Crazyflie Swarm Controller Documentation

Release 1.0

Charles Sirois

Aug 14, 2020

Getting started

CHAPTER 1

Overview

Crazyflie Swarm Controller is a ROS package to fly a swarm of crazyflie in formation.

Main features:

- Python API
- Collision free trajectory planning via a DMPC algorithm
- Fly swarm in various formations (square, circle, line...)
- Simulation
- Easy control with a joystick

This project was developped with the **Mobile Robotics and Autonomous Systems Laboratory** at **Polytechnique Montréal**.

To cite this project:

Todo: Add citation

Crazyflie Swarm Controller uses crazyflier_ros [CIT1] to send commands to the crazflies via ros.

The architecture used was inspired by the Crazyswarm project [CIT2].

1.1 Example

1.1.1 Formation Example

```
# Formation exemple
swarm = SwarmAPI()

# Link joystick buttons to commands
```

(continues on next page)

(continued from previous page)

```
swarm.start_joystick("ds4")
swarm.link_joy_button("S", swarm.take_off)
swarm.link_joy_button("X", swarm.land)
swarm.link_joy_button("O", swarm.emergency)
swarm.link_joy_button("T", swarm.toggle_ctrl_mode)

# Start swarm
swarm.set_mode("formation")
swarm.set_formation("v")

swarm.take_off()
rospy.sleep(10)

# Change formation
swarm.set_formation("pyramid")
```

1.1.2 High level controller example

```
# Trade spots demo
swarm = SwarmAPI()
swarm.set_mode("automatic")

# Take off
swarm.take_off()

# Switch positions
pose = swarm.get_positions()
goals = {}
goals["cf_0"] = pose["cf_1"]
goals["cf_1"] = pose["cf_0"]
swarm.go_to(goals)

rospy.sleep(10)

# Land
swarm.land()
```

Installation

OS Requirement

This package was made and tested for **Ubuntu 18.04 with ROS Melodic**.

The following links might be useful:

- Dual boot windows 10 and linux
- Ubuntu 18.04
- ROS Melodic installation

Note: It might be possible to use this package with a VM. Please let me know if you try it and it works. Note that a VM will add latency.

Package installation

1 - Verify Python version. Only **Python 2.7.17** was tested

```
$ python --version
```

2 - Clone package

```
$ git clone https://github.com/MRASL/crazyflie_charles.git
```

3 - Install and build package using build script

```
$ cd crazyflie_charles
$ ./build.sh
```

4 - Install python dependencies

```
$ sudo apt install python-pip
$ pip install -U pip
$ pip install git+https://github.com/rmcgibbo/quadprog.git#egg=quadprog
$ pip install -r requirements.txt
```

5 - Install ros joystick driver

```
$ sudo apt-get install ros-melodic-joy
```

6 - Test installation

6.1 - Launch server

```
$ source ros_ws/devel/setup.sh
$ roslaunch swarm_manager launch_swarm.launch sim:=True
```

6.2 - In another terminal, start demo

```
$ source ros_ws/devel/setup.sh
$ python demos/trade_spots.py
```

Warning: Make sure your ros environment has been source and roscore is running before testing this exemple. See section 1.5.

7 - (optional) Automaticaly source ros workspace by adding it to `.bashrc`

```
$ echo "source <path_to_crazyflie_charles>/ros_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Note:

Replace `<path_to_crazyflie_charles>` with your installation path.

i.e: `~/projects/crazyflie_charles`

8 - Modify PC permissions

```
$ ./pc_permissions.sh
```

Usage

Crazyflie Assembly and Test

Before going further, follow this tutorial. It explains how to assemble a crazyflie and install the PC client for linux.

Swarm Setup

1. Set crazyflie address. Each crazyflie of the swarm needs to have a unique address. The address format is

   ```
   0xE7E7E7E7<X>
   ```

   where <X> is the crazyflie number in hexadecimal.

Note: It's highly recommended to label each CF with it's number.

2. Set crazyflie channel. Each crazyradio can handle communication with up to 15 crazyflies. If more crazyflies are in your swarm, you will need a different channel for each radio.
3. If not already done, update each crazyflie firmware.

Positionning System

For now, only the LPS (with 8 anchors) was tested for positionning. Follow this guide to setup your system.

Test your setup by flying using bitcraze client.

Configuration Files

There are two configuration files:

- `swarm_conf.yaml`
- `joy_conf.yaml`

They are located at: `../crazyflie_charles/ros_ws/src/formation_manager/conf`

swarm_conf

This file is used to configure parameters of all three ros packages. It's where you can change the number of crazyflie in the swarm.

```yaml
# ../crazyflie_charles/ros_ws/src/formation_manager/conf/swarm_conf.yaml
swarm:
    n_cf: 2 # Number of CF in the swarm
    take_off_height: 0.5
    gnd_height: 0.0
    min_dist: 0.40  # Absolute min distance between CFs
    min_goal_dist: 0.40 # Absolute min distance between CFs goals

formation:
    formation_min_dist: 0.6 # Min distance between agents in formation
    formation_start_pos: [0.5, 0.5, 0.5, 0.0] # [x, y, z, yaw]

trajectory_solver:
    # trajectory_solver parameters ...
    ...

# Starting positions. Used in simulation
starting_positions:
    cf_0: [0.0, 0.0, 0.0]
    cf_1: [0.0, 0.5, 0.0]
    cf_2: [0.0, 1.0, 0.0]
    cf_3: [0.5, 0.0, 0.0]
    ...
```

joy_conf

File used to map your controller buttons. To learn how to setup a new a new controller, see this *tutorial*.

Note: If you are using a ps4 controller (dualshock), you will need to download this driver.

```yaml
# ../crazyflie_charles/ros_ws/src/formation_manager/conf/swarm_conf.yaml
ds4:
    # Map axes and joystick stick number
    axes: # Axes start at 0
        x: 1
        y: 0
        z: 5
        yaw: 2

    # Map button name and position
    buttons:
        '0': S # Square
        '1': X # Cross
        '2': O # Circle
        '3': T # Triangle
        ...

    # Map buttons on a joystick axis, i.e: d-pad
    buttons_axes:
        '9': DL
        '-9' : DR
        '10' : DU
        '-10': DD
```

(continues on next page)

(continued from previous page)

```
    # Max velocity of goal
    max_goal:
        x: 0.20
        y: 0.20
        z: 0.10
        yaw: 0.20
```

Flying

1. Turn on and place all your CFs in the flight alrea
2. Launch ros server

   ```
   $ roslaunch swarm_manager launch_swarm.launch
   ```

   There are two options when launching server:

   - `sim:=bool` (default: True): To run in simulation
   - `save:=bool` (default: False): To save flight data when closing server

3. In another terminal, execute python script

   ```
   $ cd ../crazyflie_charles/demos
   $ python trade_spots.py
   ```

Data Analysis

A python script allow to analyse the data took. To run the script

```
$ cd ../crazyflie_charles/flight_data
$ python flight_analysis.py
```

Note: It's possible to specify a file name using -d flag. If no file name specified, latest data will be loaded.

Possible commands:

- Rename data set
- List all cf in recorded data
- Plot flight path of a crazyflie
- Plot trajectory error

Note: Enter `help` to print all commands and their arguments.

Tutorials

Setup a Controller

This tutorial will show you how to find your controller layout and add it as a configuration.

1. Connect the controller to your computer

Note: You may need to download additional drivers. (i.e drivers for ps4 controller)

2. Add new controller to conf file.

```
# Add to ../crazyflie_charles/ros_ws/src/formation_manager/conf/swarm_conf.yaml
controller_name:
    axes: # Axes start at 0
        x: _
        y: _
        z: _
        yaw: _

    buttons:
        '0': _
        '1': _
        '2': _

    buttons_axes:
        '-0': DL
        '0' : DR
        '1' : DU
        '-1': DD

    max_goal:
        x: 0.20
        y: 0.20
        z: 0.10
        yaw: 0.20
```

2. Read controller input

```
$ rosrun joy joy_node
```

3. Map joystick controls

3.1. In a new terminal, print information

```
$ rostopic echo /joy
```

3.2 Add buttons to conf file

Find number of each button.

