR:', text if text else 'not found')


def run_ros():
    import rospy
    from cv_bridge import CvBridge
    from sensor_msgs.msg import Image

    rospy.init_node('course_qr_reader')
    bridge = CvBridge()
    detector = cv2.QRCodeDetector()
    seen = set()
```

```
    def on_image(message):
        frame = bridge.imgmsg_to_cv2(message, 'bgr8')
        text = decode(frame, detector)
        if text and text not in seen:
            seen.add(text)
            rospy.loginfo('QR "%s" at %.2f', text, message.header.stamp.to_sec())

    rospy.Subscriber('main_camera/image_raw_throttled', Image, on_image,
                     queue_size=1, buff_size=2**24)
    rospy.spin()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--image')
    args = parser.parse_args()
    run_image(args.image) if args.image else run_ros()
```

Разбор строки порядка:

```
text = '0-3-6-5'
order = [int(part) for part in text.split('-')]
if not order or any(not 0 <= i <= 249 for i in order):
    raise ValueError('bad route in QR code')
print(order)      # [0, 3, 6, 5]
```

Задание: «Маршрут из QR»

Реализуйте модуль целиком: чтение QR (ROS), построение маршрута, обход, посадка, журнал для автосудьи. Проверьте в симуляторе (QR-плакат можно разместить как текстурированную модель) и офлайн на фото. **Проверка:** три повтора подряд без нарушения порядка; журнал читается `score_demo.py`; при нечитаемом коде программа безопасно использует запасной порядок и записывает это событие.

27.6 Модуль 3. Цветные зоны

Условие. На поле размещены три цветные площадки (красная, зелёная, синяя) в неизвестных заранее местах. Необходимо облететь поле, определить координаты центров площадок в системе `aruco_map` и сесть на зелёную. Результат – файл `zones.csv` со строками `цвет,x,y`.

Оценка. Каждая площадка найдена: по 15 баллов (ошибка координаты до 0.2 м); посадка на зелёную: 30; отчёт: 15; штрафы за посадку на неверный цвет и за выход из зоны.

Ход решения. Детектор цвета (глава 21) даёт центр площадки на кадре; чтобы получить координаты на поле, дрон наводится на площадку (центрирование по ошибке) и по телеметрии во фрейме `aruco_map` записывает свою позицию в момент, когда площадка в центре кадра. Для трёх цветов нужны три набора порогов HSV; их подбирают по сохранённым кадрам и проверяют на наборе сцен (глава 22). Порядок обхода – «змейка» (глава 15) с шагом по формуле покрытия.

• **Задание**

Рассчитайте шаг «змейки» для обследования поля 3×2 м на высоте 1.5 м при угле обзора 60° и перекрытии 30% и время облёта при скорости 0.4 м/с без учёта разгона. **Проверка:**

$W = 2 \cdot 1.5 \tan 30^\circ = 1.73$ м; шаг ≤ 1.21 м; для ширины 2 м нужны две строки длиной по 3 м и один переход 1.2 м: путь около 7.2 м, время около 18 с.

27.7 Модуль 4. Работа с отказом

Условие. Во время полёта по маркерам один из маркеров намеренно закрывают. Программа должна: обнаружить потерю маркера по NaN в телеметрии; не продолжать полёт по устаревшим данным; записать событие `marker_lost`; выполнить безопасную посадку на месте и сохранить журнал.

Оценка. Событие обнаружено не позднее чем через 2 с – 30; нет движения по устаревшим данным – 30; посадка без повреждения – 25; журнал – 15.

Ход решения. Добавьте в каркас проверку конечности координат при каждом шаге (как в `aruco_landing.py`), счётчик последовательных неудач и состояние LAND по таймеру. Проверьте на симуляторе, убирая маркер из мира. Помните о свежести данных (глава 24).

• Задание

Опишите таблицу переходов автомата миссии для модуля 4: состояния, события, действия. **Проверка:** из состояния ROUTE при событии `marker_lost` переход в LAND с записью события; из LAND по `disarmed` – в DONE.

27.8 Модуль 5. Комплексная задача

Условие. Полигон: карта маркеров 2×4, QR-код с порядком, три цветные зоны, коробка-препятствие. За 10 минут: подготовить аппарат (модуль 1, 20 мин отдельно), прочитать порядок, посетить маркеры, найти зелёную зону, зафиксировать координаты всех зон и сесть на маркер 7. Результат: журнал событий, `zones.csv`, краткий отчёт (что выполнено, что нет, причины).

