Implementation of Decentralized Multi-Agent control laws on Nanoquadcopter platform

Submitted in partial fulfillment of the requirements

of the degree of

Master of Technology

by

V Adarsh

Roll no. 21307R001

Under the guidance of

Prof. Dwaipayan Mukherjee

Department of Electrical Engineering
Indian Institute of Technology Bombay
2024

Abstract

Multi-agent decentralized control laws have been gaining immense popularity in recent times. This is because, instead of using a single complex agent, multiple relatively simpler agents can co-ordinate with each other in some manner and achieve a task. In this thesis, three such control laws, developed by different authors, were implemented on a nanoquadcopter platform to validate the theoretical findings and test the efficacy of the control laws in real-world application. The nanoquadcopter developed by Bitcraze, crazyflie 2.1 was used for this purpose. It comes with a completely open source firmware and a python library that helped in easy in seamless implementation of control laws.

Thesis Approval

This thesis entitled **Implementation of Decentralized Multi-Agent control laws on Nanoquadcopter platform** by **V Adarsh**, Roll No. 21307R001, is approved for the degree of **Masters of Technology**.

Examiners

Digital Signature
Harish K Pillai (i01060)
26-Jun-24 05:41:39 PM

Debraj Chakraborty

Supervisor

Dwaipayan Mukherjee

Chairman

Debraj Chakraborty

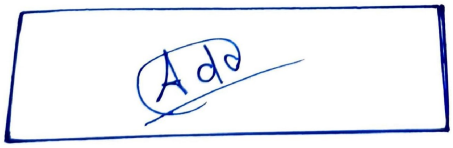
Date: **01/07/2024**

Place: **IIT Bombay**

https://ams.iitb.ac.in/d/153153-09JE2SNX3YB3SH44

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.



(Signature)

V. ADARSH

(Name)

21307R001

(Roll No.)

Date: 25/6/2024

Contents

List of Figures

Chapter 1

Introduction

Decentralized control refers to a scenario where the control of a system is distributed across various individual subsystems. Every subsystem works together to achieve a common goal. In this thesis, the subsystem will be referred to as an *agent*. Relying on one central controller reduces fault tolerance because if the central controller fails, all the agents, which rely on it become useless. Whereas, having a distributed control mitigates this issue, since each agent makes it's decisions based on information from its *neighbors* (a subset of all agents in the system), thereby enhancing overall resilience of the system.

Three such control laws have been implemented on a Nanoquadcopter platform. These control laws are mainly designed for multiple mobile agents where each of them have to sense some information from their neighbors and collectively work together to achieve a common goal. The control laws act as the control input that is given to the mobile agents, which in this case are quadcopters. Quadcopters, with 6 degrees of freedom, are omnidirectional and can freely move in all 3 directions. Further the control laws developed considered agents as point masses with no dynamics, assuming they can move in all directions. Due to these reasons quadcopters were chosen for this thesis.

This thesis aims to provide a proof of concept for the different control laws. The control laws have been implemented on a real-time system, specifically the nanonquadcopter platform, and different experiments have been conducted in an attempt to validate the theoretical findings. These experiments aimed to confirm the practicality, robustness and the efficacy of these control laws in real-world application. Although some parts of the control laws could not be successfully implemented due to challenges encountered during real-time implementation on quadcopters, the work done, and observations made along the way provides valuable insights in

extending this work and improving the practical implementation of these control laws.

1.1 Outline of thesis

Chapter 2 talks about the nanoquadcopter platform used in this work and provide a brief overview of it's technical details and how the control laws were implemented using it.

Chapter 3 introduces the first of the 3 different control laws implemented in this thesis. It's a formation control task [11] where different agents have to converge to a pre-sepcified formation in the space. The agents are treated as singe integrators.

Chapter 4 is about a containment control algorithm where different agents,referred to as followers, have to ensure they are contained within the convex hull formed by another set of agents, called leaders. This control law was developed by: Avinash Dubey, PhD student (Dept of Electrical Engineering, IIT Bombay), Prof Dwaipayan Mukherjee (Dept of Electrical Engineering, IIT Bombay), Prof Kavi Arya (Dept of Computer Science and Engineering, IIT Bombay)

Chapter 5 presents the final control law implemented, which is also a formation control task, except here the agents are treated as double integrators. The main challenge here was to develop an acceleration controller for quadcopter since the control law for the agents is acceleration. This control law was developed by: Susmitha T Rayabagi,PhD student (Dept of Electrical Engineering, IIT Bombay), Prof Dwaipayan mukherjee (Dept of Electrical Engineering, IIT Bombay), Prof Debasattam Pal (Dept of Electrical Engineering, IIT Bombay)

Supplemental material: The codes and videos of the above implementations can be found in this repository.

Chapter 2

The Nanoquadcopter Platform: Crazyflie 2.1

Crazyflie 2.1 is a nanoquadcopter developed by bitcraze [2]. It weighs only 27 grams and is very compact in size hence making it feasible for indoor flights. It is completely open source allowing the user to program and control it using their own software.

Figure 2.1: Crazyflie 2.1 [2]

The bitcraze company also develops crazyradio, which is used for communicating with crazyflie wirelessly.This device is a USB radio dongle that can be plugged into any computer USB port. The crazyflie has a onboard micro controller(STM32F405) which takes care of low level tasks like flight control and state estimation. Using the python API library developed by bitcraze,different decentralized multi-agent algorithms can be implemented in any application like Visual studio using python language and the high level commands like velocity, position, and attitude set-points can be sent to crazyflie via the crazyradio.

Specifications of crazyflie are:

- Weighs 27 grams
- (LxWxH): 92x92x29mm (motor-to-motor and including motor mount feet)
- STM32F405 main application MCU (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash), responsible for flight control and state estimation
- nRF51822 radio and power management MCU (Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash)
- IMU: 3-axis gyro, accelerometer, and magnetometer
- High precision pressure sensor (BMP388)

2.1 Positioning system

Figure 2.2: The 2 base stations in LPS [3]

The crazyflie uses lighthouse positioning system(LPS) to estimate it's position in a global frame of reference. LPS is a optically-based positioning system developed by the company Valve corporation. It uses 2 base stations (see figure 2.2) that contain Infrared LED array and 2 rotating laser planes, one horizontal and other vertical, that sweep the 3D volume. The crazyflie has a lighthouse deck that has photo-diodes mounted on them which can detect these laser sweeps and can estimate their position.

Each laser plane rotates at 60 rotations per second. At beginning of each cycle the LED array sends out an omnidirectional flash, then the vertical plane sweeps the 3D volume and then the horizontal plane sweeps the volume. And then the cycle repeats. Since the laser sweeps rotate at 60 Hz, the time taken to sweep the 3D volume is half the full rotation, i.e 8.3 ms. So at each cycle, after the LED flash, 1^{st} plane sweeps in 8.3 ms, immediately followed by the next plane. Then the cycle continues.

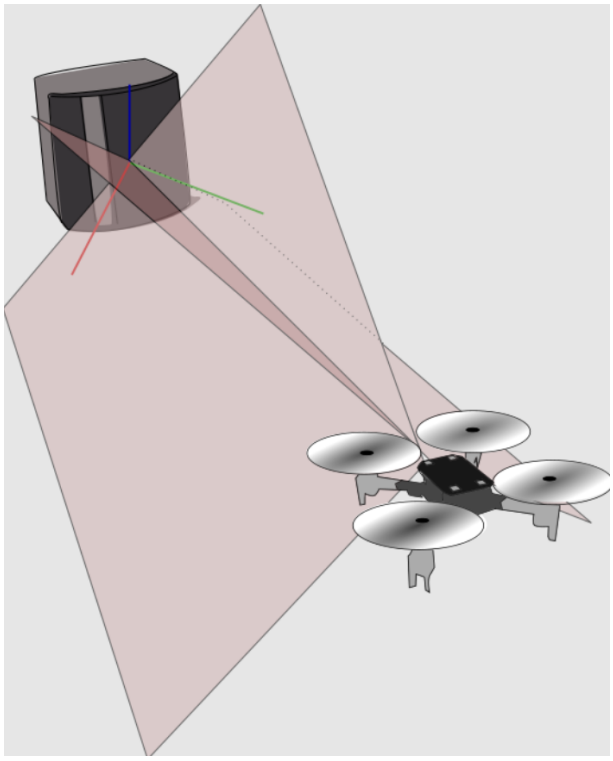


Figure 2.3: Intersection of laser planes [3]

As soon as the LED omnidirectonal flash is sent out, it can be detecetd by crazyflie photo-diodes, and the onboard microcontroller STM32 starts a counter and starts counting time until the first laser plane hits the photo-diode. Since the total sweeping time is known (8.3 ms), and using this info along with the time counted by crazyflie, the angle made by laser plane when it hits the photo-diode with the direction along which the base station looks can be calculated. Using this angle the normal to this plane can be calculated. The same can be then done wit the second laser plane. Now we have 2 planes defined by their normal vectors when both of them hit the photo-diode. This means their line of intersection gives us the vector from the base station to the crazyflie. A similar vector can be obtained by the second base station.(see figure 2.3) *(Note that these vectors are being calculated in base station frame and not global frame. The crazyflie uses a separate client software which must run on a PC in the initial calibra-*

tion process where the geometry or the rotation matrices and translation of base stations are acquired. The global frame's origin is defined by the point where the crazyflie is initially placed and the coordinate system is same as it's body frame.) The 2 vectors are then transformed to global frame.

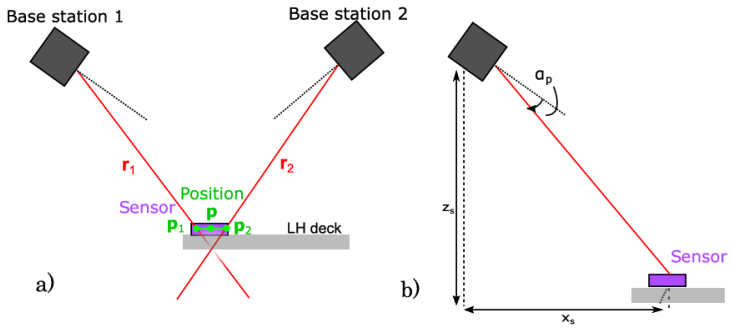


Figure 2.4: (a) 2 vectors from 2 base stations (b) Measurement model depiction for EKF.Image taken from [8]

Now these 2 vectors only encode directional information relative to base station and not the distances. Hence both of them are used in conjunction to triangulate crazyflie's precise location. The 2 vectors must definitely intersect at some point. Since these calculationa are done at very high frequency(120 Hz) the point of intersection must be very near to actual location of crazyflie. So the point of intersection of these 2 vectors from 2 base stations gives the location of crazyflie in global frame.(see figure 2.4(a)). Readers are referred to [8, 6, 3] for more details.

The crazyflie developer further modified this process by implemeneting a kalman filter to estimate the position by just using one base station. The angle of laser plane when it hits the photo-diode can be mapped to photo-diode position using a measurement model. The visualization is shown is figure 2.4(b). [8, 3] has full details of this implementation.

Hence using kalman filter and fusing readings from LPS and onboard IMU, the global position,velocity and attitude is available for crazyflie. All these calculations are already implemented in crazyflie's onboard firmware so user need not worry about this and can readily use the estimates.

2.2 The PID controller

The high level control input obtained from decentralized control laws come in form of position and velocity set-points. To track these commands crazyflie uses a cascaded PID controller. Figure 2.5 shows the block diagram of the same.

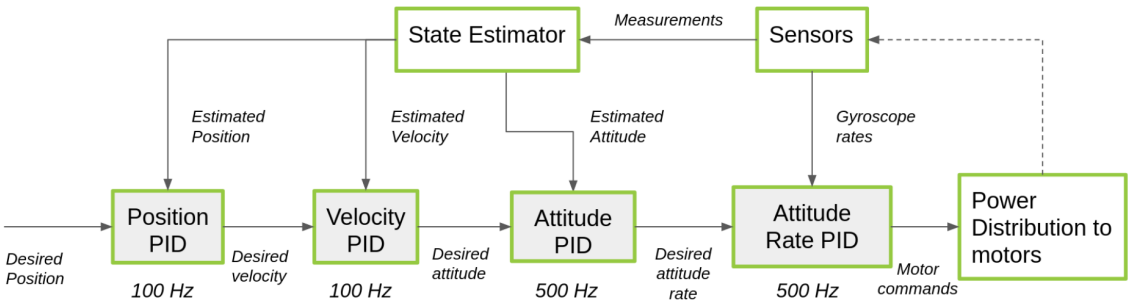


Figure 2.5: The cascaded PID controller. [2]

Position/Velocity controller:The high-level controller, which is the decentralized control law in this thesis, will send position and velocity set-points to the PID position/velocity controllers. These blocks receive the current position and velocity estimated by the kalman filter using the Lighthouse system and IMUs. Then, these blocks will generate a PID signal that correspond to desired roll and pitch angles which are sent to attitude controller.

Attitude Controller: Based on the desired attitude calculated in previous block, and the current attitude estimated by kalman filter, it determines the desired angular rates, which is the sent to rate controller.

