UPPSALA
UNIVERSITET

MAT-VET-F 20022
Examensarbete 15 hp
Juni 2020

Autonomous flight of the micro drone Crazyflie 2.1 through an obstacle course

Chiedza Chadehumbe
Josefine Sjöberg

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Autonomous flight of the micro drone Crazyflie 2.1 through an obstacle course

Chiedza Chadehumbe, Josefine Sjöberg

A drone is an unmanned aerial vehicle with multiple forms of usage. Drones can be programmed to fly with different degrees of autonomous flight. Autonomous controlled flight makes it possible for the drone to fly without human involvement and it is then controlled solely by software. The goal of this project is to program the micro drone Crazyflie 2.1 to autonomously fly through an obstacle course in the shortest amount of time and a predetermined direction. The nature and placement of the obstacles are unknown beforehand. The obstacles are detected and avoided by using the obstacle detection sensor Multi-ranger. To achieve autonomous flight two possible navigation systems were tested, the Loco Positioning System and Flow deck. Flying the Crazyflie while using Flow deck as positioning system performed best, managing to fly through the obstacle course avoiding all obstacles.

Handledare: Luca Mottola, Stefan Johansson
Ämnesgranskare: Ken Welch
Examinator: Martin Sjödin
ISSN: 1401-5757, MAT-VET-F 20022

Populärvetenskaplig sammanfattning

En drönare är en obemannad luftfarkost med många användningsområden. Drönare kan programmeras till att flyga med olika grader av autonom styrning. Autonom styrning gör det möjligt för drönaren att styras utan mänsklig inblandning och den styrs då helt av mjukvara. I detta projekt programmeras mikrodrönaren Crazyflie 2.1 till att autonomt flyga snabbast möjligt igenom en förutbestämd hinderbana med slumpmässigt utplacerade hinder. För att upptäcka hinder användes den avståndsmätande sensorn Multi-ranger. I projektet utvärderades två olika positoneringssystem, Loco positioning system och Flow deck. Av de två testade positioneringsystemen presterade Flow deck bäst och med lösningen med Flow deck lyckades drönaren flyga igenom hela hinderbanan och undvika de utplacerade hindrena.

Contents

1 Introduction **3**
- 1.1 Background . . . 3
- 1.2 Goal . . . 3

2 Hardware **3**
- 2.1 Crazyflie 2.1 . . . 3
- 2.2 Crazyflie positioning systems . . . 4
  - 2.2.1 Loco positioning system and the Loco Positioning deck . . . 5
  - 2.2.2 Optical navigation using Flow deck . . . 5
- 2.3 Crazyflie obstacle detection sensor . . . 5
  - 2.3.1 Multi-ranger deck . . . 5

3 Method **6**
- 3.1 Implementing pathfinding . . . 6
- 3.2 Implementing obstacle detection . . . 7
- 3.3 Implementing autonomous flight . . . 7
  - 3.3.1 LPS . . . 7
  - 3.3.2 Flow deck . . . 8

4 Results **8**
- 4.1 Flying through the test path using LPS . . . 8
- 4.2 Flying through the test path using Flow deck . . . 8
  - 4.2.1 Method 1 . . . 8
  - 4.2.2 Method 2 . . . 9
- 4.3 Flying through the obstacle course . . . 10

5 Discussion **12**
- 5.1 Flying through the test path using LPS . . . 12
- 5.2 Flying through the test path using Flow deck . . . 12

6 Conclusion **13**

7 Bibliography **14**

Appendices **16**

A Available equipment **16**

B Flying environment **16**

C Methods of flying **17**
- C.1 Method 1: Angling the drone . . . 17
- C.2 Method 2: Not angling the drone . . . 17

D Code **18**
D.1 LPS 18
D.2 Flow Deck 24
D.2.1 Method 1 24
D.2.2 Method 2 30

1 Introduction

1.1 Background

A drone is an unmanned aerial vehicle that has multiple forms of usage. Some of these applications are aerial photography, combat, and transport. Smaller types of drones where the wingspan is less than 15cm are called micro drones [1]. Drones can be programmed to fly with different degrees of autonomous flight. Autonomous controlled flight makes it possible for the drone to fly without human involvement and it is then controlled solely by software. The drone can then by itself detect and manage applications that are hard or dangerous for humans to do. To be able to fly autonomously the drone needs sensors to detect potential obstacles.

1.2 Goal

The goal of this project is to assemble the micro drone Crazyflie 2.1, manufactured by Bitcraze AB [2], and program it to autonomously fly through a predefined circular obstacle course in the shortest amount of time with a predetermined direction. The path is not unique as there are multiple available routes to reach the finish line. Time starts when the drone takes off at the starting point and the run concludes when the drone lands at the same take-off point, with some tolerance margin. The path the drone has taken will be logged during the run for performance evaluation. The course is given as a sequence of 3D coordinates. Where the path splits in multiple directions, an obstacle may be placed preventing the drone to fly through the shortest path. The drone must then choose the second shortest path given that no obstacles are blocking that direction, and so on. The number of obstacles as well as their nature and positioning is not known beforehand, meaning, the drone must dynamically recognize the situation and act accordingly.

To reach this goal a catalog of sensors and equipment was given to us, where the full list is specified in Appendix A. There were two different sensors that can act as the drones positioning system, the Loco Positioning System (LPS) and Flow deck and one sensor for obstacle detection, Multi-ranger. One of the aims of the project was to evaluate which one of the given positioning system sensors that works best for our goal.

This project will be the foundation of a possible future project-based course for students on the Master's Programme in Engineering Physics on Uppsala University to enroll, to learn more about autonomously controlled drones.

2 Hardware

2.1 Crazyflie 2.1

The Crazyflie 2.1, see Figure 1, is the third miniature quadcopter developed by Bitcraze AB. The Crazyflie comes as a kit and weighs 27g and measures 92x92mm when assembled. It has four 7mm coreless DC-motors that achieve a maximum takeoff weight of 42g. This enables the Crazyflie 2.1 to carry more hardware in the form of expansion decks that can provide further functionality, such as sensors. An expansion deck can be placed either on top or under the Crazyflie. A fully loaded battery gives the quadcopter around seven-minutes of flight time.

Figure 1: To the left the Crazyflie 2.1 package contents unassembled and to the right the Crazyflie 2.1 assembled, [3].

Crazyflie 2.1 is equipped with a low-latency/long-range radio as well as Bluetooth LE which enables both manual and autonomous flight. Manual flight is possible either from a mobile app using Bluetooth, available for both Android and iOS, or through the Crazyflie client installed on the Bitcraze virtual machine (VM) using a game controller. The latter requires the Crazyflie PA, a 2.4 GHz long-range USB radio dongle that sends communications between the computer and the Crazyflie. Autonomous flight is possible using software to control the Crazyflie, instead of manual control, in combination with the Crazyflie PA and some kind of positioning system giving the drone information on its position. When flying the Crazyflie there are four main dimensions of control, visualized in Figure 2. The dimensions are roll, pitch, yaw, and thrust.

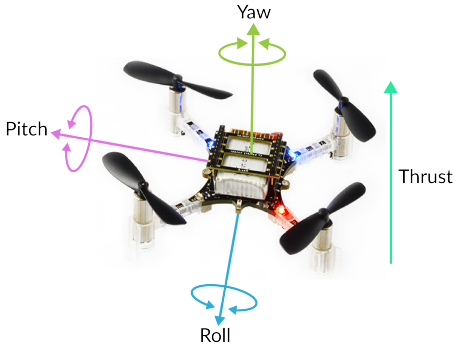


Figure 2: The main dimensions to control while flying the Crazyflie 2.1, [4]

2.2 Crazyflie positioning systems

Autonomous flight with the Crazyflie requires an expansion deck that can get its current position or measure relative distance so that it can move to a desired position in a desired velocity. In this project, two possible positioning systems were available for testing. The Loco Positioning System with the Loco Positioning deck or optical navigation using the Flow deck.

2.2.1 Loco positioning system and the Loco Positioning deck

The Loco Positioning system [5] is a local positioning system that can be used to give the Crazyflie absolute 3D coordinates by using a set of anchors, Loco Positioning nodes, that are positioned in the room as reference. On the drone, a Loco Positioning deck, see Figure 3, is placed either on top or underneath the Crazyflie. By sending radio messages between the anchors and the Loco Positioning deck, the system measures the distance from each anchor to the deck and calculates the 3D position of the Crazyflie.

