МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ОСНОВАМ ПРОГРАММИРОВАНИЯ И ТЕЛЕУПРАВЛЕНИЯ БВС

25.02.08 Эксплуатация беспилотных авиационных систем

технический профиль

ДЛЯ ПЕДАГОГИЧЕСКИХ РАБОТНИКОВ И СТУДЕНТОВ ОЧНОЙ ФОРМЫ ОБУЧЕНИЯ

Самара, 2022

Введение

Данное учебное пособие представляет собой второй том практического руководства для педагога дополнительного образования и позволяет разработать адаптивную программу для учащихся 10-11 классов. В многообразии и сложности охватываемых тем авторами учебного пособия был собран и переработан в доступную для осмысления форму отечественный и зарубежный опыт, накопленный инженерами, электронщиками и просто энтузиастами в области построения беспилотных летательных аппаратов. В повседневную действительность год от года вливаются новые технологии, формируются новые профессии и компетенции и для подготовки специалистов в области перспективной отрасли необходимо внедрять инновационные методики в систему дополнительного образования. Авторы приложили усилия для того, чтобы каждый увлеченный идеей учащийся сумел овладеть знаниями и постичь захватывающий мир беспилотных технологий, даже не имея за спиной опыта обращения с мультимоторными аппаратами.

Учебное пособие содержит тематические разделы, которые рекомендуется комбинировать в зависимости от скорости усвоения материала группой. Педагог в зависимости от образовательного учреждения может создать углубленный курс, направленный на предпрофильную подготовку специалистов в областях: прикладного программирования C++, схемотехники, оператора БПЛА, конструктора БПЛА.

Оглавление

Раздел 1. Основные понятия и инструменты.

Теория и практика построения мультироторных БПЛА изложена в 1 томе данного учебного пособия. Второй том посвящён созданию интеллектуальных автономных беспилотных систем. Учебные задачи изложены по принципу «от простого к сложному». В данном разделе описан набор инструментов, используемых для создания автономных систем.

ROS (Robot operating system) — набор библиотек и инструментов для создания сложных и надёжных робототехнических систем различного назначения и на различных платформах. Основной сайт – www.ros.org.

Linux — семейство Unix-подобных операционных систем, построенных на базе ядра Linux. Как ядро Линукс, так и программы на его основе распространяются, как правило, по модели Свободного программного обеспечения.

Raspberry PI — компактный одноплатный компьютер, применяемый в широком спектре проектов построения робототехнических устройств. Основной сайт сообщества - raspberrypi.org.

Mavlink — (Micro Air Vehicle Link) протокол обмена сообщениями между наземной станцией (Ground Control Station, а также её компонентами) и малыми беспилотными аппаратами (как летающими, так и ездящими, плавающими и т.д.).

Mavros — пакет системы ROS, обеспечивающий связь между элементами (т.н. Нодами) системы ROS по протоколу Mavlink, а также канал связи с наземной станцией. Описание - http://wiki.ros.org/mavros

Dronekit — пакет программ для разработки приложений для бортового компьютера. Проект распространяется по модели СПО. Основной сайт - dronekit.io.

OpenCV — (Open Source Computer Vision Library) – набор библиотек компьютерного зрения и обработки изображений с открытым исходным кодом. Основной сайт - opencv.org.

Полётный контроллер – электронная плата, управляющая полётом беспилотного летательного аппарата. В УМК Жужа в основном используются полётные контроллеры Mini-APM и Pixhawk.

Бортовой компьютер — дополнительный вычислительный модуль для управления полётом БПЛА (управление полётным контроллером, ориентация в пространстве по изображению с камеры, управление автономным полётом). В УМК ЖУЖА в качестве бортового компьютера используется Raspberry PI.

Установка дистрибутива Линукс

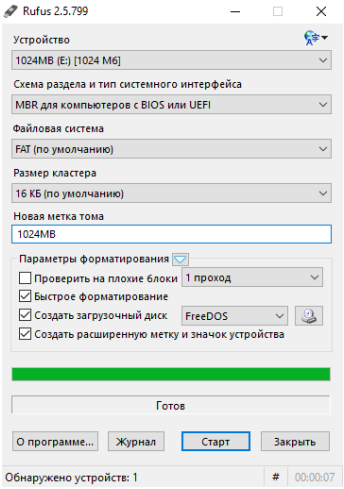
Для программирования автономных полётов дрона используется полётный контроллер Pixhawk совместно с бортовым компьютером Raspberry PI. Прежде программирования автономных полётов на Raspberry нужно установить дистрибутив операционной системы и библиотеки.

Образ дистрибутива операционной системы можно скачать с официальной страницы сообщества: https://www.raspberrypi.org/downloads/

Официальным дистрибутивом сообщества является Raspbian. Наиболее удобный способ установки – запись образа операционной системы на SD карту.

Желательно использовать SD карту не ниже 10-го класса скорости, объёмом от 8 Гб.

Для записи образа операционной системы на SD карту можно воспользоваться бесплатной утилитой Rufus (https://rufus.akeo.ie/?locale=ru_RU).



Обычный VGA монитор к Raspberry можно подключить с помощью переходника HDMI-VGA.

SD карту с записанным образом операционной системы нужно установить в Raspberry PI, подключить к микрокомпьютеру монитор и питание, после чего произойдёт первый запуск операционной системы.

После запуска микрокомпьютер нужно подключить к интернет (с помощью LAN кабеля или по WiFi) и выполнить обновление операционной системы с помощью команд:

```
sudo apt-get update
sudo apt-get upgrade
```

Также можно обновить прошивку контроллера:

Sudo rpi-update

После установки обновлений микрокомпьютер нужно перезапустить:

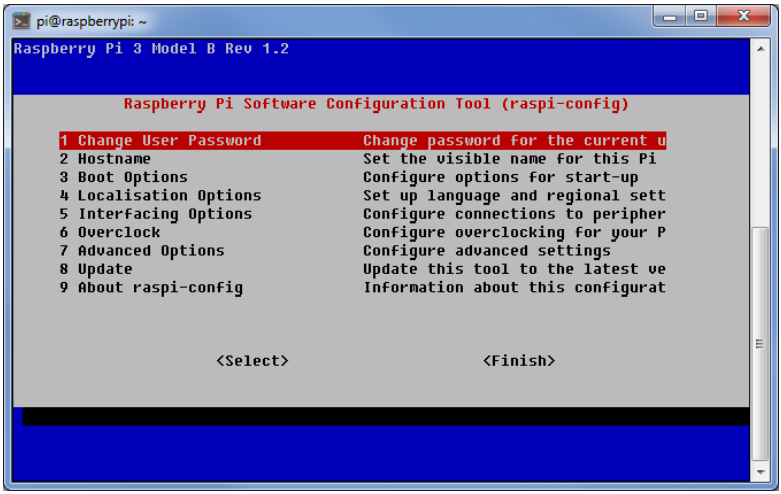
Reboot

После перезагрузки система должна уведомить об обновлении доактуальной версии.

Настройка внешнего доступа через SSH, сети WiFi и камеры

Настройка основных параметров Raspberry PI осуществляется с помощью утилиты raspi-config:

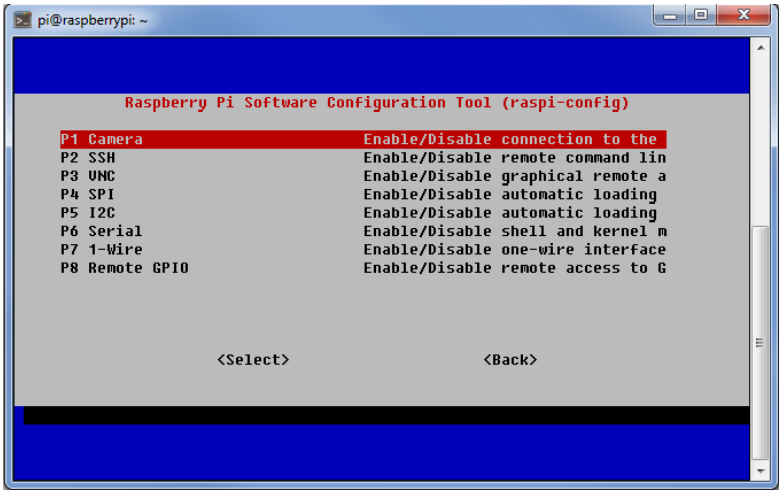
```
sudo raspi-config
```



Данная утилита позволяет настройить основные параметры компьютера Raspberry PI.

В целях безопасности рекомендуется сменить пароль пользователя pi (через пункт меню Change User Password).

Доступ к камере и доступ извне по SSH можно настроить через пункт «5 Interfacing Options»:



На вопрос «Would you like <Camera или SSH> to be enabled?» нужно ответить Yes.

Настройка WiFi адаптера:

Беспилотное летательное средство может эксплуатироваться как в помещении, так и на открытом воздухе, вне зоны действия проводных/беспроводных компьютерных сетей. Поэтому рекомендуется настраивать отдельную точку доступа по WiFi на бортовом компьютере, чтобы связь аппаратуры управления с дроном не зависела от внешних обстоятельств. Для настройки wifi адаптера в режиме точки доступа нужно выполнить следующие шаги:

1. Настроить службу wpa_supplicant на работу в режиме точки доступа:

Для этого выполнить команду:

```
> sudo nano
/etc/wpa_supplicant/wpa_supplicant.conf
```

В открывшемся окне записать следующие строки:

```
country=GB
ctrl_interface=DIR=/var/run/wpa_supplicant
GROUP=netdev
update_config=1

network={
      ssid="ZuzaDebian"
      psk="zuzazuza"
      mode=2
      key_mgmt=WPA-PSK
      pairwise=CCMP
      frequency=2437
}
```

Параметр ssid содержит имя точки доступа, а psk – пароль.

Далее нажать Ctrl+O и подтвердить запись в файл нажатием Enter. Для выхода из редактора нажать Ctrl+X.

2. Далее требуется включить статический IP адрес (в нашем примере – 192.168.11.1, но можно и другой) на беспроводном интерфейсе wlan0:

```
>  sudo nano /etc/dhcpcd.conf
```

В открывшемся файле после всех строк внизу добавить:

```
interface wlan0
static ip_address=192.168.11.1/24
```

Нажать Ctrl+O, Enter для подтверждения и Ctrl+X для выхода обратно в терминал.

3. Установить и настроить isc-dhcp-server:

Внимание! Для следующей операции требуется подключить Raspberry Pi к сети Интернет (например, с помощью через Ethernet кабеля).

> sudo apt install isc-dhcp-server
> cd /etc/network/if-up.d
> sudo touch isc-dhcp-server
> sudo nano isc-dhcp-server
В открывшемся окне прописать следующие строки:

```
#!/bin/sh
if [ "$IFACE" = "--all" ];
then sleep 10 && systemctl start isc-dhcp-
server.service &
fi
```

Нажать Ctrl+O, Enter для подтверждения и Ctrl+X для выхода обратно в терминал.

4. Перезагрузить Raspberry Pi:

```
> sudo shutdown -r now
```

В результате на Raspberry PI будет настроена точка доступа ZuzaDebian, пароль досупа = zuzazuza, ip=адрес при подключении = 192.168.11.1. Дальнейшую настройку микрокомпьютера можно производить, подключившись через беспроводное соединение по протоколу ssh.

Настройка рабочей станции для симулятора полётов/мониторинга дрона

Бортовой компьютер во время полёта находится на дроне, проводная связь с ним не возможна. Для полноценной настройки/симуляции автономных

полётов дрона нужно использовать рабочую станцию, со следующими приложениями:

ПО для наземной станции управления дроном (QGroundControl, Mission Planner).

ПО для симуляции полётов (jmavsim, gazebo)

ПО для мониторинга параметров дрона (координаты маркеров, дрона, изображения с камеры) – Rviz.

Также рабочую станцию можно использовать для учебной разработки/тестирования алгоритмов компьютерного зрения на OpenCV.

Для рабочей станции рекомендуем использовать Linux Ubuntu – на ней можно установить ROS той же версии, что и на дроне (Ubuntu 16.04 LTS на момент написания учебного пособия). Скачать дистрибутив можно на странице сообщества http://ubuntu.ru/get

Если в качестве основного рабочего компьютера/ноутбука используется Windows – можно настроить сетевой доступ к оконной среде Linux Ubuntu по протоколу VNC:

```
sudo apt-get install vino
```

Для настройки параметров доступа к удалённому десктопу нужно выполнить:

```
vino-preferences
```

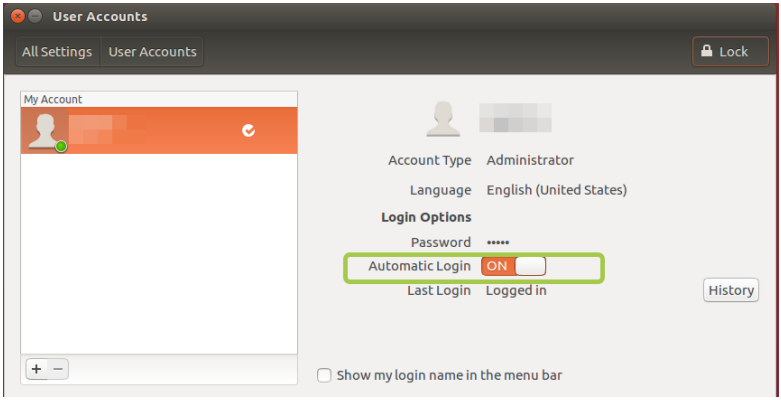
Чтобы сервер vino стартовал при запуске компьютера – нужно добавить команду его запуска в перечень запускаемых приложений: Система – Параметры – Персональные – Запускаемые приложения:

```
/usr/lib/vino/vino-server
```

Также для vnc-сервера нужно отключить шифрование:

```
dconf write /org/gnome/desktop/remote-
access/require-encryption false
```

Чтобы не вводить пароль каждый раз при входе – можно настроить автоматический вход пользователя в систему пир загрузке ОС, с помощью соответствующего переключателя в настройках профиля пользователя:



Для удалённого доступа к десктопу можно использовать RealVNC Viewer (realvnc.com).

Настройка подключения к рабочему столу:

Симулятор дрона jmavsim

Перед тем как программировать полёт реальных коптеров – более безопасно писать и тестировать код на виртуальных коптерах. jMAVSim является простым симулятором квадрокоптера, который летает под управлениея прошивки PX4 в симулированном виртуальном мире. Симулятор прост в установке и позволяет писать/тестировать программы автономного полёта: взлёт, полёт, посадка. Оригинальное описание симулятора приведено на странице https://dev.px4.io/en/simulation/jmavsim.html

Установка симулятора производится в процессе установки инструментов разработчика PX4 с помощью готовых скриптов. Процесс установки описан на странице https://dev.px4.io/en/setup/dev_env_linux.html

1. Включаем текущего пользователя в группу «dialout»:
   ```
   sudo usermod -a -G dialout $USER
   ```

2. Осуществить выход и вход в систему (чтобы изменения применились)

Далее следует скачать и запустить скрипт ubuntu_sim_nuttx.sh по ссылке https://raw.githubusercontent.com/PX4/Devguide/master/build_scripts/ubuntu_sim_nuttx.sh

```
source ubuntu_sim_nuttx.sh
```

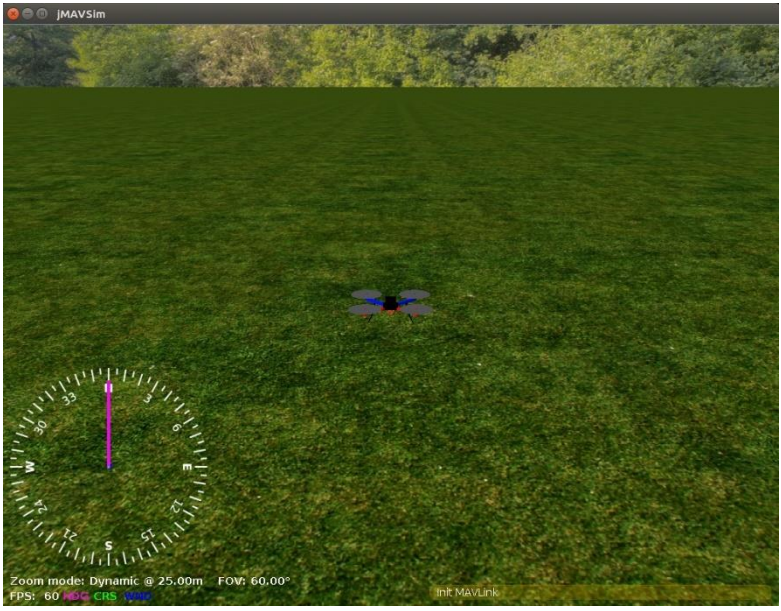
На задаваемые скриптом вопросы – согласиться (Yes). После успешного выполнения скрипта перезагрузить компьютер.

Также можно поставить программу наземной станции QGroundControl по ссылке на странице https://docs.qgroundcontrol.com/en/releases/daily_builds.html

Запуск симулятора производится командой:

```
cd src/Firmware/
make posix_sitl_default jmavsim
```

После выполнения этой команды в окне терминала отобразится командный интерфейс симулятора, а также окно с 3D-визуализацией симулятора дрона.



Раздел 2. Среда ROS (Robot Operating System)

ROS (Robot Operating System) обеспечивает разработчиков библиотеками и инструментами для создания приложений робототехники. ROS обеспечивает аппаратную абстракцию, предлагает драйверы устройств, библиотеки, визуализаторы, обмен сообщениями, менеджеры пакетов и многое другое. ROS выпускается в соответствии с условиями BSD лицензии и с открытым исходным кодом. Описание системы ROS - http://wiki.ros.org/ru

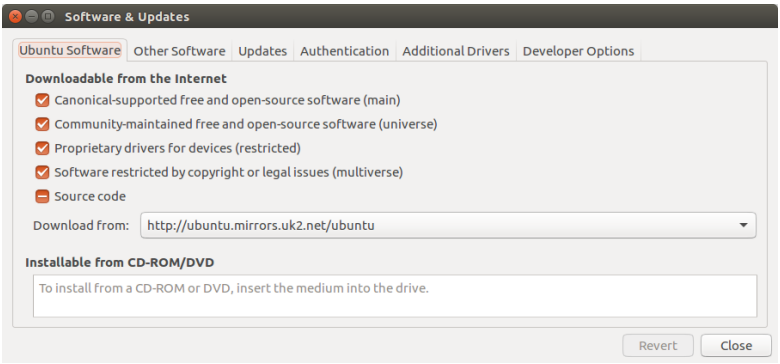
Установка системы ROS на компьютере

Одновременнно сообществом поддерживается несколько дистрибутивов системы ROS, некоторые из которых являются более стабильными (LTS – long term support, пакеты с длительным сроком поддержки), другие – с более коротким сроком поддержки, но подходящие для более новых платформ и с более новыми версиями пакетов. На английском языке процесс установки описан на странице http://wiki.ros.org/ROS/Installation . На платформе copter.space на момент написания учебного пособия используется дистрибутив ROS Kinetic.

Ниже приведён порядок установки ROS Kinetic на Linux Ubuntu. Порядок действия для установки на прочие системы можно прочитать по ссылке http://wiki.ros.org/kinetic/Installation .

ROS Kinetic поддерживает только дистрибутивы Wily (Ubuntu 15.10), Xenial (Ubuntu 16.04) and Jessie (Debian 8).

1. Нужно разрешить загрузку из интернет репозиториев Ubuntu типа Restricted, Universe, Multiverse:



2. Добавить в источники ПО загрузку пакетов с сайта ros.org:

```
sudo sh -c 'echo "deb
http://packages.ros.org/ros/ubuntu $(lsb_release
-sc) main" > /etc/apt/sources.list.d/ros-
latest.list'
```

3. Добавить ключ для загрузки ПО

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-
keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

4. Обновить пакеты Ubuntu:

```
sudo apt-get update
```

5. Установка полного дистрибутива ROS:

```
sudo apt-get install ros-kinetic-desktop-full
```

6. Инициализация менеджера rosdep. Rosdep позволяет автоматически устанавливать системные зависимости для компиляции из исходного кода, а также требуется для некоторых ключевых компонентов ROS.

```
sudo rosdep init
rosdep update
```

7. Установка переменных окружения – обычно, удобно настроить автоматическое добавление переменных окружения ROS при старте сессии командной строки:

```
echo "source /opt/ros/kinetic/setup.bash" >>
~/.bashrc
source ~/.bashrc
```

8. Установка дополнительных инструментов для компиляциии пакетов. Для того, чтобы создавать и использовать собственные рабочие пространства ROS, требуется установаить некоторые дополнительные инструменты, которые распространяются отдельно:

```
sudo apt-get install python-rosinstall python-
rosinstall-generator python-wstool build-
essential
```

На этом установка ROS завершена. Протестировать установку можно с помощью учебных заданий, доступных по ссылке http://wiki.ros.org/ROS/Tutorials .

Основные концепции системы ROS

В ROS используются три уровня концепции: уровень файловой системы, уровень вычислительного графа, уровень сообщества ROS. Ниже приведено краткое описание этих концепций. Полное описание на английском языке находится по адресу http://wiki.ros.org/ROS/Concepts.

Дополнительно в ROS определены два типа имён: Имена ресурсов пакета (Package Resource Names) и Граф

Имён Ресурсов (Graph Resource Names) – которые описаны ниже.

Файловая система ROS

Концепция файловой системы ROS в основном относится к ресурсам ROS, находящимся на диске, таким как:

- **Пакеты (Packages)**: Пакет – основная единица организации программного обеспечения в ROS. В основном пакет содержит выполняемые процессы ROS (узлы или ноды - nodes), библиотеки на основе ROS, наборы данных, конфигурационные файлы и прочие полезные данные. Пакет – это минимальная единица для компиляции и релиза в ROS.
- **Метапакеты (Metapackages)**: Метапакеты – это специализированные Пакеты, используемые для группировки других взаимосвязанных Пакетов. Используются также для обратной совместимости.
- **Манифесты Пакетов (Package Manifests)**: Манифест (package.xml) содержит данные о пакете, включая имя, версию, описание, информацию о лицензировании, зависимости и прочую информацию, такую как экспортируемые пакеты.
- **Репозитории (Repositories):** набор пакетов, принадлежащих одной и той же версии. Пакеты одной и той же системы контроля версий и одинаковой версии могут выпускаться вместе с помощью инструмента автоматизации релизов bloom (http://wiki.ros.org/bloom). Репозиторий также может содержать только один пакет.
- **Типы сообщений (Message types):** описание сообщений, хранится в

my_package/msg/MyMessageType.msg, определяет структуры данных для сообщений, передаваемых в ROS.

- **Типы сервисов (Service types):** описание сервисов, хранится в my_package/srv/MyServiceType.srv, определяет структуры данных для запроса и ответа сервисов в ROS.

Граф вычислений ROS

Граф вычислений ROS – это одноранговая сеть процессов ROS, обрабатывающих данные. Основные понятия, связанные с графом вычислений – это ноды (nodes), Мастер (Master), Сервер Параметров (Parameter Server), сообщения (messages), сервисы (services), топики (topics), контейнеры (bags). Все эти сущности передают данные для вычислительного графа ROS разными способами.

- **Ноды (Nodes):** Ноды – это процессы, выполняющие вычисления. Системы использующие ROS состоят из модулей, система управления роботом включает в себя множество нод. Например, одна нода управляет лазерным дальномером, другая – моторами колёс, третья нода определяет положение в пространстве, четвёртая планирует траекторию движения, пятая предоставляет графическое представление системы, и т.д. ROS ноды разрабатывают с использованием клиентских библиотек ROS, таких как roscpp или rospy.
- **Мастер (Master):** Мастер-процесс ROS обеспечивает регистрацию имён и наблюдение за всем вычислительным графом. Без мастер-процесса ноды не смогли бы найти друг друга,

обмениваться сообщениями или вызывать сервисы.

