МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

Разработка системы управления траекторией движения БПЛА

МАГИСТЕРСКАЯ РАБОТА

студентки 2 курса 273 группы факультета КНиИТ
02.04.03 Математическое обеспечение и администрирование информационных систем
факультета КНиИТ
Ворониной Екатерины Юрьевны

Научный руководитель
доцент кафедры ИиП _____ Е. В. Кудрина

Заведующий кафедрой
к. ф.-м. н., доцент _____ М.В. Огнева

Саратов 2025

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 2
ВВЕДЕНИЕ 3
1. Теоретическая часть 6
1.1 История развития БПЛА 6
1.2 Архитектура бортового аппаратного устройства и системы управления БПЛА 7
1.3 Математическая модель роторно-винтового БПЛА 13
1.4 Алгоритмические основы управления движением БПЛА 23
1.5 Инструментальные средства для разработки систем управления 30
2. Практическая часть 32
2.1 Постановка задачи 32
2.2 Получение и обработка данных сенсоров квадрокоптера 33
2.3 Разработка динамической модели квадрокоптера 35
2.4 Контролируемое выполнение флипа квадрокоптером 42
2.5 Демонстрация работы 48
ЗАКЛЮЧЕНИЕ 54
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 55
Приложение А. Динамическая модель квадрокоптера 61
Приложение Б. iLQR-регулятор 63
Приложение В. Адаптивный планировщик целей 68
Приложение Г. Модуль генерации управляющих сигналов 85
Приложение Д. Виртуальный мир Gazebo 95
Приложение Ж. Сбор данных о процессе выполнения оптимизации траектории 100
Приложение И. Упрощенное решение задачи флипа 102

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

БПЛА — беспилотный летательный аппарат;

БА — беспилотная авиация;

АПК — аппаратно-программный комплекс;

САУ — система автоматического управления;

ГНСС — глобальная навигационная спутниковая система;

МК — микроконтроллер;

СТУ — система траекторного управления;

СК — система координат;

НПУ — наземный пункт управления;

БАУ — бортовая аппаратура управления;

НАУ — наземная аппаратура управления;

АПИК — аналитическая программно-инвариантная конструкция;

СВС — система воздушных сигналов;

БТС — беспилотная транспортная система;

IMU — инерциально-измерительный блок;

RPM — (от англ. Revolutions Per Minute) мера частоты вращения.

ВВЕДЕНИЕ

Развитие беспилотной авиации (БА) входит в область интересов государственной промышленной политики. В связи с утверждением Правительством РФ стратегии развития беспилотной авиации до 2030 года и на перспективу до 2035 года [1], в настоящее время сфера беспилотной авиации получает значительную финансовую поддержку со стороны государства.

Нарастающий интерес к БА — общемировая тенденция. Это происходит по ряду причин. Беспилотные летательные аппараты (БПЛА), как правило, значительно дешевле пилотируемых самолетов и вертолетов. Обучение оператора обходится дешевле, чем подготовка пилота. Отсутствие человека на борту исключает необходимость в системах жизнеобеспечения, снижает массу и габариты аппарата, а также позволяет увеличивать диапазон допустимых перегрузок. Кроме того, потери беспилотников не сопряжены с человеческими жертвами, что повышает общий уровень безопасности при эксплуатации.

БПЛА можно разделить на два типа: управляемые оператором и автономные, где функции пилотирования полностью возложены на систему управления. Человеческий мозг эффективно обрабатывает визуальные и другие сенсорные данные, позволяя распознавать объекты и ситуации. Однако по причине ограниченной скорости реакции, внимания и памяти человек не всегда обеспечивает надежное управление.

За последние годы автономные БПЛА значительно повысили свою надежность и выполняют все более широкий спектр задач в различных отраслях. Однако они все еще не используют весь свой физический потенциал. Большинство таких аппаратов летает с малой скоростью, близкой к зависанию, чтобы обеспечить стабильную работу алгоритмов восприятия и избегания препятствий [2, 3].

Совершенствование автономных систем управления в аспектах скорости и маневренности позволяет расширить возможности БПЛА: увеличить дальность и скорость полета, повысить способность избегать быстро движущиеся объекты, а также обеспечивать управление в сложных, ограниченных пространствах. Несмотря на прогресс, по универсальности и надежности автономные БПЛА все

еще уступают человеку, что требует проведения дальнейших исследований для сокращения этого разрыва.

Роторно-винтовые БПЛА (квадрокоптеры) являются одними из самых динамичных платформ в беспилотной авиации. Благодаря своей маневренности они применяются в самых разнообразных сценариях: от полетов в густых лесах и городской застройке до мониторинга пожароопасных зон, инспекции инфраструктуры, логистики и сельского хозяйства [2, 3].

Одним из наиболее сложных и показательных маневров для таких систем является акробатический флип — вращение на 360° по одной из осей. Такой маневр востребован не только в демонстрационных полетах, но и в прикладных задачах — например, при резком уклонении от препятствий, смене траектории в условиях ограниченного пространства, при калибровке систем навигации (например, при инверсии ориентации антенны или датчика), а также при контролируемом выходе из срыва устойчивости, вызванного внешними возмущениями или агрессивными маневрами. Успешное выполнение флипа требует четкого управления угловыми скоростями и тягой с учетом текущего состояния дрона и его динамических ограничений.

В то время как популярные открытые решения, такие как платформа Clover [4], реализуют флип с использованием жестко заданной последовательности команд — например, путем задания фиксированной угловой скорости на ограниченное время — такие подходы фактически игнорируют динамику системы и не используют обратную связь. Они не адаптируются к различным аппаратным конфигурациям, что снижает точность и надежность выполнения маневра.

Целью данной работы является разработка системы управления траекторией движения квадрокоптера, в котором флип рассматривается как контролируемое движение с учетом полной динамической модели квадрокоптера. Поставленная цель определила следующие задачи:

1. Рассмотреть историю развития БПЛА.
2. Описать архитектуру бортового аппаратного устройства и системы управления БПЛА.

3. Представить математическую модель роторно-винтового БПЛА
4. Изучить алгоритмические основы управления движением БПЛА.;
5. Представить обзор инструментальных средств, применяемых для разработки систем управления траекторией движения БПЛА
6. Научиться получать и обрабатывать данные сенсоров квадрокоптера.
7. Разработатьдинамическую модель квадрокоптера, а также реализовать на ее основе контроллер выполнения флипа.
8. Провести верификацию контроллера в симуляционной среде.

1. Теоретическая часть

1.1 История развития БПЛА

БПЛА — это одна из самых быстро развивающихся и революционных технологий в современной аэрокосмической индустрии. Их история берет начало в начале XX века, с первых попыток создания устройств для решения специфических задач. Прототипы БПЛА появились в период Первой мировой войны, когда британцы разработали аппарат "Kite Balloons", предназначенный для обучения пилотов. Однако настоящий прорыв в развитии беспилотных технологий произошел в 1920-х годах в США с созданием "Radioplane OQ-2" под руководством Говарда Хьюза, который стал первым радиоуправляемым беспилотным аппаратом, используемым для тренировки артиллеристов.

Эволюция систем управления

Развитие БПЛА тесно связано с прогрессом в области систем управления. Изначально управление осуществлялось вручную по радиоканалу или реализовывалось в виде жестко заданной траектории, а стабилизация — с помощью механических гироскопов. В 1950–1970-х годах появились аналоговые автопилоты, обеспечивающие удержание курса и высоты. С развитием микропроцессоров и GPS в 1980–1990-х годах получили распространение цифровые автопилоты с использованием ПИД-регуляторов, инерциальных измерительных систем и алгоритмов сенсорного слияния. Начиная с 2000-х годов стали внедряться более сложные методы управления: адаптивные, робастные, предиктивные. Развивались и алгоритмы планирования траектории с учетом ограничений по динамике и препятствиям. С начала 2010-х годов акцент сместился на автономную навигацию: интеграция SLAM, визуальной одометрии, методов машинного обучения и распределенного управления для роевой координации. Современные БПЛА способны адаптироваться к динамичной среде, выполнять задачи локализации, избегания препятствий и взаимодействия с другими агентами в реальном времени.

Квадрокоптеры классифицируются по конструкции и уровням автономности [3]:

- мультироторные квадрокоптеры, с четырьмя роторами, являются наиболее

популярной конфигурацией;

- гибридные квадрокоптеры, объединяющие свойства мультироторных и крыльевых ЛА, предлагают гибкость в эксплуатации.

По уровню автономности квадрокоптеры могут быть:

- управляемыми вручную, с дистанционным управлением;
- полуавтономными, которые способны выполнять определенные задачи без постоянного вмешательства оператора;
- автономными, оснащенными системами навигации и сенсорами для самостоятельного выполнения сложных задач.

Таким образом, эволюция БПЛА от простых радиоуправляемых моделей до высокотехнологичных автономных систем с искусственным интеллектом и сложными сенсорными технологиями демонстрирует бурный рост и широкие перспективы развития в самых разных областях.

1.2 Архитектура бортового аппаратного устройства и системы управления БПЛА

На рисунке 1 представлена архитектура бортового аппаратного устройства (БАУ) БПЛА.

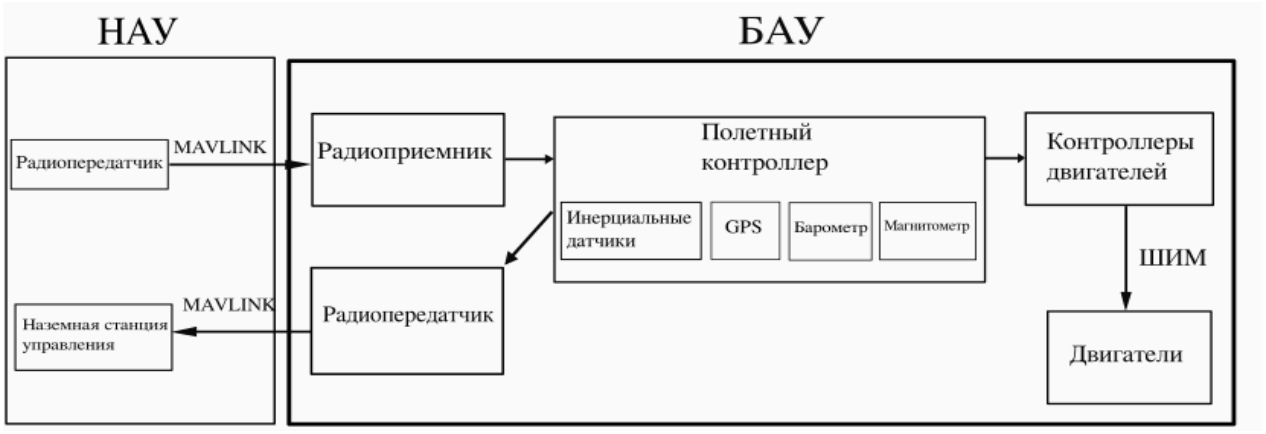

Рисунок – 1 – Бортовое аппаратное устройство БПЛА [3, 5]

Центральным элементом управления в системе является полетный контроллер. Он принимает команды с наземной станции или пульта управления, а также обрабатывает данные от инерциальных и навигационных датчиков. Основная задача полетного контроллера — стабилизация положения и реализация автономного управления в реальном времени.

Сенсорные модули

Инерциально-измерительная система (ИИС) включает акселерометр и гироскоп.

- Акселерометр позволяет получить относительное ускорение, то есть разность между истинным ускорением и гравитацией. Он может быть одно-, двух- или трехкомпонентным в зависимости от числа осей измерения. Принцип работы основан на смещении инерционной массы сенсора и преобразовании этого смещения в электрический сигнал.
- Гироскоп регистрирует угловую скорость. Основа работы механического — ротор, ось вращения которого свободна в одной или нескольких плоскостях. Схожее устройство имеют микроэлектромеханические (MEMS) гироскопы размером в несколько миллиметров [5]. Гироскопические датчики позволяют отслеживать поворот БПЛА в пространстве, а их чувствительные элементы дают выходной сигнал, пропорциональный угловой скорости.

ИИС обеспечивает данные, необходимые для расчета динамики полета. Однако, из-за дрейфа гироскопа и шумов акселерометра, данные требуют фильтрации. Для этого применяются фильтры Калмана или ПИД-регуляция [6-8].

Навигационные системы.

Стандартный ГНСС уязвима к спуфингу данных, поэтому современная навигация часто базируется на визуальной одометрии или SLAM-подходах, в которых используются данные с камер и ИИС. В работе [9] комбинация ГНСС и ИИС позволила повысить точность локализации БПЛА. А в работах [10, 11] описываются подходы к навигации основанные на слиянии ИНС и оптических датчиков.

Барометр измеряет атмосферное давление и используется для определения высоты. Различают несколько систем отсчета высоты:

- истинная — от поверхности земли;
- относительная — от произвольного уровня (например, взлетной полосы);
- эшелонная — от стандартного уровня 760 мм рт. ст.;
- абсолютная — от уровня моря.

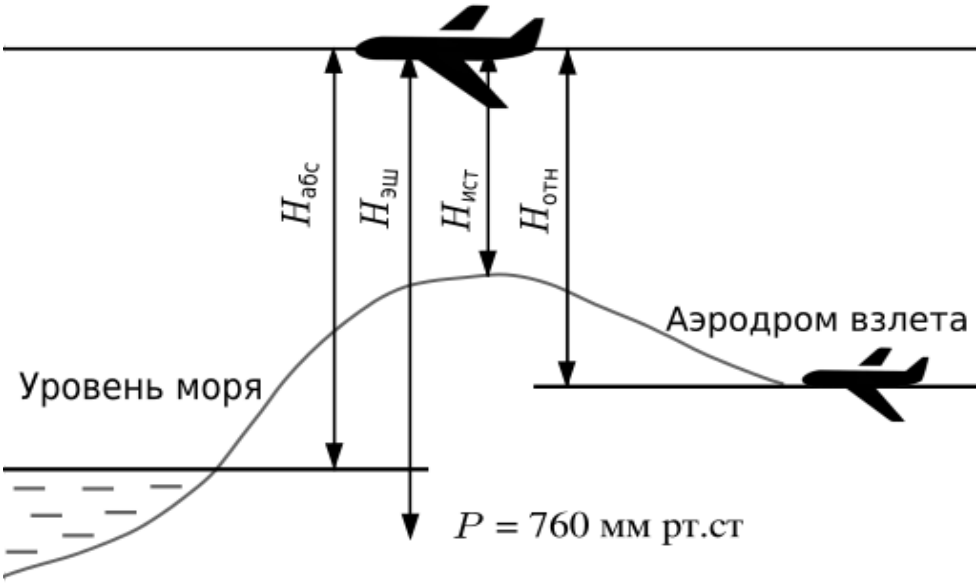

Рисунок – 2 – Классификация высот полета по системам отсчета [5]

Измерения барометра подвержены влиянию погодных условий, поэтому для компенсации ошибок применяются ультразвуковые, оптические, лазерные и радиоволновые дальномеры.

Обработка сенсорных данных: фильтрация и компенсация ошибок

Для обеспечения устойчивого и точного следования заданной траектории квадрокоптером в реальных условиях, необходимо учитывать неидеальности среды и динамические возмущения, такие как боковой ветер, турбулентность, неравномерная нагрузка, а также ошибки сенсоров. Квадрокоптер — это нелинейная, многовходовая и неустойчивая по своей природе система, обладающая высокой чувствительностью к внешним воздействиям. В таких условиях простые открытые системы управления оказываются недостаточными.

Применение ПИ и ПИД-регуляторов обусловлено необходимостью:

- подавления колебаний и быстрой компенсации отклонений, возникающих при маневрах или внешних возмущениях;
- стабилизации по углам Эйлера и высоте с минимальной перерегулировкой и временем переходного процесса;
- учетом накопленных систематических ошибок, например, из-за смещения центра масс или медленного дрейфа датчиков;
- устойчивого управления при наличии запаздывания в приводах и колебаний инерционных звеньев;

- поддержания устойчивости при активной обратной связи, особенно в замкнутом контуре управления ориентацией и положением.

Благодаря своей простоте, предсказуемости и адаптируемости к конкретным условиям полета, ПИ/ПИД-регуляторы остаются стандартом в системах управления БПЛА.

Для более сложных задач, таких как корректировка отклонений с учетом динамических изменений, важным элементом является использование фильтров для обработки данных. Одним из таких фильтров является расширенный фильтр Калмана (EKF), который активно применяется для локализации и стабилизации движения БПЛА. EKF позволяет интегрировать данные с различных сенсоров и получать более точную информацию о положении и ориентации аппарата, что критически важно для корректного функционирования регуляторов.

С помощью EKF можно эффективно обрабатывать шумные данные и получать аппроксимированные значения для более точной оценки состояния БПЛА, что, в свою очередь, улучшает работу ПИ и ПИД-регуляторов, повышая общую стабильность системы управления.

Таким образом, комбинация фильтров, таких как EKF, и регуляторов (ПИ, ПИД) позволяет обеспечить высокую точность, стабильность и адаптивность системы управления квадрокоптером, что особенно важно в динамично изменяющихся внешних условиях.

Состав БАУ также включает:

- радиомодули (для телеметрии и управления);
- электродвигатели и их контроллеры;
- литий-полимерные аккумуляторы;
- DC/DC-преобразователи и системы питания.

Полетный контроллер

Система автоматического управления (САУ), реализованная в полетном контроллере, представляет собой программно-аппаратный комплекс, предназначенный для обеспечения устойчивого, управляемого и автономного полета БПЛА. Она выполняет следующие ключевые функции [12-15]:

- Стабилизация положения в пространстве — осуществляется на основе

данных инерциальных датчиков, барометра, а также, при наличии, данных от внешней навигации (GPS/GLONASS/SLAM). САУ реализует каскадные контуры регулирования по углам Эйлера или кватерниону, используя ПИД-регуляторы или более сложные алгоритмы управления.

- Автономный полет по заданным GPS-точкам — маршрут задается заранее в виде набора координат в мировой системе. Полетный контроллер интерпретирует маршрут, вычисляет ошибки навигации и формирует управляющее воздействие для следования траектории, включая развороты, удержание курса, коррекцию по ветру и коррекцию высоты.
- Режим возврата в точку старта — активируется при потере связи с оператором, по истечении времени миссии, по снижению уровня заряда аккумулятора ниже порога или по команде. В этом режиме контроллер стабилизирует аппарат, поднимает его на заданную безопасную высоту и возвращает по обратной траектории, после чего выполняет посадку или зависание.
- Режимы зависания и посадки — обеспечиваются с помощью замкнутых контуров управления по высоте, положению и скорости. Контроллер удерживает аппарат в фиксированной точке пространства или плавно снижает его по вертикали. При наличии камеры или дальномера возможна посадка с визуальным или радиометрическим наведением на метку.
- Управление по радиоканалу — САУ преобразует команды оператора (например, от передатчика с протоколами SBUS, PPM, iBUS) в управляющие воздействия, напрямую влияющие на ШИМ-сигналы двигателей и сервоприводов. Это позволяет реализовать ручное, полуавтономное и полностью автономное управление.

Таким образом, полетный контроллер не просто стабилизирует БПЛА, но и выполняет функции низкоуровневого управления приводами, высокоуровневого планирования и контроля миссии, а также служит интерфейсом между аппаратурой и оператором.

Большинство современных решений для полетных контроллеров, таких как INAV [12], Betaflight [13], ArduPilot[14] используют микроконтроллеры (МК)

семейства STM32 от STMicroelectronics. МК на базе ARM Cortex-M обеспечивают оптимальный баланс между производительностью, энергоэффективностью и поддержкой периферии. Благодаря наличию интерфейсов I2C, SPI, UART, PWM, CAN и USB, а также широкому диапазону доступных моделей по частоте и объему памяти, STM32 остаются де-факто стандартом в сфере малых БПЛА. В то же время, на более ресурсоемких платформах, например в PX4[15] или автономных VTOL-системах, применяются более мощные процессоры, включая ARM Cortex-A, а также, на базе Linux-систем — Raspberry Pi, NVIDIA Jetson, BeagleBone или гибридные решения с сопроцессорами. Такие архитектуры позволяют реализовывать алгоритмы визуальной навигации, SLAM и onboard AI, что выходит за рамки типичных возможностей полетных контроллеров.

Современные системы управления беспилотными летательными аппаратами требуют интеграции множества компонентов, таких как сенсоры, навигационные системы и исполнительные устройства. Для обеспечения эффективного взаимодействия между ними необходим унифицированный интерфейс передачи данных. Одним из таких интерфейсов, который широко используется в промышленности, является UAVCAN.

UAVCAN (Uncomplicated Application-level Vehicular Communication And Networking) — это протокол передачи данных, адаптированный для использования в системах управления БПЛА и других роботизированных системах. Он построен на принципах, аналогичных стандартному CAN (Controller Area Network), но разработан с учетом специфики аэрокосмических и робототехнических приложений [16]. Важно отметить, что такие полетные контроллеры, как PX4 и Pixhawk, активно поддерживают интерфейс UAVCAN, что позволяет интегрировать различные компоненты БПЛА в единую, масштабируемую и надежную систему.

Основные характеристики UAVCAN включают:

- Модульность, позволяющая легко добавлять новые устройства в систему, что особенно важно для гибкости при разработке и эксплуатации сложных систем.

- Надежность передачи данных, критично важная для работы в условиях ограниченной связи или потенциальных помех.
- Широкая совместимость, поддерживающая разнообразные устройства, от сенсоров до исполнительных механизмов, что делает систему масштабируемой и адаптируемой.
- Эффективность в минимизации задержек и потерь данных, что особенно важно для реального времени работы БПЛА.
- Простота интеграции, благодаря открытым спецификациям и документации.

Протокол UAVCAN использует архитектуру "публикация-подписка", позволяя устройствам публиковать данные или подписываться на сообщения других компонентов, что способствует динамичному взаимодействию в сети без жесткой привязки к конкретным адресам [16]. Это делает его удобным и мощным инструментом для разработки современных высоконадежных систем управления БПЛА.

С учетом того, что системы PX4 и Pixhawk активно поддерживают UAVCAN, внедрение этого протокола в будущие разработки БПЛА становится все более актуальным, обеспечивая высокую гибкость, надежность и эффективность работы.

1.3 Математическая модель роторно-винтового БПЛА

Квадрокоптер представляет собой высокоманевренный летательный аппарат, однако его динамика характеризуется низкой устойчивостью, поскольку он подвержен значительным внешним возмущениям. В связи с этим система управления квадрокоптером должна решать задачи угловой и пространственной стабилизации, обеспечения заданной высоты, а также поддержания состояния зависания, посадки и полета по заранее заданной траектории [6-8].

Отработка законов управления осуществляется методами математического, имитационного и компьютерного моделирования. Для решения этой задачи необходимо понимание принципов работы квадрокоптера, механики полета и наличие математической модели, описывающей его динамику. Кинематическая и

динамическая модели служат основой для анализа поведения БПЛА в различных условиях, что позволяет оптимизировать алгоритмы управления и повысить точность выполнения заданных траекторий.

В **кинематической модели** квадрокоптера не учитываются внешние силы, воздействующие на его движение. Рассматривается изменение положения, скорости и ориентации дрона в пространстве. Подробное описание обобщенной кинематической схемы квадрокоптера приведено в [6]. В данной работе выделяются два основных типа движения:

1. Поступательное движение — это перемещение центра масс квадрокоптера в инерциальной системе отсчета, в данном случае относительно Земли. Такое движение описывается координатами и скоростью квадрокоптера в пространстве.
2. Вращательное движение квадрокоптера характеризует его ориентацию относительно инерциальной системы координат и может быть представлено с помощью углов Эйлера, кватернионов или матрицы направляющих косинусов (DCM).

В данной работе будет рассматриваться конструкция квадрокоптера, представленная на рисунке 3, а также его кинематическая схема, показанная на рисунке 4.

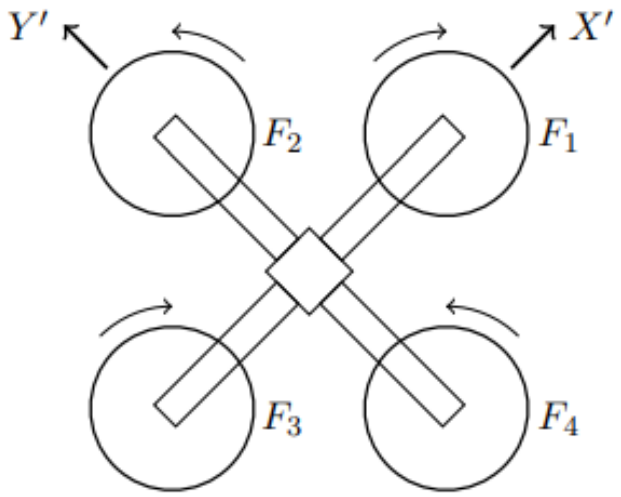

Рисунок – 3 Схематичное изображение квадрокоптера

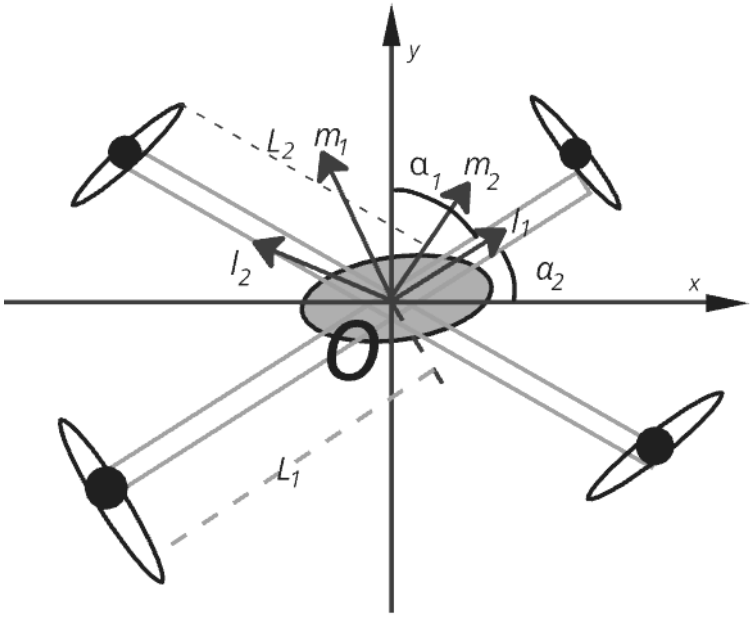

Рисунок – 4 Кинематическая схема квадрокоптера

В пространстве $OXYZ$ направление полета квадрокоптера задается осью OX. Две пары вращающихся моторов, расположенные на одинаковом расстоянии L_1 и L_2 от начала координат O, направлены вдоль осей OL_1 и OL_2 соответственно. Углы наклона штанг квадрокоптера относительно положительного направления оси OX составляют углы α_1 и α_2, при этом $\alpha_1 > \alpha_2$. Единичные векторы этих осей OL_1 и OL_2 обозначены через $\overline{l}_1$ и $\overline{l}_2$, где

$$l_1 = (sin(\alpha_1),\ cos(\alpha_1)),\ l_2 = (sin(\alpha_2), cos(\alpha_2)) \quad (1)$$

Матрица перехода R_{BL} от осей OXY к осям OL_1L_2, связанная с расположением пропеллеров, имеет следующий вид:

$$R_{BL} = \left[\overline{l}_1\ \overline{l}_2\right] = \begin{bmatrix} cos(a_1)\ cos(a_2) \\ sin(a_1)\ sin(a_2) \end{bmatrix} \quad (2)$$

Определяется также детерминант матрицы перехода:

$$det(R_{BL}) = sin(a_2 - a_1)$$

Для схемы, показанной на рисунке 1, примем, что $L_1 = L_2$, $\alpha_1 = 45^\circ$ $\alpha_2 = 135^\circ$. Поскольку пропеллеры установлены непосредственно на осях моторов, можно приближенно считать, что вращающий момент τ_i мотора,

создаваемый тягой T_i, определяется следующим соотношением:

$$\tau_i = k_\psi T_i, i = 1,2,3,4 \text{ , где} \quad (3)$$

k_ψ – конструктивный параметр.