Оценка (200 баллов). Подготовка и безопасность – 40; маркеры и порядок – 60; QR – 20; зоны – 40; посадка – 25; отчёт и журнал – 15. Штрафы: столкновение -20; выход из зоны -15; нарушение безопасности – обнуление.

Тактика. Составьте минимальную программу, которая набирает баллы при посадке на маркер 7. Затем добавляйте модули по убыванию надёжности: порядок маркеров, QR, зоны. Проверяйте после каждого шага; сохраняйте рабочую версию. Оставьте время на отчёт.

Комплексное задание

Выполните модуль 5 на симуляторе (или на аппарате в присутствии пилота). Ведите хронометраж и журнал. **Проверка:** результат оценивается автосудьёй и самооценкой по таблице; отчёт содержит причины каждого потерянного балла и план улучшения.

27.9 Чек-лист участника и типичные ошибки

Ошибка	Как избежать
Начинают с сложной программы, не проверив взлёт-посадку	Минимальная рабочая версия и сохранение после каждого успеха
Не проверяют NaN в координатах	Проверка `math.isfinite` перед использованием позиции
Неверный фрейм (`map` вместо `body`)	Таблица целей до запуска: фрейм, координаты, ожидаемое положение
Магические числа в коде	Параметры в начале файла или параметры ROS; единицы подписаны
Нет аварийного завершения	Контекст `mission()`, бюджет времени, состояние LAND
Не сохранён журнал	Журнал пишется по ходу с `flush()`; имя файла содержит номер попытки
Потеря времени на настройке	Резервные копии и протокол подготовки; повторяемые команды в файле
Нарушение безопасности	Пилот с пультом, disarm, отсутствие людей в зоне
Изменили несколько вещей сразу	Одно изменение за раз и контрольная проверка

Итоговая самооценка

Составьте собственную схему оценки для комплексного задания (критерии, баллы, штрафы), проведите по ней два прохода (до и после доработки) и сравните результат. **Проверка:** улучшение объяснено конкретными изменениями в программе или в порядке работы.

Итоги и самопроверка

- Конкурсное задание состоит из модулей с временем, критериями и штрафами; часть критериев измеряется, часть оценивает эксперт; безопасность оценивается отдельно.
- Решайте от простого: минимальная рабочая программа, затем наращивание; конечный автомат, бюджет времени, безопасное завершение, журнал событий.
- Автосудья по журналу позволяет тренироваться, зная счёт.
- Типовые модули: подготовка аппарата, обход маркеров, QR, цветные зоны, обследование поля, работа с отказом, комплексная задача.
- Официальные регламенты соревнований имеют приоритет над учебными схемами этой книги.

Проверьте себя: 1) Почему надёжность выгоднее скорости при штрафах? 2) Что такое минимальная рабочая программа? 3) Зачем нужен автосудья? 4) Что обязательно должно быть в любой конкурсной программе на случай ошибки?
Ответы: 1) штраф за ошибку часто превышает выигрыш от скорости; 2) короткая программа, которая взлетает, делает одну вещь и безопасно садится; 3) чтобы знать счёт и проверять улучшения; 4) безопасное завершение (посадка) и журнал.

ГЛАВА 28

Клевер 4 и Клевер 5: перенос навыков

Цели главы

- отделять подтверждённое документацией от предположений;
- составить матрицу совместимости для своего комплекта.

Нужно знать заранее: главы 2.

28.1 Что подтверждено документацией

В обновлённом введении COEX Клевер 5 показан с Raspberry Pi 5 и полётным контроллером MicoAirH743v2 под PX4. В материалах производителя 2026 года новая линейка связана с ROS 2. При этом раздел ROS1 и описание классической симуляции продолжают существовать: название аппаратного поколения само по себе не определяет точную версию ПО на вашем комплекте.



Рис. 28.1. Клевер 5 КОД на иллюстрации обновлённой документации COEX. CC BY-NC-SA 4.0; это фото аппаратного комплекта, а не модель Gazebo из этой книги.

Источник: COEX, обновлённое введение и аппаратная база. Проверено 11.10.2026.

Источник: Клевер COEX, новая линейка и ROS 2. Проверено 11.10.2026.

