Flying Multiple UAVs Using ROS

Wolfgang Hönig and Nora Ayanian

Department of Computer Science,
University of Southern California, Los Angeles, CA, USA
whoenig@usc.edu
ayanian@usc.edu
http://act.usc.edu

Abstract. This tutorial chapter will teach readers how to use ROS to fly a small quadcopter both individually and as a group. We will discuss the hardware platform, the Bitcraze Crazyflie 2.0, which is well suited for swarm robotics due to its small size and weight. After first introducing the `crazyflie_ros` stack and its use on an individual robot, we will extend scenarios of hovering and waypoint following from a single robot to the more complex multi-UAV case. Readers will gain insight into physical challenges, such as radio interference, and how to solve them in practice. Ultimately, this chapter will prepare readers not only to use the stack as-is, but also to extend it or to develop their own innovations on other robot platforms.

Keywords: ROS, UAV, Multi-Robot-System, Crazyflie, Swarm

1 Introduction

Unmanned aerial vehicles (UAVs) such as AscTec Pelican, Parrot AR.Drone, and Erle-Copter have a long tradition of being controlled with ROS. As a result, there are many ROS packages devoted to controlling such UAVs as individuals[1]. However, using multiple UAVs creates entirely new challenges that such packages cannot address, including, but not limited to, the physical space required to operate the robots, the interference of sensors and network communication, and safety requirements.

Multiple UAVs have been used in recent research [1,2,3,4,5], but such research can be overly complicated and tedious due to the lack of tutorials and books. In fact, even with packages that can support multiple UAVs, documentation focuses on the single UAV case, not considering the challenges that occur once multiple UAVs are used. Research publications often skip implementation details, making it difficult to replicate the results. Papers about specialized setups exist [6,7], but rely on expensive or commercially unavailable solutions.

This chapter will attempt to fill this gap in documentation. In particular, we try to provide a step-by-step guide on how to reproduce results we presented

[1] e.g. `http://wiki.ros.org/ardrone_autonomy`, `http://wiki.ros.org/mavros`, `wiki.ros.org/asctec_mav_framework`

in an earlier research paper [3], which used up to six UAVs[2]. We focus on a small quadcopter — the Bitcraze Crazyflie 2.0 — and how to use it with the `crazyflie_ros` stack, particularly as part of a group of 2 or more UAVs. We will assume that an external position tracking system, such as a motion capture system, is available because the Crazyflie is not able to localize itself with just onboard sensing. We will discuss the physical setup and how to support a single human pilot. Each step will start with the single UAV case and then extend to the more challenging multi-UAV case.

We begin with an introduction to the target platform, including the software setup of the vendor's software and the `crazyflie_ros` stack. We then show teleoperation of multiple Crazyflies using joysticks. The usage of a motion capture system allows us to autonomously hover multiple Crazyflies. We then extend this to multiple UAVs following waypoints. The chapter will also contain important insights into the `crazyflie_ros` stack, allowing the user to understand the design in-depth. This can be helpful for users interested in implementing other multi-UAV projects using different hardware or adding extensions to the existing stack.

Everything discussed here has been tested on Ubuntu 14.04 using ROS Indigo. The stack and discussed software also work with ROS Jade (Ubuntu 14.04) and ROS Kinetic (Ubuntu 16.04).

2 Target Platform

Fig. 1. Our target platform Bitcraze Crazyflie 2.0 quadcopter (left), which can be controlled from a PC using a custom USB dongle called Crazyradio PA (right). Image credit: Bitcraze AB.

As our target platform we use the Bitcraze Crazyflie 2.0 platform, an open-source, open-hardware nano quadcopter that targets hobbyists and researchers alike. Its small size (92 mm diagonal rotor-to-rotor) and weight (29 g) make it ideal for indoor swarming applications. Additionally, its size allows users to operate the UAVs safely even with humans or other robots around. The low inertia causes only few parts to break after a crash — the authors had several crashes

[2] Video available at `http://youtu.be/px9iHkA0nOI`

from a height of 3 m to a concrete floor with damage only to cheaply replaceable plastic parts. A Crazyflie can communicate with a phone or PC using Bluetooth. Additionally, a custom USB dongle called Crazyradio PA, or Crazyradio for short, allows lower latency communication. The Crazyflie 2.0 and Crazyradio PA are shown in Fig. 1.

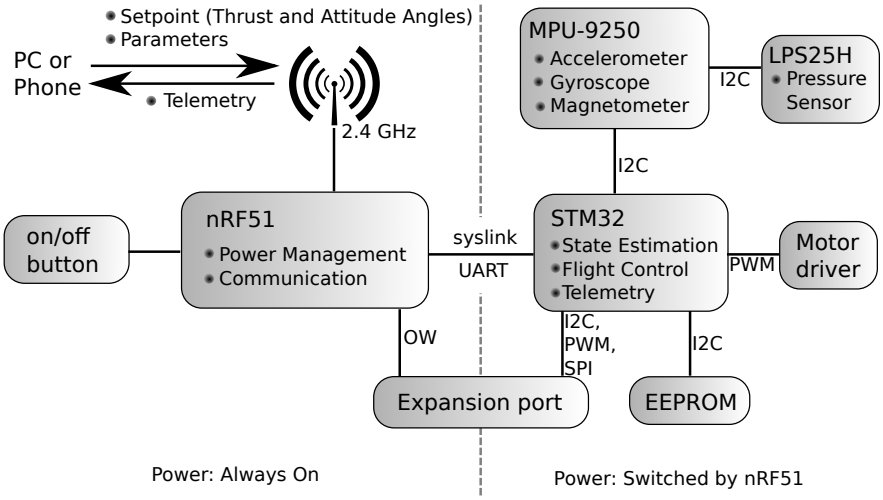

Fig. 2. Components and architecture of the Crazyflie 2.0 quadcopter. Based on images by Bitcraze AB.

A block diagram of the Crazyflie's architecture is shown in Fig 2. The communication system is used to send the setpoint, consisting of thrust and attitude, tweak internal parameters, and stream telemetry data, such as sensor readings. It is also possible to update the onboard software wirelessly. The Crazyflie has a 9-axis inertial measurement unit (IMU) onboard, consisting of gyroscope, accelerometer, and magnetometer. Moreover, a pressure sensor can be used to estimate the height. Most of the processing is done on the main microcontroller (STM32). It runs FreeRTOS as its operating system and state estimation and attitude control are executed at 250 Hz. A second microcontroller (nRF51) is used for the wireless communication and as a power manager. The two microcontrollers can exchange data over the `syslink`, which is a protocol using UART as a physical interface. An extension port permits the addition of additional hardware. The official extensions include an inductive charger, LED headlights, and buzzer. Finally, it is possible to use the platform on a bigger frame if higher payload capabilities are desired. Extensions are called "decks" and are also used

by the community to add additional capabilities[3]. The schematics as well as all firmwares are publicly available[4]. The technical specifications are as follows:

- STM32F405: main microcontroller, used for state-estimation, control, and handling of extensions. We will call this STM32.
  (Cortex-M4, 168 MHz, 192 kB SRAM, 1 MB flash).
- nRF51822: radio and power management microcontroller. We will call this nRF51.
  (Cortex-M0, 32 MHz, 16 kB SRAM, 128 kB flash).
- MPU-9250: 9-axis inertial measurement unit.
- LPS25H: pressure sensor.
- 8 kB EEPROM.
- uUSB: charging and wired communication.
- Expansion port (I2C, UART, SPI, GPIO).
- Debug port for STM32. An optional debug-kit can be used to convert to a standard JTAG-connector and to debug the nRF51 as well.

The onboard sensors are sufficient to stabilize the attitude, but not the position. In particular, external feedback is required to fly to predefined positions. By default, this is the human who teleoperates the quadcopter either using a joystick connected to a PC, or a phone. In this chapter, we will use a motion-capture system for fully autonomous flights.

The vendor provides an SDK written in Python which runs on Windows, Linux, and Mac. It can be used to teleoperate a single Crazyflie using a joystick, to plot sensor data in real-time, and to write custom applications. We will use ROS in the remainder of this chapter to control the Crazyflie; however, ROS is only used on the PC controlling one or more Crazyflies. The ROS driver sends the data to the different quadcopters using the protocol defined in the Crazyflie firmware.

The Crazyflie has been featured in a number of research papers. The mathematical model and system identification of important parameters, such as the inertia matrix, have been discussed in [8] and [9]. An updated list with applications can be found on the official webpage[5].

3 Setup

In this section we will describe how to set up the Crazyflie software. We cover both the official Python SDK and how to install the `crazyflie_ros` stack. The first is useful to reconfigure the Crazyflie as well as for troubleshooting, while the later will allow us to use multiple Crazyflies with ROS.

We assume a PC with Ubuntu 14.04 as operating system, which has ROS Indigo (desktop-full) installed[6]. It is better to install Ubuntu directly on a PC

[3] `https://www.hackster.io/bitcraze/products/crazyflie-2-0`
[4] `https://github.com/bitcraze/`
[5] `https://www.bitcraze.io/research/`
[6] `http://wiki.ros.org/indigo/Installation/Ubuntu`

rather than using a virtual machine for two reasons: First, you will be using graphical tools, such as `rviz`, which rely on OpenGL and therefore do not perform as well on a virtual machine as when natively installed. Second, the communication using the Crazyradio would have additional latency in a virtual machine since the USB signals would go through the host system first. This might cause less stable control.

