Вторник, 19. ноября 2019 14:04
Автор: b-o-r-m-a-l-e-y
есле копипастиш, то указывай авторство, хотя это лишь перевод с кучами объяснений
TODO:

- разобраться с ошибками при указании начального местоположения коптера в функции `slightly_more_complex_usage()`
- составление и работа с траекториями
- оставшиеся примеры

Разбор примеров работы с библиотекой cflib

Для этого понадобится:

- MS Visual Code (или любой другой редактор кода, мне етот нрав)
- Установленный в ваш редактор плагин Python (для автоподсветки и автодополнения кода)
- Python 3.7 или новее
- Установленная библиотека cflib. Для установки открыть консоль (Поиск->cmd) и набрать `pip install cflib`
- моск

Попробуем разобраться что к чему. Для этого нужно знать циклы, немного ООП, что такое колбеки и уметь пользоваться гуглом. По синтаксису Python советую почитать Марк Лутц - “Изучаем Python”, конечно если на это есть время.
При запуске коптеров их нужно всегда поворачивать вдоль по оси X.

Самый первый пример - scan.py

```
# -*- coding: utf-8 -*-
"""
Simple example that scans for available Crazyflies and lists them.
"""
import cflib.crtp

# Initiate the low level drivers
cflib.crtp.init_drivers(enable_debug_driver=False)

print('Scanning interfaces for Crazyflies...')
available = cflib.crtp.scan_interfaces()
print('Crazyflies found:')
for i in available:
    print(i[0])
```

Первая закоментированная строчка должна присутствовать во всех питоновских файлах и обозначает она кодировку, в которой хранится файл. Иногда интерпретатор может ругаться на её отсутствие (несмотря на то, что она является комментарием).
Пример сканирует все имеющиеся (и включенные) коптеры и выводит их список в консоль. Строка `import cflib.crtp` подключает из всей библиотеки cflib только модуль crtp. За что отвечает этот модуль, можете узнать кликнув по нему в среде разработки -> Go to Defenition и вас перекинет на файл, в котором объявлен данный модуль. Там будут комментарии о том что делают функции, объявленные в этом модуле, классы и прочее.
Строка `cflib.crtp.init_drivers(enable_debug_driver=False)` инициализирует драйвера,

необходимые для связи. Вызов функции происходит в соответствии с её иерархией: библиотека->модуль->функция. Однако есть другой популярный вариант: можно было подключить модуль следующим образом `import cflib.crtp as cr` и затем уже обращаться к функциям из модуля через `cr.init_drivers(...)`. Аргумент функции отключает режим отладки (нужда в нем возникает при разработке дополнительных модулей для библиотеки).
В переменную `available` запишется список коптеров с их параметрами (адреса и прочее), который вернет функция `scan_interfaces()`. Напоминаю, что в питоновских циклах, на каждом его шаге, переменная `i` будет принимать значение текущего (нулевого, первого и т.д до конца) элемента списка (**не индекс элемента, а сам элемент**). Поэтому при помощи цикла мы для каждого коптера выдираем параметр с индексом 0 из всех (это его адрес) и выводим в консоль.

Вид списка надо уточнить

Модули библиотеки для позиционирования одного коптера

Их всего два, однако примеров в папке examples больше:

- **MotionCommander** - задает “скоростные” сетпоинты, т.е. указывается скорость и куда лететь, а не конкретная точка в пространстве. Предназначен для использования с Flow deck (платка, которая крепится внизу коптера и при помощи оптического датчика определяет передвижение относительно земли). Включается модуль следующим образом: `from cflib.positioning.motion_commander import MotionCommander`. Однако его мы использовать не будем, ввиду ~~недальновидности при закупке обурудования~~ отсутствия Flow Deck.
- Есть ещё пример *flowsequenceSync.py*, но он тоже заточен под работу с Flow Deck, поэтому для использования с системой Loco Positioning необходимы тесты. Разница между предыдущим примером состоит в использовании метода `send_hover_setpoint(vx, vy, yawrate, zdistance)`.
- Пример *position_commander_demo.py* иллюстрирует работу с классом **PositionHlCommander**. Через этот класс можно задать координаты точек на местности. После взлета коптер летит к точке с нулевыми координатами, это в случае, если начальные координаты явно не были указаны в конструкторе класса. Иерархия включения следующая: `from cflib.positioning.position_hl_commander import PositionHlCommander`.
- Пример *autonomousSequence.py* работает также как предыдущий, но на более низком уровне. В самом примере содержится обработка данных о положении коптера и сброс данных о местоположении. Команды передаются коптеру через класс `Commander`, который находится внутри объекта класса `Crazyflie` при помощи метода `cf.commander.send_position_setpoint(x, y, z, yaw)`. Здесь `cf` - объект класса CrazyFlie. Есть небольшая особенность - если необходимо, чтобы коптер завис на одном месте, то такую команду нужно посылать много раз в течение требуемого интервала времени, иначе произойдет сброс.
- Следующий пример находится в папке ‘positioning’ с примерами и называется *initial_positioning.py*. Разница между предыдущим состоит в том, что дополнительно ко всему прочему указывается начальное положение коптера.
- Пример *autonomous_sequence_high_level.py* представляет собой работу с траекториями (в данном случае - восьмерка). При этом сама траектория загружается в память коптера. Для создания траекторий полета используется софтина, причём сама траектория при этом описывается полиномом.

Отдельные конструкции, используемые в примерах

Далее будет представлены кусочки кода, которые можно встретить при работе с библиотекой и копании в примерах. Неплохо бы в них разобраться прежде чем идти дальше.

Защита от включения модулей

```
if __name__ == '__main__':
    cflib.crtp.init_drivers(enable_debug_driver=False)

    simple_sequence()
    # slightly_more_complex_usage()
```

Питон - язык интерпретируемый, а не комплируемый, значит программа будет сразу исполняться построчно, а не собираться в единый бинарный (исполняемый) файл и затем запускаться, как это было в случае C++. Условие вверху проверяет, является ли скрипт (файл), который вы запускаете, главным. Если этот же скрипт вы включите в другой файл при помощи `import ...`, то команды, написанные внутри условия, выполнены не будут.

Менеджер контекста

```
with PositionHlCommander(scf) as pc:
    pc.forward(1.0)
    pc.left(1.0)
    pc.back(1.0)
    pc.go_to(0.0, 0.0, 1.0)
```

Стандартная проблема при программировании - если вы открыли файл для записи, то по окончании программы вы обязаны его закрыть. Точно также с выделением памяти (после использования её нужно освободить). Конструкция `with ... as...` обеспечивает отлов исключительных ситуаций.

Подумоть че во ещё добавить асинхонщина и колбеки

Разбор работы с классом PostionHlCommander (position_commander_demo.py)

Скрипт соединяется с одним коптером по URI и выполняет указанные команды.

```
"""
This script shows the basic use of the PositionHlCommander class.

Simple example that connects to the crazyflie at `URI` and runs a
sequence. This script requires some kind of location system.

The PositionHlCommander uses position setpoints.

Change the URI variable to your Crazyflie configuration.
"""
import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.position_hl_commander import PositionHlCommander

# URI to the Crazyflie to connect to
uri = 'radio://0/80/2M/E7E7E7E7E7'


def slightly_more_complex_usage():
    with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
```

```
        with PositionHlCommander(
                scf,
                x=0.0, y=0.0, z=0.0,
                default_velocity=0.3,
                default_height=0.5,
                controller=PositionHlCommander.CONTROLLER_MELLINGER) as pc:
            # Go to a coordinate
            pc.go_to(1.0, 1.0, 1.0)

            # Move relative to the current position
            pc.right(1.0)

            # Go to a coordinate and use default height
            pc.go_to(0.0, 0.0)

            # Go slowly to a coordinate
            pc.go_to(1.0, 1.0, velocity=0.2)

            # Set new default velocity and height
            pc.set_default_velocity(0.3)
            pc.set_default_height(1.0)
            pc.go_to(0.0, 0.0)


def simple_sequence():
    with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
        with PositionHlCommander(scf) as pc:
            pc.forward(1.0)
            pc.left(1.0)
            pc.back(1.0)
            pc.go_to(0.0, 0.0, 1.0)


if __name__ == '__main__':
    cflib.crtp.init_drivers(enable_debug_driver=False)

    simple_sequence()
    # slightly_more_complex_usage()
```