• Задание

Выпишите два факта о Клевере 5, подтверждённых документацией (глава 2), и одно предположение, которое нужно проверить. **Проверка:** факты и предположение разделены, у фактов есть источник.

28.2 Что переносится сразу, а что проверяется

Элемент	Переносимый навык	Проверка нового стенда
Миссия по точкам	Геометрия маршрута, таймауты, состояния	Названия и типы API
Камера	Маски, контуры, метрики	Топик, калибровка, ориентация
ArUco	ID, карта и геометрия	VPE, TF, параметры PX4
ROS	Узлы, сообщения, сервисы	ROS 1 или ROS 2; в ROS 2 – настройки надёжности доставки (QoS) и способ выполнения callback (executor)
Gazebo	Сцена, visual/collision	Версия Gazebo и тип плагинов
Модель аппарата	Масса, инерция, сенсоры	Соответствие конкретному Клеверу 5

`rospy`, `roslaunch` и catkin – инструменты ROS 1. В ROS 2 обычно используются библиотека `rclpy`, команда `ros2 launch` и система сборки colcon (аналог catkin). Замена импортов не портирует весь проект: ожидание сервисов, работа callback и QoS требуют отдельной реализации. Современный Gazebo также использует другую архитектуру плагинов, чем Gazebo Classic.

• Задание

Разделите на «переносится почти без изменений» и «проверяется заново»: алгоритм маршрута; команда установки; идея детектора; имя сервиса. **Проверка:** алгоритм и идея детектора переносятся; установка и имя сервиса проверяются.

28.3 Проверка имеющегося комплекта

На учебном стенде выясните следующие данные до установки пакетов:

1. Полное название модели и комплектации: КОД, ДЕВ или ФПВ (названия комплектаций производителя). Не предполагаем одинаковый набор датчиков и задач.
2. Версия образа бортового компьютера, ОС и ROS. В ROS 1 проверяют `rosversion -d`; в ROS 2 – `printenv ROS_DISTRO` и инструменты `ros2`.
3. Версия PX4, тип контроллера и применённые параметры оценивателя.
4. Список сервисов управления и их определения. Отдельно – камера, дальномер, TF и интерфейс Blockly.
5. Наличие официального пакета симуляции для этой модели: репозиторий с исходным кодом, номер версии (commit в системе Git), команда запуска и описание динамики.

По просмотренным источникам нельзя обещать готовую универсальную команду запуска точной модели Клевера 5 на ROS 2. В этом учебнике такая команда намеренно не выдумана. Если ваш комплект работает с совместимым ROS 1 API Clover, миссионный код может потребовать лишь адаптации конфигурации; если это ROS 2 API, нужна отдельная ROS-обвязка.

• Задание

Составьте список из трёх вопросов к поставщику комплекта о его программной платформе. **Проверка:** есть вопрос о версии ROS, о версии автопилота и об интерфейсе команд.

28.4 Практичный путь переноса

1. Отладьте алгоритм на проверенном классическом стенде: последовательность, координаты, отказ и метрики.
2. Выделите чистую часть Python без ROS: генератор пути, обработку изображения, расчёт ошибок.
3. Для новой платформы напишите адаптер управления и телеметрии с теми же понятиями: назначить цель, получить состояние, посадить.
4. Проверьте адаптер отдельно коротким опытом на официальной симуляции нового стека, если она доступна.
5. Затем повторите исходные задания и сравните поведение, а не только отсутствие ошибок запуска.

Код и динамика – разные уровни совместимости

Совпадение имён сервисов не доказывает совпадение массы, тяги, задержек сенсоров и ограничений аппарата. Модель Клевера 4 подходит для обучения логике миссии Клевера 5, но не является доказательством точной динамической верификации нового дрона. Реальные испытания требуют отдельного стенда и процедур организации.

Практика. Матрица совместимости

Составьте таблицу «Клевер – ROS – PX4 – API – симулятор – датчики». Отметьте подтверждённые и непроверенные данные. Выберите переносимое задание и перечислите изменения его ROS-обвязки.

Итоги и самопроверка

- Клевер 5 – отдельная платформа; API и установку проверяют заново.
- Алгоритмы переносятся легче, чем конфигурации.

Проверьте себя: 1) Что нужно уточнить перед переносом программы на Клевер 5? 2) Что переносится почти без изменений?
