Final Report
Semester Project
Laboratory of Intelligent Systems

Realistic Simulation Environment for Obstacle Avoidance of Quadcopter Swarms

Supervisor:
Prof Dario Floreano

Assistants:
Enrica Soria
Dr Fabrizio Schiano

Author:
Mahdi Nobar

29th May, 2019

Contents

1 Abstract

The ultimate goal of this project is to simulate obstacle avoidance behaviour of quadcopters swarm in a Gazebo. For this goal, initially the Crazyflie 2.0 was chosen as a quadcopter to be simulated and CrazyS extension of RotorS autopilot was selected to execute the swarm node. Next, it was realized that the RotorS could only send angular velocity command to the rotors of quadcopters while our controller would provide velocity command. Then it was decided to choose PX4 autopilot to communicate with Gazebo through ROS knowing that PX4 supports sending velocity command to Gazebo. Thus, the reference configuration of the parameters of the Crazyflie was provided to PX4, but it was observed that based on reference parameters the Crazyflies even could not take o. Afterwards, required drone parameters for PX4 were either collected or calculated based on available resources. Then, each of the identified parameters were tested for the Crazyflie, and it was realized that the Crazyflie could take o. However, there were a large steady state error between the commanded step velocity to the Crazyflie and its real velocity. Therefore, either the Crazyflie physical parameters are required to be evaluated by conducting a thorough identification or PX4 PID gains for all its embedded controllers should be tuned for the Crazyflie. Afterwards, it was decided to continue the project based on Iris quadcopter. Finally, obstacle avoidance maneuver of swarms of Irises were studied by utilizing the olfati-saber protocols.

2 Introduction

The use of aerial swarms to solve real-world problems has been increasing steadily, accompanied by falling prices and improving performance of communication, sensing, and processing hardware. [12] In swarm robotics, a swarm is a coordinated group of devices, be it robots or sensors, that can perform a multitude of tasks that a single device would be unable to accomplish alone in a timely fashion. For example, a search and rescue mission would be greatly improved by the ability to spread out a number of drones and sweep them over a wide area. [8]

2.1 Motivation

Various applications of quadcopters swarms have made it a possible topic for realistic simulation. Simulation has been recognized as an important research tool since the beginning of the 20^{th} century. However, the *good times* for simulation started with the development of computers and now the simulation is a powerful visualization, planning, and strategic tool in di erent areas of research and development. The simulation has also a very important role in robotics. [16] One of the main challenges of drones swarms is their capability to avoid obstacles while flocking towards the target point. In fact, the task of obstacle avoidance is to guide the drones toward the goal without collision with static obstacles. Design of a fast and e cient method of obstacle avoidance is one of the important problems for drones flocking. Therefore, in this project it is focused on the realistic simulation of the quadcopters swarms in obstacle avoidance maneuver. Initially in this project, Crazyflie 2.0 shown in figure (1) was chosen for the simulation among several small sized quadrocopters due to its features.

2.1.1 Crazyflie 2.0

The Crazyflie 2.0 is a lightweight, open source flying development platform based on a nano quadcopter as shown on figure (1).

Figure 1: *Crazyflie 2.0 nano quadcopter*

Crazyflie 2.0 has several features such as:

- High Functionality: The Crazyflie 2.0 is equipped with low-latency long-range radio as well as Bluetooth LE. This provides the option of downloading its application and utilization of a mobile device as a controller. Crazyflie 2.0 is also charged via USB. The board contains an EEPROM memory for storing configuration parameters and a 10 degree of freedom IMU with accelerometer, gyroscope, magnetometer and a high precision pressure sensor.
- Modular: It has a flexible expansion interface where a variety of expansion decks can be attached, both on the top and the bottom of the Crazyflie 2.0. From this expansion interface the user can access buses such as UART, I2C and SPI as well as PWM, analog in/out and GPIO.
- Lightweight and Small in Size: The Crazyflie 2.0 density is $9\ cm^3$, and it weighs only 27 g. The size makes it ideal for indoor applications. Even though the propellers spin at high RPMs, they are soft and the torque in the motors is very low when compared to a brushless motor. It also features $4x7\ mm$ coreless DC-motors that provide it a maximum takeo weight of 42g. The Crazyflie 2.0 is quite fast if you let it be, but even if it crashes it is still only 27g which means the kinetic energy involved in a crash is fairly low. During hard impact, the system is designed to break at the cheapest component, the motor mounts, which are available as spare parts.
- Open source: The Crazyflie 2.0 is an open source project, with source code and hardware design both documented and available. Since all of our development tools are open source (except for iOS). Aside from the firmware and software projects, there are a number of community supported APIs written in Java, Ruby, `C`, `C++`, `C#` and Javascript. [2]

In order to perform described tasks, the project timeline shown of figure (2) was defined.

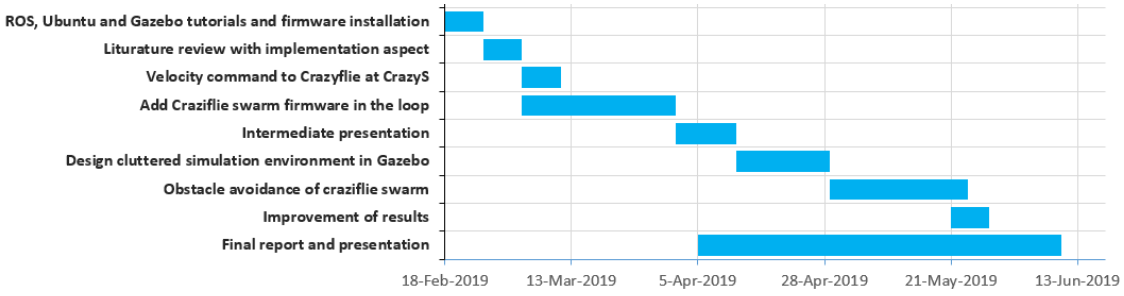

Figure 2: *Project Gantt chart*

3 State of the Art

In this section a short summary of relevant works to the project are provided. The selection rule of each article are either to relate to the simulation of drones or to their flocking in real implementation.

3.1 CrazyS

CrazyS is the name of an extension of a ROS[1] package called RotorS[2]. CrazyS integrates Crazyflie nano-quadcopter in Gazebo. Simulation of the drone in Gazebo allows to analyze the behaviour of a Crazyflie with details close to reality. This project expands the RotorS simulator such that it includes the model of the Crazyflie physical model as well as its flight control system. [5] In this project the final command of the controller to Gazebo is the rotational speed of each motors of Crazyflie. However for over purposes, we would like to provide velocity commands to the drones, and we realized that it is not possible to utilize CrazyS expansion to send velocity command to the drones, because RotorS autopilot only supports sending angular speed command of rotors of the drones. Therefore, as CrazyS is based on RotorS autopilot then it only supports sending angular speed command to the rotors of the Crazyflie. Nevertheless, our controller provides velocity command to the Crazyflie and it is required to convert it to the equivalent rotor angular speed commands to be able to utilize CrazyS.

3.2 Olfati Saber Protocols

The next relevant work was done by Reza Olfati-Saber on a novel flocking algorithm. He first presents a theoretical framework for design and analysis of distributed flocking of agents.[10] Then, it presents several testing scenarios with or without obstacles. Finally, he demonstrates that his algorithm embodies three rules of Reynolds flocking algorithm which are:

1. Flock Centering, which attempts to keep the agents close
2. Collision Avoidance, that aims to avoid collision with adjacent flocking agents
3. Velocity Matching, that is provided to match the velocity of flocking agents

[1] Robot Operating System
[2] https://github.com/ethz-asl/rotors_simulator

Then a systematic method for building a cost function which is called collective potential is proposed. The aim of the optimizing the collective potential is to penalize the deviation from a class of lattice shaped objects called α-lattice. Additionally, virtual agents related to each α-agent within a specified distance to the obstacle are defined. Such virtual agents are called β-agent and γ-agent. A β-agent is put at the position of an obstacle and acts as a repelling force for the adjacent α-agents, while a γ-agent is another virtual agent which steers the drone which has already passed the obstacle to allow the remaining drones to pass the obstacle. The γ-agent allows the rejoin of the agents during obstacle avoidance behaviour.[9]

According to the algorithm, a boundary is defined around each obstacle so that, as soon as any agent enters the boundary both a β-agent and a γ-agent are created. This imaginary γ-agent takes position equal to the sum of the position vector of β-agent with desired direction of α-agent towards the target point multiplied by a coe□cient as written on equation (1). In this equation, $n_{desired}$ is the desired flocking unit direction. Afterwards, the γ-agent is then perceived as another α-agent for each α-agents inside boundary around the obstacle. Actually, this γ-agent steers the agent around the obstacle in order to not allow the agent which encounters first with the obstacle to stay near the obstacle for a long time because of the influence of the other agents which have not yet passed the obstacle. So the role of γ-agent is crucial in rejoin maneuver of drones after obstacle avoidance.