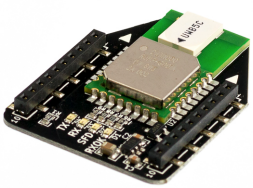


Figure 3: Loco Positioning deck, [6]

2.2.2 Optical navigation using Flow deck

The Flow deck v2 [7], seen in Figure 4, gives the Crazyflie the ability to sense its motion in both vertical and horizontal directions. It uses a time-of-flight laser ranging sensor to measure vertical distances up to 4m with mm precision and an optical flow sensor to detect and measure the horizontal motion of surfaces, enabling the drone to travel desired distances. The deck weighs 1.6g, measures 21x28x4mm, and is designed for mounting under the Crazyflie.

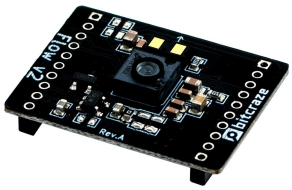


Figure 4: Flow deck v2, [7].

2.3 Crazyflie obstacle detection sensor

2.3.1 Multi-ranger deck

The Multi-ranger deck [8], seen in Figure 5, gives the Crazyflie the ability to sense objects around it in five directions: front, back, left, right, and up. It uses five time-of-flight laser ranging sensors to sense objects up to 4m away with mm precision and allowed proximity can be set by the user through the Application Programming Interface (API) for obstacle avoidance. The deck weighs 2.3g, measures 35x35x5mm, and is designed for mounting on top of the Crazyflie.

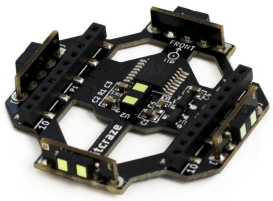


Figure 5: Multi-ranger deck, [8].

3 Method

To reach the project goal three main steps needed to be implemented: pathfinding, obstacle detection, and autonomous flight. This was implemented using a virtual machine (VM) that Bitcraze has developed [9], which has all the software needed for flight and development pre-installed. Virtual machines are software computers that provide the same functionality as physical computers; they run applications and an operating system [10]. The application logic could be placed either on a controlling station, such as a computer, that connects to the Crazyflie or directly on the Crazyflie. The Crazyflie firmware is written in C++ and the official Crazyflie API is written in Python, however, several other languages can be used from the client side. For this project, it was decided that a controlling station, using the Python API, should control the Crazyflie.

For test flying, we had access to a room at Uppsala University which had the LPS using eight anchors installed, see Appendix B. All tests were made in this room for both LPS and Flow deck, including the final test.

To be able to check how well the drone performs, the current position was logged in a CSV-file using the pre-existing Kalman filter estimation with an interval of 500 ms.

3.1 Implementing pathfinding

The path is given as a sequence of 3D-coordinates in a CSV-file, where each coordinate is followed by its connected coordinates. A graph data structure was chosen to represent the path and was implemented using the Python package NetworkX [11]. NetworkX contains a large number of tools to build and manipulate graph structures and is well documented for easy use. A directed graph was chosen since the drone is only allowed to travel a specific direction on the path. The path coordinates were represented by graph nodes and the path between them by graph edges. NetworkX provides pathfinding features for finding the shortest path between specific nodes. We added weights to each edge equaling the distance between the nodes and used their standard feature for weighted pathfinding to find the shortest path through the obstacle course. Weighted pathfinding returns the path with the smallest sum of the weights, ensuring that the shortest path is returned. Since the obstacle course starts and ends on the same node, the start node was changed by 0.0001 m in the z-direction so that two different nodes could be used as input in the NetworkX feature. This was needed since if the same node is used for start and finish, it only returns itself.

3.2 Implementing obstacle detection

For obstacle detection, we used the Multi-ranger deck with the corresponding class *multiranger* developed by Bitcraze [12]. As stated in the problem description, obstacles will only be placed in the vicinity of a splitting point. For testing our implemented solution we invented a test path, see Figure 6. Looking at the test path, the drone needs to check for obstacles when it stands on a splitting point, as in node B, and also before it reaches a splitting point looking for dead ends, as for when it travels from node G to F. If the drone detects an obstacle between two nodes, the graph edge between them is removed and the shortest path is recalculated using this new information. If the drone detects a dead end, it also re-tracks its path before it recalculates the new path, since the graph is directed. To achieve this, the drone was instructed to stop before it reaches its destination, checking for obstacles in its flight direction, and then fly the rest of the distance if the path is clear. When it reaches a node, it checks for obstacles in the next flight direction, before it starts to travel again.

When we evaluated our implemented solutions on the test path we used ourselves as stationary obstacles, making sure to stand at the same spot for each test run when the Crazyflie searched for obstacles in its path.

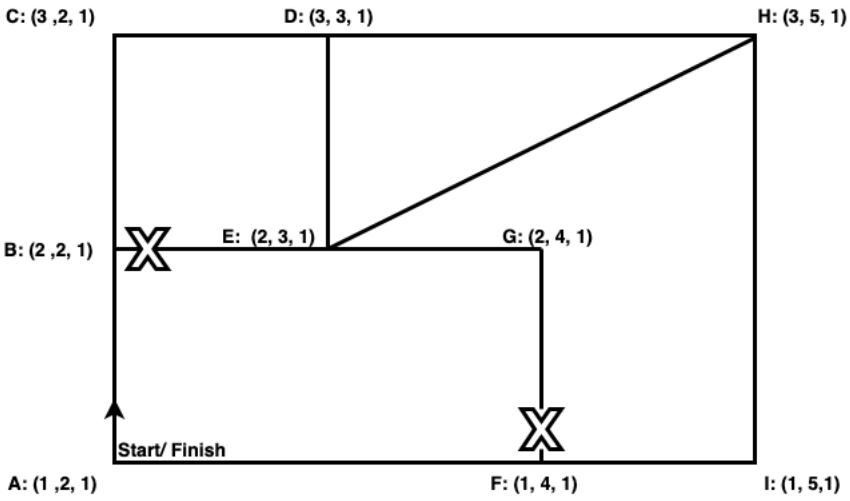


Figure 6: Test path.

3.3 Implementing autonomous flight

3.3.1 LPS

When implementing autonomous flight using the LPS, we based the code of the Bitcraze example script *AutonomousSequence*, found on Github [13]. The script uses the *commander* class [14] developed by Bitcraze, to fly, where position coordinates and angle are passed to the drone in sequence. Angle calculations were implemented so that the drone always turned to the flight

direction, enabling the use of only the front sensor for obstacle detection on the Multi-ranger, see Appendix C.1 for an illustration.

3.3.2 Flow deck

Two different ways of flying using the Flow deck were implemented and tested, where both methods used Bitcrazes *motion_commander* class [15] developed for Flow deck. Since the Flow deck does not know its absolute position we fly it using the *motion_commander* function *move_distance* to move a certain distance in the x-, y- and, z-direction. First, it was implemented that the drone should fly in the direction it is facing, meaning it should angle itself to the node it flies to, identical to the behavior when using LPS. From here on, this method is referred to as Method 1.

In the second method, it was implemented that the drone never turns, using the calculated relative angles between nodes and drone orientation to determine which Multi-ranger sensor to use, see Appendix C.2 for an illustration. From here on, this method is referred to as Method 2.

4 Results

4.1 Flying through the test path using LPS

In Table 1, 15 test runs using three different velocities are illustrated. All runs were incomplete due to overshooting the destination coordinate F, see Figure 6, resulting in the drone flying into the obstacle. All tests were made using the LPS with 8 anchors.

Table 1: Result from 15 test runs for three different velocities.

Run	0.2m/s	0.4m/s	0.6m/s
1	Incomplete	Incomplete	Incomplete
2	Incomplete	Incomplete	Incomplete
3	Incomplete	Incomplete	Incomplete
4	Incomplete	Incomplete	Incomplete
5	Incomplete	Incomplete	Incomplete

4.2 Flying through the test path using Flow deck

4.2.1 Method 1

When using Method 1 to fly through the obstacle course, it was observed that the angling of the drone resulted in undesirable behavior. This behavior manifested itself as over- or undershooting of the angle or drifting, as the drone turned itself. To examine the effect of the velocity when the drone turned, tests were made of the importance of the angular velocity. In Table 2, the result of 15 different test runs using three different angular velocities when angling the Crazyflie is presented. The reason for the incomplete runs is color-coded by their main reason for failing the test run. Red means that the drone did not angle itself correctly after adjusting its angle in a turn or due to an obstacle, either by over- or undershooting the angle or drifting while turning. Blue is when

the drone did not travel the correct distance between one of the previous coordinates and therefore missed an obstacle. The latter is independent of the angular velocity but was instead a problem due to using Flow deck as the positioning system. All the test runs resulted in the drone missing the second obstacle placed before the coordinate F except the fifth test run with 90°/s as the angular velocity where it missed the first obstacle placed closed to the coordinate B. All tests were run with the velocity of 0.2m/s.