Rate Controller: The attitude rate controller receives the current rates as gyroscope measurements. Using desired rates, the output of rate controller generates the desired thrusts needed for the all 4 motors to produce desired motion.

2.3 Real-time implementation details

Although the thesis seeks to provide a real-time implementation fo decentralized control algorithms, the actual implementation was *Pseudo-Decentralized*, meaning all the high level control of all the agents was taking place in a central computer. So when it says each agent senses information from it's neighbors, this communication was not single-hop, but happened through the central PC. For example, if the control task requires an agent to sense position information of it's neighbors and calculate a velocity that it must follow, it happened as follows:

- Each agent/crazyflie has it's position information in it's onboard microcontroller, estimated by the kalman filter using LPS and IMU readings.
- For each agent, based on the underlying communication topology defined by the graph, the position information of corresponding neighbor agent was logged into the PC through the crazyradio.
- Once all the neighbors information was logged into PC, the python code written on PC calculates the required velocity for that agent as per control law. This velocity was then sent as a high level set-point through crazyradio to the crazyflie. The cascaded PID running on it's onboard MCU takes care of the rest.

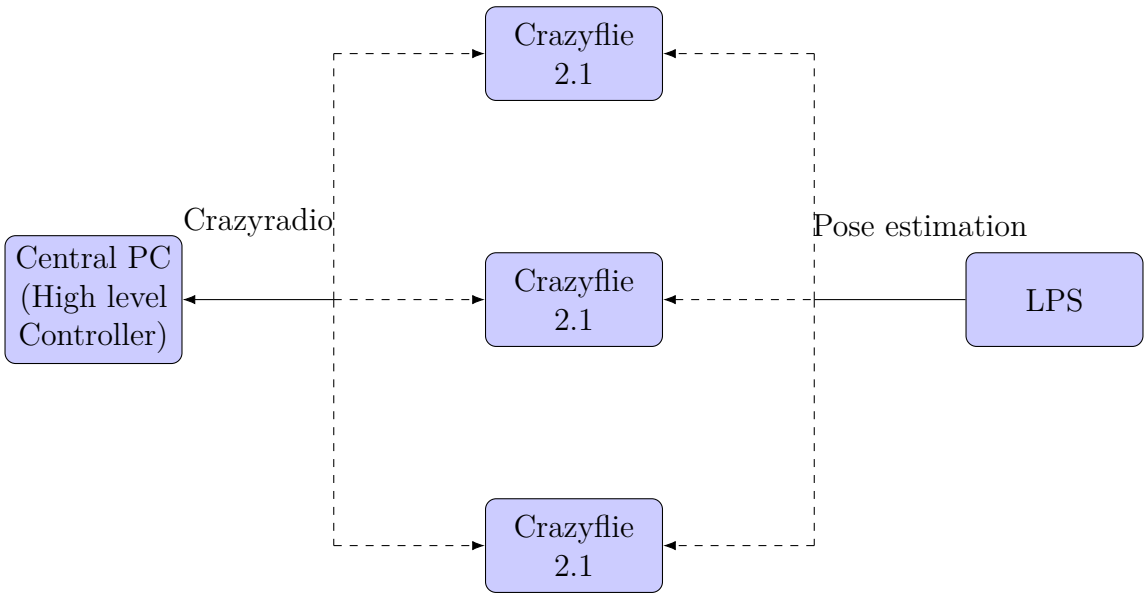


Figure 2.6: High level system diagram

The code base to implement different algorithms was written in python on the central PC and utilizes Bitcraze's python library. The library provides nice high level functions that directly take care of tasks like logging data from crazyflie, sending position,velocity and attitude set-points to onboard controller via crazyradio etc.

Since this thesis deals with multi-agent setup, each crazyflie was given a unique resource identifier (URI). This URI was used to distinguish between different agents in the multi-agent setup. The address/URI can be changed by using a client software provided by bitcraze and connecting the crzyflie to PC via crazyradio. Each radio can efficiently communicate with a maximum of 3 crazyflies without any delays. If more than 3 crazyflies are present multiple radios cab be used. Each crazyflie's URI can be configured by mentioning the identifier of the particular crazyradio in it's URI. This way each radio will only communicate with the crazyflie that has it's identifier in it's URI. For example if we want crazyradio-1 to communicate with crazyflie-1 and crazyradio-2 to communicate with crazyflie-2, this can be done by configuring the URIs of the crazyflie as:

```
URI_crazyflie1='radio://1/80/2M/E7E7E7E701'
URI_crazyflie2='radio://2/80/2M/E7E7E7E702'
```

Here, the URI has following structure:
"radio://<radio identifier>/<channel>/<bit rate>/<crazyflie address>".

The address is of form "E7E7E7E7XX" where XX is the crazyflie number that is assigned to it. For example if a certain crazyflie has to be treated as agent-1, then the address would be "E7E7E7E701".

In code implementation, their are 2 main things:(i) Logging data from crazyflies and using it as feedback for high level controller (ii) calculating the control input and sending it to crazyflie.For logging data, a callback function was setup so whenever a data packet is received by radio, it updates the necessary values. A control loop runs which implements the control law and sends set-points continuously. The logging was done asynchronously, meaning it was not dependent on the main control loop. This is preferable as the logging is not delayed by any heavy computation in control loop. The data was logged at every 10ms and control loop sent set-points at every 10ms.

To facilitate control of swarm of crazyflies, the bitcraze library's **swarm** class was used. It uses muti-threading and executes the control loop and data logging functions parallely in asynchronous way(one thread for each crazyflie) for each crazyflie.

```

import datetimeimport numpy as np
import cflib.crtp
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.swarm import CachedCfFactory
from cflib.crazyflie.swarm import Swarm
from cflib.crazyflie.commander import Commander
```

Listing 2.1: Necessary imports

The name of crazyflie python library is *cflib*. **cflib.crtp** imports the crazyflie real-time protocol class (crtp) which facilitates the wireless communication of crazyflie and crazyradio. **Logconfig** class helps setting up the data logging process. **Swarm** class is used to run the control task for a swarm of crazyflies paralleley using multi-threading.**commander** class contains high-level functions the directly takes in setpoints like velocity, position and attitude and sends it to crazyflie via the radio.

```

URI1='radio://0/80/2M/E7E7E7E701'
URI2='radio://0/80/2M/E7E7E7E702'
URI3='radio://0/80/2M/E7E7E7E703'
uris={URI1,URI2,URI3}
```

Listing 2.2: URIs

Here the dictionary of different crazyflie URIs is initialized which will be used by swarm class.

```
def log_callback(timestamp,data,logconf):
        pos_x=data['stateEstimate.x']
        pos_y=data['stateEstimate.y']
        pos_z=data['stateEstimate.z']
        roll=data['stateEstimate.roll']
        pitch=data['stateEstimate.pitch']
        yaw=data['stateEstimate.yaw']
def get_data(scf):
    logconf = LogConfig(name='Pose', period_in_ms=10)
    logconf.add_variable('stateEstimate.x', 'float')
    logconf.add_variable('stateEstimate.y', 'float')
    logconf.add_variable('stateEstimate.z', 'float')
    logconf.add_variable('stateEstimate.roll','float')
    logconf.add_variable('stateEstimate.pitch','float')
    logconf.add_variable('stateEstimate.yaw','float')
    logconf.uri=scf.cf.link_uri
    scf.cf.log.add_config(logconf)
    logconf.data_received_cb.add_callback(log_callback)
    logconf.start()
```

Listing 2.3: loggign data

Next step is to setup the data logging asynchronously. The function **get_data()** creates a log configuration variable named **logconf**. This variable has position and attitude details. The line *scf.cf.log.add_config(logconf)* starts the logging process asynchronously. Every time a log block, configured as logconf, is received the callback function **log_callback()** is called.

```
def control_loop(scf):
   # the algorithm to find the control input for each agent
   # is implemeneted here
   cf.commander.send_position_setpoint(x,y,z,yaw)
   cf.commander.send_velocity_world_setpoint(vx,vy,vz,yawRate)
   cf.commander.send_setpoint(roll,pitch,yawrate,thrust)
   cf.commander.send_zdistance_setpoint(roll,pitch,yawRate,z)
```

Listing 2.4: control loop

The **send_position_setpoint(x,y,z,yaw)** is used when the control law generates a position setpoint for the crazyflie to follow. In case of single-integrators, control law is a velocity and hence the function **send_velocity_world_setpoint(vx,vy,vz,yawRate)** can be used. In case of double-integrators control input is acceleration which requires direct control of attitude and thrust and hence the functions **send_setpoint(roll,pitch,yawrate,thrust)** and **send_zdistance_setpoint(roll,pitch,yawRate,z)** can be used.

```
    if __name__=="__main__":
        cflib.crtp.init_drivers()
        factory=CachedCfFactory(rw_cache="./cache")
        with Swarm(uris,factory=factory) as swarm:

           swarm.parallel(get_data)
           time.sleep(2)
           swarm.parallel(control_loop)
```

Listing 2.5: driver code

The **swarm** class opens an instance of swarm with the URIs as defined earlier passed onto it. This opens a instance of all the crazyflies defined by the URIs passed on to it. Then the method **swarm.parallel()** starts running the function parallely for all the crazyflies.

Chapter 3

Bearing-only formation control of single integrators

This section implements a control law proposed in [11].
The control law is for a formation control task where various mobile agents communicate with each other and exchange certain information about each other and converge to a specified target formation in 2D or 3D. It deals with development of decentralized control laws under which the group of agents are guaranteed to (at least)locally converge to desired formation. The agents here are modelled as *single integrators*

There are mainly 2 types of information exchange between agents that has been studied extensively. One is exchange of inter-agent distances. However measuring inter-agent distances in real-time using sensors can be challenging. Sensors like ultrasonic sensor fails when there is no direct line of sight available due to some occlusion. Cameras also inherently do not encode depth of the scene, and hence complex techniques like stereo-vision, use of multiple cameras or knowledge of object size is required for measuring distances. Another type of information exchange is bearing measurements, which overcomes challenges faced by distance measurements. Usually it is easier to measure angles/bearings between agents from a single cameras using techniques like *Optical Flow*. In this section the information that is exchanged between neighbors is relative bearing.

The underlying communication graph is undirected.In a decentralized setting, an important condition that ensures proper convergence of control laws is the requirement of the communication graph among agents to be *Bearing Rigid*. Bearing rigidity ensures that just by specifying the inter-agent bearings between the neighbors the position of agents can be uniquely determined up-to a translation and scaling. This property comes handy where just by sharing this relative bearing information each agent can be uniquely localized in environment where something like GPS is de-

nied. Further this property also makes the formation control problem well defined, as in it removes any ambiguity in the target formation and hence there is a unique local equilibrium to which the control law can converge.

3.1 Bearing Rigidity

Let there be n agents with positions $\{p_i\}_{i=1}^n \in \mathbb{R}^d$, where $d \geq 2$. A *configuration* is defined as $p = [p_1^T, p_2^T, ..., p_n^T]^T \in \mathbb{R}^{dn}$. A *framework* in $\mathbb{R}^d$, denoted $\mathcal{G}(p)$ is a combination of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and configuration p. For a *framework* $\mathcal{G}(p)$:

$$e_{ij} = p_j - p_i \quad g_{ij} = \frac{e_{ij}}{\|e_{ij}\|} \tag{3.1}$$

Here g_{ij} is the relative bearing vector from agent i to agent j.

Define the orthogonal projection operator for any vector $x \in \mathbb{R}^d$ as $P : \mathbb{R}^d \to \mathbb{R}^{dxd}$

$$P(x) = P_x = I - \frac{xx^T}{\|x\| \, \|x^T\|} \tag{3.2}$$

Note that $P_x = P(x)$ projects and vector onto orthogonal complement of x.

Definition 1 (Bearing Equivalency): 2 frameworks $\mathcal{G}(p)$ and $\mathcal{G}(p')$ are bearing equivalent if $P_{p_i - p_j}(p_i' - p_j') = 0$ for all $(i, j) \in \mathcal{E}$

Definition 2 (Bearing Congruency): 2 frameworks $\mathcal{G}(p)$ and $\mathcal{G}(p')$ are bearing congruent if $P_{p_i - p_j}(p_i' - p_j') = 0$ for all $i, j \in \mathcal{V}$

Definition 3 (Bearing Rigidity): A framework $\mathcal{G}(p)$ is bearing rigid if there exist $\epsilon > 0$ such that any framework $\mathcal{G}(p')$ that is bearing equivalent to $\mathcal{G}(p)$ and satisfies $\|p - p'\| < \epsilon$ is also bearing congruent.