In particular, we will follow the following steps:

1. Configure the PC such that the Crazyradio will work for any user.
2. Install the official software package to test the Crazyflie.
3. Update Crazyflie's onboard software to the latest version to ensure that it will work with the ROS package.
4. Install the `crazyflie_ros` package and run a first simple connection test.

The later sections in this chapter assume that everything is set up as outlined here to perform higher-level tasks.

3.1 Setting PC Permissions

By default, the Crazyradio will only work for a user with superuser rights when plugged in to a PC. This is not only a security concern but also makes it harder to use with ROS. In order to use it without `sudo`, we first add a group (`plugdev`) and then add ourselves as a member of that group:

```
$ sudo groupadd plugdev
$ sudo usermod -a -G plugdev $USER
```

Now, we create a udev-rule, setting the permission such that anyone who is a member of our newly created group can access the Crazyradio. We create a new rules file using `gedit`:

```
$ sudo gedit /etc/udev/rules.d/99-crazyradio.rules
```

and add the following text to it:

```
1 # Crazyradio (normal operation)
2 SUBSYSTEM=="usb", ATTRS{idVendor}=="1915",
      ATTRS{idProduct}=="7777", MODE="0664", GROUP="plugdev"
3 # Bootloader
4 SUBSYSTEM=="usb", ATTRS{idVendor}=="1915",
      ATTRS{idProduct}=="0101", MODE="0664", GROUP="plugdev"
```

The second entry is useful for firmware updates of the Crazyradio.

In order to use the Crazyflie when directly connected via USB, you need to create another file named `99-crazyflie.rules` in the same folder, with the following content:

```
1 SUBSYSTEM=="usb", ATTRS{idVendor}=="0483",
      ATTRS{idProduct}=="5740", MODE="0664", GROUP="plugdev"
```

Finally, we reload the udev-rules:

```
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

You will need to log out and log in again in order to be a member of the `plugdev` group. You can then plug in your Crazyradio (and follow the instructions in the next section to actually use it).

3.2 Bitcraze Crazyflie PC Client

The Bitcraze SDK is composed of two parts. The first is `crazyflie-lib-python`, which is a Python library to control the Crazyflie without any graphical user interface. The second is `crazyflie-client-python`, which makes use of that library and adds a graphical user interface.

We start by installing the required dependencies:

```
$ sudo apt-get install git python3 python3-pip python3-pyqt4
    python3-numpy python3-zmq
$ sudo pip3 install pyusb==1.0.0b2
$ sudo pip3 install pyqtgraph appdirs
```

To install `crazyflie-lib-python`, use the following commands:

```
$ mkdir ~/crazyflie
$ cd ~/crazyflie
$ git clone https://github.com/bitcraze/crazyflie-lib-python.git
$ cd crazyflie-lib-python
$ pip3 install --user -e .
```

Here, the Python package manager `pip` is used to install the library only for the current user. The library uses Python 3. In contrast, ROS Indigo, Jade, and Kinetic use Python 2.

Similarly, `crazyflie-client-python` can be installed using the following commands:

```
$ cd ~/crazyflie
$ git clone https://github.com/bitcraze/crazyflie-clients-python.git
$ cd crazyflie-clients-python
$ pip3 install --user -e .
```

To start the client, execute the following:

```
$ cd ~/crazyflie/crazyflie-clients-python
$ python3 bin/cfclient
```

You should see the graphical user interface, as shown in Fig. 3.

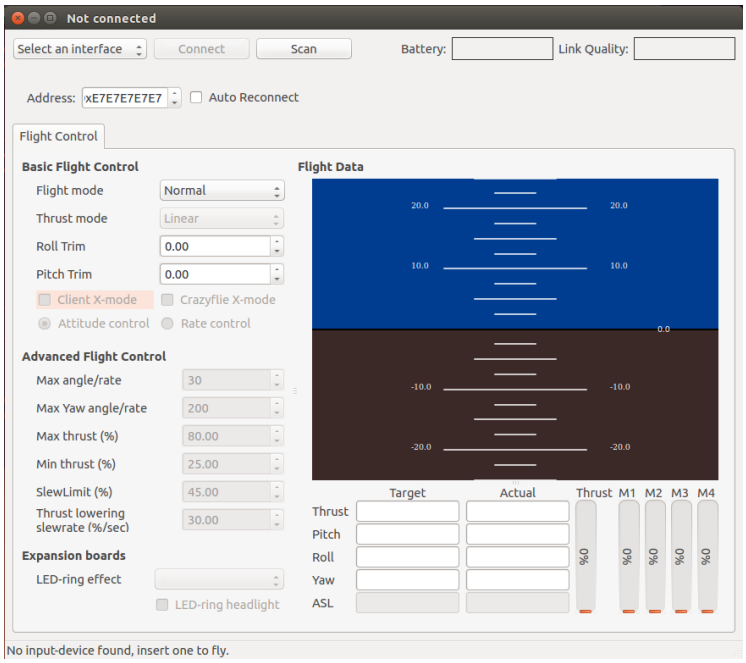

Fig. 3. Screenshot of the Bitcraze Crazyflie PC Client

Versions Might Change

Since the Crazyflie software is under active development, the installation procedure and required dependencies might change in the future. You can use the exact same versions as used in the chapter by using the following commands after `git clone`. Use the following for `crazyflie-lib-python`

```
$ git checkout a0397675376a57adf4e7c911f43df885a45690d1
```

and use the following for `crazyflie-clients-python`:

```
$ git checkout 2dff614df756f1e814538fbe78fe7929779a9846
```

If you want to use the latest version please follow the instructions provided in the `README.md` file in the respective repositories.

3.3 Firmware

Everything described in this chapter works with the Crazyflie's default firmware. You can obtain the latest compiled firmware from the repository[7] — this chapter was tested with the 2016.02 release. Make sure that you update the firmware for both STM32 and nRF51 chips by downloading the zip-file. Execute the following steps to update both firmwares:

1. Start the Bitcraze Crazyflie PC Client.
2. In the menu select "Connect"/"Bootloader."
3. Turn your Crazyflie off by pressing the power button. Turn it back on by pressing the power button for 3 seconds. The blue tail lights should start blinking: The Crazyflie is now waiting for a new firmware.
4. Click "Initiate bootloader cold boot." The status should switch to "Connected to bootloader."
5. Select the downloaded `crazyflie-2016.02.zip` and press "Program." Click the "Restart in firmware mode" button after it is finished.

If you prefer compiling the firmware yourself, please follow the instructions in the respective repositories[8].

3.4 Crazyflie ROS Stack

The `crazyflie_ros` stack contains the driver, a position controller, and various examples. We will explore the different possibilities later in this chapter and concentrate on the initial setup first.

We first create a new ROS workspace:

```
$ mkdir -p ~/crazyflie_ws/src
$ cd ~/crazyflie_ws/src
$ catkin_init_workspace
```

Next, we add the required packages to the workspace and build them:

```
$ git clone https://github.com/whoenig/crazyflie_ros.git
$ cd ~/crazyflie_ws
$ catkin_make
```

In order to use your workspace add the following line to your `~/.bashrc`:

[7] `https://github.com/bitcraze/crazyflie-release/releases`
[8] `https://github.com/bitcraze/crazyflie-firmware`, `https://github.com/bitcraze/crazyflie2-nrf-firmware`

```
$ source ~/crazyflie_ws/devel/setup.bash
```

This will ensure that all ROS related commands will find the packages in all terminals. To update your current terminal window, use `source ~/.bashrc`, which will reload the file.

You can test your setup by typing:

```
$ rosrun crazyflie_tools scan
```

This should print the *uniform-resource-identifier* (URI) of any Crazyflie found in range. For example, the output might look like this:

```
Configured Dongle with version 0.54
radio://0/100/2M
```

In this case, the URI of your Crazyflie is `radio://0/100/2M`. Each URI has several components. Here, the Crazyradio is used (*radio*). Since you might have multiple radios in use, you can specify a zero-based index on the device to use (*0*). The next number (*100*) specifies the channel, which is a number between 0 and 125. Finally, the datarate (*2M*) (one of 250K, 1M, 2M) specifies the speed to use in bits per second. There is an optional address as well, which we will discuss in section 5.1.

Versions

As before, the instructions might be different in future versions. Use the following to get the exact same version of the `crazyflie_ros` stack:

```
$ git checkout 34beecd2a8d7ab02378bcdfcb9adf5a7a0eb50ea
```

Install the following additional dependency in order to use the teleoperation:

```
$ sudo apt-get install ros-indigo-hector-quadrotor-teleop
```

If you are using ROS Jade or Kinetic, you will need to add the package to your workspace manually.

4 Teleoperation of A Single Quadcopter

In this section we will use ROS to control a single Crazyflie using a joystick. Moreover, we will gain access to the internal sensors and show how to visualize the data using `rviz` and `rqt_plot`.

This is a useful first step to understand the `cmd_vel` interface of the `crazyflie_ros` stack. Later, we will build on this knowledge to let the Crazyflie fly autonomously. Furthermore, teleoperation is useful for debugging. For example, it can be used to verify that there is no mechanical hardware defect.

In the first subsection, we assume that you have access to a specific joystick, the Microsoft XBox360 controller. We show how to connect to the Crazyflie using ROS and how to eventually fly it manually. The second subsection relaxes this assumption by discussing the required steps needed to add support for another joystick.

4.1 Using an XBox360 Controller

For this example, we assume that you have an Xbox360 controller plugged into your machine. We will show how to use different joysticks later in this section. Use the following command to run the teleoperation example:

```
$ roslaunch crazyflie_demo teleop_xbox360.launch uri:=radio://0/100/2M
```

Make sure that you adjust the URI based on your Crazyflie.

The launch file `teleop_xbox360.launch` has the following structure:

teleop_xbox360.launch

```
1  <launch>
2    <arg name="uri" default="radio://0/80/2M" />
3    <arg name="joy_dev" default="/dev/input/js0" />
4    <include file="$(find
         crazyflie_driver)/launch/crazyflie_server.launch" />
5    <group ns="crazyflie">
6      <include file="$(find
           crazyflie_driver)/launch/crazyflie_add.launch">
7        <arg name="uri" value="$(arg uri)" />
8        <arg name="tf_prefix" value="crazyflie" />
9        <arg name="enable_logging" value="True" />
10     </include>
11     <node name="joy" pkg="joy" type="joy_node" output="screen" >
12       <param name="dev" value="$(arg joy_dev)" />
13     </node>
14     <include file="$(find crazyflie_demo)/launch/xbox360.launch" />
15     <node name="crazyflie_demo_controller" pkg="crazyflie_demo"
           type="controller.py" output="screen" />
16   </group>
17   <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
         crazyflie_demo)/launch/crazyflie.rviz" />
18   <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot1"
         args="/crazyflie/battery"/>
19   <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot2"
         args="/crazyflie/rssi"/>
20 </launch>
```

In line 4 the `crazyflie_server` is launched, which accesses the Crazyradio to communicate with the Crazyflie. Lines 5 to 16 contain information about

the Crazyflie we want to control. First, the Crazyflie is added with a specified URI. Second, the `joy_node` is launched to create the `joy` topic. This particular joystick is configured by including `xbox360.launch`. This file will launch a `hector_quadcopter_teleop` node with the appropriate settings for the XBox360 controller. Furthermore, `controller.py` is started; this maps additional joystick buttons to Crazyflie specific behaviors. For example, the red button on your controller will cause the Crazyflie to turn off all propellers (emergency mode). Finally, lines 17 to 19 start `rviz` and two instances of `rqt_plot` for visualization. Figure 4 shows a screenshot of `rviz` as it visualizes the data from the inertial measurement unit as streamed from the Crazyflie at 100 Hz. The other two `rqt_plot` instances show the current battery voltage and radio signal strength indicator (RSSI), respectively.

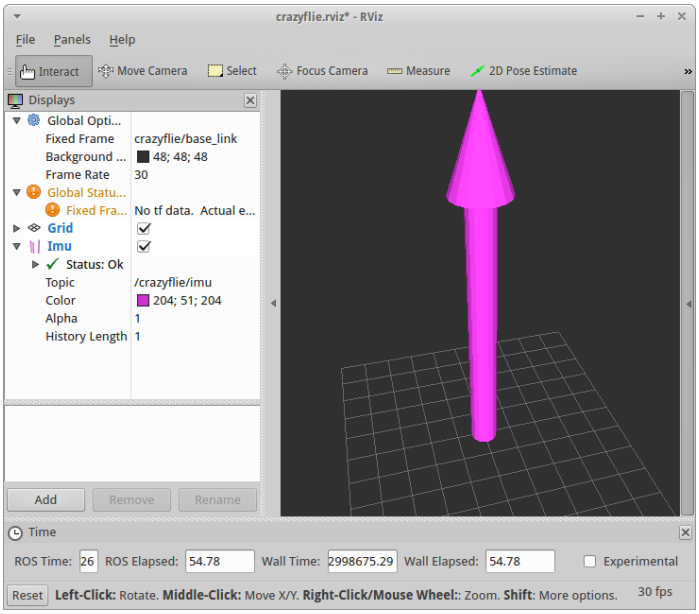

Fig. 4. Screenshot of `rviz` showing the IMU data.

If you tilt your Crazyflie, you should instantly see the IMU arrow changing in `rviz`. You can now use the joystick to fly — the default uses the left stick for thrust (up/down) and yaw (left/right) and the right stick for pitch (up/down) and roll (left/right). Also, the red B-button can be used to put the ROS driver in emergency mode. In that case, your Crazyflie will immediately turn off its engines (and if it was flying it will fall to the ground).

4.2 Add Support for Another Controller

Support for another joystick can be easily added, as long as it is recognized as a joystick by the operating system. The major difference between joysticks is the mapping between the different axes and buttons of a joystick to the desired functionality. In the following steps we first try to find the desired mapping and use that to configure the `crazyflie_ros` stack accordingly.

1. Attach your joystick. This will create a new device file, e.g. `/dev/input/js0`. You can use `dmesg` to find details about which device file was used in the system log.
2. Run the following command in order to execute the `joy_node`:

   ```
   $ rosrun joy joy_node _dev:=/dev/input/js0
   ```

3. In another terminal, execute:

   ```
   $ rostopic echo /joy
   ```

   This will print the joystick messages published by `joy_node`. Move your joystick to find the desired axes mapping. For example, you might increase the thrust on your joystick and see that the second number of the `axes` array decreases.
4. Change the axis mapping in `xbox360.launch` (or create a new file) by updating parameters `x_axis`, `y_axis`, `z_axis`, and `yaw_axis` accordingly. You can use negative axis values to indicate that this axis should be inverted. For example, in the previous example for the thrust changes, you would choose -2 as axis for `z_axis`.
5. Update the button mapping in `crazyflie_demo/scripts/controller.py` to change which button triggers high-level behavior such as emergency.

The PS3 controller is already part of the `crazyflie_ros` stack and the mapping was found in the same way as described above.

5 Teleoperation of Multiple UAVs

This section discusses the initial setup: how to assign unique addresses to each UAV, how to communicate using fewer radios than UAVs, and how to find good communication channels to decrease interference between UAVs as well as between UAVs and existing infrastructure such as WiFi.

Flying multiple Crazyflies is mainly limited by the communication bandwidth. One way to handle this issue is to have one Crazyradio per Crazyflie and to use a different channel for each of them. There are two major disadvantages to this approach:

– The number of USB ports on a computer is limited. Even if you would add additional USB hubs, this adds additional latency because USB operates serially.

– There are 125 channels available; however, not all of them might lead to good performance since the 2.4 GHz band is shared. For example, BlueTooth and WiFi operate in the same band.

Therefore, we will use a single Crazyradio to control multiple Crazyflies and share the channels used. Hence, we will need to assign unique addresses to each Crazyflie to avoid crosstalking between the different quadcopters.

5.1 Assigning A Unique Address

The communication chips used in the Crazyflie and Crazyradio (nRF51 and nRF24LU1+ respectively) permit 40-bit addresses. By default, each Crazyflie has `0xE7E7E7E7E7` assigned as address. You can use the Bitcraze PC Client to change the address using the following steps:

1. Start the Bitcraze PC Client.
2. Make sure the address field is set to `0xE7E7E7E7E7` and click "Scan." The drop-down box containing "Select an interface" should now have another entry containing the URI of your Crazyflie, for example `radio://0/100/2M` (See Fig. 5, left). Select this entry and click "Connect."
3. In the "Connect" menu, select the item "Configure 2.0." In the resulting dialog (see Fig 5, right) change the address to a unique number, for example `0xE7E7E7E701` for your first Crazyflie, `0xE7E7E7E702` for the second one and so on. Select "Write" followed by "Exit."
4. In the PC Client, select "Disconnect."
5. Restart your Crazyflie.
6. Update the address field of the client (1 in Fig. 5, left) and click "Scan." If everything was successful, you should now see a longer URI in the drop-down box containing `radio://0/100/2M/E7E7E7E701`.

If it does not work, verify that you have the latest firmware for both nRF51 and STM32 flashed. This feature might not be available or working properly otherwise. The address (and other radio parameters) are stored in EEPROM and therefore will remain even if you upgrade the firmware.

Scanning Limitation

The scan feature of both ROS driver and Bitcraze PC client assume that you know the address of your Crazyflie (it is not feasible to try 2^{40} different addresses during scanning). If you forget the address, you will need to reset the EEPROM to its default values by connecting the Crazyflie directly to the PC using a USB cable and running a Python script[a].

[a] `https://wiki.bitcraze.io/doc:crazyflie:dev:starting#reset_eeprom`

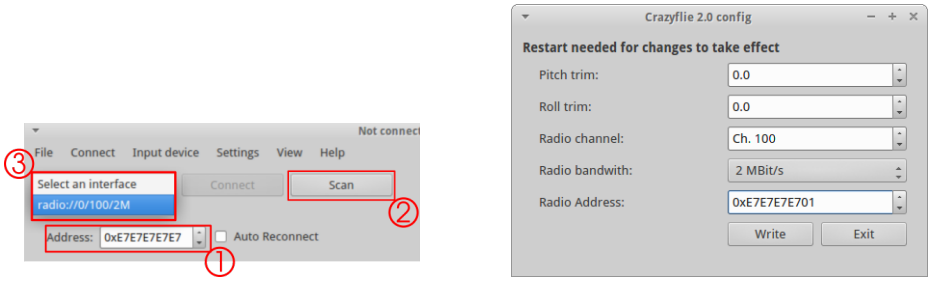

Fig. 5. Left: To connect to a Crazyflie, first enter its address, click "Scan", and finally select the found Crazyflie in the drop-down box. Right: The configuration dialog for the Crazyflie to update radio related parameters.

5.2 Finding Good Communication Parameters

The radio can be tuned by changing two parameters: datarate and channel. The datarate can be 250 kBit/s, 1 MBit/s, or 2 MBit/s. A higher datarate has a lower chance of collision with other networks such as WiFi but less range. Hence, for indoor applications the highest datarate (2 MBit/s) is recommended.

The channel number defines the offset in MHz from the base frequency of 2400 MHz. For example, channel 15 sets the operating frequency to 2415 MHz and channel 80 refers to an operating frequency of 2480 MHz. If you selected 2 MBit/s as datarate, the channels need to have a spacing of at least 2 MHz (otherwise, a 1 MHz spacing is sufficient).

Unlike WiFi, there is no channel hopping implemented in the Crazyflie. That means that the selected channel is very important because it will not change over time. On the other hand, interference can change over time; for example, a WiFi router might switch channels at runtime. Therefore, it is best if, during your flights, you can disable any interfering signal such as WiFi or wireless mouse/keyboards which use the 2.4 GHz band. If that is not possible, you can use the following experiments to find a set of good channels:

- Use the Bitcraze PC Client to teleoperate the Crazyflie in the intended space. Look at the "Link Quality" indicator on the top right. This indicator shows the percentage of successfully delivered packets. If it is low, there is likely interference.
- If you teleoperate the Crazyflie using ROS, there will be a ROS warning if the link quality is below a certain threshold. Avoid those channels. Additionally, `rqt_plot` shows the Radio Signal Strength Indicator (RSSI). This value, measured in -dBm, indicates the signal strength, which is affected both by distance and interference. A low value (e.g., 35) is good, while a high value ($>$ 80) is bad. For example, the output in Fig. 6 suggests that another channel should be used, because the second half of the plot shows additional noise caused by interference.

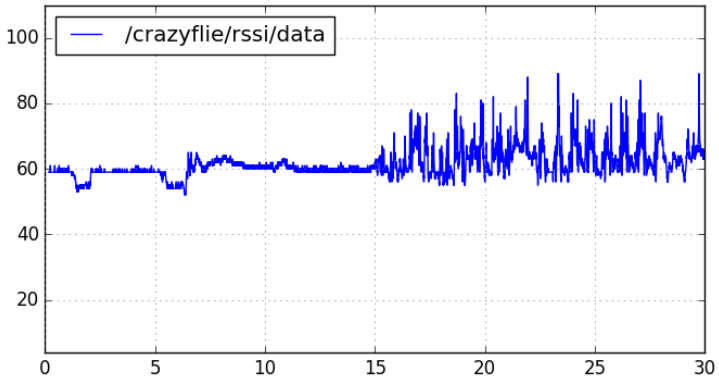

Fig. 6. Output of `rqt_plot` showing the radio signal strength indicator. The first 15 s show a good signal, while the second half shows higher values and noise caused by interference.

Once you have found a set of good channels, you can assign them to your Crazyflies, using the Bitcraze PC Client (see section 5.1 for details). You can share up to four Crazyflies per Crazyradio with reasonable performance. Hence, the number of channels you need is about a quarter of the number of Crazyflies you intend to fly.

Legal Restrictions

In some countries the 2.4 GHz band is limited to certain channels. Please refer to your local regulations before you adjust the channel.
For example, in the United States frequencies between 2483.5 and 2500 MHz are power-restricted and as a result frequently not used by WiFi routers. Hence, channels 84 to 100 might be a good choice there. Channels above 2500 MHz are not allowed to be used in the United States.

5.3 ROS Usage (Multiple Crazyflies)

Let's assume that you have two Crazyflies with unique addresses, two joysticks, and a single Crazyradio. You can teleoperate them using the following command:

```
$ roslaunch crazyflie_demo multi_teleop_xbox360.launch
    uri1:=radio://0/100/2M/E7E7E7E701
    uri2:=radio://0/100/2M/E7E7E7E702
```

This should connect to both Crazyflies, visualize their state in `rivz`, and plot real-time data using `rqt_graph`. Furthermore, each joystick can be used to teleoperate one of the Crazyflies.

The launch file looks very similar to the single UAV case:

multi_teleop_xbox360.launch

```
1  <launch>
2    <arg name="uri1" default="radio://0/90/2M" />
3    <arg name="uri2" default="radio://0/80/2M" />
4    <arg name="joy_dev1" default="/dev/input/js0" />
5    <arg name="joy_dev2" default="/dev/input/js1" />
6
7    <include file="$(find
         crazyflie_driver)/launch/crazyflie_server.launch" />
8    <group ns="crazyflie1">
9      <!-- Similar to before -->
10   </group>
11   <group ns="crazyflie2">
12       <!-- Similar to before -->
13   </group>
14   <!-- Visualization (Similar to before) -->
15 </launch>
```

In particular, we still have a single `crazyflie_server` (which now manages both Crazyflies). However, we have two different namespaces (`crazyflie1` and `crazyflie2`). The content of those namespaces is nearly identical to the single UAV case (compare lines 5 to 16 in `teleop_xbox360.launch`, section 4) and thus not repeated here for clarity.

In order to teleoperate more than two Crazyflies, you simply need to add more groups with different namespaces to the launch file. If you want the ROS driver to use a different Crazyradio, you can adjust the first number in the URI. For example, `radio://1/100/2M/E7E7E7E701` uses the second Crazyradio (or reports an error if only one is plugged in). It is important to consider the following for the usage of multiple radios:

- For improved performance, use the same channel per Crazyradio. This avoids that the radio changes channels whenever it switches between sending to different Crazyflies.
- If you do not need the IMU raw data, disable it by setting `enable_logging` to `False` when you include the `crazyflie_add.launch` file. This saves bandwidth and allows you to use more than two Crazyflies per radio.

Depending on your packet drop rate, you can use up to two Crazyflies per Crazyradio if logging is enabled and up to four otherwise. It does work with a higher number as well, but you will see decreasing controllability since the radio is used in a time-slice fashion.

6 Hovering

A first important step for autonomous flight of a quadcopter is hovering in place. This also requires the ability to take off from the ground and land after

the flight. All of these basic motions require a position controller, which takes the Crazyflie's current position as input in order to compute new commands for the Crazyflie. Hence, this position controller is replacing the teleoperating human we had before.

This section describes the `crazyflie_controller` package and how it is used for autonomous take-off, landing, and hovering. As before, first the single UAV case is considered and later it is extended to the multi-UAV case. Furthermore, this section will cover working strategies on how to use the crazyflie with optical motion capture systems such as VICON[9] and OptiTrack[10]. This is, due to the size of the UAV, a non-trivial task, particularly for swarming applications.

6.1 Position Estimate

We assume that there is already a way to track the position and preferably yaw of the Crazyflies at at least 30 Hz. It is possible to use Microsoft Kinect[11], AR tags, or Ultra-Wideband Localization [10] for this task. However, those solutions are not as accurate at specialized motion capture systems, which can reach sub-millimeter accuracy. We want to fly many small quadcopters, perhaps in a dense formation, and hence need a very accurate position feedback. Therefore, we will discuss the usage of optical motion capture systems such as VICON or OptiTrack.

We run our experiments in a space of approximately $5\,\mathrm{m} \times 4\,\mathrm{m}$ equipped with a 12-camera VICON MX motion capture system. Optical motion capture systems typically require spatially unique marker configurations for each object to track such that it is possible to identify each object[12]. Otherwise, occlusions or a short-term camera outage would result in unrecoverable tracking failures. For a small platform like the Crazyflie, there are not many ways to place markers uniquely on the existing frame. In particular, if you need more than four Crazyflies, you will need to add additional structures where you can place the markers:

- Propeller guards. They are commercially available for the Hubsan X4 toy quadrotor[13], which has identical physical dimensions. Moreover, you can use a 3D printer to print your own guard based on published files on thingiverse[14].
- Custom motor mounts. OpenSCAD[15] files can be found in the official mechanical repository[16].

[9] `http://www.vicon.com/`
[10] `https://www.optitrack.com/`
[11] `https://github.com/ataffanel/crazyflie-ros-kinect2-detector`
[12] Some solutions, like the Crazyswarm project [5], use identical marker configurations.
[13] You can search on amazon for "propeller guard hubsan x4."
[14] `http://www.thingiverse.com/search?q=crazyflie&sa=`
[15] `http://www.openscad.org/`
[16] `https://github.com/bitcraze/bitcraze-mechanics/tree/master/cf2-mount-openscad`

– Spatial extensions in form of sticks, either mounted on the rotor arms or on top of the Crazyflie as extension board.

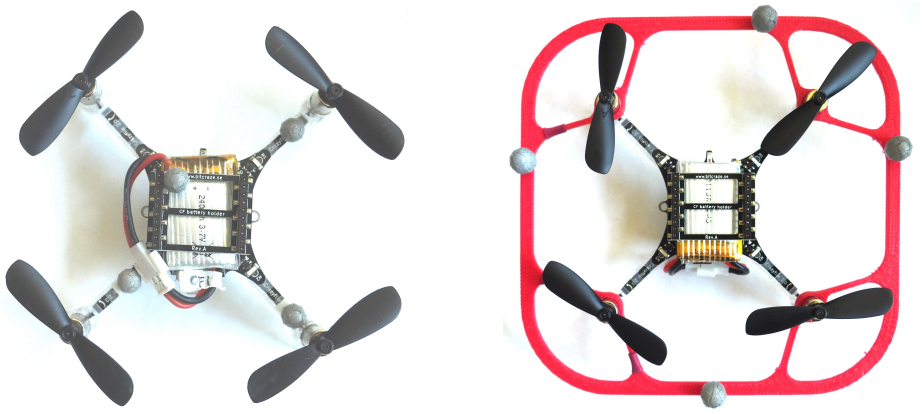

Fig. 7. Left: Crazyflie with four optical markers (6.4 mm) attached and no additional guard used. Right: Crazyflie with markers on propeller guard to allow a higher number of unique marker configurations.

For small groups of up to four Crazyflies, we place the markers directly on the Crazyflies. We flew up to six Crazyflies (using three Crazyradios) using the propeller guard approach. However, this significantly reduces flight times and changes the flight dynamics.

The exact method is highly dependent on your motion capture system, so there will be some experimentation involved. Similarly, the best markers to use depend on the system as well. We successfully use 6.4 mm and 7.9 mm spherical traditional reflective markers from B&L Engineering[17]. A smaller size impacts the flight dynamics less (and fits underneath the rotors) and is preferred as long as the motion capture system is able to detect the markers properly. We use the No-Base option of the markers and small pieces of Command Poster Strips[18] to attach them to the Crazyflie.

If you use VICON, it is best to install the `vicon_bridge` ROS package using the following steps:

```
$ cd ~/crazyflie_ws/src
$ git clone https://github.com/ethz-asl/vicon_bridge.git
$ cd ~/crazyflie_ws
$ catkin_make
```

[17] `http://www.bleng.com/`
[18] `http://www.command.com`

This will add the source to your workspace and compile it. The package assumes that you have another PC with VICON Tracker running in the same network, accessible under the hostname `vicon` and with no firewalls in between. You can test your installation by running:

```
$ roslaunch vicon_bridge vicon.launch
```

In another terminal, execute:

```
$ rosrun tf view_frames
```

and open the resulting `frames.pdf` file to check your transformations. It should look like Fig. 8.

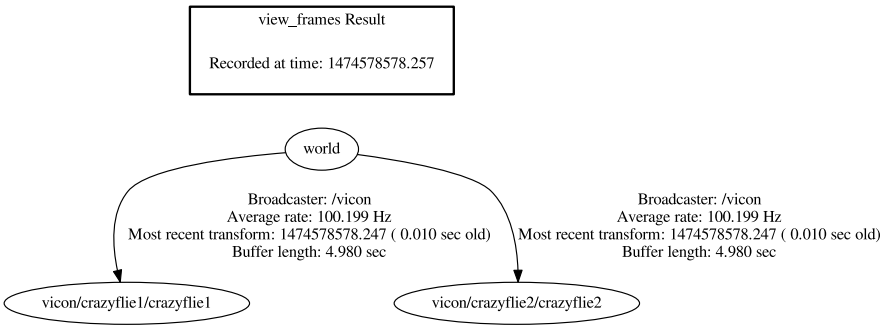

Fig. 8. Output of `view_frames` for two objects names `crazyflie1` and `crazyflie2`, respectively.

If you use OptiTrack (or any other motion capture system which supports VRPN[19]), you can install the `vrpn_client_ros` package using:

```
$ sudo apt-get install ros-indigo-vrpn-client-ros
```

In order to test it, you will need to write a custom launch file, similar to the sample file provided in the package[20]. Afterwards, you can check if it works using `view_frames`.

Coordinate System

It is important to verify that your transformations match the ROS standard[a]. That means we use a right-handed coordinate system with x for-

[19] `https://github.com/vrpn/vrpn/wiki`

[20] `https://github.com/clearpathrobotics/vrpn_client_ros/blob/indigo-devel/launch/sample.launch`

ward, y left, and z pointing up. One way to check is to launch `rviz`, and add a "TF" visualization. Move the Crazyflie around in your hand, while verifying that the visualization in `rviz` matches the expected coordinate system.

[a] `http://www.ros.org/reps/rep-0103.html`

6.2 ROS Usage (Single Crazyflie)

Here, we assume that you have a working localization for a single Crazyflie already. We assume that there is a ROS transform between the frames `/world` and `/crazyflie1` and that `radio://0/100/2M/E7E7E7E701` is the URI of your Crazyflie.

With VICON you can launch the following:

```
$ roslaunch crazyflie_demo hover_vicon.launch
    uri:=radio://0/100/2M/E7E7E7E701 frame:=crazyflie1 x:=0 y:=0
    z:=0.5
```

Once the Crazyflie is connected, you can press the blue (X) button on the XBox360 controller to take off and the green (A) button to land. If successful, the Crazyflie should hover at $(0, 0, 0.5)$. Use the red (B) button to handle any emergency situation (or unplug the Crazyradio to get the same effect). The launch file starts `rviz` as well, visualizing both the Crazyflie's current position and goal position (indicated by a red arrow).

If you are using OptiTrack, you can use `hover_vrpn.launch` rather than `hover_vicon.launch`.

The launch file is similar to before, but adds a few more elements:

hover_vicon.launch

```
1  <launch>
2    <!-- Launch file arguments -->
3    <include file="$(find
         crazyflie_driver)/launch/crazyflie_server.launch" />
4    <group ns="crazyflie">
5      <!-- Similar to before -->
6      <node name="joystick_controller" pkg="crazyflie_demo"
           type="controller.py" output="screen">
7        <param name="use_crazyflie_controller" value="True" />
8      </node>
9      <include file="$(find
           crazyflie_controller)/launch/crazyflie2.launch">
10       <arg name="frame" value="$(arg frame)" />
11     </include>
12     <node name="pose" pkg="crazyflie_demo" type="publish_pose.py"
           output="screen">
```

```
13        <param name="name" value="goal" />
14        <param name="rate" value="30" />
15        <param name="x" value="$(arg x)" />
16        <param name="y" value="$(arg y)" />
17        <param name="z" value="$(arg z)" />
18      </node>
19      <node pkg="tf" type="static_transform_publisher"
            name="baselink_broadcaster" args="0 0 0 0 0 0 1 $(arg
            frame) /crazyflie/base_link 100" />
20    </group>
21    <!-- run vicon bridge or vrpn_client_ros -->
22    <param name="robot_description" command="$(find xacro)/xacro.py
          $(find crazyflie_description)/urdf/crazyflie.urdf.xacro" />
23    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
          crazyflie_demo)/launch/crazyflie_pos.rviz" required="true"
          />
24  </launch>
```

We start by defining the arguments (not shown here for brevity) and launching the `crazyflie_server` (line 3). Within the group element, we include `crazyflie_add` (not shown). Now we get a few differences: in line 7 we set `use_crazyflie_controller` to `True` to enable the takeoff and landing behavior using the joystick. Moreover, we add a position controller node by including `crazyflie2.launch` (lines 9 to 11). The static goal position for this controller is published in the `/crazyflie/goal` topic in lines 12 to 18. The group ends by publishing a static transform from the given frame to the Crazyflie's base link. This allows us to visualize the current pose of the Crazyflie in `rviz` using the 3D model provided in the `crazyflie_description` package (lines 19 and 22).

6.3 ROS Usage (Multiple Crazyflies)

The main difference between the single UAV and multi-UAV case is that the joystick should be shared: Takeoff, landing, and an emergency should trigger the appropriate behavior on all Crazyflies. This allows us to have a single backup pilot who can trigger emergency power-off in case of an issue. The low inertia of the Crazyflies causes them to be very robust to mechanical failures when dropping from the air. We have had crashes from heights of up to 4 m on a slightly padded floor, with only propellers and/or motor mounts needing replacement. (Replacement parts are available for purchase separately.)

The `crazyflie_demo` package contains an example for hovering two Crazyflies. You can run it for VICON by executing:

```
$ roslaunch crazyflie_demo multi_hover_vicon.launch
    uri1:=radio://0/100/2M/E7E7E7E701 frame1:=crazyflie1
    uri2:=radio://0/100/2M/E7E7E7E702 frame2:=crazyflie2
```

There is also an example using VRPN (`multi_hover_vrpn.launch`). The launch file is similar to the single Crazyflie case:

multi_hover_vicon.launch