```
---
header:
  seq: 42826
  stamp:
    secs: 1596049352
    nsecs: 514089384
  frame_id: ''
axes: [-0.0, -0.0, -0.0, 0.0, 0.0, -0.0, -0.0, -0.08466958999633789, -0.4700380563735962, 0.0, 0.0, -0.0, -0.0, -0.0]
buttons: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
```

Fig. 1: Result of pressing X button. When can then map X to button #2

```
controller_name:
    ...
```

(continues on next page)

(continued from previous page)

```
    buttons:
        '0': _
        '1': _
        '2': X

    ...
```

3.3 Add axes to conf file

You have four controls to map: *x position*, *y position*, *z position* and *yaw*.

Let's say you want to map the horizontal right stick to yaw control and get this result when moving the stick to the left.

```
---
header:
  seq: 159302
  stamp:
    secs: 1596050282
    nsecs: 244312922
  frame_id: ''
axes: [-0.0, -0.0, 1.0, 0.0, 0.0, -0.0, -0.0, -0.07773064076900482, -0.4752422571182251, 0.0, 0.0, -0.0, -0.0, -0.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---
```

Fig. 2: Result of moving the right stick to the left.

Based on the picture, we can find that the horizontal right stick axis number is 2. Also, since moving it to the left gives a positive result, the axis number will be negative:

```
controller_name:
    ...

    axes:
        x:
        y:
        z:
        yaw: -2


    ...
```

Repeat this for all controls you wish to map.

4. Add controller to API:

```
# .../crazyflie_charles/ros_ws/src/swarm_manager/scripts/swarm_api/api.py
class SwarmAPI(object):
...
def start_joystick(self, joy_type=""):
"""Initialize joystick node

    Possible types are:
        - ds4
        - ADD NEW CONTROLLER

    ...
"""
...
```

5. Try new controller with api by using

```
SwarmAPI.start_joystick("new_controller")
```

Add New Formation

In this tutorial, you will learn how to create a **new formation**.

We are going to add a formation shaped like a sinus.

1. Define formation

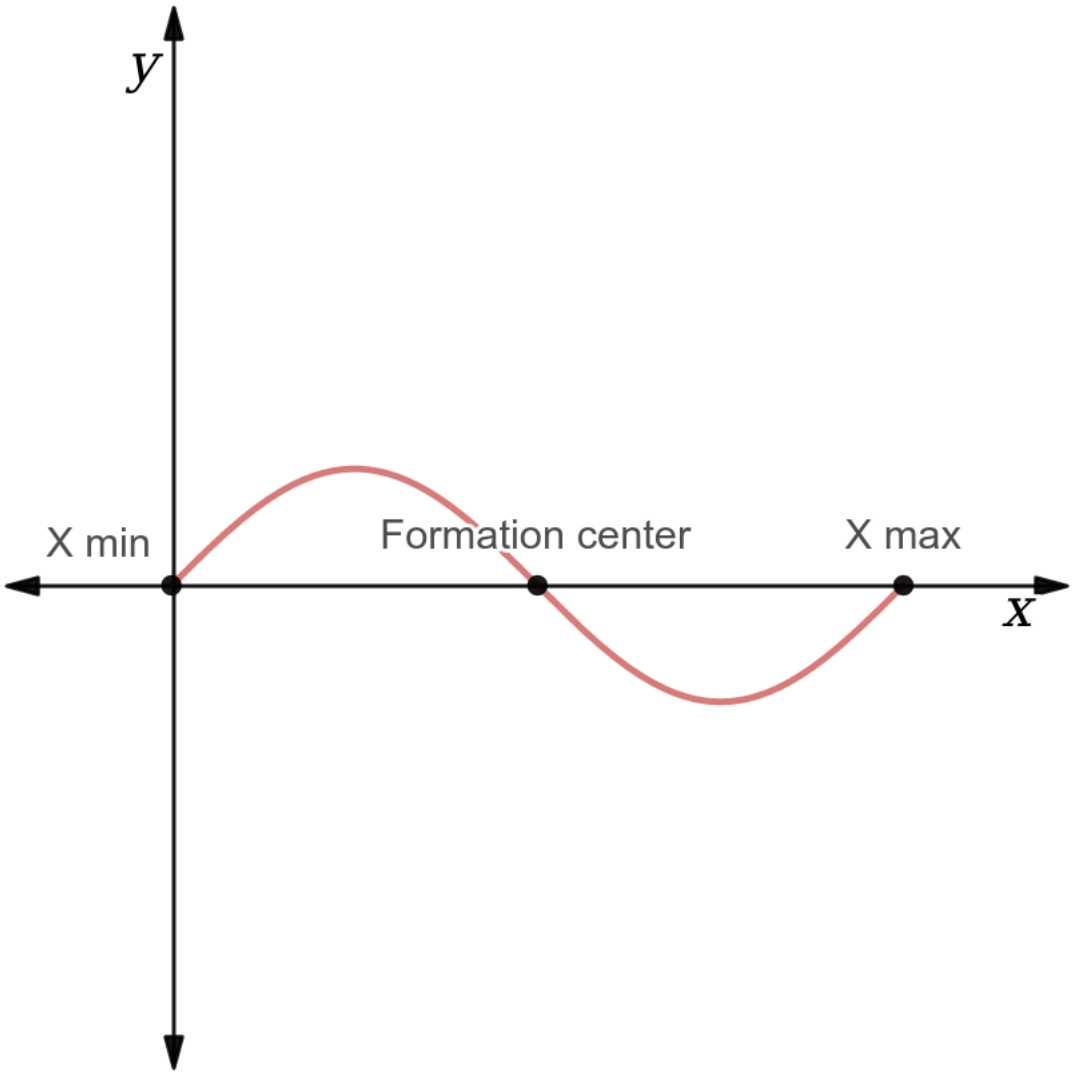

Fig. 3: Sinus formation

- Formation center: I choose the formation center to be at the middle of the the formation X axis
- Scale: Scale will be the length of the formation
- Agents repartition: Agents will be equally distant on the X axis. Y position will be $A*sin(\omega*x)$.

2. Create new formation file

File path: `.../crazyflie_charles/ros_ws/src/formation_manager/scripts/sin_formation.py`

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

"""Sinus formation
"""

from math import sin, cos, pi
import rospy
from crazyflie_driver.msg import Position

from general_formation import FormationClass, compute_info_from_center

class SinFormation(FormationClass):
    """Sinus formation

    Notes:
        scale: Total lenght of period
    """
    def __init__(self, min_dist):
        super(SinFormation, self).__init__(min_dist)

        # TODO: Add formation specific attributes

        self.compute_min_scale()

    # Setter
    def set_n_agents(self, n_agents):

        # TODO: Verify number of agents is valid

        self.update_formation_scale()
        self.compute_min_scale()

    # Computing
    def compute_min_scale(self):
        # TODO: Compute min scale where min distance between agents is R_
↪MIN
        pass

    def compute_formation_positions(self):
        for i in range(self._n_agents):
            if rospy.is_shutdown():
                break

            # Initialize agent formation goal
            self._agents_goals[i] = Position()

            # Compute formation position
            # TODO: Compute agent position from center

            # Compute information from center
            center_dist, theta, center_height = compute_info_from_
↪center([x_dist, y_dist, z_dist])
            self._center_dist[i] = center_dist
            self._angle[i] = theta
            self._center_height[i] = center_height
```

(continues on next page)

(continued from previous page)

```
        return self._agents_goals

    def update_formation_scale(self):
        #TODO: Update formation scale
        pass
```

3. Init method

   Two atributes will be required: *x_agents_dist* and *frequency*, to vary formation length based on scale.

   Another attribute will be added to easily change the sinus amplitude.

```
def __init__(self, min_dist):
    super(SinFormation, self).__init__(min_dist)

    self.agents_x_dist = 0 # [m]
    self.frequency = 0 # [rad]
    self.amplitude = 1 # [m]

    self.compute_min_scale()
```

4. Verify number of agents

   Before setting a new number of agents, it's important to make sure the number is valid. For instance, only perfect square numbers are valid for the **square formation**.