$$r_\gamma = r_\beta + \lambda n_{desired} \tag{1}$$

3.3 Modification of Olfati Saber Obstacle Avoidance Algorithm

In 2017, Iovino et al. proposes the implementation of a distributed flocking algorithm with obstacle avoidance capability for unmanned aerial vehicles swarms. In this implementation, the main limiting hypothesis of Olfati Saber flocking with obstacle avoidance about the shape of obstacles is removed. According to [10], the obstacles should have convex boundaries. Thus, the obstacles should not be like a finite wall or concave-shaped obstacles. This comes from the fact that obstacle avoidance term of Olfati algorithm has a term which provides a *clue* about the direction wherewith the α-agent should align its velocity to perform obstacle avoidance manoeuvre. This direction is proportional to the projection of the α-agent velocity vector on the local tangent plane to the obstacle surface. So there could be practical cases where the projection term of the obstacle avoidance acceleration is near to zero, e.g when there is a concave obstacle or the current velocity of the agent is perpendicular to the obstacle and remains like that. In such situations, the agent gets *stuck* at a certain distance from the obstacle and it will not move any longer. In Iovino's article the original algorithm by Olfati Saber has been extended to overcome its limitations. The basic idea is to intervene on the original algorithm when a situation occurs in which the projected velocity of the α-agent assumes a value very close to zero. Hence, in these circumstance, a desired value to the projected velocity is assigned.

3.4 Vàsàrhelyi Flocking Algorithm

Vàsàrhelyi et al. proposes a flocking model for a large number of real drones. Based on real test results, it is demonstrated that the swarm behaviour is stable for thirty drones with

high velocity. In order to have stable and scalable flocking model for real flying robots, this research addresses some challenges and aims to solve them. The challenges include:

1. Reality Gap: This challenge means that the flocking model which is working in an idealistic simulation environment is not necessarily stable in a realistic tests. In fact, the issue origins from the fact that the idealistic simulation environment do not consider all the delays, uncertainties and kinematic constraints in reality.
2. Adaptability: This comes from the fact that the swarm algorithms that have been developed for an open space are not necessarily able to have the expected performance in a confined environment.
3. Scalability: Also the scalability issue stems from the fact that the motion pattern of an algorithm, that is stable for specific number of agents and/or specific velocity, is not necessarily stable for higher velocities or larger number of agents.
4. High Dimensional Algorithm: Flocking models require tuning of substantial number of parameters for wide range of conditions in reasonable time which is an issue with real testing the flocking algorithm.

This paper fills the gaps for real flight of 30 autonomous quad-copters performing stable swarm in a cluttered environment by utilizing a scalable and optimized framework based on realistic dynamic modelling.[11]

3.5 Mixed Reality for Robotics

One of the potential applications of having a realistic simulation is to overcome some limitations in real implementation. In 2015 Hönig et al. refine the definition of Mixed Reality to accommodate seamless interaction between physical and virtual objects in any number of physical or virtual environments. In fact, Mixed Reality creates a space in which both physical and virtual elements co-exist, allowing for easy interaction between the two. Particularly, it is shown that Mixed Reality can reduce the gap between simulation and implementation by enabling the prototyping of algorithms on a combination of physical and virtual objects, including robots, sensors, and humans. Robots can be enhanced with additional virtual capabilities, or can interact with humans without sharing physical space. Mixed Reality is demonstrated with three representative experiments, each of which highlights the advantages of the proposed approach. Furthermore, a test bed for Mixed Reality with three di□erent virtual robotics environments in combination with the Crazyflie 2.0 quad-copter is presented in this article. [14]

4 Tools

4.1 Robot Operating System

The Robot Operating System (ROS) is a collection of software frameworks for robot software development. It provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place

in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. ROS can be used to send control commands from a node which can be written in Python or C++ to a realistic simulation engine like Gazebo. Then simulation results can be retrieved through ROS and be used by the controller node. Therefore, robot operating system provides possibility to evaluate a controller performance by a realistic simulator. There is a meta-package called *gazebo_ros_pkgs* which provides packages for integrating ROS with Gazebo.

4.1.1 rosbag

In order to record and playback ROS message data, a command line tool called *rosbag* is used. *rosbag* uses a file format called bags, which log ROS messages by listening to topics and recording messages as they come in. Playing messages back from a bag is the same as having the original nodes which produce data in the ROS computation graph, making bags a useful tool for recording data to be used in later development. In this project, *rosbag* is widely used in order to save the simulation results, later messages saved in *bag* file is used to plot required results.

4.2 Gazebo

Gazebo is an open-source 3D robotics simulator. Gazebo utilizes multiple high-performance physics engines, such as ODE, Bullet, Simbody, and DART (the default is ODE). It provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle) and etc.

4.3 Simulink

Simulink is a graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries.

4.4 RotorS

RotorS is a MAV gazebo simulator that supports models for aerial robots. There is an extension of *RotorS* which is called *CrazyS* that aims to modeling, developing and integrating the Crazyflie 2.0 nano-quadcopter in the physics based simulation environment Gazebo.

4.5 PX4

PX4 is an open source flight control software for drones and other unmanned vehicles. In this project the main capability of PX4 is that it provides o□-board control which is necessary to evaluate the obstacle avoidance behaviour of drone swarms based on provided algorithm. In other words, PX4 flight stack can be controlled at o□-board control mode using software running outside of the autopilot. The main advantage of this capability

in this project is that the proposed swarm obstacle avoidance algorithm provides velocity commands to each drone which then can be sent to Gazebo using o□-board mode of PX4.

5 Implementation

The ultimate goal of this project is to simulate obstacle avoidance behaviour of quadcopters swarm in a Gazebo. For this goal, initially the Crazyflie 2.0 was chosen as a quadcopter to be simulated and CrazyS extension of RotorS autopilot was selected to execute the swarm node. Next, it was realized that the RotorS could only send angular velocity command to the rotors of quadcopters while our controller would provide velocity command. Then it was decided to choose PX4 autopilot to communicate with Gazebo through ROS knowing that PX4 supports sending velocity command to Gazebo. Thus, the reference configuration of the parameters of the Crazyflie was provided to PX4, but it was observed that based on reference parameters the Crazyflies even could not take o□. Afterwards, required drone parameters for PX4 were either collected or calculated based on available resources. Then, each of the identified parameters were tested for the Crazyflie, and it was realized that the Crazyflie could take o□. However, there were a large steady state error between the commanded step velocity to the Crazyflie and its real velocity. Therefore, the Crazyflie parameters for PX4 autopilot were required to be evaluated by conducting a thorough identification. Afterwards, it was decided to continue the project based on Iris quadcopter. Then by utilizing olfati_saber protocols, obstacle avoidance maneuver of swarms of Irises were studied.

5.1 Parameter Analysis of Olfati-Saber Algorithm

In order to implement swarms of quadcopters olfati_saber algorithm is used.[10] This algorithm provides three terms of acceleration which are summed to provide the total acceleration. The first acceleration term is named acc_potential and is used to keep the drones on α-lattices. In fact, cost functions or collective potentials are defined in a way that they penalize the deviation from a class of lattice-shape objects called α-lattice. Therefore, acc_potential is an acceleration which attempts to keep the drones on α-lattice so to allow agents to stay at a specified distance from neighboring agents.

The second acceleration term is for avoiding any static obstacle on agents' path and is named acc_obstacles in the code.

Finally the third acceleration term is meant to modify the velocity of each agents in order to migrate towards the specified target point. This term is named acc_vel_matching at the code.