Table 2: Result from 15 test runs for three different angular velocities.

Run	45°/s	90°/s	180°/s
1	Incomplete	Incomplete	Incomplete
2	Incomplete	Incomplete	Incomplete
3	Incomplete	Incomplete	Incomplete
4	Incomplete	Incomplete	Incomplete
5	Incomplete	Incomplete	Incomplete

Since the result of the tests of adjusting the angular velocity resulted in 15 incomplete runs, it was decided to not do any further fine-tuning of this way of flying the Crazyflie.

4.2.2 Method 2

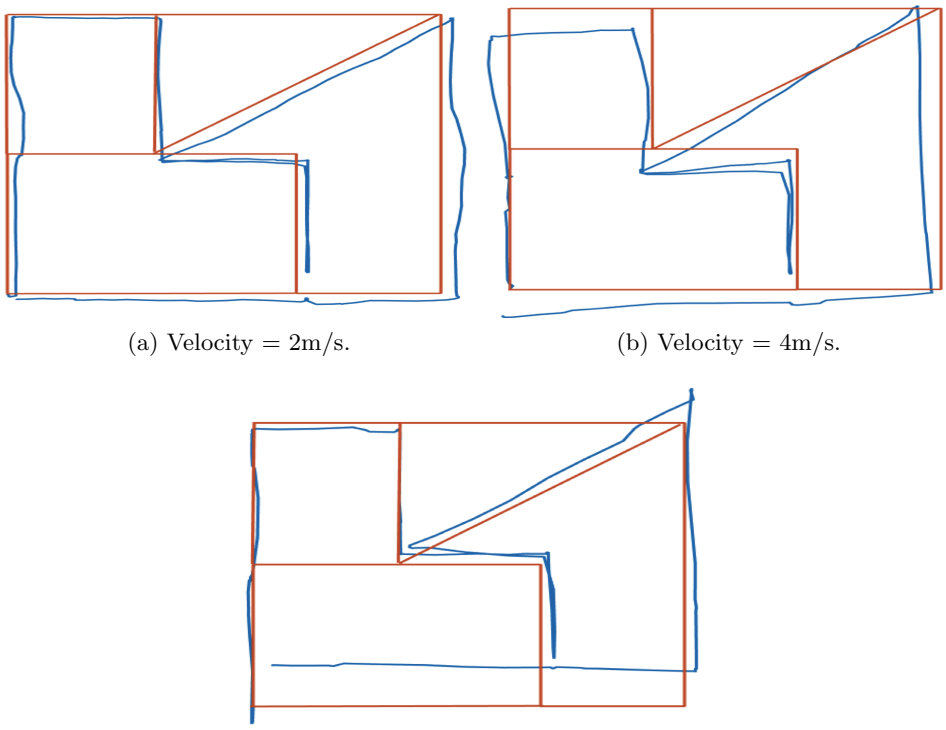
When flying the obstacle course using the sensor in the direction of travel, the angling was no longer a problem. It was instead investigated how the speed of the drone affected the drone's ability to manage the path. The result of 15 test runs with three different velocities is shown in Table 3. All the incomplete test runs were due to the Crazyflie flying inconsistent distances between the coordinates resulting in it missing the obstacle placed in front of coordinate F. The average run time for the completed runs were 1 min and 55 s with a velocity equal to 0.2m/s, 1 min and 17 s with 0.4m/s and 1 min and 6 s with 0.6m/s.

Table 3: Result from 15 test runs for three different velocities.

Run	0.2m/s	0.4m/s	0.6m/s
1	Complete	Incomplete	Incomplete
2	Incomplete	Incomplete	Incomplete
3	Incomplete	Complete	Complete
4	Incomplete	Incomplete	Complete
5	Complete	Complete	Incomplete

In Figure 7, the average path is shown for the three velocities two completed runs. The best performance was achieved with 0.2m/s as velocity.

To see if it was possible to fine-tune the solution, a correction of the position of the drone when it reached every coordinate was tested. Since the position was logged relative to the start position of the drone, the current position could be corrected using the logged value. The result of 15 test runs with three different velocities is shown in Table 4. The first aim was to evaluate the same velocities



(a) Velocity = 2m/s.

(b) Velocity = 4m/s.

(c) Velocity = 6m/s.

Figure 7: The best path the drone took of each velocity's completed runs.

as in Table 3, but since it was observed that the result for 0.4m/s was all incomplete runs, 0.3m/s was evaluated instead of 0.6m/s.

Table 4: Result from 15 test runs for three different velocities.

Run	0.2m/s	0.3m/s	0.4m/s
1	Complete	Complete	Incomplete
2	Complete	Incomplete	Incomplete
3	Complete	Incomplete	Incomplete
4	Complete	Complete	Incomplete
5	Complete	Incomplete	Incomplete

4.3 Flying through the obstacle course

The path given to us for the final test can be seen in Figure 8. The placement and nature of obstacles were unknown before the final test but were at the time of testing revealed to be chairs with the placement shown in Figure 8.

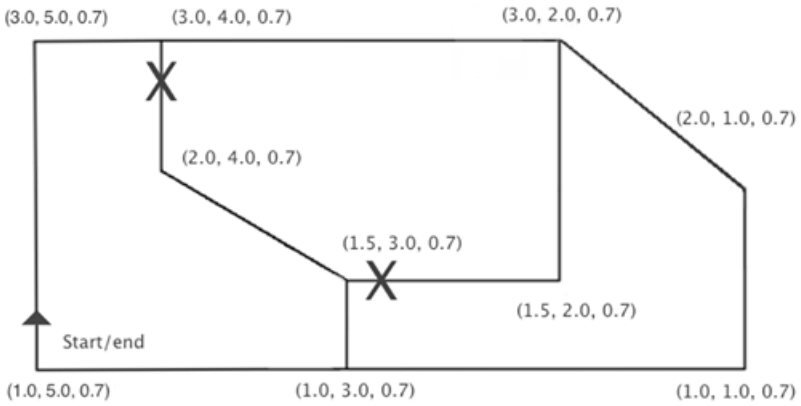


Figure 8: The final path.

During test flies through the given obstacle course without obstacles, it was observed that although the correction of the coordinates did work for when the drone moved small distances, it did not work sufficient enough for longer distances. When flying distances that were above 1m the drone often failed to fly all the way. So instead we tried to elongate the distance the drone should travel when the distance between two coordinates was longer than 1m. This was done by multiplying the distance by 1.1.

The final result was achieved using Method 2 without correction of the drone at the coordinates but with multiplying the longer distances with 1.1. This resulted in the logged position of the drone appearing to be off the path, but in actuality it was not the case, see Figure 9. The speed was set to 0.2m/s and the time it took for the drone to fly through the obstacle course was 2 minutes and 4s.

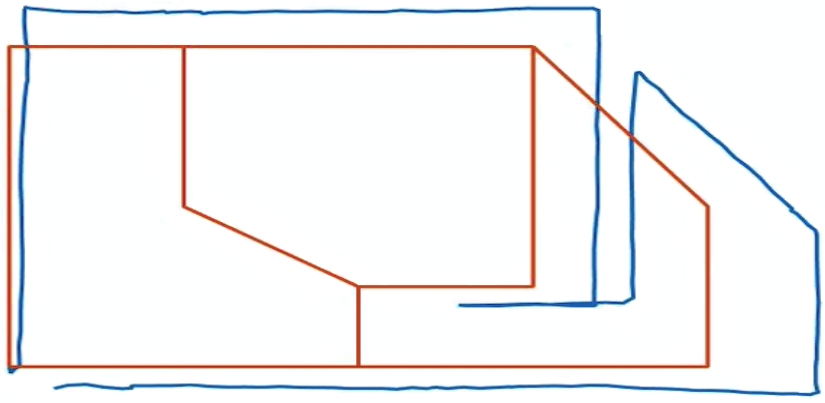


Figure 9: The result of the drone flying through the final path. In the picture it seems that the drone was off path but in real life the drone managed to fly through the obstacle course.