   However, all numbers are valid for the sinus formation.

```
def set_n_agents(self, n_agents):
    # All numbers are valid
    if n_agents > 0:
        self._n_agents = n_agents
        self._n_agents_landed = 0
    else:
        rospy.loginfo("Formation: Unsuported number of agents, landing %i␣
↪agents"\
            % self._n_agents_landed)

    rospy.loginfo("Formation: %i agents in formation" % self._n_agents)

    self.update_formation_scale()
    self.compute_min_scale()
```

5. Compute formation attributes based on scale

   First let's compute the distance between agents

```
self.agents_x_dist = self._scale / (self._n_agents - 1) if self._n_agents␣
↪> 1 else 0
```

   And then the formation frequency

```
self.frequency = (2*pi)/self._scale if self._scale > 0 else 0
```

```
def update_formation_scale(self):
    self.agents_x_dist = self._scale / (self._n_agents - 1) if self._n_
↪agents > 1 else 0
```

(continues on next page)

(continued from previous page)

```
    self.frequency = (2*pi)/self._scale if self._scale > 0 else 0
```

6. Find minimum scale

   The minimum scale is defined as the scale where the minimal distance between two agents is `R_MIN`. For this formation, to simplify calculations, we will consider as if the formation was a simple line.

   Hence, the min_scale is when the distance between agents is equal to `R_MIN` (or `_min_dist`)

```
def compute_min_scale(self):
    if self._n_agents > 1:
        self._min_scale = self._min_dist*(self._n_agents - 1)
    else:
        self._min_scale = 0.0
```

7. Compute agents position from center

   We have to compute each agent position in x, y, and z from formation center.

   - X position

```
x_dist = self.agents_x_dist*i - center_offset
```

Note: `center_offset = self._scale/2` is substracted from X position since the first agent is not at the center.

   - Y positions

```
y_dist = self.amplitude*sin(self.frequency*x_dist)
```

   - Z positions

```
z_dist = 0
```

   Completed function

```
def compute_formation_positions(self):
    center_offset = self._scale/2 # New line

    for i in range(self._n_agents):
        if rospy.is_shutdown():
            break

        # Initialize agent formation goal
        self._agents_goals[i] = Position()

        # Compute formation position
        x_dist = self.agents_x_dist*i - center_offset # New line
        y_dist = self.amplitude*sin(self.frequency*x_dist) # New line
        z_dist = 0 # New line

        # Compute information from center
        center_dist, theta, center_height = compute_info_from_center([x_
↪dist, y_dist, z_dist])
        self._center_dist[i] = center_dist
```

(continues on next page)

(continued from previous page)

```
        self._angle[i] = theta
        self._center_height[i] = center_height

    return self._agents_goals
```

8. Add formation to `formation_manager_ros`

```
# .../crazyflie_charles/ros_ws/src/formation_manager/scripts/formation_
↪manager_ros.py``
...
from sin_formation import SinFormation # New line
...

class FormationManager(object):
    ...
    def __init__(self, cf_list, min_dist, start_goal):
        ...
        #: All possible formations
        self._formations = {"square": SquareFormation(self._min_dist),
                            "v": VFormation(self._min_dist),
                            "pyramid": PyramidFormation(self._min_dist),
                            "circle": CircleFormation(self._min_dist),
                            "line": LineFormation(self._min_dist),
                            "sin": SinFormation(self._min_dist),} # New␣
↪line
        ...
    ...
```

9. Test new formation

```
swarm = SwarmAPI()
swarm.set_mode("formation")
swarm.set_formation("sin")
swarm.take_off()
```

Note: Completed sinus formation file can be found in `.../crazyflie_charles/demos`

Add New Method to API

In this tutorial, we will learn all the steps required to add a new method to the Python API. We are going to add a method that moves a single crazyflie in a **small circle**.

Note: For this tutorial, it's important that you have some knowledge of ROS. If not, I recommend you to follow these tutorials (begginner level) first.

To know which files to modify, it's important to understand the package *ROS architecture*.

There are three main layers:

- crazyflie_ros: ROS stack to control the crazyflies with the crazyradio

- swarm_manger: Package to compute swarm positions and command
- SwarmAPI: To send command to the swarm via a Python script

This architecture was used to simplify each package.

1. Create a new service

   Commands are sent to `swarm_manager` using a ROS service.

   1.1 Create service

   Since `SwarmAPI` and `swarm_controller` node comminucate using ros services, we first need to create a new service.

   The new service, will allow to specify the name of the crazyflie and return an error if the name is invalid.

   Create a new file named `CircleCf.srv` in `.../crazyflie_charles/ros_ws/src/swarm_manager/srv/`

```
string cf_name
---
bool valid_cf
```

   1.2 Add service to CMakeList

```
...
## Generate services in the 'srv' folder
add_service_files(
FILES
...
CircleCf.srv
)
...
```

   1.3 Build catkin workspace

```
$ cd .../crazyflie_charles/ros_ws
$ catkin_make
```

2. Write method in `swarm_manager` package

   2.1 Write `_circle_cf_srv` method

```
# .../crazyflie_charles/ros_ws/src/swarm_manager/swarm_controller_
↪ros.py
...
from math import sin, cos, pi
from swarm_manager.srv import CircleCf
...
class SwarmController(object):
    ...
    def _circle_cf_srv(self, srv_req):
        cf_id = srv_req.cf_name
        valid_cf = True

        self._state_machine.set_state("hover")

        if cf_id in self._crazyflies:
            # Traj parameters
```

(continues on next page)

(continued from previous page)

```
            circle_radius = 0.5 # [m]
            circle_time = 5 # [sec]
            n_points = int(circle_time/0.1)  # 0.1 is publish rate

            start_pose = self._crazyflies[cf_id].pose.pose.position
            circle_center = [start_pose.x - circle_radius, start_
↪pose.y, start_pose.z]

            for i in range(n_points + 1):
                cur_angle = i*(2*pi)/n_points

                self._crazyflies[cf_id].goals["goal"].x = circle_
↪center[0] +\
                                                          circle_
↪radius*cos(cur_angle)
                self._crazyflies[cf_id].goals["goal"].y = circle_
↪center[1] +\
                                                          circle_
↪radius*sin(cur_angle)

                self._rate.sleep()
        else:
            valid_cf = False

        return {'valid_cf': valid_cf}
```

2.2 Add new service to `swarm_controller` node

```
# .../crazyflie_charles/ros_ws/src/swarm_manager/swarm_controller_
↪ros.py``
...
class SwarmController(object):
...
    def _init_services(self):
        # Services
        ...
        rospy.Service('/circle_cf', CircleCf, self._circle_cf_srv)
        ...
...
```

3. Add method to API

3.1 Subscribe to new service

```
# .../crazyflie_charles/ros_ws/src/swarm_manager/swarm_api/api.py``
...
from swarm_manager.srv import CircleCf
...
class SwarmAPI(object):
    ...

    def _init_services(self):
        # Subscribe to srvs
        rospy.loginfo("API: waiting for services")
        ...
        self._link_service('circle_cf', CircleCf)
        ...
```

(continues on next page)

(continued from previous page)

```
    ...
```

3.2 Write method

```
# .../crazyflie_charles/ros_ws/src/swarm_manager/swarm_api/api.py``
...
class SwarmAPI(object):
    ...

    def circle_cf(self, cf_id):
        """Circle specified crazyflie around a 0.5m radius

        Note:
            This method only works in 'Automatic' mode

        Args:
            cf_id (str): Id of crazyflie
        """
        if self.current_mode != "automatic":
            rospy.logerr("Swarm needs to be in automatic mode")

        else:
            srv_res = self._services["circle_cf"](cf_name=str(cf_
↪id))
            valid_cf = srv_res.valid_cf

        if not valid_cf:
            rospy.logerr("%s is an invalid crazyflie name" % cf_id)
    ...
```

3.3 Add method to documentation

```
.. .../crazyflie_charles/docs/python_api.rst

...

.. autosummary::
    ...
    circle_cf

...
```

4. Test new method

```
swarm = SwarmAPI()

swarm.set_mode("automatic")
swarm.take_off()
rospy.sleep(3)

swarm.circle_cf("cf_0")

rospy.spin()
```

Update Project Documentation

To build on your machine

1. Go to docs folder

```
$ cd .../crazyflie_charles/docs
```

2. Build html

```
$ make html
```

3. Open `.../crazyflie_charles/docs/_build/html/index.html` in your browser

To update online doc

The project's documentation is hosted here.