Ответы: 1) версию ПО и интерфейсы комплекта; 2) идеи алгоритмов маршрута и обработки изображения.

ГЛАВА 29

Рабочая тетрадь и подсказки к решениям

Цели главы

- закрепить материал заданиями шести блоков;
- сверить решения с подсказками;
- выполнить итоговое задание.

Нужно знать заранее: соответствующих глав.

29.1 Как выполнять задания

Для каждого задания запишите прогноз до запуска и наблюдение после. Если они различаются, объясните причину. Основной уровень охватывает материал глав; дополнительные пункты требуют обобщения или самостоятельной постановки опыта.

Ответы находятся во второй части главы. Сначала следует решить задание самостоятельно. Вопрос «почему?» часто важнее правильного числа: он показывает, переносится ли решение на другой маршрут.

29.2 Блок A. Архитектура и ROS

A1. В проект добавили запись камеры. Нужно ли обязательно менять детектор? Нарисуйте новый подписчик в графе.
A2. Выберите механизм: непрерывные изображения, назначение цели, размер окна фильтра. Объясните каждое решение.
A3. В списке есть топик, но echo молчит. Предложите две причины и команды проверки.
A4. Publisher использует Float64, Subscriber – String. Исправит ли ситуацию одинаковое имя топика?
A5. Программа фильтра читает window только при старте. Вы изменили rosparam. Почему поведение не изменилось?
A6. В Python указан `debug`. Нода называется course_red_detector. Какое полное имя искать?

29.3 Блок B. Python

B1. Что выведет программа? Сначала предскажите, затем проверьте.

```
height = 1.0
height = height + 0.2
print(height > 1.1)
```

B2. Сколько итераций у `range(3)`? Какие значения принимает индекс?
B3. Почему сравнение `height == 1.0` плохо подходит для прибытия по измерению?
B4. Функция принимает скорость. Какие проверки сделать перед делением расстояния на скорость?

B5. Напишите функцию, возвращающую среднее списка. Какое решение принять для пустого списка?
B6. Для samples $[0.9, 1.0, 1.1]$ найдите среднее и максимальное абсолютное отклонение от 1 м.

29.4 Блок C. Координаты

C1. Дрон в map $(2, 1, 1)$, yaw=0. Что означает body $(1, 0, 0)$? А map $(1, 0, 0)$?
C2. Из той же точки при yaw=90° выполнен шаг body $(1, 0, 0)$. Найдите ожидаемые XYZ.
C3. Для остатка $(0.1, 0.1, 0.1)$ найдите расстояние. Пройдёт ли проверка допуска 0.15 м?
C4. Почему сумма смещений замкнутого маршрута должна быть нулевой при одной неизменной ориентации?
C5. Квадрат со стороной 0.8 м обходится при идеальной скорости 0.4 м/с. Найдите длину и нижнюю оценку времени только горизонтального обхода.
C6. Дополнительное: найдите результат body $(0.5, 0.2, 0)$ при yaw=90° из точки $(1, 2, 1)$.

29.5 Блок D. Логика миссии

D1. Ответ navigate.success true получен сразу. Что ещё нужно наблюдать перед следующей целью?
D2. В цикле ожидания нет таймаута. Назовите две причины, по которым цикл может не закончиться.
D3. После остановки Python модель остаётся в воздухе. Почему это не обязательно ошибка Gazebo?
D4. Если движение задано сервисом set_velocity (команда линейной скорости, см. справочник), а цель пропала, почему недостаточно перестать отправлять новые команды?
D5. Миссия courier пишет «доставлено». Что она действительно доказала и чего этот журнал сам по себе не доказывает?
D6. Дополнительное: предложите состояния для «взлететь – найти цель – сделать фото – вернуться – сесть» и переход при пропаже цели.

29.6 Блок E. Симулятор и эксперимент

E1. В SDF есть visual, но нет collision. Будет ли внешний вид достаточным доказательством физического препятствия?
E2. У коробки высота 0.8 м и начало в центре. Какой Z центра помещает основание на пол Z=0?
E3. При RTF=0.4 прошло 8 симуляционных секунд. Сколько примерно реальных?
E4. После spawn коробка исчезла при перезапуске. Как сделать её частью воспроизводимой сцены?
E5. В одном опыте меняли скорость, высоту и допуск. Можно ли выделить влияние скорости по единственному сравнению?