5 Discussion

5.1 Flying through the test path using LPS

The LPS uses absolute position and angle, where the angle of the drone is relative to the x-axis of the LPS coordinate system. This means that the drone can take-off from any position in the room and then fly to the starting point of the obstacle course with a desired angle. An unstable take-off is therefore not an issue. However, the absolute position is sensitive to disturbances and was tested to have an error margin of as much as ± 40cm in all directions in our testing environment. This caused the drone to hover and fly unstable, causing the drone to miss obstacles and sometimes crash. The LPS is documented to have an accuracy of approximately 10cm [5], leading us to think that there were unusually large disturbances from the surroundings of our testing area.

The unstable flight was minimized changing the number of calls made for each position when calling the function *send_position_setpoint()* from the *commander* class. Optimal velocity was found to be 0.2m/s with the number of calls set to 20.

Despite the unstable flight, the logged position did not show a large error margin leading us to think that the Kalman filter installed on the drone fluctuates with the absolute position of the LPS.

5.2 Flying through the test path using Flow deck

The Flow deck uses relative position when flying, meaning it is controlled with reference to its current state. This poses some challenges since it does not have an absolute position or angle. This means that the location of the start node needs to be known beforehand and also the first travel direction. This is where one of the biggest issues with the Flow deck comes in. The take-off procedure using the Flow deck can be unstable, where the drone drifts away from the starting point or turns and get an incorrect first direction. However, since the take-off was not part of this project, this was solved by simply re-trying the take-off continuously until a successful take-off was achieved. Also, since the optical flow sensor placed on the Flow deck is sensitive to different surfaces, colors, and lighting, some care was needed when it comes to where it will fly. If the sun hits a spot on the ground or if it flies over a carpet, the sensor might be confused and drift off its current path or crash. The Flow deck works best on matt, light surfaces in good lighting.

There is also the problem of unbalanced propellers caused by previous crashes, production errors, or a non-centered placement of the battery. To minimize the effect of these issues, one can investigate how unstable the flight is by test flying the drone beforehand. Unstable flight is characterized by the drone drifting when hovering, flying, and/or when it turns. If such behavior is present, the separate parts of the drone need to be examined and replaced if a defect is found.

Using Method 1, where the drone angles itself to the flight direction, resulted in path breaches causing the drone to miss obstacles by flying past them or by not reaching them. These breaches were mainly caused by drifting during turns. We tried to minimize this drifting by fine-tuning the angular velocity and found that for 90°/s the most stable turns were achieved. For 45°/s the drone drifted more during turns and for 180°/s the angle overshot more often. However, fine-tuning the angle velocity did not result in more completed runs since the distance estimation was incorrect.

To avoid the problem of unstable turns altogether, Method 2 was implemented where no turns are required. This resulted in a much more stable flight. However, it was noticed that the drone rarely flies the exact distance it is instructed to fly when using the Flow deck. This was fine-tuned by multiplying the longer flying distances by 1.1 to achieve a more accurate representation of the real distance. With this solution, the drone managed to fly through the final obstacle course seen in Figure 8.

6 Conclusion

Of the two tested ways of navigating the Crazyflie through the obstacle course, either by flying using LPS or using Flow deck, the Flow deck performed the best. No test runs were complete using the LPS as the positioning system, with the main reason being unstable flight caused by disturbances in the LPS.

When using the optical navigation system, Flow deck, two different methods were implemented and tested, one of which angled the Crazyflie so that only the forward sensor needed to be used on the obstacle detection sensor Multi-ranger, and one where the drone never changes its angle but uses the calculated angles between nodes to determine which sensor on the Multi-ranger to use. The second method, Method 2, with some fine-tuning performed the best and managed to avoid all obstacles during the final test and reach the finish line.

7 Bibliography

[1] Nonami, K. *Autonomous Flying Robot: Unmanned Aerial Vehicles and Micro Aerial Vehicles*, New York; Tokyo, Springer, 2010.

[2] Bitcraze webpage, *Bitcraze*.
Available at:https://www.bitcraze.io/
(Accessed: 10 June 2020).

[3] Crazyflie 2.1, *Bitcraze*.
Available at:https://store.bitcraze.io/products/crazyflie-2-1
(Accessed: 10 June 2020).

[4] Getting started with the Crazyflie 2.X, *Bitcraze*.
Available at:https://www.bitcraze.io/documentation/tutorials/getting-started-with-crazyflie-2-x/
(Accessed: 10 June 2020).

[5] Loco Positioning system, *Bitcraze*.
Available at: https://www.bitcraze.io/products/loco-positioning-system/
(Accessed: 10 June 2020).

[6] Loco Positioning Deck, *Bitcraze*.
Available at: https://www.bitcraze.io/products/loco-positioning-deck/
(Accessed: 10 June 2020).

[7] Flow deck v2, *Bitcraze*.
Available at: https://www.bitcraze.io/products/flow-deck-v2/
(Accessed: 10 June 2020).

[8] Multi-ranger deck, *Bitcraze*.
Available at: https://www.bitcraze.io/products/multi-ranger-deck/
(Accessed: 10 June 2020).

[9] The bitcraze virtual machine, *Github*.
Available at: https://github.com/bitcraze/bitcraze-vm/releases/
(Accessed: 10 June 2020).

[10] Virtual machine description, *vmware*.
Available at: https://www.vmware.com/topics/glossary/content/virtual-machine
(Accessed: 10 June 2020).

[11] NetworkX webpage, *NetworkX*.
Available at: https://networkx.github.io/
(Accessed: 10 June 2020).

[12] Multiranger class, *Github*.
Available at: https://github.com/bitcraze/crazyflie-lib-python/blob/master/cflib/utils/multiranger.py
(Accessed: 10 June 2020).

[13] AutonomousSequence.py example script, *github*.
Available at: https://github.com/bitcraze/crazyflie-lib-python/blob/master/examples/autonomousSequence.py
(Accessed: 10 June 2020).

[14] Commander class, *github*.
https://github.com/bitcraze/crazyflie-lib-python/blob/master/cflib/crazyflie/commander.py
(Accessed: 10 June 2020).

[15] Motion commander class, *github*.
https://github.com/bitcraze/crazyflie-lib-python/blob/master/cflib/positioning/motion_commander.py
(Accessed: 10 June 2020).

Appendices

A Available equipment

Equipment	Description
Loco Positioning System	8 LPS nodes, 10 Loco Positioning decks
CrazyFlie 2.1 kit	The micro drone
CrazyRadio PA	Connects a controlling station to the micro drone
Multi-ranger Deck	Ranging sensor for obstacle detection
Flow Deck v2	Optical motion sensor
Prototyping Deck	Prototyping area for placing extra hardware
Breakout Deck	For testing new hardware
Debug adapters	For debugging the drone

Table 5: List over available equipment during this project.

B Flying environment

Figure 10: A photograph from a test flight with the drone.

C Methods of flying

C.1 Method 1: Angling the drone

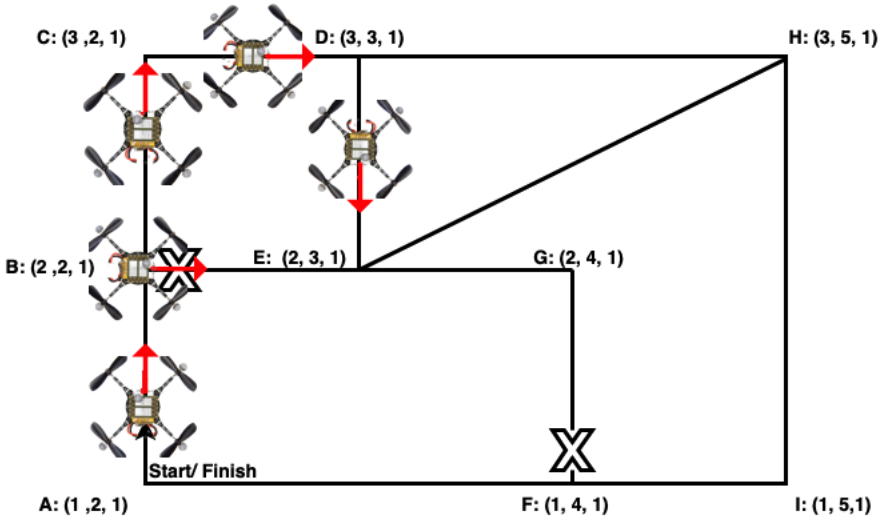


Figure 11: Flying by angling the drone.