Documentation should automaticaly rebuild when a new commit is pushed on *master*.

Note: All documentation is written in reStructuredText:

- Rst tutorial
- Sphinx and rtd tutorial.

MRASL Lab Demo

Cette section a pour but de permettre de facilement mettre en place la démonstration pour le laboratoire.

Installation des ancres

Pour une précision optimale, il est recommandé d'utiliser les 8 ancres LPS. Par contre, la demonstration fonctionne aussi avec seulement 6.

1 - Installer les ancres sur les 4 supports. L'ancre du bas devrait être à environ 20cm du sol et celle du haut à environ 2m.

2 - Disposer les ancres aux 4 coins de l'arène

3 - Mesurer la position des ancres

- Pour faciliter cette étape, je recommande d'utiliser le VICON

4 - Mettre à jour la position des ancres à l'aide de l'interface Bitcraze

5 - Alimenter les ancres

- Le plus facile est d'utliser des power banks

Fig. 4: Vue de l'arène

Lancer la démonstration

La démonstration a été testée et optimisée pour 5 drones. Cependant, elle devrait fonctionner avec n'importe quel nombre.

Note: Il est recommandé de tester les drones individuellement pour s'assurer qu'ils volent bien.

1 - Placer les drones dans l'arène

Leur position initiale n'est pas importante. Il faut seulement qu'ils soient tous à au moins 1 mètre de distance l'un de l'autre

2 - Lancer le serveur

Le projet a été installé sur l'ordinateur à droite en entrant au labo. Il se trouve à l'emplacement: `~/crazyflie_ws/crazyflie_charles`.

Ne pas oublier de mettre à jour la branche `master`.

```
$ roslaunch swarm_manager launch_swarm.launch
```

Warning: N'oublier pas d'avoir `roscore` qui roule dans un terminal

3 - Démarrer la démonstration

```
$ cd projects/crazyflie_charles/demos
$ python mrasl_demo.py
```

Fig. 5: Alimentation de deux ancres

CHAPTER 2

Python API

Python API

2.1 Python API

start_joystick
link_joy_button
take_off
stop
emergency
land
set_mode
set_formation
next_formation
prev_formation
inc_scale
dec_scale
toggle_ctrl_mode
go_to
get_positions
rotate_formation

Python API to control the swarm.

This module maked it possible to easily send command to the swarm through a Python script.

2.1.1 Example

```
# Formation exemple
swarm = SwarmAPI()

# Link joystick buttons to commands
swarm.start_joystick("ds4")
swarm.link_joy_button("S", swarm.take_off)
swarm.link_joy_button("X", swarm.land)
swarm.link_joy_button("O", swarm.emergency)
swarm.link_joy_button("T", swarm.toggle_ctrl_mode)

# Start swarm
swarm.set_mode("formation")
swarm.set_formation("v")

swarm.take_off()
rospy.sleep(10)

# Change formation
swarm.set_formation("pyramid")
```

2.1.2 `SwarmAPI` class

class swarm_api.api.**SwarmAPI**

Python API class

start_joystick(*joy_type*, *joy_dev='js0'*)

Initialize joystick node. See *here* for a tutorial on how to add new joystick types.

Possible types:

- ds4

Parameters

- **joy_type** (str) – Controller type.
- **joy_dev** (str, Optional) – Specify joystick port. Defaults to *js0*

set_joy_control(*to_control*)

To enable/disable control of formation position with joystick axes.

Parameters **to_control** (*bool*) – If True, formation will be moved by joystick axes

link_joy_button(*button_name*, *func*, *args=None*, *kwargs=None*)

Link a button to a function call

Parameters

- **button_name** (str) – Name of button, as written in `joy_conf.yaml`.
- **func** (Callable) – Function to call
- **args** (*optional*) – Function args. Defaults to None. Can be a single arg or a list of args
- **kwargs** (dict, optional) – Function kwargs. Defaults to None.

Raises KeyError – Invalid button name

Example:

```
swarm.start_joystick("ds4")
swarm.link_joy_button("S", swarm.take_off)
swarm.link_joy_button("X", swarm.land)
```

take_off()
Take off all landed CFs.

Modify `take_off_height` in `swarm_conf.yaml` to change take off height

Note: Will only take off landed CFs

---

stop()
Stop all CFs

emergency()
Call emergency srv of all CFs

land()
Land all CFs at their starting position.

set_mode(*new_mode*)
Set `SwarmController` control mode.

Possible modes are:

- Automatic: CF will plot trajectory to new goals. Send `go_to commands` from python script
- Formation: Swarm moves in formation. Formation position can be moved /w joystick.

Not implemented:

- Pilot: Like CF client. No formation
- Assisted: Control change of position /w joystick. No formation.

Note: Modes are not case sensitive

Parameters **new_mode** (`str`) – New control mode

set_formation(*formation_name*)
Set swarn formation

Parameters **formation_name** (`str`) – New formation name

next_formation()
Go to next swarm formation

prev_formation()
Go to prev swarm formation

inc_scale()
Increase formation scale by 0.5

dec_scale()
Decrease formation scale by 0.5

toggle_ctrl_mode()
Toggle control mode between absolute and relative.

In absolute: x, y, z are world axis

In relative: x, y, z depends on swarm orientation

go_to(*goals*)
Move formation and/or cf to a position using the trajectory planner.

Dict format: `"goal_name": [x, y, z, yaw]` where `"goal_name"` is either `"formation"` or `"cf_x"`

X, Y, Z in meters, Yaw in rad.

Example:: # To move formation to [2, 2, 0.5, 1.57] swarm.go_to({'formation': [2, 2, 0.5, 1.57]})

To move cf_0 to [0, 0, 0.5] and cf_1 to [1, 1, 1] goals = {} goals["cf_0"] = [0, 0, 0.5, 0] goals["cf_1"] = [1, 1, 1, 0] swarm.go_to(goals)

Parameters **goals** (dict) – New goals

get_positions(*cf_list=None*)
Get current position of crazyflies

If `cf_list` is `None`, will return position of all Cfs.

Parameters **cf_list** (list, optional) – List of cf to read positions. Defaults to None.

Returns Positions of CFs read. `"{cf_id": [x, y, z, yaw], ...}`

Return type dict

rotate_formation(*angle*, *duration*)
Rotate formation around it's center

Note: Formation control with joystick must be False to use this function `swarm.set_joy_control(False)`

Parameters

- **angle** (*float*) – Angle to turn [deg]
- **duration** (*float*) – Rotation duration [sec]

CHAPTER 3

Files Tree

```
|-- README.md
|-- build.sh: Project build script
|-- pc_persmissions.sh
|-- requirements.txt
|-- docs: Folder with all thing related to documentation
|-- demos: Example scripts
|
|-- flight_data
    |-- flight_analysis.py: Script to analyse flight flight data
    |-- user_command.py: Script to read user input
    |-- all flight data...
|
|-- ros_ws
    |-- build
    |-- devel
    |-- src
        |-- CMakeLists.txt
        |-- crazyflie_ros
        |-- formation_manager
        |-- swarm_manager
        |-- trajectory_planner
```

CHAPTER 4

ROS Architecture

To control the swarm, three different ros packages are used:

- *Swarm Manager*: Main package. Link between the other packages and crazyflie ros stack. Includes a python api.
- *Formation Manager*: To move the swarm in a specific formation (i.e square, circle, . . .)
- *Trajectory Planner*: To move agents without collisions. Used to change formation

The general architecture can be found here *General Architecture*.