E6. Дополнительное: составьте план трёх повторов для двух скоростей, указав метрику и одинаковые условия.

29.7 Блок F. Камера

F1. У изображения shape=(480,640,3). Назовите ширину и высоту.
F2. Пиксель BGR=(0,0,255). Какой основной цвет?

F3. Найденный центр в кадре шириной 640: u=480. Найдите нормированную ошибку ex.
F4. Два красных объекта оказались по краям кадра. Почему общий центр маски может быть неверной целью?
F5. Ошибка детектора NaN. Почему её нельзя заменить нулём и считать центрированием?
F6. Дополнительное: придумайте три случая, когда корректное обнаружение по цвету не означает правильную посадочную площадку.

29.8 Подсказки и ответы: архитектура

A1. Запись подключается к существующему топику камеры. Детектор менять не обязательно, если его интерфейс не должен измениться. **A2.** Изображения – топик; цель – сервис данного API; окно фильтра – параметр. **A3.** Издатель не работает или не отправляет данные, либо выбран другой топик/тип/окружение. Проверьте info, hz, процесс и master. **A4.** Нет: тип должен совпадать. **A5.** Уже созданный фильтр использует сохранённую настройку; перезапустите или специально реализуйте перечитывание. **A6.** `/course_red_detector/debug`.

29.9 Подсказки и ответы: Python

B1. True: после прибавления height примерно 1.2. Не делайте из этого примера вывод о точном представлении всех десятичных дробей. **B2.** Три итерации, индексы 0, 1, 2. **B3.** Измерения имеют шум и погрешность, float не обязан точно совпасть; используйте допуск. **B4.** Скорость конечна и положительна; расстояние должно быть корректным для выбранной задачи. **B5.** Для пустого списка разумно выдать явную ошибку, а не делить на ноль. **B6.** Среднее 1.0, максимальное отклонение около 0.1 м.

Пример решения B5:

```
def mean(values):
    if not values:
        raise ValueError('Empty list')
    return sum(values) / len(values)
```

Дополнительная проверка – конечность всех элементов, если список содержит измерения. Решение зависит от договора: либо явно отбрасывайте неверные данные и сообщайте их количество, либо прекращайте расчёт. Молчаливое превращение неизвестного измерения в ноль искажает результат.

29.10 Подсказки и ответы: координаты

C1. Body – смещение к $(3, 1, 1)$; map – абсолютная точка $(1, 0, 0)$, включая высоту ноль. **C2.** Ожидается $(2, 2, 1)$. **C3.** $\sqrt{0.03} \approx 0.173$ м, поэтому нет. **C4.** Возвращение в исходную заданную точку требует нулевого суммарного смещения; при поворотах сначала переводите смещения в общий фрейм. **C5.** Периметр 3.2 м, идеальная оценка 8 с без разгона, торможения, взлёта и посадки. **C6.** $(0.8, 2.5, 1)$.

29.11 Подсказки и ответы: миссия

D1. Остаточное расстояние, корректность позиции, связь и режим. Для съёмки добавьте скорость и стабилизацию. **D2.** Пауза симулятора, потеря позиционирования, недостижимая

цель или слишком строгий допуск. **D3.** Компоненты управления могут продолжать удерживать последнюю цель; завершение клиентского процесса не равно посадке. **D4.** Последняя заданная скорость может продолжать действовать: нужна явная смена команды и логика завершения. **D5.** Событие показывает, что программа дошла до соответствующей строки после своего критерия прибытия. Оно не доказывает работу физического захвата и не заменяет измерения точности. **D6.** Возможны CHECK, TAKEOFF, SEARCH, PHOTO, RETURN, LAND, DONE; ветка пропажи зависит от наличия достоверной позиции и выбранной стратегии.

29.12 Подсказки и ответы: симулятор

E1. Нет: visual отвечает за отображение; контакты требуют collision и подходящей физической конфигурации. **E2.** Z=0.4 м. **E3.** Около $8/0.4 = 20$ с. **E4.** Сохранить модель в world либо воспроизводимо выполнять spawn с документированными координатами. **E5.** Нет: изменились несколько факторов, причина различия неоднозначна. **E6.** Один мир и маршрут, одинаковая высота и допуск, меняется только скорость; по три повтора, отдельные журналы и метрика, например время и ошибка возврата. Для оценки разброса трёх повторов мало, но для первого сравнения это полезный минимум.