Общая подъемная сила u_z вдоль вертикальной оси OZ квадрокоптера определяется следующим уравнением:

$$u_z = T_1 + T_2 + T_3 + T_4 \quad (4)$$

Вращающий момент относительно оси рыскания OZ определяется следующим уравнением:

$$u_\psi = -(\tau_1 - \tau_2 + \tau_3 - \tau_4) = -k_\psi (T_1 - T_2 + T_3 - T_4) \quad (5)$$

Моменты $\overline{M_1}$ и $\overline{M_2}$ относительно осей OX и OY создаются за счет разности тяг противоположно действующих моментов, причем моменты каждой пары моторов находятся в плоскости OXY и определяются следующими соотношениями:

$$M_1 = -(T_1 - T_3)L_1,\ M_2 = -(T_2 - T_4)L_2 \quad (6)$$

Векторы $\overline{m_1}$ и $\overline{m_2}$ направленные перпендикулярно к осям моторов, определяются как:

$$m_1 = (cos(\alpha_1),\ -sin(\alpha_1)),\ m_2 = (-cos(\alpha_2),\ sin(\alpha_2)) \quad (7)$$

Управляющие моменты u_ϕ и u_θ в координатной плоскости OXY определяются как:

$$M_{XY} = u_x\hat{i} + u_y\hat{j} = M_1 + M_2 = -(T_1 - T_3)L_1m_1 + (T_2 - T_4)L_2m_2 \quad (8)$$

Подставив выражения для m_1 и m_2, получаем:

$$u_\phi = (T_1 - T_3)L_1sin(\alpha_1) + (T_2 - T_4)L_2sin(\alpha_2), \quad (9)$$

$$u_\theta = (T_1 - T_3)L_1cos(\alpha_1) - (T_2 - T_4)L_2cos(\alpha_2). \quad (10)$$

Таким образом, приведены аналитические выражения для подъемной силы u_z(4) и управляющих моментов u_ψ (5), крена u_ϕ (9) и тангажа u_θ (10), , которые зависят от тяг T_1, T_2, T_3, T_4 и геометрии установки моторов, характеризующейся

углами α_1, α_2, а также расстояниями L_1, L_2 , которые связаны с геометрией установки моторов.

Связь между векторами управляющих моментов $\overline{U} = \left[u_z,\ u_\phi,\ u_\theta,\ u_\psi\right]$ и тягами моторов $\overline{T} = \left[T_1, T_2, T_3, T_4\right]^T$ можно записать в матричной форме:

$$\overline{U} = D_M \overline{T} \tag{11}$$

где матрица D_M имеет следующий вид:

$$D_M = \begin{bmatrix} -k & -L_1 cos(\alpha_1) & L_2 sin(\alpha_1) \\ k & L_2 cos(\alpha_2) & L_2 sin(\alpha_2) \\ -k & L_1 cos(\alpha_1) & L_1 sin(\alpha_1) \\ k & L_2 cos(\alpha_2) & L_2 sin(\alpha_2) \end{bmatrix}, \tag{12}$$

Детерминант этой матрицы:

$$det(D_M) = -8 L_1 L_2 k_\psi \sin(\alpha_1 - \alpha_2) \tag{13}$$

Таким образом, выражение для матрицы D_M^{-1}, инверсированной матрицы, задающей связь между тягами и требуемыми моментами, выглядит следующим образом:

$$K_D = D_M^{-1} = \begin{bmatrix} \frac{1}{4} & \frac{cos(\alpha_2)}{2L_1 sin(a_1 - \alpha_2)} & \frac{sin(\alpha_2)}{2L_1 sin(a_1 - \alpha_2)} & -\frac{1}{4k_\Psi} \\ \frac{1}{4} & -\frac{cos(\alpha_2)}{2L_2 sin(a_1 - \alpha_2)} & -\frac{sin(\alpha_1)}{2L_2 sin(a_1 - \alpha_2)} & \frac{1}{4k_\Psi} \\ \frac{1}{4} & -\frac{cos(\alpha_2)}{2L_1 sin(a_1 - \alpha_2)} & -\frac{sin(\alpha_2)}{2L_1 sin(a_1 - \alpha_2)} & -\frac{1}{4k_\Psi} \\ \frac{1}{4} & \frac{cos(\alpha_1)}{2L_2 sin(a_1 - \alpha_2)} & \frac{sin(\alpha_1)}{2L_2 sin(a_1 - \alpha_2)} & \frac{1}{4k_\Psi} \end{bmatrix} \tag{14}$$

Если $\alpha_1 = 45^\circ$, $\alpha_2 = 135^\circ$ и $L_1 = L_2 = L$, то матрицы D_M (12) и K_D(14) примут следующий вид:

$$D_M = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \frac{\sqrt{2}L}{2} & \frac{\sqrt{2}L}{2} & -\frac{\sqrt{2}L}{2} & -\frac{\sqrt{2}L}{2} \\ -\frac{\sqrt{2}L}{2} & \frac{\sqrt{2}L}{2} & \frac{\sqrt{2}L}{2} & -\frac{\sqrt{2}L}{2} \\ -k_\Psi & k_\Psi & -k_\Psi & k_\Psi \end{bmatrix} \quad (15)$$

$$K_D = \begin{bmatrix} \frac{1}{4} & \frac{\sqrt{2}}{4L} & -\frac{\sqrt{2}}{4L} & -\frac{1}{4k_\Psi} \\ \frac{1}{4} & \frac{\sqrt{2}}{4L} & \frac{\sqrt{2}}{4L} & \frac{1}{4k_\Psi} \\ \frac{1}{4} & -\frac{\sqrt{2}}{4L} & \frac{\sqrt{2}}{4L} & -\frac{1}{4k_\Psi} \\ \frac{1}{4} & -\frac{\sqrt{2}}{4L} & -\frac{\sqrt{2}}{4L} & \frac{1}{4k_\Psi} \end{bmatrix} \quad (16)$$

Представленные математические выражения были использованы для разработки модели квадрокоптера в программной среде и предназначены для расчета его кинематических характеристик, таких как значения тяги моторов и их влияние на корпус квадрокоптера вокруг оси рыскания.

Динамическая модель квадрокоптера описывает эволюцию его состояния под воздействием внутренних и внешних сил, включая тягу от винтов, гравитацию, аэродинамическое сопротивление и моменты, вызванные асимметрией тяги. В отличие от чисто кинематического подхода, она позволяет учитывать инерционные эффекты и воздействие управляющих воздействий в динамике полета.

Квадрокоптер обладает шестью степенями свободы: его движение состоит из поступательного перемещения центра масс и вращения относительно центра масс. Для описания ориентации в пространстве используется параметризация с помощью кватернионов, что позволяет избежать сингулярностей и обеспечивает численно устойчивое интегрирование вращательного движения.

Состояние аппарата описывается вектором [2]:

$$x = [p,\ v,\ q,\ \omega], \quad (17)$$

где $p \in R_3$ — положение квадрокоптера в мировой системе координат,

$v \in R_3$ — линейная скорость,

$q \in R_4$ — единичный кватернион, описывающий ориентацию в пространстве,

$\omega \in R_3$ — угловая скорость в теле.

Управление осуществляется с помощью вектора угловых скоростей винтов:

$$u = \left[\omega_1\ \omega_2\ \omega_3\ \omega_4\right] \in R_4 \tag{18}$$

Линейная динамика:

Суммарная тяга от винтов в системе координат тела выражается как:

$$F_{body} = \left[0, 0, \sum_{i=1}^{4} k_f \omega_i^2\right], \tag{19}$$

где k_f — коэффициент тяги. Эта сила переводится в мировую систему координат с помощью матрицы поворота $R(q)$, соответствующей текущему кватерниону:

$$F_{world} = R(q)F_{body} - mg - c_d v, \tag{20}$$

где m — масса квадрокоптера,

$g = [0, 0, 9.81]$— гравитация,

c_d — коэффициент аэродинамического сопротивления, пропорционального скорости.

Из второго закона Ньютона получаем ускорение:

$$a = \frac{F_{world}}{m}$$

Интегрируя ускорение, получаем линейную скорость и позицию:

$$v(t + \Delta t) = v(t) + a \cdot \Delta t,$$

$$p(t + \Delta t) = p(t) + v(t) \cdot \Delta t + \frac{1}{2} a \cdot \Delta t^2 \tag{21}$$

Вращательная динамика:

Крутящие моменты от винтов рассчитываются по следующей модели:

$$\tau = \left[l(k_f(\omega_2^2 - \omega_4^2));\ l(k_f(\omega_3^2 - \omega_1^2));\ k_m(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2)\right], \tag{22}$$

где l – длина плеча,

k_f – коэффициент момента.

Угловое ускорение рассчитывается из уравнений Эйлера:

$$\widehat{\omega} = I^{-1}(\tau - \omega \times (I\omega)), \quad (23)$$

где $I \in R_3$ — тензор инерции.

Угловая скорость обновляется как:

$$\omega(t + \Delta t) = \omega(t) + \widehat{\omega} \cdot \Delta t \quad (24)$$

Ориентация тела представлена кватернионом q. Из уравнений кватернионной кинематики:

$$\widehat{q} = \frac{1}{2} q \otimes [0\ \omega] \quad (25)$$

Ориентация обновляется через численную интеграцию:

$$q(t + \Delta t) = q(t) + \widehat{q} \cdot \Delta t \quad (26)$$

$$q \leftarrow \frac{q}{||q|| + \varepsilon}$$

Здесь добавлена малая константа $\varepsilon \sim 10^{-8}$, предотвращающая деление на ноль при нормализации.

Поступательное ускорение центра масс рассчитывается по второму закону Ньютона:

$$m \cdot a = F_{\text{тяга}} - m \cdot g - F_{\text{сопр}}, \quad (27)$$

Вращательное движение определяется уравнением Эйлера для момента импульса:

$$I \cdot \widehat{\omega} = \tau - \omega \times (I \cdot \omega), \quad (28)$$

где τ — результирующий момент, создаваемый несбалансированными тягами роторов и их реактивными моментами.

ω — угловая скорость,

I — тензор инерции дрона.

Моменты включают:

- опрокидывающие моменты (крен/тангаж): создаются разностью тяг противоположных роторов;
- реактивный момент (рыскание): возникает от разности реактивных

моментов всех четырех роторов, учитываемый через коэффициент крутящего момента.

Таким образом, модель охватывает как поступательное, так и вращательное движение квадрокоптера, включая воздействие гравитации, аэродинамического сопротивления, ограничений на винты и нелинейных моментов. Выбор кватернионного представления обеспечивает численно устойчивое описание ориентации без особенностей, присущих, например, углам Эйлера.

Следует отметить, что в настоящее время гибридные стохастические модели широко применяются для моделирования и оценки параметров движения различных объектов.

Гибридные системы — это математические модели динамики, в которых непрерывное поведение, описываемое одним из заранее заданных наборов непрерывных динамических уравнений, чередуется с дискретными событиями. Эти события могут вызывать мгновенное переключение между различными режимами динамики, мгновенное изменение состояния системы (координат), либо одновременно то и другое [17,18].

В научной литературе представлено множество подходов к построению гибридных моделей. Так, в [17] описана гибридная модель движения автотранспортного средства, в которой фазы разгона и торможения моделируются с помощью дифференциальных уравнений, тогда как смена полосы движения трактуется как дискретное событие. В работе [18] предложена оригинальная нейро-стохастическая гибридная модель, объединяющая методы искусственных нейронных сетей и стохастических процессов. В [18] рассматривается алгоритм построения маршрута в навигационной задаче, использующий кусочно-линейное аппроксимирование траектории при движении через несколько геофизических полей с ограничениями на длину линейных участков.

В [19] гибридная стохастическая модель описывается как многорежимная система, состоящая из набора дискретных линейных стохастических моделей, каждая из которых описывает отдельный режим движения объекта — участок траектории, приближенно описываемый линейной моделью. Такой подход

позволяет аппроксимировать сложную, в общем случае нелинейную, траекторию последовательностью простых линейных сегментов.

Гибридные модели широко применяются для описания и управления движением квадрокоптеров, где они служат эффективным инструментом для реализации адаптивного и точного поведения в различных условиях среды. Совмещение физических моделей и методов машинного обучения позволяет достигать высокой точности в задачах управления и навигации. Развитие таких моделей открывает перспективы для повышения эффективности беспилотных летательных аппаратов и расширения их практического применения.

В частности, управление дроном при выполнении маневра, такого как флип, можно описать как гибридную систему. Полет разбивается на различные режимы — взлет, вращение, стабилизация, посадка, — каждый из которых имеет собственную непрерывную динамику, описываемую системой дифференциальных уравнений. Переходы между режимами происходят дискретно — по событиям, сигналам, условиям времени или достижения определенного состояния.

В настоящей работе применяется гибридный подход, объединяющий кинематическую и динамическую модели в рамках стохастического процесса. Такой подход позволяет учесть ограничения на вычислительные ресурсы системы, а также комбинировать различные методы расчета для достижения приемлемого качества управления полетом. Использование фильтра Калмана для оценки состояния (позиции, линейной и угловой скорости, ориентации) позволяет обеспечить устойчивую и адаптивную работу модели даже в условиях неопределенности и помех.

При этом в моделируемой системе присутствуют характерные признаки гибридной структуры:

- дискретные события, такие как переключение управляющих стратегий;
- разные модели динамики для отдельных фаз полета;
- дискретная логика переключения между этими фазами, основанная на времени, достижении заданных углов или других критериях.

1.4 Алгоритмические основы управления движением БПЛА

Современные БПЛА решают задачи, требующие высокой точности и надежности выполнения полетного задания, что достигается за счет использования эффективных алгоритмов планирования, предсказания и коррекции траектории в реальном времени.

В рамках данной работы решаются следующие основные задачи, связанные с движением БПЛА по траектории:

- построение математической модели динамики движения с учетом физических параметров квадрокоптера, ограничений на управление и аэродинамических эффектов;
- формирование трехмерного представления окружающего пространства на основе данных инерциальной навигационной системы (IMU), обрабатываемых с помощью фильтра Калмана для оценки положения, скорости, ориентации и угловой скорости;
- поиск оптимального маршрута в трехмерном пространстве с использованием подхода адаптивного планирования, учитывающего как ограничения динамики дрона, так и поступающую в реальном времени обратную связь от сенсоров;
- аппроксимация гладкой траектории по заданным контрольным точкам с помощью итеративного линейно-квадратичного регулятора (iLQR), обеспечивающего непрерывность и согласованность управляющих воздействий.

Рассмотрим более подробно современные подходы к решению каждой из перечисленных выше задач.

Построение математической модели динамики движения

При планировании пути важно учитывать кинематические и динамические ограничения квадрокоптера, включая ограничения по скорости, ускорению, моменту и способности избегать препятствий. Такие ограничения критичны для реализации безопасных и реалистичных траекторий и подробно рассматриваются в [21-25].

Динамическая модель квадрокоптера строится на основе уравнений

движения твердого тела с учетом тяги, аэродинамических сил и моментов, ограничений по управляемости, а также параметров исполнительных механизмов. Эта модель служит основой для предсказания поведения системы в рамках стратегии управления с прогнозированием позволяет учитывать физические ограничения БПЛА, отклонения траектории и обратную связь с бортовыми измерениями при планировании управляющих воздействий. Для получения гладкой и выполнимой траектории по заданным контрольным точкам применяются различные методы аппроксимации, например, используемые в работах [23,26].

Динамическая модель необходима для реализации алгоритмов оптимального управления, минимизирующие целевой функционал при соблюдении динамики системы. Также модель необходима для симуляции полета и тестирования алгоритмов в виртуальной среде, служит основой в фильтрах оценки состояния и применяется для обратного расчета управления при выполнении заданной траектории. Таким образом, динамическая модель является фундаментом для широкого спектра алгоритмов планирования, управления и локализации БПЛА.

Формирование трехмерного представления окружающего пространства — одна из ключевых задач автономной навигации, необходимая для реализации управления с обратной связью — обхода препятствий, построения траекторий и решения прикладных задач. В современных БПЛА задачи картирования часто решаются одновременно с задачей локализации, что позволяет снизить зависимость от GPS-сигнала и повысить устойчивость системы в условиях его отсутствия или помех [2, 27-29]. Этот подход реализуется в рамках SLAM-алгоритмов (Simultaneous Localization and Mapping), основанных на одновременном построении карты и определении положения БПЛА. В зависимости от используемых сенсоров, выделяют визуально-инерциальные и лидарно-инерциальные реализации. Лидарно-инерциальные методы обладают важным преимуществом — высокой устойчивостью к изменениям освещения, что особенно ценно при работе в сложных условиях, где визуальные методы теряют точность.

Одним из передовых SLAM-алгоритмов является LIO-SAM [30-31], разработанный как развитие популярного LOAM [32]. Основным улучшением LIO-SAM является использование факторного графа для оптимизации одометрии. Это позволяет гибко включать как относительные, так и абсолютные измерения, в том числе замыкания контуров, полученные из различных источников.

Камеры по сравнению с лидарами дешевле, потребляют меньше энергии и обеспечивают более высокую частоту кадров (обычно 30–120 Гц и выше), что делает их особенно привлекательными для применения на малогабаритных БПЛА. Подробный обзор методов визуально-инерциальной одометрии (VIO) представлен в [33]. Визуальные сенсоры предоставляют богатую текстурную информацию, что позволяет использовать алгоритмы компьютерного зрения. Однако, по точности измерения расстояний и геометрии объектов лидары по-прежнему превосходят камеры. Одним из основных ограничений VIO-методов является снижение точности при высоких скоростях полета из-за накопления ошибок. Для снижения этого эффекта используются различные фильтры, такие как MSCKF и ROVIO, а также алгоритм VINS-Mono, реализующий локализацию по данным от одной камеры и инерциального модуля.

Кроме VIO, существуют также методы визуальной одометрии, не использующие инерциальные данные. Такие алгоритмы, как ORB-SLAM и OpenVSLAM, строят карту мира только на основе изображений, применяя графовую оптимизацию и механизмы замыкания циклов для коррекции накопленных ошибок. Эти подходы особенно эффективны при медленном движении и отсутствии резких ускорений, например, в задачах картографирования с наземными роботами. В ряде исследований [31-33] были сравнены алгоритмы RTAB-Map, ORB-SLAM, LSD-SLAM и OpenVSLAM. ORB-SLAM3 показал наивысшую точность локализации, тогда как OpenVSLAM продемонстрировал отличную способность к замыканию контуров и обновлению карты.

Дополнительную роль в навигации играют системы распознавания

образов, особенно в условиях неопределенности, например, в помещениях или городской застройке. Такие системы анализируют визуальную информацию для идентификации объектов, определения их положения и скорости, а также для предотвращения столкновений. Одним из распространенных приемов является метод контрастов, выделяющий характерные особенности изображений — границы объектов, резкие перепады яркости и другие значимые признаки. Это позволяет более точно воспринимать и интерпретировать окружающее пространство, улучшая построение траекторий и устойчивость навигации.

Для решения задачи поиска оптимального пути в трехмерном пространстве применяются различные классы алгоритмов:

Графовые алгоритмы поиска пути

К классическим алгоритмам поиска пути относятся алгоритмы Дейкстры, Флойда [34-36] и их различные оптимизации. Также широко применяется эвристический A*. Эти алгоритмы эффективно решают задачи поиска пути в дискретных пространствах с известной картой. Другие эвристические алгоритмы, такие как Particle Swarm Optimization, Artificial Bee Colony и Harmony Search, ориентированы на оптимизацию и применяются для планирования плавных траекторий и настройки параметров управления дроном.

Алгоритмы случайного поиска

В работе [3] для планирования скоростного движения дрона в условиях густого леса был успешно применен алгоритм RRT. Помимо RRT, для задач навигации и планирования траектории также широко используются методы Probabilistic Roadmap и Rapidly-exploring Random Graph, которые эффективно исследуют сложные пространства состояний и обеспечивают надежный поиск путей в высокоразмерных и ограниченных средах.

Методы визуального восприятия

В ситуациях, когда данные 3D-облака точек недоступны, часто применяются алгоритмы анализа оптического потока. Одним из наиболее простых и распространенных методов является метод Хорна–Шунка, но требует значительных вычислительных ресурсов. Метод Лукаса–Канаде работает локально на ключевых точках и эффективен при малых смещениях, а его

пирамидальная версия расширяет диапазон движений.

Методы оптимального управления

Класс алгоритмов, направленных на построение управляющих воздействий, минимизирующих заданную целевую функцию с учетом динамики системы и ограничений на состояния и управления. В отличие от дискретных графовых методов и подходов обучения с подкреплением, оптимальное управление работает в непрерывных пространствах и позволяет учитывать физические характеристики системы, такие как масса, момент инерции, пределы по скорости и ускорению.

В современных научных исследованиях по теме оптимального управления все чаще применяются методы семейства линейных квадратичных регуляторов (LQR). Особое внимание уделяется итеративному варианту — iLQR, который эффективно справляется с задачами управления нелинейными системами. В работах [37,38] предложен и исследован гибридный подход iLQR-MPC, объединяющий преимущества итеративного линейного квадратичного регулятора и модельно-предиктивного управления. В работах показано, что такой комбинированный контроллер превосходит как отдельный iLQR, так и классический PID-регулятор — как по точности следования траектории, так и по устойчивости в переходных режимах, особенно в условиях быстрых маневров и ограничений по управляемости.

В работе [38] представлен ограниченный итеративный LQR, CILQR — расширение iLQR с учетом ограничений и неопределенностей, направленное на повышение безопасности и адаптивности системы. Авторы предложили двухэтапную стратегию прогнозирования и сценарий-зависимую функцию стоимости, что позволило добиться устойчивости поведения без необходимости ручной настройки параметров под каждый сценарий.

Кроме того, в работе [39] отмечено применение линейной версии алгоритма LQR для управления скоростью синхронных двигателей с постоянными магнитами вместо традиционного ПИД-регулятора. Такой подход позволил значительно снизить колебания скорости и момента, что положительно сказалось на общих характеристиках системы управления.

Функция стоимости играет центральную роль в методах оптимального управления. Она не только позволяет оценить ошибки по различным параметрам системы, таким как положение, ориентация и управляющие воздействия, но и направляет систему к конечной цели, минимизируя эти ошибки. Эффективное использование функции стоимости позволяет гарантировать, что система будет двигаться в нужном направлении и достигнет цели с минимальными отклонениями.

Основные компоненты функции стоимости обычно включают в себя:

- ошибки по позиции — отклонения текущего положения от целевого положения;
- ошибки по ориентации — отклонения текущей ориентации от целевой ориентации;
- ошибки в управляющих воздействиях — отклонения между текущими управляющими усилиями и теми, что необходимы для следования траектории.

Кроме того, функция стоимости может включать терминальный член, который играет важную роль в обеспечении сходимости системы к конечной цели. Это позволяет гарантировать, что, независимо от пути, который проходит система, она будет стремиться к заданному конечному состоянию.

Функция стоимости для задачи планирования траектории используемая в данной работе представлена как:

$$X = (x_k - x_{target,k})^T Q (x_k - x_{target,k}),$$

$$U = (u_k - u_{target,k})^T R (u_k - u_{target,k}),$$

$$J = \sum_{k=0}^{N-1} (X + U) + (x_N - x_{target,N})^T Q_f (x_N - x_{target,N}), \qquad (29)$$

где x_k и u_k — это вектор состояния и управляющее воздействие на шаге k,

$x_{target,k}$ и $u_{target,k}$ — это целевые значения состояния и управляющего воздействия на шаге k,

Q и R — это матрицы веса для состояния и управляющих воздействий, которые

регулируют важность минимизации ошибок в позиции и управляющих воздействиях,

Q_f — это терминальная матрица, которая контролирует вес ошибки в конечном состоянии N.

Аппроксимация гладкой кривой по контрольным точкам траектории является важным этапом планирования движения БПЛА. В работе [41] для классического подхода на основе B-сплайнов был предложен метод численной градиентной оптимизации, позволяющий смещать траекторию на безопасное расстояние от препятствий. Такая оптимизация обеспечивает не только сглаживание, но и безопасность планируемого пути. Также представлен сравнительный анализ различных методов аппроксимации траекторий, включая:

- B-сплайны;
- радиальные базисные функции (RBF);
- квинтовые полиномы;
- аппроксимацию на основе полиномов Чебышева;
- методы коллокации;
- параметрические модели с ограничениями (Bezier и подобные);
- параметрические кривые с плавной кривизной (клотоиды, кубические спирали, G²-сплайны, intrinsic-сплайны);

Каждый из этих подходов имеет свои преимущества с точки зрения гладкости, вычислительной сложности, удобства учета ограничений и устойчивости к шуму. Однако после построения целевой траектории возникает следующая важная задача — обеспечение ее надежного воспроизведения в условиях реальной динамики дрона и внешних возмущений. На этом этапе ключевым становится не только следование заранее заданной траектории, но и эффективное управление поведением системы, обеспечивающее стабильность и точность в реальных условиях. Важной частью решения этой задачи является использование функции стоимости, позволяющей оценить качество выполнения задачи и управлять отклонениями от целевой траектории.