4.1 General Architecture

Overview of the architecture used:

For an in depth description of each ros package:

- *Swarm Manager*
- *Formation Manager*
- *Trajectory Planner*

For a description of all ros topics used, see *ROS Topics*.

Note: Nodes also interact using ros services.

4.2 Swarm Manager

Overview of package architecture:

For an in depth description of each ros node:

- *SwarmAPI*: Python API
- *swarm_controller*: Main node. Controls commands sent to all CFs.
- *joy_controller*: Send joystick data to swarm_manager.
- *cf_controller*: Controls a single crazyflie to match swarm_manager output.
- *cf_sim*: Simulate position of a crazyflie.
- *cf_broadcaster*: Broadcast position of a crazyflie to view in RVIZ
- *flight_recorder*: To record and save all CFs trajectories.

For a description of all ros topics used, see *ROS Topics*.

4.2.1 SwarmAPI

Node to connect python API to ros.

ROS Features

Subscribed Topics

Nones

Published Topics

None

Services

/joy_button(swarm_manager/JoyButton) To get button pressed on joystick

Services Called

/swarm_emergency(std_srvs/Empty) From *swarm_controller*

/stop_swarm(std_srvs/Empty) From *swarm_controller*

/take_off_swarm(std_srvs/Empty) From *swarm_controller*

/land_swarm(std_srvs/Empty) From *swarm_controller*

/set_mode(swarm_manager/SetMode) From *swarm_controller*

/go_to(swarm_manager/SetGoals) From *swarm_controller*

/get_positions(swarm_manager/GetPositions) From *swarm_controller*

/set_swarm_formation(formation_manager/SetFormation) From *swarm_controller*

/next_swarm_formation(std_srvs/Empty) From *swarm_controller*

/prev_swarm_formation(std_srvs/Empty) From *swarm_controller*

/inc_swarm_scale(std_srvs/Empty) From *swarm_controller*

/dec_swarm_scale(std_srvs/Empty) From *swarm_controller*

/toggle_ctrl_mode(std_srvs/Empty) From *swarm_controller*

Parameters

None

4.2.2 swarm_controller

To control flight of the swarm. By using a state machine, this node will determine each crazyflie goal and control it's desired position by publishing to */cfx/goal*.

The goal will change depending of the services called by SwarmAPI.

Note: See *Glossary* for definition of Swarm and Formation

ROS Features

Subscribed Topics

/cfx/formation_goal **(crazyflie_driver/Position)** Position of the CF in formation

/cfx/trajectory_goal **(crazyflie_driver/Position)** Position of the CF on the trajectory, at each time step

/cfx/state **(std_msgs/String)** Current state of CF

/cfx/pose **(geometry_msgs/PoseStamped)** Current pose of CF

/joy_swarm_vel **(geometry_msgs/Twist)** Swarm velocity

Published Topics

/cfx/goal **(crazyflie_driver/Position)** Target position of CF

/formation_goal_vel **(geometry_msgs/Twist)** Formation center goal variation

Services

/take_off_swarm(std_srvs/Empty) Take off all CFs

/stop_swarm(std_srvs/Empty) Stop all CFs

/swarm_emergency(std_srvs/Empty) Emgergency stop of all CFs

/land_swarm(std_srvs/Empty) Land all CF to their starting position

/get_positions(swarm_manager/GetPositions) Get current position of CFs

/go_to(swarm_manager/SetGoals) Move CFs or formation to specified positions

/set_mode(swarm_manger/SetMode) Set control of swarm_controller

/set_swarm_formation(formation_manager/SetFormation) Set swarm to a formation

/inc_swarm_scale(std_srvs/Empty) Increase scale of formation

/dec_swarm_scale(std_srvs/Empty) Decrease scale of formation

/next_swarm_formation(std_srvs/Empty) Go to next formation

/prev_swarm_formation(std_srvs/Empty) Go to previous formation

/traj_found(std_srvs/SetBool) To call once the trajectory planner is done

/traj_done(std_srvs/Empty) To call once the trajectory is done

Services Called

/set_formation(formation_manager/SetFormation) From *Formation Manager*

/get_formations_list(formation_manager/GetFormationList) From *Formation Manager*

/formation_inc_scale(std_srvs/Empty) From *Formation Manager*

/formation_dec_scale(std_srvs/Empty) From *Formation Manager*

/set_planner_positions(trajectory_planner/SetPositions) From *Trajectory Planner*

/plan_trajectories(std_srvs/Empty) From *Trajectory Planner*

/pub_trajectories(std_srvs/Empty) From *Trajectory Planner*

Parameters

~n_cf(int)

~take_off_height(float)

~gnd_height(float)

~min_dist(float)

~min_goal_dist(float)

4.2.3 joy_controller

To control the swarm with a joystick.

Sends button pressed to *SwarmAPI* through `swarm_manager/JoyButton` service.

ROS Features

Subscribed Topics

/joy(sensor_msgs/Joy) Joystick input

Published Topics

/joy_swarm_vel **(geometry_msgs/Twist)** Swarm velocity

Services

None

Services Called

/joy_button(swarm_manager/JoyButton) From *SwarmAPI*

Parameters

~joy_topic(str, default:"joy")

~sim(bool, default: False)

~teleop(bool, default: False)

~x_axis(int, default: 4)

~y_axis(int, default: 3)

~z_axis(int, default: 2)

~yaw_axis(int, default: 1)

~x_velocity_max(float, default: 2.0)

~y_velocity_max(float, default: 2.0)

~z_velocity_max(float, default: 2.0)

~yaw_velocity_max(float, default: 2.0)

~x_goal_max(float, default: 0.05)

~y_goal_max(float, default: 0.05)

~z_goal_max(float, default: 0.05)

~yaw_goal_max(float, default: 0.05)

4.2.4 cf_controller

Node to control a single crazyflie. Each CF has it's own node.

ROS Features

Subscribed Topics

/cfx/goal **(crazyflie_driver/Position)** Goal of crazyflie

/cfx/pose **(geometry_msgs/PoseStamped)** Current pose of CF

Published Topics

/cfx/state **(std_msgs/String)** Current state of CF

/cfx/cmd_position **(crazyflie_drive/Position)** Move CF to absolute position

/cfx/cmd_hovering **(crazyflie_driver/Hover)** Hover CF

/cfx/cmd_vel **(geometry_msgs/Twist)** Control velocity of CF

/cfx/cmd_stop **(std_msgs/Empty)** Stop CF

Services

/take_off(std_srvs/Empty) Take off CF

/hover(std_srvs/Empty) Hover CF

/land(std_srvs/Empty) Land CF

/stop(std_srvs/Empty) Stop CF

/set_param (swarm_manager/SetParam) Set a parameter

Services Called

None

Parameters

~cf_name(str, default: cf_default)

~sim(bool, default: False)

~take_off_height(float)

~gnd_height(float)

4.2.5 cf_sim

Class to act as the crazyflie in simulation. Publish position of CF based on cmd_x received

ROS Features

Subscribed Topics

/cfx/cmd_position **(crazyflie_driver/Position)** Position command

/cfx/cmd_hovering **(crazyflie_driver/Hover)** Hovering command

/cfx/cmd_stop **(std_msgs/Empty)** Stop CF

/cfx/cmd_vel **(geometry_msgs/Twist)** Velocity of CF

Published Topics

/cfx/pose **(geometry_msgs/PoseStamped)** Current pose of CF

Services

/cfx/emergency(std_srvs/Empty) Simulation of emergency service

Services Called

None

Parameters

~cf_name"(str)

/starting_positions")[cf_name](list of float)

`CrazyflieSim` class

class crazyflie_sim.**CrazyflieSim**(*cf_id*, *starting_pos*)
To simulate position of CF based on received cmd.

send_pose()
Publish current pose of CF

emergency(_)
Sim emergency service

4.2.6 cf_broadcaster

Node to create a frame from the crazyflie current position

ROS Features

Subscribed Topics

/cfx/pose **(geometry_msgs/PoseStamped)** Current pose of CF

Published Topics

/tf Pose for TF package

Services

None

Services Called

None

Parameters

~world(str, world)

~cf_name(str)

~frame(str)

crazyflie_tf.**handle_crazyflie_pose**(*msg*, *args*)

Broadcast pose to tf

Parameters

- **msg** (*PoseStamped*) – CF current pose
- **args** (*list of str*) – [frame, world]

4.2.7 flight_recorder

To record and analyse flight data of all CFs

Note: See `flight_analysis.py` for plotting and analysis of flight data

ROS Features

Subscribed Topics

/cfx/goal **(crazyflie_driver/Position)** Target position of CF

/cfx/pose **(geometry_msgs/PoseStamped)** Current pose of CF

Published Topics

None

Services

None

Services Called

None

Parameters

~cf_list(list of str)

~save(bool, false)

`Recorder` Class

class flight_recorder.**Recorder**(*cf_list*, *to_save*)

To record flight data of all CFs

crazyflies = None

To store positions of all CFs. Keys are CF id

Type dict

on_shutdown()

To save data upon rospy shutdown

4.3 Formation Manager

To manage the formation of the swarm

In charge of computing the positions of all the crazyflies.