- **Сервер параметров (Parameter Server):** Сервер параметров позволяет хранить данные с доступом по ключу в централизованном хранилище. В настоящее время Сервер Параметров является частью Мастера.
- **Сообщения (Messages):** Ноды обмениваются даннми посредством передачи Сообщений. Сообщение – это структура данных, состоящая из структурированных полей. Поддерживаются стандартные примитивные типы (integer, floating point, boolean, и т.д.), а также и массивы из них. Сообщение может содержать произвольные вложенные структуры и массивы (очень похоже на структуры в языке C).
- **Топики (Topics):** Сообщения передаются через транспортную систему через механизм публикации/подписки. Нода отправляет сообщение, *публикуя (publish)* его в определённом Топике. Топик – это имя, идентифицирующее содержание сообщения. Нода, заинтересованная в определённых данных, осуществляет *подписку (subscribe)* на соответствующий Топик. Для одного топика может существовать несколько параллельно публикующиъ/подписанных на него Нод, равно как и одна Нода может публиковать сообщения в и/или подписываться на несколько Топиков. В общес случае, публикаторы/подписчики не оказывают влияния друг на друга. Идея заключается в отделении производства информации от её использования. Логически Топик может быть представлен как строго типизированная шина сообщений. У каждой шины есть наименование, и любой элемент

может подсоединиться к шине для получения и отправки сообщений соответствующего типа.

- **Сервисы (Services):** Модель публикации/ подписки является очень гибкой, но её схема односторонней передачи сообщений «многие-многим» не подходит для взаимодействий типа «запрос-ответ», которые часто нужны в распределённой системе. Механизм «запрос-ответ» реализован через Сервисы. Сервис определяется парой структур сообщений – одна для запроса и одна для ответа. Нода предоставляет сервис, используя определённое Имя сервиса, клиент использует сервис, отправляя сообщение-запрос и ожидая ответа. Клиентские библиотеки ROS обычно представляют это взаимодействие для программиста в виде вызова удалённой процедуры.
- **Контейнеры (Bags):** Контейнеры предоставляют форматы для записи и воспроизведения потоков ROS-сообщений. Контейнеры являются важным механизмом для записи данных, например, данных с сенсоров, которые трудно собрать, но необходимо сохранять для разработки и тестирования алгоритмов.

Мастер ROS играет роль сервера имён в Вычислительном графе ROS. Он хранит информацию о Топиках и Сервисах для ROS-нод. Ноды сообщают Мастеру свою регистрационную информацию. В процессе коммуникации с Мастером Ноды могут получать информацию о других зарегистрированных Нодах, устанавливать с ними связь. Мастер также осуществляет обратные вызовы к Нодам, когда регистрационная информация меняется, что позволяет одам динамически устанавливать связи по мере запуска новых Нод.

Ноды связываются с другими Нодами напрямую. Мастер только предоставляет информацию для поиска, подобно DNS серверу. Ноды, которые подписываются на топик, запрашивают связь с Нодами, которые публикуют данные в этот топик, и устанавливают эту связь через соответсвующий согласованный протокол. Наиболее часто используемый протокол – TCPROS, он использует стандартные TCP/IP сокеты.

Такая архитектура обеспечивает раздельное функционирование системы, в которой посредством распределений имён могут быть построены большие сложные распределённые системы. Имена играют важнейшую роль в ROS: ноды, топики, сервисы и параметры все имеют имена. Каждая клиентская ROS-библиотека поддерживает переназначение (remapping) имён с помощью инструментов командной строки, т.е. скомпилированная программа может быть переконфигурирована во время выполнения для обработки различной топологии Вычислительного графа.

Например, для контроля лазерного дальномера мы запускаем драйвер hokuyo_node, который опрашивает лазер и публикует sensor_msgs/LaserScan сообщения в топик scan. Дла обработки этих данных, мы можем написать, используя пакет laser_filters, ноду, которая подписывается на сообщения топика scan. После подписки наш фильтр автоматически начнёт получать сообщения от лазера.

Теперь о том, как происходит разделение. Нода hokuyo_node только публикует сообщения, не зная о том, подписан ли на них кто либо. Нода фильтра только подписывается на сообщения топика, не зная о том, публикует ли кто-то туда сообщения. Обе ноды могут

быть запущены, остановлены, перезапущены, в любом порядке, не порождая при этом ошибок.

Далее мы можем добавить роботу другой лазерный дальномер, поэтому нужно переконфигурировать систему. Всё что нужно при этом – переназначить (remap) используемые имена. При запуске ноды hokuyo_node мы можем вместо имени scan переназначить имя base_scan. В то же самое сделать с нодой – фильтром. Теперь обе эти ноды будут связаны с использованием топика base_scan, и перестанут обрабатывать сообщение тописа scan. После этого мы можем запустить ещё одну ноду hokuyo_node для нового лазерного дальномера.

Сообщество разработчиков/пользователей ROS

Концепции сообщества ROS – это ресурсы ROS, которые позволяют различным сообществам обмениваться программами и знаниями. Эти ресурсы включают:

- **Дистрибутивы (Distributions):** Дистрибутивы ROS – это коллекции вертионированных стеков (stacks) ПО, которые вы можете установить. Дистрибутивы играют роль, аналогичную дистрибутивам Линукс: они облегчают установку наборов ПО, а также поддерживают совместимость версий в наборе ПО.
- **Репозитории (Repositories):** ROS опирается на распределённые репозитории кода, различные команды могут разрабатывать и выпускать свои собственные компоненты ПО для роботов.
- **ROS Wiki:** Wiki сообщества ROS – основной способ документирования информации о ROS. Любой желающиё может зарегисрироваться и выложить собственную документацию,

корректировки/обновления, писать учебные материалы, и т.д.

- **Списки рассылки (Mailing Lists):** Список рассылки пользователей ROS – это основной канал коммуникации об обновлениях ROS, а также форум, на котором можно задать вопросы о ПО ROS.

Граф Имён Ресурсов

Граф Имён Ресурсов – это иерархическая структура, используемая для всех ресурсов в вычислительном графе ROS, таких как Ноды, Параметры, Топики и Сервисы. Эти имена играют центральную роль в больших и сложных ROS-системах, поэтому важно понимать как эти имена работают и как ими управлять.

Перед тем как перейти к дальнейшему объяснению, приведём несколько примеров имён:

- / (глобальное пространство имён)
- /foo
- /stanford/robot/name
- /wg/node1

Граф Имён Ресурсов является важным инструментом ROS для обеспечения энкапсуляции (встраивания). Каждый ресурс определён в пространстве имён, которое может быть разделено со сногими другими ресурсами. В общем случае, ресурсы могут создавать другие ресурсы внутри своего пространства имён, а также могут иметь доступ к ресурсам внутри или вне своего пространства имён. Связи могут быть установленя между ресурсами в определённых пространствах имён, но в обще случае это делается с помощью интегрирующего кода над этими пространствами имён. Такая энкапсуляция позволяет изолировать различные части системы от случайного

неправильного захвата имён ресурсов и от глобального «похищения» имён.

Разрешение имён происходит относительно, поэтому ресурсу не нужно беспокоиться о том, в каком пространстве имён он находится. Это упрощает программирование, поскольку ноды, которые работают вместе, могут быть описаны как находящиеся в пространстве имён верхнего уровня. Когда эти ноды интегрируются в бОльшую систему, они могут быть опущены (pushed down) в пространство имён, соответствующее их набору кода. Например, мы можем взять демо-проект Stanford и демо-проект Willow Garage? И объединить их в новый демо-проект с субграфами Stanford и WG. Если в обоих проектах присутствует нода ‘camera’ – они не будут конфликтовать. Инструменты (например, визуализация графа), также как и параметры (например, demo_name), которые должны быть видны всему графу, могут быть созданы в нодах верхнего уровня.

Корректные имена

Корректные имена имеют следующие характеристики:

- Начинаются с алфавитного символа ([a-z | A-Z]), тильды (~) или прямого слэша (/).
- Последующие символы могут быть алфавитно-цифровыми ([0-9 | a-z | A-Z]), символы подчёркивания (_)или прямого слэша (/)

Исключение: базовые имена (описаны ниже) не могут иметь в своём составе слэши и тильды.

Разрешение имён

Существует четыре типа Графа Имён Ресурсов в ROS: базовый (base), относительный (relative), глобальный

(global), и приватный (private). Эти типы имён имеют следующий синтаксис:

- base - *базовый*
- relative/name - *относительный*
- /global/name – *глобальный*
- ~private/name – *приватный*

По умолчанию, разрешение имён осуществляется *относительно* пространства имён ноды. Например, нода /wg/node1 имеет пространство имён /wg, поэтому имя node2 разрешится как /wg/node2.

Имена без указания пространства имён – это *базовые* имена. Базовые имена на самом деле являются подклассом относительных имён, к ним применяются такие же правила разрешения. Базовые имена часто используются для инициализации имени ноды.

Имена, которые начинаются с символа «/» являются *глобальными*. Они считаются полностью разрешёнными (resolved). Глобальных имён следует, насколько возможно, избегать, поскольку они ограничивают взаимозаменяемость кода.

Имена, начинающиеся с тильды «~» - *приватные*. Они конвертируют имя ноды в пространство имён. Например, нода node1 в пространстве имён /wg/ будет иметь приватное пространство имён /wg/node1. Приватные имена используются для передачи параметров конкретной ноде через сервер параметров. Ниже приведено несколько примеров разрешения имён:

Node	**Relative** (default)	**Global**	**Private**
`/node1`	`bar` -> `/bar`	`/bar` -> `/bar`	`~bar` -> `/node1/bar`
`/wg/node2`	`bar` -> `/wg/bar`	`/bar` -> `/bar`	`~bar` -> `/wg/node2/bar`
`/wg/node3`	`foo/bar` -> `/wg/foo/bar`	`/foo/bar` -> `/foo/bar`	`~foo/bar` -> `/wg/node3/foo/bar`

Переназначение имён (remapping)

Любое имя внутри ROS ноды может быть переназначено, когда нода запускается из командной строки.

Имена ресурсов пакета

Имена ресурсов пакета используются в ROS в концепции уровня файловой системы для упрощения ссылок на файлы и данные на диске. Имена ресурсов пакета просты: они совпадают с названием Пакета, которому принадложит ресурс плюс имя самого ресурса. Например, имя "std_msgs/String" ссылается на тип сооющения "String" в Пакете "std_msgs".

Некоторые из файлов ROS, на которые можно ссылаться с помощью Имён ресурсов пакета ключают: Типы сообщений (Message types), Типы сервисов (Service types), Типы нод (Node types).

Имена ресурсов пакета похожи на пути к файлам, только они значительно короче. Поскольку ROS умеет располагать Пакеты на диске, и анализирует их содержание. Например, описания сообщений всегда хранятся в каталоге msg и имеют расширения .msg, поэтому std_msgs/String - это короткий путь к path/to/std_msgs/msg/String.msg. Аналогично, тип ноды foo/bar эквивалентен поиску файла с названием bar в Пакете foo с разрешением на исполнение.

Корректные имена

К Именам ресурсов пакета применяются жёсткие правила именования, поскольку они часто используются в коде, генерируемом автоматически. По этой причине Пакет ROS не может иметь в наименовании никаких спецсимволов кроме подчёркивания, также имена

должны начинаться с алфавитного символа. Корректное имя соответствует следующим характеристикам:

- Первый символ является алфавитным ([a-z | A-Z])
- Последующие символы – алфавитно-цифровые ([0-9 | a-z | A-Z]), подчёркивания (_) или прямой слэш (/)
- По крайней мере один прямой слэш ('/').

Инструменты командной строки, примеры обмена сообщениями ROS

Для работы в ROS в системе должен быть запущен мастер ROS. Мастер запускается командой `roscore`.

Перечень доступных нод выводится командой `rosnode list`.

Перечень доступных топиков выводится командой `rostopic list`.

Отправка тестового сообщения в тестовый топик 2 раза в секунду:

```
rostopic pub -r 2 /test std_msgs/String "data:
'Good bye'"
```

Отображать содержимое топика test:

```
rostopic echo /test
```

Публиковать/читать сообщения в/из одного и того же топика могут одновременно несколько сеансов.

Полный перечень упражнений для освоения ROS можно найти по ссылке http://wiki.ros.org/ROS/Tutorials .

Подключение к симулятору и управление виртуальным дроном с помощью ROS

Перед тем как переходить к управлению реальным дроном – рекомендуем освоить навыки написания программ пилотирования с помощью симулятора. Ниже рассмотрим управление виртуальным дроном с помощью симулятора jmavsim.

Настройка сетевого доступа в ROS

В общем случае, программа управления полётом запускается на компьютере, отдельном от полётного контроллера дрона. Например, в качестве полётного контроллера может выступать ПК Pixracer комплекта ЖУЖА VIS, а в качестве бортового компьютера использоваться Raspberry PI. Поэтому при симуляци дрона также рекомендуем запускать симулятор на отдельном компьютере, а управлять им с помощью Raspberry PI.

Подключение симулятора и управление дроном из командной строки

Запуск симулятора рассмотрен в разделе «Симулятор дрона jmavsim».

После после запуска симулатора на бортовом компьютере необходимо настроить сетевой доступ к ROS-мастеру компьютера симулятора. Делается это с помощью настройки переменных среды сеанса ROS_MASTER_URI и ROS_IP. В ROS_MASTER_URI нужно присвоить адрес мастера на врешнем компьютере. В ROS_IP – ip-адрес компьютера, например:

```
export ROS_MASTER_URI=http://10.42.0.1:11311

export ROS_IP=10.0.2.15
```

Обратите внимание, ROS_MASTER_URI необходимо задавать в виде строки с полным адресом (http…***), а ROS_IP – только ip-адрес.

Узнать ip-адрес компьютера можно с помощью команды `ifconfig`.

После подключения к симулятору по сети – на бортовом компьютере должен появиться списко тописков mavros, что можно проверить с помощью команды `rostopic list`. Также подключение мавлинк к полётному контроллеру удобно проверять с помощью команды `rostopic echo /mavros/state` – так называемый heartbeat («сердцедиение») полётного контроллера. В случае успешного подключение в поле Connected должно быть значение True.

Если подключение прошло успешно – можно осуществить запуск виртуального дрона из командной строки. Для этого нужно запустить несколько команд параллельно в разных терминалах:

1. Публикуем целевую позицию коптера «2 метра над землёй» в соответствующий топик, 5 раз в секунду (для автоподбора аргументов можно пользоваться клавишей TAB):

```
rostopic pub -r 5 /mavros/setpoint_position/local
geometry_msgs/PoseStamped "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
pose:
  position:
    x: 0.0
    y: 0.0
    z: 2.0
  orientation:
```

```
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0"
```

2. Переводим полётный контроллер в режим OFFBOARD (в отдельном терминале):

```
rosservice call /mavros/set_mode "base_mode: 0
custom_mode: 'OFFBOARD'"
```

3. Запускаем коптер (процедура запуска коптера по-английски называется «arm», поэтому в среде коптеристов бытует русская калька «армить»): `rosservice call /mavros/cmd/arming "value: true"`

После выполнения указанных действий виртуальный дрон должен взлететь на высоту 2 метра и зависнуть. Коптер перейдёт в режим LAND и приземлится, если мы прервём публикацию сообщений в топик setpoint_position в первом терминале.

В случае, если бортовой компьютер, с которого запускаются команды, подключен к симулятору дрона по сети – переменные среды должны быть установлены в каждом из терминалов. Для этого можно написать небольшой bash-скрипт, и вызывать его при начале работы терминала с помощью команды `source`.

Подключение к симулятору и управление дроном с помощью программы на Питоне

Управление дроном из командной строки Линукс возможно, но довольно трудоёмко. Для выполнения сложных миссий необходимо использовать средства программирования.

Наиболее удобным и простым средством программирования полётных миссий на момент написания учебного пособия является язык Python (по русски - Питон) – современный мультиплатформенный объектно-отриентированный язык программирования. Разработка языка осуществляется сообществом python.org.

В данном разделе мы приведём пример программы на Питоне для управления виртуальным коптером с помощью клавиш клавиатуры: вперёд/назад/влево/вправо, u-вверх, d-вниз, q-выход из программы.

Ниже приведён листинг программы:

```
#!/usr/bin/env python
import rospy
import mavros
import mavros.command as mc
from mavros_msgs.msg import State
from geometry_msgs.msg import PoseStamped
from mavros_msgs.srv import CommandBool
from mavros_msgs.srv import SetMode
current_state=State()
# keyboard manipulation
import curses
stdscr = curses.initscr()
curses.noecho()
stdscr.nodelay(1)
stdscr.keypad(1)

def state_callback(data):
        global current_state
        current_state=data
def setpoint_callback(data):
        pass

def main():
        rospy.init_node("offbrd",anonymous=True)
        rate=rospy.Rate(20)
        state=rospy.Subscriber("/mavros/state",State,state_callback)
        setpoint_sub=rospy.Subscriber("/mavros/setpoint_position/local",P
oseStamped,setpoint_callback)
```

```
	setpoint_pub=rospy.Publisher("/mavros/setpoint_position/local",Po
seStamped,queue_size=10)
	arming_s=rospy.ServiceProxy("/mavros/cmd/arming",CommandB
ool)
	set_mode=rospy.ServiceProxy("/mavros/set_mode",SetMode)
	setpt=PoseStamped()
	setpt.pose.position.x=0
	setpt.pose.position.y=0
	setpt.pose.position.z=2
	arming_s(True)
	for i in range (0,100):
		setpoint_pub.publish(setpt)
		rate.sleep()
	set_mode(0,"OFFBOARD")
	last_request=rospy.Time.now()
	cnt = 1
	while (rospy.is_shutdown()==False):
		if (current_state.mode!="OFFBOARD" and
(rospy.Time.now() - last_request > rospy.Duration(5.0))):
			print "OFFBOARD lost!!!!!!!!!!!!!!!!!!!!!!"

			if(set_mode(0,"OFFBOARD")==True):
				print("offboard enabled")
			last_request=rospy.Time.now()
		else:
			if (current_state.armed!=True and
(rospy.Time.now() - last_request > rospy.Duration(5.0))):
				if(mc.arming(True)):
					print ("armed")
				last_request=rospy.Time.now()
		cnt = cnt+1
		setpoint_pub.publish(setpt)
		rate.sleep()

		# keyboard		hcommands handling
		c = stdscr.getch()
		if c == ord('q'): break  # Exit the while()
		elif c == ord('u'): setpt.pose.position.z += 1
		elif c == ord('d'): setpt.pose.position.z -= 1
		elif c == curses.KEY_LEFT:setpt.pose.position.y += 1
		elif c == curses.KEY_RIGHT:setpt.pose.position.y -= 1
		elif c == curses.KEY_UP:setpt.pose.position.x += 1
		elif c == curses.KEY_DOWN:setpt.pose.position.x -= 1

		if  c!=curses.ERR: print setpt.pose.position

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

```
curses.endwin()
```

Текст программы нужно записать сохранить в виде текстового файла и запустить с помощью команды `python <имя файла>`.

После запуска программы виртуальный дрон зависнет на высоте 2 метра и дальше будет подчиняться командам оператора с клавиатуры.

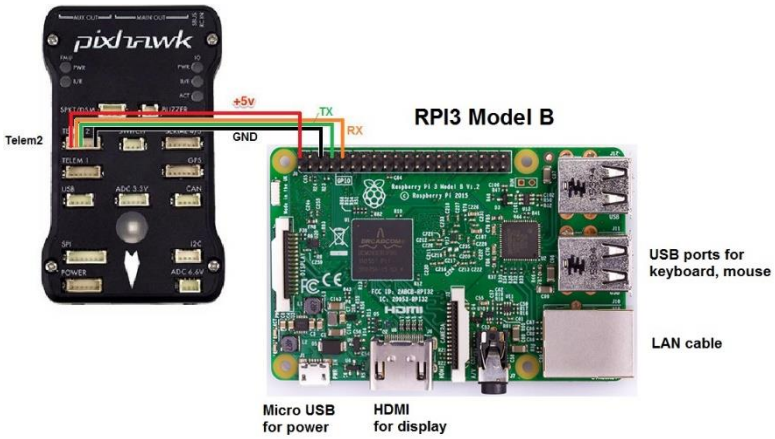
Подключение Raspberry PI к полётному контроллеру Pixhawk и управление реальным дроном с помощью ROS

Управление реальным дроном по протоколу mavros аналогично управлению виртуальным дроном, однако для реального дрона необходимо осуществить физическое подключение полётного контроллера к бортовому компьютеру (а также собрать и настроить сам дрон).

Схема подключения Raspberry PI к Pixhawk

Подключение бортового компьютера к полётному контроллеру возможно двумя способами: по интерфейсу UART через порт telem2, а также по интерфейсу USB. Ниже описаны оба способа.

Подключение UART осуществляется с помощью соединения понов GPIO Raspberry PI с портом telem2 полётного контроллера по следующей схеме:

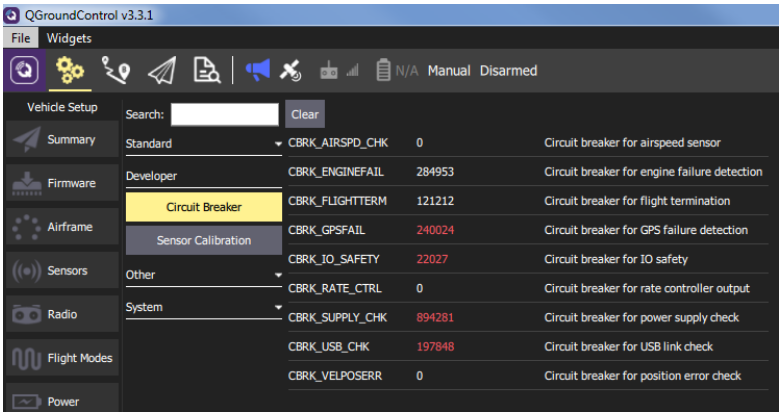


Telem2 #1 → GPIO Pin 02 (+5V)
Telem2 #2 → GPIO Pin 10 (GPIO15, RXD0)
Telem2 #3 → GPIO Pin 08 (GPIO14, RXD0)
Telem2 #6 → GPIO Pin 06 (Ground)

Подключение по mavros к полётному контроллеру выполняется с помощью команды:

```
roslaunch mavros px4.launch fcu_url:=
/dev/ttyAMA0:921600
```

При этом полётный контроллер дрона должен быть настроен и откалиброван, в программе управления QGroundControl не должно выдаваться ошибок, иначе полётный контроллер может не выдавать хартбит по mavros. Лишние проверки можно отключить с помощью специальных параметров-заглушек (circuit breakers) полётного контроллера:



Также в параметрах ПК должна быть правильно установлена скорость обмена по UART – параметр SYS_COMPANION = 921600 baud, 8N1.

Если процесс mavros выдаёт ошибку и падает при запуске – необходимо убедиться, что на Raspberry PI отключена блокировка UART операционной системой (с помощью raspi-config или в конфигурационном файле config.txt).

Автономный дрон может использоваться без радиопульта. Чтобы отключить проверку калибровки радиопульта, нужно установить параметр ПК COM_RC_IN_MODE = 1.

Альтернативный способ подключения ПК – с помощью USB – micro USB кабеля. Micro USB разъём подключается к ПК Pixhawk, USB – в свободный порт Raspberry PI.

Запуск mavros можно осуществлять командой `roslaunch mavros px4.launch`, без указания fcu_url, полётный контроллер должен быть найден при подключении по USB автоматически.

Остальные параметры подключения настраиваются аналогично подключения по telem2.

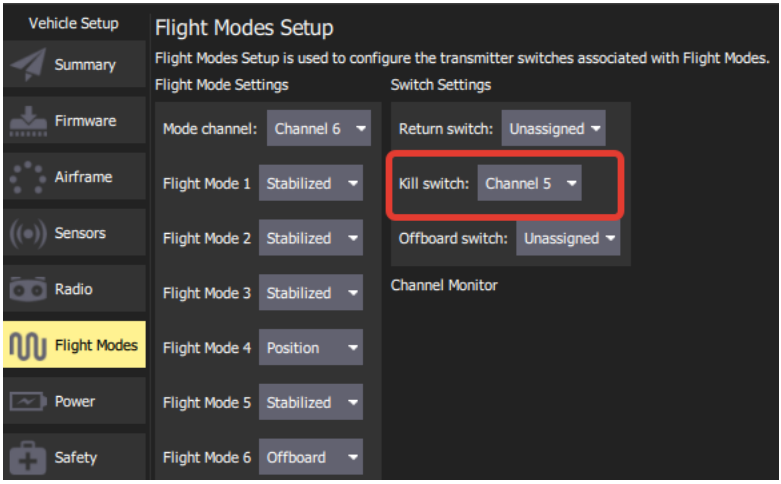
Минусом подключения по USB является то, что при необходимости прямой настройки ПК Pixhawk нужно переподключат micro-USB разъём полётного контроллера.

Проверка подключения, запуск/остановка моторов дрона

После подключения полётного контроллера к бортовому компьютеру можно проверить работу mavros с помощью арминга (запуска моторов) дрона.

Обязательно снимите пропеллеры при первом тестовом запуске моторов!

Также можно настроить экстренное отключение моторов (killswitch) на радиопульте:



Запуск моторов осуществляется с помощью команды `rosservice call /mavros/cmd/arming "value: true"`

Остановить моторы можно с помощью команды
`rosservice call /mavros/cmd/arming "value: false"`

Раздел 3. Основы компьютерного зрения и OpenCV

OpenCV (Open Source Computer Vision Library: http://opencv.org) – набор библиотек с открытым исходным кодом, распространяемый по лицензии BSD, в который включены сотни алгоритмов компьютерного зрения.

https://docs.opencv.org/3.4.1/d1/dfb/intro.html

Установка OpenCV на Raspberry PI

Установка библиотек OpenCV – из исходных кодов - длительный процесс, он поэтапно описан ниже в этой главе.

Первым делом нужно обновить стандартный набор пакетов и перезагрузить Raspberry:

```
sudo apt-get update
sudo apt-get upgrade
sudo rpi-update
sudo reboot
```

Установка инструментов для разработки:

```
sudo apt-get update
```

Установка инструментов для разработки:

```
sudo apt-get install build-essential git cmake
pkg-config
```

Установка пакетов для обработки изображений в формате JPEG, PNG, TIFF и инструментов обработки видеопотока:

```
sudo apt-get install libjpeg-dev libtiff5-dev
libjasper-dev libpng12-dev
sudo apt-get install libavcodec-dev libavformat-
dev libswscale-dev libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
```

Установка библиотеки GTK, для возможности отображения картинок на экране и построения GUI интерфейсов:

```
sudo apt-get install libgtk2.0-dev
```

Инструменты для ускорения вычислений (операций над матрицами и т.п.):

```
sudo apt-get install libatlas-base-dev gfortran
```

Заголовочные файлы для работы с языком Python:

```
sudo apt-get install python2.7-dev python3-dev
```

Далее нужно получить из интернет исходный код библиотек OpenCV (Raspberry PI должен быть подключен к Интернет для этого). В нашем примере мы получаем версию библиотек 3.0.0. По мере выхода новых релизов OpenCV номер 3.0.0 следует заменить на соответствующий:

```
cd ~
wget -O opencv.zip
https://github.com/Itseez/opencv/archive/3.0.0.z
ip
unzip opencv.zip
```

Для полной установки также можно установить дополнительные библиотеки OpenCV. Убедитесь, что версии самой OpenCV и библиотек совпадают (в нашем примере – 3.0.0):

```
wget -O opencv_contrib.zip
https://github.com/Itseez/opencv_contrib/archive
/3.0.0.zip
unzip opencv_contrib.zip
```

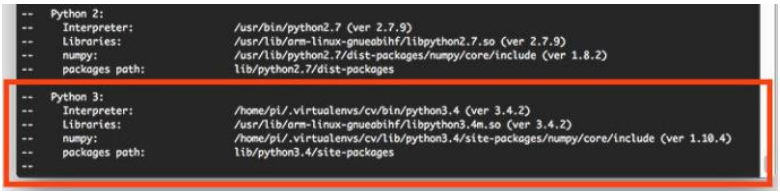
Установим менеджер пакетов для Python - pip:

```
wget https://bootstrap.pypa.io/get-pip.py
sudo python get-pip.py
```

Подготовка к сборке OpenCV из исходных кодов:

```
cd ~/opencv-3.0.0/
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE \
      -D CMAKE_INSTALL_PREFIX=/usr/local \
      -D INSTALL_C_EXAMPLES=ON \
      -D INSTALL_PYTHON_EXAMPLES=ON \
      -D
OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-
3.0.0/modules \
      -D BUILD_EXAMPLES=ON ..
```

Нужно проверить вывод команды cmake, для Python 2 и 3 он должен выглядеть примерно так:



Компилируем OpenCV:

```
make -j4
```

Ключ -j4 означает использовать при компиляции все 4 ядра. При компиляции Raspberry может зависнуть, если это происходит – можно компилировать, используя одно ядро. Это займёт больше времени, но позволит избежать ошибок, связанных с параллельной обработкой:

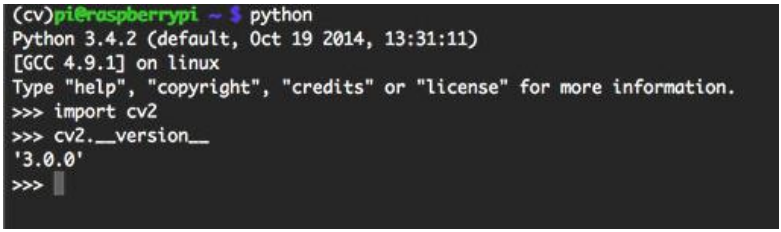
```
make clean
make
```

После успешного завершения компиляции – последний шаг – установка OpenCV:

```
sudo make install
sudo ldconfig
```

Если команды выполнились без ошибок – установка OpenCV завершена успешно. После установки можно проверить работу OpenCV с помощью Python:

```
python
>>> import cv2
>>> cv2.__version__
'3.0.0'
```



Если команда выдала версию OpenCV – установка завершена успешно и можно приступать к разработке программ под OpenCV на Python.

Основной используемый функционал OpenCV – библиотека Aruco

Оценка положения в пространстве – важная задача для приложений компьютерного зрения. Данный процесс основан на вычислении положения объекта в реальном мире на основании его 2D проекции. Обычно этот шаг довольно не прост, поэтому распространённой практикой является использование синтетических опорных (англ.fiducial – “фидуциальных”) маркеров, для упрощения алгоритма.

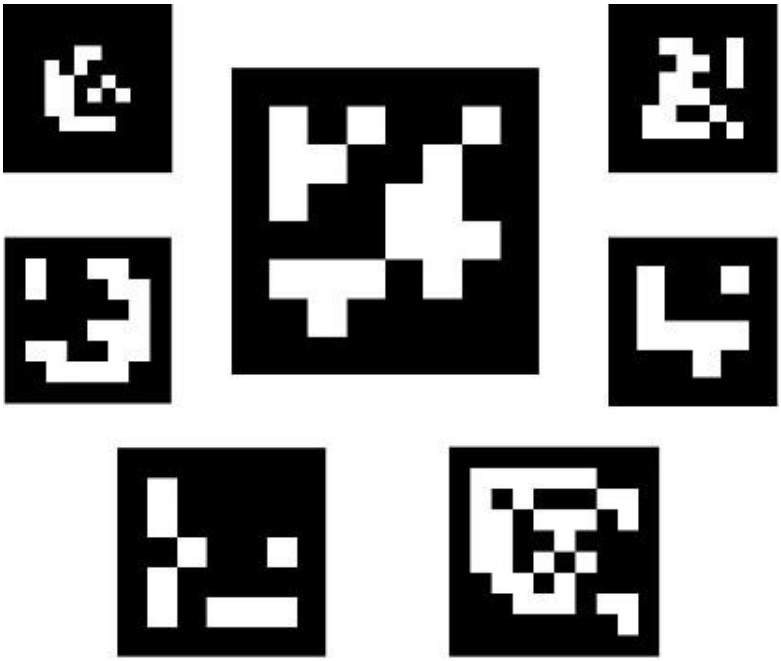
Один из наиболее популярных подходов – использование квадратных бинарных опорных маркеров. Основное преимущество таких маркеров – один маркер содержит достаточно информации (положение четырёх его углов) для определения положения камеры. Также, внутренняя бинарная

кодификация маркеров увеличивает надёжность их определения, позволяя применять алгоритмы поиска и коррекции ошибок.

Маркеры и словари

Aruco маркер – это квадрат, окаймлённый широкой чёрной границей, содержащий внутри бинарную квадратную матрицу, которая определяет его идентификатор (id). Черная граница обеспечивает его быстрый поиск на изображении, а бинарная кодификация – позволяет идентифицировать, с применение алгоритмов поиска и коррекции ошибок. Размер маркера определяет размер внутренней матрицы. Например, маркер размером 4ч4 содержит матрицу в 16 бит.

Ниже приведены примеры изображений Aruco маркеров:



Следует отметить, что маркер может быть повёрнут в пространстве под любым углом, однако, алгоритм поиска маркера определяет эти углы поворота, определяя отдельно каждый из углов маркера, основываясь на их бинарной кодификации.

Словарь маркеров – это набор маркеров, используемый в определённом приложении. Это просто перечень идентификаторов маркеров в соответствии с их бинарной кодификацией. Основные параметры словаря – размер словаря и размер маркера. Размер словаря – это количество маркеров, которые включены в словарь. Размер маркера – количество бит в матрице маркера. Модуль Aruco включает несколько предопределённых словарей с различными размерами и количеством маркеров.

Id маркера не является прямым переводом бинарной кодификации маркера в число, это просто порядковый номер маркера в словаре, к которому он принадлежит.

Генерировать изображения маркеров можно с помощью функции OpenCV DrawMarker(словарь, id маркера, размер картинки, изображение, размер границы).

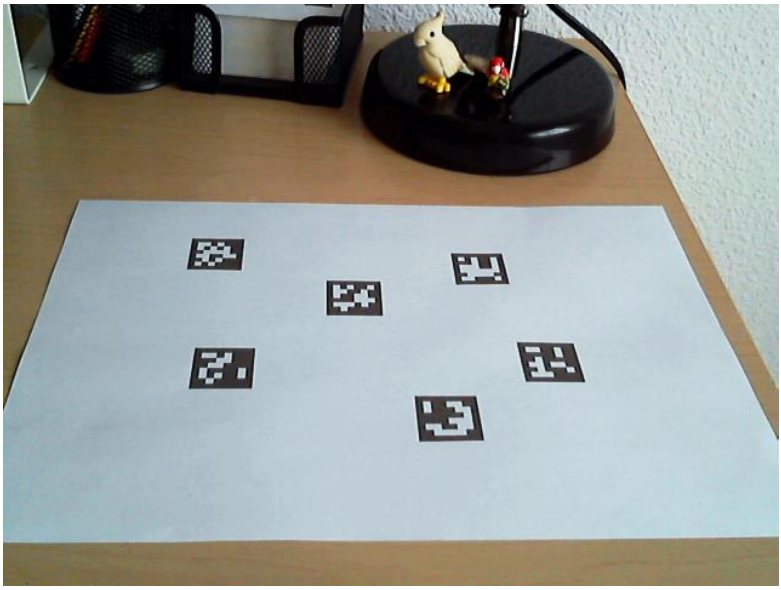
Поиска маркеров на изображении

Получив на входе изображение, содержащее Aruco маркеры, алгоритм поиска выдаёт перечень найденных маркеров. Каждый распознанный маркер характеризуется: положением его четырёх углов на картинке + id маркера.

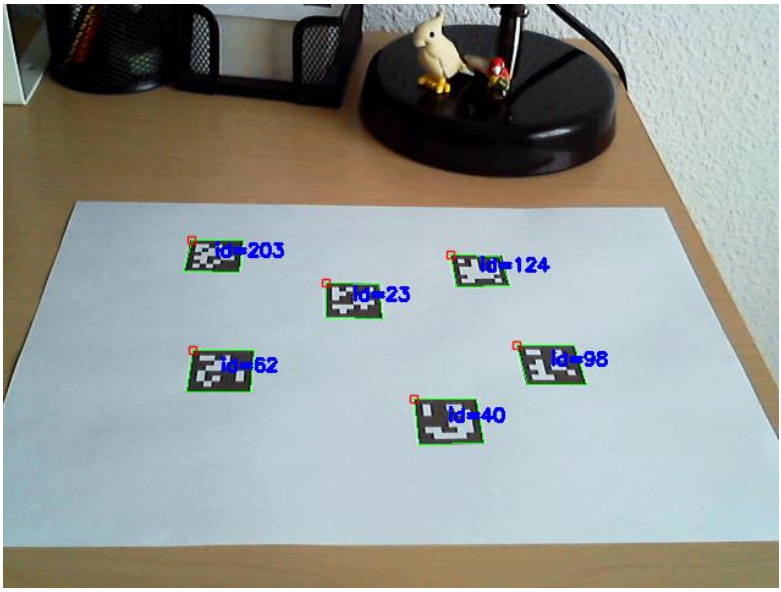
Процесс распознавания маркеров состоит из двух основных шагов:

1. Определение маркер-кандидатов. На изображении осуществляется поиск всех квадратных форм, которые могут быть маркерами.
2. После того как кандидаты определены – осуществляется анализ, являются ли они на самом деле маркерами, на основе их внутренней кодификации.

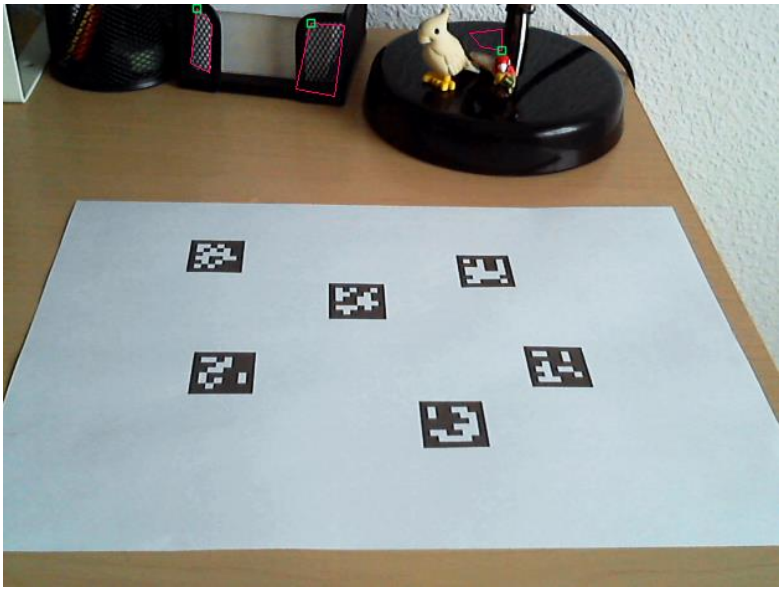
Например, при поиске маркеров на следующем изображении:



найдены следующие маркеры:



а следующие маркер-кандидаты были отклонены в ходе 2-го шага идентификации:



В aruco модуле поиск маркеров на изображении осуществляется с помощью функции detectMarkers(). Эта функция является самой важной в модуле, поскольку остальной функционал базируется на анализе маркеров, распознанных с помощью detectMarkers().

Определение положения в пространстве

После поиска маркеров на изображении – нам нужно определить положение камеры относительно маркеров.

Для того чтобы определить положение в пространстве – необходимо знать параметры калибровки Вашей камеры: матрицу камеры и коэффициенты искажения. Калиброка камеры может выполняться с помощью

функции calibrateCamera() OpenCV, а также с помощью модуля aruco, пример будет рассмотрен ниже. Калибровка выполняется один раз, если оптические параметры камеры не меняются. В результате калибровки мы получаем матрицу камеры (3x3 элемента фокальных расстояний и координат центра камеры) и коэффициенты искажения: вектор из 5 или более элементов, моделирующих искажение изображения камерой.

При определении положения Aruco маркеров в пространстве, положение каждого маркера может быть посчитано индивидуально. Также можно рассчитать одно положение относительно набора маркеров (т.н. Aruco Board).

Положение камеры относительно маркера – это 3D-трансформация из координатной системы маркера в координатную систему камеры. Положение (англ. Pose) определяется векторами ротации (вращения) и трансляции (передвижения). В Aruco модуле есть функция для оценки положения всех найденных маркеров estimatePoseSingleMarkers:

```
estimatePoseSingleMarkers(corners, 0.05,
cameraMatrix, distCoeffs, rvecs, tvecs)
```

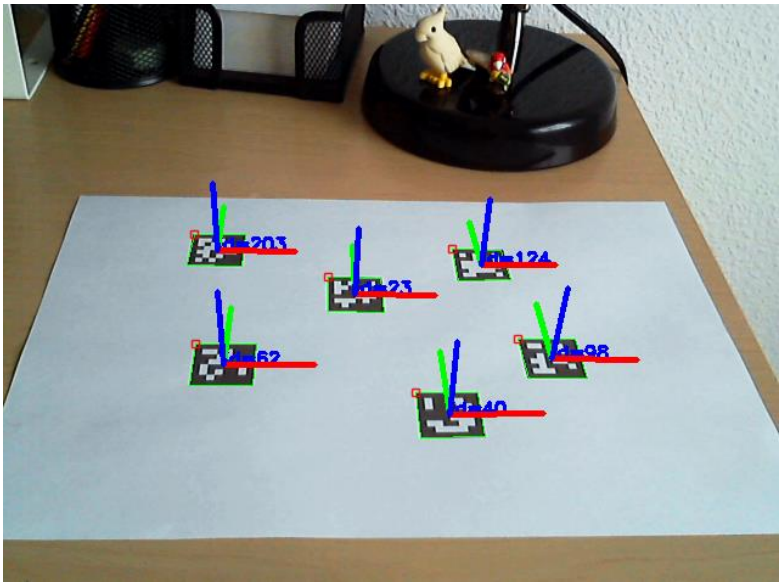
corners – набор координат углов маркеров, которые вернула функция detectMarkers()

второй параметр (0.05) – размер маркера в метрах или любых других единицах измерения. Положение камеры оценивается в этих же единицах.

cameraMatrix, distCoeffs – параметры калибровки камеры, известные заранее

rvecs, tvecs – векторы ротации и трансляции, соответственно, для каждого маркера в списке corners.

Координатная система маркера, используемая этой функцией: с началом координат в центре маркера, ось Z указывает вверх от перпендикулярно изображению маркера, X – вправо от маркера, Y – вверх от маркера:



В aruco модуле есть функция рисования осей на изображении, чтоб проверить правильность определения положения:

```
drawAxis(image, cameraMatrix, distCoeffs, rvec, tvec, 0.1)
```

image – изображение, на котором рисовать оси (обычно то же изоражение, на котором осуществляется поиск маркеров)

cameraMatrix, distCoeffs – параметры калибровки камеры

rvec, tvec – параметры положения в пространстве, для которого рисовать оси

последний параметр (0.1) – длина осей, в тех же единицах, что и tvec (обычно, метры).

Выбор словаря

Модуль aruco предоставляет класс Dictionary для описания словаря маркеров.

Кроме размера маркера и количества маркеров в словаре есть ещё один важный параметр словаря – дистанция между маркерами (inter-marker distance). Эта дистанция определяет возможности словаря для поиска и коррекции ошибок.

В общес случае, меньшее количество маркеров в словаре и больший размер маркера увеличивают дистанцию между маркерами. Однако, поиск маркеров мольших размеров более сложен, в силу большего количества маски бит маркера для поиска на изображении.

Например, если Вашему приложению нужно только 10 маркеров – лучше использовать словарь, состоящий из этих 10 маркеров, чем словарь из 1000 маркеров, т.к. в словаре из 10 маркеров дистанция между маркерами будет больше, что приведёт к более высокой устойчивости к ошибкам.

В aruco модуле есть несколько способов определения словаря, с помощью которых можно увеличить надёжность системы:

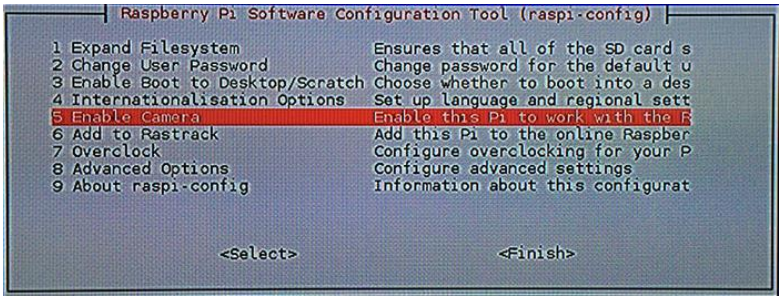
- Предопределённые словари – самый простой способ выбора словаря (функция getPredefinedDictionary);

- Автоматическа генерация словаря (функция generateCustomDictionary);
- Ручная генерация словаря (самый сложный способ, авторами не рассматривается).

Получение и обработка изображения с камеры Raspberry PI

Для получения и обработки изображений Raspberry с камеры PI предназначен модуль picamera.

На новой Raspberry PI первым делом необходимо включить камеру с помощью команды `sudo raspi-config`:



После включения камеры нужна перезагрузка Raspberry.

Перед написанием программ на Питоне рекомендуем проверить функциональность камеры с помощью инструментов командной строки. Иначе есть риск потери времени в попытках отладить программу, которая не работает из-за того, что некорректно работает сам модуль камеры.

Сделать фотографию можно с помощью команды `raspistill -o myphoto.jpg`. Данная команда активирует модуль камеры Raspberry PI, показывает окно предпросмотра, и через несколько секунд записывает

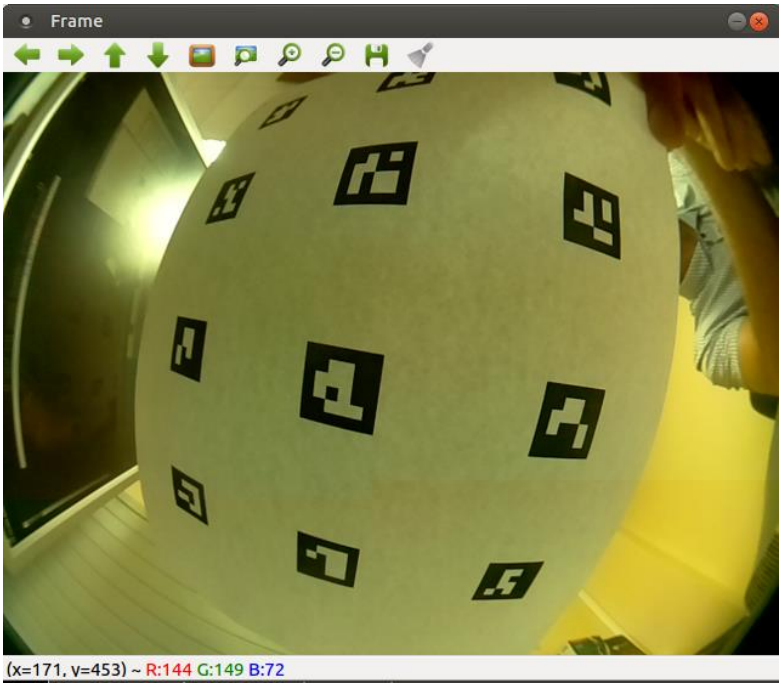
изображение в указанный файл (myphoto.jpg) в текущем рабочем каталоге.

Для использования камеры в программе на Питоне нужно выполнить установку модуля picamera с помощью команды `pip install "picamera[array]"`. Стандартный модуль picamera предоставляет методы для управления камерой, но нам также нужен опциональный подмодуль array для использования функций OpenCV. В Питоне OpenCV представляет изображения в виде массивов NumPy – подмодуль array позволяет получать NumPy массивы из камеры raspberry PI.

Рассмотрим пример программы test_image.py:

```
# coding=UTF-8
# импорт необходимых пакетов
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import cv2

# инициализация объекта для захвата картинки
camera = PiCamera()
rawCapture = PiRGBArray(camera)

# «прогрев» камеры
time.sleep(0.1)

# получаем изображение с камеры
camera.capture(rawCapture, format="bgr")
image = rawCapture.array

# отображаем картинку на экране и ждём нажатия любой клавиши
cv2.imshow("Image", image)
cv2.waitKey(0)
```

Код можно вписать в любой текстовый редактор, и выполнить с помощью команды `python test_image.py`. Если всё выполнено правильно – на экране отобразится изображение с камеры:

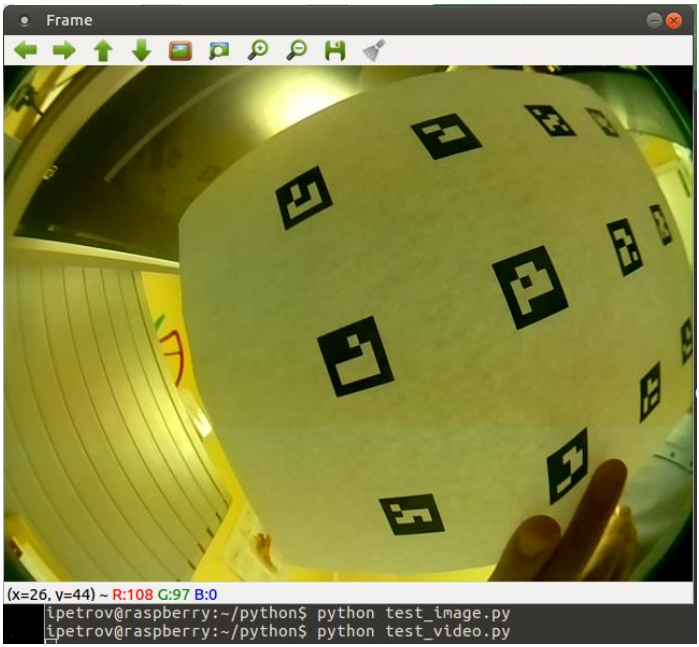


Теперь рассмотрим программу для обработки видеопотока с камеры.