```
1  <launch>
2    <!-- Launch file arguments -->
3    <include file="$(find
         crazyflie_driver)/launch/crazyflie_server.launch" />
4    <node name="joy" pkg="joy" type="joy_node" output="screen">
5      <param name="dev" value="$(arg joy_dev)" />
6    </node>
7    <group ns="crazyflie1">
8      <!-- Similar to before -->
9      <node name="joystick_controller" pkg="crazyflie_demo"
           type="controller.py" output="screen">
10       <param name="use_crazyflie_controller" value="True" />
11       <param name="joy_topic" value="/joy" />
12     </node>
13   </group>
14   <group ns="crazyflie2">
15     <!-- Similar to first group -->
16   </group>
17   <!-- Similar to before -->
```

In this case, we only need a single joystick; the `joy` node for it is instantiated in lines 4 to 6. In order to use that topic, we need to supply `controller.py` with the correct topic name (line 11).

We can summarize what we have learned so far by looking at the output of `rqt_graph`, as shown in Fig. 9. It shows the various nodes (ellipsoid), namespaces (rectangles), and topics (arrows). In particular, we have two namespaces: `crazyflie1` and `crazyflie2`. Each namespace contains the nodes used for a single Crazyflie: `joystick_controller` to deal with the user-input, `pose` to publish the (static) goal position for that particular Crazyflie, and `controller` to compute the low-level attitude commands based on the high-level user input. The attitude commands are transmitted using the `cmd_vel` topics. There is only one node, the `crazyflie_server`, which listens on those topics and transmits the data to both Crazyflies, using the Crazyradio. The `joy` node provides the input to both namespaces, allowing a single user to control both Crazyflies. Similarly, the `vicon` node is shared between Crazyflies, because the motion-capture system provides feedback (in terms of `tf` messages) of all quadcopters. The `baselink_broadcaster` nodes are only used for visualization purposes, allowing us to visualize a 3D model of the Crazyflie in `rviz`. More than two Crazyflies can be used by duplicating the groups in the launch file accordingly. This will result in more namespaces; however, the `crazyflie_server`, `vicon`, and `joy` nodes will always be shared between all Crazyflies.

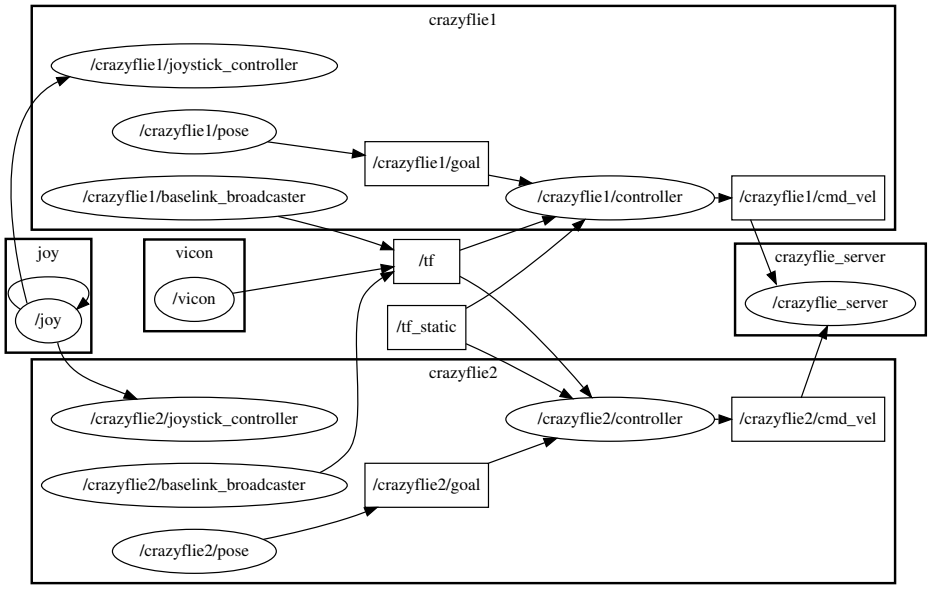

Fig. 9. Visualization of the different nodes and their communication using `rqt_graph`.

7 Waypoint Following

The hovering of the previous section is extended to let the UAVs follow specified waypoints. This is useful if you want the robots to fly specified routes, for example for delivery systems or construction tasks. As before, first the single-UAV case is presented, followed by how to use it in the multi-UAV case.

Here, we concentrate on the ROS-specific changes in a toy example where the waypoints are static and known upfront. Planning such routes for a group of quadcopters is a difficult task in itself and we refer the reader to related publications [11,12,13].

The main difference between hovering and waypoint following is that, for the latter, the goal changes dynamically. First, test the behavior of the controller for dynamic waypoint changes:

```
$ roslaunch crazyflie_demo teleop_vicon.launch
```

Here, the joystick is used to change the goal pose rather than influencing the motor outputs directly. The visualization in `rviz` shows the current goal pose as well as the quadcopter pose to provide some feedback.

Waypoint following works in a similar fashion: the first waypoint is set as goal position and, once the Crazyflie reaches its current goal (within some radius), the goal point is set to the next position. This simple behavior is implemented in `crazyflie_demo/scripts/demo.py`. Each Crazyflie can have its own waypoint defined in a Python script, for example:

```
#!/usr/bin/env python
from demo import Demo

if __name__ == '__main__':
    demo = Demo(
        [
            #x , y, z, yaw, sleep
            [0.0 , 0.0, 0.5, 0, 2],
            [1.5 , 0.0, 0.5, 0, 2],
            [-1.5 , 0.0, 0.75, 0, 2],
            [-1.5 , 0.5, 0.5, 0, 2],
            [0.0 , 0.0, 0.5, 0, 0],
        ]
    )
    demo.run()
```

Here, x, y, and z are in meters, yaw is in radians, and sleep is the delay in seconds before the goal switches to the next waypoint.

Adjust `demo1.py` and `demo2.py` to match your coordinate system and run the demo for two Crazyflies using:

```
$ roslaunch crazyflie_demo multi_waypoint_vicon.launch
```

The path for the two Crazyflies should not be overlapping because simple waypoint following does not have any time guarantees. Hence, it is possible that the first Crazyflie finishes much earlier than the second one, even if the total path length and sleep time are the same. This limitation can be overcome by generating a trajectory for each Crazyflie and setting the goal points dynamically accordingly.

The launch file looks very similar to before:

multi_waypoint_vicon.launch

```
<launch>
  <!-- Launch file arguments -->
  <include file="$(find
      crazyflie_driver)/launch/crazyflie_server.launch" />
  <node name="joy" pkg="joy" type="joy_node" output="screen">
    <param name="dev" value="$(arg joy_dev)" />
  </node>
  <group ns="crazyflie1">
    <!-- Similar to before -->
    <node name="pose" pkg="crazyflie_demo" type="demo1.py"
        output="screen">
      <param name="frame" value="$(arg frame1)" />
    </node>
  </group>
```

```
13    <group ns="crazyflie2">
14      <!-- Similar to first group -->
15    </group>
16    <!-- Similar to before -->
17  </launch>
```

Instead of publishing a static pose, each Crazyflie now executes its own `demo<x>.py` node, which in turn publishes goals dynamically. An example video demonstrating six Crazyflies following dynamically changing goals is available online[21].

8 Troubleshooting

As with most physical robots, debugging can be difficult. In order to identify and eventually solve the problem, it helps to simplify the failing case until it is easier to analyze. In this section, we provide several actions which have helped us resolve issues in the past. In particular, we first identify if the issue is on the hardware or software side, and provide recipes to address both kinds of issues.

1. Verify that the position estimate works correctly. For example, use `rviz` to visualize the current pose of all quadrotors, move a single quadrotor manually at a time and make sure that `rviz` reflects the changes accordingly.
2. Check the wireless connection between the PC and the Crazyflies. If the packet drop rate is high, the `crazyflie_server` will output ROS warnings. Similarly, you can check the LEDs on each Crazyradio; ideally the LEDs show mostly green. If there is an communication issue the LEDs will frequently flash red as well. If communication is an issue, try a different channel by following section 5.2.
3. Work your way backwards: If a swarm fails, test the individual Crazyflies (or subgroups of them). If waypoint following fails, test hovering and, if there is an issue there as well, test teleoperation using ROS followed by teleoperation using the Bitcraze PC Client.
4. Issues with many Crazyflies but not smaller subgroups can occur if there are communication issues or if the position estimate suddenly worsens. For the first case, try reducing the number of Crazyflies per Crazyradio and adjusting the channel. For the second case try to estimate the latency of your position estimator. If you have multiple objects enabled, there might be axis-flips (marker configurations might not be unique enough) or the computer doing the tracking might be adding too much latency for the controller to operate properly.
5. If waypoint following does not work, make sure that you visualize the current waypoint in `rviz`. In general, the waypoints should not jump around very much. The provided controller is a hover controller which works well if the goal point is within a reasonable range of the Crazyflie's current position.

[21] `http://youtu.be/px9iHkA0nOI`

6. If hovering does not work, you can try to tune the provided controller. For example, if you have a higher payload you might increase the proportional gains. You can find the gains in `crazyflie_controller/config/crazyflie2.yaml`.
7. If teleoperation does not work or it is very hard to keep the Crazyflie hovering in place, there is most likely an issue with your hardware. Make sure that the propellers are balanced[22] and that the battery is placed in the center of mass. When in doubt, replace the propellers.

9 Inside the crazyflie_ros Stack

This section will cover some more details of the stack. The knowledge you gain will not only help you better understand on what is happening under the hood, but also provide the foundations to change or add features. Furthermore, some of the design insights given might be helpful for similar projects.

We will start with a detailed explanation of the different packages that compose the stack and their relationship. For each package, we will discuss important components and the underlying architecture. For example, for the `crazyflie_driver` package we will explain the different ROS topics and services, why there is a server, and how the radio time-slicing works. A section on guidelines for possible extensions will conclude the chapter.

9.1 Overview

The `crazyflie_ros` stack is composed of six different packages:

crazyflie_cpp contains a C++11 implementation for the Crazyradio driver as well as the Crazyflie. It supports the logging framework streaming data in real-time and the parameter framework adjusting parameters such as PID gains. This package has no ROS dependency and only requires `libusb` and `boost`. Unlike the official Python SDK it supports multiple Crazyflies over a shared radio.

crazyflie_tools contains standalone command line tools which use the `crazyflie_cpp` library. Currently, there is a tool to find any Crazyflies in range and tools to list the available logging variables and parameters. Because there is no ROS dependency, the tools can be used without ROS as well.

crazyflie_description contains the URDF description of the Crazyflie to visualize in `rviz`. The models are currently not accurate enough to be used for simulation.

crazyflie_driver contains a ROS wrapper around `crazyflie_cpp`. The logging subsystem is mapped to ROS messages and parameters are mapped to ROS parameters. One node (`crazyflie_server`) manages all Crazyflies.

[22] `https://www.bitcraze.io/balancing-propellers/`

crazyflie_controller contains a PID position controller for the Crazyflie. As long as the position of the Crazyflie is known (e.g. by using a motion capture system or a camera), it can be used to hover or execute (non-aggressive) flight maneuvers.

crazyflie_demo contains sample scripts and launch files for teleoperation, hovering, and waypoint following for both single and multi Crazyflie cases.

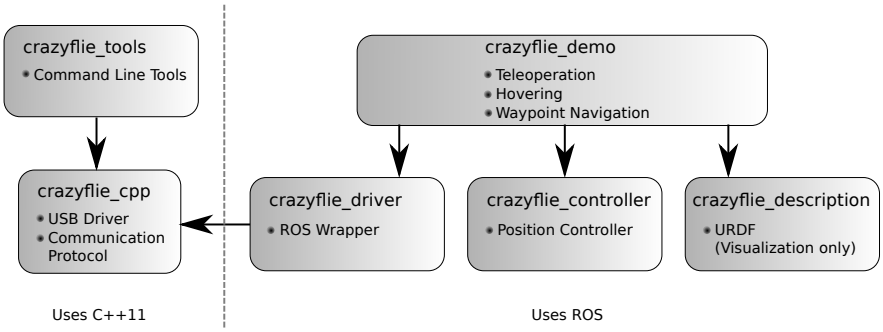

Fig. 10. Dependencies between the different packages within the `crazyflie_ros` stack.

The dependencies between the packages are shown in Fig 10. Both `crazyflie_tools` and `crazyflie_demo` contain high-level examples. Because `crazyflie_cpp` does not have any ROS dependency, it can be used with other frameworks as well. We will now discuss the different packages in more detail.

9.2 crazyflie_cpp

The `crazyflie_cpp` package is a static C++ library, with some components being header-only to maximize type-safety and efficiency. The library consists of four classes:

Crazyradio This class uses `libusb` to communicate with a Crazyradio. It supports the complete protocol[23] implemented in the Crazyradio firmware. The typical approach is to configure the radio (such as channel and datarate to use) first, followed by actual sending and receiving of data. The Crazyradio operates in Primary Transmitter Mode (PTX), while the Crazyflie operates in Primary Receiver Mode (PRX). That means that the Crazyradio is sending data (with up to 32 bytes of payload) using the radio and, if the data is successfully received, will receive an acknowledgement from the Crazyflie. The acknowledgment packet might contain up to 32 bytes of user-data as well. However, since the acknowledgment has to be sent immediately, the acknowledgment is not a direct response to the request sent. Instead, the

[23] `https://wiki.bitcraze.io/projects:crazyradio:protocol`

communication can be seen as two asynchronous data streams, with one stream going from the Crazyradio to the Crazyflie and another stream for the reverse direction. If a request-respond like protocol is desired, it has to be implemented on top of the low-level communication infrastructure. The Crazyradio will automatically resend packets if no acknowledgment has been received.
Below is a small example on how to use the class to send a custom packet:

```
1  Crazyradio radio(0); // Instantiate an object bound to the
       first Crazyflie found
2  radio.setChannel(100); // Update the base frequency to 2500 MHz
3  radio.setAddress(0xE7E7E7E701); // Set the address to send to
4  // Send a packet
5  uint8_t data[] = {0xCA, 0xFE};
6  Crazyradio::Ack ack;
7  radio.sendPacket(data, sizeof(data), ack);
8  if (ack.ack) {
9      // Parse ack.data and ack.size
10 }
```

Exceptions are thrown in cases of error, for example if no Crazyradio could be found or if the user does not have the permission to access the USB dongle.

Crazyflie This class implements the protocol of the Crazyflie[24] and provides high-level functions to send new setpoints and update parameters. In order to support multiple Crazyflies correctly, it instantiates the Crazyradio automatically. A global static array of Crazyradio instances and mutexes is used. Whenever the Crazyflie needs to send a packet, it first uses the mutex to lock its Crazyradio, followed by checking if the radio is configured properly. If not, the configuration (such as address) is updated and finally the packet is sent. The mutex ensures that multiple Crazyflies can be used in separate threads, even if they share a Crazyradio. The critical section of sending a packet causes the radio to multiplex the requests in time. Therefore, the bandwidth is split between all Crazyflies which share the same radio.
Below is a small example demonstrating how multiple Crazyflies can be used with the same radio:

```
1  Crazyflie cf1("radio://0/100/2M/E7E7E7E701"); // Instantiate
       first Crazyflie object
2  Crazyflie cf2("radio://0/100/2M/E7E7E7E702"); // Instantiate
       second Crazyflie object
3  // launch two threads and set new setpoint at 100 Hz
4  std::thread t1([&] {
5    while (true) {
```

[24] https://wiki.bitcraze.io/projects:crazyflie:crtp

```
6     cf1.sendSetpoint(0, 0, 0, 10000); // send roll, pitch,
          yaw, and thrust
7     std::this_thread::sleep_for(std::chrono::milliseconds(10));
8   }
9 });
10 std::thread t2([&] {
11   while (true) {
12     cf2.sendSetpoint(0, 0, 0, 20000); // send roll, pitch,
          yaw, and thrust
13     std::this_thread::sleep_for(std::chrono::milliseconds(10));
14   }
15 });
16 t1.join();
17 );
```

First, two `Crazyflie` objects are instantiated. Then two threads are launched using C++11 and lambda functions. Each thread sends an updated setpoint consisting of roll, pitch, yaw, and thrust to its Crazyflie at about 100 Hz.

LogBlock<T> This set of templated classes is used to stream out sensor data from the Crazyflie. The logging framework on the Crazyflie allows to create so-called log blocks. Each log block is a struct with a maximum size of 28 bytes, freely arranged based on global variables available for logging in the Crazyflie firmware. The list of available variables and their types can be queried at runtime (`requestLogToc` method in the `Crazyflie` class).
This templated version provides maximum typesafety at the cost that you need to know at compile time which log blocks to request.

LogBlockGeneric This class is very similar to `LogBlock<T>` but also allows the user to dynamically create log blocks at runtime. The disadvantages of this approach are that it does not provide typesafety and that it is slightly slower at runtime.

9.3 crazyflie_driver

We first give a brief overview of the ROS interface, including services, subscribed topics, and published topics. In the second part we describe the usage and internal infrastructure in more detail.

Most of the services and topics are within the namespace of a particular Crazyflie, denoted with ⟨crazyflie⟩. For example, if you have two Crazyflies, there will be namespaces `crazyflie1` and `crazyflie2`.

The driver supports the following services:

add_crazyflie Adds a Crazyflie with known URI to the `crazyflie_server` node. Typically, this is used with the helper application from `crazyflie_add` from a launch file.
Type: crazyflie_ros/AddCrazyflie

⟨crazyflie⟩/emergency Triggers an emergency state, in which no further messages to the Crazyflie are sent. The onboard firmware will stop all rotors if

it did not receive a message for 500 ms, causing the Crazyflie to fall shortly after the emergency was requested.
Type: std_srvs/Empty

⟨**crazyflie**⟩**/update_params** Uploads updated values of the specified parameters to the Crazyflie. The parameters are stored locally on the ROS parameter server. This service first reads the current values and then uploads them to the Crazyflie.
Type: crazyflie_ros/UpdataParams

The driver subcribes the following topics:

⟨**crazyflie**⟩**/cmd_vel** Encodes the setpoint (attitude and thrust) of the Crazyflie. This can be used for teleoperation or automatic position control.
Type: geometry_msgs/Twist

The following topics are being published:

⟨**crazyflie**⟩**/imu** Samples the inertial measurement unit of the Crazyflie every 10 ms, including the data from the gyroscope and accelerometer. The orientation and covariance are not known and therefore not included in the messages.
Type: sensor_msgs/Imu

⟨**crazyflie**⟩**/temperature** Samples the temperature as reported by the barometer every 100 ms. This might not be the ambient temperature, as the Crazyflie tends to heat up during operation.
Type: sensor_msgs/Temperature

⟨**crazyflie**⟩**/magnetic_field** Samples the magnetic field as measured by the IMU every 100 ms. Currently, the onboard magnetometer is not calibrated in the firmware. Therefore, external calibration is required to use it for navigation. Type: sensor_msgs/MagneticField

⟨**crazyflie**⟩**/pressure** Samples the air pressure as measured by the barometer every 100 ms in mbar.
Type: std_msgs/Float32

⟨**crazyflie**⟩**/battery** Samples the battery voltage every 100 ms in V.
Type: std_msgs/Float32

⟨**crazyflie**⟩**/rssi** Samples the Radio Signal Strength Indicator (RSSI) of the onboard radio in −dBm.
Type: std_msgs/Float32

The `crazyflie_driver` consists of two ROS nodes: `crazyflie_server` and `crazyflie_add`. The first manages all Crazyflies in the system (using one thread for each), while the second one is just a helper node to be able to add Crazyflies from a launch file.

It is possible to launch multiple `crazyflie_server`'s, but these cannot share a Crazyradio. This is mainly a limitation of the operating system, which limits the ownership of a USB device to one process. In order to hide this implementation detail, each Crazyflie thread will operate in its own namespace. If you use `rostopic`, the topics of the first Crazyflie will be in the `crazyflie1` namespace

(or whatever `tf_frame` you assigned to it), even though the code is actually executed within the `crazyflie_server` context. Each Crazyflie offers a topic `cmd_vel` which is used to send the current setpoint (roll, pitch, yaw, and thrust) and, if logging is enabled, topics such as `imu`, `battery`, and `rssi`. Furthermore, services are used to trigger the emergency mode and to re-upload specified parameters. The values of the parameters themselves are stored within the ROS parameter server. They are added dynamically once the Crazyflie is connected, because parameter names, types, and values are all dynamic and dependent on your firmware version. For that reason, it is currently not possible to use the `dynamic_reconfigure` package, because in this case the parameter names and types need to be known at compile time. Instead, a custom service call needs to be triggered containing a list of parameters to update once a user changed a parameter on the ROS parameter server. The following Python example can be used to turn the headlight on (if the LED expansion is installed):