Ну, с основой функцией (которая псевдо-main) надеюсь понятно - инициализация драйверов и запуск одной из функций, объявленных выше (можно раскомментировать). Нужно ли напоминать про защиту от включения (см. предыдущий параграф)?
Пройдемся по включениям модулей. `crtp` - нужен для инициализации драйверов. Следующие два класса - основные для работы с коптерами. Первый `Crazyflie` - стандартный, асинхронный (вызванные методы не ждут, например, получения данных, а возвращаются сразу, а уже когда событие происходит, вызывается коллбек, назначенный на это действие). Второй `SyncCrazyflie` реализует синхронную обертку над классом `Crazyflie`, и предоставляет “блокирующие методы”, которые возвращаются только по окончании исполнения. Это удобно для создания простеньких полетных скриптов. `PositionHlCommander` - класс для отправки полетных точек.
Разберем теперь более общую функцию `slightly_more_complex_usage()`. При помощи оператора контекста создается объект класса для сихронной работы с коптерами. В качестве параметров ему передаются:

- URI - идентификатор коптера, строка вида `'radio://0/80/2M/E7E7E7E7E7'`;
- объект основного (асинхронного) класса `CrazyFlie`. Ему в качестве параметра `rw_cache='./cache'` передается директория, в которую будет записываться кеш исполнения

скрипта (телеметрия и ошибки, если таковые появятся). Данная запись означает, что в директории, где запущен скрипт создастся файлик с именем *cache*, куда будет записываться кеш.

Далее, опять же с помощью оператора контекста создается объект класса `PositionHlCommander`. После успешного создания объекта этого класса, коптер взлетит на заданную параметрами высоту. Если явно высоту мы не задали, то он взлетит на полметра, это задано по умолчанию в библиотеке. Для создания ему передаем следующие параметры:

- объект синхронного класса `SyncCrazyflie`;
- начальные координаты коптера (x, y, z);
- `default_velocity` - скорость движения по умолчанию;
- `default_height` - высоту по умолчанию;
- `controller` - режим полетного контроллера `CONTROLLER_PID` или `CONTROLLER_MELLINGER` (инфы о разнице режимов пока найти не удалось). Запись через точку `PositionHlCommander.CONTROLLER_MELLINGER` необходима, т.к. мы обращаемся к константе объявленной в соответствующем модуле.

Названия методов говорят сами за себя. Координаты передаются в метрах. Все методы можно посмотреть перейдя в файл, в котором и содержится описание класса `PositionHlCommander`. Определенно стоит попробовать сделать это. Самый простой способ - открыть пример *position_commander_demo.py* нажать на имя класса -> Go to Definition. Если у вас установлена библиотека *cflib*, то откроется файлик с классом.

Основное отличие второй функции `simple_sequence()` в том, что при создании объектов при помощи менеджеров контекста, ему не передаются дополнительные параметры. Нужно помнить, что при этом они будут заданы исходя из параметров по умолчанию. Для того чтобы посмотреть, какие они, необходимо открыть соответствующий модуль библиотеки. Координаты коптера по умолчанию равняются 0. Т.е. если вы запускаете коптер из центра, а позиции якорей прописаны так, что один из углов комнаты - точка с нулевыми координатами, после взлета коптер полетит к точке с нулевыми координатами. При вызове функции `slightly_more_complex_usage()` почему то начинают лететь ошибки и коптер нормально не взлетает.

Разбор работы с классом Commander (positioning/initial_position.py)

Разбор работы с классом Swarm (swarm/hl-commander-swarm.py)