4.3.1 Avalaible formations

- Square
- Pyramid
- Circle
- V
- Ligne

4.3.2 ROS Features

Subscribed Topics

/formation_goal_vel **(geometry_msgs/Twist)** Formation center goal variation

Published Topics

/cfx/formation_goal **(crazyflie_driver/Position)** Goal of a single CF in the formation

/formation_goal **(crazyflie_driver/Position)** Goal of formation

/formation_pose **(geometry_msgs/Pose)** Position of the formation

Services

/set_formation(formation_manager/SetFormation) Set swarm to a formation

/formation_inc_scale(std_srvs/Empty) Increase scale of formation

/formation_dec_scale(std_srvs/Empty) Decrease scale of formation

/toggle_ctrl_mode(std_srvs/Empty) To change between absolute and relative control mode

/get_formations_list(formation_manager/GetFormationList) Return a list with all possible formations

Services Called

None

Parameters

~n_cf(int)

4.4 Trajectory Planner

Package to compute collision free trajectories for each agent.

The algorithm [CIT3] used is based on Distributed Model Predictive Control.

4.4.1 Usage

1. Set start position and goals of each agent to compute trajectories(`/set_planner_positions` srv)
2. Start trajectory solver (`/plan_trajectories` srv)
3. Wait for trajectory solver to be done
4. Start trajectory publishing (`/pub_trajectories` srv)
5. Wait for trajectory publishing to be done

4.4.2 ROS Features

Subscribed Topics

None

Published Topics

/cfx/trajectory_goal(crazyflie_driver/Position) Position of the CF on the trajectory, at each time step

Services

/set_planner_positions(crazyflie_charles/SetPositions) Set position (start or goal) of each crazyflie. See *TrajectoryPlanner.set_positions()* for more information.

/plan_trajectories(std_srvs/Empty) Start solver to find a trajectory for each crazyflie.

/pub_trajectories(std_srvs/Empty) Call to start publishing all trajectories

Services Called

/traj_found(std_srvs/SetBool) Service called once a trajectory is found.

data is True if valid trajectories are found.

data is false otherwise (collision, no solution in constraints).

/traj_done(std_srvs/Empty) Service called once the last step of the trajectory is reached.

Parameters

~cf_list(str, default: ['cf1'])

4.4.3 *TrajectoryPlanner* Class

class trajectory_planner_ros.**TrajectoryPlanner**(*cf_list*, *solver_args*)

To plan trajectories of CFs

agents_dict = None

dict of str: Keys are the id of the CF. Items are a dict containing: Agent, trajectory_publisher, start_yaw

Type dict of str

to_plan_trajectories = None

True if a trajectories are to be planned

Type bool

trajectory_found = None

True if a trajectory has been found for current position

Type bool

to_publish_traj = None

True if a trajectories are to be published

Type bool

set_positions(*srv_req*)

Set start position or goal of each agent

Parameters **srv_req** (*SetPositions*) – List /w position type(start or goal) and positions of each agent

plan_trajectories(_)

Start trajectory planning of each agent

start_publishing_srv(_)

Start publishing CF trajectories

publish_trajectories()
Publish computed trajectory

run_planner()
Execute the correct method based on booleen states

4.5 ROS Topics

Description of the ros topics used.

Topic	Msg type	Description
/cfx/pose	(geometry_msgs/PoseStamped)	Pose of a CF
/cfx/state	(std_msgs/String)	State of a CF
/cfx/cmd_position	(crazyflie_drive/Position)	Position command
/cfx/cmd_hovering	(crazyflie_driver/Hover)	Hovering command
/cfx/cmd_vel	(geometry_msgs/Twist)	Velocity of CF
/cfx/cmd_stop	(std_msgs/Empty)	Stop CF
/cfx/trajectory_goal	(crazyflie_driver/Position)	Trajectory goal of a CF
/cfx/formation_goal	(crazyflie_driver/Position)	Formation goal of a CF
/formation_goal	(crazyflie_drive/Position)	Goal of formation
/formation_pose	(geometry_msgs/Pose)	Position of formation
/formation_goal_vel	(geometry_msgs/Twist)	Formation goal velocity
/joy_swarm_vel	(geometry_msgs/Twist)	Joystick swarm command

4.5.1 Crazyflie

/cfx/pose

(geometry_msgs/PoseStamped)

Current pose of CF

/cfx/state

(std_msgs/String)

Current state of CF

/cfx/goal

(crazyflie_driver/Position)

Target position of CF

/cfx/cmd_position

(crazyflie_drive/Position)

Position command

/cfx/cmd_hovering

(crazyflie_driver/Hover)

Hovering command

/cfx/cmd_vel

(geometry_msgs/Twist)

Velocity of CF

/cfx/cmd_stop

(std_msgs/Empty)

Stop CF

4.5.2 Swarm

/cfx/trajectory_goal

(crazyflie_driver/Position)

Position of the CF on the trajectory, at each time step

/cfx/formation_goal

(crazyflie_driver/Position)

Goal of a single CF in the formation

/formation_goal

(crazyflie_driver/Position)

Goal of formation

/formation_pose

(geometry_msgs/Pose)

Position of the formation

/formation_goal_vel

(geometry_msgs/Twist)

Formation center goal variation

/joy_swarm_vel

(geometry_msgs/Twist)

Swarm velocity, from joystick

4.6 Swarm Manager

Module to move the swarm and use `TrajectoryPlanner` and `FormationManager` packages.

4.6.1 `SwarmController` class

class swarm_controller.**SwarmController**

Main class

control_swarm()

Main method, publish on topics depending of current state

update_swarm_param(*param*, *value*)

Update parameter of all swarm

Parameters

- **param** (*str*) – Name of parameter
- **value** (*int64*) – Parameter value

check_collisions()

To check that the minimal distance between each CF is okay.

If actual distance between CF is too small, calls emergency service.

update_formation(*formation_goal=None*)

To change formation of the swarm.

Formation center will be *formation_goal* if is specified. If not, it will be stay at the same place

Parameters **formation_goal** (*list, optional*) – New formation goal: [x, y, z, yaw]. Defaults to None.

go_to_goal(*target_goals=None*, *land_swarm=False*)

Controls swarm to move each CF to it's desired goal while avoiding collisions.

If some CFs are in extra (in formation) will land them.

Handles extra CF landing.

Parameters

- **land_swarm** (*bool, optional*) – If true, land swarm to initial pos. Defaults to False.
- **goals** (*dict, optional*) – To specify target goals. Used when not in formation mode. Defaults to None.

4.6.2 `CrazyfliePy` class

class swarm_controller.**CrazyfliePy**(*cf_id*)
Link between the swarm manager and a crazyflie.

Controls one robot.

publish_goal()
Publish current CF goal

call_srv(*srv_name*, *args=None*, *kwargs=None*)
Call a CF service

Parameters

- **srv_name** (*str*) – Name of srv to call
- **args** (*list, optional*) – Service args. Defaults to None.
- **kwargs** (*dict, optional*) – Service kwargs. Defaults to None.

update_initial_pose()
Update initial position to current pose

update_goal(*goal_name=''*)
Update current goal to one of other CF goal (trajectory, formation or current goal)

Notes: "" or "goal": Set to current goal "traj" or "trajectory": Set to goal to trajectory goal "formation: Set goal to formation goal

Parameters **goal_name** (*str, optional*) – To specify goal to set. Defaults to "".