Moreover, the meaning of each parameter of olfati_saber algorithm implementation is provided in Table 1.

parameter	meaning
c_vm	Acceleration coe□cient for migration towards the target point acc_vel_matching. It specifies the weight of migration acceleration. Larger values result in more influence of migration acceleration so the agents go faster towards the target.
a	Parameter of potential function ϕ which satisfies $0 < a \leq b$
b	Parameter of potential function ϕ which satisfies $0 < a \leq b$
d	Global minimum for the attractive/repulsive potential function ψ_α which attempts to keep agents inside perception_radius on alpha-lattice. This parameter is in fact the distance between latices [i.e the distance that adjacent drones should attempt to maintain].
delta	Variable of bump function ρ [10] which specifies the start point of decreasing smoothly from value 1 to 0 which is used to smoothly cut-o□ the potential function. This variable satisfies $0 < delta < 1$. As $delta$ changes from 1 to 0 in allows the potential acceleration to cut-o□ more smoothly when drones go out of perception_radius. If $delta = 1$ the bump function is a step function.
r	It is the communication radius between the agents. As long as agents have distance less than communication radius, potential acceleration is applied to keep agents on α -lattices.
k	A factor to manipulate the shape of bump function ρ
lambda	Weight of target unit vector to specify the position of γ-agent according to equation (1)
c_pm_obs	It is the weight of repulsive acceleration from β-agent
v_migration	It is migration gain which specifies how fast to migrate towards the target
perception_radius	The boundary around each α-agent in which the potential acceleration is applied to keep all α-agents inside this boundary to flock on α-lattices
max_agents	maximum number of α-agents around each agent including each agent itself which are inside perception_radius to be kept on α-lattices

Table 1: Hyper-parameters' description for the olfati_saber algorithm

5.2 Crazyflie Parameters

In order to simulate the flight of quadcopters it is required to provide kinematic and basic physics description to Gazebo. This information is provided as XML Macros[1] format. By default there are some physical parameters which are necessary to be identified for each type of robot that should be simulated. For this reason, required parameters of Crazyflie are collected in Table 2 based on corresponding references available on Table 3. Also

[1]xacro: xacro is a scripting mechanism that allows more modularity and code re-use when defining a URDF model. When using it, what is actually uploaded to the parameter servers (per default as the "robot_description" parameter) actually is a URDF, as that is generated from the xacro file in the launch file. In fact, xacro is another way of defining a URDF. It facilitates defining several models in the world, for instance a "wheel" macro can be generated and instantiated multiple times with di□erent parameters to put several wheels on the robot, as opposed to copying and pasting the same code several times manually.

calculation for obtaining some of the parameters which are computed by combination of information from di□erent resources are provided.

Parameter Name	Reference Number	Unit	Value	Required Unit	Required Value
mass	[p.r.1][1]	kg	2.70e-2	kg	2.70e-2
	[p.r.2]	kg	2.80e-2	kg	2.80e-2
body_width	[p.r.1]	m	4.50e-2	m	4.50e-2
body_height	[p.r.1]	m	3.00e-2	m	3.00e-2
mass_rotor	[p.r.1]	kg	5.00e-04	kg	5.00e-04
arm_length	[p.r.1]	m	4.60e-02	m	4.60E-02
rotor_o□set_top	[p.r.1]	m	2.40e-02	m	2.40e-02
radius_rotor	[p.r.1]	m	2.25e-02	m	2.25e-02
motor_constant	[p.r.1]	$kg.m/s^2$	1.28e-08	$kg.m/rad^2$	
	[p.r.3],[p.r.4],[p.r.5]	$kg.m/rad^2$	9.80e-08	$kg.m/rad^2$	9.80e-08
	[p.r.2],[p.r.6],[p.r.7]	kg.m	9.09e-08	$kg.m/rad^2$	
moment_constant	[p.r.1]	m	5.96e-03	m	5.96e-03
	[p.r.4]		6.73E-03	m	
	[p.r.3],[p.r.4],[p.r.5]		6.82e-04	m	
	[p.r.2],[p.r.6],[p.r.7]		2.41e-04	m	
time_constant_up	[p.r.1]	s	1.25e-02	s	1.25e-02
time_constant_down	[p.r.1]	s	2.50e-02	s	2.50e-02
max_rot_velocity	[p.r.1]	rad/s	2.62e+03	rad/s	2.62e+03
rotor_drag_coe□cient	[p.r.1]		8.06e-05		8.06e-05
rolling_moment_coe□cient	[p.r.1]		1.00E-06		1.00E-06
Body_inertia ixx	[p.r.1]	$kg.m^2$	1.66e-05	$kg.m^2$	1.66e-05
	[p.r.2]	$kg.m^2$	1.66E-05	$kg.m^2$	1.66E-05
Body_inertia ixy	[p.r.1]	$kg.m^2$	0.0	$kg.m^2$	0.0
	[p.r.2]	$kg.m^2$	8.31e-07	$kg.m^2$	8.31e-07
Body_inertia ixz	[p.r.1]	$kg.m^2$	0.0	$kg.m^2$	0.0
	[p.r.2]	$kg.m^2$	7.18e-07	$kg.m^2$	7.18e-07
Body_inertia iyy	[p.r.1]	$kg.m^2$	1.66e-05	$kg.m^2$	1.66E-05
	[p.r.2]	$kg.m^2$	1.67e-05	$kg.m^2$	1.67e-05
Body_inertia iyz	[p.r.1]	$kg.m^2$	0.0	$kg.m^2$	0.0
	[p.r.2]	$kg.m^2$	1.80e-06	$kg.m^2$	1.80e-06
Body_inertia izz	[p.r.1]	$kg.m^2$	2.93e-05	$kg.m^2$	2.93e-05
	[p.r.2]	$kg.m^2$	2.93e-05	$kg.m^2$	2.93e-05

Table 2: Kinematic and basic physics parameters of Crazyflie 2.0

[p.r.1]	Sileno G., CrazyS, GitHub repository, https://github.com/gsilano/CrazyS/blob/master/rotors_description/urdf/crazyflie2.xacro
[p.r.2]	Forster, J. (2015, August). System Identification of the Crazyflie 2.0 Nano Quadrocopter. Retrieved from http://mikehamer.info/assets/papers/Crazyflie%20Modelling.pdf
[p.r.3]	Tobias. (2015, February). Measuring propeller RPM: Part 3. Retrieved from https://www.bitcraze.io/2015/02/measuring-propeller-rpm-part-3/
[p.r.4]	Marques, N. (2017, July). PX4/sitl_gazebo. Retrieved from https://github.com/PX4/sitl_gazebo/issues/110
[p.r.5]	Spare 7x16 mm coreless DC motor with connector specification. Retrieved from https://www.seeedstudio.com/Crazyflie-2-0-Spare-7x16-mm-coreless-DC-motor-with-connector-p-2115.html
[p.r.6]	Vernacchia, M. (2019, January). Gazebo Motor & Propeller Model Notes. Retrieved from https://github.com/mvernacc/gazebo_motor_model_docs/blob/master/notes.pdf
[p.r.7]	Bitcraze Wiki. (2015 July). Analyses of finding a PWM to thrust transfer function. Retrieved from https://wiki.bitcraze.io/misc:investigations:thrust

Table 3: References corresponding to table 2

In the following, computation of parameters in highlighted cells of Table 2 which are based on combination of different references are provided. For motor_constant based on [p.r.3],[p.r.4],[p.r.5] of Table 3 and by assuming gravitational acceleration is $g = 9.81\ m/s^2$ it can be written:

$$\Omega_{max} = 2618\ rad/s = 25012\ rpm \tag{2}$$

$$\begin{aligned} motor\_constant &= \frac{Thrust_{max}}{\Omega_{max}^2} \\ &= \frac{25012^2 \times 1.0942e-10}{2618^2} \times 9.81 \\ &= 9.80 \times 10^{-8}\ kg.m/rad^2 \end{aligned} \tag{3}$$

Also based on references [p.r.2],[p.r.6] and [p.r.7] of Table 3 by assuming the air density equal to $\rho = 1.225\ kg/m^3$ the motor_constant and moment_constant are calculated as follows:

$$\begin{aligned} motor\_constant &= \frac{57.9 \times 0.001 \times 9.81}{1.225 \times (\frac{23882}{60})^2 \times (0.0225 \times 2)^4} \times \frac{1.225 \times (0.0225 \times 2)^4}{(2 \times \pi)^2} \\ &= 9.09 \times 10^{-8}\ kg.m \end{aligned} \tag{4}$$

$$\begin{aligned} moment\_constant = \frac{57.9 \times 0.001 \times 9.81 \times 0.005964552 + \frac{1.563383e-5}{(1.225 \times (\frac{23882}{60})^2 \times (0.0225 \times 2)^5)}}{1.225 \times (\frac{23882}{60})^2 \times (0.0225 \times 2)^4} \times \\ \times 0.0225 \times 2 = 2.41 \times 10^{-4} \end{aligned} \tag{5}$$

5.3 PX4 o□-board control with MAVROS for Iris quad-copter

This section presents an installation guide for PX4 o□-board control of an Iris quad-copter to slowly take o□ to an altitude of 2 meters is explained. In order to do that a node containing control source file is added to the available ROS package. First lets define the concept of o□board mode at PX4 autopilot. O□-board mode is primarily used for controlling vehicle movement and attitude, and supports only a very limited set of MAVLink commands. It can be used to control vehicle position, velocity, or thrust and to control vehicle attitude.

