Промежуточная аттестация. Модуль 9.

ФИО

РЕШЕНИЕ КЕЙСОВ. Диагностические инструменты.

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование задачи | Решение |
| 1 | Разработка простого полётного контроллера. | 1.1) Изучение основных принципов работы полётных контроллеров.  Полётный контроллер (далее по тексту – ПК) представляет из себя электронное устройство, которое координирует работу различных компонентов полётной аппаратуры, таких как датчики, двигатели и системы связи. ПК обрабатывает данные, получаемые от встроенных датчиков, и в соответствии с алгоритмами управления выдает команды.  ПК может состоять из нескольких основных компонентов:   1. Микроконтроллер: получает данные от датчиков и отправляет команды на двигатели. 2. Датчики и сенсоры: собирают информацию о положении, скорости и высоте полёта БЛА. Можно выделить основные сенсоры и датчики: акселерометры, гироскопы, барометры, магнитометры, GPS-модули. 3. Модули связи: обеспечивают связь между БЛА и оператором или НСУ. 4. Элементы управления: обеспечивают управление моторами, преобразуя сигналы от полётного контроллера (например, электронные регуляторы скорости ESC).   ПК являются основными устройствами для всех типов БЛА, включают основные функции:   1. Обработка данных: ПК собирают и обрабатывают данные, получаемые от различных датчиков, что позволяет улучшить качество и безопасность полётов БЛА. 2. Стабилизация полёта: ПК регулируют положение и ориентацию БЛА, анализируют данные, получаемые от датчиков, корректирую углы наклона и пр., тем самым обеспечивая стабильный и безопасный полёт БЛА. 3. Поддержка навигации: ПК позволят БЛА выполнять сложные манёвры, выполнять полёт в режиме автопилота, таким образом обеспечивая безопасный полёт БЛА (например, реализацию функции «возврат на место взлёта») и выполнять заданную программу полёта, которая может включать в себя мониторинг, проверку инфраструктуры или доставки грузов. 4. Учёт внешних факторов: ПК позволяет учитывать влияние погодных условий, например, таких как ветер, и корректировать маршрут и высоту полёта, тем самым повышая безопасность и эффективность полёта БЛА.   Предлагается рассмотреть простой ПК, который будет состоять из акселерометра (MPU6050), гироскопа (MPU6050), барометра (BMP180), GPS-модуля (NEO-6M) и Bluetooth модуля (HC-06). |
| 1.2) Разработка алгоритмов для базовых функций полётного контроллера (взлёт, стабилизация, посадка).  Рассмотрим разработку алгоритмов для взлёта, стабилизации и посадки БЛА:   1. Необходимо выполнить инициализацию всех сенсоров:   Предварительно необходимо загрузить дополнительные библиотеки:  - pip install numpy pandas requests  - pip install dronekit Adafruit-BMP (установка в виртуальной среде)  - pip install bleak  - pip install --upgrade dronekit  - pip install smbus2  - pip install adafruit-circuitpython-bmp3  - pip install pyserial  - pip freeze > requirements.txt (для создания файла requirements.txt)  - python.exe -m pip install --upgrade pip  - pip install –r requirements.txt (для установки требуемых пакетов библиотек)  - pip install serial (для работы с GPS и Bluetooth)  Запускаем в виртуальной среде\_virtual. Инициализация GPS.  from pymavlink import mavutil  import time  # Подключаемся к виртуальному полетному контроллеру  # Необходимо заменить 'udp:127.0.0.1:14552' на свой адрес и порт  connection\_string = 'udp:127.0.0.1:14552'  master = mavutil.mavlink\_connection(connection\_string)  # Ожидание первого heartbeat  master.wait\_heartbeat()  print("Подключено к полетному контроллеру")  # Инициализация GPS  # Необходимо убедиться, что виртуальный контроллер поддерживает включение GPS  # Отправляются GPS данные  gps\_data = {      'lat': 53.059993,  # Необходимо заменить на свою широту      'lon': 63.206572,   # Необходимо заменить на свою долготу      'alt': 0,      # Необходимо заменить на свою высоту      'fix\_type': 3      # 3 — трехмерная фиксация  }  # Отправка GPS данных в контроллер  *def* send\_gps\_data():      master.mav.send(mavutil.mavlink.MAVLink\_gps\_raw\_int\_message(          0,               # Time since boot (ms)          gps\_data['fix\_type'],  # Fix type  *int*(gps\_data['lat'] \* 1e7),  # Latitude  *int*(gps\_data['lon'] \* 1e7),   # Longitude  *int*(gps\_data['alt'] \* 1000),   # Altitude (millimeters)          0,               # GPS horizontal accuracy (mm)          0,               # GPS vertical accuracy (mm)          0,               # GPS satellites visible          0                # DGPS status      ))  # Основной цикл  try:      while True:          send\_gps\_data()          print(*f*"Отправлены данные GPS: {gps\_data}")          time.sleep(1)  # Отправляем данные каждые 1 секунду  except *KeyboardInterrupt*:      print("Остановка...")  finally:      master.close()  PS C:\Users\user\ArduPilot> & "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/virtual/Scripts/python.exe" "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/initialization.py" |
| 1.3) Реализация алгоритмов на языке программирования, поддерживаемом симулятором ArduPilot.  from pymavlink import mavutil  import time  # Подключаемся к виртуальному полетному контроллеру  connection\_string = 'udp:127.0.0.1:14552'  master = mavutil.mavlink\_connection(connection\_string)  # Ждем, пока не получим первый heartbeat  print("Подключение к полетному контроллеру...")  master.wait\_heartbeat()  print("Подключено!")  # GPS данные (обновленные)  gps\_data = {      'lat': 53.059993,  # Замените на свою широту      'lon': 63.206572,   # Замените на свою долготу      'alt': 0,           # Замените на свою высоту      'fix\_type': 3       # 3 — трехмерная фиксация  }  # Функция для отправки GPS данных  *def* send\_gps\_data():      master.mav.send(mavutil.mavlink.MAVLink\_gps\_raw\_int\_message(          0,                     # Время с момента запуска (мс)          gps\_data['fix\_type'],  # Тип фиксации  *int*(gps\_data['lat'] \* 1e7),  # Широта  *int*(gps\_data['lon'] \* 1e7),   # Долгота  *int*(gps\_data['alt'] \* 1000),   # Высота (миллиметры)          0,                      # Горизонтальная точность (мм)          0,                      # Вертикальная точность (мм)          0,                      # Количество видимых спутников          0                       # Статус DGPS      ))      print(*f*"Отправлены данные GPS: {gps\_data}")  # Функция для отправки команды взлета  *def* arm\_and\_takeoff(*aTargetAltitude*):      print("Запускаем...")      master.arducopter\_arm()  # Разблокировка (армер)      print("Дрон разблокирован (армер).")      time.sleep(5)  # Ждем, пока дрон не разблокируется (армится)      # Запускаем взлет      master.mav.command\_long\_send(          master.target\_system,     # ID системы          master.target\_component,  # ID компонента          mavutil.mavlink.MAV\_CMD\_NAV\_TAKEOFF,  # Команда взлета          0,                         # Подтверждение          0,                         # Не используется          0,                         # Широта взлета (автоматически)          0,                         # Долгота взлета (автоматически)          aTargetAltitude,          # Целевая высота          0, 0, 0, 0               # Не используется      )      print(*f*"Взлетаем на высоту {aTargetAltitude} метров.")  # Основная программа  if \_\_name\_\_ == "\_\_main\_\_":      # Отправляем начальные GPS данные      send\_gps\_data()      arm\_and\_takeoff(10)  # Взлет на 10 метров      print("Взлет завершён.")  PS C:\Users\user\ArduPilot> & "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/virtual/Scripts/python.exe" "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/initialization\_takeoff.py"  633;CПодключение к полетному контроллеру... |