C.2 Method 2: Not angling the drone

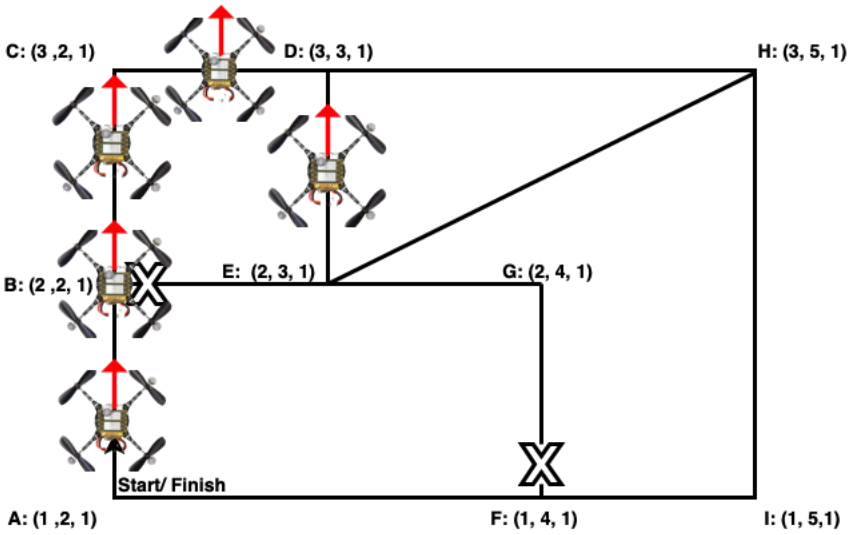


Figure 12: Flying by not angling the drone.

D Code

D.1 LPS

```
# -*- coding: utf-8 -*-
#
#     ||          ____  _ __
#  +------+      / __ )(_) /_______________ _____  ___
#  | 0xBC |     / __  / / __/ ___/ ___/ __ `/_  / / _ \
#  +------+    / /_/ / / /_/ /__/ /  / /_/ / / /_/  __/
#   ||  ||    /_____/_/\__/\___/_/   \__,_/ /___/\___/
#
#  Copyright (C) 2016 Bitcraze AB
#
#  Crazyflie Nano Quadcopter Client
#
#  This program is free software; you can redistribute it and/or
#  modify it under the terms of the GNU General Public License
#  as published by the Free Software Foundation; either version 2
#  of the License, or (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software
#  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
#  MA  02110-1301, USA.

"""
This script flies one crazyflie autonomously through an obstacle course.
The coordinates are read from a file, placed in a directed graph and the
shortest path through the course is calculated and sent to the drone.
The drone can detect obstacles 50 cm away (def in is_close) and can handle
obstacles at splitting points and dead ends.

The script is designed for the Loco Positioning System.
"""

import logging
import time
import csv
import os
import sys
import networkx as nx
import numpy as np
import math

import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.motion_commander import MotionCommander
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncLogger import SyncLogger
from cflib.utils.multiranger import Multiranger
```

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

#Bitcraze
def wait_for_position_estimator(scf):
    print('Waiting for estimator to find position...')

    log_config = LogConfig(name='Kalman Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

    threshold = 0.001

    with SyncLogger(scf, log_config) as logger:
        for log_entry in logger:
            data = log_entry[1]

            var_x_history.append(data['kalman.varPX'])
            var_x_history.pop(0)
            var_y_history.append(data['kalman.varPY'])
            var_y_history.pop(0)
            var_z_history.append(data['kalman.varPZ'])
            var_z_history.pop(0)

            min_x = min(var_x_history)
            max_x = max(var_x_history)
            min_y = min(var_y_history)
            max_y = max(var_y_history)
            min_z = min(var_z_history)
            max_z = max(var_z_history)

            # print("{} {} {}".
            #       format(max_x - min_x, max_y - min_y, max_z - min_z))

            if (max_x - min_x) < threshold and (
                    max_y - min_y) < threshold and (
                    max_z - min_z) < threshold:
                break

#Bitcraze
def reset_estimator(scf):
    cf = scf.cf
    cf.param.set_value('kalman.resetEstimation', '1')
    time.sleep(0.1)
    cf.param.set_value('kalman.resetEstimation', '0')

    wait_for_position_estimator(cf)

#Bitcraze
```

```
def position_callback(timestamp, data, logconf):
    x = data['kalman.stateX']
    y = data['kalman.stateY']
    z = data['kalman.stateZ']
    #print('pos: ({}, {}, {})'.format(x, y, z))
    with open('position.csv', 'a') as csvfile:
        writer = csv.writer(csvfile,delimiter=',')
        writer.writerow([x, y, z])
    csvfile.close()

#Bitcraze
def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=500)
    log_conf.add_variable('kalman.stateX', 'float')
    log_conf.add_variable('kalman.stateY', 'float')
    log_conf.add_variable('kalman.stateZ', 'float')

    scf.cf.log.add_config(log_conf)
    log_conf.data_received_cb.add_callback(position_callback)
    log_conf.start()


def is_close(range):
    MIN_DISTANCE = 0.5  # m

    if range is None:
        return False
    else:
        return range < MIN_DISTANCE


def recalc_path(current_coord, next_coord, end_coord):
    G.remove_edge(current_coord,next_coord)
    last_coord = next_coord
    nodes_shortest = nx.dijkstra_path(G,source=current_coord, target=end_coord)
    next_coord = nodes_shortest[1]
    angles = angles_path(nodes_shortest)
    return next_coord, nodes_shortest, angles


def make_graph():
    G = nx.DiGraph()
    with open('copycoordinates.csv', 'r') as file:
        reader = csv.reader(file)
        line_count = 0
        for row in reader:
            nr_nodes = int(len(row)/3)
            nr_paths = nr_nodes-1
            x = []
            y = []
            z = []
            for i in range(0,nr_nodes):
                x.append(float(row[0+3*i]))
                y.append(float(row[1+3*i]))
                z.append(float(row[2+3*i]))
            for i in range(0,nr_nodes-1):
```

```
                Weight = abs(x[0]-x[i+1]+y[0]-y[i+1]+z[0]-z[i+1])
                G.add_edge((x[0],y[0],z[0]),(x[i+1],y[i+1],z[i+1]), weight = Weight)

    return G


def change_startcoord():
    with open('testpath.csv', 'r') as file:
        reader = csv.reader(file)
        lines = list(reader)
        lines[0][2] = float(lines[0][2])-0.0001
    file.close()
    with open('copycoordinates.csv', 'w') as file:
        writer = csv.writer(file)
        writer.writerows(lines)
    file.close()
    return lines



def angles_path(nodes_shortest):
    angles = []
    for i in range(0, len(nodes_shortest)-1):
        p0 = nodes_shortest[i]    #current
        p1 = nodes_shortest[i+1]  #next
        v1 = np.array([p1[0]-p0[0], p1[1]-p0[1]])  #Direction
        v2 = np.array([1, 0])                      #Reference
        angle = np.math.atan2(np.linalg.det([v2,v1]),np.dot(v2,v1))
        angles.append(np.degrees(angle))
    angles.append(0)
    return angles

def dy(distance,m):
    return m*dx(distance,m)

def dx(distance,m):
    return math.sqrt(distance**2/(m**2+1))

def fly(scf, G, lines, multiranger):
    cf = scf.cf
    nodes_shortest = nx.dijkstra_path(G,source=(float(lines[0][0]),float(lines[0][1]),
        float(lines[0][2])), target=(float(lines[0][0]),float(lines[0][1]),float(lines
        [0][2])+0.0001))
    angles = angles_path(nodes_shortest)
    current_coord = nodes_shortest[0]
    next_coord = nodes_shortest[1]
    end_coord = nodes_shortest[len(nodes_shortest)-1]
    times = 20

    coord_index = 1
    traveled_path = []
    traveled_path.append(current_coord)
    #Fly to the starting point
    for j in range(times):
        cf.commander.send_position_setpoint(current_coord[0],
                                            current_coord[1],
```

```
                                        current_coord[2],
                                        angles[coord_index-1])
        time.sleep(0.1)
    print(current_coord)
    while current_coord != end_coord:

        #Check if there is a obstacle placed at the splitting point
        if is_close(multiranger.front):
            print('Obstacle placed at splitting point')
            next_coord, nodes_shortest, angles = recalc_path(current_coord, next_coord
                , end_coord)
            coord_index = 1
            for j in range(times):
                cf.commander.send_position_setpoint(current_coord[0],
                                        current_coord[1],
                                        current_coord[2],
                                        angles[coord_index-1])
                time.sleep(0.1)

        else:
            #If no obstacle is found, go to next coordinate but stop before and check
                for dead end
            #print('No obstacle found')
            d = 0.4
            distance_x = next_coord[0]-current_coord[0]
            distance_y = next_coord[1]-current_coord[1]
            distance_z = next_coord[2]-current_coord[2]
            if distance_x == 0:
                m = 0
            else:
                m = distance_y/distance_x
            pause_coord = (next_coord[0] - dx(d, m), next_coord[1]-dy(d, m))
            for j in range(times):
                cf.commander.send_position_setpoint(pause_coord[0],
                                                    pause_coord[1],
                                                    current_coord[2],
                                                    angles[coord_index-1])
                time.sleep(0.1)
            print('pause coord')

            #Check for dead end move back to the latest splitting point
            if is_close(multiranger.front):
                print('Dead end found, retrack')
                for j in range(times):
                    cf.commander.send_position_setpoint(current_coord[0],
                                        current_coord[1],
                                        current_coord[2],
                                        angles[coord_index])
                    time.sleep(0.1)
                for i in range(len(traveled_path)):
                    if len(G.adj[(traveled_path[-(i+1)])])>=2:
                        splitting_point = traveled_path[-(i+1)]
                        break
                    else:
                        pass
                #If the last splitting point is the last coordinate go back to it
```

```
            if splitting_point == current_coord:
                next_coord, nodes_shortest, angles = recalc_path(current_coord,
                    next_coord, end_coord)
                coord_index = 1
                for j in range(times):
                    cf.commander.send_position_setpoint(current_coord[0],
                                        current_coord[1],
                                        current_coord[2],
                                        angles[coord_index])
                    time.sleep(0.1)
            else:
                #If the last splitting point is several coordinates away, retrack
                    to it
                print('current_coord')
                print(current_coord)
                retrack = nx.dijkstra_path(G,source=splitting_point, target=
                    current_coord)
                #print(retrack)
                G.remove_edge(current_coord,next_coord)
                for i in range(len(retrack)-1):
                    for j in range(times):
                        cf.commander.send_position_setpoint(retrack[-(i+2)][0],
                                        retrack[-(i+2)][1],
                                        retrack[-(i+2)][2],
                                        0)

                        time.sleep(0.1)
                    current_coord = (retrack[-(i+2)][0], retrack[-(i+2)][1],
                        retrack[-(i+2)][2])

                    traveled_path.pop()
                    print('current_coord')
                    print(current_coord)
                next_coord, nodes_shortest, angles = recalc_path(current_coord,
                    retrack[1], end_coord)
                coord_index = 1
                for j in range(times):
                    cf.commander.send_position_setpoint(current_coord[0],
                                        current_coord[1],
                                        current_coord[2],
                                        angles[coord_index-1])
                    time.sleep(0.1)

        #If no dead end is found, move all the way to the node and turn towards
            next direction
        else:
            print('fly to all the way to next')
            for j in range(times):
                cf.commander.send_position_setpoint(next_coord[0],
                                        next_coord[1],
                                        next_coord[2],
                                        angles[coord_index])
                time.sleep(0.1)
            coord_index += 1
            if coord_index < len(nodes_shortest):
                current_coord = next_coord
```

```
                    traveled_path.append((current_coord))
                    print('current_coord')
                    print(current_coord)
                    next_coord = nodes_shortest[coord_index]
                else:
                    break
    cf.commander.send_stop_setpoint()
    # Make sure that the last packet leaves before the link is closed
    # since the message queue is not flushed before closing

    print(next_coord)
    return end_coord


if __name__ == '__main__':
    try:
        os.remove('position.csv')
    except:
        print('Already deleted.')
    lines = change_startcoord()
    G = make_graph()
    cflib.crtp.init_drivers(enable_debug_driver=False)
    with SyncCrazyflie(URI, cf=Crazyflie(rw_cache='./cache')) as scf:
        reset_estimator(scf)
        start_position_printing(scf)
        with Multiranger(scf) as multiranger:
            fly(scf, G, lines, multiranger)
```

D.2 Flow Deck

D.2.1 Method 1

```
# -*- coding: utf-8 -*-
#
#     ||          ____  _ __
#  +------+      / __ )(_) /_______________ _____  ___
#  | 0xBC |     / __  / / __/ ___/ ___/ __ `/_  / / _ \
#  +------+    / /_/ / / /_/ /__/ /  / /_/ / / /_/  __/
#   ||  ||    /_____/_/\__/\___/_/   \__,_/ /___/\___/
#
#  Copyright (C) 2017 Bitcraze AB
#
#  Crazyflie Nano Quadcopter Client
#
#  This program is free software; you can redistribute it and/or
#  modify it under the terms of the GNU General Public License
#  as published by the Free Software Foundation; either version 2
#  of the License, or (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software
```

```
#  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
#  MA  02110-1301, USA.
"""
This script flies one crazyflie autonomously through an obstacle course.
The coordinates are read from a file, placed in a directed graph and the
shortest path through the course is calculated and sent to the drone.
The drone can detect obstacles 50 cm away (def in is_close) and can handle
obstacles at splitting points and dead ends.

The script is designed for the floe deck.
"""

import logging
import time
import csv
import os
import sys
import networkx as nx
import numpy as np
import math

import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.motion_commander import MotionCommander
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncLogger import SyncLogger
from cflib.utils.multiranger import Multiranger



URI = 'radio://0/80/2M'
if len(sys.argv) > 1:
    URI = sys.argv[1]

#Bitcraze
def position_callback(timestamp, data, logconf):
    x = data['kalman.stateX']
    y = data['kalman.stateY']
    z = data['kalman.stateZ']
    #print('pos: ({}, {}, {})'.format(x, y, z))
    with open('position.csv', 'a') as csvfile:
        writer = csv.writer(csvfile,delimiter=',')
        writer.writerow([x, y, z])
    csvfile.close()

#Bitcraze
def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=500)
    log_conf.add_variable('kalman.stateX', 'float')
    log_conf.add_variable('kalman.stateY', 'float')
    log_conf.add_variable('kalman.stateZ', 'float')
    scf.cf.log.add_config(log_conf)
    log_conf.data_received_cb.add_callback(position_callback)
    log_conf.start()
```

```
def is_close(range):
    MIN_DISTANCE = 0.5  # m

    if range is None:
        return False
    else:
        return range < MIN_DISTANCE


def recalc_path(current_coord, next_coord, end_coord):
    G.remove_edge(current_coord,next_coord)
    last_coord = next_coord
    nodes_shortest = nx.dijkstra_path(G,source=current_coord, target=end_coord)
    next_coord = nodes_shortest[1]
    angles = angles_path(nodes_shortest)
    angle = angle_nodes(current_coord, next_coord, last_coord)
    return next_coord, nodes_shortest, angles, angle

def make_graph():
    G = nx.DiGraph()
    with open('copycoordinates.csv', 'r') as file:
        reader = csv.reader(file)
        line_count = 0
        for row in reader:
            nr_nodes = int(len(row)/3)
            nr_paths = nr_nodes-1
            x = []
            y = []
            z = []
            for i in range(0,nr_nodes):
                x.append(float(row[0+3*i]))
                y.append(float(row[1+3*i]))
                z.append(float(row[2+3*i]))
            for i in range(0,nr_nodes-1):
                Weight = abs(x[0]-x[i+1]+y[0]-y[i+1]+z[0]-z[i+1])
                G.add_edge((x[0],y[0],z[0]),(x[i+1],y[i+1],z[i+1]), weight = Weight)

    return G

def change_startcoord():
    with open('testpath.csv', 'r') as file:
        reader = csv.reader(file)
        lines = list(reader)
        lines[0][2] = float(lines[0][2])-0.0001
    file.close()

    with open('copycoordinates.csv', 'w') as file:
        writer = csv.writer(file)
        writer.writerows(lines)
    file.close()
    return lines


def angles_path(nodes_shortest):
    angles = []
```

```
    for i in range(1, len(nodes_shortest)-1):
        p0 = nodes_shortest[i]
        p1 = nodes_shortest[i-1]
        p2 = nodes_shortest[i+1]
        v1 = np.array([p1[0]-p0[0], p1[1]-p0[1]])
        v2 = np.array([p2[0]-p0[0], p2[1]-p0[1]])
        angle = np.math.atan2(np.linalg.det([v1,v2]),np.dot(v1,v2))
        angles.append(180-np.degrees(angle))
    angles.append(0)
    print(angles)
    return angles

def angle_nodes(current, next, last):
    v1 = np.array([last[0]-current[0], last[1]-current[1]])
    v2 = np.array([next[0]-current[0], next[1]-current[1]])
    angle = np.degrees(np.math.atan2(np.linalg.det([v2,v1]),np.dot(v2,v1)))
    return angle


def fly(G, lines, mc, multiranger):
    nodes_shortest = nx.dijkstra_path(G,source=(float(lines[0][0]),float(lines[0][1]),
        float(lines[0][2])), target=(float(lines[0][0]),float(lines[0][1]),float(lines
        [0][2])+0.0001))
    angles = angles_path(nodes_shortest)
    current_coord = nodes_shortest[0]
    next_coord = nodes_shortest[1]
    end_coord = nodes_shortest[len(nodes_shortest)-1]
    print(current_coord)
    coord_index = 1
    traveled_path = []
    latest_angles = []
    traveled_path.append(current_coord)

    while current_coord != end_coord:
        time.sleep(1)
        angle=angles[coord_index-1]