1. Create a new package in the catkin workspace, called o□board_example_package

```
cd ~/catkin_ws/src
catkin_create_pkg offboard_example_package std_msgs
roscpp geometry_msgs mavros_msgs
```

2. Move to the source folder of the new package and create a source file

```
cd offboard_example_package/src
touch offboard_example_package/src/offb_node.cpp
```

3. Paste the source code from https://dev.px4.io/en/ros/mavros_o□board.html in o□b_node.cpp

```
subl offb_node.cpp
```

4. Uncomment line 137 and 150 to 152 of the CMakeLists.txt file to define the executable node and required catkin libraries

```
cd ~/src/offboard_example_package
subl CMakeLists.txt
```

5. Create launch directory for the new package

```
mkdir -p launch
```

6. Paste the PX4.launch file from the mavros directory to the launch directory inside the new package

```
roscd mavros
cp launch/px4.launch
~/catkin_ws/src/offboard_example_package/launch
```

Workaround: If you get “roscd: No such package ‘mavros’”, but mavros is properly installed you can navigate to it manually as:

```
cd ~/catkin_ws/src/mavros/mavros/
```

7. Now build the catkin workspace

```
cd ~/catkin_ws
catkin build
source ~/catkin_ws/devel/setup.bash
```

8. Change add_executable($PROJECT_NAME_node src/offboard_example_package_node.cpp) to add_executable($PROJECT_NAME_node src/offb_node.cpp) at line 137

```
cd ~/catkin_ws/src/offboard_example_package/launch
subl CMakeLists.txt
```

9. In the px4.launch file, replace "<!– <arg name="fcu_url" default="/dev/ttyACM0:57600" />" with "<arg name="fcu_url" default="udp://:14540@localhost:14557" />"

```
subl ~/catkin_ws/src/offboard_example_package/launch/
px4.launch
```

10. Now to run the simulation do the steps below:

```
roscore
rosrun offboard_example_package offboard_example_package_node
cd ~/src/Firmware
make posix_sitl_default gazebo
make offboard_example_package px4.launch
```

6 Results and Discussion

6.1 Obstacle avoidance of Iris swarms

In this section, the parameters of the *Olfati_saber* swarm algorithm are tuned. Then for different cluttered environments inter-agent distance as well as distance from obstacles based on specific parameters are represented, and step by step the effect of parameters on Obstacle avoidance of Iris swarms based on realistic simulation results is demonstrated.

6.1.1 Obstacle avoidance of three Iris swarms from one pillar without γ-agent

In this scenario, a single pillar is located on the flying path of three Iris drones as shown on figure (3). Drones should be able to pass the obstacle without hitting it and without hitting each other or diverging from each other. For this end, the mission is defined according to Table 4.

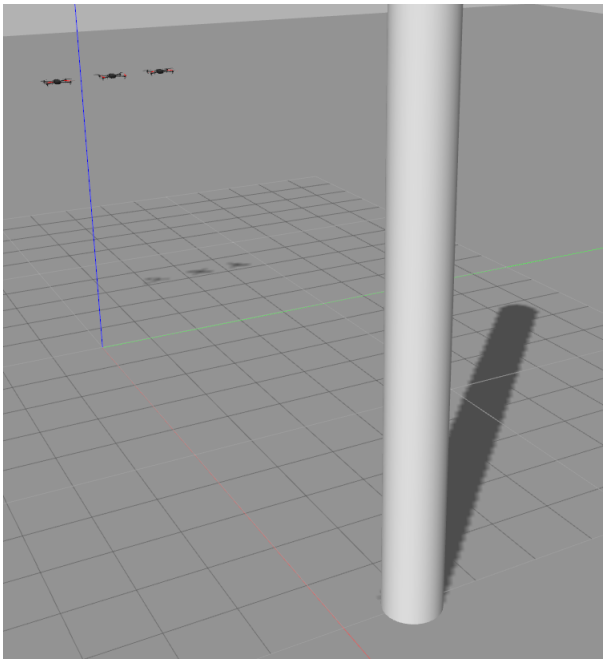

Figure 3: One pillar is put on the way-point of drone swarms

In this mission, three *iris* after take o mode start o-board control based on velocity command of the controller based on *Olfati-saber* algorithm without considering γ agent. The controller navigates drones to a target point aligned with global y and z axis of the average coordinates of drones at the end of take o mode. Besides, a pillar is placed exactly on the same $y-z$ coordinates of the target point. Therefore, it is expected that the drones flock towards the target point by avoiding the pillar and keeping the the distance in between each other. On figure the only pillar is named as *obstacle 1*.

Parameter	Value	dimension	Description
self.startpoint	[0., 1., 5.0]	m	Average Coordinates of starting point of o-board mode
self.waypoint	[200. 1. 5.]	m	Coordinates of target
self.intruders_positions	[10. 1. 5.]	m	Center of mass of pillar
self.intruders_velocities	[0. 0. 0.]	m/s	Fixed obstacle
radius	0.4	m	Radius of pillar
length	20.0	m	Length of pillar

Table 4: *Mission description for scenario one*

Based on parameters on Table 5 and scenario described on Table 4 drones have collision with pillar. This collision is shown on figure (4). In fact, because minimum distance between the drones and the pillar is obtained 64 cm, which is less than radius of pillar plus half of rotor to rotor and radius of rotor blade plus distance (i.e 64.2 cm < 40 + 27.5 + 6.4 = 73.9 cmm), so there is a collision between the drones and the pillar. This collision is

evident in the Gazebo graphical representation of the simulation. Also figure (4) indicates that the drones the obstacle diverge from each other. Also there is an oscillation happening after passing three meters from the obstacle that is because of the influence of hitting the obstacle which applies an impact to the drones.

Parameter	Value
perception_radius	10.0
max_agents	10
migration_gain	4.0
c_vm	20
a	1
b	5
d	1.0
delta	0.2
k	1
r	10
r0	5
c_pm_obs	100

Table 5: *Simulation parameters - Scenario one*

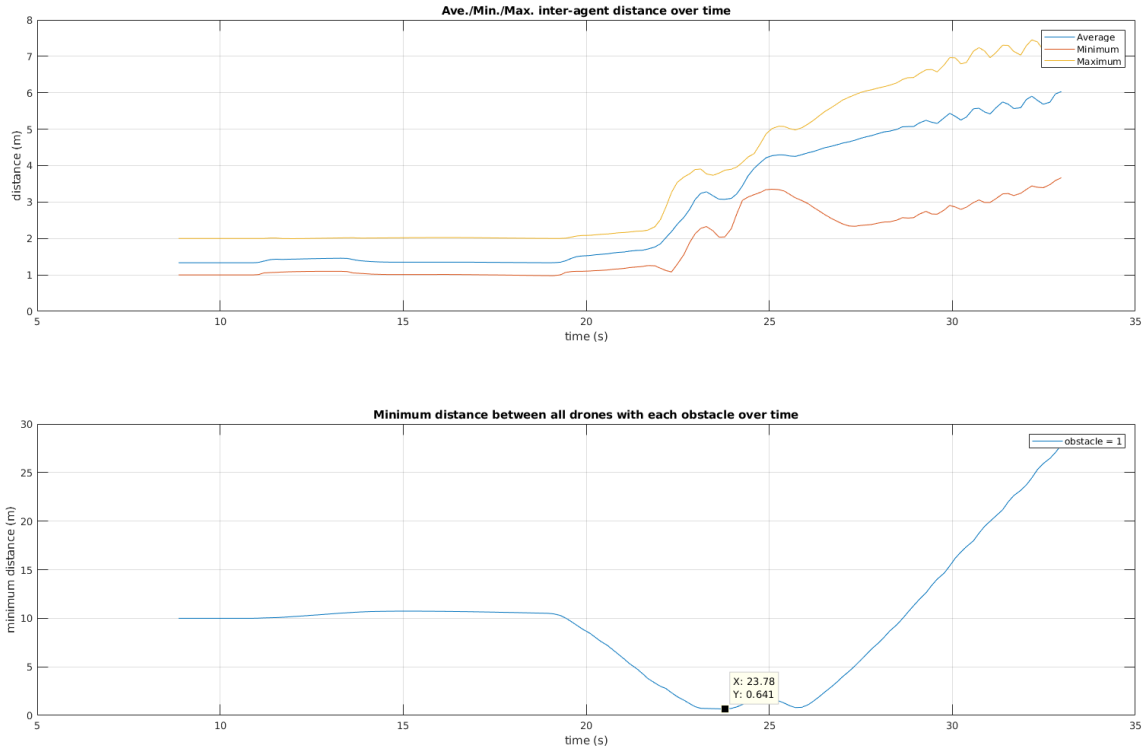

Figure 4: Inter-agent (top) and drone-obstacle (bottom) distance over time based on Tables 4 and 5