Следует отметить, что в методе iLQR аппроксимация происходит на уровне локальных линейных и квадратичных приближений динамики и функции

стоимости, что позволяет интегрировать планирование и управление с учетом реальных физических ограничений системы напрямую в процессе оптимизации. В то время как классические методы аппроксимации траекторий, такие как сплайны и полиномы, в основном ориентированы на геометрическое сглаживание и могут учитывать ограничения через дополнительные условия, iLQR работает с динамикой и критериями управления, что обеспечивает более точное моделирование скорости, ускорения и отклонений. Это позволяет iLQR быстрее сходиться к локально оптимальному решению и адаптироваться к изменяющимся условиям и возмущениям в реальном времени.

1.5 Инструментальные средства для разработки систем управления

Системы управления БПЛА, как правило, реализуются на языках C и C++, поскольку они обеспечивают прямой доступ к аппаратным регистрам, малое время отклика и контроль над ресурсами, что критично для выполнения задач в реальном времени. Эти языки широко используются в низкоуровневом ПО, включая слои HAL, RTOS, драйверы сенсоров и реализацию контуров стабилизации. Язык Python применяется преимущественно в высокоуровневых компонентах — например, для разработки и тестирования алгоритмов планирования, визуальной навигации, взаимодействия с симуляторами и анализа логов, благодаря широкому экосистемному окружению (NumPy, SciPy, ROS 2, OpenCV и др.). MATLAB/Simulink используется в инженерной практике для моделирования динамики, анализа устойчивости и автоматической генерации кода C для внедрения в микроконтроллеры.

Одной из ведущих платформ для разработки робототехнических систем является ROS 2 — модульная и масштабируемая система, ориентированной на распределенную архитектуру. Использование протокол DDS обеспечивает надежную и быструю передачу сообщений, что критично для управления в реальном времени. ROS 2 поддерживает языки программирования C++ и Python, а также предоставляет инструменты для отладки и визуализации. Более того ROS 2 тесно интегрирована с автопилотом PX4 [42,43].

При выбору Python в качестве ЯП для реализации алгоритмов

планирования траекторий и управления динамикой БПЛА применяются библиотеки с поддержкой автоматического дифференцирования, критичного для точного вычисления производных. Среди популярных решений — JAX, Pytorch и Pydrake. JAX более высокопроизводительный благодаря JIT-компиляции с использованием XLA и эффективной работе на GPU/TPU, обеспечивая точные производные и гибкость в дифференцировании сложных функций. Pytorch тоже поддерживает автоматическое дифференцирование, но уступает JAX в оптимизации под ускорители.

В Pydrake нельзя напрямую передавать произвольную функцию как динамику системы. Вместо этого необходимо строить полную структуру системы через классы, унаследованные от LeafSystem, определять входы, выходы, непрерывные состояния. Такой подход требует больше усилий и не позволяет быстро менять форму динамики. Кроме того, автоматическое дифференцирование в Pydrake осуществляется через типы AutoDiffXd, что требует явной работы с контекстами системы и объектами, содержащими производные. Это делает вычисление градиентов и якобианов громоздким по сравнению с JAX, в котором можно просто применить jax.jacobian() или jax.grad() к любой функции, написанной на Python.

2. Практическая часть

2.1 Постановка задачи

Разработать систему управления траекторией движения роторно-винтового БПЛА, способную эффективно контролировать параметры движения квадрокоптер при выполнении флипа, включая стабилизацию ориентации аппарата и его положение с учетом внешних возмущений и физических ограничений. Система управления должна быть адаптивной, учитывающей изменяющиеся условия полета и динамические параметры квадрокоптера, такие как изменение скорости вращения роторов, инерционные эффекты и другие факторы, влияющие на выполнение маневра.

В данной работе использовался квадрокоптер с симметричной X-образной рамой. Основные параметры дрона приведены ниже:

- масса летательного аппарата: 0.82 кг;
- постоянная времени двигателя: 0.02 с;
- момент инерции вращающихся частей мотора: 6.56×10^{-6} кг·м²;
- коэффициент тяги: 1.48×10^{-6} Н/(рад/с)²;
- коэффициент крутящего момента: 9.4×10^{-8} Н·м/(рад/с)²;
- коэффициент аэродинамического сопротивления: 0.1 Н/(м/с);
- коэффициенты аэродинамического момента по осям: 0.003 Н·м/(рад/с)² для всех трех осей (X, Y, Z);
- моменты инерции по осям: 0.045 кг·м² по каждой из осей (X, Y, Z);
- максимальная угловая скорость вращения пропеллера: 2100 рад/с;
- плечо момента (расстояние от центра масс до точки приложения тяги): 0.15 м;
- шум процессов, используемый в фильтрации силы и момента: 1.25×10^{-7} Н·м²·с и 0.0005 Н²·с соответственно;
- параметры модели шумов и начальных дисперсий для акселерометра и гироскопа заданы в пределах:
    - начальная дисперсия смещений: 1×10^{-5};
    - постоянные шумы акселерометра и гироскопа: 1×10^{-4} м²/с⁴ и 1×10^{-5} рад²/с² соответственно;

2.2 Получение и обработка данных сенсоров квадрокоптера

Для получения данных сенсоров, использовались следующие телеметрические сообщения из PX4-Autopilot:

- VehicleAttitude — ориентация дрона в виде кватерниона и угловые скорости;
- VehicleImu — данные акселерометра и гироскопа в теле дрона;
- ActuatorOutputs — нормированные управляющие сигналы моторов в диапазоне [-1;1];
- VehicleLocalPosition — оценка положения, скорости и ускорения в системе NED. Используется для тестирования и валидации в процессе симуляции;
- VehicleAngularVelocity — угловые скорости;
- VehicleAngularAccelerationSetpoint — угловые ускорения;
- VehicleMagnetometer — вектор магнитного поля. Используется в EKF для корректировки yaw;
- SensorBaro — давление с барометра для коррекции вертикальной компоненты положения;
- EscStatus — частоты вращения моторов (RPM);

Была реализована инерциальная навигация на основе данных с IMU, барометра и магнитометра. Основной задачей модуля является оценка текущей скорости и положения дрона в мировой системе координат. Навигация выполняется путем пошаговой интеграции изменения скорости, полученного с акселерометров, с учетом текущей ориентации устройства. Для перехода из локальной системы координат в глобальную используется кватернион, описывающий ориентацию дрона, с последующим преобразованием при помощи библиотеки scipy.spatial.transform.Rotation. После перевода ускорений в глобальные координаты добавляется гравитационная компонента, и производится накопление скорости и положения.

```
def vehicle_imu_callback(self, msg: VehicleImu):
delta_velocity=np.array(msg.delta_velocity, dtype=np.float32)
delta_velocity_dt = msg.delta_velocity_dt * 1e-6  # перевод из мкс
в секунды
```

```
# Проверка наличия ориентации и положительного времени интеграции
if delta_velocity_dt > 0.0:
# преобразование ориентации в мировую систему координат
rotation=Rot.from_quat(self.vehicleAttitude_q)
# преобразование ускорения в мировую систему координат
     delta_velocity_world = rotation.apply(delta_velocity)
     gravity = np.array([0.0,0.0,-9.80665],dtype=np.float32)
# учет гравитации
     delta_velocity_world+=gravity*delta_velocity_dt
# обновление линейной скорости
     self.vehicleImu_velocity_w+=delta_velocity_world
# обновление позиции
self.position+=self.vehicleImu_velocity_w*delta_velocity_dt
```

Высота определяется согласно следующей формуле

```
SEA_LEVEL_PRESSURE=101325.0
self.baro_altitude=44330.0*(1.0-(msg.pressure/SEA_LEVEL_PRESSURE)
** 0.1903)
```

Реализованный подход позволяет оценивать положение дрона в пространстве при отсутствии внешних навигационных систем, однако со временем приводит к накоплению ошибки. В случае кратковременных маневров, таких как флип, такой дрейф может быть допустим. Тем не менее, для повышения устойчивости и точности оценки состояния даже в течение коротких динамически насыщенных участков полета применяется расширенный фильтр Калмана (EKF). Он позволяет согласовывать предсказания по динамической модели с данными от инерциальных сенсоров, корректируя оценки с учетом шумов и погрешностей измерений. EKF особенно полезен в условиях нелинейной динамики дрона, помогая стабилизировать навигационную оценку при резких изменениях ориентации и ускорений.

Предсказание состояния дрона (положения, скорости, ориентации) выполняется на основе физических уравнений движения из приложения А и данных с сенсоров. Фильтрация реализована с использованием библиотеки `filterpy.kalman.ExtendedKalmanFilter`. Коэффициенты матриц процессного Q и измерительного R шума подбирались с учетом характеристик сенсоров и особенностей модели:

```
self.ekf.Q = np.diag([
0.01, 0.01, 0.001,                  # x, y, z
0.001, 0.001, 0.001,                # vx, vy, vz
0.01, 0.01, 0.01, 0.01,             # qw, qx, qy, qz
0.01, 0.01, 0.01                    # wx, wy, wz
])
self.ekf.R = np.diag([
0.05, 0.05, 0.005,                 # позиция x, y, z
0.005, 0.005, 0.005,               # скорость vx, vy, vz
0.05, 0.05, 0.05, 0.05,            # qw, qx, qy, qz
0.05, 0.05, 0.05,                  # wx, wy, wz
0.005                              # барометр
])
```

Коэффициенты матриц ковариаций процессного шума Q и измерительного шума R были определены с учетом особенностей инерциальной навигационной системы без внешних навигационных сигналов. В матрице Q для линейных параметров заданы значения порядка $10^{-3} - 10^{-2}$, отражая накопление ошибок интегрирования модели движения. Для ориентации, представленной кватернионами, и угловых скоростей установлены более высокие значения — 10^{-2}, обусловленные интеграционным дрейфом гироскопов. В матрице измерительного шума R увеличены значения для компонент, связанных с гироскопом, чтобы учитывать шум и дрейф IMU. Для барометрических высотных данных и линейных скоростей значения шума снижены, что обусловлено его более высокой точностью и стабильностью.

2.3 Разработка динамической модели квадрокоптера

В модели дрона учитываются основные параметры:

- масса — 0.82 кг;
- моменты инерции — по 0.045 кг·м² вдоль каждой из осей;
- длина плеча (расстояние от центра масс до мотора) — 0.15 м;
- коэффициенты тяги и момента мотора — 1.48×10^{-6} Н/(рад/с)² и 9.4×10^{-8} Н·м/(рад/с)² соответственно;

- максимальная угловая скорость роторов — 2100 рад/с, что соответствует приблизительно 20054 об/мин (RPM);
- учтены аэродинамическое сопротивление и ограничение на угловую скорость — не более 25 рад/с.

Дополнительно в модели:

- учтено гравитационное ускорение и переход между телесной и мировой системой координат с использованием кватернионов;
- реализован расчет тяги и управляющих моментов, создаваемых каждым мотором на основе квадрата скорости вращения роторов;
- введена нормализация кватерниона ориентации на каждом шаге для устранения накопления численной ошибки;
- учитываются гироскопические моменты, возникающие из-за взаимодействия угловой скорости и момента инерции, что особенно важно при флипе;
- применяется численное интегрирование методом Эйлера как для поступательного, так и для вращательного движения;
- учитываются силы, действующие на дрон в мировой системе координат, включая суммарную тягу, аэродинамическое сопротивление и вес;

Для моделирования динамики квадрокоптера реализована функция f(x, u, dt), реализация которой приведена в приложении А, вычисляющая новое состояние дрона на шаге интегрирования dt на основе текущего состояния x и управляющего воздействия u. Расчеты элементов вектора состояния производились в соответствии с уравнениями (1) – (28).

Кватернион нормализуется для предотвращения гимбального замка во время флипа. Из него формируется матрица поворота от тела к мировой системе координат:

```
R_bw = jnp.array(R.from_quat(quat).as_matrix())
```

Обработка управляющего сигнала

Управление u содержит RPM моторов. Для физического моделирования они ограничиваются максимальным значением и возводятся в квадрат:

```
rpm = jnp.clip(u, 0.0, max_speed)
```

```
w_squared = rpm ** 2
thrusts = kf * w_squared  # Тяга каждого мотора
```

Тяга каждого мотора суммируется и используется для расчета результирующей силы в теле дрона:

```
Fz_body = jnp.array([0.0, 0.0, jnp.sum(thrusts)])
```

Эта сила преобразуется в мировую систему координат с учетом гравитации и сопротивления воздуха:

```
F_world = R_bw @ Fz_body - m * g - drag * vel
acc = F_world / m
```

На основе полученного ускорения интегрируются скорость и положение:

```
new_vel = vel + acc * dt
new_pos = pos + vel * dt + 0.5 * acc * dt ** 2
```

Расчет моментов

Моменты создаются разностью тяг моторов и учитывают симметрию конструкции:

```
tau = jnp.array([
arm * (thrusts[1] - thrusts[3]),  # Момент roll
arm * (thrusts[2] - thrusts[0]),  # Момент pitch
km * (w_squared[0] - w_squared[1] + w_squared[2] - w_squared[3])  # Момент yaw
])
```

Добавляется кориолисово слагаемое

```
omega_cross = jnp.cross(omega, I @ omega)
```

Угловое ускорение рассчитывается с помощью инверсии тензора инерции:

```
omega_dot = jnp.linalg.solve(I, tau - omega_cross)
new_omega = omega + omega_dot * dt
```

Кватернионное приращение вычисляется через угловую скорость:

```
omega_quat = jnp.concatenate([jnp.array([0.0]), omega])
dq = 0.5 * self.quat_multiply(quat, omega_quat)
new_quat = quat + dq * dt
new_quat /= jnp.linalg.norm(new_quat + 1e-8)
```

На выходе формируется объединенное состояние:

```
x_next = jnp.concatenate([new_pos, new_vel, new_quat, new_omega])
```

Таким образом, функция возвращает новое состояние дрона, а именно:

- положение и линейную скорость в мировой системе координат;
- кватернион ориентации;
- угловую скорость в теле дрона.

Демонстрация оценки параметров дрона

В ходе тестирования системы навигации было выполнено логирование состояния дрона при подъеме на высоту 5 метров. Среда симуляции предоставляет эталонные координаты дрона, которые используются для сравнения с оценками подсистемы. Как показано на рисунке 5, оценка состояния успешно сглаживает фактическую траекторию, демонстрируя точное и стабильное поведение алгоритма. На рисунке 6 видно, что высокая оценка ошибки по X в процессе достигала максимума около 0.5 метров, а затем снижалась до примерно 0.1 метра. По Y наблюдался резкий скачок ошибки, который также был успешно сглажен. Наиболее стабильную и точную оценку обеспечивает Z благодаря использованию данных барометра. В целом, оценка состояния эффективно сглаживает фактическую траекторию, демонстрируя точность и стабильность работы алгоритма.

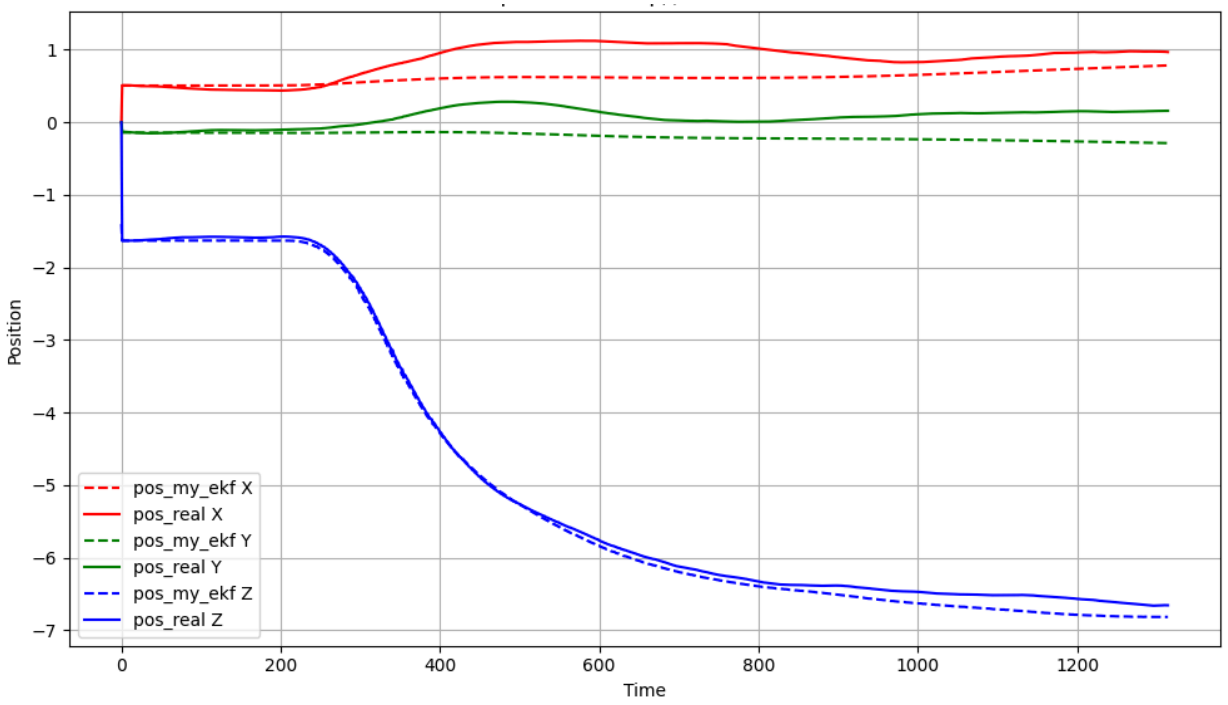

Рисунок – 5 Измерения X,Y,Z

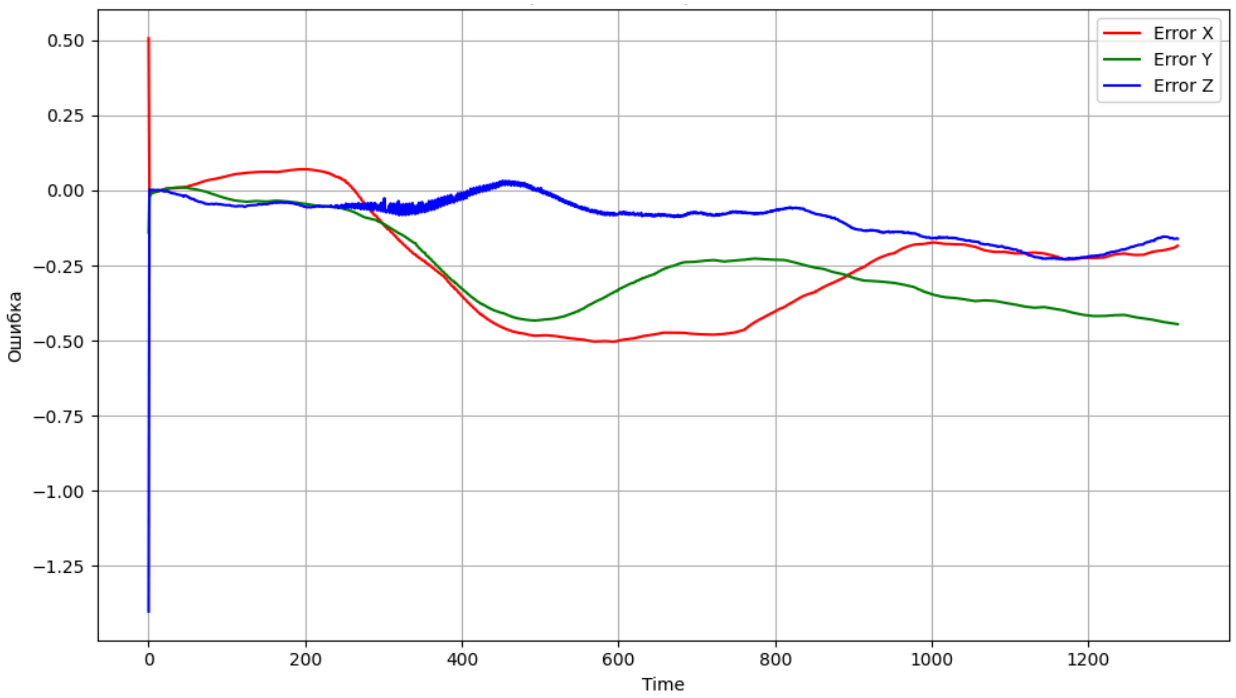

Рисунок – 6 Погрешности координат X,Y,Z

Из рисунков 7, 8 видно, что при плавном подъеме в точку (0.0, 0.0, 5.0) и последующем зависании ориентационные параметры демонстрируют незначительные колебания, возникающие в процессе стабилизации. Эти колебания постепенно затухают после срабатывания регуляторов (PID-контуров) дрона, обеспечивая устойчивое поведение в целевой позиции.

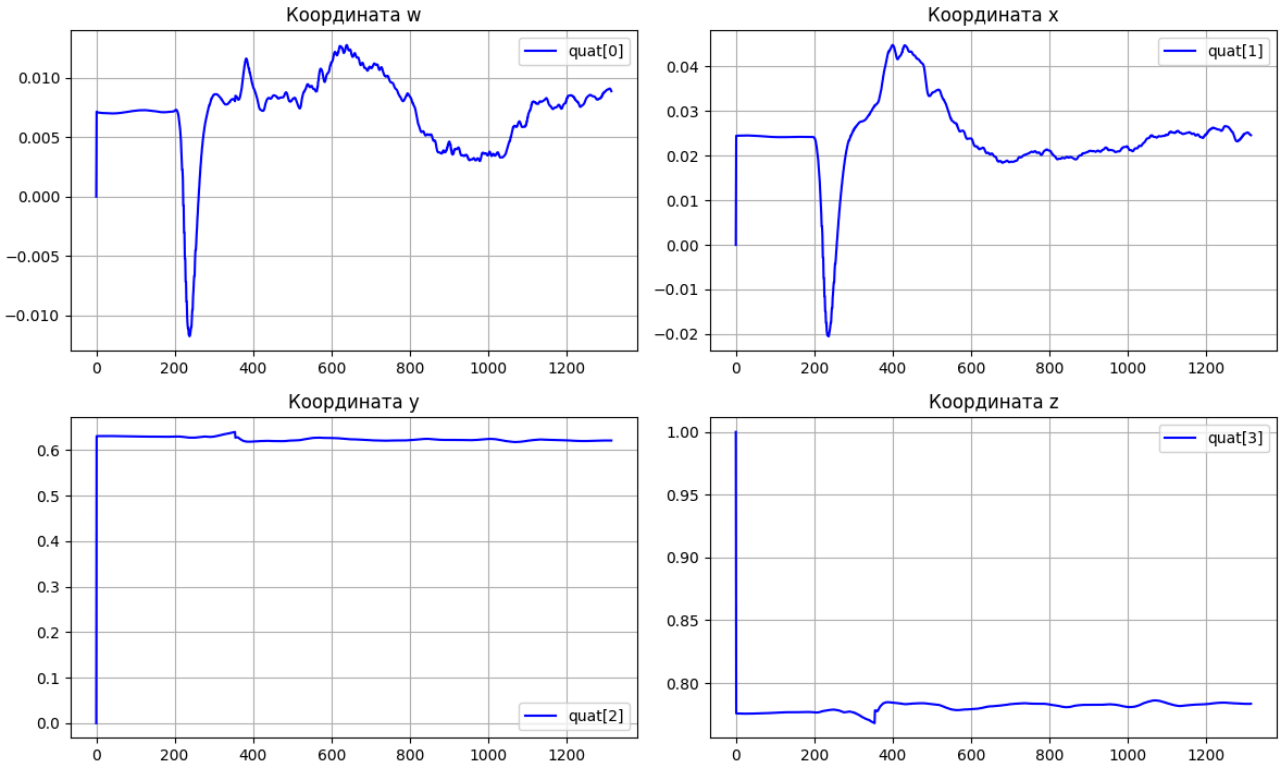

Рисунок – 7 Изменения ориентации в кватернионной форме

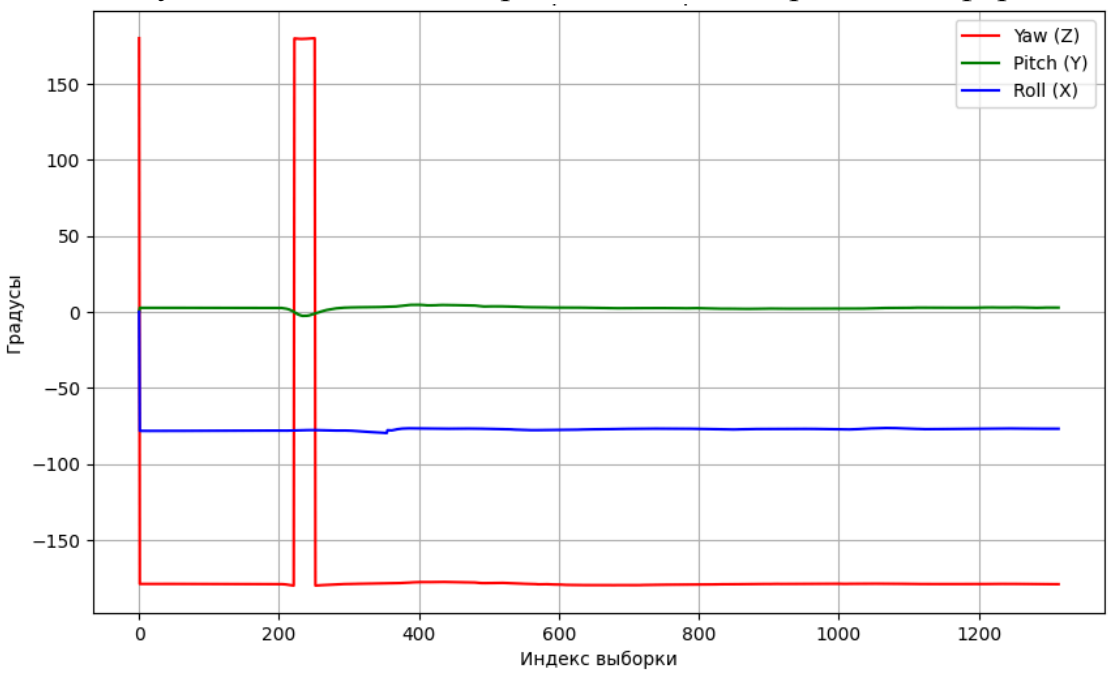

Рисунок – 8 Углы Эйлера

На рисунке 9 представлена динамика скорости вдоль Z. Видно, что во время подъема происходит нарастание вертикальной скорости, которая постепенно снижается по мере приближения к заданной высоте. Это поведение соответствует фазе активного набора высоты с последующим переходом к стабилизированному зависанию.