| 1.4) Интеграция виртуальных сенсоров и актюаторов в полётный контроллер.  from pymavlink import mavutil  import time  # Подключаемся к виртуальному полетному контроллеру  connection\_string = 'udp:127.0.0.1:14552'  master = mavutil.mavlink\_connection(connection\_string)  # Ожидаем получения первого heartbeat  print("Подключение к полетному контроллеру...")  master.wait\_heartbeat()  print("Подключено!")  # Описание GPS данных с заданными координатами  gps\_data = {      'lat': 53.059993,  # Широта      'lon': 63.206572,  # Долгота      'alt': 0,           # Высота в метрах      'fix\_type': 3       # 3 — трехмерная фиксация  }  # Функция для отправки GPS данных  *def* send\_gps\_data():      # Отправка данных GPS в формате MAVLink      master.mav.send(mavutil.mavlink.MAVLink\_gps\_raw\_int\_message(          0,                             # Время с момента запуска (мс)          gps\_data['fix\_type'],         # Тип фиксации  *int*(gps\_data['lat'] \* 1e7),   # Широта  *int*(gps\_data['lon'] \* 1e7),   # Долгота  *int*(gps\_data['alt'] \* 1000),  # Высота (миллиметры)          0,                             # Горизонтальная точность (мм)          0,                             # Вертикальная точность (мм)          0,                             # Количество видимых спутников          0                              # Статус DGPS      ))      print(*f*"Отправлены данные GPS: {gps\_data}")  # Основной цикл отправки данных GPS  try:      while True:          send\_gps\_data()  # Отправляем GPS данные          time.sleep(1)    # Отправляем данные каждые 1 секунду  except *KeyboardInterrupt*:      print("Остановка интеграции GPS...")  finally:      master.close()  PS C:\Users\user\ArduPilot> & "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/virtual/Scripts/python.exe" "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/virtual\_gps\_int.py"  Подключение к полетному контроллеру... |
| 1.5) Тестирование и отладка разработанного полётного контроллера в симуляторе ArduPilot.  В качестве примера приводится скринкопия полёта БЛА по заданному маршруту без привязки к физическому полётному контроллеру. |
| 2 | Симуляция  полётов. | 2.1) Установка и настройка симулятора ArduPilot  Далее описана установка ArduPilot для операционной системы Windows.   1. Потребуется установка Python. 2. Потребуется установка Git. 3. Загрузка необходимых библиотек (часть из них была описана выше) 4. ArduPilot был установлен в папку «user». 5. Node.js (LTS) был установлен. 6. Путь к Node.js был добавлен в переменную окружения PATH. 7. Была выполнена настройка SITL. 8. На основе исходных данных была подготовлена 3D модель квадрокоптера – quadcopter\_X. 9. Файл был сохранён в директорию ardupilot/Tools/sitl.   Ниже приведён файл quadcopter\_X.urdf:  <robot name="quadcopter">  <link name="base\_link">  <visual>  <geometry>  <mesh filename="model.quadcopter\_X" />  </geometry>  </visual>  </link>  <link name="propeller1">  <visual>  <geometry>  <cylinder radius="1.5" length="10"/> <!-- Propeller size -->  </geometry>  <material name="black">  <color rgba="0 0 0 1"/>  </material>  </visual>  </link>  <link name="propeller2">  <visual>  <geometry>  <cylinder radius="1.5" length="10"/>  </geometry>  <material name="black">  <color rgba="0 0 0 1"/>  </material>  </visual>  </link>  <link name="propeller3">  <visual>  <geometry>  <cylinder radius="1.5" length="10"/>  </geometry>  <material name="black">  <color rgba="0 0 0 1"/>  </material>  </visual>  </link>  <link name="propeller4">  <visual>  <geometry>  <cylinder radius="1.5" length="10"/>  </geometry>  <material name="black">  <color rgba="0 0 0 1"/>  </material>  </visual>  </link>  <link name="flight\_controller">  <visual>  <geometry>  <box size="10 10 2"/> <!-- Size of the flight controller -->  </geometry>  <material name="blue">  <color rgba="0 0 1 1"/>  </material>  </visual>  </link>  <link name="mpu6050">  <visual>  <geometry>  <box size="2 2 1"/> <!-- Size of MPU6050 -->  </geometry>  <material name="red">  <color rgba="1 0 0 1"/>  </material>  </visual>  </link>  <link name="bmp180">  <visual>  <geometry>  <box size="2 2 1"/> <!-- Size of BMP180 -->  </geometry>  <material name="green">  <color rgba="0 1 0 1"/>  </material>  </visual>  </link>  <link name="gps\_module">  <visual>  <geometry>  <box size="2 2 1"/> <!-- Size of GPS Module -->  </geometry>  <material name="yellow">  <color rgba="1 1 0 1"/>  </material>  </visual>  </link>  <link name="bluetooth\_module">  <visual>  <geometry>  <box size="2 2 1"/> <!-- Size of Bluetooth Module -->  </geometry>  <material name="purple">  <color rgba="0.5 0 0.5 1"/>  </material>  </visual>  </link>  <joint name="base\_to\_propeller1" type="fixed">  <parent link="base\_link"/>  <child link="propeller1"/>  <origin rpy="0 0 0" xyz="-25 0 2.5"/>  </joint>  <joint name="base\_to\_propeller2" type="fixed">  <parent link="base\_link"/>  <child link="propeller2"/>  <origin rpy="0 0 0" xyz="25 0 2.5"/>  </joint>  <joint name="base\_to\_propeller3" type="fixed">  <parent link="base\_link"/>  <child link="propeller3"/>  <origin rpy="0 0 0" xyz="0 -25 2.5"/>  </joint>  <joint name="base\_to\_propeller4" type="fixed">  <parent link="base\_link"/>  <child link="propeller4"/>  <origin rpy="0 0 0" xyz="0 25 2.5"/>  </joint>  <joint name="base\_to\_flight\_controller" type="fixed">  <parent link="base\_link"/>  <child link="flight\_controller"/>  <origin rpy="0 0 0" xyz="0 0 6"/>  </joint>  <joint name="controller\_to\_mpu6050" type="fixed">  <parent link="flight\_controller"/>  <child link="mpu6050"/>  <origin rpy="0 0 0" xyz="3 0 1"/>  </joint>  <joint name="controller\_to\_bmp180" type="fixed">  <parent link="flight\_controller"/>  <child link="bmp180"/>  <origin rpy="0 0 0" xyz="-3 0 1"/>  </joint>  <joint name="controller\_to\_gps" type="fixed">  <parent link="flight\_controller"/>  <child link="gps\_module"/>  <origin rpy="0 0 0" xyz="0 3 1"/>  </joint>  <joint name="controller\_to\_bluetooth" type="fixed">  <parent link="flight\_controller"/>  <child link="bluetooth\_module"/>  <origin rpy="0 0 0" xyz="0 -3 1"/>  </joint>  </robot> |