```
# coding=UTF-8
# импорт необходимых пакетов
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import cv2

# инициализация объекта захвата
camera = PiCamera()
camera.resolution = (640, 480)
camera.framerate = 32
rawCapture = PiRGBArray(camera, size=(640, 480))

# «прогрев» камеры
time.sleep(0.1)

# получаем кадр от камеры
for frame in camera.capture_continuous(rawCapture,
format="bgr", use_video_port=True):
        # получаем кадр с камеры
```

```
image = frame.array

# отображаем кадр на экране
cv2.imshow("Frame", image)
key = cv2.waitKey(1) & 0xFF

# очищаем поток для следующего кадра
rawCapture.truncate(0)

# по клавише `q` выходим из цикла
if key == ord("q"):
        break
```

Код нужно вписать в файл test_video.py и запустить с помощью команды `python test_video.py`. На экране должен отобразиться видеопоток с камеры:



Публикация изображений камеры Raspberry PI через ROS

Для публикации изображений с использованием OpenCV в ROS предназначен пакет cv_camera (http://wiki.ros.org/cv_camera).

Данный пакет осуществляет обработку изображений с камеры с помощью объекта cv::VideoCapture библиотеки OpenCV.

Перед запуском модуля cv_camera необходимо запустить video for Linux – драйвер камеры Raspberry PI. Это можно сделать с помощью команды:

```
sudo modprobe bcm2835-v4l2
```

Либо добавить строку `bcm2835-v4l2` в файл /etc/modules.

Запуск пакета cv_camera возможен как из командной строки, так и в составе ROS пакета с помощью утилиты roslaunch.

Из командной строки пакет запускается командой rosrun. В командной строке можно также указать параметры запуска:

```
rosrun cv_camera cv_camera_node _rate:=10
_image_width:=320 _image_height:=240
```

Пакет cv_camera публикует следующие ROS-топики:

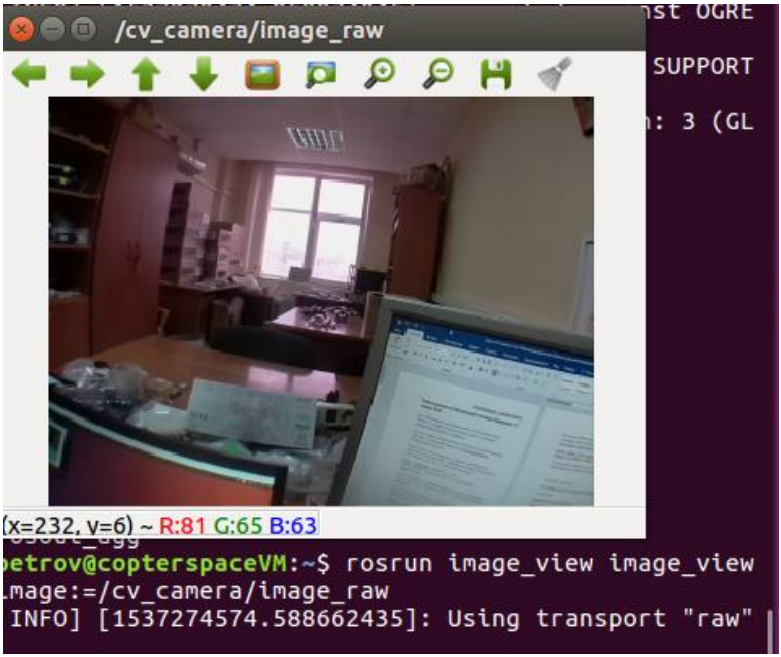
image_raw (тип sensor_msgs/Image) – изображение с камеры (кадры)

camera_info (тип sensor_msgs/CameraInfo) – информация о калибровке камеры

Просмотр публикуемых изображений осуществляется с помощью пакета image_view (http://wiki.ros.org/image_view):

```
rosrun image_view image_view
image:=/cv_camera/image_raw
```

После удачного запуска на экране отобразится окно с изображением с камеры Raspberry PI:



Запуск публикации изображений с камеры с помощью пакета ROS

Запуск пакетов ROS из командной строки не удобен по ряду причин: командную строку долго набирать с клавиатуры, в ней неудобно настраивать параметры, командная строка не запускается сама при старте системы. Поэтому далее мы выполним создание собственного учебного пакета ROS zuza, в рамках которого будем запускать автоматически публикацию

изображений с камеры, а также вести дальнейшие разработки для создания системы автономной навигации.

ROS пакет создаётся с помощью системы catkin – официальной системы компиляции/сборки пакетов ROS (на момент написания учебного пособия). В catkin используются макросы cmake и скрипты Python для реализации дополнительного функционала поверх системы cmake (https://ru.wikipedia.org/wiki/CMake). Работа catkin похожа на cmake, но в catkin добавлен автоматический поиск пакетов и возможность одновременной сборки нескольких взаимосвязанных проектов. Подробно концепция системы сборки catkin описана на странице http://wiki.ros.org/catkin/conceptual_overview .

Для создания catkin пакета нужно создать рабочую область в каталоге пользователя:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

Команда catkin_make – инструмент для работы с рабочими областями catkin (англ. *catkin workspaces*). После первого выполнения она создаёт ссылку на файл CMakeLists.txt в каталоге src, а также создаёт каталоги ‘build’ и ‘devel’. Внутри каталога devel находятся несколько предустановочных скриптов, которые настраивают системные переменные для нашей рабочей области. Нужно запустить соответствующий установочный файл с помощью команды source:

```
source devel/setup.bash
```

Чтобы проверить правильность инициализации переменных окружения установочным скриптом,

убедитесь, что переменная окружения ROS_PACKAGE_PATH включает в себя текущий каталог, в котором Вы находитесь:

```
echo $ROS_PACKAGE_PATH
/home/pi/catkin_ws/src:/opt/ros/kinetic/share
```

Далее мы переходим к созданию ROS пакета. Каждый catkin пакет должен соответствовать нескольким условиям:

- Содержать в себе файл с описанием пакета package.xml соответствующего формата;
- Содержать в себе файл CMakeLists.txt, который будет использовать система сборки catkin;
- Каждый пакет должен храниться в своём собственном каталоге.

Создание пакета производится с помощью команд:

```
cd ~/catkin_ws/src
catkin_create_pkg zuza std_msgs rospy roscpp
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

Данные команды создадут и соберут пустой catkin пакет в указанном каталоге (в нашем примере – catkin_ws).

Для того чтобы пакеты в catkin_ws были доступны сразу при запуске сеанса, нужно добавить строчку в файл .bashrc:

```
source /home/pi/catkin_ws/devel/setup.bash
```

Создадим launch-файл zuza.launch для старта пакета cv_camera в каталоге ~/catkin_ws/src/zuza/launch , следующего содержания:

```
<launch>
```

```
      <!-- main nodelet manager -->
    <node pkg="nodelet" type="nodelet"
name="nodelet_manager" args="manager" output="screen"
clear_params="true">
        <param name="num_worker_threads" value="4"/>
    </node>

      <!-- camera node - настройка публикации
изображения с помощью модуля cv_camera -->
      <node pkg="nodelet" type="nodelet"
name="main_camera" args="load
cv_camera/CvCameraNodelet nodelet_manager"
clear_params="true">
        <param name="frame_id"
value="main_camera_optical"/>
        <param name="camera_info_url"
value="file://$(find
zuza)/camera_info/fe130_320_01.yaml"/>
            <!-- setting camera FPS -->
        <param name="rate" value="100"/>
        <param name="cv_cap_prop_fps" value="40"/>
        <param name="image_width" value="640"/>
        <param name="image_height" value="480"/>
    </node>
</launch>
```

Данный файл описывает параметры запуска модулей nodelet (http://wiki.ros.org/nodelet) и cv_camera (http://wiki.ros.org/cv_camera) в режиме нодлета с указанными параметрами. Пакет nodelet является механизмом запуска разных алгоритмов внутри одного процесса с передачей массивов данных по ссылке (без копирования) между этими алгоритмами.

Запуск нашего пакета на выполнение осуществляется с помощью команды `roslaunch zuza zuza.launch`.

Проверить работу пакета можно с помощью утилит командной строки (rostopic list и прочих), а также с помощью программы image_view:

```
rosrun image_view image_view
image:=/main_camera/image_raw
```

Подробно про создание и запуск ROS пакетов можно прочитать по ссылке:
http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage

После успешного запуска работы камеры через ROS можно приступать к калибровке камеры.

Калибровка камеры

Система ROS содержит специальный пакет, предназначенный для калибровки камеры. Подробнее о нём можно прочитать по ссылке http://wiki.ros.org/camera_calibration . Ниже мы рассмотрим последовательность действий для калибровки монокулярной камеры.

Перед началом калибровки камеры необходимо подготовить шаблон чёрно-белой плоской «шахматной доски» с известным размером квадратов. В нашем примере используется шахматная доска 9x7 квадратов размером 108 мм. Также нужно обеспечить публикацию изображений с камеры через ROS.

Установите программу осуществляется с помощью команды: `rosdep install camera_calibration`

Убедитесь, что камера публикует изображения через ROS с помощью команды `rostopic list`. По умолчанию большинство камер ROS предоставляют топики `/camera/camera_info` и `/camera/image_raw`.

Для начала процесса калибровки в параметрах нужно указать топики камеры и размеры шахматной доски. Калибровка запускается следующей командой:

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/camera/image_raw camera:=/camera
```

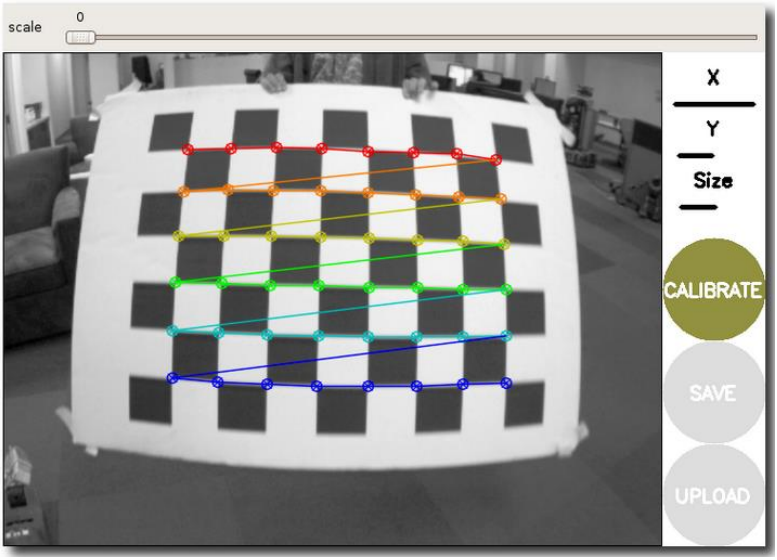
Данная команда открывает окно калибровки, в котором обозначаются распознанные углы шахматной доски:

Если окно не открылось – попробуйте в команду запуска калибровки добавить параметр `--no-service-check`
Если углы шахматной доски не обозначены цветными квадратами – убедитесь, что **количество внутренних углов** в шахматной доске соответствует размеру, указанному в параметрах команды.

Чтобы получить правильные результаты калибровки – нужно перемеofnm шахматную доску по всем углам камеры: вверх, вниз, вправо, влево, ближе/дальше от камеры:

После каждого перемещения подержите доску на месте, пока не отобразится подсветка углов в окне калибровки. По мере перемещения три индикатора отображают накопление необходимого объёма данных для калибровки: X – перемещений вправо-влево, Y – перемещений вверх/вниз, Size – перемещений ближе/дальше от камеры.

Когда загорится кнопка Calibrate – это означает, что накоплено достаточно данных для калибровки, Вы можете нажать Calibrate и посмотреть результат. Калибровка может занять около минуты. Окно может стать серым, но нужно подождать, пока выполнится калибровка.



По завершении калибровки вы увидите её результаты в окне терминала и откалиброванное изображение в окне калибровки:

В результате успешной калибровки на откорректированном изображении все прямые линии реального мира должны также стать прямыми. В результате плохой калибровки обычно получаются пустые или кривые изображения, или изображения с непрямыми линиями.

После успешной калибровки вы видите слайдер вверху окна калибровки для изменения размера ректифицированного изображения. Scale = 0 означает что изображение отображено таким образом, что отражаются все точки ректифицированного изображения. У ректифицированного изображения нет чёрного бордюра, но некоторые точки оригинального изображения – исключены. Scale = 1.0 – означает, что все точки оригинального изображения видны, но ректифицированное изображение имеет чёрные края там, где нет соответствующих точек на оригинальном изображении.

Пример выдачи параметров калибровки в окно терминала:

```
D =  [-0.33758562758914146, 0.11161239414304096,
-0.00021819272592442094, -3.029195446330518e-05]
K =  [430.21554970319971, 0.0,
306.6913434743704, 0.0, 430.53169252696676,
227.22480030078816, 0.0, 0.0, 1.0]
R =  [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
1.0]
P =  [1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0]
# oST version 5.0 parameters


[image]

width
640
```

```
height
480

[narrow_stereo/left]

camera matrix
430.215550 0.000000 306.691343
0.000000 430.531693 227.224800
0.000000 0.000000 1.000000

distortion
-0.337586 0.111612 -0.000218 -0.000030 0.0000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
```

По нажатии кнопки COMMIT параметры калибровки будут отправлены камере для постоянного хранения (запишутся в yaml файл, указанный при инициализации камеры, в нашем примере - fe130_320_01.yaml). Окно GUI закрывается, в консоли отображается сообщение «writing calibration data to …».

С помощью инструмента Camera Calibration Parser (http://wiki.ros.org/camera_calibration_parsers) можно создать yml – файл, который можно использовать практически с любым драйвером камеры ROS с помощью параметра `camera_info_url`. Подробнее про калибровку камеры можно прочитать по ссылке: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration .

Распознавание маркеров и оценка положения камеры в пространстве

Для расчёта положения камеры относительно маркеров в пространстве построим несложный алгоритм, выполняющий следующие шаги:

1. Распознать маркеры на изображении
2. Нарисовать распознанные маркеры на изображении
3. Посчитать положение маркеров в пространстве (получить вектор трансляции и ротации)
4. Нарисовать оси на распознанных маркерах – чтоб убедиться визуально, что наш алгоритм работает правильно.

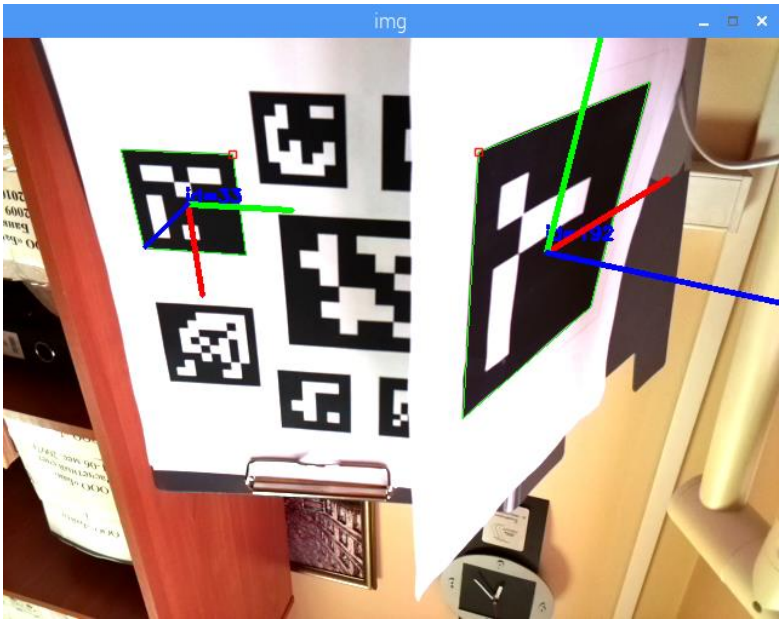
Ниже приведён пример кода:

```
# coding=UTF-8
################## Детектор маркетов ARUCO
# import the necessary packages
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import numpy as np
import cv2.aruco
#import glob

# Load previously saved data
with np.load('B.npz') as X:
   mtx, dist, _, _ = [X[i] for i in ('arr_0','arr_1','arr_2','arr_3')]
#print 'mtx=',mtx,'dist=',dist
#exit(0)

# initialize the camera and grab a reference to the raw camera capture
camera = PiCamera()
camera.resolution = (640, 480)#(800, 600)#
camera.framerate = 24 #32
rawCapture = PiRGBArray(camera, size=(640, 480)) #(800, 608)

# allow the camera to warmup
time.sleep(0.1)
font = cv2.FONT_HERSHEY_SIMPLEX
```

```
framenumber = 0
markerLength = 13.3 # 133 миллиметра сторона маркера

dictionary =
cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_ARUCO_ORIGINAL)

# capture frames from the camera
for frame in camera.capture_continuous(rawCapture, format="bgr",
use_video_port=True):
    # grab the raw NumPy array representing the image, then initialize the
timestamp
    # and occupied/unoccupied text
    img = frame.array
    # увеличим яркость, дабы робот не ослеп
    cv2.convertScaleAbs(img,img,1.8, 0)#img = img * 2 #convertTo(img,-
1,1.5,50)
    # обработаем картинку
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    corners, ids, rejected = cv2.aruco.detectMarkers(gray, dictionary)
    if len(corners)>0:
        cv2.aruco.drawDetectedMarkers(img, corners, ids)
        rvecs = cv2.aruco.estimatePoseSingleMarkers(corners, markerLength,
mtx, dist)
        print 'corners = ',corners
        print 'rvecs =',rvecs
        #exit(0)
        i=0
        while i<len(rvecs[0]):
            cv2.aruco.drawAxis(img, mtx, dist, rvecs[0][i], rvecs[1][i], 5)
            i=i + 1
    #"r: "+str(rvecs[0][0])+
    #cv2.putText(img," v:"+str(rvecs[0][0]),(10,450), font,
1,(0,128,0),1,cv2.LINE_AA)
    cv2.imshow('img', img)
    #cv2.imshow("Frame", gray) #image)
    key = cv2.waitKey(1) & 0xFF

    # clear the stream in preparation for the next frame
    rawCapture.truncate(0)
    framenumber=framenumber+1
    #print framenumber, ret
    # if the `q` key was pressed, break from the loop
    if key == ord("q"):#or framenumber>20:
        break