Figure-3.1 shows a framework that is not bearing rigid, both images on left and right have same edge bearings meaning bearing equivalent but the bearing between 1,3 or 2,4 is not the same hence not congruent so the formation specified by these edge bearings is not rigid. And this is why rigidity is important for unique localization since in this case there is

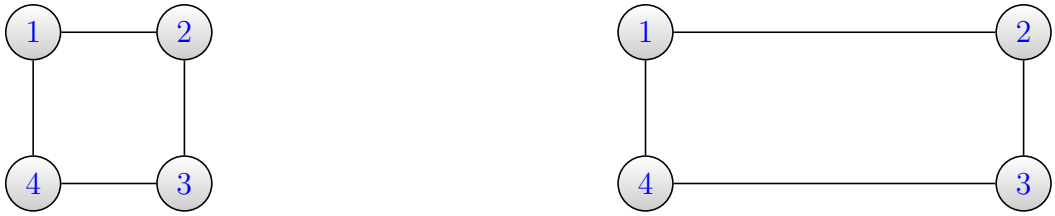


Figure 3.1: 2 frameworks that are bearing equivalent but not congruent

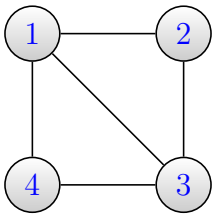


Figure 3.2: Rigid framework

ambiguity as to what is the location of 4 agents whose inter-edge bearings are specifies as in the figure. However if we add an edge between agent 2 and 4 as shown in fig-3.2, the framework becomes rigid. There are no smooth continuous motions of vertices or agents that preserve edge bearings. However, keep in mind that a bearing rigid framework is unique upto translation and scaling.

Let a framework have m edges, then denote $g = [g_1^T, g_2^T, ..., g_m^T]$ as the bearing of those edges. The bearing function can then be defines as:

$$F_B(p) = [g_1^T, g_2^T, ..., g_m^T]^T \tag{3.3}$$

Bearing function defines all bearings of framework. The *bearing rigidity matrix* can then be defined as:

$$R(p) = \frac{\partial F_B(p)}{\partial p} \in \mathbb{R}^{dm \times dn} \tag{3.4}$$

Infinitesimal bearing motion is defined as motion of agents that preserve edge bearings. Denoting $\delta p = \frac{dp}{dt}$ be the motion of agents that preserve edge bearings, we can write $\frac{dF_B(p)}{dt} = 0$, which by chain rule we get $\frac{\partial F_B(p)}{\partial p}\frac{dp}{dt} = 0$, so by (3.4) all such motions δp lie in null space of $R(p)$. Such a motion is called trivial if it corresponds to scaling or translation and a framework is *infinitesimally bearing rigid* if all it's infinitesimal bearing motions are trivial.

Theorem 1. *If a framework $\mathcal{G}(p)$ is infinitesimally bearing rigid then it's rigidity matrix's rank is $dn-d-1$ and it's null space is $< \mathbf{1} \otimes I_d, p >$.[11]*

1 is a d-dimensional vector of 1s. $\otimes$ denotes Kronecker's product. To understand theorem-1, it can be seen that from definition of infinitesimal bearing rigidity, all edge bearing preserving motions lie in null space of $R(p)$ and they are trivial,meaning the only motions that preserve edge bearings are translation and rotation. It can also be observed that $< \mathbf{1} \otimes I_d >$ is actually a translation and it is $d - dimensional$ and $< p >$ is scaling so $< \mathbf{1} \otimes I_d, p >$ is overall $d+1$ dimensional. Hence by rank-nullity theorem the rigidity matrix has a rank of $dn - d - 1$.

3.2 Bearing-only Formation Control law

All the vector quantities presented in this section are in global frame. Denote $p_i \in \mathbb{R}^d$ as position of agents-i. The single-integrator dynamics of agent-i is:

$$\dot{p}_i = v_i \tag{3.5}$$

where v_i i sthe velocity control input. Denote $p = [p_1^T, ..., p_n^T]^T \in \mathbb{R}^{dn}$ and $v = [v_1^T, ..., v_n^T]^T \in \mathbb{R}^{dn}$. The underlying communication graph G has m edges. The edge and bearing vectors are as defined in (3.1). Let $g = [g_1^T, ..., g_m^T]^T$ be the vector of current edge bearings. Let $g^* = [g_1^{T*}, ..., g_m^{T*}]^T$ be the desired bearing vector of the target formation.

The control law must drive the agents to a formation where the target bearings are satisfied, basically $g = [g_1^T, ..., g_m^T]^T \to g^*$.

Define $e_F = \frac{1}{2} \|g - g^*\|^2$

Then by taking negative gradient of above and equating it to zero (basically minimization of the abov error function) gives the following nonlinear control law:

$$v_i(t) = -\sum_{j \in \mathcal{N}_i} P_{g_{ij}(t)} g_{ij}^* \tag{3.6}$$

where $P_{g_{ij}} = I_d - g_{ij} g_{ij}^T$

Control law is bounded:
By definition of projection operator as in (3.2), $P_x = P_x^T$,$P_x = P_x^2$. Moreover it's eigenvalues are $\{0, 1^{(d-1)}\}$. The multiplicity of eigenvalue 1 is d-1 because for any vector that lies along the d-1 dimensions that are orthogonal to x, the projection operator does not do anything to it since it already lies in orthogonal complement. Hence the multiplicity d-1. This means maximum singular value is also 1. Consider

$$\left\|P_{g_{ij}(t)} g_{ij}^*\right\| \leq \left\|P_{g_{ij}(t)}\right\| \left\|g_{ij}^*\right\| \tag{3.7}$$

$$= \sqrt{\lambda_{max}(P_{g_{ij}(t)}^T P_{g_{ij}(t)})} \tag{3.8}$$

$$= 1 \tag{3.9}$$

Hence

$$\|v_i\| \leq |\mathcal{N}_i| \tag{3.10}$$

Geometric meaning of control law:

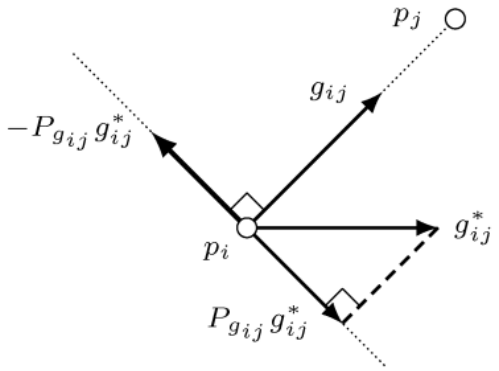


Figure 3.3: geometric illustration of control law. [11]

From figure-3.3 it can be seen that the control law attempts to move agent-i perpendicular to the current bearing vector, which ensures the bearing error is reduced and also the motion is tangential, as if the current edge is rotating in a circle with center being it's midpoint, so it maintains distance between agents as well as the centroid. Since bearing rigidity still can be ambiguous in case of translation and scaling, the law in (3.6) ensures that the scale and centroid of the formation is maintained constant throughout, so this means the scale and centroid of target formation is fixed based on initial positions which removes ambiguity. The same can be mathematically shown (for full details readers can see [11]).

Let H be the incidence matrix in of the underlying graph. $\bar{H} = H \otimes I_d$. Then if $g_k = \frac{e_k}{\|e_k\|}$,$k \in \{1, ..., m\}$

$$\frac{\partial g_k}{\partial e_k} = \frac{1}{\|e_k\|}(I_d - \frac{e_k}{\|e_k\|}\frac{e_k^T}{\|e_k\|}) = \frac{1}{\|e_k\|}P_{g_k} \tag{3.11}$$

using (3.3) and (3.4),

$$R(p) = \frac{\partial F_B(p)}{\partial p} = \frac{\partial F_{B(p)}}{\partial e}\frac{\partial e}{\partial p} = diag(\frac{P_{g_k}}{\|e_k\|})\bar{H} \tag{3.12}$$

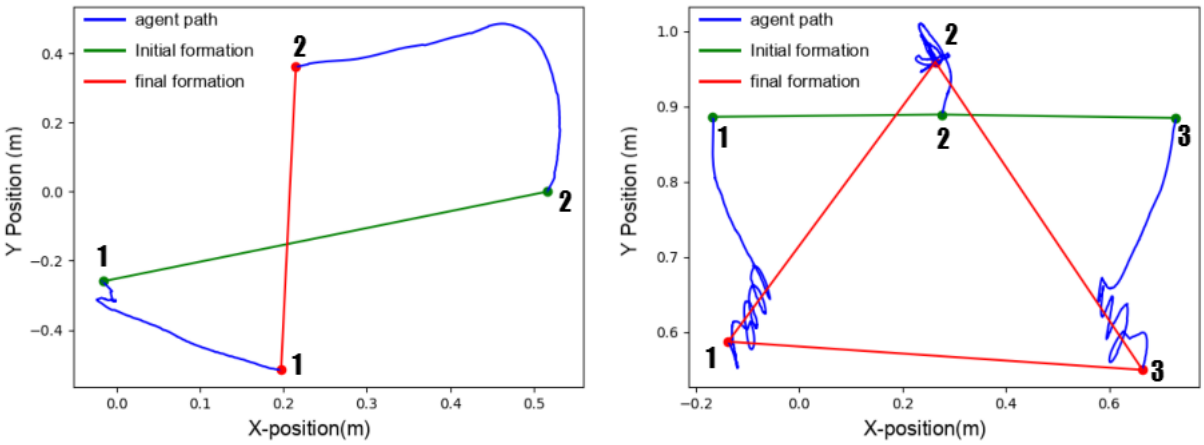
The control law in (3.6) can be written in compact form for all agents as:

$$v = \bar{H}^T diag(P_{g_k})g^* = \tilde{R}^T g^* \tag{3.13}$$

This means $v = \dot{p}$ lies in $range(\tilde{R}^T)$ and $range(\tilde{R}^T) \perp null(\tilde{R})$, thus $\dot{p} \perp null(\tilde{R})$. Also by (3.12), $null(\tilde{R}) = null(R(p))$ and by 1, $Null(R(p)) =< \mathbf{1} \otimes I_d, p >$, $\dot{p} \perp < \mathbf{1} \otimes I_d, p >$
Then centroid can be denoted as $\bar{p} = (\mathbf{1} \otimes I_d)^T p/n$.
Hence $\dot{\bar{p}} = (\mathbf{1} \otimes I_d)^T \dot{p}/n = 0$. This shows in-variance of centroid under the control law.

3.3 Implementation on crazyflie

In this implementation the bearing information was calculated by simply logging the position information of neighbor crazyflies and using the equation (3.1). The velocity control input given by (3.6) was sent to crazyflie onboard MCU via crazyradio.



(a) formation-1 (b) formation-2

Figure 3.4: Different formations

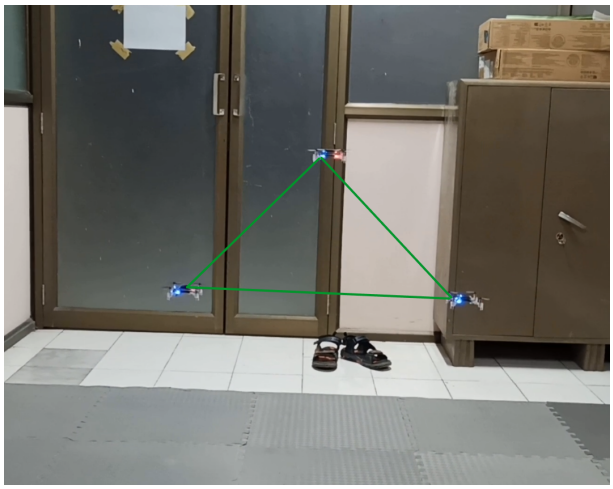


Figure 3.5: crazyflies in triangle formation

2 different formations were realized as shown above. The figure 3.4a had target bearings: $g_{12}^* = -g_{21}^* = [0, 1, 0]^T$ and figure 3.4b had target bearings:$g_{12}^* = -g_{21}^* = [\frac{1}{2}, \frac{\sqrt{3}}{2}, 0]^T, g_{23}^* = -g_{32}^* = [\frac{1}{2}, \frac{-\sqrt{3}}{2}, 0]^T, g_{13}^* = -g_{31}^* = [1, 0, 0]^T$. It can be seen from the plots that the crazyflies have almost converged to the desired shape. Figure 3.5 shows a snap from the real-time implementation.

Chapter 4

Containment Control

Consider a set of agents called *leaders* that move across the space following a predefined trajectory maintaining a formation shape. Now, consider a second set of agents called *followers*. The aim of containment control here is to design a control input for the follower agents such that they always remain within the convex hull formed by the leader agents.

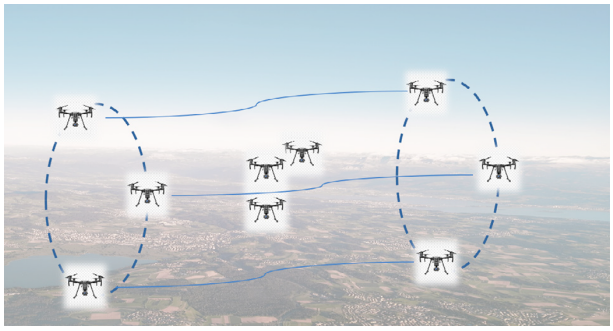


Figure 4.1: follower drones inside convex hull of leaders [1]

A swarm of leader drones/agents traverse a path in 3D space. This path is unknown to followers. The containment control problem defined here requires leaders to follow this path while the followers remain inside the convex hull of leaders. Followers only have access to relative position information of their neighboring leaders. It can be assumed there is a directed edge from a leader agent to a follower, signified the follower can sense the leaders position.

This problem can also be seen as a mix of formation control ans consensus. The leaders moving in a predefined trajectory means they are require to maintain a desired formation, while the requirement of followers being within convex hull can be seen as followers contained within a region, or, a loose consensus among followers since they ate required to converge to a region Rather than a point.