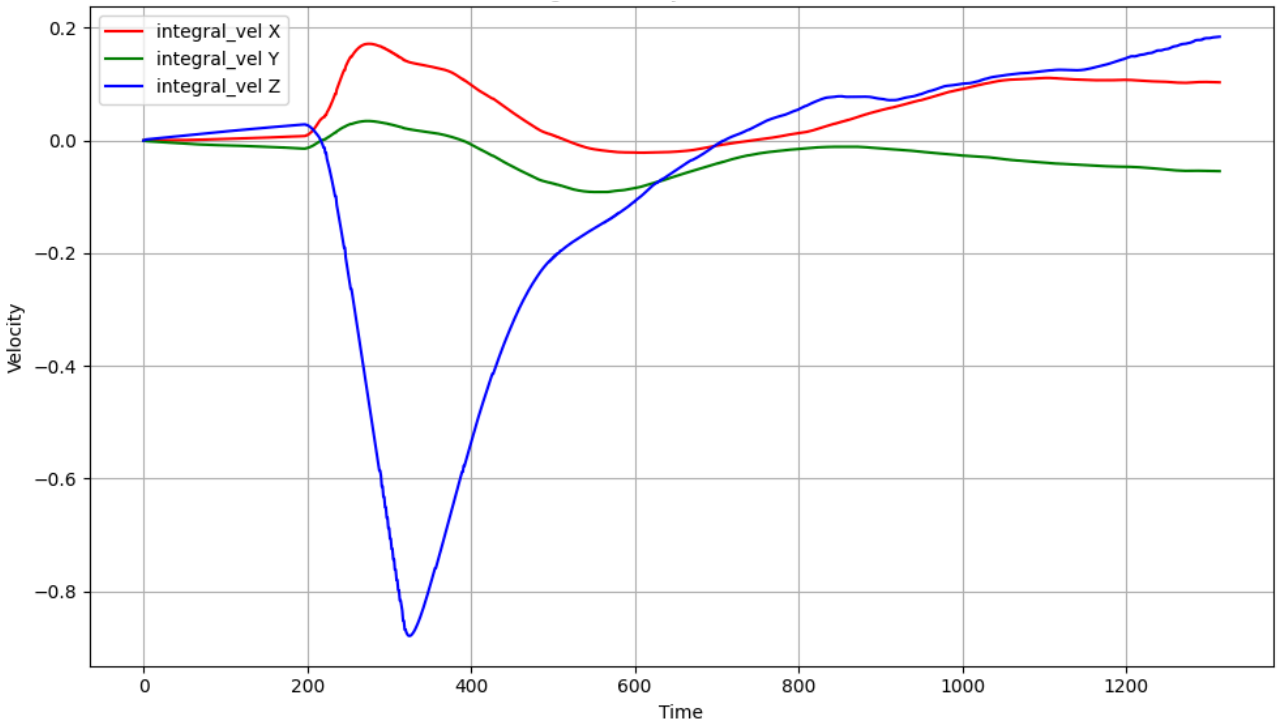

Рисунок – 9 Скорость по осям x,y,z

На рисунке 10 показано поведение угловой скорости в момент взлета. В начале наблюдаются флуктуации, обусловленные активной работой регуляторов в фазе подъема. После стабилизации дрона и достижения стационарного положения угловая скорость уменьшается и остается практически постоянной, что соответствует неподвижности аппарата по всем осям.

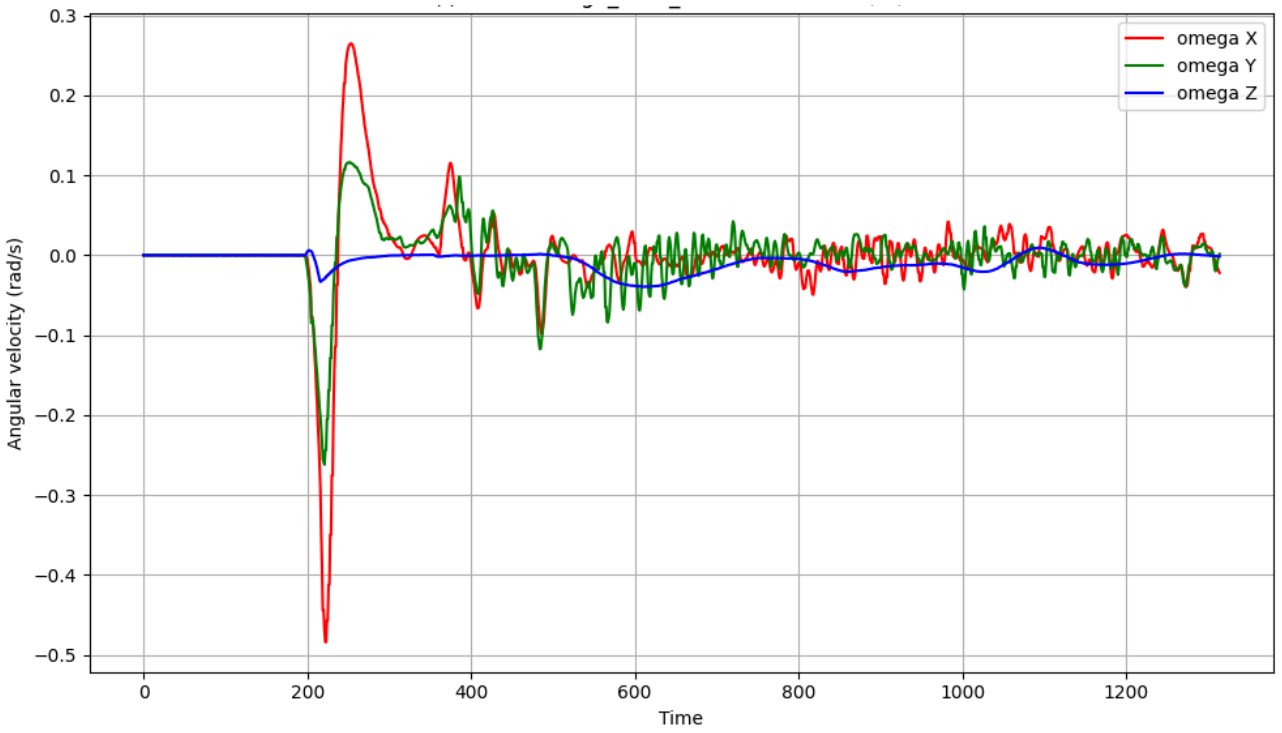

Рисунок – 10 Угловая скорость по осям x,y,z

Таким образом, реализована реалистичная модель, пригодная для использования в симуляциях и контроллерах, включая MPC управление и EKF-оценивание состояния.

2.4 Контролируемое выполнение флипа квадрокоптером

В ходе работы была реализована система оптимального управления на основе метода iLQR с использованием библиотеки JAX.

Функция simulate_trajectory реализует вычисление траекторий для батча управляющих воздействий с использованием JAX vmap, что позволяет эффективно выполнять параллельное вычисление следующего состояния для каждого управляющего сигнала:

```
@jit
def simulate_trajectory(x0, U):
#инициализация списка состояний
    X = [x0]
#векторизация функции f по управляющему воздействию
    f_batch = vmap(f, in_axes=(None, 0, None))
    U = jnp.array(U)
#вычисление новых состояний
    X_new = f_batch(x0, U, dt)
    X.extend(X_new)
    return jnp.stack(X)
```

Линеаризация динамики — вычисление матриц Якоби для перехода по состоянию и управлению:

```
@jit
def linearize_dynamics(x, u):
    A = jacobian(f, argnums=0)(x, u, dt)
    B = jacobian(f, argnums=1)(x, u, dt)
    return A, B
```

Квадратизация функции стоимости — вычисление градиентов и гессианов по состоянию и управлению:

```
@jit
def quadratize_cost(x, u, x_target, u_target, Q, R):
# Определение скалярной стоимости по отклонению от цели
    def scalar_cost(x_, u_):
        dx = x_ - x_target
        du = u_ - u_target
        return dx @ Q @ dx + du @ R @ du
# Градиент стоимости по состоянию
    lx = grad(scalar_cost, argnums=0)(x, u)
# Градиент стоимости по управлению
    lu = grad(scalar_cost, argnums=1)(x, u)
# Гессиан по состоянию
```

```
    lxx = hessian(scalar_cost, argnums=0)(x, u)
# Гессиан по управлению
    luu = hessian(scalar_cost, argnums=1)(x, u)
# Якобиан градиента по управлению по состоянию
    lux=jacobian(grad(scalar_cost,argnums=1),argnums=0)(x, u)
    return lx, lu, lxx, luu, lux
```

Ключевой частью обратного прохода является решение уравнений Риккати с регуляризацией для численной устойчивости:

```
@jit
def backward_pass(X, U, x_target_traj, u_target_traj, Q, R, Qf):
    ...
#регуляризация Quu для обеспечения обратимости
    Quu_reg = Quu + 1e-6 * jnp.eye(m)
#инвертирование регуляризованной матрицы Quu
    Quu_inv = jnp.linalg.inv(Quu_reg)
#матрица обратной связи по состоянию
    K = -Quu_inv @ Qux
#компонент управления, не зависящий от состояния
    kff = -Quu_inv @ Qu
    ...
    return K_list, k_list
```

Обновление траектории и управляющих сигналов с учетом коэффициентов оптимизации:

```
@jit
def forward_pass(X, U, k_list, K_list, alpha):
    ...
    # Цикл по всему горизонту оптимизации
    for k in range(horizon):
        # вычисление отклонения
        dx = x - X[k]
        # расчет корректировки управляющего воздействия:
           # базовое управляющее воздействие k_list плюс обратная
связь K_list умноженная на dx
        du = k_list[k] + K_list[k] @ dx
         #Обновление управляющего воздействия с учетом шага альфа
(линейное интерполирование)
        u_new = U[k] + alpha * du
        # Сохранение обновленного управления
        U_new.append(u_new)
     # Прогоняем систему вперед: считаем новое состояние, используя
функцию динамики f, текущий dt
        x = f(x, u_new, dt)
        …
```

В функции стоимости (см. в приложении Б) реализована квадратичная оптимизация с использованием матриц весов. Были подобраны специфические коэффициенты стоимости, отражающие физику устройства и особенности управления во время быстрого маневра.

Весовая матрица состояния Q определяет, какие компоненты состояния наиболее критичны в процессе флипа.

```
Q = jnp.diag(jnp.array([
    1.0, 1.0, 10.0,       # x, y — менее важны, z — важна
    1.0, 1.0, 1.0,        # vx, vy, vz
    0.0, 50.0, 50.0, 0.0, # ориентация
    5.0, 5.0, 1.0         # угловые скорости
]))
```

В частности:

- высокий вес по оси z отражает важность удержания высоты, что помогает уменьшить просадку дрона при выполнении маневра, особенно в фазе переворота, когда тяга частично направлена в сторону;
- координаты по осям x и y получают сравнительно меньшие веса в матрицах стоимости, поскольку в процессе флипа допустимы умеренные смещения в горизонтальной плоскости. Жесткое ограничение этих параметров может привести к усложнению задачи управления и ухудшению устойчивости. Кроме того, в используемой системе позиционирование по X и Y основано исключительно на инерциальных данных, подверженных накоплению ошибок, что делает чрезмерное стремление к точности в этих осях нецелесообразным в контексте быстрого маневра;
- ориентация по осям qx и qy (50.0) придает важность крену и тангажу — они критичны для корректного выполнения флипа. Компоненты qz и qw не штрафуются напрямую, так как они частично зависят от qx/qy при нормализации кватерниона;
- угловые скорости по X и Y (5.0) также имеют значительный вес: именно они определяют скорость и симметрию вращения, а точный контроль здесь позволяет избежать излишнего кручения и обеспечивает быстрое восстановление;
- угловая скорость по оси Z менее важна (1.0), так как поворот вокруг вертикали не играет ключевой роли в флипе по роллу.

Матрица управления R задает штраф за управляющее воздействие, то есть

за усилия моторов.

```
R = jnp.diag(jnp.array([
0.001, 0.001, 0.001, 0.001  # все моторы слабо штрафуются
]))
```

Малые значения выбраны осознанно — система не ограничивается в мощности, позволяя использовать максимум доступного управления, что особенно важно при выполнении резких, кратковременных маневров, как флип.

Матрица конечной стоимости Qf усиливает требования к состоянию дрона в конце маневра.

```
Qf = jnp.diag(jnp.array([
    1.0, 1.0, 10.0,  # позиции: x,y — менее важны, z — важна
    0.1, 0.1, 0.1,           # скорости
    0.0, 100.0, 100.0, 0.0, # ориентация (qx, qy)
    10.0, 10.0, 1.0          # угловые скорости
]))
```

Здесь:

- позиция по Z сохраняет высокий вес (10.0), чтобы выделить важность выйти точно на заданную высоту после флипа;
- ориентация по qx и qy (100.0) становится еще важнее, поскольку необходимо остановиться в целевой ориентации;
- угловые скорости по X и Y получают повышенные веса, чтобы дрон завершил маневр без остаточного вращения.

Таким образом, была реализована полная итеративная процедура iLQR с автоматическим вычислением производных с помощью JAX, что позволило эффективно решать задачу оптимального управления нелинейной динамикой квадрокоптера при выполнении маневра флипа. Полная реализация iLQR регулятора приведена в приложении Б.

Планировщик отвечает за генерацию управляющей траектории для выполнения маневра флипа квадрокоптера. Основу логики составляет конечный автомат, реализующий четыре фазы: взлет, флип, восстановление и посадка. Переходы между фазами выполняются только при достижении заданных условий по высоте, крену иориентации, что обеспечивает устойчивость. Реализация

контроллера приведена в приложении В.

Главная особенность реализации — адаптивность управления флипом через обратную связь. В процессе флипа текущий угол крена сравнивается с теоретически ожидаемым, рассчитываемым по времени внутри фазы.

```
#вычисление локального времени от начала флипа, ограничивая его интервалом [0,flip_duration]
t_local=np.clip(current_time-self.flip_started_time,0.0,self.flip_duration)

#ожидаемое значение угла крена в текущий момент времени,
# предполагается линейное изменение от 0 до 2π за время маневра
roll_expected=2*np.pi*t_local/self.flip_duration
# Получение текущего угла крена из текущего кватерниона ориентации
roll_current,_,_=self.euler_from_quaternion(q_current)

#ошибка между ожидаемым и текущим креном
roll_error=roll_expected-roll_current
#адаптивное усиление увеличивается при отклонении roll от ожидаемого
# Используется функция tanh для сглаживания и ограничения значения добавки
gain_adaptive=gain_base+0.3*np.tanh(roll_error)
```

На основе ошибки ориентации и текущего времени в фазе флипа вычисляется адаптивный прирост целевого крена. То есть, если реальный поворот отстает от ожидаемого, узел усиливает целевое значение roll и угловую скорость, компенсируя отклонения.

Переходы между фазами выполняются исключительно на основе обратной связи — переход в следующее состояние происходит только при достижении требуемых условий: заданной высоты, близости roll к 2π, возврата ориентации к горизонтальной. Это повышает устойчивость управления и позволяет системе быть устойчивой к нестандартным динамикам квадрокоптера.

Таким образом, данный узел реализует замкнутую по обратной связи систему генерации управляющих воздействий, устойчиво выполняющую сложный маневр флипа с использованием адаптивных параметров и текущего состояния дрона.

Для передачи управляющих воздействий на автопилот PX4 в формате [roll, pitch, yaw, thrust] реализована система преобразования RPM в управляющие моменты и тягу. Преобразованные значения нормализуются следующим образом: компоненты roll, pitch и yaw — в диапазоне от -1 до 1, thrust — от 0 до 1.

```
    def rpm_to_control(rpm, arm, kf, km):
        w_squared = rpm ** 2
        thrusts = kf * w_squared
        thrust = jnp.sum(thrusts)
        roll_torque  = arm * (thrusts[1] - thrusts[3])   # M2 - M4
        pitch_torque = arm * (thrusts[2] - thrusts[0])   # M3 - M1
        yaw_torque      = km * (w_squared[0] - w_squared[1] +
w_squared[2] - w_squared[3])
        return jnp.array([roll_torque, pitch_torque, yaw_torque,
thrust])
```

Нормализованные управляющие воздействия публикуются в формате сообщения ActuatorControls в топик /fmu/in/actuator_controls_0 через интерфейс PX4 Fast RTPS Bridge. Для приема таких команд автопилот PX4 предварительно настраивается: активируется режим OFFBOARD и включается система распределения управления путем установки параметра SYS_CTRL_ALLOC = 1. Программная реализация данного модуля приведена в приложении Г.

Настройка частот выполнения функций

Для выбора времени кванта управления ограничим возможные значения сверху и снизу. Типовое значение для дронов при обычном полете составляет 100 мс и выше, что достаточно для выполнения итерации при навигации в спокойном полете по траектории. Так как для выполнения флипа требуются данные реальных сенсоров, их приемлемые характеристики соответствуют времени 20 мс, для меньших значений требуются скоростные специализированные сенсоры давления, акселерометра, гироскопа и магнитометра. Обеспечение фильтрации данных также требует времени порядка 20 мс для МК полетного контроллера. Поэтому для моей системы время выбрано в 20 мс.

2.5 Демонстрация работы

Для тестирования и демонстрации работы разработанной системы управления использовалась среда симуляции Gazebo, в которой был создан виртуальный мир и модель квадрокоптера (см. Приложение Д). Учтены масса, инерция, аэродинамика и динамика моторов, что позволяет проводить испытания в условиях, приближенных к реальным.

Для оценки производительности алгоритма во время флипа, собирались временные метки начала и окончания расчета управляющего воздействия на каждой итерации оптимизации (см. Приложение Ж).

Результаты измерений изображены на рисунках 11, 12.

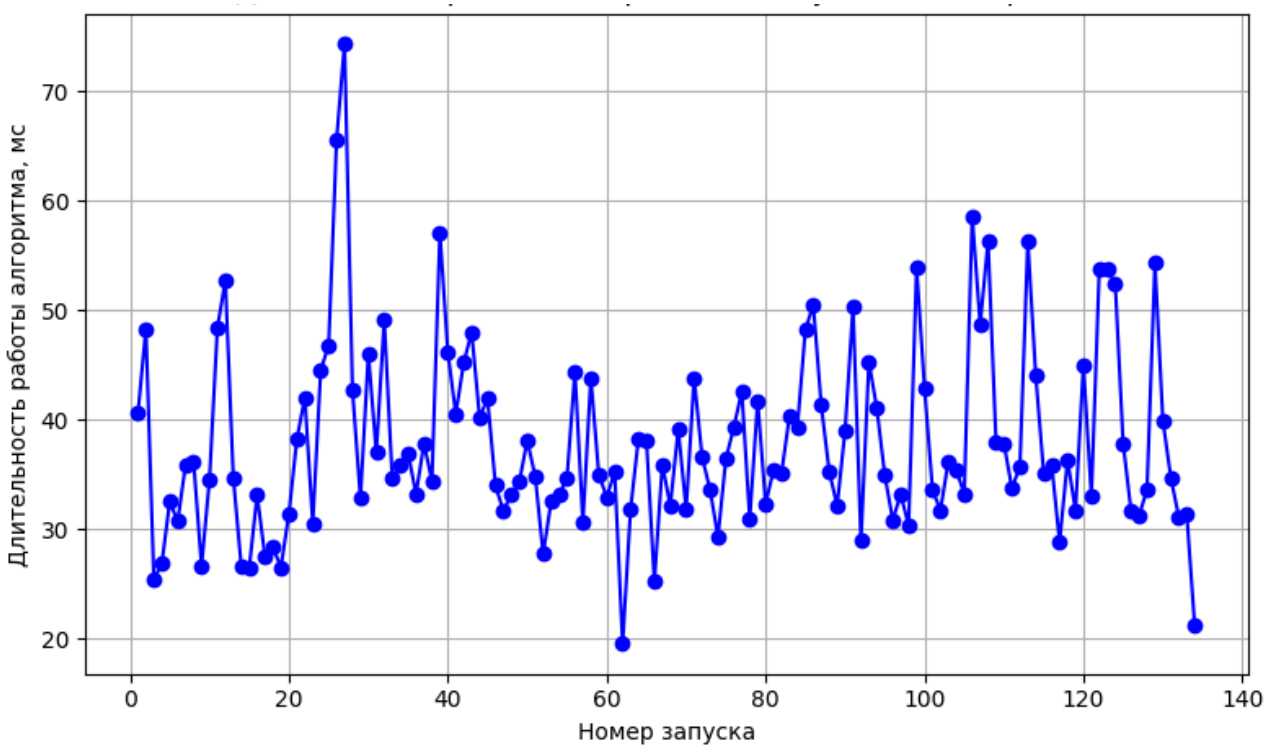

Рисунок – 11 Длительность работы алгоритма поиска оптимальной траектории

Как видно из графика, среднее время расчета оптимальной траектории составляет ~50 мс, что соответствует частоте управления 20 Гц. Таким образом, результаты симуляционных испытаний подтвердили, что время расчета соответствует требованиям мягкого реального времени, что обеспечивает возможность эффективного управления динамикой дрона во время выполнения флипа.

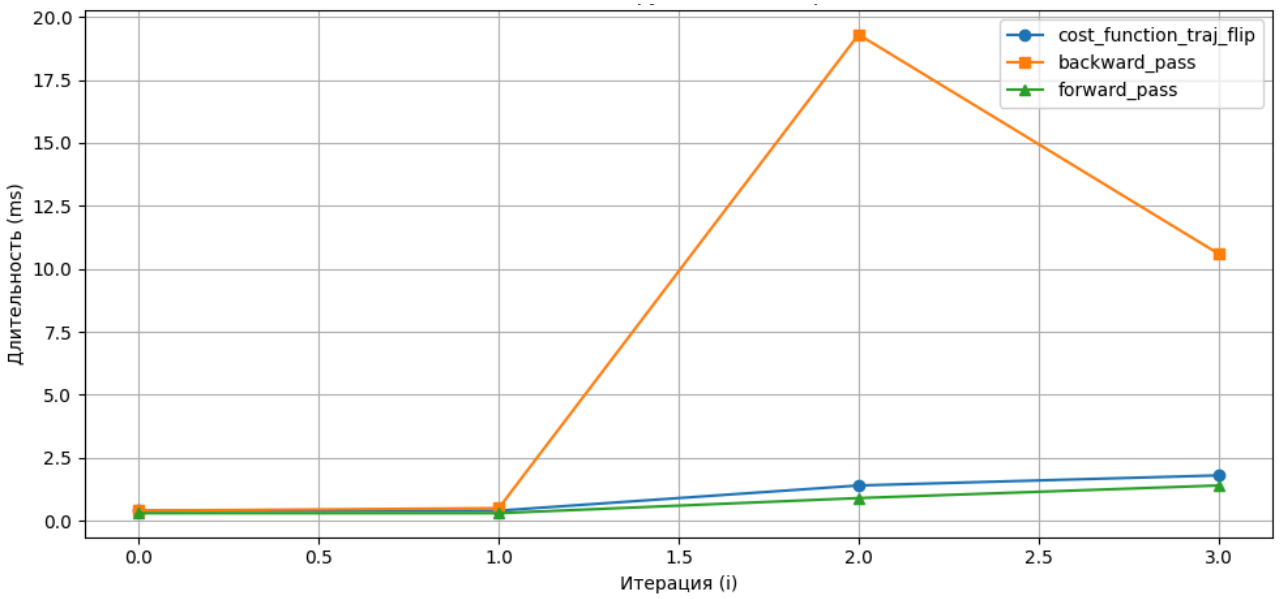

Рисунок – 12 Длительность этапов алгоритма поиска оптимальной траектории

Динамика позиции и ориентации дрона в момент флипа изображена на рисунках 13, 14 соответственно.

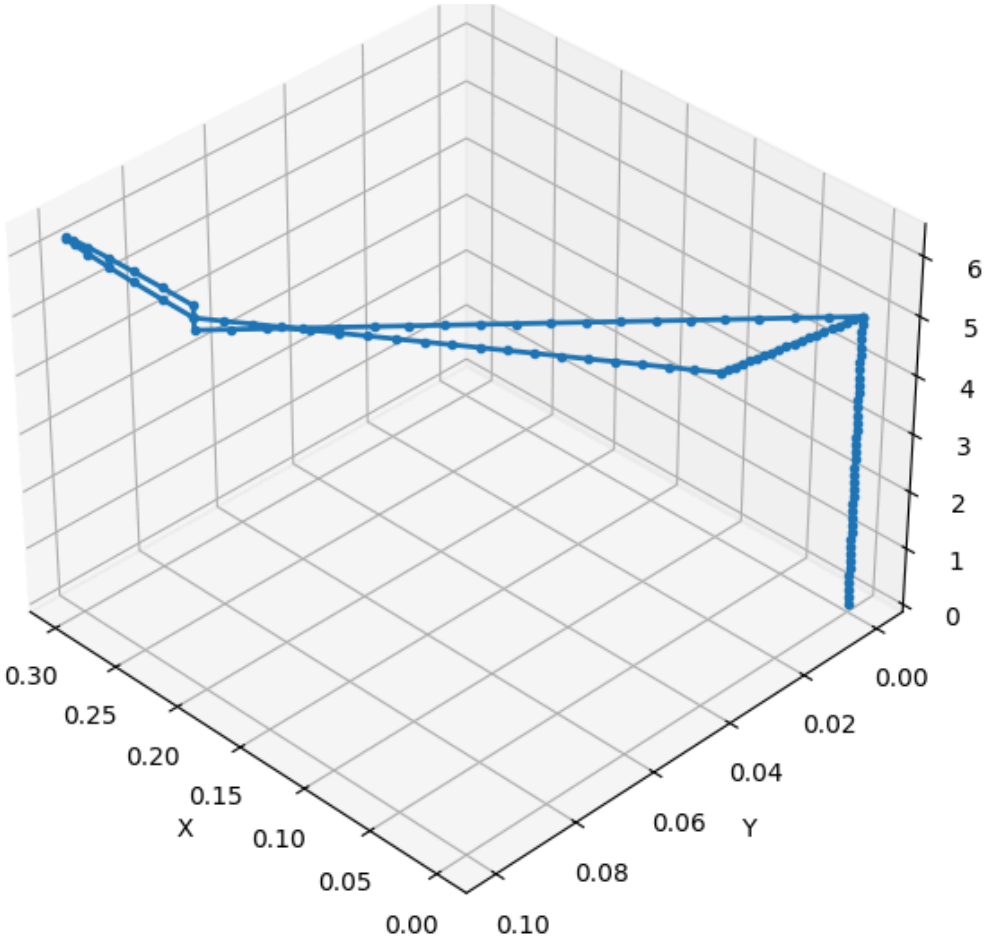

Рисунок – 13 Траектория дрона во время флипа

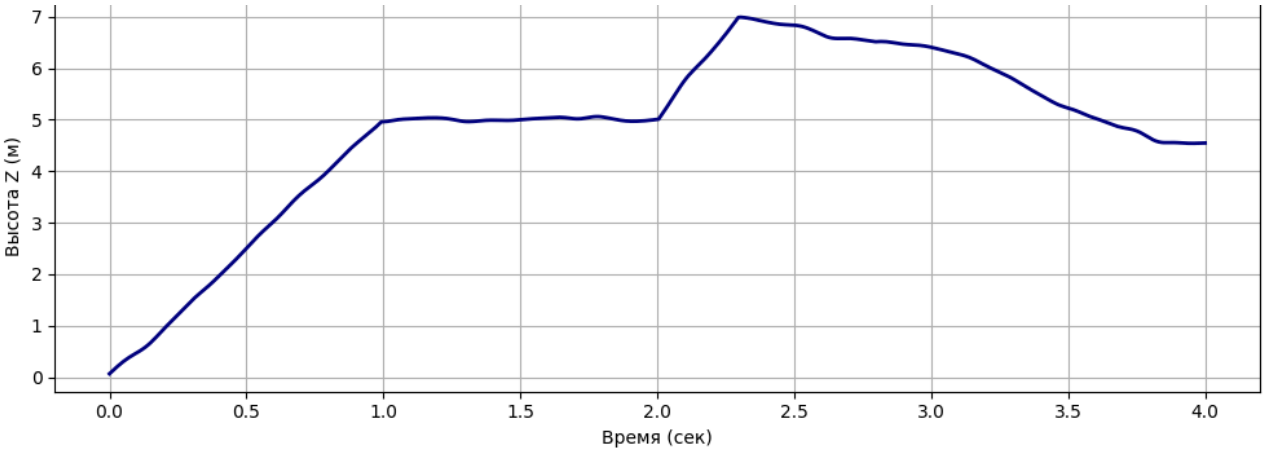

Рисунок – 14 Траектория дрона по высоте

Как видно из рисунков 13 и 14, в процессе выполнения флипа дрон испытывает незначительные отклонения по осям XY — около 0.27 м, что сопоставимо с размером дрона. Также выделяется умеренное снижение по оси OZ — порядка 2 м.