#cv2.destroyAllWindows()
```

Приведенный код реализует данный алгоритм. Для выполнения его нужно сохранить в текстовый файл и выполнить с помощью команды python <имя файла>.

При корректной работе программы на экране отобразится видеопоток с отметкой распознанных маркеров:



Раздел 4. Среда визуализации RVIZ

RVIZ – инструмент с открытым исходным кодом, предназначенный для визуализации процессов и отладки алгоритмов робототехнической системы.

Подробно об инструменте можно прочитать на странице сообщества разработчиков http://wiki.ros.org/rviz . Ниже мы приведём описание основных функций.

Установка RVIZ

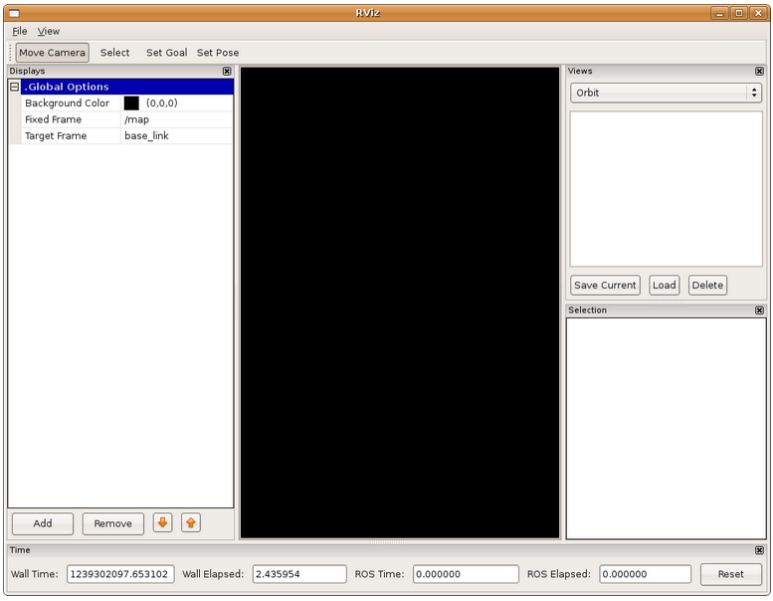
В случае установки полного desktop - дистрибутива ROS kinetic - rviz будет установлен в его составе.

Если он не установлен – установка осуществляется с помощью команды `sudo apt-get install ros-kinetic-rviz`.

Запускается среда визуализации с помощью команды `rviz`.

Визуализация в RVIZ

Когда rviz запускается в первый раз – на экране отображается пустое окно:

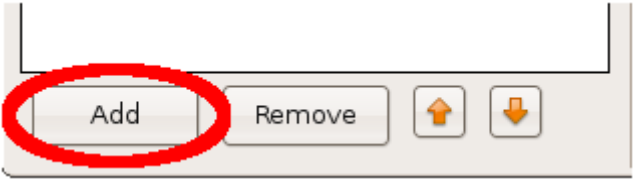


Большой чёрный квадрат – это 3D – вид сцены (пустой, поскольку ещё пока ничего не видим). Слева – список дисплеев Displays, на котором отображаются все настроенные дисплеи. В начальный момент на нём отображаются только глобальные настройки.

Дисплеи

Дисплей в rviz – это нечто, имеющее отображение в 3D – мире и имеющее опции настройки. Например: облако точек, позиция робота, и т.д.

Добавление нового дисплея – кнопка Add на панели внизу:

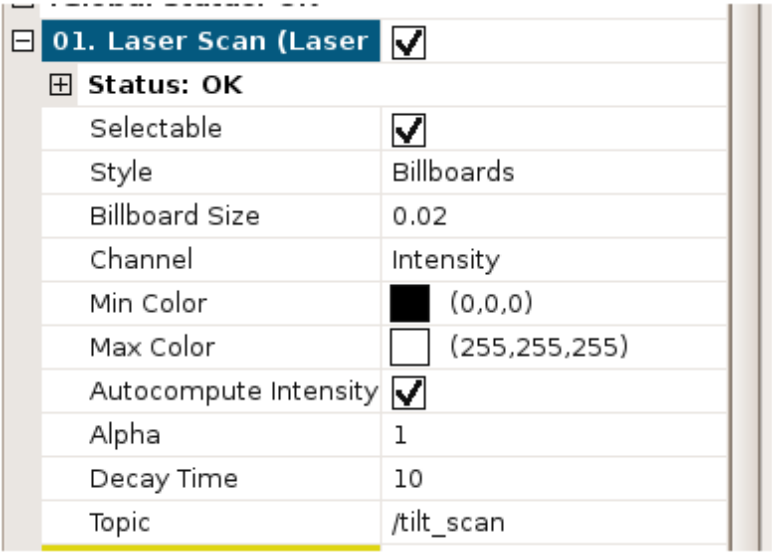


Появится окно добавления дисплея:

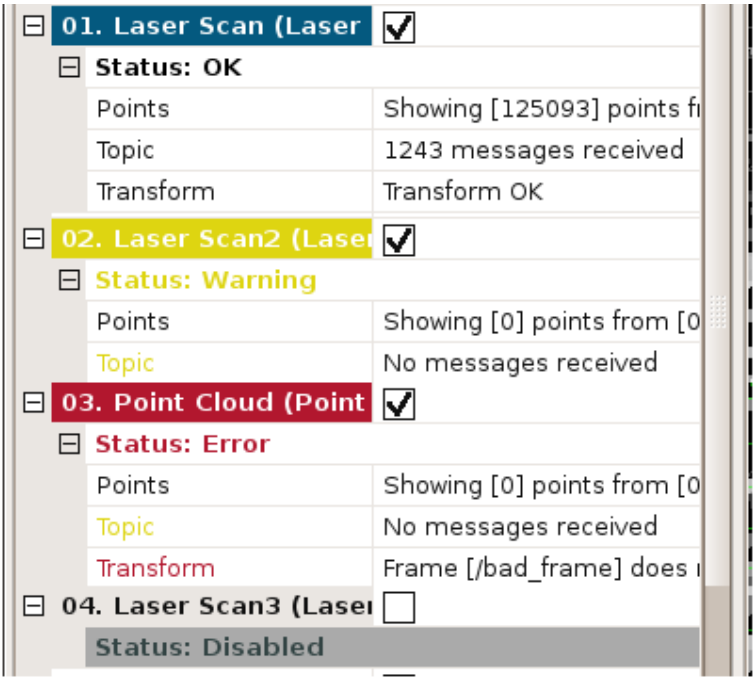
В списке вверху перечислены типы дисплеев. Тип дисплея определяет, какой тип данных дисплей визуализирует. В текстовом окне приводится описание выбранного типа дисплея. Внизу следует задать наименование дисплея. Например, если у Вашего

робота два лазерных сканера – вы можете создать два дисплея типа «Laser Scan» и назвать их «Лазер База» и «Лазер Голова».

У каждого дисплея есть собственные параметры, например:



У каждого дисплея также отображается статус, чтобы пользователь знал, всё ли с ним в порядке. Есть 4 варианта статуса: OK, Warning (предупреждение), Error (ошибка) или Disabled (выключен). Статус отображается в названии дисплея в виде цвета фона, а также в строке Status в параметрах дисплея:



Перемещение дисплеев выше/ниже реализовано с помощью механизма Drag'n'Drop указателем мыши.

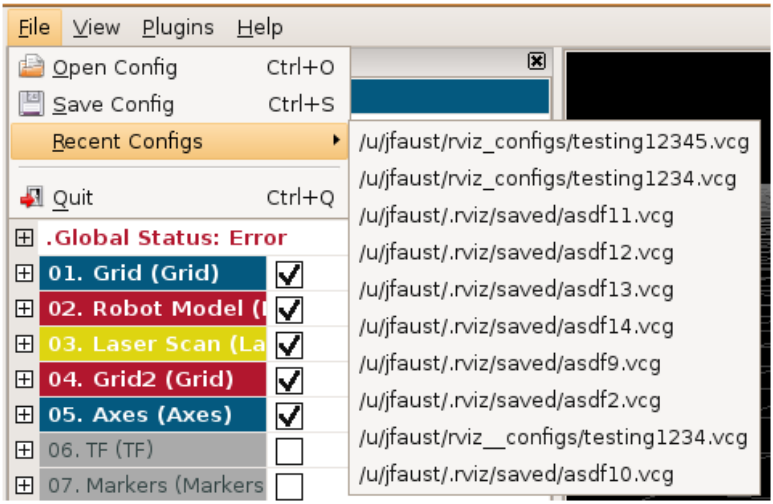
Встроенные типы дисплеев:

Название	**Описание**	**Используемые сообщения**
Axes	Отображает набор осей координат	
Effort	Отображает прилагаемое усилие прилагаемое к механизму робота	sensor_msgs/ JointStates
Camera	Создаёт новое 3D окно с точки зрения камеры, и накладывает изображение 3D мира на него	sensor_msgs/ Image, sensor_msgs/ CameraInfo
Grid	Рисует 2D или 3D сетку	

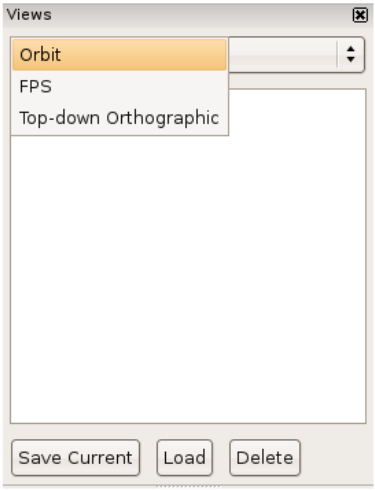
Grid Cells	Рисует ячейки сетки, обычно препятствия из стека навигации	nav_msgs/ GridCells
Image	Создаёт новое окно рендеринга с изображением. В отличие от дисплея типа Camera, этот тип дисплея не использует CameraInfo	sensor_msgs/ Image
Interactive Marker	Отображает 3D от одного или нескольких серверов интерактивных маркеров, и обеспечивает взаимодействие указателя мыши с ними	visualization_msgs/ InteractiveMarker
Laser Scan	Показывает данные лазерного сканера, с различными опциями режимов рендеринга, аккумуляции и т.д.	sensor_msgs/ LaserScan
Map	Отображает карту навигации	nav_msgs/ OccupancyGrid
Markers	Позволяет программисту отобразить любые примитивные формы с помощью ROS топика	visualization_msgs/Marker, visualization_msgs/ MarkerArray
Path	Отображает путь движения из стека навигации	nav_msgs/ Path
Point	Отображает точку в виде маленькой сферы	geometry_msgs/ PointStamped
Pose	Рисует положение в пространстве в виде стрелки или трёх осей	geometry_msgs/ PoseStamped
Pose Array	Рисует «облако» стрелок, по одной для каждого положения из массива положений	geometry_msgs/PoseArray
Point Cloud(2)	Показывает облако точек, с различными опциями режима рендеринга, накопления и т.д.	sensor_msgs/ PointCloud, sensor_msgs/ PointCloud2
Polygon	Рисует многоугольник	geometry_msgs/Polygon

Odometry	Накапливает положения одометрии	nav_msgs/ Odometry
Range	Показывает конус, представляющий датчик расстояния – сонар или инфракрасный сенсор	sensor_msgs/ Range
Robot Model	Показывает визуальное представление робота в пространстве (в соответствии с текущим преобразованием tf)	
TF	Отражает иерархию трансформаций TF	
Wrench	Отображает усиливание скручивания в виде стрелки (сила) + окружности (момент)	geometry_msgs/ WrenchStamped
Oculus	Преобразует 3D сцену rviz для 3В очков Oculus	

Различные конфигурации дисплеев часто полезны для разных задач визуализации. Поэтому rviz позволяет загружать и сохранять различные конфигурации. Конфигурация содержит: дисплеи + их параметры, параметры инструментов, тип камеры + начальную точку зрения. Недавно использованные конфигурации отображаются в меню Recent Configs:

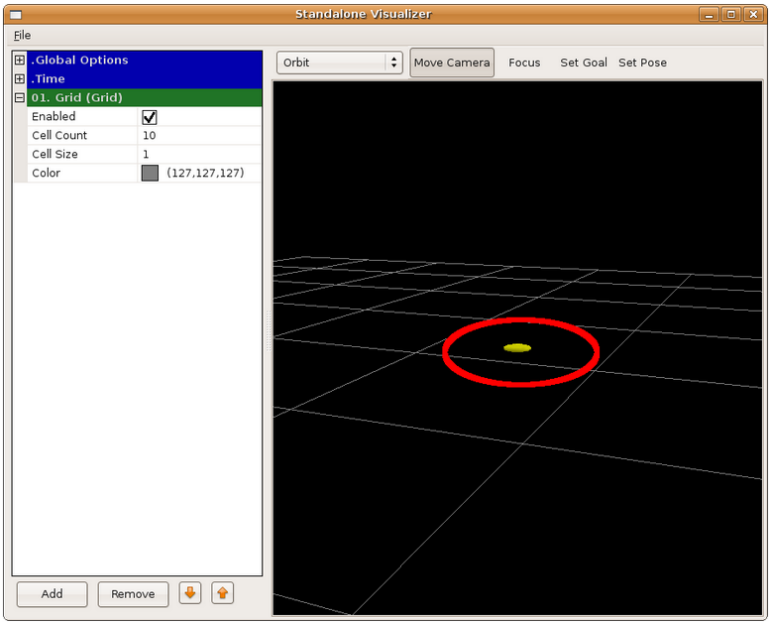


Для просмотра 3D сцены доступны несколько типов камер, по мере развития rviz добавляются новые типы:



Тип камеры определяет как способы управления камерой, так и разные типы проекции (ортографический, перспективный и т.д.)

Орбитальная камера (Orbital Camera) всегда вращается вокруг точки фокуса, всё время смотря в эту точку. Точка фокуса отображается в виде маленького диска, когда вы перемещаете камеру:



Управление: левая кнопка мыши – вращение камеры вокруг точки фокуса, средняя кнопка – перемещение фокальной точки, правая кнопка или колёсико мышки – приближение к/ удаление от фокальной точки.

Камера от первого лица (FPS – First Person Camera) – представление сцены от лица зрителя.

Управление: левая кнопка мыши – вращение камеры, средняя кнопка – перемещение камеры, правая кнопка или колёсико мышки – перемещение зрителя вперёд/назад.

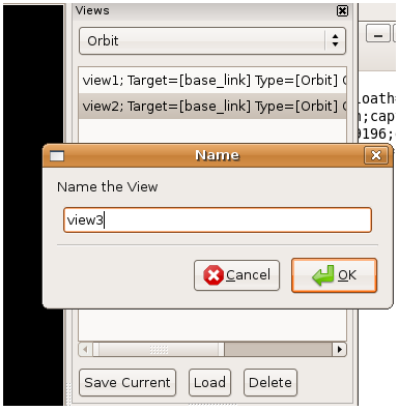
Ортографическая камера (Top-down Orthographic) – всегда направлена вниз по оси Z и отображает ортографическую проекцию (т.е. объекты по мере удаления от камеры не становятся больше/меньше).

Управление: левая кнопка мыши – вращение вокруг оси Z, средняя кнопка – перемещение камеры в плоскости XY, правая кнопка или колёсико мышки – увеличение/уменьшение изображения.

XY Orbit – то же, что орбитальная камера, но точка фокуса ограничена плоскостью XY.

Third Person Follower – камера сохраняет постоянный угол зрения относительно установленной системы координат. В отличие от XY Orbit камера вращается вместе с поворотом целевой системы координат.

Панель Views (Виды) также позволяет сохранять несколько именованных видов и переключаться между ними. Вид состоит из целевой системы координат, типа и положения камеры. Вид может быть сохранён с помощью кнопки Save Current.



Вид состоит из типа контроллера, конфигурации вида (положение, ориентация, и т.д.), целевой системы координат.

Виды сохраняются для пользователя ОС, а не в файлах конфигурации.

Системы координат

RVIZ использует пакет трансформации TF для перевода данных из исходной системы координат в глобальную систему. О двух системах координат визуализатора важно знать: Fixed Frame и Target frame.

Наиболее важная система координат – это **Fixed Frame**. Эта система координат соответствует системе координат «мир» робота. Обычно она называется «map», или «world». В случае протокола mavros этим параметрам соответствует «local_origin».

Если fixed frame по ошибке присвоен, например, базе робота, тогда все объекты о которых робот знает, появятся в визуализаторе в положении относительно робота, в тех положениях, в которых их обнаружили сенсоры. Для получения корректных результатов, **fixed frame не должен перемещаться относительно реального мира**.

Если в настройках меняется fixed frame – все отображаемые данные будут очищены, а не трансформированы.

Target frame – это целевая система координат для вида камеры. Например, если target frame – это карта местности map – мы увидим робота, двигающегося относительно карты. Если target frame – это база робота – на 3D сцене робот будет стоять на месте, а всё остальное будет двигаться относительно него.

Также эти системы координат описаны в видео: https://www.youtube.com/watch?v=nD1Faww4m2c

В визуализаторе также реализован набор инструментов, доступных в соответствующей панели: движение камеры, выбор объектов на 3D сцене, задание цели движения, оценка положения в пространстве.

Панель времени (Time) наиболее полезна при использовании симулятора: она позволяет отследить как много времени ROS затрачено относительно реального времени. Также она позволяет сбросить внутреннее время визуализатора – при этом сбрасываются все дисплеи и внутренний кэш модуля TF. Если не использовать симуляцию – эта панель скорее бесполезна и её можно закрыть, освободив место на экране.

RVIZ может отображать 3D стерео, если это поддерживает графический адаптер, монитор и очки. Установка стерео описана по ссылке http://wiki.ros.org/rviz/Tutorials/Rviz%20in%20Stereo

Также новые типы дисплеев могут быть подключены в rviz в виде внешних плагинов.

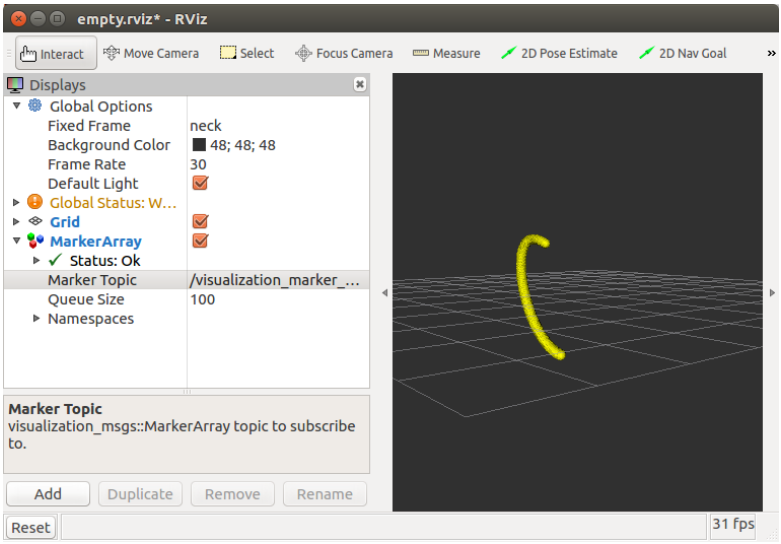
Пример программы отображения маркеров в RVIZ

Рассмотрим простую программу публикации маркеров в пространстве виртуального мира.

Программа в цикле создаёт набор из 100 маркеров, которые расположены в трёхмерном пространстве по синусоиде «змейкой». Ниже приведён текст программы:

```
# coding=UTF-8
#!/usr/bin/env python

from visualization_msgs.msg import Marker
from visualization_msgs.msg import MarkerArray
import rospy
import math

topic = 'visualization_marker_array'
publisher = rospy.Publisher(topic, MarkerArray)

rospy.init_node('register')

markerArray = MarkerArray()

count = 0
MARKERS_MAX = 100

while not rospy.is_shutdown():
   marker = Marker()
   marker.header.frame_id = "/neck"
   marker.type = marker.SPHERE
   marker.action = marker.ADD
   marker.scale.x = 0.2
   marker.scale.y = 0.2
   marker.scale.z = 0.2
   marker.color.a = 1.0
   marker.color.r = 1.0
   marker.color.g = 1.0
   marker.color.b = 0.0
   marker.pose.orientation.w = 1.0
   marker.pose.position.x = math.cos(count / 50.0)
   marker.pose.position.y = math.cos(count / 40.0)
   marker.pose.position.z = math.cos(count / 30.0)

   # Добавляем маркер в MarkerArray, самый старый - удаляем
   if(count > MARKERS_MAX):
       markerArray.markers.pop(0)
   markerArray.markers.append(marker)

   # Перенумеруем ID маркеров
   id = 0
   for m in markerArray.markers:
       m.id = id
       id += 1

   # Публикуем массив маркеров
   publisher.publish(markerArray)
   count += 1
   rospy.sleep(0.01)
```

В результате выполнения программы в rviz можно наблюдать меняющийся во времени набор маркеров:



Пример программы отображения положения карты маркеров в пространстве с использованием RVIZ

В предыдущих главах мы отдельно рассмотрели механизмы получения видеопотока с камеры через ROS, поиска и оценки положения Aruco маркера в пространстве, а также механизм визуализации объектов в 3D пространстве с помощью rviz.

Следующим промежуточным этапом разработки системы автономной навигации является объединение этих шагов, полный код программы приведён в файле zuza_marker.py. Алгоритм реализует следующие шаги:

1. Получить и сохранить информацию о калибровке камеры.
2. Подписаться на топик изображений камеры, начать получать кадры. В каждом кадре:
    a. Найти Aruco маркеры и определить их положение в пространстве;
    b. Сгенерировать маркеры в rviz, соответствующие найденным Aruco маркерам;
    c. Раз в 5 секунд посчитать реальную частоту обработки изображений, и опубликовать её вместе с нарисованными маркерами в отладочном видеопотоке.

Ниже мы рассмотрим этот алгоритм по отдельным процедурам.

Получение и сохранение данных калибровки камеры

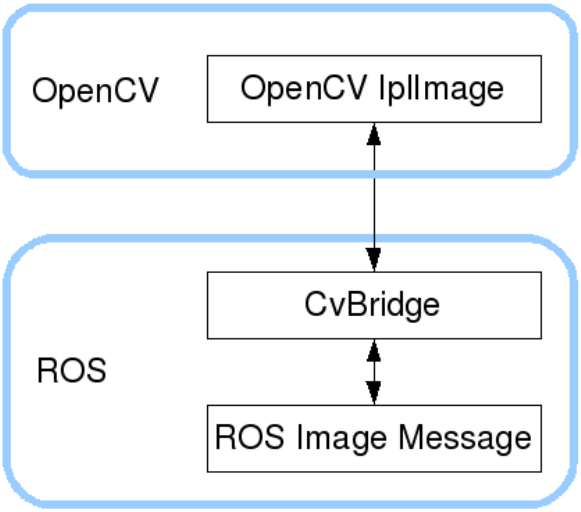
Данные калибровки камеры сохраним в глобальных переменных нашей программы: mtx – матрица камеры, dist – коэффициенты искажения. В ROS Эти данные камера публикует в топик camera_info. Можно получать их при каждой обработке кадра, но это лишняя трата производительности процессора. Поэтому мы используем процедуру, которая подписывается на топик /main_camera/camera_info, ожидает, пока обработчик сохранит значение информации камеры, после этого переводит полученные данные в numpy-массивы, которые можно затем использовать в функциях OpenCV. Ниже приведён код этих процедур, а также код, импортирующий нужные нам модули:

```
#!/usr/bin/env python
# coding=UTF-8
import rospy
from sensor_msgs.msg import CameraInfo, Image
import numpy as np
```

```
import time
import cv2
from cv_bridge import CvBridge, CvBridgeError
from visualization_msgs.msg import Marker,
MarkerArray

import tf.transformations as t
from geometry_msgs.msg import Quaternion
import copy

my_fps=0
my_last_fps=0
my_time_fps=time.time()
bridge = CvBridge()
dictionary =
cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT
_4X4_1000)
markerLength = 0.165

my_cam_info=None
mtx=None
dist=None

def callback_cinfo(cinfo):
    global my_cam_info
    my_cam_info=cinfo

def process_camera_info():
    global mtx, dist
    my_sub =
rospy.Subscriber('/main_camera/camera_info',
CameraInfo, callback_cinfo)
    while my_cam_info==None:
        rospy.sleep(0.01)
    my_sub.unregister()
    mtx=np.zeros((3,3))
    dist=np.zeros((1,5))
    for i in range(3):
      for j in range(3):
        mtx[i,j]=my_cam_info.K[i*3+j]
    for i in range(5):
      dist[0,i]=my_cam_info.D[i]
```

Обработчик сообщения изображения кадра

Процедура-обработчик сообщения кадра ROS получает изображение в виде сообщения типа Image модуля sensor_msgs.msg. Для преобразования этого сообщения в массив, который можно обработать с помощью OpenCV используется модуль ROS cv_bridge (http://wiki.ros.org/cv_bridge). Данный модуль предназначен специально для конвертации изображений между ROS сообщениями и OpenCV:



В глобальной переменной my_fps программа считает количество кадров, обработанных процедурой за последние 5 секунд. Вывод значения fps на изображение осуществляется с помощью функции cv2.putText.

Поиск и отображение маркеров на изображении производится с помощью знакомых нам процедур detectMarkers, drawDetectedMarkers.

Определение положения маркеров в пространстве относительно оптического центра камеры – процедура estimatePoseSingleMarkers. В неё кроме координат углов и размеров маркера передаются оптические параметры камеры.

Далее для каждого распознанного на изображении маркера создаётся маркер визуализации в rviz.

Углы вращения функция OpenCV estimatePoseSingleMarkers возвращает в виде вектора вращения. А в системе ROS поворот объекта (в т.ч. и маркера) задаётся в виде Кватерниона. Для преобразования поворота из вектора вращения в кватернион мы сначала получаем матрицу вращения с помощью функции cv2.Rodrigues, а затем матрицу вращения преобразуем в кватернион с помощью функции quaternion_from_matrix модуля tf. transformations. При этом функция quaternion_from_matrix требует матрицу вращения размером 4x4 элемента, т.к. она может преобразовывать вращение в т.ч. относительно точки, отличной от начала координат. Но из OpenCV мы получаем вектор вращения относительно оптического центра камеры (начала координат), поэтому полученную в cv2.Rodrigues матрицу 3x3 мы накладываем на единичную диагональную матрицу 4x4.

Ниже приведён полный код процедуры:

```
def callback_image_raw(image_raw):
    global my_fps, my_last_fps, my_time_fps,
dictionary, mtx, dist
    try:
      cv_image = bridge.imgmsg_to_cv2(image_raw,
"bgr8")
    except CvBridgeError as e:
      print(e)
```

```
    # calculate FPS
    cur_time2 = time.time()
    if cur_time2-my_time_fps>5:
        my_last_fps=my_fps
        my_fps=1
        my_time_fps=cur_time2
    else:
        my_fps+=1

    #Detect markers
    corners, ids, rejected =
cv2.aruco.detectMarkers(cv_image, dictionary)
    if len(corners)>0:
        cv2.aruco.drawDetectedMarkers(cv_image,
corners, ids)
        rvecs,tvecs, _ObjPoints =
cv2.aruco.estimatePoseSingleMarkers(corners,
markerLength, mtx, dist)
        i=0
        markerArray = MarkerArray()
        Duration1sec = rospy.Duration(1)
        while i<len(rvecs):
          cv2.aruco.drawAxis(cv_image, mtx, dist,
rvecs[i], tvecs[i], markerLength)
          ######################## begin marker
publishing ##############
          marker = Marker()
          marker.id = i
          marker.lifetime = Duration1sec
          marker.header.frame_id = "/local_origin"
          marker.type = marker.CUBE
          marker.action = marker.ADD
          marker.scale.x = markerLength
          marker.scale.y = markerLength*1.2
          marker.scale.z = 0.02
          marker.color.a = 1.0
          marker.color.r = 0
          marker.color.g = 1
          marker.color.b = 0
          #преобразуем вектор вращения в матрицу
вращения
          # quaternion_from_matrix понимает и
вращение относительно точки <> началу координат
          # поэтому матрица = 4x4
          mat4 = np.identity(4)
```

```
            mat4[:3, :3], jacob =
cv2.Rodrigues(rvecs[i][0])
            q = t.quaternion_from_matrix(mat4)
            marker.pose.orientation = Quaternion(*q)

            marker.pose.position.x = tvecs[i][0][0]
            marker.pose.position.y = tvecs[i][0][1]
            marker.pose.position.z = tvecs[i][0][2]
            markerArray.markers.append(marker)

            marker2 = copy.deepcopy(marker)
            marker2.id = i+1
            marker2.type = marker.ARROW
            marker2.scale.x = markerLength*1.5
            marker2.scale.y = markerLength/10
            marker2.color.r = 1
            marker2.color.g = 0
            marker2.color.b = 0
            markerArray.markers.append(marker2)

            ######################## end marker
publishing ##############
            i+=2
          # Publish the MarkerArray
          publisher_mrk.publish(markerArray)
    #Draw FPS on image
    cv2.putText(cv_image,"fps:
"+str(my_last_fps/5),(10,30),
cv2.FONT_HERSHEY_SIMPLEX, 1,(0,255,0),2,cv2.LINE_AA)

    try:

publisher_img.publish(bridge.cv2_to_imgmsg(cv_image,
"bgr8"))
    except CvBridgeError as e:
      print(e)
```

Основная программа

Код основной программы инициализирует ROS-ноду zuza_vpe, запоминает параметры камеры (process_camera_info), инициализирует публикаторы изображения отладки (publisher_img) и маркеров rviz (publisher_mrk), а также подписку на кадры с камеры

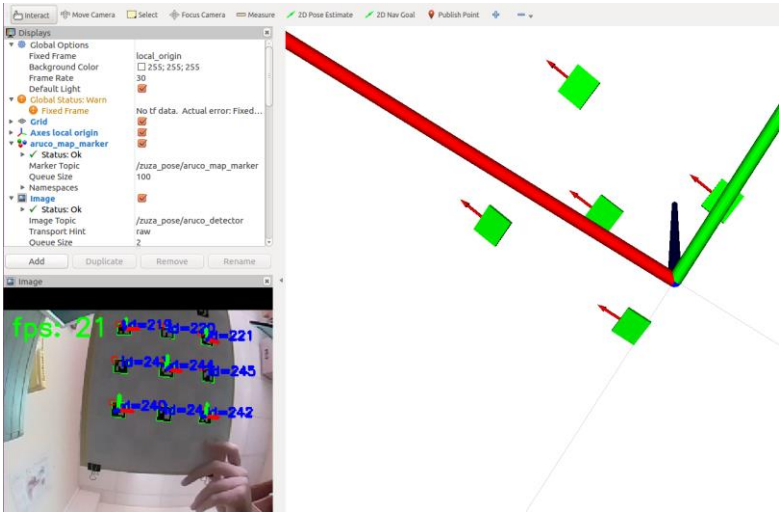
(my_sub). Дальнейшая обработка кадра, полученного с камеры, осуществляется в процедуре callback_image_raw.

Ниже приведён текст основной программы:

```
if __name__ == '__main__':
    rospy.init_node('zuza_pose')
    process_camera_info()
    print my_cam_info
    print "image translation starting........."
    publisher_img =
rospy.Publisher('zuza_pose/aruco_detector',
Image, queue_size=2)
    publisher_mrk =
rospy.Publisher('zuza_pose/aruco_map_marker',
MarkerArray, queue_size=2)
    my_sub =
rospy.Subscriber('/main_camera/image_raw',
Image, callback_image_raw)
    try:
      rospy.spin()
    except KeyboardInterrupt:
      print("Shutting down")
    my_sub.unregister()
```

Перед тем, как запускать программу, нам нужно запустить roscore и ноду, транслирующую изображения (`roslaunch zuza zuza.launch`), в отдельных сессиях терминала. Способ автоматизации запуска программ будет рассмотрен с следующем разделе. Детектор маркеров запускается на выполнение командой `python zuza_marker.py`.

После запуска детектора маркеров, результаты его работы удобно проверять с помощью rviz. При появлении на изображении камеры Aruco-маркеров – распознанные маркеры отображаются в rviz по своим координатам:



С помощью рулетки можно проверить соответствие расстояний, отображаемых в rviz расстояниям до маркеров в реальном мире. Нужно добиться максимального соответствия картины в rviz реальным координатам маркеров, с помощью калибровки камеры и настройки размера маркера в программе.

Раздел 5. Реализация зависания дрона под Aruco маркерами

В предыдущем разделе мы создали прототип системы визуальной одометрии – оценки положения камеры в пространстве с помощью обработки видеопотока с этой камеры. Для того, чтобы создать механизм ориентации дрона в пространстве по маркерам, нужно дополнить эту систему следующими компонентами:

- Генерация и распознавание карты маркеров. В принципе, дрон может зависать и над одним маркером, но навигация с помощью машинного зрения станет невозможной как только камера потеряет маркер из вида. Поэтому лучше сразу начинать использовать карту маркеров, которая, в вырожденном случае, может состоять и из одного маркера;
- Связь с полётным контроллером с помощью пакета mavros. Весь обмен информацией с полётным контроллером осуществляется через отправку сообщений в/подписку на соответствующие ROS топики;
- Преобразование систем координат дрона, камеры и карты маркеров. За системы координат и пространственные преобразования в ROS отвечает библиотека tf2;
- Вспомогательные сервисы – автозапуск модулей при загрузке Raspberry PI. В предыдущих разделах, если проделывать приведённые примеры на «чистом» Линуксе – все сервисы начиная с roscore приходилось запускать вручную в отдельных сеансах. При использовании реального дрона это не удобно, поэтому мы сразу перейдём к рассмотрению механизма автозапуска сервисов в Линукс.

Автозапуск пакета программ с помощью roslaunch – сервиса и systemd

Для управления службами Линукс в Raspbian используется менеджер systemd. Подробно о его работе можно прочитать по ссылке https://wiki.archlinux.org/index.php/Systemd_(Русский) .

Добавление автозапуска сервиса мы рассмотрим на примере roscore. Для автозапуска сервиса нужно создать два файла: roscore.service – описание сервиса и roscore.env – описание переменных среды, которые использует сервис. Содержание файлов приведено ниже. Файлы можно поместить в подкаталог deploy рабочего пространства catkin_ws.

Файл roscore.service:

```
[Unit]
Description=Launcher for the ROS master,
parameter server and rosout logging node
After=network.target

[Service]
EnvironmentFile=/home/pi/catkin_ws/src/zuza/depl
oy/roscore.env
ExecStart=/opt/ros/kinetic/bin/roscore
Restart=on-abort

[Install]
WantedBy=multi-user.target
```

Файл roscore.env:

```
ROS_ROOT=/opt/ros/kinetic/share/ros
ROS_DISTRO=kinetic
ROS_PACKAGE_PATH=/home/pi/catkin_ws/src:/opt/ros
/kinetic/share
ROS_PORT=11311
ROS_MASTER_URI=http://localhost:11311
CMAKE_PREFIX_PATH=/home/pi/catkin_ws/devel:/opt/
ros/kinetic
PATH=/opt/ros/kinetic/bin:/usr/local/sbin:/usr/l
ocal/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/game
s:/usr/local/games:/snap/bin
LD_LIBRARY_PATH=/opt/ros/kinetic/lib
PYTHONPATH=/home/pi/catkin_ws/devel/lib/python2.
7/dist-
```

```
packages:/opt/ros/kinetic/lib/python2.7/dist-
packages
ROS_IP=192.168.11.1
```

После создания файлов описания добавить сервис в автозапуск нужно командой:

```
sudo systemctl enable
/home/pi/catkin_ws/src/zuza/deploy/roscore.servi
ce
```

После успешного выполнения этой команды сервис `roscore` будет запускаться автоматически. В чём можно убедиться: перезагрузив Raspberry PI, в терминале вводим команду `rostopic list` – она выдаёт список топиков и не ругается на не запущенный мастер ROS.

Аналогично, мы можем добавить в автозапуск наш ROS-пакет zuza (который мы запускали с помощью команды `roslaunch zuza zuza.launch`).

Создадим файл zuza.service с описанием сервиса:

```
[Unit]
Description=ZUZA ROS package
Requires=roscore.service
After=roscore.service

[Service]
EnvironmentFile=/home/pi/catkin_ws/src/zuza/depl
oy/roscore.env
ExecStart=/opt/ros/kinetic/bin/roslaunch zuza
zuza.launch --wait
Restart=on-abort

[Install]
WantedBy=multi-user.target
```

И добавим сервис в автозапуск командой:

```
sudo systemctl enable
/home/pi/catkin_ws/src/zuza/deploy/zuza.service
```

После этого наш ROS-пакет zuza будет запускаться автоматически при старте системы. В этом можно убедиться с помощью команды rostopic.list и программы image_view – просмотр видеопотока с камеры main_camera.

Добавление python-программы в ROS пакет zuza

Текущая версия ROS-пакета zuza запускает только модуль cv_camera, для запуска программы обработки видеопотока нужно отдельно выполнять команду `python zuza_marker.py`. После отладки кода любую Python-программу можно также добавить в ROS-пакет, чтобы она запускалась при старте системы.

В начале кода программы должна быть указана ссылка на интерпретатор Python, так называемый Шебанг (https://ru.wikipedia.org/wiki/Шебанг_(Unix)):

```
#!/usr/bin/env python
```

При этом запускаемый файл должен иметь разрешение на запуск кода программы. Разрешение можно добавить командой `chmod +x zuza_marker.py`, или в любом редакторе свойств файла.

Также важно, чтобы формат конца строк в файле python-скрипта соответствовал стандартам Unix (LF). В редакторе Notepad++ это соответствие устанавливается с помощью команды:

В launch-файл zuza.launch нужно добавить строку с описанием ноды со ссылкой на файл Python:

```
<node name="zuza_vpe" pkg="zuza"
type="zuza_marker.py" output="screen">
```

После этого нода zuza_vpe будет запускаться автоматически в составе ROS-пакета zuza.

Управлять состоянием сервиса можно с помощью опций командной строки команды systemctl:

`sudo systemctl status zuza` – отобразить состояние сервиса zuza

`sudo systemctl stop zuza` – остановить сервис zuza

`sudo systemctl start zuza` – запустить сервис zuza

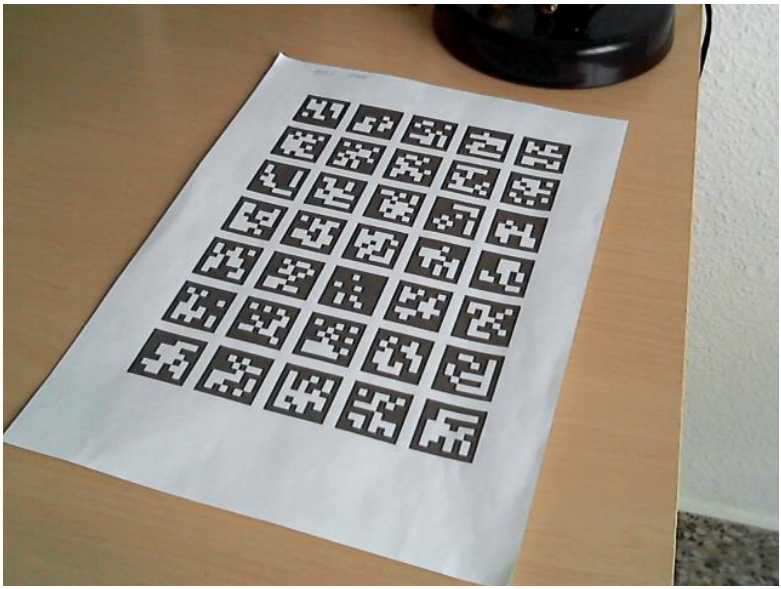
После остановки сервиса с опцией `stop` – ROS-пакет можно запустить на выполнение с помощью команды `roslaunch zuza zuza.launch`. При этом вывод сообщений будет осуществляться в терминал – что удобно для просмотра отладочных сообщений.

Также журнал сообщений сервиса можно просматривать с помощью команды `journalctl -u zuza`.

Генерация карты маркеров, передача и чтение ROS-параметров

Для ориентации дрона в помещении удобно использовать объект Aruco Board – набор (дословно – «доска») маркеров, который можно рассматривать как один большой маркер для определения позиции камеры относительно него. Наиболее популярный

набор – маркеры в одной плоскости, т.к. их проще напечатать:



Однако, Aruco Board не ограничен таким размещением маркеров, и может представлять из себя произвольный набор, размещённый в 2D или 3D пространстве.

Разница между Aruco Board и набором отдельных маркеров в том, что положение маркеров Aruco Board относительно друг друга известно априори. Поэтому любые найденные углы известных маркеров могут быть использованы для оценки положения камеры относительно всего набора Aruco Board. Когда мы используем набор несвязанных маркеров – положение каждого маркера оценивается индивидуально, но мы не знаем положение маркеров в пространстве относительно друг друга. Преимущества использования объекта Aruco Board следующие:

- Более точное и надёжное определение положения в пространстве. Всего несколько маркеров достаточно для определения положения. Положение можно рассчитать даже только по части карты, видимой камере.
- Рассчитанное по карте положение, как правило, более точное, т.к. используется большее количество точек соответствия (углов маркеров).

Поддержка карт маркеров реализована в модуле Aruco с помощью объекта Aruco.Board, у которого есть три параметра:

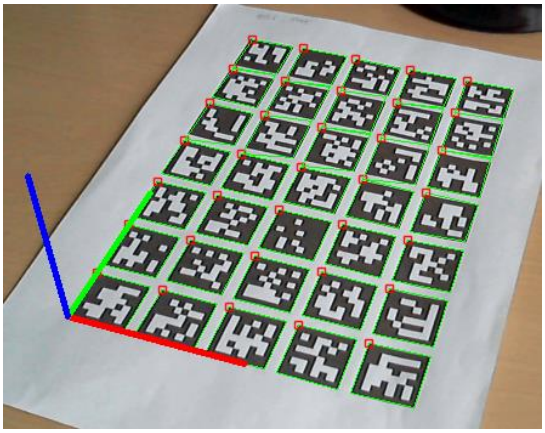
Структура **objPoints** – список координат углов карты маркеров в 3D пространстве. Для каждого маркера углы перечисляются по часовой стрелке, начиная с верхнего левого угла;

Параметр **dictionary** – определяет, к какому словарю относятся маркеры карты;

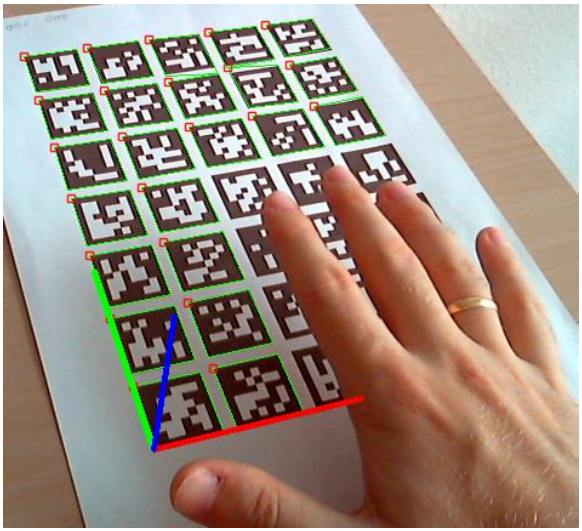
Структура **ids** – содержит список идентификаторов маркеров, перечисленных в objPoints, в словаре dictionary.

Определение положения в пространстве карты осуществляется также, как и определение положения маркера. Разница только в шаге определения положения камеры относительно карты, с помощью функции estimatePoseBoard.

Функция drawAxis() может быть использована для проверки полученного положения, например:



При этом положение карты может быть определено, если и не все маркеры видны:



Положение карты определяется с помощью найденных маркеров.

Параметры карты в нашу программу мы будем передавать с помощью сервера параметров ROS (http://wiki.ros.org/Parameter Server).

Наша карта представляет из себя прямоугольную область, равномерно заполненную маркерами одного размера и неизменной ориентации. Для описания нашей карты маркеров будем использовать 4 параметра:

Количество маркеров по оси X – **markers_x**

Идентификаторы маркеров карты – **marker_ids**

Размер стороны маркера в м – **markers_side**

Расстояние между маркерами в м – **markers_sep**

Добавим описание ноды с параметрами в файл zuza.launch:

```
<!-- zuza pose estimator -->
<node name="zuza_pose" pkg="zuza"
type="zuza_board.py" output="screen">
<param name="markers_x" value="3"/>
<param name="markers_side" value="0.188"/>
<param name="markers_sep" value="0.376"/>
<rosparam param="marker_ids">[222, 223, 224,
219, 220, 221, 243, 244, 245, 240, 241,
242]</rosparam>
</node>
```

Скопируем сам файл программы zuza_marker.py в zuza_board.py, и начнём доработку кода для оценки положения дрона в пространстве по карте маркеров.

Первым делом, добавим процедуру для чтения параметров и генерации объекта карты маркеров, который мы будем использовать в дальнейшем:

```
def get_ArucoBoard():
    global aruco_map_width, aruco_map_height
    markers_x=rospy.get_param('/zuza_pose/markers_x')

marker_ids=np.array(rospy.get_param('/zuza_pose/marke
r_ids'))
    markers_y=len(marker_ids)/markers_x
    markerLength =
rospy.get_param('/zuza_pose/markers_side')
    markers_sep =
rospy.get_param('/zuza_pose/markers_sep')
    objPoints=np.array([],np.float32)
    for j in range(markers_y):
        for i in range(markers_x):
            cur_x = (markers_sep+markerLength)*i
            cur_y = (markers_sep+markerLength)*j
            objPoint=np.array([[cur_x, cur_y, 0],
[cur_x+markerLength, cur_y, 0], [cur_x+markerLength,
cur_y+markerLength, 0], [cur_x, cur_y+markerLength,
0]],np.float32)
            objPoints=np.append(objPoints,objPoint)

board=cv2.aruco.Board_create(objPoints.reshape(marker
s_x*markers_y,4,3),dictionary,marker_ids)
    aruco_map_width=cur_x+markerLength
    aruco_map_height=cur_y+markerLength
    return board
```

Функция получает параметры markers_x, marker_ids, markerLength, markers_sep из файла запуска ROS ноды zuza.launch, затем с помощью cv2.aruco.Board_create создаёт и возвращает объект-карту маркеров, который мы будем использовать в нашей программе.

Кроме карты маркеров нам нужна связь с полётным контроллером Pixhawk/Pixracer. Связь осуществляется по протоколу mavlink с помощью модуля mavros, краткое описание технологии применения которого проведено в следующем разделе.

Автозапуск mavros – связь с полётным контроллером

Пакет mavros обеспечивает связь системы ROS с различными автопилотами по протоколу mavlink. Также данный протокол обеспечивает связь по протоколу UDP со станцией наземного управления (например, QGroundControl). Подробное описание пакета mavros приведено по ссылке http://wiki.ros.org/mavros.

Для визуализации коптера в пространстве мы также будем использовать пакет mavros_extras (http://wiki.ros.org/mavros_extras).

Для запуска протокола mavros в рамках нашей ноды, добавим в начало файла zuza.launch ссылку на файл запуска пакета mavros:

```
<!-- mavros -->
<include file="$(find
zuza)/launch/mavros.launch">
<arg name="fcu_conn" value="usb"/>
<arg name="fcu_ip" value="127.0.0.1"/>
<arg name="gcs_bridge" value="tcp"/>
<arg name="viz" value="true"/>
</include>
```

Данный фрагмент содержит ссылку на файл запуска mavros.launch, которому передаются параметры:

- Способ соединения с полётным контроллером – **fcu_conn** – может принимать значения «uart» или «usb»;
- Ip адрес полётного контроллера - **fcu_ip**
- Протокол связи с наземной станцией – **gcs_bridge**
- Включать ли визуализацию коптера (пакет mavros_extras) – **viz**.

Сам файл mavros.launch - следующего содержания:

```
<launch>
    <arg name="fcu_conn" default="uart"/>
    <arg name="fcu_ip" default="127.0.0.1"/>
    <arg name="gcs_bridge" default="tcp"/>
    <arg name="viz" default="true"/>
    <arg name="respawn" default="true"/>

    <node pkg="mavros" type="mavros_node"
name="mavros" required="false"
clear_params="true" respawn="$(arg respawn)"
respawn_delay="5" output="screen">
            <!-- UART connection -->
            <param name="fcu_url"
value="/dev/ttyAMA0:921600" if="$(eval fcu_conn
is None or fcu_conn == 'uart')"/>

            <!-- USB connection -->
            <param name="fcu_url"
value="/dev/ttyACM0" if="$(eval fcu_conn ==
'usb')"/>

            <!-- sitl -->
            <param name="fcu_url"
value="udp://@$(arg fcu_ip):14557" if="$(eval
fcu_conn == 'udp')"/>

            <!-- gcs bridge -->
            <param name="gcs_url" value="tcp-
l://0.0.0.0:5760" if="$(eval gcs_bridge ==
'tcp')"/>
            <param name="gcs_url"
value="udp://0.0.0.0:14550@14550" if="$(eval
gcs_bridge == 'udp')"/>
            <param name="gcs_url" value="udp-
b://0.0.0.0:14550@14550" if="$(eval gcs_bridge
== 'udp-b')"/>
```

```
            <param name="gcs_url" value="udp-
pb://0.0.0.0:14550@14550" if="$(eval gcs_bridge
== 'udp-pb')"/>
            <param name="gcs_url" value=""
if="$(eval not gcs_bridge)"/>
            <param name="gcs_quiet_mode"
value="true"/>
            <param name="conn/timeout"
value="10"/>

            <!-- default px4 params -->
            <rosparam command="load"
file="$(find mavros)/launch/px4_config.yaml"/>

            <!-- additional params -->
            <param
name="local_position/frame_id"
value="local_origin"/>
            <param name="local_position/tf/send"
value="true"/>
            <param
name="local_position/tf/frame_id"
value="local_origin"/>
            <param
name="local_position/tf/child_frame_id"
value="fcu"/>
            <param
name="global_position/tf/send" value="false"/>
            <param name="imu/frame_id"
value="fcu"/>
            <rosparam param="plugin_blacklist">
                - safety_area
                - image_pub
                - vibration
                - distance_sensor
                - rangefinder
                - 3dr_radio
                - actuator_control
                - hil_controls
                - vfr_hud
                - px4flow
                - vision_speed_estimate
```

```
                - fake_gps
                - cam_imu_sync
                - hil
                - adsb
            </rosparam>
    </node>

    <!-- Copter visualization -->
    <include file="$(find
zuza)/launch/copter_visualization.launch"
if="$(arg viz)"/>
</launch>
```

Также, если включена визуализация коптера, используется файл copter_visualization.launch:

```
<launch>
<remap to="mavros/local_position/pose"
from="local_position"/>
<remap to="mavros/setpoint_position/local"
from="local_setpoint"/>
<node name="copter_visualization"
pkg="mavros_extras"
type="copter_visualization"/>
    <param
name="copter_visualization/fixed_frame_id"
value="local_origin"/>
    <param
name="copter_visualization/child_frame_id"
value="fcu"/>
    <param
name="copter_visualization/marker_scale"
value="0.5"/>
    <param
name="copter_visualization/max_track_size"
value="500"/>
    <param
name="copter_visualization/num_rotors"
value="4"/>
</launch>
```

В файле визуализации коптера, в частности, задаются:

- Название системы координат «мира» коптера («local_origin»);
- Название системы координат полётного контроллера («fcu»)
- Количество моторов у коптера (4)
- Масштаб отображения коптера. Масштаб отображения = 0.5 примерно соответствует реальному размеру УМК Жужа.

Launch-файлы можно включать один в другой с помощью директивы `include file`. Это удобно для настройки сложного робота, состоящего из многих ROS нод.

После включения запуска mavros в нашем пакете zuza, после запуска пакета (команда `roslaunch zuza zuza.launch`) мы увидим множество сообщений от пакета mavros. При успешном запуске процесс ноды mavros НЕ должен завершаться с ошибкой («Process mavros has died»). Успешный запуск mavros проверяется с помощью команды rostopic echo /mavros/state. Если mavros успешно запущен – в терминал примерно раз в секунду будут выдаваться сообщения о состоянии подключения полётного контроллера – так называемый «heartbeat»:

```
pi@raspberrypi:~ $ rostopic echo /mavros/state
header:
  seq: 20
  stamp:
    secs: 1533089194
    nsecs: 974550539
  frame_id: ''
connected: True
armed: False
guided: False
```

```
mode: "MANUAL"
system_status: 3
---
```

При правильном подключении состояние связи отображается как «подключен = Истина» (`connected: True`).

Если mavros не запускается или хартбит-сообщения не выдаются – нужно проверить связь полётного контроллера с Raspberry PI и соответствие указанного в параметрах в файле zuza.launch способа связи (usb или uart) тому, который используется на практике.

В некоторых случаях uart может быть отключен в настройках самой Raspberry PI, если попытка связи по uart неудачная – нужно либо включить uart в настройках Raspberry (файл config.txt), либо переключить связь на USB.

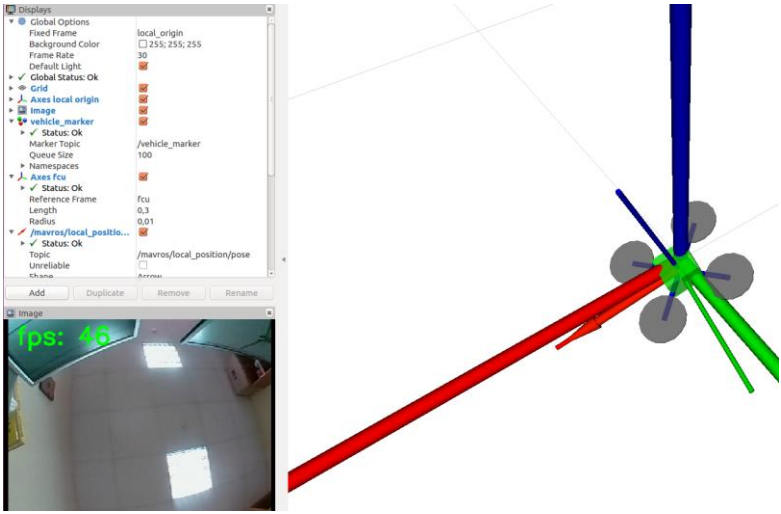
Чтобы инициализировать подсистему visual position estimator полётного контроллера – недостаточно подключиться к нему по протоколу mavros. Нужно также начать отправлять пустые нулевые сообщения в топик /mavros/vision_pose/pose . Делаем это при помощи несложной процедуры на Питоне:

```
def publish_dummy_vp(event):
    ps = PoseStamped()
    ps.header.stamp = rospy.Time.now()
    ps.header.frame_id = 'local_origin'
    ps.pose.orientation.w = 1;
    publisher_vp.publish(ps);
```

В основную программу добавим строки:

```
publisher_vp =
rospy.Publisher('/mavros/vision_pose/pose',
PoseStamped, queue_size=1)
dummy_timer=rospy.Timer(rospy.Duration(0.5),
publish_dummy_vp)
```

После этого запустим наш ROS пакет командой `roslaunch zuza zuza.launch`. В rviz можем наблюдать визуализацию коптера – маркеры из топика /vehicle_marker:



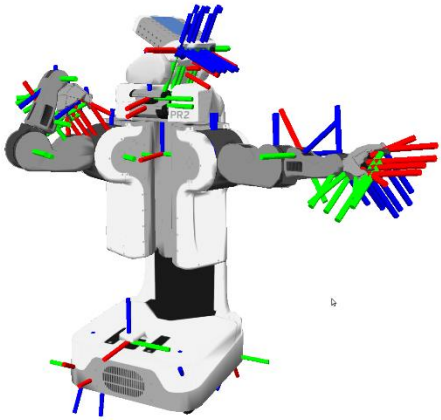
Также в rviz можно визуально контролировать положение системы координат коптера («fcu») относительно точки инициализации полётного контроллера («local_origin»).

Более подробно системы координат в ROS мы рассмотрим в следующем разделе.

Системы координат в ROS (модуль tf2)

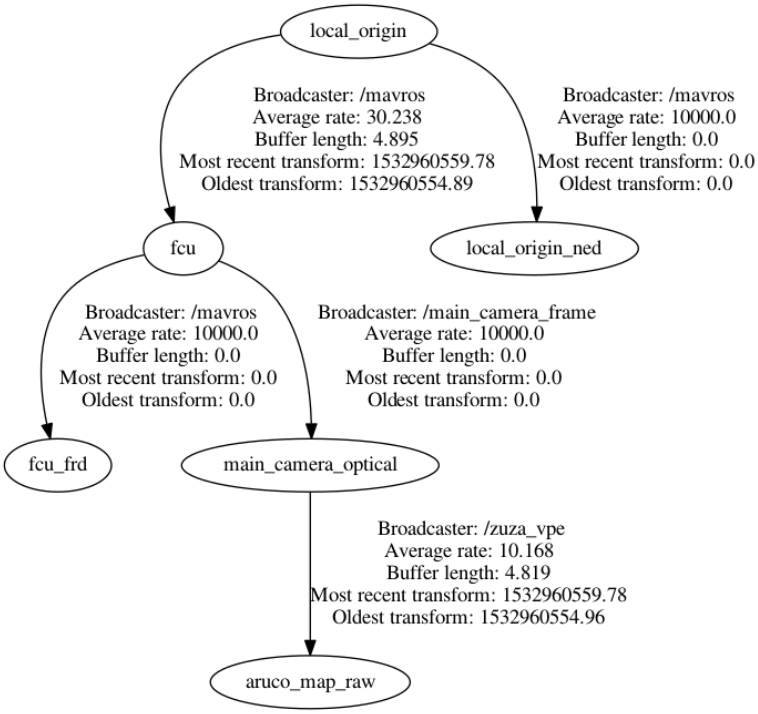
За поддержку систем координат в ROS отвечает библиотека трансформации (англ. «transform library») **tf2** (http://wiki.ros.org/tf2). Пакет tf2 отслеживает множество систем координат во времени. Связи между системами координат выстраиваются в виде дерева, что позволяет пользователю трансформировать точки, вектора и прочие геометрические элементы между любыми двумя системами координат в заданный момент времени.

При визуализации система координат каждого элемента обычно представляется в виде трёх осей: X-красный, Y-зелёный, Z-синий:



Пошаговое руководство по изучению пакета tf2 можно прочитать по ссылке http://wiki.ros.org/tf2/Tutorials.

Дерево систем координат для визуальной одометрии в нашем примере выглядит следующим образом:



Для ориентации коптера в пространстве используются следующие системы координат (или «фреймы», от англ. «frames»):

local _origin – система координат «мира» коптера. Устанавливается в точке инициализации полётного контроллера;

fcu – «flight control unit» - положение полётного контроллера относительно точки инициализации;

main_camera_optical – положение камеры Raspberry PI относительно полётного контроллера. Задаётся таким образом, что на изображении ось X указывает направо, Y – вниз, Z – вглубь изображения (ОТ матрицы камеры);

aruco_map_raw – положение карты найденных Aruco маркеров в пространстве. Задаётся относительно оптического центра камеры.

При работе с tf2 в программном коде решаются две основные задачи:

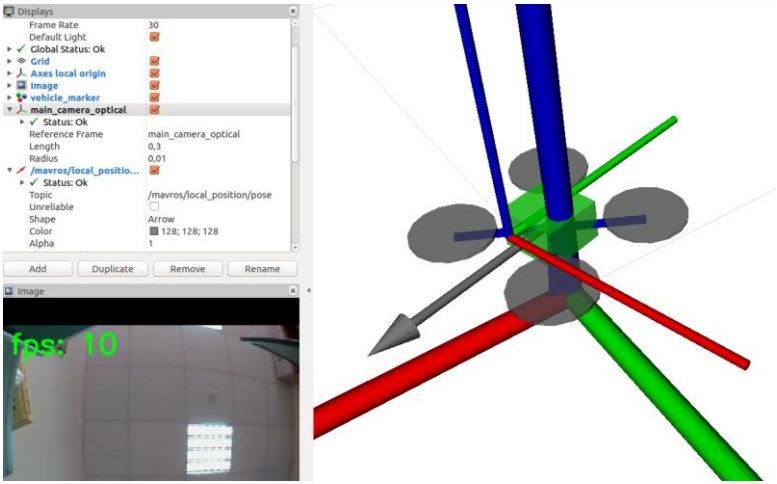
- Получение трансформации (англ. «listening for transforms») – получение информации о положении одной системы координат относительно другой;
- Публикация трансформации (англ. «broadcasting transforms») – отправка информации о положении систем координат относительно друг друга в систему. Публикация трансформации может быть как статической (англ. «static», т.е. не изменяемой во времени), так и динамической (англ. «dynamic»).

Пример статической трансформации – положение камеры относительно полётного контроллера дрона. Для публикации статической трансформации необязательно писать код, достаточно добавить строку с описанием статической трансформации в файл zuza.launch:

```
<node pkg="tf2_ros"
type="static_transform_publisher"
name="main_camera_frame" args="0.06 0 0.02
1.5707963 0 0 fcu main_camera_optical"/>
```

Параметры статической трансформации задаются в строковом аргументе args в порядке: смещение_X, смещение_Y, смещение_Z, рысканье, тангаж, крен. Длина смещение указывается в метрах, угол – в радианах.

После настройки положения камеры относительно корпуса полётного контроллера можно проверить правильность этой настройки с помощью rviz:



В следующем разделе учебного пособия мы определим и опубликуем в ROS положение карты маркеров в пространстве относительно камеры, и проверим результат в rviz.

Определение и публикация координат карты маркеров в пространстве

В предыдущих разделах мы разработали процедуру создания карты маркеров Aruco Board на основании данных, полученных из ROS параметров, переданных при запуске ROS ноды.

Модифицируем эту процедуру – добавим в неё создание объекта – маркера Rviz для визуализации, который будет представлять из себя прямоугольник размером с нашу карту Aruco маркеров. Результат создания маркера кмшя поместим в глобальную

переменную aruco_map_marker. Дополненный код процедуры приведён ниже:

```
from visualization_msgs.msg import Marker,
MarkerArray
aruco_map_width=0
aruco_map_height=0
aruco_map_marker=None

def get_ArucoBoard():
    global aruco_map_width, aruco_map_height,
aruco_map_marker
    markers_x=rospy.get_param('/zuza_pose/markers_x')

marker_ids=np.array(rospy.get_param('/zuza_pose/marke
r_ids'))
    markers_y=len(marker_ids)/markers_x
    markerLength =
rospy.get_param('/zuza_pose/markers_side')
    markers_sep =
rospy.get_param('/zuza_pose/markers_sep')
    objPoints=np.array([],np.float32)
    for j in range(markers_y):
        for i in range(markers_x):
            cur_x = (markers_sep+markerLength)*i
            cur_y = (markers_sep+markerLength)*j
            objPoint=np.array([[cur_x, cur_y, 0],
[cur_x+markerLength, cur_y, 0], [cur_x+markerLength,
cur_y+markerLength, 0], [cur_x, cur_y+markerLength,
0]],np.float32)
            objPoints=np.append(objPoints,objPoint)

board=cv2.aruco.Board_create(objPoints.reshape(marker
s_x*markers_y,4,3),dictionary,marker_ids)
    aruco_map_width=cur_x+markerLength
    aruco_map_height=cur_y+markerLength
    ## begin create Aruco map marker ##
    aruco_map_marker = Marker()
    aruco_map_marker.lifetime = rospy.Duration(1)
    aruco_map_marker.header.frame_id =
'main_camera_optical'
    aruco_map_marker.type =
aruco_map_marker.LINE_STRIP
    aruco_map_marker.action = aruco_map_marker.ADD
    aruco_map_marker.scale.x = 0.02
    aruco_map_marker.scale.y = 0.02
```

```
    aruco_map_marker.scale.z = 0.02
    color=ColorRGBA()
    color.a = 1.0
    color.r = 0.5
    color.g = 1
    color.b = 1

    pt=Point()
    aruco_map_marker.points.append(copy.deepcopy(pt))
    aruco_map_marker.colors.append(color)
    pt.x=aruco_map_width
    aruco_map_marker.points.append(copy.deepcopy(pt))
    aruco_map_marker.colors.append(color)
    pt.y=aruco_map_height
    aruco_map_marker.points.append(copy.deepcopy(pt))
    aruco_map_marker.colors.append(color)
    pt.x=0
    aruco_map_marker.points.append(copy.deepcopy(pt))
    aruco_map_marker.colors.append(color)
    pt.y=0
    aruco_map_marker.points.append(copy.deepcopy(pt))
    aruco_map_marker.colors.append(color)
    ##end create Aruco map marker #
    return board
```

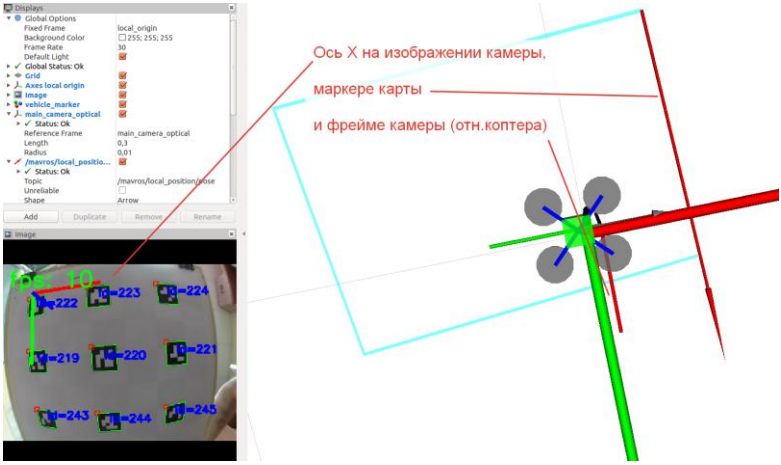
Переделаем процедуру обработки полученного через ROS кадра камеры – добавим определение положения карты в пространстве с помощью `cv2.aruco.estimatePoseBoard` и публикацию маркера карты. Текст изменённой процедуры – ниже:

```
def callback_image_raw(image_raw):
    global my_fps, my_last_fps, my_time_fps,
dictionary, mtx, dist, static_map_br
    try:
      cv_image = bridge.imgmsg_to_cv2(image_raw,
"bgr8")
    except CvBridgeError as e:
      print(e)

    # calculate FPS
    cur_time2 = time.time()
    if cur_time2-my_time_fps>5:
        my_last_fps=my_fps
```

```
        my_fps=1
        my_time_fps=cur_time2
    else:
        my_fps+=1

    #Detect markers
    corners, ids, rejected =
cv2.aruco.detectMarkers(cv_image, dictionary)
    if len(corners)>0:
        num_mark, rvec, tvec =
cv2.aruco.estimatePoseBoard(corners, ids,
my_aruco_board, mtx, dist)
        i=0
        markerArray = MarkerArray()
        Duration1sec = rospy.Duration(1)
        if num_mark>0:
            ### begin marker publishing ###
            marker = copy.deepcopy(aruco_map_marker)
            marker.id = i
            mat4 = np.identity(4)
            mat4[:3, :3], jacob = cv2.Rodrigues(rvec)

roll,pitch,yaw=t.euler_from_matrix(mat4[:3, :3])
            q=t.quaternion_from_euler(roll,pitch,yaw)
            marker.pose.orientation = Quaternion(*q)

            marker.pose.position.x = tvec[0]
            marker.pose.position.y = tvec[1]
            marker.pose.position.z = tvec[2]
            markerArray.markers.append(marker)

            marker2 = copy.deepcopy(marker)
            marker2.id = i+1
            marker2.type = marker.ARROW
            marker2.scale.x = aruco_map_width*1.5
            marker2.scale.y = markerLength/10
            marker2.color.r = 1
            marker2.color.g = 0
            marker2.color.b = 0
            marker2.color.a = 1
            markerArray.markers.append(marker2)
            ### end marker publishing ###
            publisher_mrk.publish(markerArray)

            cv2.aruco.drawDetectedMarkers(cv_image,
corners, ids)
```

```
            cv2.aruco.drawAxis(cv_image, mtx, dist,
rvec, tvec, min(aruco_map_width, aruco_map_height)/2)
    #Draw FPS on image
    cv2.putText(cv_image,"fps:
"+str(my_last_fps/5),(10,30),
cv2.FONT_HERSHEY_SIMPLEX, 1,(0,255,0),2,cv2.LINE_AA)

    try:

publisher_img.publish(bridge.cv2_to_imgmsg(cv_image,
"bgr8"))
    except CvBridgeError as e:
      print(e)
```

Результаты работы программы проверим с помощью rviz. Чтобы увидеть карту – добавим отображение топика zuza_pose/aruco_map_marker:



Обратите внимание, при правильной настройке – направления осей (на рисунке – ось X отмечена красным цветом) на изображении карты, маркере карты и системе координат камеры main_camera_optical – должны совпадать.

Таким образом, в нашей системе мы реализовали как расчёт положения коптера относительно начала координат, так и расчёт и визуализацию положения карты относительно камеры коптера.

В следующей главе мы реализуем расчёт положения коптера в пространстве относительно карты маркеров.

Оценка и публикация положения дрона в пространстве на основании полученных координат карты маркеров

На этапе финальной интеграции нашего визуального одометра в систему управления дроном мы разработаем две ROS-ноды следующего назначения:

zuza_pose – оценка позиции карты в пространстве относительно камеры и передача этой позиции в ROS topic zuza_pose/pose

zuza_vpe – получение позиции из топика zuza_pose/pose, инициализация начального положения карты, расчёт сдвигов и публикация положения коптера в топик /mavros/vision_pose/pose

Нода zuza_pose делается с помощью модификации файла zuza_board.py: после публикации маркера карты в процедуре callback_image_raw добавим передачу позиции карты в ROS топик zuza_pose/pose:

```
## begin pose publishing ##
ps = PoseStamped()
ps.header.stamp = image_raw.header.stamp
ps.header.frame_id = 'main_camera_optical'
ps.pose.position.x, ps.pose.position.y,
ps.pose.position.z=tvec
ps.pose.orientation = Quaternion(*q)
publisher_pose.publish(ps)
```

```
## end pose publishing ##
```

При этом в основную программу нужно добавить создание публикатора publisher_pose:

```
publisher_pose=
rospy.Publisher('zuza_pose/pose', PoseStamped
,queue_size=1)
```

А также удалить все процедуры, связанные с отправкой/обработкой mavlink – сообщений (publish_dummy_vp, callback_handle_pose). Их мы перенесём в программу расчёта позиции коптера zuza_vpe.

ROS нода zuza_vpe подписывается на топик zuza_pose/pose и осуществляет подсчёт координат коптера по следующему алгоритму:

- Вычисляет позицию камеры в системе координат «мира» коптера local_origin с помощью tfBuffer.transform;
- Если видим карту в первый раз или прошёл таймаут 2 секунды – запоминаем положение карты с помощью статической трансформации tf2, публикуемой через static_map_br. Запоминается горизонтальное положение карты (фрейм aruco_map), крен, тангаж (roll, pitch) – обнуляются;
- Считаем позицию текущей карты относительно той, что запомнили (фрейма aruco_map), и результаты расчёта отправляем в полётный контроллер через топик /mavros/vision_pose/pose.

Также используем процедуры publish_dummy_vp, callback_handle_pose, рассмотренные выше, для инициализации полётного контроллера.

Ниже приведён полный код ноды zuza_vpe:

```
#!/usr/bin/env python
# coding=UTF-8
import rospy
import tf.transformations as t
from geometry_msgs.msg import Quaternion, Point,
TransformStamped#, PoseStamped#, Vector3
import math
import copy
import tf2_ros
from tf2_geometry_msgs import PoseStamped

rospy.init_node('zuza_vpe')
static_map_br=tf2_ros.StaticTransformBroadcaster()
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)
lookup_timeout = rospy.Duration(0.005)
last_published = rospy.Time(0)

def callback_aruco_map_raw(pose_raw):
    global static_map_br, last_published
    delta_x,delta_y,delta_z=0,0,0
    cur_time=rospy.get_rostime()
    ## begin reset VPE ##
    ps = PoseStamped()
    ps.header = pose_raw.header
    ps.pose = pose_raw.pose
    try:
        pose_lo =
tfBuffer.transform(ps,'local_origin',lookup_timeout)
    except tf2_ros.ConnectivityException,
tf2_ros.ExtrapolationException:
        exit()#simply wait for nexf frame
    if (last_published.to_sec()==0) or
(cur_time.to_sec()-last_published.to_sec()>2.0):
        static_transformStamped = TransformStamped()
        static_transformStamped.header =
pose_lo.header

static_transformStamped.child_frame_id="aruco_map"
        static_transformStamped.transform.translation
= pose_lo.pose.position

roll,pitch,yaw=t.euler_from_quaternion((pose_lo.pose.
orientation.x, pose_lo.pose.orientation.y,
```

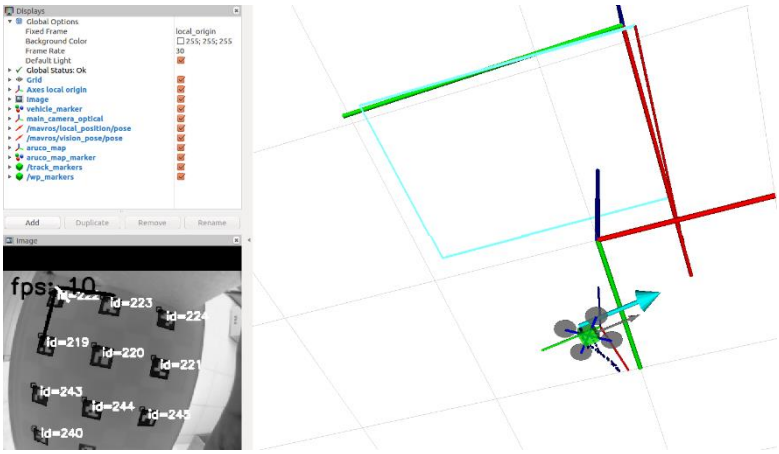
```
pose_lo.pose.orientation.z,
pose_lo.pose.orientation.w))

static_transformStamped.transform.rotation=Quaternion
(*t.quaternion_from_euler(0,0,yaw))

static_map_br.sendTransform(static_transformStamped)
        print '########### VPE reset
###########:\n\r',static_transformStamped
    # calculate offsets between static aruco_map and
aruco_map_raw
    pose_map =
tfBuffer.transform(ps,'aruco_map',lookup_timeout)
    delta_x = pose_map.pose.position.y
    delta_y = -pose_map.pose.position.x
    delta_z = -pose_map.pose.position.z
    #add fcu real Z to map difference
    tr_fcu =
tfBuffer.lookup_transform('local_origin', 'fcu',
pose_raw.header.stamp, lookup_timeout)
    delta_z+=tr_fcu.transform.translation.z
    ## begin pose publishing ##
    ps = PoseStamped()
    ps.header.stamp = pose_raw.header.stamp
    ps.header.frame_id = 'local_origin'

roll,pitch,yaw=t.euler_from_quaternion((pose_raw.pose
.orientation.x, pose_raw.pose.orientation.y,
pose_raw.pose.orientation.z,
pose_raw.pose.orientation.w))
    ps.pose.orientation =
Quaternion(*t.quaternion_from_euler(0,0,-yaw));
    ps.pose.position.x=delta_x
    ps.pose.position.y=delta_y
    ps.pose.position.z=delta_z
    publisher_vp.publish(ps);
    last_published=cur_time
    ## end pose publishing ##

def publish_dummy_vp(event):
    ps = PoseStamped()
    ps.header.stamp = rospy.Time.now()
    ps.header.frame_id = 'local_origin'
    ps.pose.orientation.w = 1;
    publisher_vp.publish(ps);
```

```
def callback_handle_pose(mavros_pose):
    global my_sub
    rospy.loginfo("Got mavros pose, stop publishing
zeroes.")
    dummy_timer.shutdown()
    handle_pose_sub.unregister()
    my_sub = rospy.Subscriber('/zuza_pose/pose',
PoseStamped, callback_aruco_map_raw)

if __name__ == '__main__':
    publisher_vp =
rospy.Publisher('/mavros/vision_pose/pose',
PoseStamped, queue_size=1)
    dummy_timer=rospy.Timer(rospy.Duration(0.5),
publish_dummy_vp)
    handle_pose_sub =
rospy.Subscriber('mavros/local_position/pose',
PoseStamped, callback_handle_pose)
    try:
      rospy.spin()
    except KeyboardInterrupt:
      print("Shutting down")
    my_sub.unregister()
```

Не забудем добавить строки запуска наших нод (ВМЕСТО запуска предыдущих нод на Питоне) в файл zuza.launch:

```
<!-- zuza pose estimator -->
<node name="zuza_pose" pkg="cam_pkg"
type="zuza_pose.py" output="screen">
<param name="markers_x" value="3"/>
<param name="markers_side" value="0.188"/>
<param name="markers_sep" value="0.376"/>
<rosparam param="marker_ids">[222, 223, 224, 219,
220, 221, 243, 244, 245, 240, 241, 242]</rosparam>
</node>