4.1 Containment control theory

Consider a setting where there are N leaders and M followers, with leaders being numbered first from $\{1, 2, ..., N\}$ and then the followers $\{N+1, N+2, ..., M+N\}$. Containment control problem can be formally defined as:

Definition(Containment Control):
Let $\{x_1, ...x_N\}$ be N-leaders position. Define convex hull of leaders at time t as $co_L(t) = \sum_i \sigma_i x_i(t) | \sum_i \sigma_i = 1$. Then containment control is solved if all positions of followers x_f converge into convex hull spanned by leader positions. Meaning,

$$\lim_{t\to\infty} dis(x_f(t), co_L(t)) = 0, i \in \{N+1, .., M+N\} \tag{4.1}$$

Theorem 2. *If $P(x)$ is a polynomial with complex coefficients, all zeros of derivative polynomial $P'(x)$ belong to the convex hull of the set of zeros of P.*

Meaning, consider a polynomial:
$P(z) = \prod(z - a_i)$, with $a_1, ...a_n$ complex numbers. Then for any root z of P', we have: $z = \sum_i \alpha_i a_i | \sum_i \alpha_i = 1$.
Theorem 2 is called **Gauss-Lucas** Theorem. It is this theorem that helps in developing a control law for followers so that they stay within convex hull of leaders.
Wikipedia provides a simple proof of this theorem: Let,

$$P(z) = \prod_{i=1}^{n} (z - a_i) \tag{4.2}$$

be a polynomial with complex roots $a_1, .., a_n$. Then for any root z of the derivative polynomial P', we have $P'(z) = 0$. Consider,

$$\frac{\mathrm{d}}{\mathrm{d}z} ln(P(z)) = \frac{P'(z)}{P(z)} = 0 \tag{4.3}$$

$$\frac{\mathrm{d}}{\mathrm{d}z} ln(P(z)) = \frac{\mathrm{d}}{\mathrm{d}z} ln \prod_{i=1}^{n} (z - a_i) \tag{4.4}$$

$$= \sum_{1}^{n} \frac{\mathrm{d}}{\mathrm{d}z} ln(z - a_i) \tag{4.5}$$

$$= \sum_{1}^{n} \frac{1}{z - a_i} \tag{4.6}$$

$$= \sum_{1}^{n} \frac{\bar{z} - \bar{a_i}}{|z - a_i|^2} = 0 \tag{4.7}$$

Hence,

$$\sum_{1}^{n} \frac{\bar{z}}{|z - a_i|^2} = \sum_{1}^{n} \frac{\bar{a_i}}{|z - a_i|^2} \tag{4.8}$$

Taking conjugate both sides and rearranging we get,

$$z = \frac{\sum_1^n \frac{a_i}{|z-a_i|^2}}{\sum_1^n \frac{1}{|z-a_i|^2}} \tag{4.9}$$

Let $\alpha_i = \frac{\frac{1}{|z-a_i|^2}}{\sum_1^n \frac{1}{|z-a_i|^2}}$, $z = \sum_1^n \alpha_i a_i$. Since $\sum_1^n \alpha_i = 1$, z, which is root of P', is a convex combination of roots of P.

For containment control, if we restrict ourselves to 2D, any position of leader agents can be denoted as a complex number: $p_l = x_l + iy_l$. Then, for a system of N leaders, we have a $N - degree$ polynomial P with complex roots as leader positions. The derivative polynomial P' having N-1 degree will gives is $N - 1$ roots lying withing convex hull of roots of P which are nothing but leader positions. Hence if followers are instructed to follow these $N - 1$ roots the containment control problem is solved. After placing $N - 1$ followers inside the leaders hull, we can consider these combined system of $2N - 1$ agents and treat them as leaders and again, repeating the same procedure, place $2N - 2$ followers inside again, and can keep repeating this process.

Such a process, despite making sense, has a major issue. Recall, each follower to needs to sense their neighbor's position. Meaning, the first set of $N - 1$ followers needs to establish a connection with the N leaders so that they sense their positions and apply theorem 2 to find the a feasible location inside the convex hull. The second set of agents, i.e the $2N - 2$ agent needs to establish connection with previously places $2N - 1$ agents and son on. This is a heavy requirement for each agent and may not be practically feasible.

To solve this issue, the method can be tweaked as follows.

- Start with N leaders in some shaope (polygon)
- divide this N-polygon into $N-2$ non-overlapping triangle.
- For each of these triangles 2 unique followers can be places inside them
- Within each triangle there now exist 2 agents, and collectively 5 agents in total. For any triangle, with 2 points indie, the triangle can further be divided into 5 new non-overlapping triangles.
- repeat the process.

This way each agent needs to establish connection with only 3 previously placed agents. Consider a example of where we start with 3 leaders initially. There is one triangular region. So 2 unique followers can be placed. With these 2 agents inside triangle , the triangle can be further divided into 5 new non-overlapping triangles and 10 more followers can be placed inside them. (see figure 4.2). Initially 3 "blue" agents form the triangle. Then 2 "red" followers are placed within. Next, the triangle, using these 2 "red" followers is divided into 5 non-overlapping triangles and 10 new "green" followers placed within.

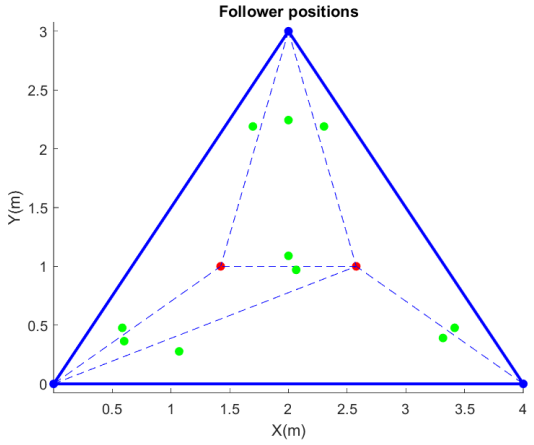


Figure 4.2: agents placed inside a triangle

One might also note that simply calculating centroid of the N-polygon will give us a location within the convex hull. But note that it gives only one location as opposed to N-1 locations in this implementation. Further, the stress on non-overlapping triangles is important. Because if they are overlapping , 2 follower locations can become arbitrarily close or even coincide. By keeping them non-overlapping, the theorem 2 ensures that 2

locations of followers never coincide, since they will always be contained within that triangle.

The follower agent must sense their neighbor's position, use theorem 2 to find new positions ro follow. But since eth leaders are moving, how does these follower location vary with leader movement? In other words, if the polynomial P is formed by the locations of the 3 leaders forming a triangle, then, how does the roots of P' vary if these 3 leaders move a bit. Because, the coefficients of derivative polynomial depend on these locations and hence the roots. This is crucial, because if the leaders are slightly perturbed, or the derivative polynomial coefficients slightly perturbed, and the resulting 2 roots vary drastically(though still within the triangle), it might be tough for follower agents to tightly track these locations.

Let z_1, z_2, z_3 be leader locations. Then $P(z) = (z - z_1)(z - z_2)(z - z_3)$,

$$P'(z) = az^2 + bz + c \tag{4.10}$$

where $a = 3, b = -2(z_1 + z_2 + z_3), c = z_1z_2 + z_2z_3 + z_3z_1$.

The coefficients of P' are smooth and continuous functions of leader locations. Now if leader move smoothly and continuously, the coefficient of P' will also change smoothly and continuously. Further by implicit function theorem [10], for any polynomial, it's root vary smoothly and continuously with it's coefficient. *Thus followers locations vary smoothly and continuously, as long as leaders move in a smooth and continuous manner*
Figure 4.3 provides a confirmation for the same, where the leaders (green trajectory) move smoothly and follower locations (in red) also vary similarly.

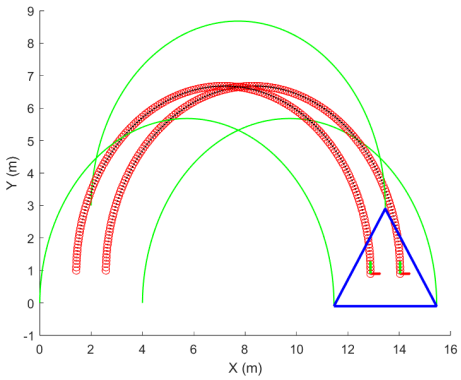


Figure 4.3: variance of derivative polynomial roots with original polynomial roots

4.2 Implementation on crazyflie 2.1

As discussed earlier, the containment control algorithm can iteratively keep on placing as many followers as possible. However due to practical constraints like space issue on flying arena, level-1 containment was implemented. Meaning, starting with N-leaders, the N-polygon was divided into N-2 non-overlapping triangles and 2 followers were placed inside each. The process was terminated here.

The steps involved in practical implementation were:

1. Start with leader drones kept such that they form some convex polygon
2. The follower drones are initially uniformly spread out outside this convex polygon.
3. The central PC takes in the positions of leader drones and finds N-2 non-overlapping triangles
4. After this each follower was assigned to the triangle that is closest to it.(not more than 2 per triangle).
5. This means that follower now has the leader drones forming the vertices of that closest triangle as it's neighbors.
6. Then based on the main containment control algorithm, for each follower, based on it's neighbor locations, 2 roots were calculated and the follower was commanded to go to the root nearest to it.
7. This will act as a position set-point which will be sent to crazyflie onboard MCU via crazyradio.

4.2.1 Finding non-overlapping triangle

Leader drones were placed in a desired convex formation and follower drones were placed outside the formation. The locations of these leaders were be logged into the PC using crazyradio. To find the non-overlapping triangles, it is necessary to know the sequence in which these leaders are arranged. Consider the figure-4.4. It can be seen that the order of leaders are $1, 2, 3, 4, 5$ in clockwise direction. Then the non-overlapping triangles can be found as $tri = \{(1, 2, 3), (1, 3, 4), (1, 4, 5)\}$. The leader locations that are logged into the PC need not be in any such fixed order, it can be random. Hence it is mandatory to find the sequence of leaders either clockwise or counter-clockwise. For this a **ray sweep algorithm** was used.

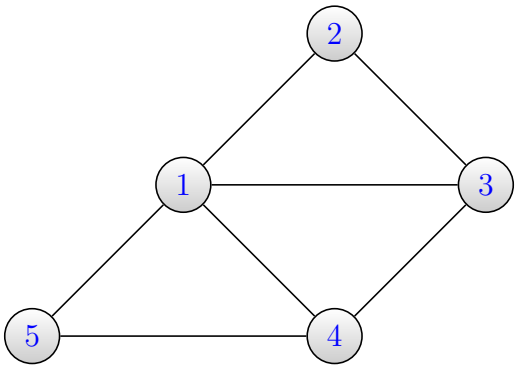


Figure 4.4: non-overlapping triangles in pentagon

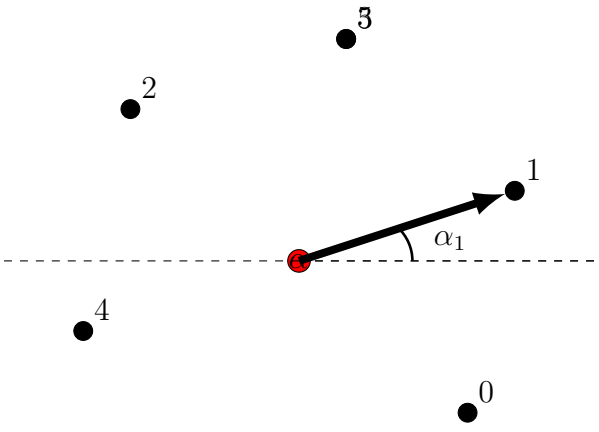


Figure 4.5: Diagram illustrating ray sweep

Suppose there are 5 leaders placed as a pentagon. The PC can log the positions of these leaders but to determine their sequence, ray sweep is used. As shown in figure 4.5, using the leader positions, a centroid is calculated. A ray originating at centroid sweeps the polygon's plane and records the

angle it makes at each vertex with the horizontal. These angles are then sorted and the vertices are arranged in sequence according to them. This was a sorted list of leaders is obtained.

Once the sequence of leaders was found, non-overlapping triangles were determined.

Algorithm 1: Finding non-overlapping triangle

$n \leftarrow$ no. of leaders;
$L \leftarrow$ Sorted list of leaders;
$Tri \leftarrow$ List containing non-overlapping triangles;
$i \leftarrow 1$;
while $i \leq n - 2$ **do**
    $Tri \leftarrow Tri \cup \{L(1), L(i+1), L(i+2)\}$;
    $i \leftarrow i + 1$;
end

4.2.2 Assigning followers to triangles

The first step here is to determine how many followers will be placed inside each triangle based on number of available followers. Since each triangle can accommodate a maximum of 2 followers,it is required that no. of followers $M \leq 2(N-2)$. If $M = 2(N-2)$ then the problem is trivial. If $M < 2(N-2)$, initially it is assumed every triangle has 2 followers each. Then iterating over all triangles in a cyclic manner, for each triangle one follower is reduced. This happens until total number of followers now inside the all the triangles in equal to M. The same is shown in algorithm-2.