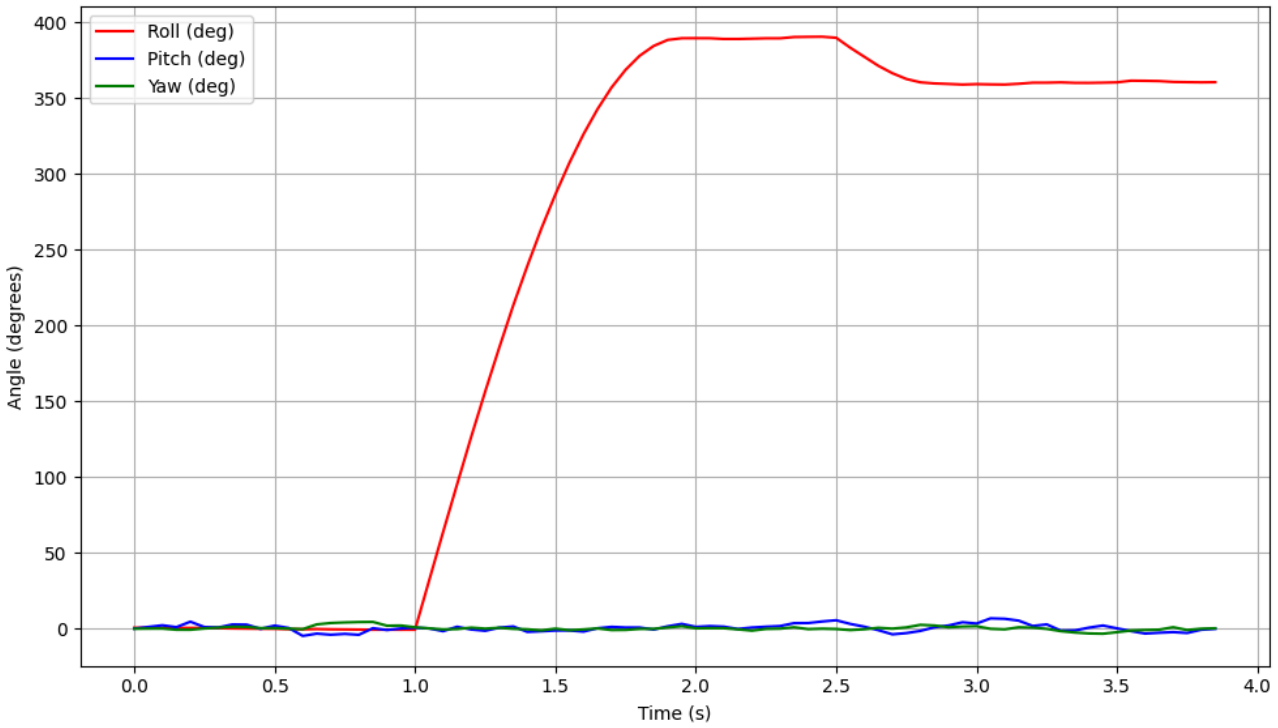

Рисунок – 15 Изменение ориентации дрона во время флипа

На рисунке 15 показано, что дрон успешно совершил флип по оси ролла, компенсировав небольшое смещение, возникшее в ходе маневра. После перелета примерно на 20 градусов дрон остановил вращение и восстановил устойчивое положение за 0.5 секунды.

Для сравнительного анализа также было разработано упрощенное решение задачи (см. приложение И), основанное на открытом фрагменте кода выполнения

маневра флип с платформы Clover[4]:

```
void controlFlip(){
    if (t-lastTime>0.1) {
        flipStage=WAIT;
        stageTime=t;
    }
    lastTime=t;
    if (flipStage==ACCELERATE) {
        thrustTarget=0.1;
        if (t-stageTime>ACCELERATE_TIME) {
            flipStage=ROTATE;
            stageTime=t;
        }
    } else if (flipStage==ROTATE) {
        thrustTarget=0.2;
    // ETC …
```

Приведенный выше фрагмент реализации флипа, основанный на фиксированных временных интервалах и заранее заданных угловых скоростях не учитывает важные динамические характеристики квадрокоптера — такие как взаимодействие тяговых сил, моменты инерции, аэродинамическое сопротивление и гироскопические эффекты. Такой упрощенный подход ограничивает точность и устойчивость управления в ходе выполнения маневра. На практике реализация данного подхода оказалась неэффективной: подбор временных интервалов не дал стабильных результатов, а жестко заданная длительность фаз полета не позволила корректно завершить флип. В связи с этим подход был модифицирован — переход между стадиями маневра стал зависеть от обратной связи с дрона, в частности, от текущего значения угла крена.

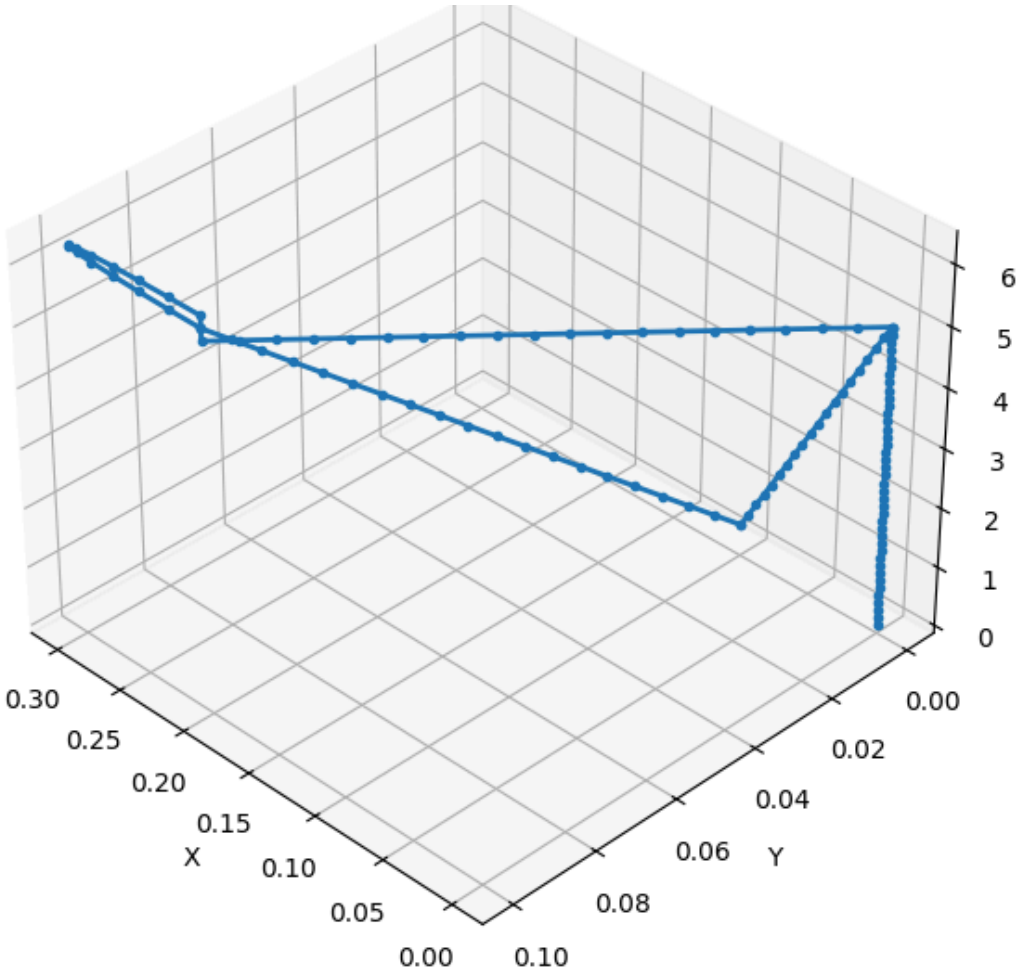

Рисунок – 16 Траектория позиции дрона в момент реализации флипа (примитивный алгоритм)

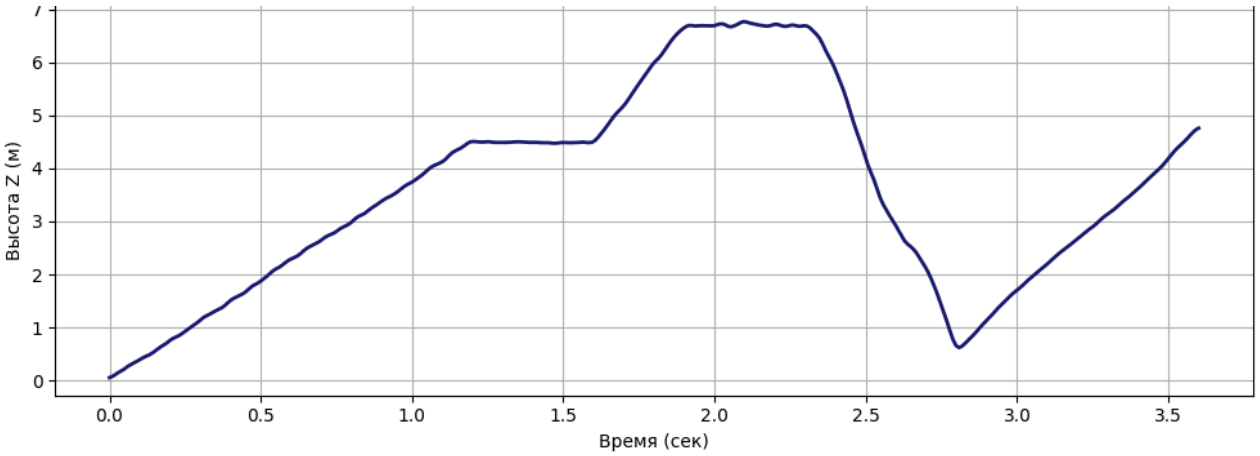

Рисунок – 17 Динамика дрона по оси OZ

Анализ графиков на рисунках 16 и 17 показывает, что при использовании простого управления наблюдается снижение по оси OZ, превышающее аналогичный показатель оптимизированного решения, основанного на модели, более чем в два раза.

ЗАКЛЮЧЕНИЕ

Таким образом все поставленные задачи были выполнены. И поставленная цель была достигнута. Ключевой вклад заключается в разработке адаптивной системы управления на основе динамической модели квадрокоптера с использованием оптимизатора iLQR. Для проведения тестирования была создана специализированная симуляционная среда на базе Gazebo, позволяющая верифицировать алгоритмы в условиях, приближенных к реальным.

Реализованная система управления структурирована и модульна: алгоритмы разбиты на фазы и снабжены полной отрицательной обратной связью, что позволяет оформить их как библиотечные модули и интегрировать в другие ROS/ROS 2-системы.

Сравнительный анализ показал заметные преимущества предложенного подхода по сравнению с примитивным управлением. Как демонстрируют результаты моделирования, система на базе iLQR позволяет существенно снизить просадку по высоте и обеспечить более устойчивое и предсказуемое поведение дрона при выполнении сложных маневров, таких как флип. Снижение по оси OZ при использовании модели оказалось более чем в два раза меньше по сравнению с простым решением. При этом отклонения по осям XY (около 0.27 м) сопоставимы с размерами дрона и считаются допустимыми, а восстановление после флипа произошло за 0.5 секунды.

Несмотря на наличие научных публикаций, в которых рассматривается применение iLQR для управления квадрокоптерами, на данный момент не существует открытых проектов, реализующих такой подход на стеке ROS2, PX4. Таким образом, разработка представляет собой уникальный пример интеграции современных средств робототехники с оптимизационным управлением в мягком реальном времени, что делает проект актуальным и практически значимым.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Правительство РФ. Распоряжение Правительства Российской Федерации от 21 июня 2023 г. № 1630-р. Об утверждении Стратегии развития беспилотной авиации РФ на период до 2030 г. и на перспективу до 2035 г. и плана мероприятий по ее реализации / Правительство РФ // Собрание законодательства Российской Федерации. - М. : Юридическая литература, 2023. - № 27.

2 Терентьев, В.М. Развитие теоретических основ и новые методологические подходы для разработки системы траекторного управления перспективными беспилотными летательными аппаратами / В.М. Терентьев. - М. : Инновационное машиностроение, 2023.

3 Локерсио А. и др. Обучение высокоскоростному полету в дикой природе // Science Robotics. - 2021.- Т. 6, № eabg5810. - DOI: 10.1126/scirobotics.abg5810.

4 Clover snippets [Электронный ресурс] : документация по управлению дроном /; 2024- . clover.coex.tech - URL: https://clover.coex.tech/ru/snippets.html#flip (дата обращения: 20.02.2025). - Загл. с экрана. - Последнее изменение страницы: 30.02.2024. - Яз. рус

5 В.С. Фетисов Л.М. Неугодникова, В.В. Адамовский Р.А. Красноперов. Беспилотная авиация: терминология, классификация, современное состояние. - Уфа : Фотон, 2014.

6 Зеленский В.А., Ковалев М.А., Овакимян Д.Н., Кириллов В.С. Математическая модель обобщенной кинематической схемы квадрокоптера и ее программная реализация // Вестник Самарского университета. Аэрокосмическая техника, технологии и машиностроение. 2024. Т. 23, No 1. DOI: 10.18287/2541-7533-2024-23-1-7-20

7 Калягин М.Ю., Волошин Д.А., Мазаев А.С. Моделирование системы управления полетом квадрокоптера в среде Simulink и Simscape Multibody // Труды МАИ. Выпуск № 112, 2020, DOI: 10.34759/trd-2020-112-20

8 Г.В. Лысухо, А.Л. Масленников. Квадрокоптер: динамика и управление //

Политехнический молодежный журнал, 2020, №5. DOI: 10.18698/2541-8009-2020-04-604.

9 Sun, R.; Zhang, W.; Zheng, J.; Ochieng, W.Y. GNSS/INS Integration with Integrity Monitoring for UAV No-fly Zone Management. Remote Sens. 2020, 12, 524. DOI:10.3390/rs12030524

10 Hiba, A.; Gáti, A.; Manecy, A. Optical Navigation Sensor for Runway Relative Positioning of Aircraft during Final Approach. Sensors 2021, 21, 2203. DOI:/10.3390/s21062203

11 Ян Ган, Мэн Вэй-Да, Дун Хоу-Го, Фэн Нин-Нин Использование ключевых точек и линий изображения в задаче визуально-инерциальной одометрии в реальном времени // Гироскопия и навигация. - 2023. - Том 31. №4 С. 96, 113.

12 INAV: Navigation-enabled flight control software. GitHub [Электронный ресурс] : веб-сервис для хостинга IT-проектов и их совместной разработки; - GitHub®, 2025- . - URL: https://github.com/iNavFlight/inav/tree/master/src/main/target (дата обращения: 1.05.2025). - Загл. с экрана. - Последнее изменение страницы: 2 июня 2024 года. - Яз. англ.

13 Betaflight: Navigation-enabled flight control software. GitHub [Электронный ресурс] : веб-сервис для хостинга IT-проектов и их совместной разработки; - GitHub®, 2025- . - URL: https://github.com/betaflight/betaflight/tree/master/src/main/target (дата обращения: 1.05.2025). - Загл. с экрана. - Последнее изменение страницы: 2 июня 2024 года. - Яз. англ.

14 ArduPilot Development Team. ArduPilot - Open Source Autopilot Software. URL: https://ardupilot.org/ (дата обращения: 20.02.2025).

15 PX4 Development Team. PX4 Autopilot - Open Source Flight Control Software. URL: https://px4.io/ (дата обращения: 20.02.2025).

16 Habr [Электронный ресурс] : открытый репозиторий ПО / Использование UAVCAN для модульной электроники БПЛА, или как не спалить дрона, перепутав провода ; 2020- . - URL:

https://habr.com/ru/companies/innopolis/articles/513192/ (дата обращения: 20.04.2024). - Загл. с экрана. - Последнее изменение страницы: 30 июл 2020 в 20:26. - Яз. рус.

17 Голубков А.В. Моделирование движения объекта по сложной траектории с обнаружением изменения и идентификацией режимов движения : дис. канд. физико-математических наук : 05.13.18 : защищена 30.05.2022

18 Гибридные нейро-стохастические модели обработки первичной информации в системах железнодорожной автоматики / А. И. Долгий, И. Д. Долгий, В. С. Ковалев, С. М. Ковалев // Известия ВолгГТУ. - 2011. - № 11 9(82).

19 Точилин, П. А. Задачи достижимости и синтеза управлений для гибридных систем / П. А. Точилин, А. Б. Куржанский. - МГУ, 2008., Точилин, П. А. Задачи достижимости и синтеза управлений для гибридных систем : Автореферат; канд. физ.-мат. наук: 01.01.02 / П. А. Точилин ; Место защиты: Моск. гос. ун-т им. М.В. Ломоносова. - 2008.

20 Шаошань Лю Лиюнь Ли, Цзе Тан Шуаш Ву Жан-Люк Годье. Разработка беспилотных транспортных средств; Под ред. В. С. Яценков. - М. : ДМК Пресс, 2021.

21 Степанов, О. Алгоритм планирования информативного маршрута в задаче навигации с использованием карты / О.А. Степанов, А.С. Носов // XXVIII Санкт-Петербургская международная конференция по интегрированным навигационным системам : сборник материалов, 31 мая – 02 2021 года. — Санкт-Петербург : "Концерн "Центральный научно-исследовательский институт "Электроприбор 2021.

22 Y. Pan, Q. Lin, H. Shah и JM Dolan, «Безопасное планирование для самостоятельного вождения с помощью адаптивного ограниченного ILQR», Международная конференция IEEE/RSJ по интеллектуальным роботам и системам (IROS) 2020 г. , Лас-Вегас, Невада, США, 2020 г., стр. 2377–2383, doi: 10.1109/IROS45743.2020.9340886.

23 Pan, Jia. Collision-free and smooth trajectory computation in cluttered environments / Jia Pan, Liangjun Zhang, Dinesh Manocha // The International

Journal of Robotics Research. - 2012. - Vol. 31.

24 Zhou, Boyu & Gao, Fei & Wang, Luqi & Liu, Chuhao & Shen, Shaojie. (2019). Robust and Efficient Quadrotor Trajectory Generation for Fast Autonomous Flight. IEEE Robotics and Automation Letters. DOI: 10.1109/LRA.2019.2927938.

25 Kulathunga G., Devitt D., Fedorenko R., Klimchik A. S., Path Planning Followed by Kinodynamic Smoothing for Multirotor Aerial Vehicles (MAVs), Rus. J. Nonlin. Dyn., 2021, Vol. 17, no. 4.

26 Y., Bestaoui. 3D flyable curves for an autonomous aircraft [Text] / Bestaoui Y. // 9th International Conference on Mathematical Problems in Engineering, Aerospace and Sciences. - 2012. - Vol. 1493.

27 Andréa Macario Barros, Maugan Michel, Yoann Moline, Gwenolé Corre, Frédérick Carrel. A Comprehensive survey of visual SLAM algorithms. Robotics, 2022, №11.

28 Yu, H.; Wang, Q.; Yan, C.; Feng, Y.; Sun, Y.; Li, L. DLD-SLAM: RGB-D Visual Simultaneous Localisation and Mapping in Indoor Dynamic Environments Based on Deep Learning. Remote Sens. 2024, № 16.

29 Vedadi, Amirhosein & Yousefi-Koma, Aghil & Yazdankhah, Parsa & Mozayyan, Amin. (2023). Comparative Evaluation of RGB-D SLAM Methods for Humanoid Robot Localization and Mapping. 10.1109/ICRoM60803.2023.10412425.

30 Shan, Tixiao and Englot, Brendan and Meyers, Drew and Wang, Wei and Ratti, Carlo and Rus Daniela. LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping // IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 5135-5142, 2020. (язык Англ.)

31 T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti and D. Rus, "LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping," 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Las Vegas, NV, USA, 2020, DOI: 10.1109/IROS45743.2020.9341176.

32 Zhang, Ji & Singh, Sanjiv. (2014). LOAM : Lidar Odometry and Mapping in real-time. Robotics: Science and Systems Conference (RSS).

33 Yang, Y.; Xiong, X.; Yan, Y. UAV Formation Trajectory Planning Algorithms: A Review. Drones 2023, 7, 62. https://doi.org/10.3390/drones7010062

34 Peter E. Hart Nils J. Nilsson, Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths / Bertram Raphael Peter E. Hart, Nils J. Nilsson // IEEE Transactions on Systems Science and Cybernetics. - 1968. - Vol. 4, no. 2.

35 Likhachev M., Stenz A. R* Search / Stenz A. Likhachev M. // Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008 / Ed. by Carla P. Gomes Dieter Fox. Washington, DC, U.S. : AAAI Press, 2008.

36 Daniel Harabor, Alban Grastien. Online graph pruning for pathfinding on grid maps [Text] / Alban Grastien Daniel Harabor // Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011 / Ed. by Dan Roth Wolfram Burgard. - Washington, DC, U.S. : AAAI Press, 2011.

37 Xia, J.; Zhou, Z. Model Predictive Control Based on ILQR for Tilt-Propulsion UAV. Aerospace 2022, 9, 688. https://doi.org/10.3390/aerospace9110688.

38 Saleh Z.H. и др. Enhanced Dynamic Control of Quadcopter PMSMs Using an ILQR-PCC System for Improved Stability and Reduced Torque Ripples // Journal of Robotics and Control (JRC). - 2024. - Vol. 5, No. 6. - DOI: 10.18196/jrc.v5i6.23159

39 Lee Dasol, Shim David. RRT-based path planning for fixed-wing UAVs with arrival time and approach direction constraints / Shim David Lee, Dasol //International Conference on Unmanned Aircraft Systems (ICUAS). - [S. l. : s. n.], 2014. - 05.

40 Y., Bestaoui. 3D flyable curves for an autonomous aircraft [Text] / Bestaoui Y. // 9th International Conference on Mathematical Problems in Engineering, Aerospace and Sciences. - 2012. - Vol. 1493.

41 ROS. Официальный сайт Robot Operating System. URL:https://www.ros.org/ (дата обращения: 28.03.2024).

42 PX4 Development Team. ROS 2 User Guide — PX4 Documentation. URL:

https://docs.px4.io/main/en/ros2/user_guide.html (дата обращения: 28.03.2024).