| 2.2) Импорт модели БАС в симулятор.   1. Для работы с созданной 3D моделью квадрокоптера используется SITL симулятор. 2. Для запуска SITL симулятора была выполнена команда ./Tools/autotest/sim\_vehicle.py -v ArduCopter из корневого каталога ArduPilot. 3. Интеграция модели квадрокоптера в Visual Studio Code.   import xml.etree.ElementTree as ET  # Создание корневого элемента  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None):      link = ET.SubElement(robot, "link", *name*=name)      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      if sensor:          servo = ET.SubElement(link, "sensor", *name*=sensor["name"], *type*=sensor["type"])          ET.SubElement(servo, "update\_rate").text = *str*(sensor["update\_rate"])  # Добавление всех links  add\_link("base\_link", {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"})  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue)  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # Запись результата в файл  tree = ET.ElementTree(robot)  tree.write("quadcopter.urdf", *encoding*="utf-8", *xml\_declaration*=True)  Создан файл quadcopter.urdf, который учитывает исходные параметры ПК. |
| 2.3) Настройка параметров симуляции для эмуляции реальных условий полёта.  Добавим параметры высоты, скорости взлёта и скорость вращения пропеллеров:  import xml.etree.ElementTree as ET  # Создание корневого элемента  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link с возможностью указания скорости вращения пропеллеров, скорости взлета и высоты  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)      # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)  # Масса в килограммах          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      # Если это пропеллер, добавим скорость вращения      if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)  # Добавление всех links с физическими параметрами  add\_link("base\_link",            {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"},  *mass*=1.5)  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  # Переменные для изменения скорости вращения пропеллеров,  # скорости взлета и высоты взлета  propeller\_speed\_value = 6000  # об/мин  takeoff\_speed\_value = 3  # м/с  max\_flight\_height = 10  # м  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black, *mass*=0.2, *inertia*=[0.01]\*6, *propeller\_speed*=propeller\_speed\_value)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue, *mass*=0.3)  # Добавление других компонентов  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # Запись результата в файл  tree = ET.ElementTree(robot)  tree.write("quadcopter\_sim.urdf", *encoding*="utf-8", *xml\_declaration*=True)  # Дополнительные параметры, комментарии о скорости взлета и высоте  with open("quadcopter\_sim.urdf", "a", *encoding*="utf-8") as *file*:  # Кодировка  *file*.write("\n<!-- Параметры полета -->\n")  *file*.write("<!-- Скорость взлета: {} м/с -->\n".format(takeoff\_speed\_value))  *file*.write("<!-- Максимальная высота: {} м -->\n".format(max\_flight\_height))  *file*.write("<!-- Скорость вращения пропеллеров: {} об/мин -->\n".format(propeller\_speed\_value))  Создан файл quadcopter\_sim.urdf, который учитывает исходные параметры ПК и дополнительные параметры для симуляции. |
| 2.4) Проведение симуляционных полётов для тестирования полётного контроллера.  Для проведения симуляционных полётов был создан файл virtual\_sensors\_sim.py  import xml.etree.ElementTree as ET  import time  import random  # Импортируем для симуляции данных сенсоров  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link с возможностью указания скорости вращения пропеллеров, скорости взлета и высоты  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)      # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)  # Масса в килограммах          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      # Если это пропеллер, добавим скорость вращения      if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)      # Добавляем информацию о сенсорах      if sensor:          sensor\_elem = ET.SubElement(link, "sensor", *type*=sensor["type"], *name*=sensor["name"])          ET.SubElement(sensor\_elem, "update\_rate").text = *str*(sensor["update\_rate"])  # Добавление всех links с физическими параметрами  add\_link("base\_link",            {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"},  *mass*=1.5)  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  # Переменные для изменения скорости вращения пропеллеров,  # скорости взлета и высоты взлета  propeller\_speed\_value = 6000  # об/мин  takeoff\_speed\_value = 3  # м/с  max\_flight\_height = 10  # м  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black, *mass*=0.2, *inertia*=[0.01]\*6, *propeller\_speed*=propeller\_speed\_value)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue, *mass*=0.3)  # Добавление сенсоров  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # Функция для мониторинга сенсоров  *def* monitor\_sensors(*num\_measurements*=0):      measurements = 0      while num\_measurements == 0 or measurements < num\_measurements:          # Генерируем случайные данные для сенсоров (замените на фактические данные от ваших сенсоров)          imu\_data = {              "acceleration": (random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81)),              "gyro": (random.uniform(-250, 250), random.uniform(-250, 250), random.uniform(-250, 250)),              "temperature": random.uniform(-10, 50)          }          barometer\_data = {              "pressure": random.uniform(950, 1050),  # Давление в гПа              "temperature": random.uniform(-10, 50)   # Температура          }          gps\_data = {              "latitude": random.uniform(37.0, 38.0),              "longitude": random.uniform(-123.0, -122.0),              "altitude": random.uniform(0, 100)  # Высота в метрах          }          # Выводим данные сенсоров на экран          print("IMU Data:", imu\_data)          print("Barometer Data:", barometer\_data)          print("GPS Data:", gps\_data)          measurements += 1          time.sleep(2)  # Пауза между измерениями  # Запускаем мониторинг в отдельном потоке или процессе, если используете многопоточность  if \_\_name\_\_ == "\_\_main\_\_":      try:          num\_measurements = 50  # Можно указать желаемое количество измерений (0 для бесконечного)          monitor\_sensors(num\_measurements)      except *KeyboardInterrupt*:          print("Monitoring stopped.") |
| 2.5) Анализ результатов симуляции и внесение корректировок в алгоритмы управления.  PS C:\Users\user\YandexDisk\VS Code\DRONES\IURII\_Repository\Flight Controller> & "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/virtual/Scripts/python.exe" "c:/Users/user/YandexDisk/VS Code/DRONES/IURII\_Repository/Flight Controller/test\_virtual\_sensors\_sim.py"  ...INFO:root:IMU Data: {'acceleration': (-4.102592961067285, -6.023465403377657, 5.007215474667342), 'gyro': (-202.75010238840036, 241.89636997583943, -130.2547797371236), 'temperature': 7.668106517062011}  INFO:root:Barometer Data: {'pressure': 998.8067478256421, 'temperature': 27.67647092673159}  INFO:root:GPS Data: {'latitude': 37.03758680647693, 'longitude': -122.87685182891137, 'altitude': 83.95825390574726}  INFO:root:IMU Data: {'acceleration': (-6.515410043469204, 1.115398728941198, -5.9266682110238325), 'gyro': (-101.23387940252854, 190.52839110734914, 73.49336559305112), 'temperature': 42.549592796467536}  INFO:root:Barometer Data: {'pressure': 1012.6707455058432, 'temperature': 8.740028318508426}  INFO:root:GPS Data: {'latitude': 37.34464658293757, 'longitude': -122.25278046641269, 'altitude': 16.477358770943738}  .  ----------------------------------------------------------------------  Ran 4 tests in 12.521s  OK  В результате симуляции полётного испытания с сенсоров были собраны данные и проведено модульное тестирование.  import unittest  import xml.etree.ElementTree as ET  import random  import time  import threading  import logging  # Настройка логирования  logging.basicConfig(*level*=logging.INFO)  # Функция для добавления link  *def* add\_link(*robot*, *name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)        # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])        visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")        # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])        if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])        if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)      # Добавляем информацию о сенсорах      if sensor:          sensor\_elem = ET.SubElement(link, "sensor", *type*=sensor["type"], *name*=sensor["name"])          ET.SubElement(sensor\_elem, "update\_rate").text = *str*(sensor["update\_rate"])  # Функция для мониторинга сенсоров  *def* monitor\_sensors(*num\_measurements*=0):      measurements = 0      while num\_measurements == 0 or measurements < num\_measurements:          # Генерируем случайные данные для сенсоров          imu\_data = {              "acceleration": (random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81)),              "gyro": (random.uniform(-250, 250), random.uniform(-250, 250), random.uniform(-250, 250)),              "temperature": random.uniform(-10, 50)          }          logging.info("IMU Data: %s", imu\_data)          barometer\_data = {              "pressure": random.uniform(950, 1050),              "temperature": random.uniform(-10, 50)          }          logging.info("Barometer Data: %s", barometer\_data)          gps\_data = {              "latitude": random.uniform(37.0, 38.0),              "longitude": random.uniform(-123.0, -122.0),              "altitude": random.uniform(0, 100)          }          logging.info("GPS Data: %s", gps\_data)          measurements += 1          time.sleep(2)  # Пауза между измерениями  *class* TestQuadcopterModel(*unittest*.*TestCase*):  *def* setUp(*self*):          """Создание тестового элемента robot перед каждым тестом."""  *self*.robot = ET.Element("robot", *name*="quadcopter")  *def* test\_add\_link(*self*):          """Тестирование добавления link."""          add\_link(*self*.robot, "base\_link", {"type": "box", "size": "1 1 1"}, *mass*=1.0)  *self*.assertEqual(len(*self*.robot.findall("link")), 1)  # Убедимся, что link добавлен          link = *self*.robot.find("link[@name='base\_link']")  *self*.assertIsNotNone(link)  # Убедимся, что link существует          mass = link.find("inertial/mass")  *self*.assertEqual(mass.text, "1.0")  # Убедимся, что масса установлена правильно  *def* test\_add\_sensor(*self*):          """Тестирование добавления сенсора."""          sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}          add\_link(*self*.robot, "mpu6050", {"type": "box", "size": "2 2 1"}, *sensor*=sensor)          sensor\_elem = *self*.robot.find("link[@name='mpu6050']/sensor")  *self*.assertIsNotNone(sensor\_elem)  # Убедимся, что сенсор добавлен  *self*.assertEqual(sensor\_elem.get("name"), "mpu6050\_sensor")  # Проверка имени сенсора  *self*.assertEqual(*int*(sensor\_elem.find("update\_rate").text), 100)  # Проверка частоты обновления  *def* test\_monitor\_sensors(*self*):          """Тестирование функции мониторинга сенсоров."""          num\_measurements = 5          with *self*.assertLogs(*level*='INFO') as log:              monitor\_sensors(num\_measurements)  *self*.assertEqual(len(log.output), num\_measurements \* 3)  # 3 сообщения: IMU, Barometer и GPS  *def* test\_monitor\_sensors\_no\_limit(*self*):          """Тестирование функции мониторинга без ограничения (возможно, с использованием таймаута)."""  *def* monitor\_in\_background():              monitor\_sensors(0)  # Бесконечный мониторинг          monitor\_thread = threading.Thread(*target*=monitor\_in\_background)          monitor\_thread.start()          time.sleep(2)  # Ждем некоторое время          monitor\_thread.join(*timeout*=0.5)  # Остановка потока  if \_\_name\_\_ == '\_\_main\_\_':      unittest.main() |
| 3 | Полётные испытания и анализ данных. | 3.1) Симуляция полётных испытаний с использованием симулятора ArduPilot.  import xml.etree.ElementTree as ET  import time  import random  # Импортируем для симуляции данных сенсоров  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link с возможностью указания скорости вращения пропеллеров, скорости взлета и высоты  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)      # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)  # Масса в килограммах          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      # Если это пропеллер, добавим скорость вращения      if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)      # Добавляем информацию о сенсорах      if sensor:          sensor\_elem = ET.SubElement(link, "sensor", *type*=sensor["type"], *name*=sensor["name"])          ET.SubElement(sensor\_elem, "update\_rate").text = *str*(sensor["update\_rate"])  # Добавление всех links с физическими параметрами  add\_link("base\_link",            {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"},  *mass*=1.5)  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  # Переменные для изменения скорости вращения пропеллеров,  # скорости взлета и высоты взлета  propeller\_speed\_value = 1800  # об/мин  takeoff\_speed\_value = 3  # м/с  max\_flight\_height = 100  # м  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black, *mass*=0.2, *inertia*=[0.01]\*6, *propeller\_speed*=propeller\_speed\_value)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue, *mass*=0.3)  # Добавление сенсоров  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # Функция для мониторинга сенсоров  *def* monitor\_sensors(*num\_measurements*=0):      measurements = 0      while num\_measurements == 0 or measurements < num\_measurements:          # Генерируем случайные данные для сенсоров (замените на фактические данные от ваших сенсоров)          imu\_data = {              "acceleration": (random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81), random.uniform(-9.81, 9.81)),              "gyro": (random.uniform(-250, 250), random.uniform(-250, 250), random.uniform(-250, 250)),              "temperature": random.uniform(-10, 50)          }          barometer\_data = {              "pressure": random.uniform(950, 1050),  # Давление в гПа              "temperature": random.uniform(-10, 50)   # Температура          }          gps\_data = {              "latitude": random.uniform(37.0, 38.0),              "longitude": random.uniform(-123.0, -122.0),              "altitude": random.uniform(0, 100)  # Высота в метрах          }          # Выводим данные сенсоров на экран          print("IMU Data:", imu\_data)          print("Barometer Data:", barometer\_data)          print("GPS Data:", gps\_data)          measurements += 1          time.sleep(2)  # Пауза между измерениями  # Запускаем мониторинг в отдельном потоке или процессе, если используете многопоточность  if \_\_name\_\_ == "\_\_main\_\_":      try:          num\_measurements = 30  # Можно указать желаемое количество измерений (0 для бесконечного)          monitor\_sensors(num\_measurements)      except *KeyboardInterrupt*:          print("Monitoring stopped.") |