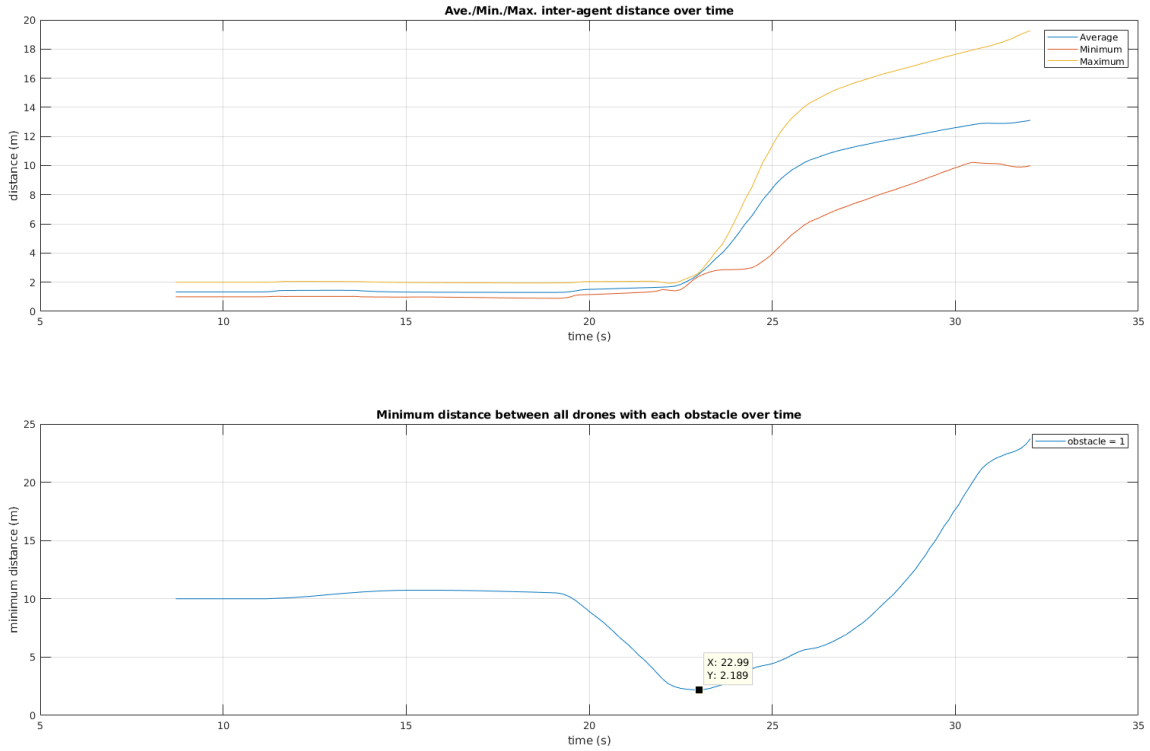

Figure 5: *Inter-agent and drone-obstacle distance over time with* $c_p m_o bs = 120$

In order to avoid the obstacle, the coe□cient of the obstacle avoidance acceleration is increased from 100 to 120 and results are plotted on figure (5). According to figure (5), drones can successfully avoid the pillar with minimum distance 2.189 meter, but the distance between them increases to approximately 18 m at $time = 30seconds$. This happens because of the fact that navigation velocity coe□cient c_vm is too high and the middle drone that requires more time to change its path to avoid the pillar falls behind other two drones. The coe□cient c_vm could not be chosen too small as according to the results drones could not continue flocking after passing the pillar inasmuch as obstacle avoidance term would have very large repulsion on the drones. So the drones could not maintain the distance in between each other, and they diverge from each other after passing the obstacle. Therefore, the coe□cient c_vm is decreased from 20 to 18 and results for 60 seconds shown on figure (6) are obtained which demonstrates that the drones avoid the pillar with minimum distance of $115cm$, and they keep the maximum distance of 10 meters in between each other so the average distance between them increases up to 15 meters at simulation time equal to 60 seconds; however, it is expected to converge to the global minimum of the potential function which is according to Table 5 is equal to 1.0 meter. This divergence happens because of the fact that the done which passes the obstacle first starts to migrate towards the target point and the remaining two drones after passing the obstacle are not able to reach the first drone. So as it is shown on figure (6) the minimum distance between drones starts to decrease but the average distance increases inasmuch as the first drone passes the pillar much earlier and migrates fast towards the target. In fact, this behaviour is the result of combination of choices for the hyper-parameters of olfati_saber swarm

algorithm. Thus, the drones after successfully avoiding the obstacle cannot converge to the specified distance because of the fact that for the obstacle avoidance strong repulsive force is required to be applied and it influences the behaviour of flocking after passing the obstacle in a way that drones cannot keep the position on the α-lattice which has 1.0 meter in between grids.

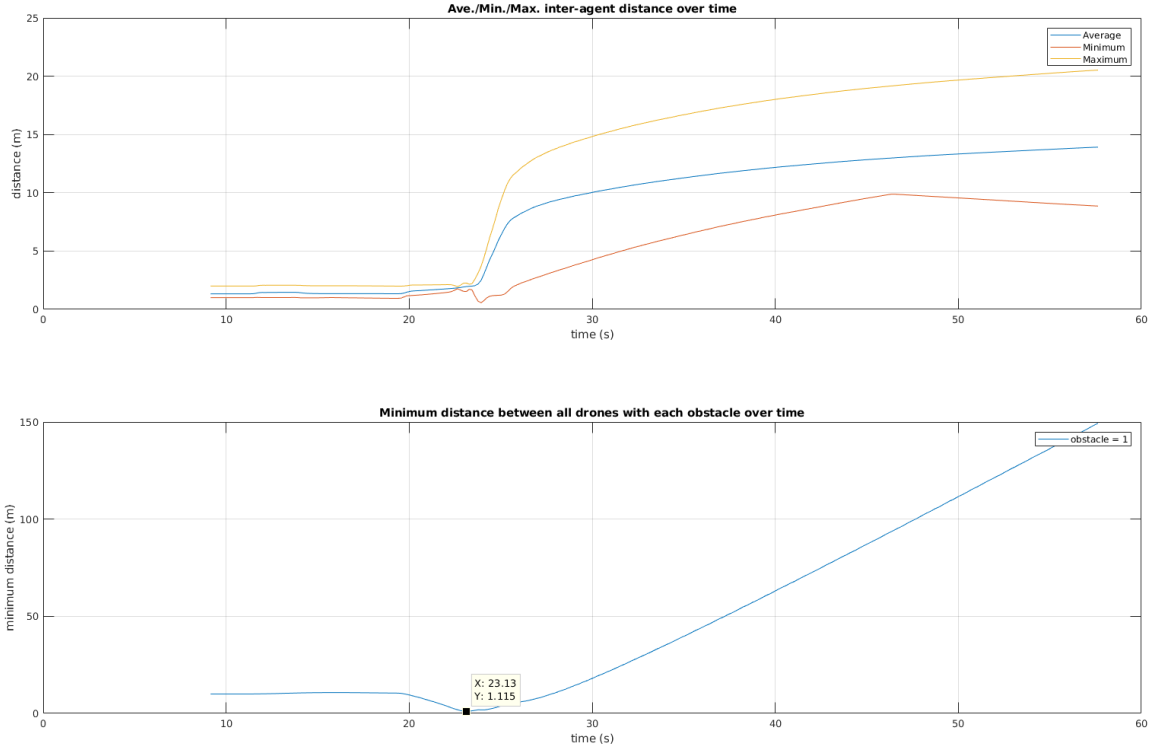

Figure 6: *Inter-agent and drone-obstacle distance over time with* $c_p m_o bs = 120$ *and* $c\_vm = 18$

6.1.2 Two Iris swarms Obstacle avoidance from one pillar with γ-agent

In subsection 6.1.1 it was shown that the divergence of drones caused by obstacle avoidance maneuver prevents the drones to be able to converge to the specified $alpha$*-lattice* as the drones would have far distance between each other after avoidance from the pillar. Now the introduction of a γ-agent allows the drone which first passes the obstacle to steer around the obstacle. Therefore, high speed gain for migration can be used without allowing the drones to separate too much from each other after the obstacle avoidance maneuver. This allows the agents to rejoin after having passed the obstacle. Thus, the distance between the drones does not take large values that would not allow the convergence of the drones.

Parameter	Value
perception_radius	10.0
max_agents	10
migration_gain	2.0
c_vm	15
a	1
b	5
d	2.0
delta	0.2
k	1
r	10
r0	4
c_pm_obs	5

Table 6: *Simulation parameters for two Iris one pillar - Scenario based on Table 4*

According to figure (7), two Iris drones can avoid the pillar with minimum distance between the tip of Iris propeller and the pillar wall equal to $97.86 - 73.9 = 23.96cm$. Furthermore, the drones converge to the specified α-lattice distance for two agents equal to 2 meter at simulation time 75 seconds after approximately 45 seconds spent in the obstacle avoidance maneuver.