        #Check if there is a obstacle placed at the splitting point
        if is_close(multiranger.front):
            print('Obstacle placed close to the splitting point')
            next_coord, nodes_shortest, angles, correction_angle = recalc_path(
                current_coord, next_coord, end_coord)

            #If obstacle is found turn to the coordinate next in line
            if correction_angle > 180:
                mc.turn_left(360-correction_angle, 360.0/2.0)
                print('turn left')

            else:
                mc.turn_right(correction_angle,360.0/2.0)
                print('turn right')

            coord_index = 1

       #If no obstacle is found, go to next coordinate but stop before and check if it
            is a dead end
```

```
        else:
            distance_x = next_coord[0]-current_coord[0]
            distance_y = next_coord[1]-current_coord[1]
            distance_z = next_coord[2]-current_coord[2]
            x_distance = math.sqrt((distance_x*distance_x)+(distance_y*distance_y))
                -0.2
            mc.move_distance(x_distance-0.2, 0.0, distance_z)
            time.sleep(1)

            #Check to see if it is a dead end if it is move back to the latest
                splitting point
            if is_close(multiranger.front):
                mc.move_distance(-(x_distance-0.2), 0, -(distance_z))

                for i in range(len(traveled_path)):
                    if len(G.adj[(traveled_path[-(i+1)])])>=2:
                        splitting_point = traveled_path[-(i+1)]
                        break
                    else:
                        pass

                #If the last splitting point is the last coordinate go back to it
                if splitting_point == current_coord:
                    next_coord, nodes_shortest, angles, correction_angle = recalc_path
                        (current_coord, next_coord, end_coord)

                    if correction_angle > 180:
                        mc.turn_left(360-correction_angle, 360.0/2.0)

                    else:
                        mc.turn_right(correction_angle,360.0/2.0)


                #If the last splitting point is several coordinates away, retrack to
                    it
                else:
                    print(current_coord)
                    retrack = nx.dijkstra_path(G,source=splitting_point, target=
                        current_coord)
                    print(retrack)
                    G.remove_edge(current_coord,next_coord)

                    for i in range(len(retrack)-1):
                        distance_x = retrack[-(i+2)][0]-current_coord[0]
                        distance_y = retrack[-(i+2)][1]-current_coord[1]
                        distance_z = retrack[-(i+2)][2]-current_coord[2]
                        x_distance = math.sqrt((distance_x*distance_x)+(distance_y*
                            distance_y))
                        latest_angle = latest_angles[-(i+1)]

                        if latest_angle > 180:
                            mc.turn_right(360-latest_angle, 360.0/2.0)

                        else:
                            mc.turn_left(latest_angle,360.0/2.0)
```

```
                        mc.move_distance(x_distance, 0.0, distance_z)
                        traveled_path.pop()
                        current_coord = retrack[-(i+2)]
                        print('current_coord')
                        print(current_coord)

                    next_coord, nodes_shortest, angles, correction_angle = recalc_path
                        (current_coord, retrack[1], end_coord)
                    time.sleep(1)

                    if correction_angle > 180:
                        mc.turn_left(360-correction_angle, 360.0/2.0)

                    else:
                        mc.turn_right(correction_angle,360.0/2.0)

                    time.sleep(1)
                    coord_index = 1

            #If no dead end is found, move all the way to the node
            else:
                mc.move_distance(0.20, 0.0, 0.0)
                time.sleep(1)

                if angle > 180:
                    mc.turn_left(360-angle, 360.0/2.0)
                    latest_angles.append(angle)

                else:
                    mc.turn_right(angle,360.0/2.0)
                    latest_angles.append(angle)


                time.sleep(1)

                #Update current coordinate to next coordinate in the shortest path
                coord_index += 1
                if coord_index < len(nodes_shortest):
                    current_coord = next_coord
                    traveled_path.append((current_coord))
                    print(current_coord)
                    next_coord = nodes_shortest[coord_index]
                else:
                    break
    print(next_coord)


# Only output errors from the logging framework
logging.basicConfig(level=logging.ERROR)


if __name__ == '__main__':
    try:
        os.remove('position.csv')
    except:
```

```
        print('Already deleted.')
    lines = change_startcoord()
    G = make_graph()

    cflib.crtp.init_drivers(enable_debug_driver=False)
    with SyncCrazyflie(URI, cf=Crazyflie(rw_cache='./cache')) as scf:
        start_position_printing(scf) #Log the drone's path
        with MotionCommander(scf) as mc:
            with Multiranger(scf) as multiranger:
                mc.up(0.3)
                fly(G,lines, mc, multiranger)
```

D.2.2 Method 2

```
# -*- coding: utf-8 -*-
#
#     ||          ____  _ __
#  +------+      / __ )(_) /_______________ _____  ___
#  | 0xBC |     / __  / / __/ ___/ ___/ __ `/_  / / _ \
#  +------+    / /_/ / / /_/ /__/ /  / /_/ / / /_/  __/
#   ||  ||    /_____/_/\__/\___/_/   \__,_/ /___/\___/
#
#  Copyright (C) 2017 Bitcraze AB
#
#  Crazyflie Nano Quadcopter Client
#
#  This program is free software; you can redistribute it and/or
#  modify it under the terms of the GNU General Public License
#  as published by the Free Software Foundation; either version 2
#  of the License, or (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software
#  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
#  MA  02110-1301, USA.
"""
This script flies one crazyflie autonomously through an obstacle course.
The coordinates are read from a file, placed in a directed graph and the
shortest path through the course is calculated and sent to the drone.
The drone can detect obstacles 50 cm away (def in is_close) and can handle
obstacles at splitting points and dead ends.

The script is designed for the floe deck.
"""

import logging
import time
import csv
import os
import sys
```

```
import networkx as nx
import numpy as np
import math

import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.motion_commander import MotionCommander
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncLogger import SyncLogger
from cflib.utils.multiranger import Multiranger



URI = 'radio://0/80/2M'
if len(sys.argv) > 1:
    URI = sys.argv[1]

#Bitcraze
def position_callback(timestamp, data, logconf):
    x = data['kalman.stateX']
    y = data['kalman.stateY']
    z = data['kalman.stateZ']
    #print('pos: ({}, {}, {})'.format(x, y, z))
    with open('position.csv', 'a') as csvfile:
        writer = csv.writer(csvfile,delimiter=',')
        writer.writerow([x, y, z])
    csvfile.close()

def get_position():
    with open('position.csv', 'r') as file:
        reader = csv.reader(file)
        lines = list(reader)
    file.close()
    return lines[-1]

#Bitcraze
def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=500)
    log_conf.add_variable('kalman.stateX', 'float')
    log_conf.add_variable('kalman.stateY', 'float')
    log_conf.add_variable('kalman.stateZ', 'float')
    scf.cf.log.add_config(log_conf)
    log_conf.data_received_cb.add_callback(position_callback)
    log_conf.start()


def is_close(range):
    MIN_DISTANCE = 0.5  # m

    if range is None:
        return False
    else:
        return range < MIN_DISTANCE
```