29.13 Подсказки и ответы: камера

F1. Ширина 640, высота 480. **F2.** Красный. **F3.** $(480 - 320)/320 = 0.5$. **F4.** Среднее двух областей может лежать вне обеих. **F5.** NaN обозначает отсутствующее или непригодное измерение, ноль – найденный центр кадра. **F6.** Красная стена, красная коробка, блик/текстура неподходящей формы или две похожие станции. Для правильной цели нужны дополнительные признаки и правило выбора.

29.14 Итоговое задание: собственный учебный полигон

Сделайте мир с тремя станциями и препятствием. Программа должна посетить станции, сохранить событие в каждой, вернуться и сесть. Подготовьте схему координат, параметры запуска, исходник, журнал и одно исследование ошибки. Камера – дополнительный уровень: сохраните кадр в станции и подтвердите его свежесть.

Результат сопровождается ответами на вопросы: где находится источник позиции, какие интерфейсы использует миссия, как подтверждается прибытие, что происходит при отказе и какие результаты относятся только к модели. Файлы должны позволять независимо повторить хотя бы одну короткую часть опыта.

Что считается самостоятельной работой

Не количество строк и не скорость печати. Вы можете объяснить цель каждой части, изменить один параметр с прогнозом, найти причину одного отказа и показать измерение результата. Использование готового помощника допустимо, если понимаете его интерфейс и ограничения.

Итоги и самопроверка

- Сначала решите самостоятельно, потом смотрите ответ.

Проверьте себя: 1) Какой блок заданий вам даётся хуже? 2) К какой главе стоит вер-

нуться?
Ответы: личный ответ: вернитесь к главе этого блока.

ГЛАВА А

Справочник для рабочего стола

A.1 API, который нужен в книге

Уточните определения в своей установке через `rossrv show`. Таблица – краткий ориентир, а не полная спецификация всех полей.

Сервис	Тип	Назначение
`get_telemetry`	`clover/GetTelemetry`	Положение, скорости, состояние в заданном фрейме
`navigate`	`clover/Navigate`	Назначить точку и скорость движения цели
`navigate_global`	`clover/NavigateGlobal`	Географическая цель при доступном GPS
`set_position`	`clover/SetPosition`	Поток позиционных целей
`set_velocity`	`clover/SetVelocity`	Команда линейной скорости
`set_yaw`	`clover/SetYaw`	Изменить ориентацию по yaw
`land`	`std_srvs/Trigger`	Начать посадку

Полётные прокси возвращают `success` и `message`; при отказе анализируйте сообщение. Телеметрия не предоставляет такое же поле `success`: проверяйте связь и доступность данных. Для неизменного yaw в навигации используют `float('nan')`. В других числовых полях NaN допустим только там, где это прямо описано API.

A.2 Быстрая диагностика

Симптом	Возможная причина	Что проверить первым
`No module named rospy`	Не активирован ROS, неверный Python	`source` и `which python3` в Ubuntu
Не найден пакет Clover	Не активирован workspace или нет сборки	`rospack find clover`, `catkin_make`
Ожидание сервиса	Launch не работает, неверный namespace	`rosservice list`, терминал launch
`connected=false`	Нет связи MAVROS с PX4	`/mavros/state`, вывод SITL
Координаты NaN	Нет позиции или TF	Источник позиции, `frame_id`, `/tf`
Взлёт отклонён	PX4 preflight check или неподходящий режим	`response.message`, QGC, консоль PX4

Симптом	Возможная причина	Что проверить первым
Миссия «висит»	Пауза мира или отсутствует цель	Play, `/clock`, deadline
Blockly пуст/не запускается	Страница или backend не настроены	Web-сервер, `blocks`, rosbridge
Мир не изменился	Загрузка другого world	`course_world` в локальном launcher
Проходит сквозь коробку	У модели только visual	`collision`, положение и размер
Карта видна, полёт неверный	Несогласованные размер/позиции/оценка	Карта ROS, world, VPE и TF
Pos на полу, armed=true	Нет автоматического disarm	`COM_DISARM_LAND` и mode
`cc1plus: Killed`	Не хватает памяти для сборки	Снизить jobs до `-j2` или `-j1`

Это диагностические гипотезы. Проверяйте факт командой или логом, прежде чем менять конфигурацию. Записывайте полную первую ошибку, а не только последнюю строку вывода.