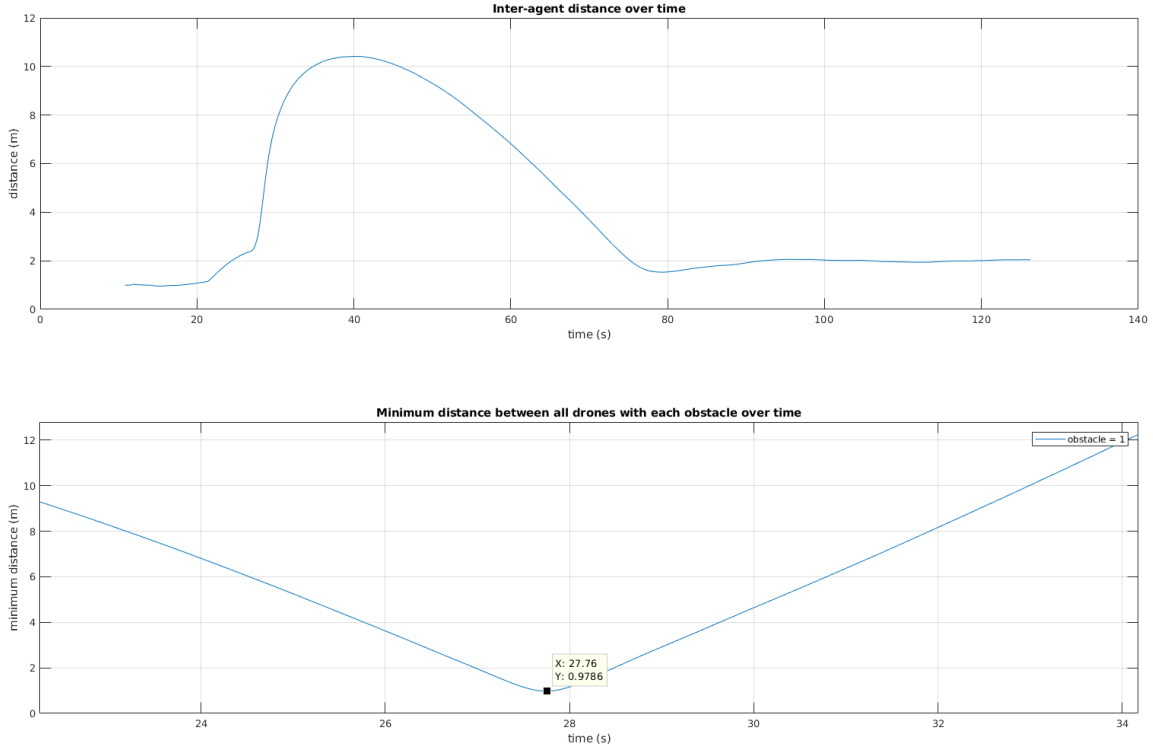

Figure 7: *Inter-agent and drone-obstacle distance over time for scenario and parameters based on Tables (4) and (6)*

6.1.3 Three Iris swarms obstacle avoidance from one pillar with γ-agent

The result of the obstacle avoidance behaviour of three Iris drones from one pillar based on tuned parameters for two Iris on Table 6 and scenario described on Table 4 is shown on

figure (8). According to figure (8), drones cannot avoid the obstacle (minimum distance $= 60.02cm < 73.9cm$) with the tuned parameters for two Irises. This happens because there is an interconnection between acceleration for keeping the flocking, obstacle avoidance and migration towards the target, and also because of the fact that the general scenario has been changed, it is required to modify the hyper-parameters to have the overall desirable behaviour. Furthermore, PX4 autopilot requires time to be able to apply the velocity command as according to the step velocity command experiment according to figure (9) the setting time for the velocity is 20 seconds.

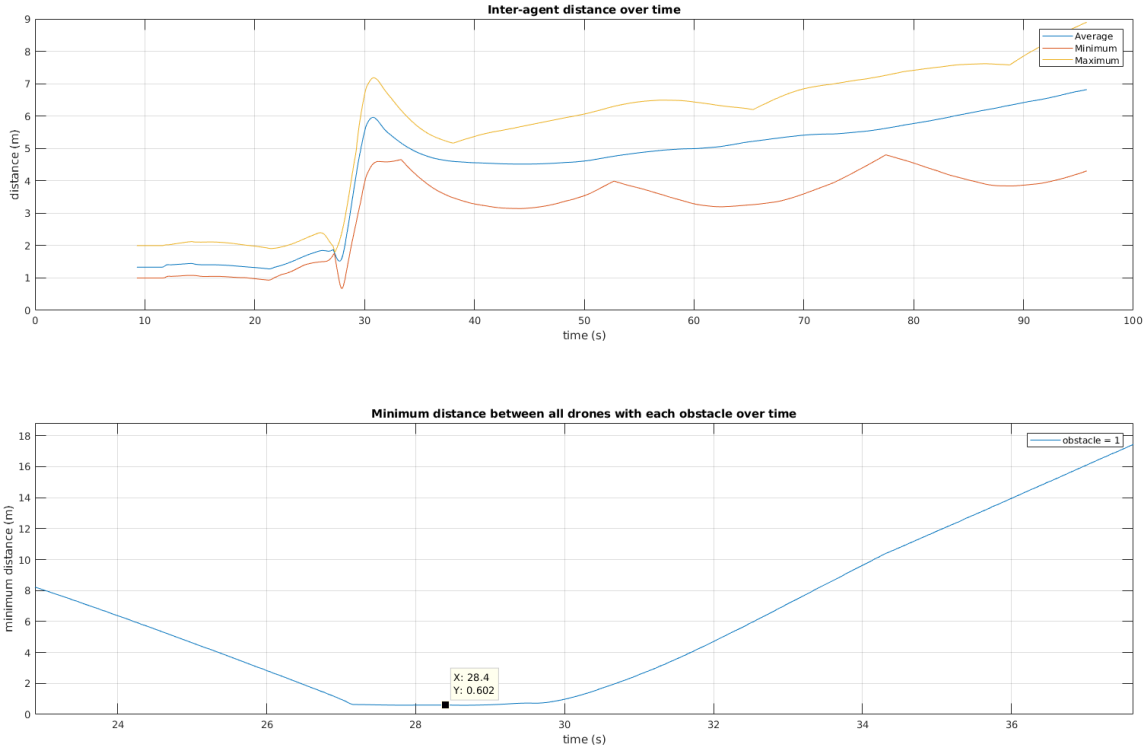

Figure 8: *Inter-agent and drone-obstacle distance over time for scenario and parameters based on Tables 4 and 6*

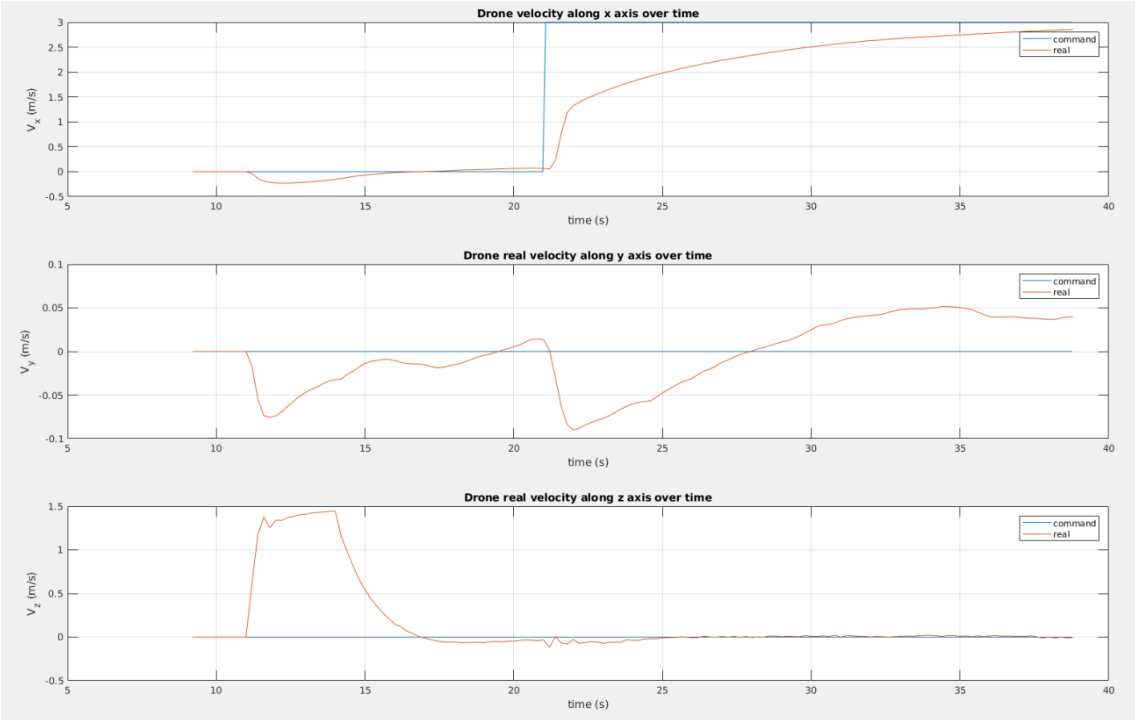

Figure 9: *Commanded step velocity on (x,y,z) direction to one Iris at PX4 o -board mode with PID Reference [1] vs Iris real velocity simulated on Gazebo*

Thereafter, the obstacle avoidance acceleration coe□cient is increased from 5 to 10, and the results shown in figure (10) are obtained. Figure (10) demonstrates that Iris avoids the pillar with distance $79.08 - 73.9 = 5.18cm$, but in spite of that drones cannot converge to the specified α-lattice distance on Table 6 which is $d = 2.0$ meters.