Algorithm 2: Determine no. of followers per triangle

$n \leftarrow$ no. of leaders;
$m \leftarrow$ no. of followers;
$Tri \leftarrow$ List containing non-overlapping triangles;
$numfollpertri \leftarrow [2] * |Tri|$;
$k \leftarrow 2(n-2)$;
$i \leftarrow 1$;
while $k! = m$ **do**
    $numfollpertri[i] \leftarrow numfollpertri[i] - 1$;
    $i \leftarrow i + 1$;
    $i \leftarrow mod(i, n-2)$;
    $k \leftarrow k - 1$;
end

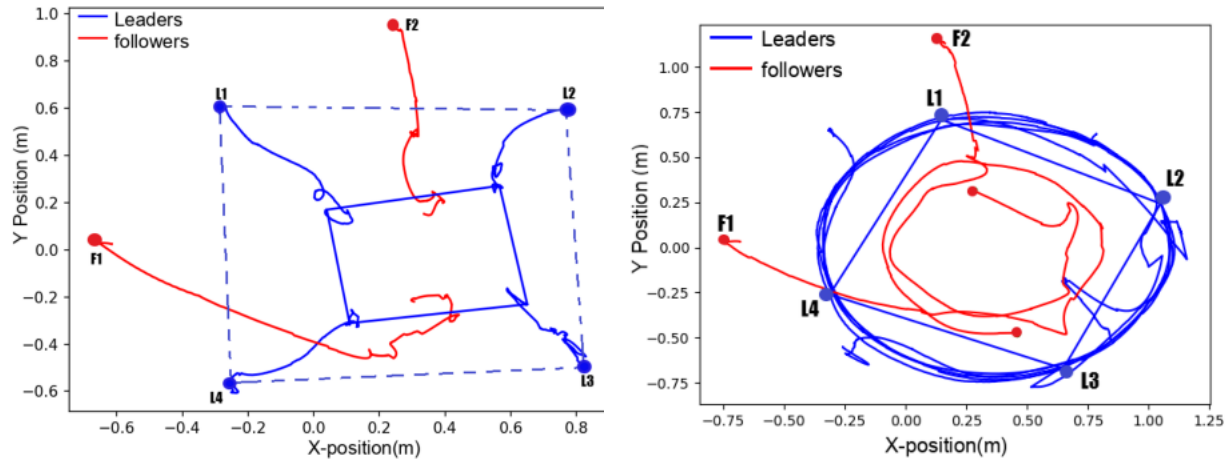
After this each follower was assigned a particular triangle. The PC logged positions of all agents initially. Using this data, for each follower,it's distance with each of the $N-2$ triangle's centroids was calculated and the triangle closest to it was assigned to it. Now that particular follower has the leaders forming that triangle as it's neighbors.

4.2.3 Main Control

After assigning followers to the triangles, for each follower, using theorem-2, the roots were calculated based on it's neighbor's positions. Among the 2 roots, the one closer to it's current location was chosen. This acted as the position set-point for the crazyflie, which was tracked using the onboard MCU's cascaded PID controller.

4.2.4 Implementation Results

In actual implementation, 4 leaders were used and 2 followers. The below images show the results obtained.



(a) Leader scaling down

(b) Leaders circling

Figure 4.6: Containment of followers

Figure-4.6a shows a scenario where the leader formation was commanded to scale down in size. It can be seen that the followers also sort of scale down, as in, converge inside the smaller scaled down formation. Figure-4.6b shows a scenario where leader formation was commanded to rotate/-circle. It can be seen that followers also circle with leaders while being inside the convex hull of leader. (The red circle trajectory lies within blue

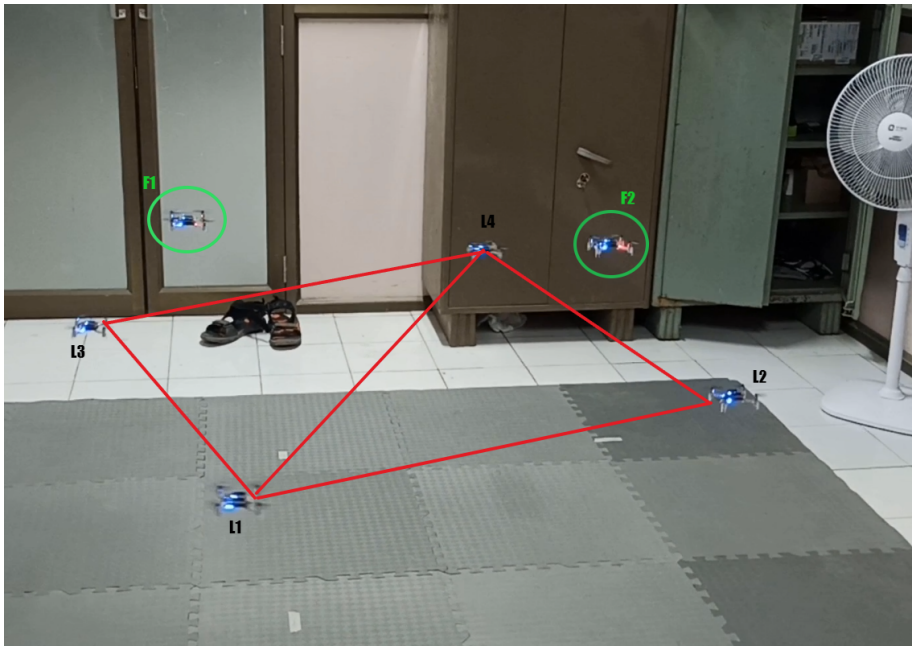


Figure 4.7: Snapshot of containment of followers using crazyflies

circle trajectory). Figure-4.7 shows a snapshot from the real-time implementation. The followers marked with green circle are contained inside the leader convex hull (red square). Follower-1 is assigned the triangle (L1,L3,L4) and follower-2 is assigned (L1,L2,L4).

Chapter 5

Formation Control of double integrators

5.1 Bearing-only Formation Control of double integrators

This control law was developed by: Susmitha T Rayabagi,PhD student (Dept of Electrical Engineering, IIT Bombay), Prof Dwaipayan mukherjee (Dept of Electrical Engineering, IIT Bombay), Prof Debasattam Pal (Dept of Electrical Engineering, IIT Bombay)
The control law is developed for formation tracking using bearing-only information , with agents modelled as double-inetgrators. It uses a leader-follower structure, where a leaders agent moves with a constant velocity. The agents must converge to a target formation and keep moving with the leader while maintaining the formation, thus formation tracking. The underlying graph here is directed and the graph is realized by *Henneberg Construction.*

5.1.1 Henneberg Construction

Henneberg construction is a way of constructing bearing rigid directed graphs. The construction here results in a Leader-first-follower structure(LFF), meaning, there is one leader. All other followers except the first-followers have precisely 2 neighbors wit first-follower having one neighbor, the leader.

The construction starts with 2 vertices v_1, the leader and v_2, the first follower and a directed edge (v_2, v_1) as shown in step-1 of figure-5.1. The subsequent steps perofrm 2 operations: *vertex addition* and *edge splitting*

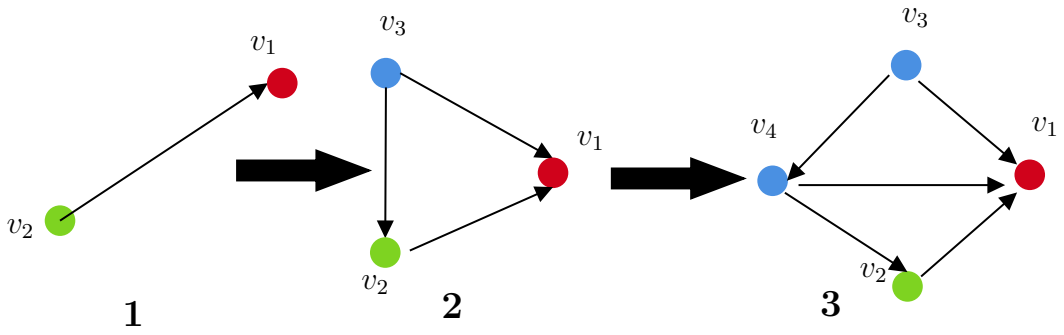


Figure 5.1: Henneberg Construction

- *Vertex Addition:* Add vertex v_k to graph with 2 new directed edges (v_k, v_i) and (v_k, v_j) to the existing v_i, v_j. Step-2 of figure-5.1 shows this where v_3 is added with 2 new directed edge $v_3, v_2)$ and (v_3, v_1)
- *Edge Split:* A vertex v_i with 2 neighbors is chosen and one of it's edges removed. The new vertex v_k is added with 3 directed edges. Step-3 of figure-5.1 shows this. Existing edge (v_3, v_2) is removed.v_4 is added with 3 new edged $(v_4, v_1), (v_4, v_2)$ and (v_3, v_4).

Every vertex except v_2(first-follower) has 2 neighbors.

5.1.2 Problem formulation

Consider n agents in $\mathbb{R}^2$. Let $p_i \in \mathbb{R}^2$ denote position of i^{th} agent, $v_i, u_i \in \mathbb{R}^2$ be the velocity and acceleration. For i^{th} agent teh dynaimcs are given by:

$$\dot{p}_i = v_i, \quad \dot{v}_i = u_i \tag{5.1}$$

Each agent is required to measure relative bearing and bearing rates. The bearing is defined similarly as in (3.1), and projection operator is defined as in (3.2)
Bearing rate is denoted as $\dot{g}_{ij}$

$$\dot{g}_{ij} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{p_j - p_i}{\|p_j - p_i\|} \tag{5.2}$$

$$= \frac{\dot{p}_j - \dot{p}_i}{\|p_j - p_i\|} - g_{ij}\frac{(\dot{p}_j - \dot{p}_i)^T}{\|p_j - p_i\|}g_{ij} \tag{5.3}$$

Hence,

$$\dot{g}_{ij} = \frac{P_{g_{ij}}}{\|e_{ij}\|}(\dot{p}_j - \dot{p}_i) \tag{5.4}$$

The bearing rate carries some information on relative velocities. Since this problem is not only formation control but also formation tracking, the convergence to a specific formation mainly deals with convergence of positions of agents to desired positions, but for tracking, the relative velocities also needs to follow some desired trajectory which is implicitly imposed by the control law,and hence measuring bearing rates is required in this setting.

It should be noted that the graph constructed from henneberg construction in figure-5.1, one of the follower, namely the first follower, v_2 has only one neighbor. This gives v_2 an additional degree of freedom to translate along the vector $v_2 - v_1$ under any control law while still maintaining bearing. This results in scaling of the formation. To prevent this the following assumption is made: Leader carries a imaginary circular disk attached to it of fixed radius.

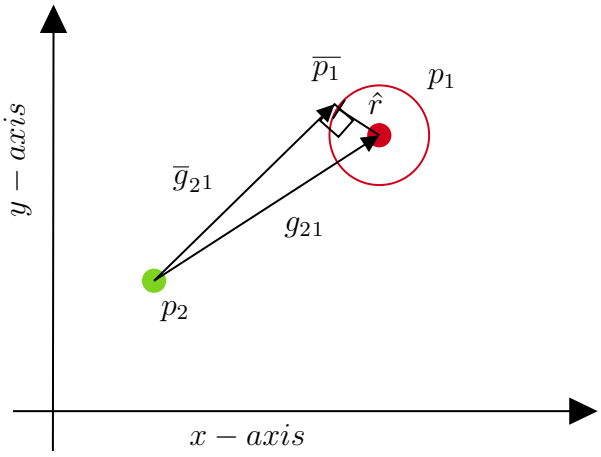


Figure 5.2: virtual follower for first follower

See figure-5.2. The leader p_1 carries a imaginary disk of radius $\hat{r}$. First follower p_2 is required to measure a bearing $\bar{g}_{21}$, which is the relative bearing between first follower and it's pseudo-neighbor $\bar{p}_1$ located along the tangent from p_2 to the leader's circular disk.

As mentioned earlier, presence of only one neighbor for first-follower allows first follower to translate along $p_2 - p_1$ edge while still maintaining

the bearing constraint. So the distance d_{21} is no fixed. But if, in addition to specifying target bearing g^*_{21}, the bearing $\bar{g}^*_{21}$ is also fixed in target formation, it effectively fixes the distance d_{21}. Consider the triangle $p_1p_2\bar{p}_1$ in figure-5.2. Let angle between the sides p_2p_1 and $p_2\bar{p}_1$ be α, then clearly

$$\|p_2 - p_1\| = d_{21} = \frac{\hat{r}}{sin\alpha} \tag{5.5}$$

If g^*_{21} and $\bar{g}^*_{21}$ are fixed along with $\hat{r}$, then α is also fixed, which in-turn fixes d_{21}

The control input must ensure all the target bearings specifying the target formation is achieved asymptotically and the followers translate with the velocity equal to the leader velocity.