Приложение А. Динамическая модель квадрокоптера

```
# ======= CONSTANTS =======
SEA_LEVEL_PRESSURE = 101325.0
EKF_DT = 0.01
# ======= DRONE CONSTRUCT PARAMETERS =======
MASS = 0.82
INERTIA = np.diag([0.045, 0.045, 0.045])
ARM_LEN = 0.15
K_THRUST = 1.48e-6
K_TORQUE = 9.4e-8
MOTOR_TAU = 0.02
MAX_SPEED = 2100.0
DRAG = 0.1
MAX_RATE = 25.0    # рад/с, ограничение на угловую скорость
(roll/pitch)

def f(self, x, u, dt):
    # Константы модели
    m = MASS              # Масса дрона
    I = INERTIA           # Матрица инерции (3x3)
    arm = ARM_LEN         # Длина луча от центра до мотора
    kf = K_THRUST         # Коэффициент тяги (Н / (рад/с)^2)
    km = K_TORQUE         # Коэффициент момента (Нм / (рад/с)^2)
    drag = DRAG           # Линейное сопротивление воздуха
       g = jnp.array([0.0, 0.0, 9.81])   # Ускорение свободного
падения
       max_speed = MAX_SPEED   # Максимальная скорость вращения
моторов

    # Распаковка состояния
    pos = x[0:3]          # Позиция [x, y, z]
    vel = x[3:6]          # Линейная скорость [vx, vy, vz]
    # Нормализованный кватернион ориентации
    quat = x[6:10] / (jnp.linalg.norm(x[6:10]) + 1e-8)
    omega = x[10:13]      # Угловая скорость [wx, wy, wz]

    # Матрица поворота из тела в мировой фрейм
    R_bw = jnp.array(R.from_quat(quat).as_matrix())

       # Ограничение значений управления (скоростей моторов) по
максимуму
    rpm = jnp.clip(u, 0.0, max_speed)
    w_squared = rpm ** 2

    # Вычисление тяги от каждого мотора
    thrusts = kf * w_squared

    # Полная тяга в теле: вдоль оси z (вверх в системе тела)
    Fz_body = jnp.array([0.0, 0.0, jnp.sum(thrusts)])

       # Преобразование тяги в мировую систему координат и учет
гравитации и сопротивления
    F_world = R_bw @ Fz_body - m * g - drag * vel
```

```
    # Линейное ускорение
    acc = F_world / m

    # Интеграция положения и скорости (по формуле с ускорением)
    new_vel = vel + acc * dt
    new_pos = pos + vel * dt + 0.5 * acc * dt ** 2

    # Момент от моторов: [roll, pitch, yaw]
    tau = jnp.array([
        arm * (thrusts[1] - thrusts[3]),  # Момент по оси X (roll)
            arm * (thrusts[2] - thrusts[0]),   # Момент по оси Y
(pitch)
             km  *  (w_squared[0]  -  w_squared[1]  +  w_squared[2]  -
w_squared[3])  # Момент по Z (yaw)
    ])

    # Кориолисово слагаемое: ω × (I * ω)
    omega_cross = jnp.cross(omega, I @ omega)

    # Угловое ускорение: I⁻¹ (τ - ω × (Iω))
    omega_dot = jnp.linalg.solve(I, tau - omega_cross)
    new_omega = omega + omega_dot * dt

    # Дифференциал кватерниона через угловую скорость
      omega_quat  =  jnp.concatenate([jnp.array([0.0]),  omega])   #
Псевдокватернион
    dq = 0.5 * self.quat_multiply(quat, omega_quat)
    new_quat = quat + dq * dt
     new_quat /= jnp.linalg.norm(new_quat + 1e-8)  # Нормализация
кватерниона

    # Объединенное новое состояние
         x_next  =  jnp.concatenate([new_pos,  new_vel,  new_quat,
new_omega])
    return x_next
```

Приложение Б. iLQR-регулятор

```
class ILQROptimizer:
    def __init__(self, logger):
        self.logger = logger
        self.datetime = datetime.now().strftime("%Y%m%d_%H%M%S")
        self.log_base = os.path.join("MY_iLQR_LOG", self.datetime)
        self.last_log_time = time.time()  # инициализация
        self.last_reduced_log_time =  time.time()

    def log_ilqr_reduced(self, data:str):
        os.makedirs(self.log_base, exist_ok=True)
        now = time.time()
         elapsed_ms = (now - self.last_reduced_log_time) * 1000  #
в миллисекундах
        self.last_log_time = now
                        log_path   =   os.path.join(self.log_base,
"ILQR_solve()_reduced.txt")

        with open(log_path, "a") as f:
            f.write(f"[+{elapsed_ms:.1f}ms] {data}\n")

    def log_ilqr(self, data: str):
        os.makedirs(self.log_base, exist_ok=True)
        now = time.time()
            elapsed_ms = (now - self.last_log_time) * 1000   # в
миллисекундах
        self.last_log_time = now
        log_path = os.path.join(self.log_base, "ILQR_solve().txt")

        with open(log_path, "a") as f:
            f.write(f"[+{elapsed_ms:.1f}ms] {data}\n")

     def solve(self,  x0,  u_init,  x_target_traj,  u_target_traj,
Q, R, Qf,
               max_iters=5, tol=1e-3, alpha=1.0):
        X =  simulate_trajectory(x0, u_init)
              #self.log_ilqr(f"X=self.simulate_trajectory(self.x0,
self.u_init): X={X}")
        U =  u_init
           self.log_ilqr_reduced(f"start solve x_target_traj.shape
{x_target_traj.shape}")
        for i in range(max_iters):
                                            self.log_ilqr(f"=======
i={i}=======\ncost_function_traj_flip start")

                        cost_prev  =  cost_function_traj_flip(X,  U,
x_target_traj,  u_target_traj, Q, R, Qf)
                               self.log_ilqr("cost_function_traj_flip
end\nbackward_pass start")

              K_list, k_list = backward_pass(X, U,  x_target_traj,
u_target_traj, Q, R, Qf)
            self.log_ilqr("backward_pass end\nforward_pass start")
```

```
                X_new, U_new = forward_pass(X, U, k_list, K_list,
alpha)
                                            self.log_ilqr("forward_pass
end\ncost_function_traj_flip start")

                 cost_new = cost_function_traj_flip(X_new, U_new,
x_target_traj, u_target_traj, Q, R, Qf)
            self.log_ilqr("cost_function_traj_flip end")

            if jnp.abs(cost_prev - cost_new) < tol:
                break
            X, U = X_new, U_new
        self.log_ilqr(f"************end solve*************")
        self.log_ilqr_reduced(f"end solve X.shape={X.shape}")
        return X, U, i, cost_new


"""
f: функция динамики: f(x, u, dt) → x_next
fx_batch: функция для вычисления Якобианов A, B по заданному
состоянию и управлению
dt: шаг по времени
horizon: длина горизонта предсказания
Q, R, Qf: матрицы весов стоимости
n: размерность состояния
m: размерность управления
"""
@jit
def simulate_trajectory(x0, U):# распараллеленная
    X = [x0]
       # Используем vmap для параллельного вычисления следующего
состояния
     f_batch = vmap(f, in_axes=(None, 0, None))  # Здесь U — это
батч управляющих сигналов
    U = jnp.array(U)
    X_new = f_batch(x0, U, dt)
    X.extend(X_new)
    return jnp.stack(X)

@jit
def linearize_dynamics(x, u):
    A = jacobian(f, argnums=0)(x, u, dt)
    B = jacobian(f, argnums=1)(x, u, dt)
    return A, B

@jit
def quadratize_cost(x, u, x_target, u_target, Q, R):
    def scalar_cost(x_, u_):
        dx = x_ - x_target
        du = u_ - u_target
        return dx @ Q @ dx + du @ R @ du

    lx = grad(scalar_cost, argnums=0)(x, u)
    lu = grad(scalar_cost, argnums=1)(x, u)
    lxx = hessian(scalar_cost, argnums=0)(x, u)
```

```
    luu = hessian(scalar_cost, argnums=1)(x, u)

    lux = jacobian(grad(scalar_cost, argnums=1), argnums=0)(x, u)
    return lx, lu, lxx, luu, lux

def log_matrix(logger, name, matrix):
    """Универсальная функция логирования матрицы в ROS 2."""
    matrix_np = np.array(matrix)  # если это JAX, то .to_py() тоже
может подойти
    logger.info(f"{name} shape: {matrix_np.shape}")
    logger.info(f"{name} contents:\n{matrix_np}")

@jit
def backward_pass(X, U, x_target_traj, u_target_traj, Q, R, Qf):
    Vx = grad(lambda x: jnp.dot((x - x_target_traj[-1]), Qf @ (x -
x_target_traj[-1])))(X[-1])
    Vxx = Qf
    K_list = []
    k_list = []
    for k in reversed(range(horizon)):
        xk = X[k]
        uk = U[k]
        xt = x_target_traj[k]
        ut = u_target_traj[k]

        A, B = linearize_dynamics(xk, uk)

        # log_matrix(logger, "Matrix A", A)
        # log_matrix(logger, "Matrix B", B)
        # log_matrix(logger, "Matrix Vxx", Vxx)
        lx, lu, lxx, luu, lux = quadratize_cost(xk, uk, xt, ut, Q,
R)

        # log_matrix(logger, "Matrix lux", lux)

        Qx = lx + A.T @ Vx
        Qu = lu + B.T @ Vx
        Qxx = lxx + A.T @ Vxx @ A
        Quu = luu + B.T @ Vxx @ B
        Qux = lux + B.T @ Vxx @ A #Qux = lux.T + B.T @ Vxx @ A  #
обе части (m, n)

        Quu_reg = Quu + 1e-6 * jnp.eye(m)  # регуляризация
        Quu_inv = jnp.linalg.inv(Quu_reg)

        K = -Quu_inv @ Qux
        kff = -Quu_inv @ Qu

        Vx = Qx + K.T @ Quu @ kff + K.T @ Qu + Qux.T @ kff
        Vxx = Qxx + K.T @ Quu @ K + K.T @ Qux + Qux.T @ K

        K_list.insert(0, K)
        k_list.insert(0, kff)
    return K_list, k_list
```

```
@jit
def forward_pass(X, U, k_list, K_list, alpha):
    x = X[0]
    X_new = [x]
    U_new = []
    for k in range(horizon):
        dx = x - X[k]
        du = k_list[k] + K_list[k] @ dx
        u_new = U[k] + alpha * du
        U_new.append(u_new)
        x = f(x, u_new, dt)
        X_new.append(x)
    return jnp.stack(X_new), jnp.stack(U_new)

@jit
def    cost_function_traj_flip(x_traj,    u_traj,    x_target_traj,
u_target_traj, Q, R, Qf):
            def   compute_step_cost(x,   u,   x_target,   u_target,
is_terminal=False):
        # Вычисляем ошибку по позиции
        position_error = x[0:3] - x_target[0:3]
           position_cost  =  jnp.dot(position_error,  jnp.dot(Q[0:3,
0:3], position_error))  # Матрица для позиции

        # Вычисляем ошибку по ориентации (кватернионы)
              q_current  =  x[6:10]  /  jnp.linalg.norm(x[6:10])   #
Нормализация кватернионов
                                 q_target    =    x_target[6:10]    /
jnp.linalg.norm(x_target[6:10])
         dot_product = jnp.clip(jnp.dot(q_current, q_target), -1.0,
1.0)  # Ограничиваем скалярный продукт
         orientation_error = 2.0 * jnp.arccos(jnp.abs(dot_product))
# Ошибка по ориентации
         orientation_cost = orientation_error**2 * Q[6, 6]  # Штраф
по элементам ориентации

        # Вычисляем ошибку по управлению
        control_error = u - u_target
                control_cost  =  jnp.dot(control_error,  jnp.dot(R,
control_error))  # Диагональная матрица для управления

           # Если это последний шаг (терминальный), то добавляем
терминальный штраф
        if is_terminal:
            terminal_position_error = position_error
            terminal_orientation_error = orientation_error
                  terminal_cost  =  jnp.dot(terminal_position_error,
jnp.dot(Qf[0:3, 0:3], terminal_position_error)) + \
                               terminal_orientation_error**2 * Qf[6,
6]
             return position_cost + orientation_cost + control_cost
+ terminal_cost

        return position_cost + orientation_cost + control_cost
```

```
    # Суммируем стоимость по всем шагам траектории
    n_steps = x_traj.shape[0]
    step_costs = []

    for i in range(n_steps):
            is_terminal = (i == n_steps - 1)   # Последний шаг — терминальный
            step_cost = compute_step_cost(x_traj[i], u_traj[i], x_target_traj[i], u_target_traj[i], is_terminal)
        step_costs.append(step_cost)

    total_cost = jnp.sum(jnp.array(step_costs))
    return total_cost
```