<!-- zuza vpe calculator -->
<node name="zuza_vpe" pkg="cam_pkg"
type="zuza_vpe.py" output="screen"/>
```

Запустим код на выполнение командой `roslaunch zuza zuza.launch` и проверим результат с помощью rviz:



Для полной наглядности, в окно rviz выводим:

- Grid + axes **local_origin** – система координат «мира» коптера;
- Изображение карты с обозначенными маркерами – из топика **zuza_pose/aruco_detector**;
- Изображение самого коптера – **vehicle_marker**;
- Оси системы координат камеры **main_camera_optical**;
- Позицию коптера по полётному контроллеру **mavros/local_position/pose** и по визуальной одометрии **mavros/vision_pose/pose**;
- Оси системы координат карты **aruco_map** и маркер карты **aruco_map_marker**;
- Точки траектории дрона **track_markers**.

На неподвижном коптере и карте нужно добиться, чтоб системы координат полётного контроллера (стрелка mavros/local_position/pose) и визуальной одометрии

(стрелка /mavros/vision_pose/pose) совпадали. Если координаты не совпадают – подождать, пока эти системы координат совместятся, либо заново инициализировать систему координат карты (закрыть камеру от маркеров на 2 секунды и снова открыть).

С помощью ручного перемещения коптера с камерой по всем трём осям следует убедиться, что его перемещения правильно отражаются в rviz. После успешной проверки можно переходить к выполнению тестового полёта – зависания коптера под картой маркеров.

Подготовка и выполнение тестового полёта – зависания дрона под картой маркеров

Перед первым полётом следует убедиться с помощью QGroundControl в правильной настройке параметров полётного контроллера:

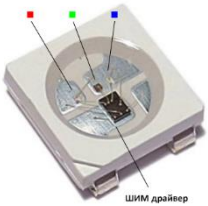
- SYS_MC_EST_GROUP = local_position_estimator
- LPE_FUSION включены только флажки vision position, landing target, land detector, остальные - отключены. Итоговое значение 28
- ATT_W_MAG = 0 – отключить компас
- ATT_EXT_HDG_M =1 – vision
- LPE_VIS_DELAY = 0 sec, LPE_VIS_XY = 0.1 m, LPE_VIS_Z = 0.3 m
- COM_DISARM_LAND = 1 s
- LNDMC_ROT_MAX = 45 deg
- LNDMC_THR_RANGE = 0.5
- LNDMC_Z_VEL_MAX = 1 m/s

Для проверки в полёте работы системы визуальной одометрии следует выполнить следующие шаги:

1. Зарядить полностью аккумуляторную батарею;

2. Поставить дрон на ровную поверхность под картой маркеров, закрыть камеру от маркеров, включить пульт управления коптером;
3. Включить питание дрона, дождаться загрузки Raspberri PI, подключиться к сети wifi дрона;
4. Показать маркеры камере и убедиться с помощью rviz, что позиции полётного контроллера (local_position) и визуальной одометрии (vision_pose) совпадают. Также можно убедиться в правильном отражении в rviz движения коптера по трём осям;
5. Заармить моторы, в ручном режиме поднять коптер на небольшую высоту, 10-20 см от пола;
6. Переключиться в полётный режим Position, стик газа перевести в среднее положение;
7. Коптер должен удерживать позицию в пространстве самостоятельно.

Раздел 6. Управление светодиодной лентой



Светодиодная лента состоит из светодиодов WS2812 (или WS2812B) с последовательной попиксельной адресацией - каждый пиксель ленты содержит светодиоды трёх цветов и ШИМ-драйвер, позволяющий управлять яркостью светодиодов.

Каждый светодиод WS2812B имеет один вход (DIN) и один выход (DO). Выход каждого светодиода подключается ко входу следующего. Подавать сигналы нужно на вход самого первого светодиода, таким образом, он запустит цепь, и данные будут поступать от первого ко второму, от второго к третьему и т. д.

Светодиоды работают на напряжении 5В, ток потребления = ~60мА/пиксель, что позволяет подключить ленту непосредственно к порту GPIO Raspberry PI. Подключение осуществляется по следующей схеме:

Лента	Raspberry PI
GND	GND
+5V	+5V
DIN	GPIO 21

Библиотеку работы с лентой из языка Python можно найти по ссылке https://github.com/rpi-ws281x/rpi-ws281x-python . Для установки библиотеки достаточно выполнить команду:

```
sudo pip install rpi_ws281x
```

В папке examples github приведён пример программы управления светодиодной лентой.

Инициализация ленты осуществляется командой:

```
strip = Adafruit_NeoPixel(LED_COUNT, LED_PIN,
LED_FREQ_HZ, LED_DMA, LED_INVERT,
LED_BRIGHTNESS, LED_CHANNEL)

strip.begin()
```

в нашем случае LED_PIN = 21, также нужно указать число пикселей в ленте в переменной LED_COUNT, остальные настройки можно оставить по умолчанию.

Значения пикселей кодируются с помощью команды:

```
strip.setPixelColor(i, color)
```

Зажигается лента командой:

```
strip.show()
```

Ниже приведён полный код программы strandtest.py:

```
#!/usr/bin/env python3
# NeoPixel library strandtest example
# Author: Tony DiCola (tony@tonydicola.com)
#
# Direct port of the Arduino NeoPixel library strandtest
example.  Showcases
# various animations on a strip of NeoPixels.

import time
from rpi_ws281x import *
import argparse

# LED strip configuration:
LED_COUNT      = 16      # Number of LED pixels.
LED_PIN        = 21      # GPIO pin connected to the pixels
(18 uses PWM!).
#LED_PIN        = 10      # GPIO pin connected to the pixels
(10 uses SPI /dev/spidev0.0).
LED_FREQ_HZ    = 800000  # LED signal frequency in hertz
(usually 800khz)
```

```
LED_DMA        = 10      # DMA channel to use for generating
signal (try 10)
LED_BRIGHTNESS = 255     # Set to 0 for darkest and 255 for
brightest
LED_INVERT     = False   # True to invert the signal (when
using NPN transistor level shift)
LED_CHANNEL    = 0       # set to '1' for GPIOs 13, 19, 41,
45 or 53



# Define functions which animate LEDs in various ways.
def colorWipe(strip, color, wait_ms=50):
    """Wipe color across display a pixel at a time."""
    for i in range(strip.numPixels()):
        strip.setPixelColor(i, color)
        strip.show()
        time.sleep(wait_ms/1000.0)

def theaterChase(strip, color, wait_ms=50, iterations=10):
    """Movie theater light style chaser animation."""
    for j in range(iterations):
        for q in range(3):
            for i in range(0, strip.numPixels(), 3):
                strip.setPixelColor(i+q, color)
            strip.show()
            time.sleep(wait_ms/1000.0)
            for i in range(0, strip.numPixels(), 3):
                strip.setPixelColor(i+q, 0)

def wheel(pos):
    """Generate rainbow colors across 0-255 positions."""
    if pos < 85:
        return Color(pos * 3, 255 - pos * 3, 0)
    elif pos < 170:
        pos -= 85
        return Color(255 - pos * 3, 0, pos * 3)
    else:
        pos -= 170
        return Color(0, pos * 3, 255 - pos * 3)

def rainbow(strip, wait_ms=20, iterations=1):
    """Draw rainbow that fades across all pixels at once."""
    for j in range(256*iterations):
        for i in range(strip.numPixels()):
            strip.setPixelColor(i, wheel((i+j) & 255))
        strip.show()
        time.sleep(wait_ms/1000.0)

def rainbowCycle(strip, wait_ms=20, iterations=5):
    """Draw rainbow that uniformly distributes itself across
all pixels."""
```

```
    for j in range(256*iterations):
        for i in range(strip.numPixels()):
            strip.setPixelColor(i, wheel((int(i * 256 /
strip.numPixels()) + j) & 255))
        strip.show()
        time.sleep(wait_ms/1000.0)

def theaterChaseRainbow(strip, wait_ms=50):
    """Rainbow movie theater light style chaser
animation."""
    for j in range(256):
        for q in range(3):
            for i in range(0, strip.numPixels(), 3):
                strip.setPixelColor(i+q, wheel((i+j) % 255))
            strip.show()
            time.sleep(wait_ms/1000.0)
            for i in range(0, strip.numPixels(), 3):
                strip.setPixelColor(i+q, 0)

# Main program logic follows:
if __name__ == '__main__':
    # Process arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--clear',
action='store_true', help='clear the display on exit')
    args = parser.parse_args()

    # Create NeoPixel object with appropriate configuration.
    strip = Adafruit_NeoPixel(LED_COUNT, LED_PIN,
LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_BRIGHTNESS,
LED_CHANNEL)
    # Intialize the library (must be called once before
other functions).
    strip.begin()

    print ('Press Ctrl-C to quit.')
    if not args.clear:
        print('Use "-c" argument to clear LEDs on exit')

    try:

        while True:
            print ('Color wipe animations.')
            colorWipe(strip, Color(255, 0, 0))  # Red wipe
            colorWipe(strip, Color(0, 255, 0))  # Blue wipe
            colorWipe(strip, Color(0, 0, 255))  # Green wipe
            print ('Theater chase animations.')
            theaterChase(strip, Color(127, 127, 127))  #
White theater chase
            theaterChase(strip, Color(127,   0,   0))  # Red
theater chase
```

```
            theaterChase(strip, Color(  0,   0, 127))  #
Blue theater chase
            print ('Rainbow animations.')
            rainbow(strip)
            rainbowCycle(strip)
            theaterChaseRainbow(strip)

    except KeyboardInterrupt:
        if args.clear:
            colorWipe(strip, Color(0,0,0), 10)
```

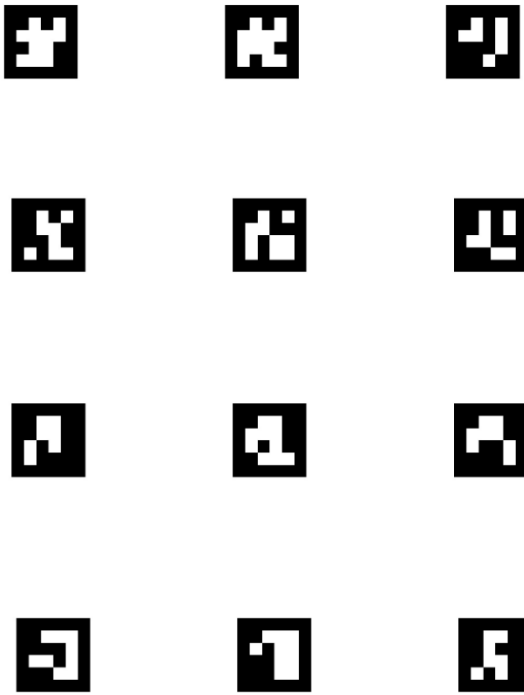
Запуск программы должен осуществляться с от суперпользователя командой sudo python strandtest.py.

Запуск от суперпользователя необходим для доступа к устройству /dev/mem, что подтвердили разработчики модуля (https://github.com/rpi-ws281x/rpi-ws281x-python/issues/9).

Раздел 7. Автономный полёт с помощью пакета clever

Пакет clever – набор программ с открытым исходным кодом, предназначенный для автономной навигации БПЛА с помощью карты Aruco меток. Исходные коды программ пакета clever доступны по ссылке https://github.com/CopterExpress/clever

Для реализации автономных полётов нужно изготовить и разместить карту Aruco меток, на полу или на потолке помещения. Пример карты, используемой по умолчанию при тестировании конструктора Жужа приведён ниже:



С изображения маркеров возможно сгенерировать самостоятельно, любо взять из фала в составе ПО комплекта Жужа.

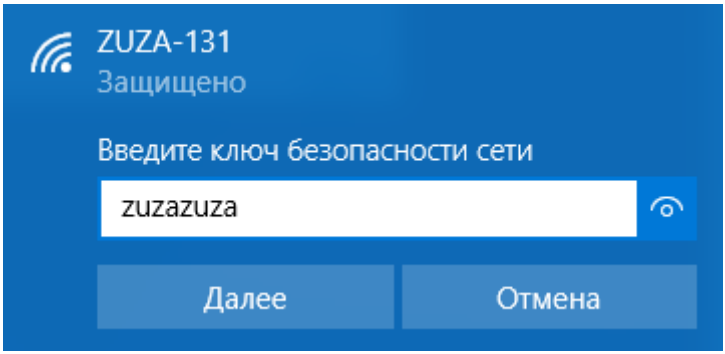
Подготовка к первому полёту состоит из следующих шагов:

1. Зарядить полностью аккумуляторную батарею.

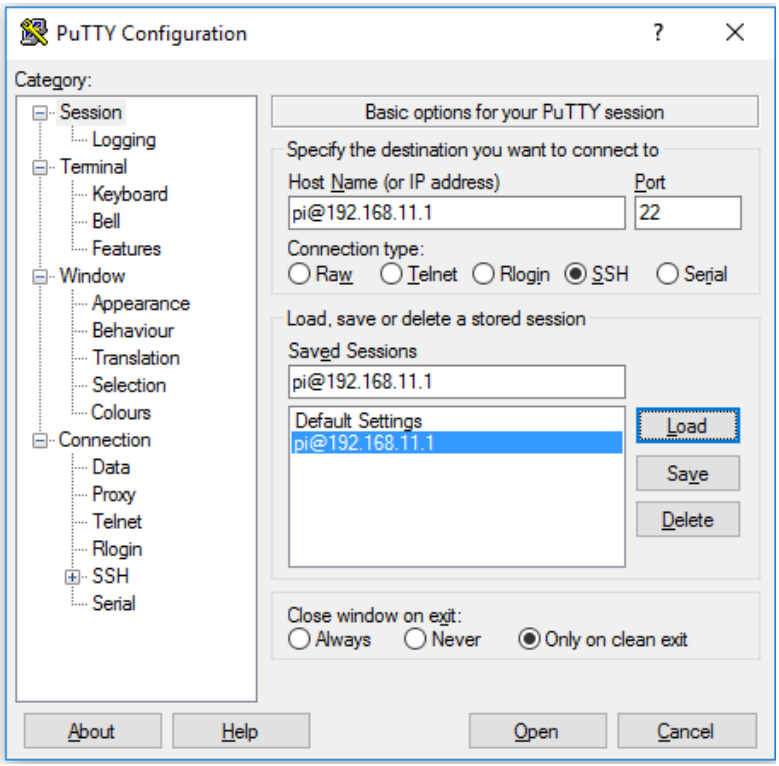
Полёт с нагрузкой в виде бортового компьютера, модуля захвата груза ведёт к повышенному расходу заряда АКБ. Перед полётом необходимо убедиться, что АКБ полностью заряжена.

2. Проверить связь полётного контроллера с бортовым компьютером.

Включить дрон, подключиться к сети WiFi бортового компьютера Raspberry PI:



Подключиться к бортовому компьютеру по SSH:



Выполнить команду:

```
rostopic echo /mavros/state
```

В результате вывода команды в консоль должны выводиться сообщения о состоянии полётного контроллера, примерно раз в секунду, вида:

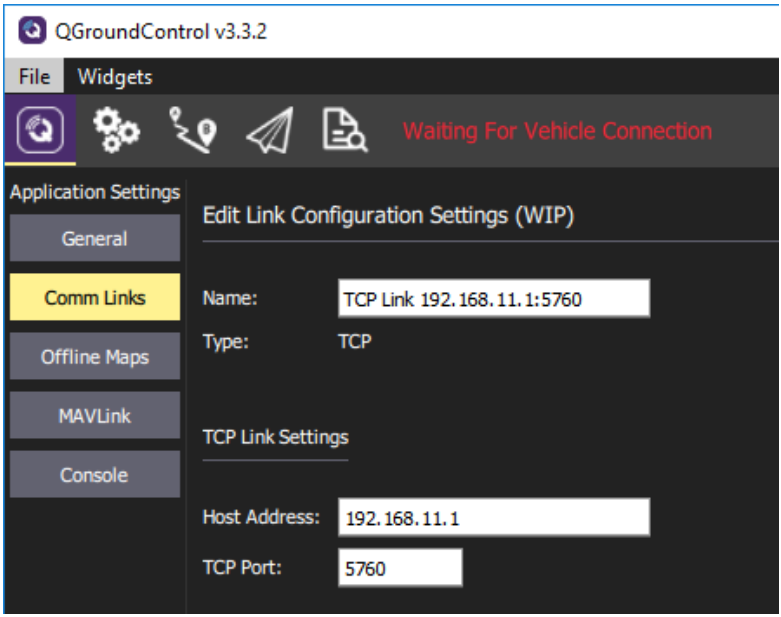
```
header:
  seq: 117
  stamp:
    secs: 1511970692
    nsecs: 604845363
  frame_id: ''
```

```
connected: True
armed: False
guided: False
mode: "MANUAL"
system_status: 3
---
```

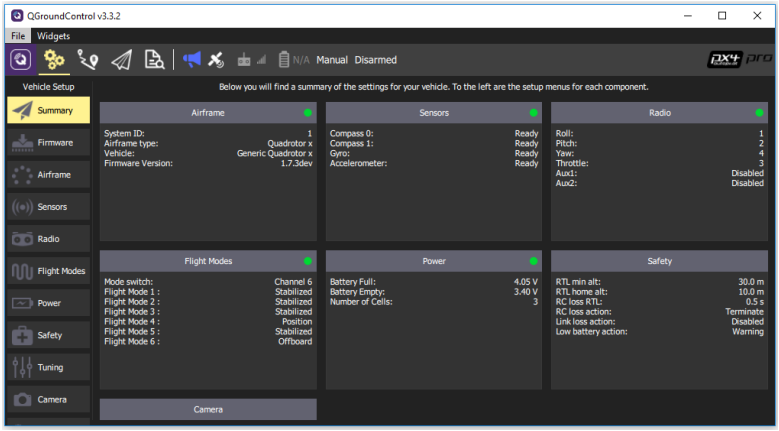
Поле `connected: True` говорит о том, что связь бортового компьютера с полётным контроллером установлена.

3. Запустить и подключить к дрону программу управления с наземной станции QGroundControl.

Подключение QGroundControl производится с помощью механизма Comm Link, параметры подключения (в стандартной настройке):



После установки соединения по кнопке Connect на экране отразятся текущие настройки дрона:



После этого с помощью QGroundControl можно проверить функционирование переключателя аварийного отключения с пульта радиоуправления (англ. killswitch). При активации/деактивации аварийного отключения QGroundControl извещает пользователя сообщением «Manual killswitch engaged» и «Manual killswitch off», соответственно.

4. Проверить настройки поля меток с помощью RVIZ.

Для подключения RVIZ к ROS-мастеру бортового компьютера дрона необходимо установить переменные среды $ROS_MASTER_URI и $ROS_IP вручную или с помощью скрипта:

```
export ROS_MASTER_URI=http://192.168.11.1:11311
export ROS_IP=10.0.2.15
echo 'ROS_MASTER_URI='$ROS_MASTER_URI
echo 'ROS_IP='$ROS_IP
```

В значение ROS_IP нужно проставить ip-адрес компьютера, на котором запускается rviz. Узнать его

можно с помощью команды ifconfig. Запуск скрипта нужно выполнять командой `source <имя файла скрипта>`.

После установки переменных можно проверить хартбит полётного контроллера с помощью команды rostopic echo /mavros/state. Если хартбит появился – запускаем визуализатор с помощью команды rviz.

В rviz рекомендуется настроить визуализацию следующих настроек и топиков:

Global Options/Fixed Frame = local_origin

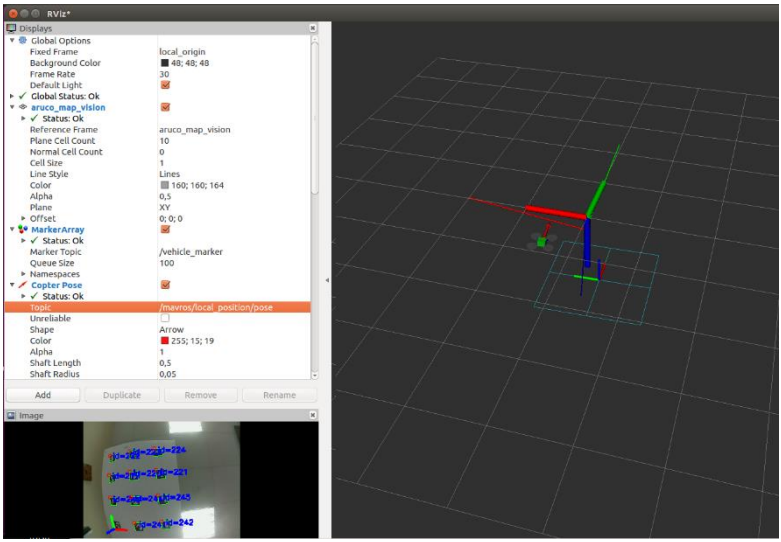
/vehicle_marker – изображение виртуального коптера

/mavros/local_position/pose – положение коптера в пространстве

aruco_map_vision – положение карты меток в пространстве

/simple_offboard/target – целевое положение коптера в пространстве

/aruco_pose/debug – изображение карты меток с распознанными маркерами



Перемещая коптер по трём осям (вверх/вниз, вправо/влево, вперёд/назад) нужно убедиться, что виртуальный коптер в RVIZ перемещается в соответствующем направлении, и дрон стабильно видит метки и правильно определяет своё положение в пространстве с их помощью.

Если метки видятся не стабильно – нужно убедиться, что фокус камеры настроен правильно и камера получает чёткое изображение меток.

Если направления перемещения виртуального дрона не соответствуют перемещениям реального – нужно убедиться в правильности настройки карты меток (файл aruco.launch) и настроек расположения камеры (файл main_camera.launch). В случае изменения настроек пакет clever нужно перезапустить с помощью команды `sudo systemctl restart clever`.

Если дрон определяет своё положение в пространстве правильно – можно выполнить тестовый полёт.

5. Выполнить тестовый полёт: взлёт, зависание и посадка.

Тестовый полёт можно выполнить с помощью программы run_land5_r.py, листинг которой приведён ниже. Программа настроена на полёт по потолочным меткам. Полёт управляется с помощь сервиса navigate, текущее положение в пространстве определяется с помощью сервиса get_telemetry модуля simple_offboard пакета clever.

```
from __future__ import print_function

import rospy
from clever import srv
from mavros_msgs.srv import CommandBool
from mavros_msgs.srv import SetMode

rospy.init_node('testnode')

navigate = rospy.ServiceProxy('/navigate',
srv.Navigate)
set_mode =
rospy.ServiceProxy('/mavros/set_mode', SetMode)
get_telemetry =
rospy.ServiceProxy('/get_telemetry',
srv.GetTelemetry)
arming =
rospy.ServiceProxy('/mavros/cmd/arming',
CommandBool)


# Perform land
def land():
```

```
    navigate(x=0, y=0, z=-10, speed=1.3,
frame_id='fcu_horiz')
    rospy.sleep(3)
    mode_to("AUTO.LAND")
    rospy.sleep(3)
    arming(False)
    print('-----LANDED-----')

# Perform take off
def take_off(h, speed=1.0, delta=0.2):
    navigate(x=0, y=0, z=h, speed=speed,
frame_id='fcu_horiz', update_frame=False,
auto_arm=True)
    rospy.sleep(4)
    print('\tReached', h)

print('#### Program start ####')
# main program

frame_speed = 4
delta_pre = 0.08
delta_after = 0.1

take_off(1.0, 0.5)
rospy.sleep(5)

#print('#### Navigate y=0.5 ####')
#navigate(x=0, y=0.5, z=0, speed=0.5,
frame_id='fcu_horiz', update_frame=False,
auto_arm=False)
#rospy.sleep(4)
my_x=1.0
my_y=0.5
my_z=1
my_yaw=0

print('#### Navigate aruco map x=',my_x,'
y=',my_y,' z=',my_z,' yaw=',my_yaw,' ####\n')
navigate(x=my_x, y=my_y, z=my_z, yaw=my_yaw,
speed=0.5, frame_id='aruco_map_vision',
update_frame=True, auto_arm=False)
rospy.sleep(1)
```

```
# keyboard manipulation
import curses
stdscr = curses.initscr()
curses.noecho()
stdscr.nodelay(1)
stdscr.keypad(1)

while (rospy.is_shutdown()==False):
      # keyboard  hcommands handling
      c = stdscr.getch()
      if c == ord('q'): break  # Exit the
while()
      elif c == ord('u'): my_z += 0.2
      elif c == ord('d'): my_z -= 0.2
      elif c == ord('l'): my_yaw += 0.2
      elif c == ord('r'): my_yaw -= 0.2
      elif c == curses.KEY_LEFT:my_x += 0.2
      elif c == curses.KEY_RIGHT:my_x -= 0.2
      elif c == curses.KEY_UP:my_y += 0.2
      elif c == curses.KEY_DOWN:my_y -= 0.2

      if  c!=curses.ERR:
            print('#### Navigate aruco map
x=',my_x,' y=',my_y,' z=',my_z,' yaw=',my_yaw,
'####\r')
            navigate(x=my_x, y=my_y, z=my_z,
yaw=my_yaw, speed=0.5,
frame_id='aruco_map_vision', update_frame=True,
auto_arm=False)
            #rospy.sleep(1)

curses.endwin()

land()
```

Программа запускается в рабочем каталоге пользователя pi командой `python run_land5_r.py`.

После запуска программы дрон осуществляет взлёт на высоту 1м и зависает на месте. После этого дроном можно управлять с помощью команд клавиатуры консоли:

клавиши вперёд/назад/вправо/влево – перемещение дрона в соответствующем направлении

клавиши U – движение вверх, **D** – движение вниз, **L** – поворот налево, **R** – поворот направо, **Q** – выход из программы и посадка дрона.

При первых запусках программы следует быть внимательным, при потере дроном ориентации в пространстве рекомендуется поймать его за шасси руками, после чего выключить двигатели с помощью переключателя аварийного отключения с радиопульта.

На примере приведённой программы преподаватели и студенты могут разработать свои программы автономных миссий.

Команда copter.space желает Вам успешных полётов!