4.7 Formation Manager

Module to compute the position of agents in a formation.

A formation is defined as a certain number of agents moving in a specific shape.

Note: An agent is NOT a crazyflie. A crazyflie is the actual robot. An agent is a position in a formation

This package uses two classes:

- *FormationManager* to switch between formations
- FormationClass to compute the position of each agent for a formation. One for each formation.

To set a new formation, set_formation service is called. The new positions of agents will be computed using the desired FormationClass child class.

Based on the position of each crazyflie, *FormationManager* will then link each CF to an agent. Once a CF is linked to an agent, this CF will follow this agent's position in the formation.

4.7.1 `FormationManager` class

class formation_manager_ros.**FormationManager**(*cf_list*, *min_dist*, *start_goal*)
To change between formations and associate agents and CFs

Parameters

- **cf_list** (list of str) – List of all CFs in swarm
- **min_dist** (float) – Minimum distance between agents in formation
- **start_goal** (list of float) – Formation start goal

Variables

- **abs_ctrl_mode** (bool) – In abs ctrl mode, moves in world/ In rel ctrl mode, moves relative to yaw
- **scale** (float) – Formation scale
- **formation** (str) – Current formation
- **extra_agents** (list of str) – Id of CFs in extra

init_formation(*initial_positions*)
Initialize formation goal and cf positions.

initial_positions used to associate CF and agents

Parameters **initial_positions** (*dict of list*) – Keys: Id of CF, Items: Initial position [x, y, z]

link_swarm_and_formation()
Link each agent of formation to a CF of the swarm and initialize formation goals

run_formation()
Execute formation

4.7.2 FormationClass class

Abstract Class that represents a general formation.

Cylindrical coordinates are used to specify the position of each agent in the formation.Center of formation is defined as (0, 0, 0)

class general_formation.**FormationClass**(*min_dist*)
Basic formation type

Parameters **min_dist** (float) – Minimum distance between agents in formation

get_agents_goals()
Return goal of each agent in formation

Returns Keys: agents id, Items: goal [x, y, z]

Return type dict of list

set_scale(*new_scale*)
Set scale of the formation

Parameters **new_scale** (*float*) – New formation scale

update_agents_positions(*formation_goal*, *crazyflies=None*)
Compute goal of each agent and updates corresponding CF goal.

Agent goal is calculated based on distance and angle of agent id from formation center.

If crazyflies is not None, will update their goal.

Parameters

- **formation_goal** (*Position*) – Goal of the formation

- **crazyflies** (*dict, optional*) – Information of each Crazyflie

find_extra_agents()
Find id of agents in extra.

Extra agents can happen when there are too many agents for a formation. i.e: Making a square with 10 agents will result in one extra agent.

set_n_agents(*n_agents*)
Make sure there number of CF is supported by formation and sets it

Parameters **n** (*int*) – Number of CF

compute_min_scale()
Find minimum scale to make sure distance between agents is greater than min_dist

compute_formation_positions()
Compute position of each agent from formation center

Position are defined by radius from center (x, y plane), height from center and radius angle

update_formation_scale()
Compute new formation information after the scale is changed.

i.e: Distance/angle between agents

Unique to each formation

general_formation.**compute_info_from_center**(*agent_position*)
Calculate distance and angle from formation center

Formation center is considered to be at 0, 0, 0

Parameters **cf_position** (*list of float*) – Position from [0 , 0 ,0][x, y, z]

Returns [distance from center, angle from center, height from center]

Return type list of float

general_formation.**calculate_rot**(*start_orientation*, *rot*)
Apply rotation to a quaternion

Parameters

- **start_orientation** (*Quaternion*) – Initial orientation
- **rot** (*Vector3*) – Angular speed to apply

Returns Result

Return type Quaternion

general_formation.**yaw_from_quat**(*quaternion*)
Returns yaw from a quaternion

Parameters **quaternion** (*Quaternion*) –

Returns Yaw

Return type float

general_formation.**quat_from_yaw**(*yaw*)
Compute a quaternion from yaw

Pitch and roll are considered zero

Parameters **yaw** (*float*) – Yaw

Returns Quaternion

4.8 Trajectory Planner

Package used to find a collision less trajectory between a number of agents.

4.8.1 Algorithm

The algorithm used is the one presented in this paper: [CIT3]

4.8.2 Exemple

```
A1 = Agent([0.0, 2.0, 0.0])
A1.set_goal([4.0, 2.0, 0.0])

A2 = Agent([4.0, 1.9999, 0.0], goal=[0.0, 1.9999, 0.0])

solver = TrajectorySolver([A1, A2])
solver.set_obstacle(obstacles)

solver.set_wait_for_input(False)
solver.set_slow_rate(1.0)

solver.solve_trajectories()
```

4.8.3 Run Demo Scripts

It's possible to test the trajectory algorithm by running demo scripts (`.../trajectory_planner/demos`.)

To execute a demo, simply uncomment the desired demonstration in `demos.py` -> `demo` and run the script (`demo.py`).

It's possible to add new demonstrations by adding them in either: `demos_formation.py`, `demos_positions.py` or `demos_random.py`.

4.8.4 Benchmark Algo Performances

The algorithm benchmarks can be calculated using `.../trajectory_planner/demos/performance.py`. The script will output:

- The success rate (%)
- Total compute time (sec)
- Total travel time (sec)

Those statistics are always calculated by running the same 9 demos.

Todo: Script to compare and analyse results of different configurations (issue #86). With a graphic interface?

4.8.5 Profile algorithm

To find hotspots in the algorithm:

```
$ cd .../trajectory_planner/demos
$ python -m cProfile -o perfo.prof performance.py
$ snakeviz perfo.prof
```

Note: Snakeviz is required to visualize results: `pip install snakeviz`

4.8.6 Solver Class

class trajectory_solver.**TrajectorySolver**(*agent_list=None*, *solver_args=None*, *verbose=True*)

To solve trajectories of all agents

Parameters

- **agent_list** (*list of Agents*) – Compute trajectory of all agents in the list
- **verbose** (bool) – If True, prints information

Variables

- **horizon_time** (float) – Horizon to predict trajectory [sec]
- **step_interval** (float) – Time steps interval (h) [sec]
- **steps_in_horizon** (int) – Number of time steps in horizon (k)
- **interp_time_step** (float) – Interpolation time step [sec]
- **k_t** (int) – Current time step
- **k_max** (int) – Max time step
- **at_goal** (bool) – True when all agents reached their goal
- **in_collision** (bool) – True when a collision is detected between two agents
- **agents** (list of Agent) – List of all agents
- **n_agents** (int) – Number of agents
- **all_agents_traj** (3k x n_agents array) – Latest predicted trajectory of each agent over horizon
- **agents_distances** (list of float) –
- **kapa** (int) – For more aggressive trajectories
- **error_weight** (float) – Error weigth
- **effort_weight** (float) – Effort weigth
- **input_weight** (float) – Input variation weigth
- **relaxation_weight_p** (float) – Quadratic relaxation weigth
- **relaxation_weight_q** (float) – Linear relaxation weigth
- **relaxation_max_bound** (float) – Maximum relaxation, 0
- **relaxation_min_bound** (float) – Minimum relaxation

- **r_min** (float) – Mimimum distance between agents [m]
- **a_max** (float) – Maximum acceleration of an agent [m/s^2]
- **a_min** (float) – Minimum acceleration of an agent [m/s^2]
- **has_fix_obstacle** (bool) – True if there are fix obstacles to avoid
- **obstacle_positions** (list of tuple) – Coordinates of obstacles to avoid (x, y, z)

set_obstacles(*obstacle_positions*)
Add obstacle has an agent with a cst position

Parameters **obstacle_positions** (list of tuple) – Position of each obstacle (x, y, z)

initialize_agents()
Initialize positions and starting trajectory of all agents

initialize_matrices()
Compute matrices used to determine new states

$$A = \begin{bmatrix} I_3 & hI_3 \\ 0_3 & I_3 \end{bmatrix}$$

$$B = \begin{bmatrix} \frac{h^2}{2} I_3 \\ hI_3 \end{bmatrix}$$

$$\Lambda = \begin{bmatrix} B & 0_3 & ... & 0_3 \\ AB & B & ... & 0_3 \\ ... & ... & ... & ... \\ A^{k-1}B & A^{k-2}B & ... & B \end{bmatrix}$$

$$A_0 = \begin{bmatrix} A^T & (A^2)^T & ... & (A^k)^T \end{bmatrix}^T$$

update_agents_info()
To update agents information

Has to be called when starting position or goal of an agent changed

solve_trajectories()
Compute a collision free trajectory for each agent.