Приложение В. Адаптивный планировщик целей

```
import csv
import time
from std_msgs.msg import String
from datetime import datetime
from quad_flip_msgs.msg import OptimizedTraj
from rclpy.qos import QoSProfile
import threading
from jax import jit, grad, jacobian, hessian, vmap, lax
import jax.numpy as jnp
from scipy.spatial.transform import Rotation as Rot
#CONSTANTS
SEA_LEVEL_PRESSURE = 101325.0
EKF_DT = 0.01
#DRONE CONSTRUCT PARAMETERS
MASS = 0.82
INERTIA = np.diag([0.045, 0.045, 0.045])
ARM_LEN = 0.15
K_THRUST = 1.48e-6
K_TORQUE = 9.4e-8
MOTOR_TAU = 0.02
MAX_SPEED = 2100.0
DRAG = 0.1
MAX_RATE = 25.0   # ограничение на угловую скорость (roll/pitch) рад/с

#Гиперпараметры
dt = 0.1
horizon = 10  # Горизонт предсказания
n = 13           # Размерность состояния квадрокоптера (позиция, скорость, ориентация, угловая скорость)
m = 4         # Размерность управления (4 мотора)

# ****************** Настройка стоимостей iLQR ******************
Q = jnp.diag(jnp.array([
    1.0, 1.0, 10.0,       # x, y — менее важны, z — важна
    1.0, 1.0, 1.0,        # vx, vy, vz
    0.0, 50.0, 50.0, 0.0, # ориентация
    5.0, 5.0, 1.0         # угловые скорости
]))
R = jnp.diag(jnp.array([
        0.001, 0.001, 0.001, 0.001  # все моторы слабо штрафуются
    ]))
Qf = jnp.diag(jnp.array([
     1.0, 1.0, 10.0,          # позиции: x, y — меньше важны, z — важна
    0.1, 0.1, 0.1,        # скорости
    0.0, 100.0, 100.0, 0.0, # ориентация (qx, qy)
    10.0, 10.0, 1.0       # угловые скорости
]))
# ===== MATRIX OPERTIONS =====
# QUATERNION UTILS (SCIPY-based)
def quat_to_rot_matrix_numpy(quat):
    # Кватернион: [w, x, y, z]
```

```
    w, x, y, z = quat
    R = np.array([
          [1 - 2*(y**2 + z**2),      2*(x*y - z*w),        2*(x*z +
y*w)],
          [2*(x*y + z*w),            1 - 2*(x**2 + z**2), 2*(y*z -
x*w)],
         [2*(x*z - y*w),           2*(y*z + x*w),       1 - 2*(x**2
+ y**2)]
    ])
    return R
def quat_multiply_numpy(q, r):
    # Кватернионы [w, x, y, z]
    w0, x0, y0, z0 = q
    w1, x1, y1, z1 = r
    return np.array([
        w0*w1 - x0*x1 - y0*y1 - z0*z1,
        w0*x1 + x0*w1 + y0*z1 - z0*y1,
        w0*y1 - x0*z1 + y0*w1 + z0*x1,
        w0*z1 + x0*y1 - y0*x1 + z0*w1
    ])
def f_numpy(x, u, dt):
    m = MASS
    I = INERTIA
    arm = ARM_LEN
    kf = K_THRUST
    km = K_TORQUE
    drag = DRAG
    g = np.array([0.0, 0.0, 9.81])
    max_speed = MAX_SPEED

    pos = x[0:3]
    vel = x[3:6]
    quat = x[6:10]
    omega = x[10:13]
    quat_norm = np.linalg.norm(quat)
    if quat_norm < 1e-8:
        quat = np.array([1.0, 0.0, 0.0, 0.0])
    else:
        quat = quat / quat_norm
    R_bw = quat_to_rot_matrix_numpy(quat)
    rpm = np.clip(u, 0.0, max_speed)
    w_squared = rpm ** 2
    thrusts = kf * w_squared
    Fz_body = np.array([0.0, 0.0, np.sum(thrusts)])
    F_world = R_bw @ Fz_body - m * g - drag * vel
    acc = F_world / m
    new_vel = vel + acc * dt
    new_pos = pos + vel * dt + 0.5 * acc * dt ** 2
    tau = np.array([
        arm * (thrusts[1] - thrusts[3]),
        arm * (thrusts[2] - thrusts[0]),
             km * (w_squared[0] - w_squared[1] + w_squared[2] -
w_squared[3])
    ])
```

```
    omega_cross = np.cross(omega, I @ omega)
    omega_dot = np.linalg.solve(I, tau - omega_cross)
    new_omega = omega + omega_dot * dt
    omega_quat = np.concatenate(([0.0], new_omega))
    dq = 0.5 * quat_multiply_numpy(quat, omega_quat)
    new_quat = quat + dq * dt
      new_quat /= np.linalg.norm(new_quat) + 1e-8   # безопасная
нормализация
        x_next = np.concatenate([new_pos, new_vel, new_quat,
new_omega])
    return x_next
@jit
def quat_multiply(q1, q2):
    """
    Умножение кватернионов q1 * q2
    q = [w, x, y, z]
    """
    w1, x1, y1, z1 = q1
    w2, x2, y2, z2 = q2
    w = w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2
    x = w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2
    y = w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2
    z = w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2
    return jnp.array([w, x, y, z])
@jit
def quat_to_rot_matrix(q):
    x, y, z, w = q
    xx, yy, zz = x * x, y * y, z * z
    xy, xz, yz = x * y, x * z, y * z
    wx, wy, wz = w * x, w * y, w * z
    return jnp.array([
        [1 - 2 * (yy + zz),     2 * (xy - wz),     2 * (xz + wy)],
        [    2 * (xy + wz), 1 - 2 * (xx + zz),     2 * (yz - wx)],
        [    2 * (xz - wy),     2 * (yz + wx), 1 - 2 * (xx + yy)]
    ])
@jit
def f(x, u, dt):
    m = MASS
    I = INERTIA
    arm = ARM_LEN
    kf = K_THRUST
    km = K_TORQUE
    drag = DRAG
    g = jnp.array([0.0, 0.0, 9.81])
    max_speed = MAX_SPEED
    pos = x[0:3]
    vel = x[3:6]
    quat = x[6:10]
    omega = x[10:13]
    # нормализация кватерниона через jax.lax.cond
    quat_norm = jnp.linalg.norm(quat)
    quat = lax.cond(
        quat_norm < 1e-8,
        lambda _: jnp.array([1.0, 0.0, 0.0, 0.0]),
        lambda _: quat / quat_norm,
```

```
        operand=None
    )
    R_bw = quat_to_rot_matrix(quat)
    rpm = jnp.clip(u, 0.0, max_speed)
    w_squared = rpm ** 2
    thrusts = kf * w_squared
    Fz_body = jnp.array([0.0, 0.0, jnp.sum(thrusts)])
    F_world = R_bw @ Fz_body - m * g - drag * vel
    acc = F_world / m
    new_vel = vel + acc * dt
    new_pos = pos + vel * dt + 0.5 * acc * dt ** 2
    tau = jnp.array([
        arm * (thrusts[1] - thrusts[3]),
        arm * (thrusts[2] - thrusts[0]),
             km * (w_squared[0] - w_squared[1] + w_squared[2] -
w_squared[3])
    ])
    omega_cross = jnp.cross(omega, I @ omega)
    omega_dot = jnp.linalg.solve(I, tau - omega_cross)
    new_omega = omega + omega_dot * dt
    omega_quat = jnp.concatenate([jnp.array([0.0]), new_omega])
    dq = 0.5 * quat_multiply(quat, omega_quat)
    new_quat = quat + dq * dt
      new_quat /= jnp.linalg.norm(new_quat + 1e-8)   # безопасная
нормализация
        x_next = jnp.concatenate([new_pos, new_vel, new_quat,
new_omega])
    return x_next
class MyEKF(ExtendedKalmanFilter):
    def __init__(self, dim_x, dim_z):
        super().__init__(dim_x, dim_z)
        self.dt = EKF_DT
        self.f = f
     def predict_x(self, u=np.zeros(4)):# predict new state with
dynamic physic model
        return f_numpy(x=self.x, u=u, dt=self.dt)# custom fx(x, u,
dt) function
class ModelPredictiveControlNode(Node):
    def __init__(self):
        super().__init__('dynamic_model_node')
        qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self.datetime = datetime.now().strftime("%Y%m%d_%H%M%S")
        self.log_base = os.path.join("MY_LOG", self.datetime)
                self.ilqr_log_base = os.path.join("MY_iLQR_LOG",
self.datetime)
        # PUBLISHERS
self.server_pub          =          self.create_publisher(String,
'/drone/server_msg', qos_profile)
        # == == == == == == == == == == == == == =SUBSCRIBERS= ==
= == == == == == == == == == == == == == == == == == == == == ==
```

```
self.create_subscription(SensorCombined,
'/fmu/out/sensor_combined',          self.sensor_combined_callback,
qos_profile)
self.create_subscription(VehicleAngularVelocity,
'/fmu/out/vehicle_angular_velocity',
self.angular_velocity_callback, qos_profile)
self.create_subscription(VehicleAttitude,
'/fmu/out/vehicle_attitude',        self.vehicle_attitude_callback,
qos_profile)

self.create_subscription(VehicleAngularAccelerationSetpoint,
                  '/fmu/out/vehicle_angular_acceleration_setpoint',
self.vehicle_angular_acceleration_setpoint_callback, qos_profile)

self.create_subscription(VehicleImu,'/fmu/out/vehicle_imu',self.ve
hicle_imu_callback, qos_profile)

# ****** RPM *******
self.create_subscription(EscStatus,          '/fmu/out/esc_status',
self.esc_status_callback, qos_profile)
self.create_subscription(VehicleLocalPosition,
'/fmu/out/vehicle_local_position',
self.vehicle_local_position_callback, qos_profile)
self.create_subscription(SensorBaro,        '/fmu/out/sensor_baro',
self.sensor_baro_callback, qos_profile)
self.create_subscription(VehicleMagnetometer,
'/fmu/out/vehicle_magnetometer',
self.vehicle_magnetometer_callback, qos_profile)

# DATA USED IN METHODS
self.angularVelocity = np.zeros(3, dtype=np.float32)
self.angular_acceleration = np.zeros(3, dtype=np.float32)
self.vehicleImu_velocity_w  =  np.zeros(3,  dtype=np.float32)  #  в
мировых координатах
self.sensorCombined_linear_acceleration       =       np.zeros(3,
dtype=np.float32)
self.position  =  np.zeros(3,  dtype=np.float32)  #  drone  position
estimates with IMU localization
self.motor_rpms = jnp.zeros(4)
self.vehicleAttitude_q   =   np.array([0.0,   0.0,   0.0,   1.0],
dtype=np.float32) # quaternion from topic
self.magnetometer_data = np.zeros(3, dtype=np.float32)
self.baro_pressure = 0.0
self.baro_altitude = 0.0
self.mag_yaw = 0.0
# FOR SITL TESTING
self.vehicleLocalPosition_position = np.zeros(3, dtype=np.float32)
# OTHER TOPIC DATA
# [TOPIC NAME]_[PARAM NAME] OR [TOPIC NAME] IF PARAM = TOPIC NAME
self.sensorCombined_angular_velocity         =         np.zeros(3,
dtype=np.float32)
self.angularVelocity_angular_acceleration      =      np.zeros(3,
dtype=np.float32)
self.baro_temperature = 0.0 # temperature in degrees Celsius
# Гиперпараметры для EKF
```

```
# * вектор состояния 13 штук: позиция, скорость, ориентация (4),
угловые скорости
#  * вектор измерений 14 штук: позиция, линейная скорость,
ориентация (4), барометрическая высота
self.ekf = MyEKF(dim_x=13, dim_z=13)
self.ekf.x = np.zeros(13)
self.measurnments = np.zeros(13)
self.z= np.zeros(13)
self.ekf.x[6] = 1.0  # qw = 1 (единичный кватернион)
# Ковариация состояния
#self.ekf.P *= 0.1
# Процессный шум
self.ekf.Q = np.diag([
0.001, 0.001, 0.001,         # x, y, z
0.01, 0.01, 0.01,            # vx, vy, vz
0.0001, 0.0001, 0.0001, 0.0001,  # qw, qx, qy, qz
0.00001, 0.00001, 0.00001        # wx, wy, wz
])
# Измерительный шум (z не используется из позиции, вместо него —
баро)
self.ekf.R = np.diag([
0.1, 0.1,                    # позиция x, y (м²)
0.0001, 0.0001, 0.0001,      # скорость vx, vy, vz
0.00001, 0.00001, 0.00001, 0.00001,  # qw, qx, qy, qz
0.00001, 0.00001, 0.00001,   # wx, wy, wz
0.5                          # баро (вместо позиции z)
])
#  Параметры ModelPredictiveController
self.optimizer = ILQROptimizer(logger=self.get_logger())
self.phase = 'init'
self.takeoff_altitude = -5.0  # м TODO DOES IT WOULD BE NEGATIVE?
self.takeoff_tol = 0.1
self.flip_started_time = None
self.flip_duration = 1.0  # с, продолжительность флипа
self.recovery_time = 2.0  # с, стабилизация после флипа
self.recovery_start_time = None
self.landing_altitude = 0.2  # м
self.roll_abs_tol = 0.1  # допуск 0.1 рад

now_str = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
self.log_file_name_pos = f"{now_str}_pos.log"
self.log_file_name_quat = f"{now_str}_quat.log"
self.log_file_name_vel = f"{now_str}_vel.log"
self.log_file_name_ang_vel = f"{now_str}_ang_vel.log"

# ======= TIMERS =======
self.EKF_timer = self.create_timer(EKF_DT, self.EKF)
self.mpc_controller           =           self.create_timer(0.05,
self.mpc_control_loop)
# == == == =CLIENT SERVER INTERACTION= == == ==
self.create_subscription(String,              '/drone/client_msg',
self.client_msg_callback, qos_profile)#
self.pub_optimized_traj    =    self.create_publisher(OptimizedTraj,
'/drone/optimized_traj', qos_profile)
```

```
        self.optimized_traj_f = False
        self.X_opt = np.zeros((horizon + 1, n))  # (N+1) x n
        self.u_optimal = np.zeros((horizon, m))  # N x m
        self.i_final = 0
        self.cost_final = 0.0
        self.done = False
        self.to_client_f = False
        self.mpc_lock = threading.Lock()
        #self.x0 = self.ekf.x.copy()
           #self.u_init = jnp.tile(self.actuator_motors, (horizon,
1))
        self.x_target_traj = jnp.zeros((horizon + 1, 13))
          self.u_target_traj = jnp.tile(self.motor_rpms, (horizon,
1))
          self.current_time = self.get_clock().now().nanoseconds *
1e-9
    def esc_status_callback(self, msg: EscStatus):
        rpms = [esc.esc_rpm for esc in msg.esc[:msg.esc_count]]
        self.motor_rpms = np.clip(np.array(rpms), 0.0, MAX_SPEED)
                          #self.get_logger().info(f"self.motor_rpms:
{self.motor_rpms}")

    def send_msg_to_client(self, msg):
        server_msg = String()
        server_msg.data = msg
        self.server_pub.publish(server_msg)

    def client_msg_callback(self, msg):
        """GET CLIENT MESSAGES"""
        command = msg.data.strip().lower()
        #self.get_logger().info(f"Received command: {command}")
        if command == "takeoff":
            self.phase = "takeoff"
            self.to_client_f = True
            self.optimized_traj_f = True
            self.send_msg_to_client("mpc_on")
        else:
            self.get_logger().warn(f"Unknown command: {command}")

    def send_optimized_traj(self):
        if self.optimized_traj_f:
            msg = OptimizedTraj()
                                                    msg.x_opt    =
np.asarray(self.X_opt).flatten().astype(np.float32).tolist()
                                                    msg.u_opt    =
np.asarray(self.u_optimal).flatten().astype(np.float32).tolist()
            msg.i_final = int(self.i_final)
            msg.cost_final = float(self.cost_final)
            msg.done = self.done
            self.pub_optimized_traj.publish(msg)
            # Логирование в CSV
            self.log_optimized_traj()
    def log_optimized_traj(self):
        log_base = self.log_base
```

```
                            file_path    =    os.path.join(log_base,
'optimized_traj_log.csv')
                        text_log_path    =    os.path.join(log_base,
'optimized_traj_log.txt')
        os.makedirs(os.path.dirname(file_path), exist_ok=True)

        X_flat = np.asarray(self.X_opt[0]).flatten()
        u_flat = np.asarray(self.u_optimal).flatten()
        i_final = [self.i_final]
        cost_final = [self.cost_final]

        data = [X_flat, u_flat, i_final, cost_final]
        labels = ['X_opt', 'u_opt', 'i_final', 'cost_final']

        # --- Логирование CSV ---
        new_file = not os.path.exists(file_path)
        if new_file:
            headers = []
            for label, arr in zip(labels, data):
                if len(arr) > 1:
                            headers.extend([f"{label}[{i}]" for i in
range(len(arr))])
                else:
                    headers.append(label)

            with open(file_path, mode='w', newline='') as f:
                writer = csv.writer(f)
                writer.writerow(headers)

        row_values = [float(v) for arr in data for v in arr]
        with open(file_path, mode='a', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(row_values)

        # --- Логирование в текстовый файл ---
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        with open(text_log_path, 'a') as f:
            f.write(f"=== Log time: {timestamp} ===\n")
            f.write(f"Final iteration: {self.i_final}\n")
            f.write(f"Final cost: {self.cost_final}\n")
            f.write(f"X_opt (first 10 values): {X_flat[:10]}\n")
            f.write(f"u_opt (first 10 values): {u_flat[:10]}\n\n")
    def quaternion_from_roll(self, roll_rad):
        r = R.from_euler('x', roll_rad)
        return r.as_quat()
    def roll_from_quaternion(self, q):
        """ Вычисление угла roll из кватерниона """
        qw, qx, qy, qz = q
        sinr_cosp = 2 * (qw * qx + qy * qz)
        cosr_cosp = 1 - 2 * (qx**2 + qy**2)
        return jnp.arctan2(sinr_cosp, cosr_cosp).item()
    def log_ilqr(self, data: str):
        os.makedirs(self.ilqr_log_base, exist_ok=True)
                    log_path   =   os.path.join(self.ilqr_log_base,
"ILQR_run_mpc().txt")
```

```
        with open(log_path, "a") as f:
            f.write(data + "\n")
    def log_mpc(self, data: str):
        os.makedirs(self.ilqr_log_base, exist_ok=True)
                    log_path   =   os.path.join(self.ilqr_log_base,
"MPC_target.txt")
        with open(log_path, "a") as f:
            f.write(data + "\n")
    def takeoff_targets(self):
                    for   i   in   range(horizon):            pos   =
jnp.array(self.measurnments[0:3]).copy().at[2].set(self.takeoff_al
titude)
            vel = jnp.array(self.measurnments[3:6])
            q = jnp.array(self.measurnments[6:10])
            omega = jnp.array(self.measurnments[10:13])
                                                self.x_target_traj    =
self.x_target_traj.at[i].set(jnp.concatenate([pos,     vel,     q,
omega]))
                      self.log_ilqr(f"======  self.motor_rpms.shape
{self.motor_rpms.shape} ======")
                                                self.u_target_traj    =
self.u_target_traj.at[i].set(self.motor_rpms.copy())
                                              self.x_target_traj     =
self.x_target_traj.at[horizon].set(self.x_target_traj[horizon     -
1])
    def flip_targets(self ):
                       t_local   =   jnp.clip(self.current_time   -
self.flip_started_time, 0.0, self.flip_duration)
        roll_expected = 2 * jnp.pi * t_local / self.flip_duration

        # Получаем кватернион текущей ориентации
        q_current = self.measurnments[6:10]

        # Оценка roll из кватерниона
        self.roll_current = self.roll_from_quaternion(q_current)

        roll_error = roll_expected - self.roll_current
        gain_base = 0.8
        gain_adaptive = gain_base + 0.3 * jnp.tanh(roll_error)
              roll_target  =  self.roll_current  +  gain_adaptive  *
roll_error
        for i in range(horizon):
            alpha_i = i / horizon
            angle_i = roll_target * alpha_i
            pos = self.x0[0:3]
            vel = jnp.zeros(3)
            # Генерируем кватернион для целевой ориентации
            q = self.quaternion_from_roll(angle_i)

              omega_magnitude = 2 * jnp.pi / self.flip_duration +
0.2 * roll_error
            omega = jnp.array([omega_magnitude, 0.0, 0.0])
                                                self.x_target_traj    =
self.x_target_traj.at[i].set(jnp.concatenate([pos,     vel,     q,
omega]))
```

```
                                                self.u_target_traj    =
self.u_target_traj.at[i].set(self.recovery_thrust.copy())
        # Оставляем последний элемент траектории неизменным
self.x_target_traj                                                    =
self.x_target_traj.at[horizon].set(self.x_target_traj[horizon     -
1])
    def recovery_targets(self):
 t_local = jnp.clip(self.current_time - self.recovery_start_time,
0.0, self.recovery_time)
                roll_desired = 2 * jnp.pi * (1 - t_local /
self.recovery_time)
        # Получаем кватернион текущей ориентации
        q_current = self.measurnments[6:10]
        # Оценка roll из кватерниона
        self.roll_current = self.roll_from_quaternion(q_current)

        roll_error = roll_desired - self.roll_current
        gain = 0.6 + 0.4 * (abs(roll_error) / jnp.pi)
        roll_target = self.roll_current + gain * roll_error

        for i in range(horizon):
            alpha_i = i / horizon
             angle_i = self.roll_current + alpha_i * (roll_target -
self.roll_current)

            pos = self.x0[0:3]
            vel = jnp.zeros(3)

            # Генерируем кватернион для целевой ориентации
            q = self.quaternion_from_roll(angle_i)

               omega_mag = -2 * jnp.pi / self.recovery_time * (1 +
0.2 * abs(roll_error) / jnp.pi)
            omega = jnp.array([omega_mag, 0.0, 0.0])

                                                self.x_target_traj    =
self.x_target_traj.at[i].set(jnp.concatenate([pos,     vel,     q,
omega]))
                                                self.u_target_traj    =
self.u_target_traj.at[i].set(self.recovery_thrust.copy())

        # Оставляем последний элемент траектории неизменным
                                              self.x_target_traj      =
self.x_target_traj.at[horizon].set(self.x_target_traj[horizon     -
1])

    def land_targets(self ):
        return

    def run_mpc_thread(self):
        with self.mpc_lock:
            start_time = time.time()
  self.log_ilqr(f"============= phase: {self.phase}=============")
            try:
         self.current_time = self.get_clock().now().nanoseconds *
```

```
1e-9
         if self.phase == 'takeoff':
               self.send_msg_to_client("mpc_on")# на всякий случай
если сообщене не дойдет с одного раза,
      # чтобы переключить контроллер полета на прием управления
траектории
                self.takeoff_targets()
                self.log_ilqr(f"takeoff")
               if abs( - self.takeoff_altitude) < self.takeoff_tol:
                    self.phase = 'flip'
                    self.flip_started_time = self.current_time
                elif self.phase == 'flip':
                     self.flip_targets()
self.log_ilqr(f"flip\nabs(roll_current)={abs(self.roll_current)}")
                      if jnp.isclose(self.roll_current, 2 * jnp.pi,
atol=0.1):# выражение устойчивее к шуму чем аналогичное с abs
               self.phase = 'recovery'
               self.recovery_start_time = self.current_time
                elif self.phase == 'recovery':
                     self.recovery_targets()
        if abs(self.roll_current) <= self.roll_abs_tol:
                          self.phase = 'land'

                elif self.phase == 'land':
                          self.to_client_f = False
                          self.optimized_traj_f = False
                          self.done = True
                          self.send_msg_to_client("land")
                """
                      Вычисляет оптимальную траекторию состояний и
управляющих воздействий
                от текущего состояния self.x0, используя iLQR.
                """
                self.log_mpc(f"x0:{self.measurnments}")
                self.log_mpc(f"self.motor_rpms:{self.motor_rpms}")

self.log_mpc(f"self.x_target_traj:{self.x_target_traj}")

self.log_mpc(f"self.u_target_traj:{self.u_target_traj}")
                          # Используем ILQR для расчета оптимальной
траектории
              measurnments_init = self.measurnments  # (13,)
             motor_rpms_init = jnp.tile(self.motor_rpms, (horizon,
1))  # (horizon, 4)
                            X_opt,  U_opt,  i_final,  cost_final  =
self.optimizer.solve(
                    x0=measurnments_init,
                    u_init=motor_rpms_init,
                    Q=Q,
                    R=R,
                    Qf=Qf,
                    x_target_traj=self.x_target_traj,
                    u_target_traj=self.u_target_traj
                )
```

```
                        self.X_opt = np.array(X_opt)                # Преобразование из jnp в np
                self.u_optimal = np.array(U_opt[0])
                self.i_final = i_final
                        self.cost_final = float(cost_final)     # Обеспечиваем float, а не jnp.scalar

                self.send_optimized_traj()
            except Exception as e:
                        self.log_ilqr(f"Ошибка при выполнении MPC: {str(e)}")
                        # Выводим traceback ошибки для детальной диагностики
                import traceback
                self.log_ilqr(f"{traceback.format_exc()}")
            finally:
                end_time = time.time()
                elapsed = end_time - start_time
                     self.log_ilqr(f"[mpc_control_loop] END phase: {self.phase}, duration: {elapsed:.3f} s")
                self.mpc_running = False
    def mpc_control_loop(self):
        # Запуск в отдельном потоке
        if self.optimized_traj_f:
            threading.Thread(target=self.run_mpc_thread).start()
    def ekf_logger(self):
        pos_my_ekf = self.measurnments[0:3]
        pos_real = self.vehicleLocalPosition_position
        quat_my_ekf = self.measurnments[6:10]
        px4_quat = self.vehicleAttitude_q
        vel_my_ekf = self.measurnments[3:6]
        integral_vel = self.vehicleImu_velocity_w
        omega_my_ekf = self.measurnments[10:13]
        omega_from_sensor = self.angularVelocity
        log_base = self.log_base
        # CSV файлы остаются прежними
        self._write_to_csv(
            os.path.join(log_base, 'pos_log.csv'),
            ['pos_my_ekf', 'pos_real'],
            [pos_my_ekf, pos_real],
            error_pairs=[(0,1)]
        )
        self._write_to_csv(
            os.path.join(log_base, 'quat_log.csv'),
            ['quat_my_ekf', 'px4_quat'],
            [quat_my_ekf, px4_quat],
            error_pairs=[(0, 1)]
        )
        self._write_to_csv(
            os.path.join(log_base, 'vel_log.csv'),
            ['vel_my_ekf', 'integral_vel'],
            [vel_my_ekf, integral_vel],
            error_pairs=[(0, 1)]
        )
        self._write_to_csv(
```

```
            os.path.join(log_base, 'ang_vel_log.csv'),
            ['omega_my_ekf', 'omega_from_sensor'],
            [omega_my_ekf, omega_from_sensor],
            error_pairs=[(0, 1)]
        )
        # TXT-файл только с EKF-данными
        log_txt_path = os.path.join(log_base, 'log_my_ekf.txt')
        with open(log_txt_path, 'a') as f:
            f.write('--- EKF Data ---\n')
            f.write(f'pos_my_ekf: {pos_my_ekf}\n')
            f.write(f'pos_real: {pos_real}\n')
            f.write(f'pos error: {abs(pos_real-pos_my_ekf)}\n')
            f.write(f'quat_my_ekf: {quat_my_ekf}\n')
            f.write(f'px4_quat: {px4_quat}\n')
            f.write(f'vel_my_ekf: {vel_my_ekf}\n')
            f.write(f'omega_my_ekf: {omega_my_ekf}\n')
            f.write(f'omega_from_sensor: {omega_from_sensor}\n')
            f.write('\n')
           def _write_to_txt(self, file_path, labels, data,
error_pairs=None):
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        if error_pairs is None:
            error_pairs = []
        with open(file_path, mode='a') as f:
            # Записываем данные
            for label, values in zip(labels, data):
                 formatted_values = ', '.join(f'{float(v):.6f}' for
v in values)
                f.write(f"{label}: [{formatted_values}]\n")
            # Записываем ошибки
            for i, j in error_pairs:
                diff = np.array(data[i]) - np.array(data[j])
                 formatted_diff = ', '.join(f'{float(v):.6f}' for v
in diff)
                                f.write(f"{labels[i]} - {labels[j]}:
[{formatted_diff}]\n")
            f.write("\n")  # разделитель между записями
           def _write_to_csv(self, file_path, labels, data,
error_pairs=None):
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        if error_pairs is None:
            error_pairs = []
        new_file = not os.path.exists(file_path)
        if new_file:
            headers = []
            for i, (label, arr) in enumerate(zip(labels, data)):
                if len(arr) > 1:
                            headers.extend([f"{label}[{j}]" for j in
range(len(arr))])
                else:
                    headers.append(label)
            for i, j in error_pairs:
                label_i, label_j = labels[i], labels[j]
                arr_len = len(data[i])
                 headers.extend([f"{label_i}-{label_j}[{k}]" for k
```

```
in range(arr_len)])
            with open(file_path, mode='w', newline='') as f:
                writer = csv.writer(f)
                writer.writerow(headers)
        row_values = [float(v) for arr in data for v in arr]
        for i, j in error_pairs:
            diff = np.array(data[i]) - np.array(data[j])
            row_values.extend([float(v) for v in diff])
        with open(file_path, mode='a', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(row_values)
    def sensor_baro_callback(self, msg):
        #self.get_logger().info("sensor_baro_callback")
        self.baro_temperature = msg.temperature
        self.baro_pressure = msg.pressure
           self.baro_altitude = 44330.0 * (1.0 - (msg.pressure /
SEA_LEVEL_PRESSURE) ** 0.1903)
        self.measurnments[2] = -self.baro_altitude
                    #self.get_logger().info(f"self.baro_altitude  =
{self.baro_altitude}")
    def get_yaw_from_mag(self):
        r = Rot.from_quat(self.vehicleAttitude_q)
        mag_world = r.apply(self.magnetometer_data)
        mag_x = mag_world[0]
        mag_y = mag_world[1]
        yaw_from_magnetometer = np.arctan2(-mag_y, mag_x)
        return yaw_from_magnetometer
        def        vehicle_magnetometer_callback(self,        msg:
VehicleMagnetometer):
        #self.get_logger().info("vehicle_magnetometer_callback")
            self.magnetometer_data = np.array(msg.magnetometer_ga,
dtype=np.float32)
        self.mag_yaw = self.get_yaw_from_mag()
    # ИСТИННАЯ ПОЗИЦИЯ ДЛЯ ОЦЕНКИ ИНЕРЦИАЛНОЙ ЛОКАЛИЗАЦИИ
               def   vehicle_local_position_callback(self,   msg:
VehicleLocalPosition):
          #self.get_logger().info(f"vehicle_local_position_callback
{msg.x} {msg.y} {msg.z}")
        self.vehicleLocalPosition_position[0] = msg.x
        self.vehicleLocalPosition_position[1] = msg.y
        self.vehicleLocalPosition_position[2] = msg.z
    # ЛИНЕЙНОЕ УСКОРЕНИЕ, УГЛОВОЕ УСКОРЕНИЕ, КВАТЕРНИОН
    def sensor_combined_callback(self, msg: SensorCombined):
          dt_gyro = msg.gyro_integral_dt * 1e-6  # микросекунды ->
секунды
            gyro_rad = np.array(msg.gyro_rad, dtype=np.float32)  #
угловая скорость (рад/с)
        self.sensorCombined_angular_velocity = gyro_rad
           delta_angle = gyro_rad * dt_gyro # Угловое приращение
(рад)
        self.sensorCombined_delta_angle = delta_angle
                        self.sensorCombined_linear_acceleration  =
np.array(msg.accelerometer_m_s2, dtype=np.float32)

                    def    angular_velocity_callback(self,    msg:
```

```
VehicleAngularVelocity):
        self.angularVelocity = np.array(msg.xyz, dtype=np.float32)
                        self.angularVelocity_angular_acceleration   =
np.array(msg.xyz_derivative, dtype=np.float32)
        #self.new_x[10:13] = self.angularVelocity
        # хорошая
                       #self.get_logger().info(f"self.angularVelocity
{self.angularVelocity[0]}                {self.angularVelocity[1]}
{self.angularVelocity[2]}")

    def vehicle_attitude_callback(self, msg: VehicleAttitude):
        # In this system we use scipy format for quaternion.
         # PX4 topic uses the Hamilton convention, and the order is
q(w, x, y, z). So we reorder it
            self.vehicleAttitude_q = np.array([msg.q[1], msg.q[2],
msg.q[3], msg.q[0]], dtype=np.float32)

     def vehicle_angular_acceleration_setpoint_callback(self, msg:
VehicleAngularAccelerationSetpoint):
        self.angular_acceleration = msg.xyz

    def vehicle_imu_callback(self, msg: VehicleImu):
                    delta_velocity  =  np.array(msg.delta_velocity,
dtype=np.float32)  # м/с
        delta_velocity_dt = msg.delta_velocity_dt * 1e-6  # с
             # Проверяем наличие ориентации и валидного времени
интеграции
        if delta_velocity_dt > 0.0:
            rotation = Rot.from_quat(self.vehicleAttitude_q)
            delta_velocity_world = rotation.apply(delta_velocity)
                        gravity  =  np.array([0.0,  0.0,  -9.80665],
dtype=np.float32)
            delta_velocity_world += gravity * delta_velocity_dt
            self.vehicleImu_velocity_w += delta_velocity_world
                    self.position  +=  self.vehicleImu_velocity_w  *
delta_velocity_dt
    def publish_motor_inputs(self):
        msg = Float32MultiArray()
        msg.data = self.motor_inputs.tolist()
        self.motor_pub.publish(msg)

    def log_ekf_measurements_txt(self):
                         log_file   =   os.path.join(self.log_base,
'ekf_measurements_log.txt')
        os.makedirs(os.path.dirname(log_file), exist_ok=True)

        labels = [
            'x', 'y', 'z',
            'vx', 'vy', 'vz',
            'qw', 'qx', 'qy', 'qz',
            'wx', 'wy', 'wz'
        ]
                    timestamp   =   datetime.now().strftime('%Y-%m-%d
%H:%M:%S.%f')[:-3]
        with open(log_file, mode='a') as f:
```

```
            f.write(f"[{timestamp}] ")
            for label, val in zip(labels, self.z):
                f.write(f"{label}={val:.6f} ")
            f.write("\n")
    def EKF(self):
        """ Основная функция обновления фильтра Калмана. """
        self.z = np.array([
            self.position[0],    # x
            self.position[1],    # y
             -self.baro_altitude, # высота по барометру TODO  BARO
IS NEG
            self.vehicleImu_velocity_w[0],      # vx
            self.vehicleImu_velocity_w[1],      # vy
            self.vehicleImu_velocity_w[2],      # vz
            self.vehicleAttitude_q[0],  # qw
            self.vehicleAttitude_q[1],  # qx
            self.vehicleAttitude_q[2],  # qy
            self.vehicleAttitude_q[3],  # qz
            self.angularVelocity[0],   # wx
            self.angularVelocity[1],   # wy
            self.angularVelocity[2]    # wz
        ])
        self.measurnments = self.z.copy()
        #self.ekf.x = self.ekf.predict_x(self.motor_rpms)
        #self.ekf.update(z, HJacobian=self.HJacobian, Hx=self.hx)
        # LOG RESULTS
        self.ekf_logger()
        self.log_ekf_measurements_txt()

        # Проверка выхода динамической модели
           x_next = f_numpy(x=self.z, u=np.array(self.motor_rpms,
dtype=np.float32), dt=dt)
      def hx(self, x):
         """ Модель измерений: что бы показали датчики при текущем
состоянии. """
        return np.array([
            x[0],  # x
            x[1],  # y
            x[2], # baro z
            x[3],  # vx
            x[4],  # vy
            x[5],  # vz
            x[6],  # qw
            x[7],  # qx
            x[8],  # qy
            x[9],  # qz
            x[10], # wx
            x[11], # wy
            x[12] # wz
        ])
    def HJacobian(self, x):
        """ Якобиан модели измерений. """
        H = np.zeros((13, 13))  # 13 измерений на 13 состояний
        H[0, 0] = 1.0  # x
        H[1, 1] = 1.0  # y
```

```
H[2, 3] = 1.0  # vx
H[3, 4] = 1.0  # vy
H[4, 5] = 1.0  # vz
H[5, 6] = 1.0  # qw
H[6, 7] = 1.0  # qx
H[7, 8] = 1.0  # qy
H[8, 9] = 1.0  # qz
H[9, 10] = 1.0  # wx
H[10, 11] = 1.0  # wy
H[11, 12] = 1.0  # wz
H[12, 2] = 1.0   # z (барометр)
return H
```