```
1 import rospy
2 from crazyflie_driver.srv import UpdateParams
3 rospy.wait_for_service("/crazyflie1/update_params")
4 update_params = rospy.ServiceProxy("/crazyflie1/update_params",
      UpdateParams)
5 rospy.set_param("/crazyflie1/ring/headlightEnable", 1)
6 update_params(["ring/headlightEnable"])
```

After the service has become available, a service proxy is created and can be used to call the service whenever a parameter needs to be updated. Updating the parameter sets the parameter to a new value followed by a service call, which will trigger an upload to the Crazyflie.

Another important part of the driver is the logging system support. If logging is enabled, the Crazyflie will advertise a number of fixed topics. In order to receive custom logging values (or at custom frequencies), you will either need to change the source code or use custom log blocks. The latter has the disadvantage that it is not typesafe (it just uses an array of floats as message type) and that it will be slightly slower at runtime. You can use custom log blocks as follows:

customLogBlocks.launch

```
1 <launch>
2   <arg name="uri" default="radio://0/80/2M" />
3   <include file="$(find
        crazyflie_driver)/launch/crazyflie_server.launch" />
4   <group ns="crazyflie">
5     <node pkg="crazyflie_driver" type="crazyflie_add"
          name="crazyflie_add" output="screen">
6       <param name="uri" value="$(arg uri)" />
7       <param name="tf_prefix" value="crazyflie" />
8       <rosparam>
```

```
        genericLogTopics: ["log1", "log2"]
        genericLogTopicFrequencies: [10, 100]
        genericLogTopic_log1_Variables: ["pm.vbat"]
        genericLogTopic_log2_Variables: ["acc.x", "acc.y", "acc.z"]
      </rosparam>
    </node>
  </group>
</launch>
```

Here, additional parameters are used within the `crazyflie_add` node to specify which log blocks to get. The first log block only contains `pm.vbat` and is sampled every 10 ms. A new topic named `/crazyflie1/log1` will be published. Similarly, the `/crazyflie1/log2` topic will contain three values (x, y, and z of the accelerometer), published every 100 ms.

The easiest way to find the names of variables is by using the Bitcraze PC Client. After connecting to a Crazyflie select "Logging Configurations" in the "Settings" menu. A new dialog will open and list all variables with their respective types. Each log block can only hold up 28 bytes and the minimum update period is 10 ms. You can also use the `listLogVariables` command line tool which is part of the `crazyflie_tools` package to obtain a list with their respective types.

9.4 crazyflie_controller

The Crazyflie is controlled by a cascaded PID controller. The inner attitude controller is part of the firmware. The inputs are the current attitude, as estimated using the IMU sensor data, and the setpoint (attitude and thrust), as received over the radio. The controller runs at 250 Hz.

The `crazyflie_controller` node runs another outer PID controller, which takes the current and goal position as input and produces a setpoint (attitude and thrust) for the inner controller. This cascaded design is typical if the sensor update rates are different [11]. In this case, the IMU can be sampled much more frequently than the position.

A PID controller has absolute, integral, and differential terms on an error variable:

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt}, \tag{1}$$

where $u(t)$ is the control output and K_P, K_I and K_D are scalar parameters. The error $e(t)$ is defined as the difference between the goal and current value. The `crazyflie_controller` uses four independed PID controllers for x, y, z, and yaw, respectively. The controller also handles autonomous takeoff and landing. The integral part of the z-PID controller is initialized during takeoff with the estimated required base thrust to keep the Crazyflie hovering. The takeoff routine linearly increases the thrust, until the takeoff is detected by the external position

system. A state machine switches to the PID controller, using the current thrust value as initial guess for the integral part of the z-axis PID controller. This avoids retuning of a manual offset in case the payload changes or a different battery is used.

The current goal can be changed by publishing to the `goal` topic. However, since the controller makes the hover assumption, large jumps between different control points should be avoided.

The various parameters can be tuned in a config file (`crazyflie_controller/config/crazyflie2.yaml`), or a custom config file can be loaded instead of the default one (see `crazyflie_controller/launch/crazyflie2.launch` for an example).

9.5 Possible Extensions

The overview of the `crazyflie_ros` stack should allow you to reuse some of its architecture ideas or to extend it further. For example, you can use the Crazyradio and `crazyflie_cpp` for any other remote-controlled robot which requires a low-latency radio link. The presented controller of the `crazyflie_controller` package is a simple hover controller. A non-linear controller, as presented in [14] or [11] might be an interesting extension to improve the controller performance. Higher-level behaviors, such as following a trajectory rather than just goal points, could make more interesting flight patterns possible. Finally, including simulation for the Crazyflie[25] could help research and development by enabling simulated experiments.

10 Conclusion

In this chapter we showed how to use multiple small quadcopters with ROS in practice. We discussed our target platform, the Bitcraze Crazyflie 2.0, and guided the reader step-by-step to the process of letting multiple Crazyflies following waypoints. We tested our approach on up to six Crazyflies, using three radios. We hope that this detailed description will help other researchers use the platform to verify algorithms on physical robots.

More recent research has shown that the platform can even be used for swarms of up to 49 robots [5]. In the future, we would like to provide a similar step-by-step tutorial about the additional required steps to guide other researchers in working on larger swarms. Furthermore, it would be interesting to make the work more accessible to a broader audience once more inexpensive but accurate localization systems become available.

[25] e.g. adding support to the RotorS package (`http://wiki.ros.org/rotors_simulator`)

11 Authors' Biographies

Wolfgang Hönig has been a Ph.D. student at ACT Lab at University of Southern California since 2014. He holds a Diploma in Computer Science from Technical University Dresden, Germany. He is the author and maintainer of the `crazyflie_ros` stack.

Nora Ayanian is an Assistant Professor at University of Southern California. She is the director of the ACT Lab at USC and received her Ph.D. from the University of Pennsylvania in 2011. Her research focuses on creating end-to-end solutions for multirobot coordination.

References

1. N. Michael, J. Fink, and V. Kumar, "Cooperative manipulation and transportation with aerial robots," *Autonomous Robots*, vol. 30, no. 1, pp. 73–86, 2011.
2. F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D'Andrea, "The flight assembled architecture installation: Cooperative construction with flying machines," *IEEE Control Systems*, vol. 34, no. 4, pp. 46–64, 2014.
3. W. Hönig, C. Milanes, L. Scaria, T. Phan, M. Bolas, and N. Ayanian, "Mixed reality for robotics," in *IEEE/RSJ Intl Conf. Intelligent Robots and Systems*, 2015, pp. 5382 – 5387.
4. A. Mirjan, F. Augugliaro, R. D'Andrea, F. Gramazio, and M. Kohler, *Robotic Fabrication in Architecture, Art and Design 2016*. Cham: Springer International Publishing, 2016, ch. Building a Bridge with Flying Robots, pp. 34–47.
5. J. A. Preiss, W. Hönig, G. S. Sukhatme, and N. Ayanian, "Crazyswarm: A large nano-quadcopter swarm," in *IEEE/RSJ Intl Conf. Intelligent Robots and Systems (Late Breaking Results)*, 2016.
6. N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "The GRASP multiple micro-uav testbed," *IEEE Robot. Automat. Mag.*, vol. 17, no. 3, pp. 56–65, 2010.
7. S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. DAndrea, "A platform for aerial robotics research and demonstration: The flying machine arena," *Mechatronics*, vol. 24, no. 1, pp. 41–54, 2014.
8. B. Landry, "Planning and control for quadrotor flight through cluttered environments," Master's thesis, MIT, 2015.
9. J. Förster, "System identification of the crazyflie 2.0 nano quadrocopter," Bachelor's Thesis, ETH Zurich, 2015.
10. A. Ledergerber, M. Hamer, and R. D'Andrea, "A robot self-localization system using one-way ultra-wideband communication," in *IEEE/RSJ Intl Conf. Intelligent Robots and Systems*, 2015, pp. 3131–3137.
11. D. Mellinger, "Trajectory generation and control for quadrotors," Ph.D. dissertation, University of Pennsylvania, 2012.
12. A. Kushleyev, D. Mellinger, C. Powers, and V. Kumar, "Towards a swarm of agile micro quadrotors," *Autonomous Robots*, vol. 35, no. 4, pp. 287–300, 2013.
13. W. Hönig, T. K. S. Kumar, H. Ma, S. Koenig, and N. Ayanian, "Formation change for robot groups in occluded environments," in *IEEE/RSJ Intl Conf. Intelligent Robots and Systems*, 2016.
14. T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor UAV on SE(3)," in *IEEE Conf. on Decision and Control*, 2010, pp. 5420–5425.