A.3 Минимальная памятка терминала

```
# Every new Ubuntu terminal
source /opt/ros/noetic/setup.bash
source "$HOME/catkin_ws/devel/setup.bash"

# Terminal A: simulator (only one instance)
roslaunch clover_simulation simulator.launch

# Terminal B: inspect, then run a single mission
cd "$HOME/clover_course_files/examples"
python3 inspect_state.py
python3 first_flight.py

# Request landing if the API is still available
rosservice call /land '{}'

# Confirm state; landing request alone is not completion
rosservice call /get_telemetry "{frame_id: 'map'}"
```

A.4 Глоссарий

Armed / disarmed
Моторы включены (аппарат «взведён» к полёту) / выключены.

Callback
Функция, которую библиотека вызывает сама при получении сообщения или запроса.

catkin
Система сборки пакетов ROS 1; рабочее пространство `catkin_ws`.

Launch-файл
Текстовый файл (XML), перечисляющий ноды, которые нужно запустить, и их параметры; запускается командой `roslaunch`.

Нода
Работающая программа, участвующая в обмене средствами ROS.

Топик
Именованный поток сообщений одного типа: издатели отправляют, подписчики получают.

Сервис
Обмен «запрос–ответ» между двумя нодами.

Параметр
Именованная настройка, хранящаяся в ROS.

Пакет
Каталог с программами, launch-файлами и описанием зависимостей.

Фрейм (система координат)
Начало отсчёта и направления осей, в которых заданы координаты; в Клевере `map`, `body`, `navigate_target`, `aruco_map`.

Yaw, pitch, roll
Рыскание (поворот вокруг вертикали), тангаж и крен; в ROS измеряются в радианах.

Телеметрия
Данные о состоянии аппарата: связь, режим, armed, положение, скорость.

Миссия
Программа, задающая последовательность целей и проверок полёта.

Виртуальная машина (VM)
Программа, запускающая в окне «второй компьютер» со своей операционной системой.

HSV
Описание цвета через оттенок, насыщенность и яркость; удобно для выделения цвета.

Маска
Изображение, где пиксели, прошедшие проверку, отмечены белым, остальные – чёрным.

Метрика, RMSE
Числовая величина для сравнения результатов; среднеквадратичная ошибка.

Конечный автомат
Модель, в которой система находится в одном из состояний, а события переводят её между ними.

SITL
Автопилот как программа на компьютере вместо аппаратного полётного контроллера.

OFFBOARD
Режим PX4, принимающий задачи движения от внешней программы.

Setpoint
Текущее желаемое состояние, по которому работает регулятор.

TF / frame
Система координат и её преобразования к другим системам во времени.

VPE
Передача внешней оценки позиции автопилоту, например по визуальной карте.

Optical flow
Измерение движения изображения для оценки движения аппарата; масштаб связан с расстоянием и геометрией камеры.

RTF
Отношение скорости симуляционного времени к реальному.

URDF / xacro
: Описание робота и шаблонный язык для его построения.

SDF / world
: Описание моделей, сенсоров, физики и мира симулятора.

MAVLink / MAVROS
: Протокол обмена с автопилотом и его интерфейс в ROS.

Collision
: Геометрия контактов в физическом движке.

Namespace
: Префикс имён узлов, топиков и сервисов, полезный при нескольких аппаратах.

A.5 Что отложено за пределы основного содержания

Рой требует отдельных PX4-инстансов, уникальных MAVLink ID, портов, namespace и TF для каждого аппарата. Несколько моделей в сцене не создают эту инфраструктуру автоматически. Обучение нейросетевых агентов дополнительно требует определённой среды, наблюдений, наград и корректного сброса состояния; сначала нужен воспроизводимый базовый полёт. Цифровой двойник конкретного Клевера 5 требует проверенной модели его динамики и сенсоров.

A.6 Локальный launcher: устройство генератора

Полный исходник – `examples/make_course_launch.py`. Он работает в Ubuntu после создания пакета `clover_course`:

1. Через `rospack` находит установленный simulator.launch.
2. Проверяет вложенные launch-файлы Gazebo и Clover.
3. Добавляет аргумент выбора мира, затем передаёт его в Gazebo.
4. Передаёт флаги ArUco и Blockly во вложенный Clover launcher.