Control law:

$$\dot{v}_i = -k_p \sum_{j \in \mathcal{N}_i} P_{g_{ij}} g^*_{ij} + k_v \sum_{j \in \mathcal{N}_i} \dot{g}_{ij} \tag{5.6}$$

Challenge faced here was, equation-(5.6) gives an acceleration input for quad-copter. Designing a controller that accurately tracks acceleration input was challenging since it was not pre-implemented in crazyflie's onboard controller. Two different approaches were used to design a controller, but both of them could not accurately track an acceleration input. The formation control task was implemented using approach-1 but results obtained were not satisfactory. The work done in developing the controller using 2 approaches is discussed in the upcoming sections.

5.2 Acceleration Controller: Approach-1

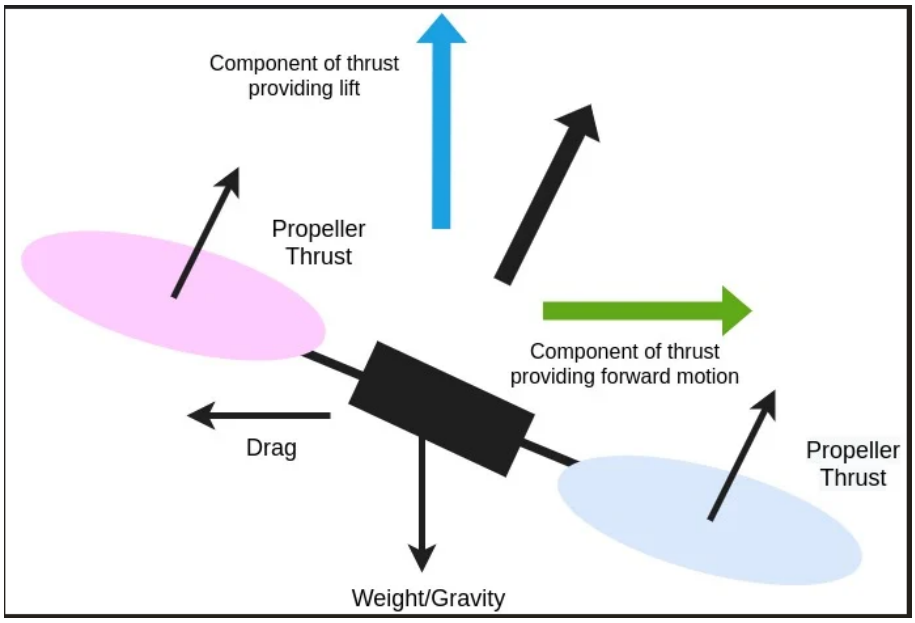


Figure 5.3: Quadcopter thrust producing lateral movement when tilted [4]

The controller developed in this section is sort-of an open-loop controller, meaning there is not feedback of current acceleration taken into account to generate control signals.

As shown in fig-5.3, if a quadcopter tilt by some angle, the thrust along it's body $z - axis$ has a component along forward direction, meaning it generates a force along thss direction, or an acceleration occurs in this direction.

The *roll* is defined as the rotation of quadcopter about it's body x-axis and *pitch* is the rotation about body y-axis. By doing an appropriate roll and pitch maneuver, a component of body z-axis thrust appears in the XY-plane hence leading to an acceleration in XY-plane. Note that, in this controller, only XY-acceleration was controller and the resulting control input to quadcopter were only the roll and pitch angles that would tilt the quadcopter such that the resulting component of thrust in XY-plane is precisely the commanded acceleration.

Let commanded acceleration in inertial frame (IF) be a_c=[a_x,a_y,a_z]. Let the rotation matrix to go from body frame(BF) to inertial frame(IF) be:

$$R_{IF}^{BF} = \begin{bmatrix} cos\theta cos\psi & cos\theta sin\psi & -sin\theta \\ cos\psi sin\theta sin\phi - sin\psi cos\phi & sin\psi sin\theta sin\phi + cos\psi cos\phi & sin\phi cos\theta \\ cos\psi sin\theta cos\phi + sin\psi sin\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi & cos\phi cos\theta \end{bmatrix}$$

At level condition when quadcopter is hovering, it generates a body-z specific thrust of [0,0,g], where g is acceleration due to gravity. Then the required roll and pitch angles or generate a_c is found by:

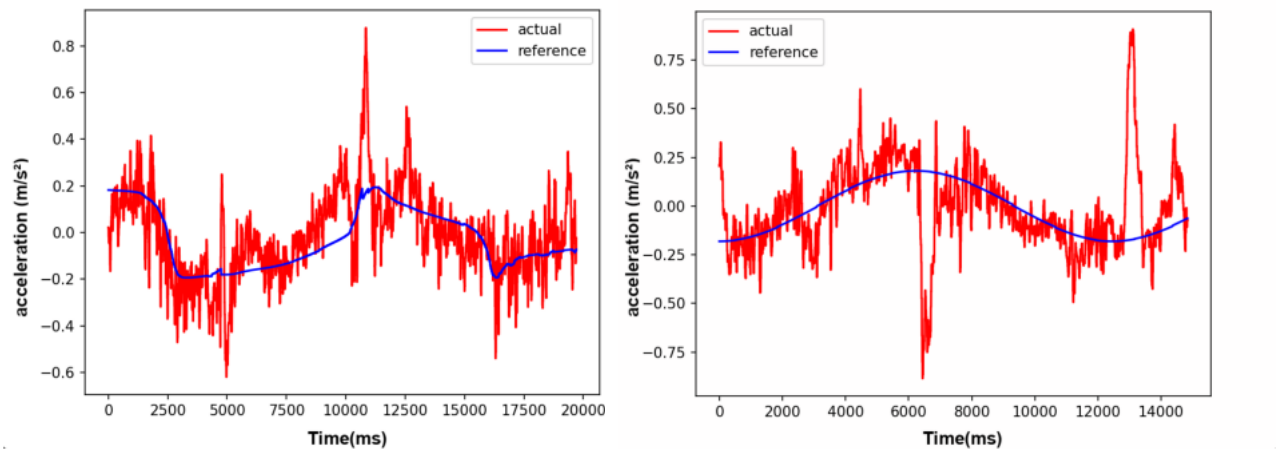
$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = R_{IF}^{BF} \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} \tag{5.7}$$

$$\theta = sin^{-1}[\frac{a_x}{g}] \quad \phi = -sin^{-1}[\frac{a_y}{gcos\theta}] \tag{5.8}$$

5.2.1 Implementation on crazyflie

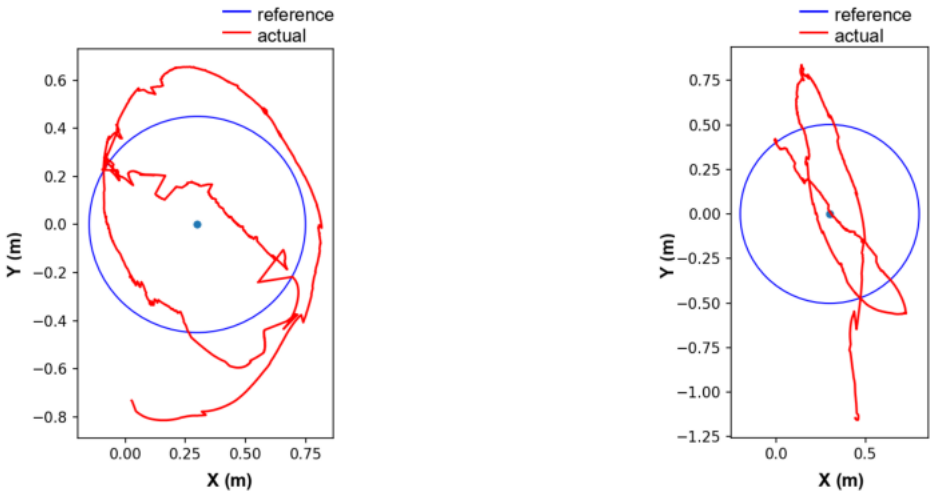
The calculated roll and pitch angles were sent to onboard PID controller of crazyflie. To test the efficacy of this controller, some reference acceleration commands were given such that quadcopetr would move in a circle.

Figure-5.4 shows the results obtained for tracking of acceleration command by quadcopter. Figure-5.5 shows the corresponding path followed by quadcopter vs the actual circular path it should have followed as per the reference acceleration command. It can be seen that there are a lot of noisy components in quadcopter acceleration. This prevents the quadcopter from tightly tracking the reference, and since position is double integration of acceleration, even a small error is acceleration tracking can get accumulated to a large error in position tracking, an hence the path so highly deviated from the desired circle.



(a) (b)

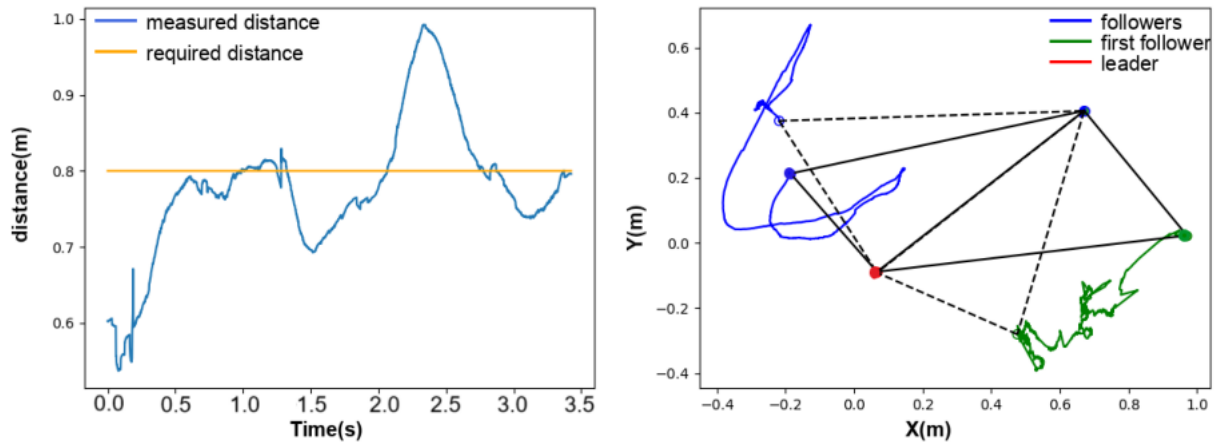
Figure 5.4: measured vs commanded acceleration



(a) (b)

Figure 5.5: Path taken by quadcopter

This controller was used to implemenet the double-integrtor formation control law.



(a) evolution of d_{21} (b) formation evolution

Figure 5.6: double integrator formation control implemeneted using approach-1

In figure-5.6 The desired formation was supposed to be a square. The dashed lines denote initial formation and solid lines are final formation. The "green" trajectory is that of first follower and the target bearing that it was supposed to have with leader "red" was $[1, 0, 0]$. Clearly the desired formation was not achieved properly and also the distance d_{21} was fixed at 0.8 meters. From the figure it is evident the controller designed by approach-1 did not give satisfactory results.

Since quadcopter is subject to small vibrations, lots of disturbance acts on it causing It to improperly track acceleration. Hence a constant correcting mechanism was needed where at each instant current acceleration can be fed-back an based on the Error new control input can be calculated to reduce the error. This approach is explained in next section.

5.3 Acceleration Controller: Approach-2

The controller implmenetd in this section is a closed-loop controller. It was inspired by the controllers developed by the authors in [7] and [9].

The idea here is to use **Nonlinear Dynamic Inversion (NDI)**. Using NDI, highly nonlinear systems can be controlled as if they were a linear system. It effectively cancels out the non-linearities in the system to be tracked by including them in the control law. NDI is popular in aerospace literature since things like aeroplane,quadcopters have highly non-linear dynamics.

A simple general-case example of NDI is presented below.
Consider an affine nonlinear system, where the word affine means the state dynamics are linear in input.

$$\begin{aligned} \dot{X} &= f(X) + [g(X)]u \\ Y &= h(X) \end{aligned} \tag{5.9}$$

$X \in \mathbb{R}^n, u \in \mathbb{R}^m, Y \in \mathbb{R}^p$.
f and h are nonlinear functions.$[g_Y(X)]$ is a matrix. u is the control input to be designed using NDI and Y is the variable to be tracked. Y will b referred to as output which has to be driven to a desired set-point by designing u. Driving output dynamics:

$$\begin{aligned} \dot{Y} &= \frac{\partial h}{\partial X}\dot{X} \\ \implies \dot{Y} &= \frac{\partial h}{\partial X}[f(X) + [g(X)]u] \\ \implies \dot{Y} &= f_Y(X) + [g_Y(X)]u \end{aligned} \tag{5.10}$$

where $f_Y(X) = \frac{\partial h}{\partial X} f(X)$ and $[g_Y(X)] = \frac{\partial h}{\partial X}[g(X)]$ is a non-singular matrix.
Suppose the output Y needs to be driven to Y^*. Define error $E(t) = Y(t) - Y*(t)$. The input u is designed such that the eror dynamics behave linearly. By considering 1^{st} order linear dynamics,

$$\dot{E} + KE = 0 \tag{5.11}$$

K is chosen such that error quickly converges to 0. Expanding the above

$$\begin{aligned} (\dot{Y} - \dot{Y}^*) + K(Y - Y^*) &= 0 \\ f_Y(X) + [g_Y(X)]u &= \dot{Y}^* - K(Y - Y^*) \end{aligned} \tag{5.12}$$

Then the controller u is found as:

$$u = [g_Y(X)^{-1}[\dot{Y}^* - K(Y - Y^*) - f_Y(X)] \tag{5.13}$$

Note that substituting "u" in output, Y dynamics cancels out nonlinear terms and changes the dynamics to linear one.