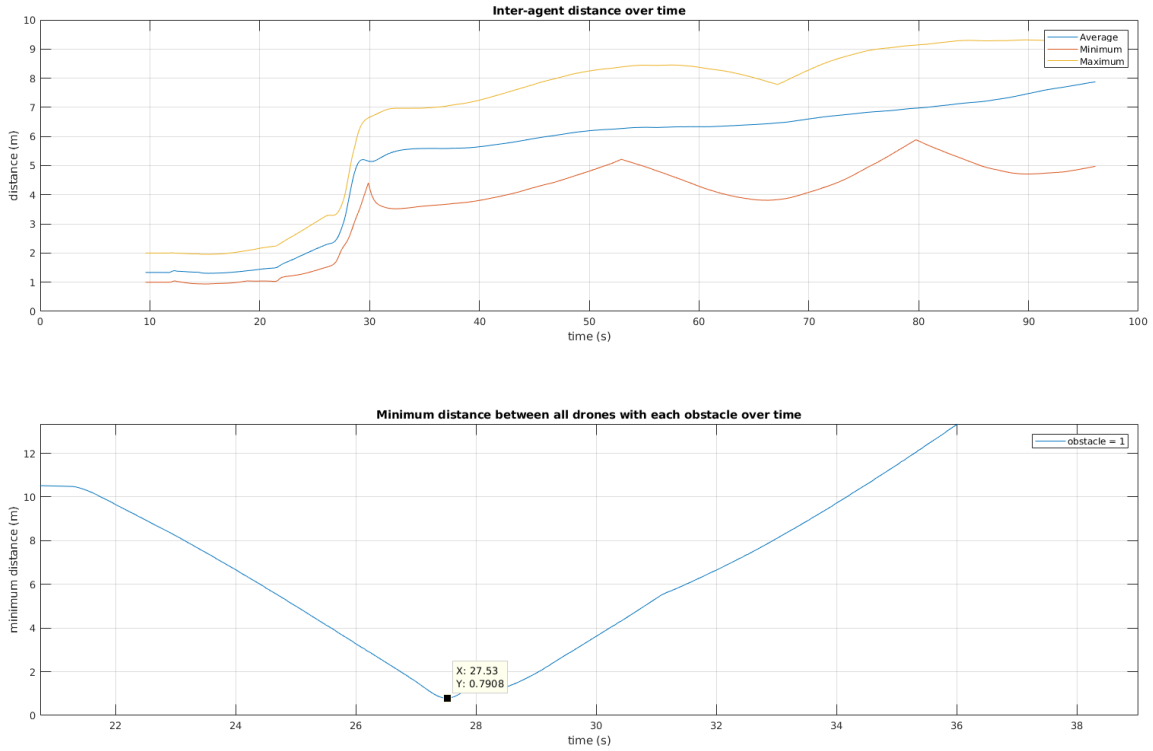

Figure 10: *Inter-agent and drone-obstacle distance over time for scenario and parameters based on Tables 4 and 6 but with* $c\_pm\_obs = 10$

In order to solve the convergence problem of the drones to the specified flocking inter-agent distance, the migration acceleration coe□cient c_vm is decreased from 15 to 5 to let the drones join each other after obstacle avoidance. However, the decrease in c_vm would cause the height of drones to decrease after switching from *take-o* to *o -board* mode which was still unknown and undesirable behaviour. This behaviour might happen because of the influence of PX4 PID controller which causes slightly steady state error that influences the performance of the swarm algorithm. So in order to not have this misbehaviour on z axis, the velocity command on z channel is set to zero and the results on figure (11) based on parameters on Table 7 is obtained.

Parameter	Value
perception_radius	10.0
max_agents	10
migration_gain	2.0
c_vm	5
a	1
b	5
d	2.0
delta	0.2
k	1
r	10
r0	4
c_pm_obs	10

Table 7: *Simulation parameters for two iris one pillar - Scenario based on Table 4*

According to figure (11), three Iris drones successfully avoid the obstacle with minimum distance $82.94 - 73.9 = 9.04\ cm$. Also, this figure shows that the quadcopters converge to the specified inter-agent distance on Table 7 which is $d = 2.0\ m$ at simulation time $90\ sec$ after approximately 50 seconds that the drones hove avoided the pillar.

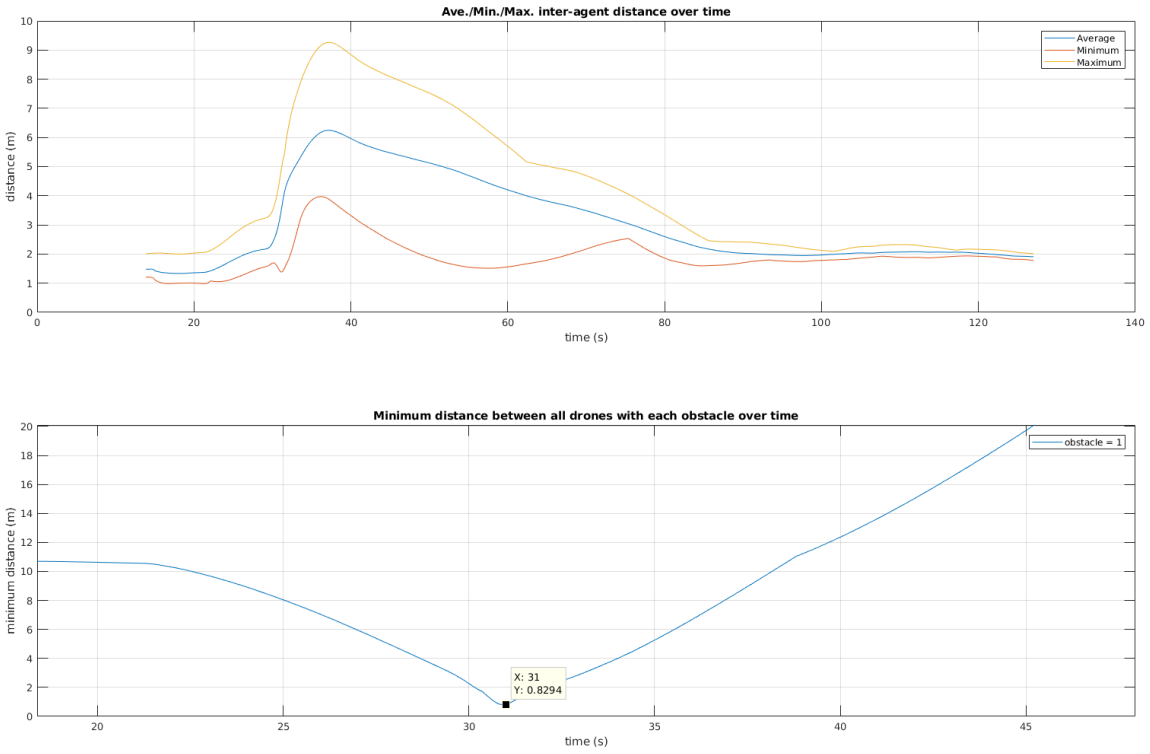

Figure 11: *Inter-agent and drone-obstacle distance over time for scenario and parameters based on Tables 4 and 7*

7 Conclusion

According to the obstacle avoidance maneuver results of the Iris quadcopters, it is concluded that for each scenario the hyper parameters of olfati_saber algorithm require tuning. Furthermore, the Iris drones behave as are expected by changing any of hyper parameters of the algorithm. Consequently, it was demonstrated that by tuning the olfati_saber hyper parameters, three and two Iris quadcopters can successfully avoid a single pillar on their way point, and they maintain their specified distance from the flocking center after the obstacle avoidance. However, there are noise at all Gazebo sensors including IMU and GPS that prevents to have repeatable results at each simulation despite the fact that all simulation parameters are kept the same.