| 3.2) Сбор данных сенсоров во время полётов.  В процессе симуляционных полётных испытаний были собраны данные для дальнейшего анализа.  IMU Data: {'acceleration': (-7.394404027721876, -9.752381689485324, -2.8715785169182), 'gyro': (18.185140802281126, 231.21995298686244, 223.09090280860136), 'temperature': 45.70076946986898}  Barometer Data: {'pressure': 1028.560890858493, 'temperature': 32.74964196196926}  GPS Data: {'latitude': 37.62452162360686, 'longitude': -122.18646244392784, 'altitude': 89.10705431564806}  IMU Data: {'acceleration': (7.1725724079087545, 3.322007606567171, 1.664732183874591), 'gyro': (2.444520142968912, -164.57633158098494, -54.826837192694654), 'temperature': 1.7950620796752936}  Barometer Data: {'pressure': 998.3017256604243, 'temperature': 16.307271851277495}  GPS Data: {'latitude': 37.03330082924111, 'longitude': -122.25710639142007, 'altitude': 70.42461685435713}  IMU Data: {'acceleration': (-8.059508808312488, 4.671717903133457, 3.246331130155708), 'gyro': (203.47035469673244, -226.55858723034396, -132.59053086010618), 'temperature': 1.3255728070260364}  Barometer Data: {'pressure': 1024.2417445841604, 'temperature': 39.71673753522318}  GPS Data: {'latitude': 37.81481276206917, 'longitude': -122.01664938256299, 'altitude': 90.28195309247546}  IMU Data: {'acceleration': (0.5927434258050521, 3.144013652101135, 4.190744741479792), 'gyro': (-173.49564501718908, 68.60601271105611, 238.06717486083426), 'temperature': 38.75366264027742}  Barometer Data: {'pressure': 956.3039589155729, 'temperature': -4.798684765596639}  GPS Data: {'latitude': 37.47773589427667, 'longitude': -122.12308366408224, 'altitude': 25.48152088928849}  IMU Data: {'acceleration': (-9.670008934704635, 1.9631238102420827, 5.963108014036882), 'gyro': (-18.34406549829177, -98.64898151476825, -57.146652244703176), 'temperature': -7.452066095327313}  Barometer Data: {'pressure': 1013.3499114344803, 'temperature': 13.382212121296938}  GPS Data: {'latitude': 37.41010099573292, 'longitude': -122.80507913151945, 'altitude': 94.50004820401855}  IMU Data: {'acceleration': (1.1305789778450457, -3.6722425884604313, -0.6652956202276208), 'gyro': (-73.62891163332768, 183.49851729651704, 167.70290599790945), 'temperature': 5.9447864336942295}Barometer Data: {'pressure': 991.0434833515814, 'temperature': 41.38856967033137}  GPS Data: {'latitude': 37.91366889370642, 'longitude': -122.69613756062981, 'altitude': 39.87793423780622}  IMU Data: {'acceleration': (8.307142912601355, 1.7850203827030917, -2.1726367996885747), 'gyro': (152.76860955674704, -201.9984023063241, 17.210155785984398), 'temperature': 23.176461476440203}  Barometer Data: {'pressure': 1014.4279612601299, 'temperature': 31.59639082036294}  GPS Data: {'latitude': 37.80967985478813, 'longitude': -122.67428176405512, 'altitude': 97.20287127646702}  IMU Data: {'acceleration': (-1.5103109084627668, 1.9432143008546259, 1.9518901084607538), 'gyro': (-99.06935043531851, 142.82713562876478, 225.92236428409552), 'temperature': 31.61807997219882}  Barometer Data: {'pressure': 1035.567951032504, 'temperature': 27.295765781794124}  GPS Data: {'latitude': 37.33059041799349, 'longitude': -122.69120874880029, 'altitude': 22.116799989868174}  IMU Data: {'acceleration': (-6.111841211568342, -0.9286175853033214, -4.391425075139307), 'gyro': (-203.47337253519126, 194.19816556150465, 123.74652586425645), 'temperature': 37.5149586567572}  Barometer Data: {'pressure': 1036.0187697713836, 'temperature': 2.748774607839799}  GPS Data: {'latitude': 37.969533783584524, 'longitude': -122.81588315296563, 'altitude': 51.53446720326767}  IMU Data: {'acceleration': (-7.054731752005579, 3.232256845263894, 7.675395106232498), 'gyro': (65.48611711563791, 90.02752609958839, 71.9167632891104), 'temperature': -0.4958189485237767}  Barometer Data: {'pressure': 1036.4964554757169, 'temperature': -9.30626054870684}  GPS Data: {'latitude': 37.213743162646416, 'longitude': -122.113446069569, 'altitude': 0.17192175513133234}  IMU Data: {'acceleration': (-5.6651594097332865, -1.0467454992918483, -8.739611217402347), 'gyro': (249.8162771278378, -203.82618222842774, -198.41669145344886), 'temperature': 39.64167730400638}Barometer Data: {'pressure': 1048.0668993568459, 'temperature': 24.67468803884279}  GPS Data: {'latitude': 37.64902500153865, 'longitude': -122.93081638379337, 'altitude': 28.27023553617799}  IMU Data: {'acceleration': (3.0062758344170675, -0.22472005947408213, 8.983418969637489), 'gyro': (-15.284912769939325, 172.48631907928996, -220.5465732713423), 'temperature': -2.2630477338026633}  Barometer Data: {'pressure': 976.973367752254, 'temperature': 29.221391135944096}  GPS Data: {'latitude': 37.29670307314825, 'longitude': -122.74507974165678, 'altitude': 93.52192841491924}  IMU Data: {'acceleration': (6.565905296124681, 5.274381340229024, 2.8666501516313296), 'gyro': (-9.535136217466714, 220.72442370125896, 71.76084801113154), 'temperature': -7.335308743154576}  Barometer Data: {'pressure': 963.9564409745869, 'temperature': 9.232887338599024}  GPS Data: {'latitude': 37.63998055790353, 'longitude': -122.17985292602943, 'altitude': 84.86600970058892}  IMU Data: {'acceleration': (-0.9549710207182525, -8.384585931147214, 7.606973268455201), 'gyro': (34.03056417651959, 25.378934672143885, 77.51145918014942), 'temperature': -3.6569913552766558}  Barometer Data: {'pressure': 1028.3087987960337, 'temperature': 32.22699383426655}  GPS Data: {'latitude': 37.02772425991719, 'longitude': -122.3574353730459, 'altitude': 65.14071015654574}  IMU Data: {'acceleration': (-3.3289483239983255, 2.20635491130877, -2.3297919069557285), 'gyro': (155.24121926093875, 99.9206746182594, 30.800765993428058), 'temperature': 13.987331131361614}  Barometer Data: {'pressure': 1033.7654697271932, 'temperature': 27.06869618513459}  GPS Data: {'latitude': 37.54491903923182, 'longitude': -122.84223090377289, 'altitude': 34.92047501952635}  IMU Data: {'acceleration': (1.3355856252741045, 0.49656657977559604, 8.59856157766516), 'gyro': (157.94689904732616, -203.63183941389866, 182.5449604034098), 'temperature': 20.87487456036725}  Barometer Data: {'pressure': 985.0143455497226, 'temperature': 6.2944096459646985}  GPS Data: {'latitude': 37.6452302820938, 'longitude': -122.74743538365176, 'altitude': 57.731016303118366}  IMU Data: {'acceleration': (-1.3894259107809308, 4.717129397637905, -8.61725963619792), 'gyro': (32.67172424952912, -121.33090321464852, -104.54868057026857), 'temperature': -6.803850873043881}  Barometer Data: {'pressure': 1040.6673814692535, 'temperature': 28.348587557658412}  GPS Data: {'latitude': 37.10816739436275, 'longitude': -122.49150192655794, 'altitude': 53.25684833235373}  