Core of the algorithm.

Returns If the algorithm was succesfull float: Total trajectory time

Return type bool

print_final_positions()
Print final position of all agents

plot_trajectories()
Plot all computed trajectories

4.8.7 Agent Class

class agent.**Agent**(*agent_args*, *start_pos=None*, *goal=None*)
Represents a single agent

agent_idx = None
Index of agent in positions

n_steps = None
Number of steps in horizon

Type int

final_traj = None
Agent final trajectory

close_agents = None
float): Distance of each agent within a certain radius

Type (dict of int

collision_step = None
Step of prediction where collision happens

set_starting_position(*position*)
Set starting position

Parameters **position** (*list of float*) – [x, y, z]

set_goal(*goal*)
Set agent goal

Parameters **goal** (*list of float*) – [x, y, z]

set_all_traj(*all_trajectories*)
Set last predicted trajectories of all agents

Parameters **all_trajectories** (*6*k x n_agents array*) –

add_state(*new_state*)
Add new state to list of positions

Parameters **new_state** (*array*) – Trajectory at time step

initialize_position(*n_steps*, *all_agents_traj*)
Initialize position of the agent.

Sets first horizon as a straight line to goal at a cst speed

Parameters

- **n_steps** (*int*) – Number of time steps of horizon
- **all_agents_traj** (*3k x n_agents array*) – Last predicted traj of each agent (ptr)

check_goal()
Check if agent reached it's goal.

Goal is considered reach when the agent is in a radius smaller than `goal_dist_thres` at a speed lower than `goal_speed_thres`.

Returns True if goal reached

Return type bool

check_collisions()
Check current predicted trajectory for collisions.

1 - For all predicted trajectory, check distance of all the other agents

2 - If distance < Rmin: In collision 3 - If collision: Find all close agents

Returns Step of collision (-1 if no collision) (dict of float): Close agents and their distance at collision step

Return type (int)

interpolate_traj(*time_step_initial*, *time_step_interp*)
Interpolate agent's trajectory using a Bezier curbe.

Parameters

- **time_step_initial** (*float*) – Period between samples
- **time_step_interp** (*float*) – Period between interpolation samples

4.8.8 Trajectory Plotter

Module to plot the trajectories of agents.

Circles represent the agents, dashed line the predicted trajectory over the horizon

class trajectory_plotting.**TrajPlot**(*agent_list*, *time_step*, *interp_time_step*, *wait_for_input=False*, *plot_dots=False*)
To plot trajectories of agents

slow_rate = None
To slow animation

Type int

set_wait_for_input(*to_wait*)
To wait or not for input before switching frame

Parameters **to_wait** (*bool*) – To wait

set_slow_rate(*slow_rate*)
Set slow rate of animation.

Rate of 1 is real time. Rate of 2 is twice slower

Parameters **slow_rate** (*float*) – Rate of slow

set_axes_limits(*xmax*, *ymax*)
Set x and y axes max limits

Parameters

- **xmax** (*float*) –
- **ymax** (*float*) –

set_dot_plotting(*to_plot*)
To plot or not agent's predicted trajectory over horizon as dots

Parameters **to_wait** (*bool*) – To plot dots

update_objects(*agent_list*)
Update agents

Parameters **agent_list** (*list of Agent*) – All agents with their trajectories and goal

init_animated_objects()
Creates all objects to animate.

Each agent has:

- A circle (current position)
- A dashed line (predicted trajectory)
- An X (goal)

Notes

Structure of animated object. Idx: 0: circle of agent 1 1: line of agent 1 2: circle of agent 2 3: line of agent 2 ... -1: time text

init_animation()
Initialize animation

animate(*frame*)
Animate

Parameters **frame** (*int*) – Current frame

run()
Start animation

plot_obstacle(*obstacles*)
Plot obstacle

CHAPTER 5

References

5.1 Glossary

Crazyflie: Bitcraze nan quadcopter

Swarm: Group of all the crazyflies

Formation: Arrangment of agents in certain patern

Agent: Position in a formation

5.2 Ressources

Here, you can find links towards various ressources that helped developing this project.

5.2.1 Coding

- Git tutorial
- Google docstring
- Python Style Guide
- Sphinx and RTD tutorial

5.2.2 Other Research with Crazyflie

- Bitcraze Research page
- Multirobot System - blog
- Modquad - blog

5.2.3 Repositories using Crazyflie

- Bitcraze - Bitcraze official repo
- CrazySwarm - 49 CF controlled with Vicon. Very useful.
- Crazyflie-tools - Used to avoid obstacles (see Landry B. thesis)
- Tunnel Mod - Peer2Peer communication between CF

5.2.4 Crazyflie

- Crazyflie 2.1 - Getting started
  - Controler types
  - Drone Dynamics
  - Position Control Architecture
- Getting started guides
  - First setup
  - LPS System
  - Expansion Decks
  - Development
- Bitcraze Website
  - Wiki
  - Blog
  - Forum
  - Crazyflie-lib-doc

5.2.5 ROS

- Official Tutorials
- ROS Books
  - ROS Robotics By Example
  - Robot Operating System (ROS) - The Complete Reference (Volume 2)

5.2.6 Others

- MRALS lab documentation

5.3 Bibliography

- *Glossary*
- *Ressources*

- *Bibliography*
- genindex
- modindex
- search

Bibliography

[CIT1] Hönig, W., & Ayanian, N. (2017). Flying multiple UAVs using ROS. In Robot Operating System (ROS) (pp. 83-118). Springer, Cham.

[CIT2] Preiss, J. A., Honig, W., Sukhatme, G. S., & Ayanian, N. (2017, May). Crazyswarm: A large nano-quadcopter swarm. In 2017 IEEE International Conference on Robotics and Automation (ICRA) (pp. 3299-3304). IEEE.

[CIT3] C. E. Luis and A. P. Schoellig, "Trajectory Generation for Multiagent Point-To-Point Transitions via Distributed Model Predictive Control," in IEEE Robotics and Automation Letters, vol. 4, no. 2, pp. 375-382, April 2019, doi: 10.1109/LRA.2018.2890572.

[CIT4] C. E. Luis, M. Vukosavljev and A. P. Schoellig, "Online Trajectory Generation With Distributed Model Predictive Control for Multi-Robot Motion Planning," in IEEE Robotics and Automation Letters, vol. 5, no. 2, pp. 604-611, April 2020, doi: 10.1109/LRA.2020.2964159.

[CIT5] R. R. Negenborn and J. M. Maestre, "Distributed Model Predictive Control: An Overview and Roadmap of Future Research Opportunities," in IEEE Control Systems Magazine, vol. 34, no. 4, pp. 87-97, Aug. 2014, doi: 10.1109/MCS.2014.2320397.

[CIT6] J. A. Preiss, W. Hönig, N. Ayanian and G. S. Sukhatme, "Downwash-aware trajectory planning for large quadrotor teams," 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, 2017, pp. 250-257, doi: 10.1109/IROS.2017.8202165.

[CIT7] Y. Chen, M. Cutler and J. P. How, "Decoupled multiagent path planning via incremental sequential convex programming," 2015 IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, 2015, pp. 5954-5961, doi: 10.1109/ICRA.2015.7140034.

Python Module Index

Index

U

Y