The example shown above is NDI with first-order error dynamics. To design an acceleration controller for crazyflie, NDI control law using first-order error-dynamics was developed first. That did not give satisfactory simulation results and hence, finally, NDI control law using second-order error-dynamics was developed which gave better simulation results.

5.3.1 NDI controller using 1^{st}-order error dynamics

The translational model of quadcopter is given using second-order dynamics:

$$\begin{aligned} \dot{p} &= v \\ \dot{v} &= g i_z + \frac{T}{m} b_z + m^{-1} f_{ext} \end{aligned} \tag{5.14}$$

where
p=position,
v=velocity,
g=acceleration due to gravity($= 9.8 m/s^2$),
m=mass of crazyflie,
T = thrust magnitude along body z-axis,
b_z=body z-axis expressed in inertial frame(IF) and
f_{ext}=is the external force acting on quadcopter.

Let the rotation matrix needed to go from inertial frame to body frame be denoted as:

$$R = \begin{bmatrix} cos\theta cos\psi & cos\theta sin\psi & -sin\theta \\ cos\psi sin\theta sin\phi - sin\psi cos\phi & sin\psi sin\theta sin\phi + cos\psi cos\phi & sin\phi cos\theta \\ cos\psi sin\theta cos\phi + sin\psi sin\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi & cos\phi cos\theta \end{bmatrix}$$

Then

$$\frac{T}{m} b_z = \frac{T}{m} R^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \frac{T}{m} \begin{bmatrix} cos\psi sin\theta cos\phi + sin\psi sin\phi \\ sin\psi sin\theta cos\phi - cos\psi sin\phi \\ cos\phi cos\theta \end{bmatrix} \tag{5.15}$$

where ϕ, θ, ψ are roll,pitch and yaw. Clearly $\frac{T}{m} b_z = f(\phi, \theta, \psi, T)$
The quantity to be controlled here is the acceleration denoted as $a = \dot{v}$. The control law must drive a to desired a^*.(assume desired acceleration as constant set-point)

Defining error as $e = a - a^*$ and noting that gravity is always constant and variation of external force with time is tough to estimate and hence assuming it's derivative as 0, using equation-5.14 error dynamics can be defined as:

$$\dot{e} = \dot{a} = \ddot{v} = \frac{d}{dt} \frac{T}{m} b_z = \frac{d}{dt} f(\phi, \theta, \psi, T) \tag{5.16}$$

Assuming yaw is kept constant, the function f is now dependent on 3 variables ϕ, θ, T. Differentiating it,

$$\dot{e} = \frac{T}{m} \begin{bmatrix} (c\phi s\psi - s\phi c\psi s\theta) & (c\phi c\psi c\theta) & s\phi s\psi + c\phi c\psi s\theta \\ (-s\phi s\psi\theta - c\psi c\phi) & c\phi s\psi c\theta & c\phi s\psi s\theta - c\psi s\phi \\ -s\phi c\theta & -c\phi s\theta & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{T} \end{bmatrix} \tag{5.17}$$

Let,

$$G = \begin{bmatrix} T(c\phi s\psi - s\phi c\psi s\theta) & T(c\phi c\psi c\theta) & s\phi s\psi + c\phi c\psi s\theta \\ T(-s\phi s\psi\theta - c\psi c\phi) & T(c\phi s\psi c\theta) & c\phi s\psi s\theta - c\psi s\phi \\ T(-s\phi c\theta) & T(-c\phi s\theta) & c\phi c\theta \end{bmatrix} \tag{5.18}$$

approximating $\dot{\phi} = \phi - \phi_0, \dot{} = \theta - \theta_0, \dot{T} = T - T_0$ in discrete setting, where ϕ_0, θ_0, T_0 are the values in previous time-step, and denoting $u = \begin{bmatrix} \phi \\ \theta \\ T \end{bmatrix}$

$$\dot{e} = \frac{1}{m} G(u - u_0) \tag{5.19}$$

using $\dot{e} + Ke = 0$,

$$\implies \frac{1}{m} G(u - u_0) + K(a - a^*) = 0 \tag{5.20}$$

putting $a = a_0$ as acceleration measured in previous time-step, the final control law is:

$$u = u_0 + mG^{-1}K(a^* - a_0) \tag{5.21}$$

Equation-(5.21) gives the control input to quadcopter that attempts to track a acceleration set-point. The inputs are roll,pitch and body-z thrust. *This control law was used in a simulation setup, the results of which are given in the section 5.3.3*

5.3.2 NDI controller using 2^{nd}-order error dynamics

Using the same transnational model of quadcopter as in equation(5.14) and defining error similarly as in previous subsection, the error dynamics are second-order here.

$\ddot{e} + K_d\dot{e} + K_p e = 0$, where $\dot{e}$ is defined the same as in equation-(5.19).

$$\ddot{e} = \frac{d}{dt}\dot{e} \tag{5.22}$$

$$\ddot{e} = \frac{T}{m}\begin{bmatrix}(-s\phi s\psi - c\phi c\psi s\theta) & (-c\phi c\psi s\theta) & 0\\ (-c\phi s\psi\theta + c\psi s\phi) & -c\phi s\psi s\theta & 0\\ -c\phi c\theta & -c\phi c\theta & 0\end{bmatrix}\begin{bmatrix}\dot{\phi}^2\\ \dot{\theta}^2\\ \dot{T}^2\end{bmatrix} + \frac{T}{m}\begin{bmatrix}(c\phi s\psi - s\phi c\psi s\theta) & (c\phi c\psi c\theta) & s\phi s\psi + c\phi c\psi s\theta\\ (-s\phi s\psi\theta - c\psi c\phi) & c\phi s\psi c\theta & c\phi s\psi s\theta - c\psi s\phi\\ -s\phi c\theta & -c\phi s\theta & c\phi c\theta\end{bmatrix}\begin{bmatrix}\ddot{\phi}\\ \ddot{\theta}\\ \ddot{T}\end{bmatrix} \tag{5.23}$$

Let

$$H = \begin{bmatrix}T(-s\phi s\psi - c\phi c\psi s\theta) & T(-c\phi c\psi s\theta) & 0\\ T(-c\phi s\psi\theta + c\psi s\phi) & T(-c\phi s\psi s\theta) & 0\\ -Tc\phi c\theta & -Tc\phi c\theta & 0\end{bmatrix} \tag{5.24}$$

$$u = \begin{bmatrix}\phi\\ \theta\\ T\end{bmatrix}, \quad v = \begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{T}\end{bmatrix} \tag{5.25}$$

G is defined as in eq-(5.18). Then

$$\ddot{e} = \frac{1}{m}[H(u - u_0)^2 + G(v - v_0)] \tag{5.26}$$

Assuming the rate of changes of ϕ, θ, T are small, $(u - u_0)^2$ 0 and using $\ddot{e} + K_d\dot{e} + K_p e = 0$, where $\dot{e}$ is given as in (5.19),

$$\frac{1}{m}G(v - v_0) + \frac{1}{m}K_d G(u - u_0) + K_p(a - a^*) = 0 \tag{5.27}$$

Equation-(5.27) is a linear under-determined system since there are 2 unknowns u, v and one equation. This can be solved by imposing a constraint to minimize the control input and use Lagrange multiplier method.

Denoting:

$$\begin{aligned} A &= \begin{bmatrix} \frac{1}{m}G & \frac{1}{m}K_d G \end{bmatrix} \\ X &= \begin{bmatrix} v - v_0 \\ u - u_0 \end{bmatrix} \\ b &= K_p(a^* - a_0) \end{aligned} \tag{5.28}$$

Equation-(5.27) can be reformulated as below constrained optimization problem:

$$\begin{aligned} minimize\ \ J &= \frac{1}{2}X^T R X \\ subject\ to\ \ AX &= b \end{aligned} \tag{5.29}$$

Where $R \in \mathbb{R}^{6\times 6}$ is the control input penalizing matrix.Using Lagrange multiplier method, the solution is:

$$\begin{aligned} X &= R^{-1}A^T(AA^T)^{-1}b \\ \begin{bmatrix} v \\ u \end{bmatrix} &= X + \begin{bmatrix} v_0 \\ u_0 \end{bmatrix} \end{aligned} \tag{5.30}$$

This controller gives roll,pitch,thrust,roll-Rate,pitch-Rate as input to quadcopter to track the reference acceleration. *This control law was used in a simulation setup, the results of which are given in the section 5.3.3*

5.3.3 Simulation

Simulation setup

The simulation was setup to mimic actual crazyflie behaviour as closely as possible. The crazyflie uses cascaded PID as shown in figure-2.5. The state variables were *position p,velocity v,orientation o and angular rates* Ω. The states were updated by the following translational and rotational dynamics:

$$\begin{aligned} \dot{p} &= v \\ \dot{v} &= g i_z + \frac{T}{m} b_z \end{aligned} \tag{5.31}$$

$$\begin{aligned} \dot{o} &= \Omega \\ \dot{\Omega} &= J^{-1}(\tau - \Omega \times J\Omega) \end{aligned} \tag{5.32}$$

Where τ is the torque and J is the moment of inertia matrix. The system identification data was taken from [5] and moment of inertia matrix has a value:

$$J = \begin{bmatrix} 16.571710 & 0.830806 & 0.718277 \\ 0.830806 & 16.655602 & 1.800197 \\ 0.718277 & 1.800197 & 29.261652 \end{bmatrix} \times 10^{-6} kgm^2 \tag{5.33}$$

$m = 0.027\ kg$ for crazyflie

The NDI control law derived in equation-(5.21) and equation-(5.30) gives attitude and attitude rates as inputs to quadcopter. Crazyflie uses a cascaded PID as shown in figure-2.5. For this purpose, attitude and attitude rate controller was implemented for simulation. The attitude and attitude rate set-points given by NDI control law was fed to this cascaded PID controller. The output of the attitude rate controller is used along with thrust set-point from NDI control law to decide the PWM given to each of the 4 motors on crazyflie. The PWM applied to each motor determines the thrust and torque of the quadcopter. Authors in [5] developed a quadratic mapping from PWM applied to each motor and resultant torque and thrust produced by them. This map was used to find the torque and thrust. The torque and thrust values are needed to update the states using

the dynamics mentioned in (5.31) and (5.32). The NDI control law derived in equation-(5.21) and equation-(5.30) uses attitude, attitude rates and thrust in previous time step to calculate the new control input. Hence, the after updating the states, the new roll,pitch,roll Rate,pitch Rate, and thrust mapped earlier was fed back to NDI controller which then generated the control input for next time step. The process can be summarized as follows:

1. NDI generates thrust(T_c),roll(ϕ_c),pitch(θ_c),roll-rate($\dot{\phi}_c$),pitch-rate($\dot{\theta}_c$) as control input to crazyflie.
2. ϕ_c and θ_c is given as set-points to attitude PID controller
3. attitude PID controller generates desired roll and pitch rates denoted as $\dot{\phi}_{cf}$ and $\dot{\theta}_{cf}$.
4. The commanded rates from NDI and the one from attitude PID controller are averaged and passed as new rate setpoints to attitude rate PID controller.
5. The rate PID controller generates output r, p for following desired roll and pitch rates.
6. The thrust setpoint from NDI, T_c is in newtons, which is mapped to a value between 0-65535 and then is used along with r, p to find PWMs for all 4 motors.
   - $m_1 = T_c - r + p$
   - $m_2 = T_c - r - p$
   - $m_3 = T_c + r - p$
   - $m_4 = T_c + r + p$
7. The above PWMs are mapped to thrust and torque which are then used to update the crazyflie dynamics.
8. from the updated states roll(ϕ_0),pitch(θ_0),roll-rate($\dot{\phi}_0$),pitch-rate($\dot{\theta}_0$), and the mapped thrust (T_0) are fed back to NDI which then calculates the control input for next time step and the process is repeated.

The figure-5.7 shows the block diagram of this process.