IMU Data: {'acceleration': (4.4058678088837, 6.56460341676271, 7.089772373523237), 'gyro': (76.50260139217875, 171.96419724003175, 91.80605201590782), 'temperature': 30.68829600702086}  Barometer Data: {'pressure': 970.8582020103617, 'temperature': 26.98765767897305}  GPS Data: {'latitude': 37.545331467072366, 'longitude': -122.56615105746769, 'altitude': 72.18591113816318}  IMU Data: {'acceleration': (1.763174897393469, -5.72025039768054, 7.383778523120492), 'gyro': (85.8303156762691, -138.89862951417615, -7.885595721008656), 'temperature': 43.46021669080196}  Barometer Data: {'pressure': 1005.8619630721786, 'temperature': 36.84135549840156}  GPS Data: {'latitude': 37.460311179146316, 'longitude': -122.72513079861955, 'altitude': 86.9335508119413}  IMU Data: {'acceleration': (6.3710809874675665, -2.9538719017547876, 3.366483748376771), 'gyro': (72.63983416121346, -147.72589783172396, 68.41679705235828), 'temperature': 35.308757388242874}  Barometer Data: {'pressure': 1046.6462350942554, 'temperature': 15.77099854200955}  GPS Data: {'latitude': 37.205068409566096, 'longitude': -122.998548248593, 'altitude': 41.444883014656156}  IMU Data: {'acceleration': (0.7449155823033493, -3.6165078627379668, 4.6392575810965955), 'gyro': (161.81331252882615, 237.98355089880675, -152.19795398518576), 'temperature': 42.67108873904669}  Barometer Data: {'pressure': 964.3093965668669, 'temperature': 10.608398593610488}  GPS Data: {'latitude': 37.217619268810026, 'longitude': -122.60009411585217, 'altitude': 86.34483479435751}  IMU Data: {'acceleration': (1.8360006144602252, 2.3044923904720225, 0.15514987211222575), 'gyro': (-23.899030204871167, 92.79623963525711, 78.16837588578653), 'temperature': 34.10571758485106}  Barometer Data: {'pressure': 1048.6393091756192, 'temperature': 13.799347899095721}  GPS Data: {'latitude': 37.71899142007425, 'longitude': -122.76354049654543, 'altitude': 76.34604231071279}  IMU Data: {'acceleration': (-8.468213173151431, -0.27443030193617624, -1.9927954157313588), 'gyro': (-68.43683275690032, -249.73771191923217, 59.92521389864805), 'temperature': 36.4950652642814}  Barometer Data: {'pressure': 959.335785248945, 'temperature': 17.916139115093863}  GPS Data: {'latitude': 37.33181298521079, 'longitude': -122.42346400866793, 'altitude': 42.07504632565854}  IMU Data: {'acceleration': (-0.22684259434175402, 3.780692787626851, -7.874040494593995), 'gyro': (-217.18878886848194, 248.55133767262333, -214.98984895438244), 'temperature': 44.22546792046427}Barometer Data: {'pressure': 955.5550396339938, 'temperature': 30.380968283193113}  GPS Data: {'latitude': 37.91539085255275, 'longitude': -122.2068790815938, 'altitude': 62.541063664369766}  IMU Data: {'acceleration': (9.582318276334663, 9.761072463709555, 2.4922995719204586), 'gyro': (27.38497671023316, -165.0753865711394, 68.42435486669177), 'temperature': 13.574835786141243}  Barometer Data: {'pressure': 985.239718030942, 'temperature': 37.835953584830605}  GPS Data: {'latitude': 37.47580926721807, 'longitude': -122.95823382815495, 'altitude': 98.89596576712782}  IMU Data: {'acceleration': (-6.145409231205244, -9.695375352579983, 3.576625910507442), 'gyro': (-172.95779380760297, -54.39245730855319, 38.51580597802234), 'temperature': 18.90126311624661}  Barometer Data: {'pressure': 1037.8994961372355, 'temperature': 24.32151869611441}  GPS Data: {'latitude': 37.52159184323151, 'longitude': -122.84082182399538, 'altitude': 97.62452194844933}  IMU Data: {'acceleration': (-7.388026466358212, -8.11179370677721, -0.06641454524596924), 'gyro': (186.21128498013883, 184.6835365823727, -178.9608991540639), 'temperature': 29.231667798299497}  Barometer Data: {'pressure': 1036.814989425536, 'temperature': -8.87456253645345}  GPS Data: {'latitude': 37.82305482300036, 'longitude': -122.95766852165353, 'altitude': 83.25555273387302}  IMU Data: {'acceleration': (-5.354306536563237, -7.3530980994262025, -9.180302156071397), 'gyro': (112.36994528002725, -101.30871510743921, 129.54328886996785), 'temperature': 43.16838277720344}  Barometer Data: {'pressure': 982.3761678028585, 'temperature': 25.06549642009614}  GPS Data: {'latitude': 37.795740158405565, 'longitude': -122.02692745269844, 'altitude': 43.1623672497935}  IMU Data: {'acceleration': (-6.236955659708327, 5.443724917929396, -4.9627226008600065), 'gyro': (62.39232685756548, -236.0319714313851, 156.67107396967776), 'temperature': 5.440212872332372}  Barometer Data: {'pressure': 973.3824069560922, 'temperature': 37.946735827806734}  GPS Data: {'latitude': 37.064755991318705, 'longitude': -122.92693321668352, 'altitude': 2.444696906241639}  IMU Data: {'acceleration': (-3.0817134072244254, 4.945427149072422, -0.7180153359472108), 'gyro': (20.29476804151477, -72.3039265506304, 196.55737220442563), 'temperature': -1.741546465425328}  Barometer Data: {'pressure': 999.1240801876885, 'temperature': 11.114365344357871}  GPS Data: {'latitude': 37.05221557831556, 'longitude': -122.88393669845269, 'altitude': 47.22841053653957} |
| 3.3) Обработка и анализ собранных данных.  import pandas as pd  import matplotlib  matplotlib.use('Agg')  # Используйте бэкенд без GUI  import matplotlib.pyplot as plt  import re  # Функция для чтения данных из файла  *def* read\_data(*file\_path*):      with open(file\_path, 'r') as *file*:          data = *file*.read().strip()      return data  # Укажите путь к вашему файлу analyze.txt  file\_path = 'analyze.txt'  # Чтение данных из файла  data = read\_data(file\_path)  # Разделение данных на строки  lines = data.split('\n')  # Списки для каждого типа данных  imu\_list = []  barometer\_list = []  gps\_list = []  # Обработка строк данных  for line in lines:      if "IMU Data" in line:          imu\_data = re.findall(*r*"[-+]?\d\*\.\d+|\d+", line)          if len(imu\_data) >= 7:              imu\_list.append([*float*(imu\_data[i]) for i in range(6)] + [*float*(imu\_data[6])])      elif "Barometer Data" in line:          barometer\_data = re.findall(*r*"[-+]?\d\*\.\d+|\d+", line)          if len(barometer\_data) == 2:              barometer\_list.append([*float*(i) for i in barometer\_data])      elif "GPS Data" in line:          gps\_data = re.findall(*r*"[-+]?\d\*\.