Приложение Г. Модуль генерации управляющих сигналов

```
import time
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Vector3, Twist, PoseStamped
from scipy.spatial.transform import Rotation as R
from std_msgs.msg import Float32
from rclpy.qos import QoSProfile, ReliabilityPolicy,
HistoryPolicy, DurabilityPolicy
from px4_msgs.msg import (  OffboardControlMode,
TrajectorySetpoint,
    VehicleStatus,
    VehicleRatesSetpoint, VehicleTorqueSetpoint,
VehicleAttitudeSetpoint, VehicleCommand, ActuatorMotors
)
import numpy as np
from enum import Enum
from std_msgs.msg import Float32MultiArray
from std_msgs.msg import String
from rclpy.qos import QoSProfile
from quad_flip_msgs.msg import OptimizedTraj

from pymavlink import mavutil
import threading


from px4_msgs.msg import  EscStatus

BOUNCE_TIME = 0.6
ACCELERATE_TIME = 0.07
BRAKE_TIME = ACCELERATE_TIME
ARM_TIMEOUT = 5.0
OFFBOARD_TIMEOUT = 5.0
OFFBOARD_MODE = 14  # код режима OFFBOARD в PX4
class DroneState(Enum):
    INIT = 7
    DISARMED = 0
    ARMING = 1
    ARMED = 2
    OFFBOARD = 3
    TAKEOFF = 4
    FLIP = 6
    LANDING = 16
    MPC_MANAGEMENT = 15

# dynamic drone control params
# TODO Move it to FILE
horizon = 50 # Горизонт предсказания
n = 13  # Размерность состояния квадрокоптера (позиция, скорость,
ориентация, угловая скорость)
m = 4  # Размерность управления (4 мотора)
```

```
# миксер rpm --> [roll_cmd, pitch_cmd, yaw_cmd, thrust_cmd]
def rpm_to_control(rpm, arm, kf, km):
    w_squared = rpm ** 2
    thrusts = kf * w_squared

    # Общая тяга
    thrust = jnp.sum(thrusts)

    # Моменты по осям
    roll_torque  = arm * (thrusts[1] - thrusts[3])   # M2 - M4
    pitch_torque = arm * (thrusts[2] - thrusts[0])   # M3 - M1
    yaw_torque   = km * (w_squared[0] - w_squared[1] +
w_squared[2] - w_squared[3])

    # Вернуть управляющие воздействия
    return jnp.array([roll_torque, pitch_torque, yaw_torque,
thrust])

def rpm_to_control_normalized(rpm, arm, kf, km, max_thrust,
max_torque):
    control = rpm_to_control(rpm, arm, kf, km)
    roll_cmd   = control[0] / max_torque
    pitch_cmd  = control[1] / max_torque
    yaw_cmd    = control[2] / max_torque
    thrust_cmd = control[3] / max_thrust
    return jnp.array([roll_cmd, pitch_cmd, yaw_cmd, thrust_cmd])

class PIDController:
    def __init__(self, Kp: float, Ki: float, Kd: float) -> None:
        """Инициализация PID-контроллера с заданными
коэффициентами."""
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd

        self.prev_error = 0.0
        self.integral = 0.0

    def compute(self, setpoint: float, measurement: float) ->
float:
        """
        Вычисляет управляющее воздействие на основе ошибки между
заданным значением и измерением.

        :param setpoint: Желаемое значение (например, RPM)
        :param measurement: Текущее измеренное значение (например,
RPM)
        :return: Управляющее воздействие
        """
        error = setpoint - measurement
        self.integral += error
        derivative = error - self.prev_error

        output = self.Kp * error + self.Ki * self.integral +
self.Kd * derivative
```

```
        self.prev_error = error

        return output

    def get_rotate_pwm(self, target_rpm, current_rpm):
        """
        Возвращает значение PWM на основе текущих и целевых RPM с
использованием PID-регулятора.

        :param target_rpm: Целевое значение оборотов
        :param current_rpm: Текущее значение оборотов
        :return: Расчетное PWM значение в диапазоне [1000, 2000]
        """
        pid_output = self.compute(target_rpm, current_rpm)
        pwm = 1500.0 + pid_output
        return float(np.clip(pwm, 1000.0, 2000.0))

class FlipControlNode(Node):
    def __init__(self):
        super().__init__('flip_control_node')

        #qos_profile = QoSProfile(depth=10)
        qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self.vehicle_command_publisher =
self.create_publisher(VehicleCommand, '/fmu/in/vehicle_command',
qos_profile)
        self.offboard_control_mode_publisher =
self.create_publisher(OffboardControlMode,
'/fmu/in/offboard_control_mode', qos_profile)
        self.trajectory_setpoint_publisher =
self.create_publisher(TrajectorySetpoint,
'/fmu/in/trajectory_setpoint', qos_profile)
        self.vehicle_rates_publisher =
self.create_publisher(VehicleRatesSetpoint,
'/fmu/in/vehicle_rates_setpoint', qos_profile)
        self.vehicle_torque_publisher =
self.create_publisher(VehicleTorqueSetpoint,
'/fmu/in/vehicle_torque_setpoint', qos_profile)
        self.publisher_rates =
self.create_publisher(VehicleRatesSetpoint,
'/fmu/in/vehicle_rates_setpoint', qos_profile)
        self.publisher_att =
self.create_publisher(VehicleAttitudeSetpoint,
'/fmu/in/vehicle_attitude_setpoint', qos_profile)
        self.publisher_actuator_motors =
self.create_publisher(ActuatorMotors, '/fmu/in/actuator_motors',
qos_profile)
        '/fmu/in/actuator_controls0'

        self.create_subscription(VehicleStatus,
```

```
'/fmu/out/vehicle_status', self.vehicle_status_callback,
qos_profile)

        # == == == =STATE CONTROL= == == ==
        self.main_state = DroneState.INIT

        # == == == =PX4 STATES= == == ==
        self.arming_state = 0
        self.nav_state = 0
        self.vehicle_status = VehicleStatus()
        self.stage_time = time.time()
        self.offboard_is_active = False
        self.offboard_state = False

        self.create_timer(0.1, self.update)
        self.create_timer(0.1, self.offboard_heartbeat)
        self.create_timer(0.1, self.drone_managenment)
        #self.create_timer(0.01, self.flip_thrust_max)
        #self.create_timer(0.001, self.flip_thrust_recovery)
        #self.create_timer(0.0001, self.flip_pitch_t)
        # == == == =Timer flags == == == =
        self.flip_thrust_max_f = False
        self.flip_thrust_recovery_f = False
        self.flip_pitch_f = False

        # ****** RPM *******
        self.create_subscription(EscStatus, '/fmu/out/esc_status',
self.esc_status_callback, qos_profile)
        # MPC INTEGRATION API
        self.pub_to_mpc = self.create_publisher(String,
'/drone/client_msg', qos_profile)#
        self.create_subscription(OptimizedTraj,
'/drone/optimized_traj', self.optimized_traj_callback,
qos_profile)
        self.create_subscription(String, '/drone/server_msg',
self.server_msg_callback, qos_profile)

        self.received_x_opt = np.zeros((horizon + 1, n))  # (N+1)
x n
        self.received_u_opt = np.zeros((horizon, m))  # N x m
        self.received_i_final = 0
        self.received_cost_final = 0.0
        self.received_done =  False
        self.target_u, self.target_x = [], []
        self.takeoff_alt = 0.0
        self.mpc_takeoff = False
        self.drone_managenment_f = False
        self.rpm_to_pwm_pid = PIDController(Kp=1.0, Ki=0.1,
Kd=0.01)
        self.target_pwm = np.zeros(4, dtype=int)
        self.target_rpm = np.zeros(4, dtype=int)
        self.motor_rpms = np.zeros(4, dtype=int)
        self.max_rpm = 15000
        self.master =
mavutil.mavlink_connection('udp:127.0.0.1:14550')
```

```
        # Отправим heartbeat, чтобы PX4 начал диалог
        self.master.mav.heartbeat_send(
            mavutil.mavlink.MAV_TYPE_GCS,
            mavutil.mavlink.MAV_AUTOPILOT_INVALID,
            0, 0, 0
        )

        # Ждем heartbeat от PX4
        self.get_logger().info("Жду heartbeat от PX4...")
        self.master.wait_heartbeat()
        self.get_logger().info("MAVLink: heartbeat received")

        #self.create_timer(0.1, self.send_pwm_loop)

    def send_pwm_loop(self):
        # Значения PWM на каналы (обычно 1 — roll, 2 — pitch, 3 —
throttle, 4 — yaw)
        #self.get_logger().info(f"self.drone_managenment_f:
{self.drone_managenment_f}  self.main_state {self.main_state}")
        if rclpy.ok() and self.drone_managenment_f:

            self.master.mav.command_long_send(
                self.master.target_system,
                self.master.target_component,
                mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM,
                0,
                1, 0, 0, 0, 0, 0, 0)  # Армирование дрона
            self.master.mav.set_mode_send(
                self.master.target_system,
                mavutil.mavlink.MAV_MODE_FLAG_CUSTOM_MODE_ENABLED,
                VehicleStatus.NAVIGATION_STATE_OFFBOARD)  # где
PX4_OFFBOARD_MODE_ID — числовой ID режима offboard

            # Отправка RC override: 8 каналов, остальным — 0xffff
(игнорировать)
            self.master.mav.rc_channels_override_send(
                self.master.target_system,
                self.master.target_component,
                *self.target_pwm,  # ch1 to ch8
                0xffff, 0xffff, 0xffff, 0xffff  # ch9 to ch12
(игнорировать)
            )
            self.get_logger().info(f"Sent PWM override:
{self.target_pwm}")


    def rpm_to_pwm(self,rpm, rpm_min=0, rpm_max=10000,
pwm_min=1000, pwm_max=2000):
        rpm = max(min(rpm, rpm_max), rpm_min)
        pwm = pwm_min + (rpm - rpm_min) * (pwm_max - pwm_min) /
(rpm_max - rpm_min)
        return int(pwm)

    def esc_status_callback(self, msg: EscStatus):
        rpms = [esc.esc_rpm for esc in msg.esc[:msg.esc_count]]
```

```
        self.motor_rpms = np.array(rpms)
        #self.get_logger().info(f"self.motor_rpms:
{self.motor_rpms}")

        pwms = [self.rpm_to_pwm(rpm) for rpm in self.motor_rpms]
        #self.get_logger().info(f"Estimated PWM: {pwms}")

    def get_pwm(self):
        self.target_pwm = [

int(self.rpm_to_pwm_pid.get_rotate_pwm(self.target_rpm[i],
self.motor_rpms[i]))
            for i in range(4)
        ]
        #self.get_logger().info(f'self.target_rpm:
{self.target_rpm}')

    def drone_managenment(self):
        if self.drone_managenment_f:
            self.send_motor_commands()

    def optimized_traj_callback(self, msg: OptimizedTraj):
        self.received_x_opt = np.array(msg.x_opt,
dtype=np.float32)
        self.received_u_opt = np.array(msg.u_opt,
dtype=np.float32)
        self.target_rpm = np.array(msg.u_opt, dtype=np.float32)
        self.get_pwm()
        self.received_i_final = msg.i_final
        self.received_cost_final = msg.cost_final
        self.received_done = msg.done
        #self.get_logger().info(f'px4 OptimizedTraj: {msg}')

    def send_message_to_server(self, msg):#
        ros_msg = String()
        ros_msg.data = msg
        self.pub_to_mpc.publish(ros_msg)
        #self.get_logger().info(f'Sent to MPC: {msg}')

    def server_msg_callback(self, msg):
        data = msg.data
        #self.get_logger().info(f'server_msg_callback: {data}
self.drone_managenment_f= {self.drone_managenment_f}')
        if data =='land':
            self.main_state == DroneState.LANDING
            self.drone_managenment_f = False
        elif data =='mpc_on':
            self.main_state = DroneState.MPC_MANAGEMENT
            self.drone_managenment_f = True

    def vehicle_status_callback(self, msg):
        """Обновляет состояние дрона."""
        #self.get_logger().info('vehicle_status_callback')
        self.vehicle_status = msg
        self.arming_state = msg.arming_state
```

```
        if msg.nav_state ==
VehicleStatus.NAVIGATION_STATE_OFFBOARD:
            self.offboard_state = True
        else:
            self.offboard_state = False
            self.get_logger().info(f"Текущий режим:
{msg.nav_state}")

    def rpm_to_normalized(self, rpms):
        return [(rpm / self.max_rpm) ** 2 for rpm in rpms]
# MOTOR MANAGEMENT
    def send_motor_commands(self):
        # нормализованные значения [0,1] — переводим в pwm/thrust
команды
        motor_inputs = self.rpm_to_normalized(self.target_rpm)  #
типичный размер: 4
        motor_array = np.clip(motor_inputs, 0.0,
1.0).astype(float).tolist()

        # Дополняем до 12 значений нулями
        if len(motor_array) < 12:
            motor_array += [0.0] * (12 - len(motor_array))
        elif len(motor_array) > 12:
            motor_array = motor_array[:12]  # обрежем лишнее, если
что-то пошло не так

        # Создаем и публикуем сообщение
        msg = ActuatorMotors()
        msg.control = motor_array
        self.publisher_actuator_motors.publish(msg)

        self.get_logger().info(f'Sent motor_commands:
{motor_array}')

    def reset_rate_pid(self):
        self.publish_rate_setpoint(roll_rate=0.0, pitch_rate=0.0,
yaw_rate=0.0)
        self.get_logger().info(f'reset_rate_pid')
    def set_thrust(self, thrust):
            msg = VehicleAttitudeSetpoint()
            msg.thrust_body[2] = -thrust
            self.publisher_att.publish(msg)
# COMMANDS
    def send_takeoff_commanad(self, altitude: float):
        takeoff_cmd = VehicleCommand()
        takeoff_cmd.command =
VehicleCommand.VEHICLE_CMD_NAV_TAKEOFF

        takeoff_cmd.param7 = altitude  # Целевая абсолютная высота
(в метрах)

        # Остальные параметры можно оставить по умолчанию
        takeoff_cmd.param1 = 0.0  # Минимальная высота (не
используется)
```

```
        takeoff_cmd.param2 = 0.0  # Прецизионный режим
        takeoff_cmd.param3 = 0.0  # пусто
        takeoff_cmd.param4 = float('nan')  # yaw
        takeoff_cmd.param5 = float('nan')  # latitude
        takeoff_cmd.param6 = float('nan')  # longitude

        takeoff_cmd.target_system = 1
        takeoff_cmd.target_component = 1
        takeoff_cmd.source_system = 1
        takeoff_cmd.source_component = 1
        takeoff_cmd.from_external = True

        self.vehicle_command_publisher.publish(takeoff_cmd)
        self.get_logger().info(f'Sending takeoff command to
altitude {altitude:.2f} m.')

    def send_land_command(self):
        land_cmd = VehicleCommand()
        land_cmd.command = VehicleCommand.VEHICLE_CMD_NAV_LAND
        # param1: Abort alt (0 = disable)
        land_cmd.param1 = 0.0
        # param2: Precision land mode (0 = disabled, 1 = enabled)
        land_cmd.param2 = 0.0
        # param3: Empty
        land_cmd.param3 = float('nan')
        # param4: Desired yaw angle (NaN = keep current)
        land_cmd.param4 = float('nan')
        # param5, param6: latitude, longitude (NaN = current
position)
        land_cmd.param5 = float('nan')
        land_cmd.param6 = float('nan')
        # param7: Altitude (NaN = current position)
        land_cmd.param7 = float('nan')

        self.vehicle_command_publisher.publish(land_cmd)
        self.get_logger().info('Sending land command.')

    def publish_vehicle_command(self, command, param1=0.0,
param2=0.0):
        """Отправка команды дрону."""
        msg = VehicleCommand()
        msg.param1 = param1
        msg.param2 = param2
        msg.command = command
        msg.target_system = 1
        msg.target_component = 1
        msg.source_system = 1
        msg.source_component = 1
        msg.from_external = True
        msg.timestamp = int(time.time() * 1e6)
        self.vehicle_command_publisher.publish(msg)
        #self.get_logger().info(f'publish_vehicle_command')

# ВКЛЮЧИТЬ РЕЖИМЫ
    def set_stabilization_mode(self):
```

```
        """Переводит дрон в режим стабилизации (STABILIZE)."""
        msg = VehicleCommand()
        msg.command = 1  # Команда для перехода в режим
стабилизации (STABILIZE)
        msg.target_system = 1
        msg.target_component = 1
        msg.source_system = 1
        msg.source_component = 1
        msg.from_external = True
        msg.timestamp = int(time.time() * 1e6)
        self.vehicle_command_publisher.publish(msg)
        self.get_logger().info(f'set_stabilization_mode')

    def arm(self):
        """Send an arm command to the vehicle."""
        self.publish_vehicle_command(
            VehicleCommand.VEHICLE_CMD_COMPONENT_ARM_DISARM,
param1=1.0)
        self.get_logger().info('Arm command sent')

    """ Дрон должен постоянно получать это сообщение чтобы
оставаться в offboard """
    def offboard_heartbeat(self):
         if self.offboard_is_active:
                #self.get_logger().info("Sending SET_MODE
OFFBOARD")
                """Publish the offboard control mode."""
                msg = OffboardControlMode()
                msg.position = True
                msg.velocity = False
                msg.acceleration = False
                msg.attitude = False
                msg.body_rate = False
                msg.timestamp =
int(self.get_clock().now().nanoseconds / 1000)
                self.offboard_control_mode_publisher.publish(msg)
    def set_offboard_mode(self):
            """Switch to offboard mode."""
            self.offboard_is_active = True
    """Управление на моторы нужно подавать непрерывно и с частотой
не мене 0.01 сек"""
    def flip_thrust_max(self):
        if self.flip_thrust_max_f:
            self.set_thrust(1.0)
    def flip_thrust_recovery(self):
        if self.flip_thrust_recovery_f:
            self.set_thrust(0.6)
    def flip_pitch_t(self):
        if self.flip_pitch_f:
            #self.set_rates(17.0, 0.0, 0.0, 0.25)# roll_max_rate
should be 1000 in QGC vechicle setup
            self.set_rates(25.0, 0.0, 0.0, 0.25)


    # main spinned function
```

```
    def update(self):

#self.get_logger().info(f"self.main_state={self.main_state}
self.flip_state={self.flip_state}")
        #self.get_logger().info(f"UPDATE self.alt={self.alt}
self.vehicle_local_position.z={self.vehicle_local_position.z}")
        if self.main_state == DroneState.INIT:
            self.set_offboard_mode()
            self.arm()
            if self.arming_state ==
VehicleStatus.ARMING_STATE_ARMED:
                self.main_state = DroneState.ARMED

        elif self.main_state == DroneState.ARMED:
             if self.offboard_state:
                 self.send_message_to_server("takeoff")#
            #pass

            #self.send_takeoff_commanad(5.0)

        # elif self.main_state == DroneState.MPC_MANAGEMENT:

        elif self.main_state == DroneState.LANDING:
            self.send_land_command()

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

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

```
Panels:
  - Class: rviz_common/Displays
    Help Height: 78
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /Status1
        - /Pose1
        - /Marker1
        - /Marker1/Topic1
        - /Path1
        - /Path2
      Splitter Ratio: 0.5
    Tree Height: 907
  - Class: rviz_common/Selection
    Name: Selection
  - Class: rviz_common/Tool Properties
    Expanded:
      - /2D Goal Pose1
      - /Publish Point1
    Name: Tool Properties
    Splitter Ratio: 0.5886790156364441
  - Class: rviz_common/Views
    Expanded:
      - /Current View1
    Name: Views
    Splitter Ratio: 0.5
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz_default_plugins/Grid
      Color: 160; 160; 164
      Enabled: true
      Line Style:
        Line Width: 0.029999999329447746
        Value: Lines
      Name: Grid
      Normal Cell Count: 0
      Offset:
        X: 0
        Y: 0
        Z: 0
      Plane: XY
      Plane Cell Count: 10
      Reference Frame: <Fixed Frame>
      Value: true
    - Alpha: 1
      Axes Length: 1
      Axes Radius: 0.10000000149011612
      Class: rviz_default_plugins/Pose
```

```
    Color: 255; 25; 0
    Enabled: true
    Head Length: 0.30000001192092896
    Head Radius: 0.10000000149011612
    Name: Pose
    Shaft Length: 1
    Shaft Radius: 0.05000000074505806
    Shape: Axes
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /px4_visualizer/vehicle_pose
    Value: true
  - Class: rviz_default_plugins/Marker
    Enabled: true
    Name: Marker
    Namespaces:
      arrow: true
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /px4_visualizer/vehicle_velocity
    Value: true
  - Alpha: 1
    Buffer Length: 1
    Class: rviz_default_plugins/Path
    Color: 25; 255; 0
    Enabled: true
    Head Diameter: 0.30000001192092896
    Head Length: 0.20000000298023224
    Length: 0.30000001192092896
    Line Style: Lines
    Line Width: 0.029999999329447746
    Name: Path
    Offset:
      X: 0
      Y: 0
      Z: 0
    Pose Color: 255; 85; 255
    Pose Style: None
    Radius: 0.029999999329447746
    Shaft Diameter: 0.10000000149011612
    Shaft Length: 0.10000000149011612
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /px4_visualizer/vehicle_path
    Value: true
  - Alpha: 1
```

```
    Buffer Length: 1
    Class: rviz_default_plugins/Path
    Color: 25; 0; 255
    Enabled: true
    Head Diameter: 0.30000001192092896
    Head Length: 0.20000000298023224
    Length: 0.30000001192092896
    Line Style: Lines
    Line Width: 0.029999999329447746
    Name: Path
    Offset:
      X: 0
      Y: 0
      Z: 0
    Pose Color: 255; 85; 255
    Pose Style: None
    Radius: 0.029999999329447746
    Shaft Diameter: 0.10000000149011612
    Shaft Length: 0.10000000149011612
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /px4_visualizer/setpoint_path
    Value: true
Enabled: true
Global Options:
  Background Color: 255; 255; 255
  Fixed Frame: map
  Frame Rate: 30
Name: root
Tools:
  - Class: rviz_default_plugins/Interact
    Hide Inactive Objects: true
  - Class: rviz_default_plugins/MoveCamera
  - Class: rviz_default_plugins/Select
  - Class: rviz_default_plugins/FocusCamera
  - Class: rviz_default_plugins/Measure
    Line color: 128; 128; 0
  - Class: rviz_default_plugins/SetInitialPose
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /initialpose
  - Class: rviz_default_plugins/SetGoal
    Topic:
      Depth: 5
      Durability Policy: Volatile
      History Policy: Keep Last
      Reliability Policy: Reliable
      Value: /goal_pose
  - Class: rviz_default_plugins/PublishPoint
```

```
      Single click: true
      Topic:
        Depth: 5
        Durability Policy: Volatile
        History Policy: Keep Last
        Reliability Policy: Reliable
        Value: /clicked_point
    Transformation:
      Current:
        Class: rviz_default_plugins/TF
    Value: true
    Views:
      Current:
        Class: rviz_default_plugins/Orbit
        Distance: 24.51659393310547
        Enable Stereo Rendering:
          Stereo Eye Separation: 0.05999999865889549
          Stereo Focal Distance: 1
          Swap Stereo Eyes: false
          Value: false
        Focal Point:
          X: 0
          Y: 0
          Z: 0
        Focal Shape Fixed Size: true
        Focal Shape Size: 0.05000000074505806
        Invert Z Axis: false
        Name: Current View
        Near Clip Distance: 0.009999999776482582
        Pitch: 0.795397937297821
        Target Frame: <Fixed Frame>
        Value: Orbit (rviz)
        Yaw: 0.6553981304168701
      Saved: ~
  Window Geometry:
    Displays:
      collapsed: true
    Height: 1136
    Hide Left Dock: true
    Hide Right Dock: true
    QMainWindow State:
000000ff00000000fd0000000400000000000028e00000416fc0200000008fb00
0000120053006500 6c006500630074006900 6f006e00000001e10000009b000000
5c00ffffffb0000001e0054006f006f006c002000500072006f00700065007200
7400690065007302000001ed000001df00000185000000a3fb0000001200560069
006500770073002000 54006f006f02000001df000002110000018500000122fb00
0000200054006f006f006c002000500072006f00700065007200740069006500730
0320300000028800000 11d000002210000017afb000000100044006900730070000
6c006100790073000000003d00000416000000c900ffffffb0000002000730065
006c006500630074006900 6f006e002000620075006600660065007202000001380
000000aa0000023a00000294fb000000140057006900640065005300740065007 2
0065006f02000000e6000000d2000003ee0000030bfb0000000c004b0069006e00
650063007402000001860000010600000030c000002610000000100000010f000002
f4fc0200000003fb0000001e0054006f006f006c002000500072006f0070006500
7200740069006500730100000041000000780000000000000000fb0000000a0056
```

```
00690065007700730000000003d000002f4000000a400ffffffb000000120053000
65006c00650063007400690006f006e010000025a000000b2000000000000000000
0000020000049000000a9fc0100000001fb0000000a00560069006500770073030
0000004e00000080000002e10000019700000003000004420000003efc01000000
02fb0000000800540069006d00650100000000000004420000000000000000fb00
000008005400690006d00650100000000000004500000000000000000000000039c00
00041600000004000000040000000800000008fc00000001000000020000000100
00000a0054006f006f006c00730100000000ffffffff0000000000000000
  Selection:
    collapsed: false
  Tool Properties:
    collapsed: false
  Views:
    collapsed: true
  Width: 924
  X: 72
  Y: 27
```

Приложение Ж. Сбор данных о процессе выполнения оптимизации траектории

```
=== Log time: 2025-05-19 22:50:18 ===
Final iteration: 4
Final cost: 138147.28125
X_opt    (first    10    values):    [-4.4307336e-02    -2.5175688e-01
-2.2765790e-01 -2.5914019e-02
 -1.7244798e-01 -5.8699165e+01  3.0741501e-03  2.3382830e-02
  6.5383494e-01  7.5626957e-01]
u_opt (first 10 values): [1180. 1200. 1190. 1200.]

=== Log time: 2025-05-19 22:50:18 ===
Final iteration: 4
Final cost: 145510.03125
X_opt    (first    10    values):    [-4.6305872e-02    -2.6506522e-01
-2.2830836e-01 -2.6632650e-02
 -1.7755447e-01 -6.0183941e+01  3.0568789e-03  2.3447664e-02
  6.5371835e-01  7.5636840e-01]
u_opt (first 10 values): [1220. 1245. 1230. 1245.]

=== Log time: 2025-05-19 22:50:18 ===
Final iteration: 4
Final cost: 151833.5625
X_opt    (first    10    values):    [-4.8037294e-02    -2.7658129e-01
-2.2635698e-01 -2.7423382e-02
 -1.8201660e-01 -6.1434353e+01  3.0348089e-03  2.3489693e-02
  6.5377158e-01  7.5632131e-01]
u_opt (first 10 values): [1270. 1300. 1290. 1300.]

=== Log time: 2025-05-19 22:50:18 ===
Final iteration: 4
Final cost: 160766.625
X_opt    (first    10    values):    [-5.0504226e-02    -2.9286703e-01
-2.2635698e-01 -2.8600039e-02
 -1.8782124e-01 -6.3153927e+01  3.0478456e-03  2.3489464e-02
  6.5380388e-01  7.5629330e-01]
u_opt (first 10 values): [1350. 1380. 1360. 1380.]

=== Log time: 2025-05-19 22:50:18 ===
Final iteration: 4
Final cost: 169123.828125
X_opt    (first    10    values):    [-5.2828319e-02    -3.0810258e-01
-2.2830836e-01 -2.9489323e-02
 -1.9274496e-01 -6.4717430e+01  3.1174475e-03  2.3490917e-02
  6.5387279e-01  7.5623333e-01]
u_opt (first 10 values): [1420. 1460. 1440. 1460.]

=== Log time: 2025-05-19 22:50:19 ===
Final iteration: 1
Final cost: 179216.90625
X_opt    (first    10    values):    [-5.5353627e-02    -3.2451746e-01
-2.2635698e-01 -3.0621555e-02
 -1.9789775e-01 -6.6359253e+01  3.2099700e-03  2.3485614e-02
  6.5374029e-01  7.5634772e-01]
```

```
u_opt (first 10 values): [0. 0. 0. 0.]

=== Log time: 2025-05-19 22:50:19 ===
Final iteration: 1
Final cost: 187177.15625
X_opt    (first    10    values):    [-5.7605188e-02    -3.3893576e-01
-2.2700745e-01 -3.1908542e-02
 -2.0237067e-01 -6.7766533e+01  3.2638162e-03  2.3437249e-02
  6.5326667e-01  7.5675809e-01]
u_opt (first 10 values): [0. 0. 0. 0.]
```

Приложение И. Упрощенное решение задачи флипа

```
def flip_thrust_max(self):
        if self.flip_thrust_max_f:
            self.set_thrust(1.0)

    def flip_thrust_recovery(self):
        if self.flip_thrust_recovery_f:
            self.set_thrust(0.6)

    def flip_pitch_t(self):
        if self.flip_pitch_f:
                self.set_rates(17.0, 0.0, 0.0, 0.25)# roll_max_rate
should be 1000 in QGC vechicle setup


    # main spinned function
    def update(self):
        #self.get_logger().info(f"flip_stage: {self.flip_stage}")
              #self.get_logger().info(f"UPDATE  self.alt={self.alt}
self.vehicle_local_position.z={self.vehicle_local_position.z}")
        if self.main_state == DroneState.INIT:
            self.set_offboard_mode()
            self.arm()
            self.main_state = DroneState.ARMING

        elif self.main_state == DroneState.ARMING:
                                self.get_logger().warn(f"arm_state:
{self.arming_state}")
            self.get_logger().info(f"nav_state: {self.nav_state}")
                                        if   self.arming_state   ==
VehicleStatus.ARMING_STATE_ARMED:
                self.get_logger().info('ARMING_STATE_ARMED')
                self.main_state = DroneState.TAKEOFF
            else:
                self.set_offboard_mode()
                self.arm()

        elif self.main_state == DroneState.TAKEOFF:
                            self.publish_position_setpoint(0.0,  0.0,
self.takeoff_height)
                          #self.get_logger().info(f'self.arming_state
{self.arming_state} {self.health_warning}')
                                #self.get_logger().info(f"position.z:
{self.vehicle_local_position.z}                self.takeoff_height:
{self.takeoff_height} res:{self.vehicle_local_position.z  + 0.2 <=
-self.takeoff_height}")
                     if  self.vehicle_local_position.z   -  0.5  <=
-self.takeoff_height:
                self.main_state = DroneState.READY_FOR_FLIP

        elif self.main_state == DroneState.READY_FOR_FLIP:
                                #self.get_logger().info(f'self.roll
angular_velocity={self.angular_velocity[0]}
abs(self.roll={abs(self.roll)}')
```

```
            if self.roll is not None:
                    if (abs(self.angular_velocity[0]) < 0.05 and
(abs(self.roll) < 2.0)):
                    self.main_state = DroneState.FLIP
                    self.stage_time =  time.time()

        ## PITCH FLIP
        elif self.main_state == DroneState.FLIP:
            # Обновление накопленного roll с учетом wrap-around
            roll_diff = self.roll - self.prev_roll
            if roll_diff > 180.0:
                roll_diff -= 360.0
            elif roll_diff < -180.0:
                roll_diff += 360.0

            self.roll_accum += roll_diff
            self.prev_roll = self.roll

            self.get_logger().info(
                                    f"[FLIP]  roll={self.roll:.2f},
roll_diff{roll_diff},             roll_accum={self.roll_accum:.2f},
alt={self.alt:.2f}, flip_state={self.flip_state.name}")

            # 1) Взлет на высоту
            if self.flip_state == DroneFlipState.INIT:
                if self.alt > -6.0:
                    self.flip_thrust_max_f = True
                else:
                    self.flip_thrust_max_f = False
                    self.flip_pitch_f = True
                      self.roll_accum = 0.0  # сброс перед началом
вращения
                    self.prev_roll = self.roll
                    self.flip_state = DroneFlipState.FLIP_INIT

            # 2) Отслеживание прогресса флипа
              elif self.flip_state == DroneFlipState.FLIP_INIT and
self.roll_accum > 45.0:
                self.flip_state = DroneFlipState.FLIP_TURNED_45

             elif self.flip_state == DroneFlipState.FLIP_TURNED_45
and self.roll_accum > 315.0:
                self.flip_state = DroneFlipState.FLIP_TURNED_315
                self.flip_pitch_f = False

            # 3) Восстановление после переворота
                                    elif   self.flip_state   ==
DroneFlipState.FLIP_TURNED_315:
                if abs(self.alt) < 5.0:
                    self.flip_thrust_recovery_f = True
                else:
                    self.flip_thrust_recovery_f = False
                    self.main_state = DroneState.LANDING
```