5. Сохраняет результат в отдельный пакет, сохраняя исходный файл.

При изменении структуры upstream-файла скрипт выдаёт ошибку для ручной проверки. На Windows без ROS этот генератор не запускается.

Источники и атрибуция

Дата проверки ссылок: **11 октября 2026 года**. Старый домен `clover.coex.tech` при исследовании перенаправлял на `klever-doc.tech/ROS1`. Документация, ветки GitHub и ссылки загрузок могут обновляться; сохраняйте паспорт именно вашего рабочего стенда.

Основные технические источники

1. COEX: обзор симулятора. Архитектура SITL и модель Клевера 4.
2. COEX: установка виртуальной машины. Импорт и состав образа.
3. CopterExpress: релизы clover_vm. Версии PX4/Clover готовых образов.
4. COEX: сборка на собственной машине. Нативный рецепт Ubuntu 20.04/Noetic/PX4 v1.12.3.
5. COEX: использование симулятора. Запуск, миры, камера, физика и производительность.
6. CopterExpress: simulator.launch. Проверка верхних и вложенных аргументов.
7. CopterExpress: spawn_drone.launch. Модель xacro и настройки сенсоров.
8. CopterExpress: clover.launch. Blocks, ArUco, ROS-компоненты.
9. COEX: программирование. Общий порядок работы с миссиями.
10. COEX: введение в ROS. Принципы нод, топиков, сервисов и имён.
11. ROS: официальные примеры rospy. Издатель, подписчик, сервис и параметры. Программы книги написаны отдельно для учебной лаборатории.
12. COEX: API автономного полёта. Сервисы, поля, системы координат.
13. COEX: примеры кода. Ожидание цели и посадки; учебный модуль книги расширяет эту идею проверками и таймаутами.
14. COEX: системы координат. Body, map, navigate_target и TF.
15. COEX: блочное программирование. Wait, Run, Stop, Land и Python.
16. COEX: навигация по карте ArUco. Формат карты, генерация и VPE.
17. COEX: камера и компьютерное зрение. CvBridge, callback и debug-топики.
18. Open Robotics: Gazebo Classic. Окончание поддержки и отличие нового Gazebo.
19. Gazebo Classic: компоненты. World, модели, сервер и графический клиент.
20. OpenCV: цветовые пространства. BGR, HSV и пороговые маски; установленная в Ubuntu версия OpenCV может быть старше текущей документации.
21. ROS PMC: последний sync Noetic. Дата завершения поддержки.
22. COEX: обновлённое введение. Аппаратная база Клевера 5.
23. Клевер COEX: новая линейка, март 2026. Направление ROS 2.

Иллюстрации из интернета

Снимки Gazebo, QGroundControl, rqt/TF и RTF взяты из раздела симулятора COEX; интерфейс и меню сохранения – из раздела Blockly; изображение Клевера 5 – из введения. Авторство: COEX / документация Клевер, лицензия CC BY-NC-SA 4.0. WebP преобразованы в PNG, содержание снимков сохранено. Оригинальные URL, имена файлов, даты и лицензии перечислены в `assets/sources.json`; реестр также включает неиспользованные снимки.

Дополнительные снимки редакции 3: блоки Clover Blocks (раздел Blockly) и ArUco-метки, отладочное изображение и оси карты (карта маркеров, маркеры) – COEX, CC BY-NC-SA 4.0. Фотографии дрона (Jason Mavrommatis), платы Raspberry Pi (Vishnu Mohanan) и экрана с кодом (Chris Ried) взяты с Unsplash по бесплатной лицензии Unsplash; адреса: XYrjl3j7smo, rZKdS0wI8Ks, ieic5Tq8YMk.

Рисунки `gen_*.png` (маршруты, кадр–маска–центр, пиксели, обратная связь) созданы программой `tools/make_figures.py` из кода примеров; кадр детектора искусственный.

Схемы архитектуры, координат, Blockly-последовательности, маршрута и автомата состояний по-

строены специально для пособия средствами TikZ.

Статус проверки

Материал сверялся по документации и исходникам, а не по установленному полётному стенду. Сборка PDF проверяется XeLaTeX, программы – синтаксически и отдельными локальными проверками, схемы SDF – как XML. Полные интеграционные полёты ROS/Gazebo, распознавание на живых кадрах и перенос на реальный аппарат требуют приёмки на конкретном стенде.