\d+|\d+", line)          gps\_list.append([*float*(i) for i in gps\_data])  # Собираем данные GPS  # Создание DataFrame из собранных данных  imu\_df = pd.DataFrame(imu\_list, *columns*=['acc\_x', 'acc\_y', 'acc\_z', 'gyro\_x', 'gyro\_y', 'gyro\_z', 'temperature'])  barometer\_df = pd.DataFrame(barometer\_list, *columns*=['pressure', 'temp'])  gps\_df = pd.DataFrame(gps\_list, *columns*=['latitude', 'longitude', 'altitude'])  # Функция анализа и сохранения IMU данных  *def* analyze\_imu\_data(*df*):      plt.figure(*figsize*=(14, 4))        plt.subplot(1, 3, 1)      plt.plot(df['acc\_x'], *label*='Acceleration X')      plt.plot(df['acc\_y'], *label*='Acceleration Y')      plt.plot(df['acc\_z'], *label*='Acceleration Z')      plt.title('IMU Acceleration')      plt.xlabel('Sample Number')      plt.ylabel('Acceleration (m/s²)')      plt.legend()      plt.grid()        plt.subplot(1, 3, 2)      plt.plot(df['gyro\_x'], *label*='Gyro X', *color*='r')      plt.plot(df['gyro\_y'], *label*='Gyro Y', *color*='g')      plt.plot(df['gyro\_z'], *label*='Gyro Z', *color*='b')      plt.title('IMU Gyroscope')      plt.xlabel('Sample Number')      plt.ylabel('Gyroscope (°/s)')      plt.legend()      plt.grid()        plt.subplot(1, 3, 3)      plt.plot(df['temperature'], *label*='Temperature', *color*='purple')      plt.title('IMU Temperature')      plt.xlabel('Sample Number')      plt.ylabel('Temperature (°C)')      plt.legend()      plt.grid()      plt.tight\_layout()      plt.savefig('imu\_data\_analysis.png')  # Сохранить график в файл      plt.close()  # Закрыть текущую фигуру, чтобы освободить память  # Функция анализа и сохранения данных барометра  *def* analyze\_barometer\_data(*df*):      plt.figure(*figsize*=(7, 4))        plt.plot(df['pressure'], *label*='Pressure', *color*='blue')      plt.plot(df['temp'], *label*='Temperature', *color*='red')      plt.title('Barometer Data')      plt.xlabel('Sample Number')      plt.ylabel('Pressure (hPa) / Temperature (°C)')      plt.legend()      plt.grid()      plt.tight\_layout()      plt.savefig('barometer\_data\_analysis.png')  # Сохранить график в файл      plt.close()  # Закрыть текущую фигуру  # Функция анализа и сохранения GPS данных  *def* analyze\_gps\_data(*df*):      plt.figure(*figsize*=(7, 4))        plt.plot(df['latitude'], *label*='Latitude', *color*='green')      plt.plot(df['longitude'], *label*='Longitude', *color*='orange')      plt.plot(df['altitude'], *label*='Altitude', *color*='purple')      plt.title('GPS Data')      plt.xlabel('Sample Number')      plt.ylabel('GPS Coordinates / Altitude')      plt.legend()      plt.grid()      plt.tight\_layout()      plt.savefig('gps\_data\_analysis.png')  # Сохранить график в файл      plt.close()  # Закрыть текущую фигуру  # Выполнение анализа данных  try:      analyze\_imu\_data(imu\_df)      analyze\_barometer\_data(barometer\_df)      analyze\_gps\_data(gps\_df)  except *Exception* as e:      print(*f*"An error occurred during analysis: {e}") |
| 3.4) Визуализация данных для выявления аномалий и ошибок. |
| 3.5) Внесение корректировок в алгоритмы управления на основе анализа данных.  Для сглаживания показателей необходимо добавить PID-регуляторы в модель квадрокоптера.  import xml.etree.ElementTree as ET  import random  import time  import pandas as pd  import matplotlib  matplotlib.use('Agg')  # Используйте бэкенд без GUI  import matplotlib.pyplot as plt  import re  # Класс PID-регулятора  *class* PID:  *def* \_\_init\_\_(*self*, *kp*, *ki*, *kd*, *setpoint*):  *self*.kp = kp  # Пропорциональный коэффициент  *self*.ki = ki  # Интегральный коэффициент  *self*.kd = kd  # Дифференциальный коэффициент  *self*.setpoint = setpoint  # Целевая величина  *self*.last\_error = 0  # Последняя ошибка  *self*.integral = 0  # Интеграл ошибки  *def* update(*self*, *measured\_value*):          error = *self*.setpoint - measured\_value  # Ошибка  *self*.integral += error  # Интегрирование ошибки          derivative = error - *self*.last\_error  # Производная ошибки  *self*.output = (*self*.kp \* error) + (*self*.ki \* *self*.integral) + (*self*.kd \* derivative)  # Управляющий сигнал  *self*.last\_error = error  # Обновление последней ошибки          return *self*.output  # Создание корневого элемента  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)      # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)  # Масса в килограммах          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      # Если это пропеллер, добавим скорость вращения      if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)  # Добавление всех links с физическими параметрами  add\_link("base\_link",            {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"},  *mass*=1.5)  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  # Переменные для изменения скорости вращения пропеллеров, скорости взлета и высоты взлета  propeller\_speed\_value = 1900  # об/мин  takeoff\_speed\_value = 3  # м/с  max\_flight\_height = 80  # м  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black, *mass*=0.2, *inertia*=[0.01]\*6, *propeller\_speed*=propeller\_speed\_value)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue, *mass*=0.3)  # Добавление других компонентов  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # PID-регулирование для управления скоростью вращения пропеллеров и высотой  propeller\_pid = PID(1.0, 0.1, 0.05, propeller\_speed\_value)  height\_pid = PID(1.0, 0.1, 0.05, max\_flight\_height)  # Списки для хранения значений для графиков  propeller\_speed\_history = []  desired\_propeller\_speed\_history = []  height\_history = []  desired\_height\_history = []  # Симуляция регулирования  for \_ in range(20):  # Симуляция 20 итераций      # Симуляция значений от сенсоров      current\_propeller\_speed = random.uniform(1800, 2300)  # случайная скорость пропеллеров      current\_height = random.uniform(60, 100)  # случайная высота      # Обновляем PID-регуляторы      new\_propeller\_speed = propeller\_pid.update(current\_propeller\_speed)      new\_height = height\_pid.update(current\_height)      # Сохраняем значения для графиков      propeller\_speed\_history.append(current\_propeller\_speed)      desired\_propeller\_speed\_history.append(new\_propeller\_speed)      height\_history.append(current\_height)      desired\_height\_history.append(new\_height)      print(*f*"Current Propeller Speed: {current\_propeller\_speed*:.2f*}, New Desired Speed: {new\_propeller\_speed*:.2f*}")      print(*f*"Current Height: {current\_height*:.2f*}, New Desired Height: {new\_height*:.2f*}")      time.sleep(0.5)  # Задержка для имитации времени между измерениями  # Построение и сохранение графиков  plt.figure(*figsize*=(12, 6))  # График скорости пропеллеров  plt.subplot(2, 1, 1)  plt.plot(propeller\_speed\_history, *label*='Current Propeller Speed', *color*='b')  plt.plot(desired\_propeller\_speed\_history, *label*='Desired Propeller Speed', *color*='r', *linestyle*='--')  plt.title('Propeller Speed')  plt.xlabel('Time (s)')  plt.ylabel('Speed (RPM)')  plt.legend()  plt.grid()  # График высоты  plt.subplot(2, 1, 2)  plt.plot(height\_history, *label*='Current Height', *color*='g')  plt.plot(desired\_height\_history, *label*='Desired Height', *color*='orange', *linestyle*='--')  plt.title('Flight Height')  plt.xlabel('Time (s)')  plt.ylabel('Height (m)')  plt.legend()  plt.grid()  # Сохранение графиков в файл  plt.tight\_layout()  plt.savefig('quadcopter\_control\_analysis.png')  plt.close()  import xml.etree.ElementTree as ET  import random  import time  import pandas as pd  import matplotlib  matplotlib.use('Agg')  # Используйте бэкенд без GUI  import matplotlib.pyplot as plt  import re  # Класс PID-регулятора  *class* PID:  *def* \_\_init\_\_(*self*, *kp*, *ki*, *kd*, *setpoint*):  *self*.kp = kp  # Пропорциональный коэффициент  *self*.ki = ki  # Интегральный коэффициент  *self*.kd = kd  # Дифференциальный коэффициент  *self*.setpoint = setpoint  # Целевая величина  *self*.last\_error = 0  # Последняя ошибка  *self*.integral = 0  # Интеграл ошибки  *def* update(*self*, *measured\_value*):          