```
def recalc_path(current_coord, next_coord, end_coord):
    G.remove_edge(current_coord,next_coord)
    nodes_shortest = nx.dijkstra_path(G,source=current_coord, target=end_coord)
    next_coord = nodes_shortest[1]
    angles = angles_path(nodes_shortest)
    return next_coord, nodes_shortest, angles

def make_graph():
    G = nx.DiGraph()
    with open('copycoordinates.csv', 'r') as file:
        reader = csv.reader(file)
        line_count = 0
        for row in reader:
            nr_nodes = int(len(row)/3)
            nr_paths = nr_nodes-1
            x = []
            y = []
            z = []
            for i in range(0,nr_nodes):
                x.append(float(row[0+3*i]))
                y.append(float(row[1+3*i]))
                z.append(float(row[2+3*i]))
            for i in range(0,nr_nodes-1):
                Weight = abs(x[0]-x[i+1]+y[0]-y[i+1]+z[0]-z[i+1])
                G.add_edge((x[0],y[0],z[0]),(x[i+1],y[i+1],z[i+1]), weight = Weight)

    return G

def change_startcoord():
    with open('testpath.csv', 'r') as file:
        reader = csv.reader(file)
        lines = list(reader)
        splitting_points = []
        lines[0][2] = float(lines[0][2])-0.0001
    file.close()

    with open('copycoordinates.csv', 'w') as file:
        writer = csv.writer(file)
        writer.writerows(lines)
    file.close()
    return lines

def remove_lines(end_coord):
    with open('position.csv', 'r') as file:
        reader = csv.reader(file)
        line = list(reader)
        diff_x = float(line[9][0]) - end_coord[0]
        diff_y = float(line[9][1]) - end_coord[1]
        diff_z = float(line[9][2]) - end_coord[2]
        counter = 0
        for rows in line:
            if counter > 8:
                line[counter][0] = float(line[counter][0])-diff_x
                line[counter][1] = float(line[counter][1])-diff_y
                line[counter][2] = float(line[counter][2])-diff_z
            else:
```

```
                pass
            counter+=1
    file.close()
    with open('position.csv', 'w') as file:
        writer = csv.writer(file)
        writer.writerows(line[9:])
    file.close()

def diff(start_coord):
    with open('position.csv', 'r') as file:
        reader = csv.reader(file)
        line = list(reader)
        diff_x = float(line[9][0]) - start_coord[0]
        diff_y = float(line[9][1]) - start_coord[1]
    file.close()
    return diff_x, diff_y

def correct(diff_x, diff_y, next_coord):
    pos = get_position()
    current_x = pos[0]-diff_x
    current_y = pos[1]-diff_y
    correction_x = next_coord[0]-current_x
    correction_y = next_coord[1]-current_y
    return correction_x, correction_y

def sensor(forward_sensor):
    if forward_sensor == 'left':
        multisensor = multiranger.left
    elif forward_sensor == 'right':
        multisensor = multiranger.right
    elif forward_sensor == 'front':
        multisensor = multiranger.front
    else:
        multisensor = multiranger.back
    return multisensor

def direction_of_travel(angle):
    if angle > 45 and angle < 135 :
        direction = 'left'
    elif angle >-45 and angle < 45:
        direction =  'front'
    elif angle > -135 and angle < -45:
        direction = 'right'
    else:
        direction = 'back'
    return direction

def angles_path(nodes_shortest):
    angles = []

    for i in range(0, len(nodes_shortest)-1):
        p0 = nodes_shortest[i] #current
        p1 = nodes_shortest[i+1]  #next
        v1 = np.array([p1[0]-p0[0], p1[1]-p0[1]])
        v2 = np.array([1, 0])
        angle = np.math.atan2(np.linalg.det([v2,v1]),np.dot(v2,v1))
```

```
        angles.append(np.degrees(angle))
    return angles


def fly(G, lines, mc, multiranger):
    velocity = 0.6
    nodes_shortest = nx.dijkstra_path(G,source=(float(lines[0][0]),float(lines[0][1]),
        float(lines[0][2])), target=(float(lines[0][0]),float(lines[0][1]),float(lines
        [0][2])+0.0001))
    angles = angles_path(nodes_shortest)
    current_coord = nodes_shortest[0]
    diff_x, diff_y = diff(current_node)
    next_coord = nodes_shortest[1]
    end_coord = nodes_shortest[len(nodes_shortest)-1]
    print(current_coord)
    coord_index = 0
    forward_sensor = 'front'
    traveled_path = []
    traveled_path.append(current_coord)
    while current_coord != end_coord:
        time.sleep(1)
        multisensor = sensor(forward_sensor)

        #Check if there is a obstacle placed at the splitting point
        if is_close(multisensor):
            print('Obstacle placed in splitting point')
            next_coord, nodes_shortest, angles = recalc_path(current_coord, next_coord
                , end_coord)
            coord_index = 0
            angle=angles[coord_index]
            forward_sensor = direction_of_travel(angle)
            print('Using sensor after splitting point')
            print(forward_sensor)

        #If no obstacle is found, go to next coordinate but stop before and check if
            it is a dead end
        else:
            distance_x = next_coord[0]-current_coord[0]
            distance_y = next_coord[1]-current_coord[1]
            distance_z = next_coord[2]-current_coord[2]
            mc.move_distance(0.8*distance_x, 0.8*distance_y, 0.0,velocity)
            time.sleep(1)

            multisensor = sensor(forward_sensor)

            #Check for dead end move back to the latest splitting point
            if is_close(multisensor):
                mc.move_distance(-(0.8*distance_x), -(0.8*distance_y), 0.0, velocity)

                for i in range(len(traveled_path)):
                    if len(G.adj[(traveled_path[-(i+1)])])>=2:
                        splitting_point = traveled_path[-(i+1)]
                        break
                    else:
                        pass
```

```
            #If the last splitting point is the last coordinate go back to it
            if splitting_point == current_coord:
                next_coord, nodes_shortest, angles = recalc_path(current_coord,
                    next_coord, end_coord)
                coord_index=0
                angle=angles[coord_index]
                forward_sensor = direction_of_travel(angle)
                print('Using sensor after splitting point == current_coord dead
                    end')
                print(forward_sensor)

            #If the last splitting point is several coordinates away, retrack to
                it
            else:
                print(current_coord)
                retrack = nx.dijkstra_path(G,source=splitting_point, target=
                    current_coord)
                G.remove_edge(current_coord,next_coord)

                for i in range(len(retrack)-1):
                    distance_x = retrack[-(i+2)][0]-current_coord[0]
                    distance_y = retrack[-(i+2)][1]-current_coord[1]
                    distance_z = retrack[-(i+2)][2]-current_coord[2]
                    mc.move_distance(distance_x, distance_y, 0.0, velocity)
                    traveled_path.pop()
                    current_coord = retrack[-(i+2)]
                    print('current_coord')
                    print(current_coord)

                next_coord, nodes_shortest, angles = recalc_path(current_coord,
                    retrack[1], end_coord)
                coord_index=0
                angle=angles[coord_index]
                forward_sensor = direction_of_travel(angle)
                print('Using sensor after dead end')
                print(forward_sensor)
                time.sleep(1)

        #If no dead end is found, move all the way to the node
        else:
            mc.move_distance(0.2*distance_x, 0.2*distance_y, 0.0, velocity)
            time.sleep(1)

            #Correct position in relation to the current logged position
            correction_x, correction_y = correct(diff_x, diff_y, next_coord)
            mc.move_distance(correction_x, correction_y, 0.0, velocity)

            #Update current coordinate to next coordinate in the shortest path
            coord_index += 1
            if coord_index+1 < len(nodes_shortest):
                current_coord = next_coord
                traveled_path.append((current_coord))
                print(current_coord)
                next_coord = nodes_shortest[coord_index+1]
                print(next_coord)
```

```
                else:
                    break
                angle=angles[coord_index]
                forward_sensor = direction_of_travel(angle)
                print('Using sensor after updating angle')
                print(forward_sensor)

    print(next_coord)


# Only output errors from the logging framework
logging.basicConfig(level=logging.ERROR)


if __name__ == '__main__':
    try:
        os.remove('position.csv')
    except:
        print('Already deleted.')

    lines = change_startcoord()
    G = make_graph()

    cflib.crtp.init_drivers(enable_debug_driver=False)
    with SyncCrazyflie(URI, cf=Crazyflie(rw_cache='./cache')) as scf:
        start_position_printing(scf) #Log the drone's path
        with MotionCommander(scf) as mc:
            with Multiranger(scf) as multiranger:
                mc.up(0.3)
                fly(G,lines, mc, multiranger)
```