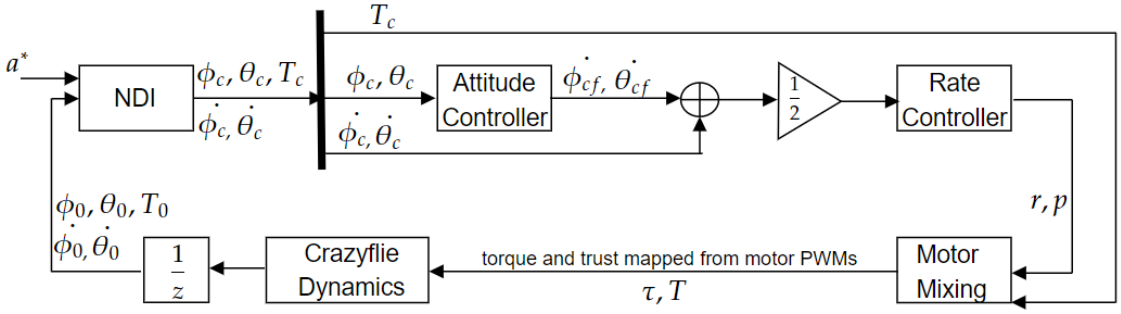
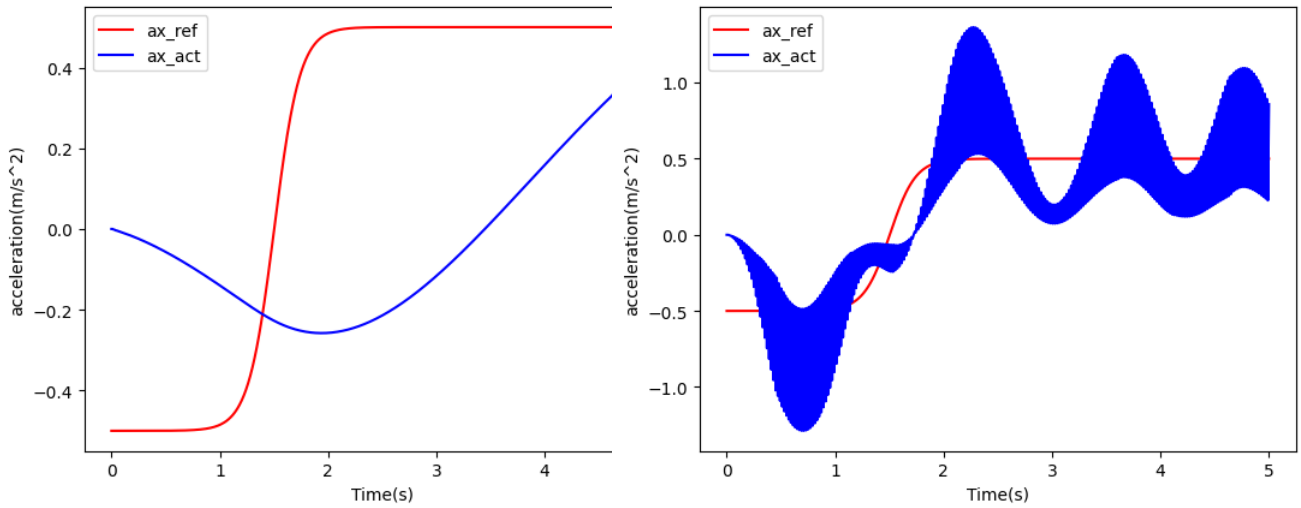


Figure 5.7: Block diagram of simulation

simulation results for First-order NDI



(a) response with low K value

(b) response with large K

Figure 5.8: simulation results

As can be see, in figure-5.8a, the K value was chosen to be very low, which led to very slow tracking of desired acceleration. Figure-5.8b shows the response for larger K. It improved the speed of tracking but produced too many unwanted oscillations. Hence, to improve performance, second-order NDI was implemented next.

simulation results for Second-order NDI

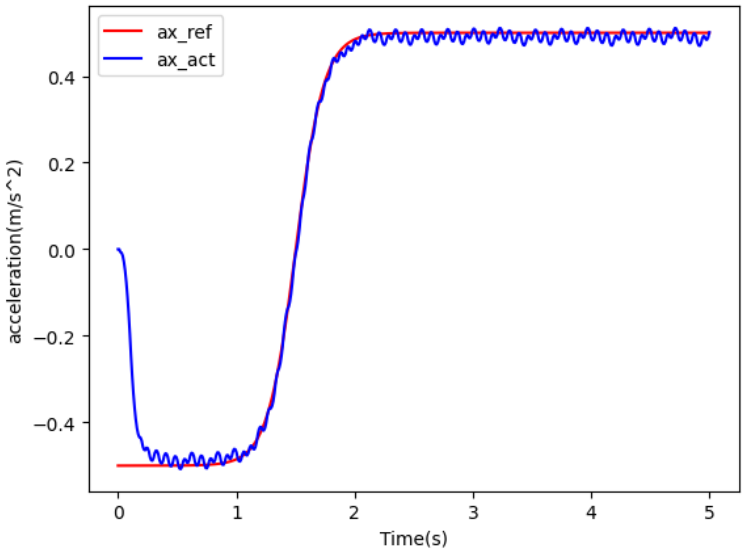


Figure 5.9: simulation results.

The performance here is much netter compared to first order NDI. With second-order NDI, damping can be introduced by tuning K_d. Hence, K_p can be increased for faster response and K_d can be adjusted accordingly to handle unnecessary oscillation.

Parameters used for simulation:
$K_p = 99000$
$K_d = 100$
$R = \begin{bmatrix} 10^{-5} I_{3\times3} & 0 \\ 0 & 10^{-6} I_{3\times3} \end{bmatrix}$

5.3.4 Implementation on crazyflie

Second order NDI control law was implemented on crazyflie.

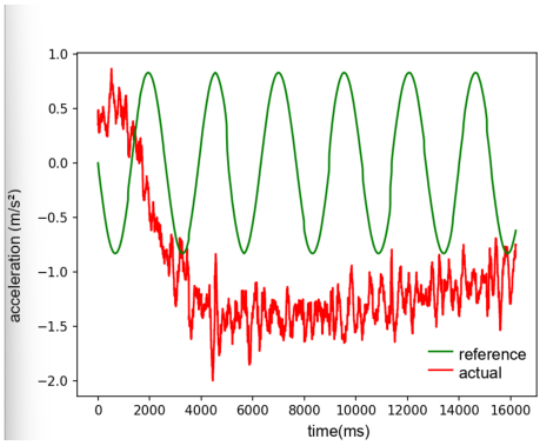
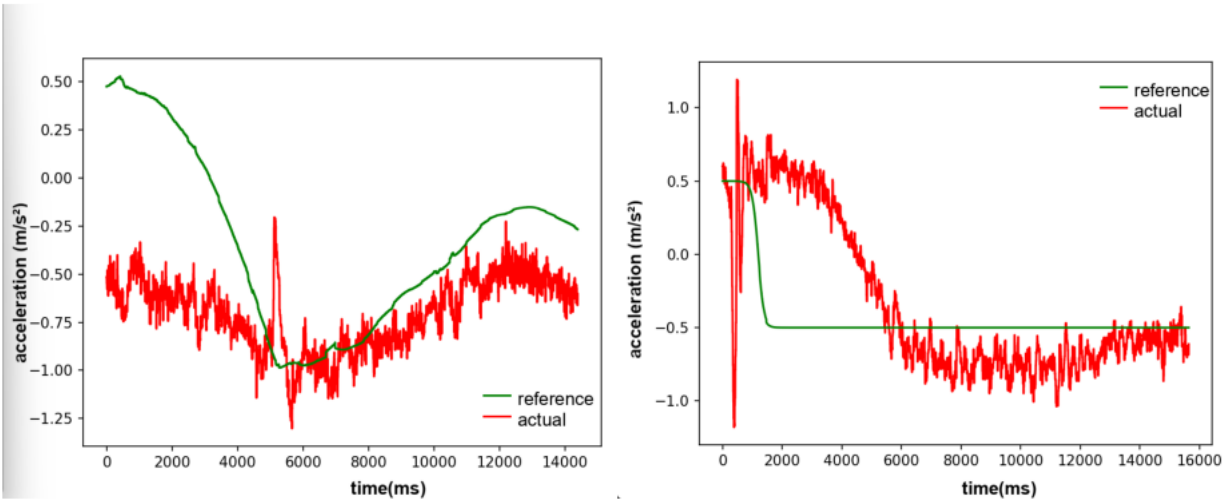


Figure 5.10: Hardware implementation results



(a) (b)

Figure 5.11: Hardware implementation results

Unlike simulation, the results obtained here were not satisfactory. In figure-5.10, at about 4 milliseconds mark, the tracking stops. This was because the flying arena was not that big, so crazyflie was commanded not cross that boundary and hence it was unable to properly follow the reference acceleration. Figure-5.11a and figure-5.11b show slightly better results, however they still are unsatisfactory.

Remarks on unsatisfactory Results

The attempts to design a proper acceleration controller for crazyflie in this thesis did not provide satisfactory results. Some of the reasons that could

have contributed to this are:

1. The results shown above in figures 5.10,5.11a and 5.11b were implemented such that the height of crazyflie was maintained constant. The crazyflie firmware has inbuilt functionality to control attitude and attitude rates at a fixed height. Basically, it was a XY-acceleration control. However, the control law-(5.30) generates thrust input also along with attitude and attitude rates. That thrust is necessary to provide lateral acceleration. But to maintain height, crazyflie uses a separate thrust calculated from PID controller instead of this thrust input. This means, in order to maintain same height, the thrust to track the acceleration generated by control law is ignored in this scenario. A possible solution is to somehow decouple the process of maintaining height and following the thrust input from control law. This was both thrust can be incorporated into the final functioning of crazyflie.

2. The acceleration feedback obtained from accelerometers were very noisy even after filtering. The external forces acting on crazyflie leads to noisy acceleration measurements, and resulted in a deviation of about 0.5 from true accelerations. This means, if the reference acceleration is also small and in same order of magnitude as noise, noise will play a more significant role in that case. But if a large enough acceleration was given, it can be easily differentiated from noise, and hence effect of noise can be reduced, which might result in better performance. The flying arena used for this thesis was small in size so giving such large accelerations was not possible, other wise crazyflie would very quickly crash onto the walls of this limited space.

3. The firmware fro crazyflie is completely ope-source. So it can be fully modified. To improve the performance of NDI controller, some parameters in onboard PID can be tuned. For this a test bench is required, where the quadcopter can be placed, where it is not free to fly off. Then the gains can be tuned accordingly. However, this test bench was not available during the work done for this thesis. So any attempt to tune these gains caused crazyflie to randomly crash every-time and hence tuning of gains could not be done properly.

Chapter 6

Conclusion

During the course of this work, different multi-agent decentralized control laws were implemented on a nanoquadcopter platform: *Crazyflie 2.1*. The practical implementations were done to test the efficacy of the control-laws in a real world scenario.
The crazyflie firmware was completely open source and the python library developed by Bitcraze provided an easy and seamless way to implement different algorithms with crazyflies. The lighthouse positioning system(LPS) provided accurate positions of crazyflie which was helpful for bearing-only control laws where agents were required to take relative-bearing information from their neighbors.

Many multi-agent control laws model the agents as double-integrator , hence a controller that can accurately track acceleration inputs is necessary. Crazyflie firmware did not have inbuilt controller to track acceleration input, hence attempt was made to develop on from scratch. Two different approaches were used. However neither gave satisfactory results in the end. The performance of the controller developed using Nonlinear dynamic inversion (NDI) could be improved by decoupling the thrust to maintain a specific height(if only XY acceleration is considered) and the thrust that is commanded by NDI to track XY-accelerations. The experiments could be performed in a bigger arena in future so quadcopter gets more free space to move around. A test bench for crazyflie can be developed or 3D printed, where the crazyflie can be placed and the gains of onboard PID can be tuned for improving the performance NDI controller.

The work done in this thesis for developing an acceleration controller did not provide satisfactory results, hence in future, the observations made in this thesis(section 5.3.4) can be utilized to extend the work done in this thesis and thus improve the results.

Bibliography

[1] K. ArjunDas, Avinash Kr Dubey, and Dwaipayan Mukherjee. "Containment control of heterogeneous multi-agent systems". In: *IFAC-PapersOnLine* 55.22 (2022), pp. 363–368.

[2] Bitcraze. *Crazyflie 2.1*. June 2024. URL: https://www.bitcraze.io/development/development-overview/.

[3] Bitcraze. *Lighthouse positioning*. June 2024. URL: https://www.bitcraze.io/documentation/system/positioning/ligthouse-positioning-system/.

[4] Control.com. *A Technical Overview of Drones and their Autonomous Applications*. June 2024. URL: https://control.com/technical-articles/a-technical-overview-of-drones-and-their-autonomous-applications/.

[5] Julian Förster. "System Identification of the Crazyflie 2.0 Nano Quadrocopter". Bachelor's thesis. ETH Zurich, 2015.

[6] Alexander Shtuchkin. *Vive positioning*. June 2024. URL: https://github.com/ashtuchkin/vive-diy-position-sensor/?tab=readme-ov-file.

[7] Ewoud JJ Smeur, Guido CHE de Croon, and Qiping Chu. "Cascaded incremental nonlinear dynamic inversion for MAV disturbance rejection". In: *Control Engineering Practice* 73 (2018), pp. 79–90.

[8] Arnaud Taffanel et al. "Lighthouse positioning system: Dataset, accuracy, and precision for UAV research". In: *arXiv preprint* (2021). arXiv: 2104.11523 [cs.RO].

[9] Ezra Tal and Sertac Karaman. "Accurate tracking of aggressive quadrotor trajectories using incremental nonlinear dynamic inversion and differential flatness". In: *IEEE Transactions on Control Systems Technology* 29.3 (2020), pp. 1203–1218.

[10] wikipedia. *Polynomial geometric properties*. June 2024. URL: https://en.wikipedia.org/wiki/Geometrical_properties_of_polynomial_roots.

[11] Shiyu Zhao and Daniel Zelazo. "Bearing rigidity and almost global bearing-only formation stabilization". In: *IEEE Transactions on Automatic Control* 61.5 (2015), pp. 1255–1268.