error = *self*.setpoint - measured\_value  # Ошибка  *self*.integral += error  # Интегрирование ошибки          derivative = error - *self*.last\_error  # Производная ошибки  *self*.output = (*self*.kp \* error) + (*self*.ki \* *self*.integral) + (*self*.kd \* derivative)  # Управляющий сигнал  *self*.last\_error = error  # Обновление последней ошибки          return *self*.output  # Создание корневого элемента  robot = ET.Element("robot", *name*="quadcopter")  # Функция для добавления link  *def* add\_link(*name*, *geometry*, *material*=None, *sensor*=None, *mass*=0, *inertia*=None, *propeller\_speed*=None):      link = ET.SubElement(robot, "link", *name*=name)      # Установка физики      if mass > 0:          inertia\_elem = ET.SubElement(link, "inertial")          ET.SubElement(inertia\_elem, "mass").text = *str*(mass)  # Масса в килограммах          if inertia:              inertia\_elem = ET.SubElement(inertia\_elem, "inertia")              ET.SubElement(inertia\_elem, "ixx").text = *str*(inertia[0])              ET.SubElement(inertia\_elem, "ixy").text = *str*(inertia[1])              ET.SubElement(inertia\_elem, "ixz").text = *str*(inertia[2])              ET.SubElement(inertia\_elem, "iyy").text = *str*(inertia[3])              ET.SubElement(inertia\_elem, "iyz").text = *str*(inertia[4])              ET.SubElement(inertia\_elem, "izz").text = *str*(inertia[5])      visual = ET.SubElement(link, "visual")      geom = ET.SubElement(visual, "geometry")      # Определение типа геометрии      if geometry["type"] == "mesh":          ET.SubElement(geom, "mesh", *filename*=geometry["filename"])      elif geometry["type"] == "box":          ET.SubElement(geom, "box", *size*=geometry["size"])      elif geometry["type"] == "cylinder":          ET.SubElement(geom, "cylinder", *radius*=geometry["radius"], *length*=geometry["length"])      if material:          mat = ET.SubElement(visual, "material", *name*=material["name"])          ET.SubElement(mat, "color", *rgba*=material["rgba"])      # Если это пропеллер, добавим скорость вращения      if propeller\_speed is not None:          ET.SubElement(link, "propeller\_speed").text = *str*(propeller\_speed)  # Добавление всех links с физическими параметрами  add\_link("base\_link",            {"type": "mesh", "filename": "C:/Users/user/ardupilot/Tools/gazebo/models/quadcopter\_X/quadcopter\_X.stl"},  *mass*=1.5)  propeller\_geometry = {"type": "cylinder", "radius": "1.5", "length": "10"}  material\_black = {"name": "black", "rgba": "0 0 0 1"}  # Переменные для изменения скорости вращения пропеллеров, скорости взлета и высоты взлета  propeller\_speed\_value = 1900  # об/мин  takeoff\_speed\_value = 3  # м/с  max\_flight\_height = 80  # м  for i in range(1, 5):      add\_link(*f*"propeller{i}", propeller\_geometry, material\_black, *mass*=0.2, *inertia*=[0.01]\*6, *propeller\_speed*=propeller\_speed\_value)  flight\_controller\_geometry = {"type": "box", "size": "10 10 2"}  material\_blue = {"name": "blue", "rgba": "0 0 1 1"}  add\_link("flight\_controller", flight\_controller\_geometry, material\_blue, *mass*=0.3)  # Добавление других компонентов  mpu6050\_sensor = {"name": "mpu6050\_sensor", "type": "IMU", "update\_rate": 100}  add\_link("mpu6050", {"type": "box", "size": "2 2 1"}, {"name": "red", "rgba": "1 0 0 1"}, mpu6050\_sensor)  bmp180\_sensor = {"name": "bmp180\_sensor", "type": "Barometer", "update\_rate": 50}  add\_link("bmp180", {"type": "box", "size": "2 2 1"}, {"name": "green", "rgba": "0 1 0 1"}, bmp180\_sensor)  gps\_sensor = {"name": "gps\_sensor", "type": "GPS", "update\_rate": 1}  add\_link("gps\_module", {"type": "box", "size": "2 2 1"}, {"name": "yellow", "rgba": "1 1 0 1"}, gps\_sensor)  add\_link("bluetooth\_module", {"type": "box", "size": "2 2 1"}, {"name": "purple", "rgba": "0.5 0 0.5 1"})  # Функция для добавления joint  *def* add\_joint(*name*, *parent*, *child*, *origin*):      joint = ET.SubElement(robot, "joint", *name*=name, *type*="fixed")      ET.SubElement(joint, "parent", *link*=parent)      ET.SubElement(joint, "child", *link*=child)      ET.SubElement(joint, "origin", *rpy*="0 0 0", *xyz*=origin)  # Добавление всех joints  add\_joint("base\_to\_propeller1", "base\_link", "propeller1", "-25 0 2.5")  add\_joint("base\_to\_propeller2", "base\_link", "propeller2", "25 0 2.5")  add\_joint("base\_to\_propeller3", "base\_link", "propeller3", "0 -25 2.5")  add\_joint("base\_to\_propeller4", "base\_link", "propeller4", "0 25 2.5")  add\_joint("base\_to\_flight\_controller", "base\_link", "flight\_controller", "0 0 6")  add\_joint("controller\_to\_mpu6050", "flight\_controller", "mpu6050", "3 0 1")  add\_joint("controller\_to\_bmp180", "flight\_controller", "bmp180", "-3 0 1")  add\_joint("controller\_to\_gps", "flight\_controller", "gps\_module", "0 3 1")  add\_joint("controller\_to\_bluetooth", "flight\_controller", "bluetooth\_module", "0 -3 1")  # PID-регулирование для управления скоростью вращения пропеллеров и высотой  propeller\_pid = PID(1.0, 0.1, 0.05, propeller\_speed\_value)  height\_pid = PID(1.0, 0.1, 0.05, max\_flight\_height)  # Списки для хранения значений для графиков  propeller\_speed\_history = []  desired\_propeller\_speed\_history = []  height\_history = []  desired\_height\_history = []  # Симуляция регулирования  for \_ in range(20):  # Симуляция 20 итераций      # Симуляция значений от сенсоров      current\_propeller\_speed = random.uniform(1800, 2300)  # случайная скорость пропеллеров      current\_height = random.uniform(60, 100)  # случайная высота      # Обновляем PID-регуляторы      new\_propeller\_speed = propeller\_pid.update(current\_propeller\_speed)      new\_height = height\_pid.update(current\_height)      # Сохраняем значения для графиков      propeller\_speed\_history.append(current\_propeller\_speed)      desired\_propeller\_speed\_history.append(new\_propeller\_speed)      height\_history.append(current\_height)      desired\_height\_history.append(new\_height)      print(*f*"Current Propeller Speed: {current\_propeller\_speed*:.2f*}, New Desired Speed: {new\_propeller\_speed*:.2f*}")      print(*f*"Current Height: {current\_height*:.2f*}, New Desired Height: {new\_height*:.2f*}")      time.sleep(0.5)  # Задержка для имитации времени между измерениями  # Построение и сохранение графиков  plt.figure(*figsize*=(12, 6))  # График скорости пропеллеров  plt.subplot(2, 1, 1)  plt.plot(propeller\_speed\_history, *label*='Current Propeller Speed', *color*='b')  plt.plot(desired\_propeller\_speed\_history, *label*='Desired Propeller Speed', *color*='r', *linestyle*='--')  plt.title('Propeller Speed')  plt.xlabel('Time (s)')  plt.ylabel('Speed (RPM)')  plt.legend()  plt.grid()  # График высоты  plt.subplot(2, 1, 2)  plt.plot(height\_history, *label*='Current Height', *color*='g')  plt.plot(desired\_height\_history, *label*='Desired Height', *color*='orange', *linestyle*='--')  plt.title('Flight Height')  plt.xlabel('Time (s)')  plt.ylabel('Height (m)')  plt.legend()  plt.grid()  # Сохранение графиков в файл  plt.tight\_layout()  plt.savefig('quadcopter\_control\_analysis.png')  plt.close() |