8 Appendix

8.1 Code of plotting rosbag contents

```
%load bag file and read its messages
clear all
close all
clc
sample=100;%the step size to ignore messages received in between
%[e.g. sample=1 means all messages are read]
% load the bag file
bag = rosbag('3_iris_1_pillar_1_2.bag')
bag.AvailableTopics
%bag.MessageList
bagselect = select(bag, 'Topic', '/gazebo/model_states');
%start = bag.StartTime
%bagselect3 = select(bag, 'Time', [start+1 start + 3], 'Topic', '/gazebo/
    model_states')
msgs = readMessages(bagselect,1:sample:bagselect.NumMessages);
%msgs2=msgs{2}
%%
%drones and obstacles position vector extraction
num_drones=3;%number of drones
num_obs=1; %number of obstacles
r_d=zeros(length(msgs),3,num_drones);
for d=(1+num_obs+1):(1+num_obs+num_drones)
    r_d(:,1,d-1-num_obs)=cellfun(@(m) double(m.Pose(d, 1).Position.X),msgs);
    r_d(:,2,d-1-num_obs)=cellfun(@(m) double(m.Pose(d, 1).Position.Y),msgs);
    r_d(:,3,d-1-num_obs)=cellfun(@(m) double(m.Pose(d, 1).Position.Z),msgs);
end
r_ob=zeros(length(msgs),3,num_obs);
for ob=(1+1):(1+num_obs)
%     TO DO: receive obstacle positions from Gazebo
%     r_ob(:,1,ob-1)=cellfun(@(m) double(m.Pose(ob, 1).Position.X),msgs);
%     r_ob(:,2,ob-1)=cellfun(@(m) double(m.Pose(ob, 1).Position.Y),msgs);
%     r_ob(:,3,ob-1)=cellfun(@(m) double(m.Pose(ob, 1).Position.Z),msgs);
    r_ob(:,1,ob-1)=10.;
    r_ob(:,2,ob-1)=1.;
    r_ob(:,3,ob-1)=5.;
end
%%
%distance calculation and plot over time
if num_drones==1 %we exclude when we have only one drone for finding nchoosek
    [combination]
    drone_combinations=[1 1];
else
    drone_combinations=nchoosek(1:num_drones,2);
end
dot_drones_obstacles=zeros(length(msgs),3,num_drones,num_obs);
dot_drones = zeros(length(msgs),3,size(drone_combinations,1));
distance_drones_obstacles=zeros(length(msgs),num_drones,num_obs);
distance_drones=zeros(length(msgs),size(drone_combinations,1));
min_distance=zeros(length(msgs),num_drones,num_obs);
max_distance=zeros(length(msgs),num_drones,num_obs);
mean_distance=zeros(length(msgs),num_drones,num_obs);
min_distance_AllDrones_EachObstacle=zeros(length(msgs),num_obs);
leg33=[];%used for plot legends
leg22=[];
leg44=[];
for k=1:size(drone_combinations,1)
```

```
55      dot_drones(:,:,k)=[r_d(:,:,drone_combinations(k,1))-r_d(:,:,
        drone_combinations(k,2))];
56      for i=1:length(msgs)
57          distance_drones(i,k)=norm(dot_drones(i,:,k));
58      end
59
60  end
61  figure(1)
62  subplot(2,1,1)
63  plot(bag.MessageList.Time(1:100:bagselect.NumMessages),mean(distance_drones
        (:,:),2))
64  hold on
65  plot(bag.MessageList.Time(1:100:bagselect.NumMessages),min(distance_drones
        (:,:),[],2))
66  plot(bag.MessageList.Time(1:100:bagselect.NumMessages),max(distance_drones
        (:,:),[],2))
67  legend('Average','Minimum','Maximum')
68  title('Ave./Min./Max. inter-agent distance over time')
69  xlabel('time (s)')
70  ylabel('distance (m)')
71  grid on
72  for ob=1:num_obs
73      for d=1:num_drones
74          dot_drones_obstacles(:,:,d,ob)=[r_d(:,:,d)-r_ob(:,:,ob)];
75          dot_drones_obstacles(:,3,d,ob)=0;%here it is assumed obsticales are
76  %         pillar so z coordinate of drones are always equal to the obstacle
77          for i=1:length(msgs)
78              distance_drones_obstacles(i,d,ob)=norm(dot_drones_obstacles(i,:,d
        ,ob));
79              min_distance(i,d,ob)=min(distance_drones_obstacles(1:i,d,ob));
80              max_distance(i,d,ob)=max(distance_drones_obstacles(1:i,d,ob));
81              mean_distance(i,d,ob)=mean(distance_drones_obstacles(1:i,d,ob));
82          end
83      end
84      min_distance_AllDrones_EachObstacle(:,ob)=min(distance_drones_obstacles
        (:,:,ob),[],2);
85      figure(1)
86      subplot(2,1,2)
87      plot(bag.MessageList.Time(1:100:bagselect.NumMessages),
        min_distance_AllDrones_EachObstacle(:,ob))
88      leg3=['obstacle = ' '' num2str(ob) ''];
89      leg33=strvcat(leg33,leg3);
90      hold on
91      title('Minimum distance between all drones with each obstacle over time')
92      xlabel('time (s)')
93      ylabel('minimum distance (m)')
94      grid on
95  end
96  figure(1)
97  subplot(2,1,2)
98  legend(leg33)
99
100 %%
101 %read commanded velocity message messages
102 bagselect_velocity_command = select(bag, 'Topic', '/iris_1/geometry/velocity'
        );
103 msgs_velocity_command = readMessages(bagselect_velocity_command,1:1:
        bagselect_velocity_command.NumMessages);
104
105 bagselect_velocity_obstacle = select(bag, 'Topic', '/iris_1/geometry/
        velocity_obstacle');
```

```
106 msgs_velocity_obstacle = readMessages(bagselect_velocity_obstacle,1:1:
        bagselect_velocity_obstacle.NumMessages);
107 bagselect_velocity_obstacle = select(bag, 'Topic', '/iris_1/geometry/
        velocity_obstacle');
108 msgs_velocity_obstacle = readMessages(bagselect_velocity_obstacle,1:1:
        bagselect_velocity_obstacle.NumMessages);
109 bagselect_velocity_potential = select(bag, 'Topic', '/iris_1/geometry/
        velocity_potential');
110 msgs_velocity_potential = readMessages(bagselect_velocity_potential,1:1:
        bagselect_velocity_potential.NumMessages);
111 bagselect_velocity_matching = select(bag, 'Topic', '/iris_1/geometry/
        velocity_matching');
112 msgs_velocity_matching = readMessages(bagselect_velocity_matching,1:1:
        bagselect_velocity_matching.NumMessages);
113
114 %
115 %drones velocity vector extraction
116 v_d=zeros(length(msgs),3,num_drones);
117 v_d_command=zeros(length(msgs_velocity_command),3,num_drones);
118 v_d_obstacle=zeros(length(msgs_velocity_command),3,num_drones);
119 v_d_potential=zeros(length(msgs_velocity_command),3,num_drones);
120 v_d_matching=zeros(length(msgs_velocity_command),3,num_drones);
121 for d=(1+num_obs+1):(1+num_obs+num_drones)
122     d_h=d-num_obs-1;
123     v_d(:,1,d-1-num_obs)=cellfun(@(m) double(m.Twist(d, 1).Linear.X),msgs);
124     v_d(:,2,d-1-num_obs)=cellfun(@(m) double(m.Twist(d, 1).Linear.Y),msgs);
125     v_d(:,3,d-1-num_obs)=cellfun(@(m) double(m.Twist(d, 1).Linear.Z),msgs);
126     v_d_command(:,1,d_h)=cellfun(@(m) double(m(d_h, 1).X),
        msgs_velocity_command);
127     v_d_command(:,2,d_h)=cellfun(@(m) double(m(d_h, 1).Y),
        msgs_velocity_command);
128     v_d_command(:,3,d_h)=cellfun(@(m) double(m(d_h, 1).Z),
        msgs_velocity_command);
129     v_d_obstacle(:,1,d_h)=cellfun(@(m) double(m(d_h, 1).X),
        msgs_velocity_obstacle);
130     v_d_obstacle(:,2,d_h)=cellfun(@(m) double(m(d_h, 1).Y),
        msgs_velocity_obstacle);
131     v_d_obstacle(:,3,d_h)=cellfun(@(m) double(m(d_h, 1).Z),
        msgs_velocity_obstacle);
132     v_d_potential(:,1,d_h)=cellfun(@(m) double(m(d_h, 1).X),
        msgs_velocity_potential);
133     v_d_potential(:,2,d_h)=cellfun(@(m) double(m(d_h, 1).Y),
        msgs_velocity_potential);
134     v_d_potential(:,3,d_h)=cellfun(@(m) double(m(d_h, 1).Z),
        msgs_velocity_potential);
135     v_d_matching(:,1,d_h)=cellfun(@(m) double(m(d_h, 1).X),
        msgs_velocity_matching);
136     v_d_matching(:,2,d_h)=cellfun(@(m) double(m(d_h, 1).Y),
        msgs_velocity_matching);
137     v_d_matching(:,3,d_h)=cellfun(@(m) double(m(d_h, 1).Z),
        msgs_velocity_matching);
138 end
139
140 %%
141 %plot command and real velocity of one drone
142 figure(5)
143 subplot(3,1,1)
144 plot(bagselect_velocity_command.MessageList.Time,v_d_command(:,1,1))
145 hold on
146 plot(bagselect_velocity_obstacle.MessageList.Time,v_d_obstacle(:,1,1))
147 hold on
```

```
plot(bagselect_velocity_potential.MessageList.Time,v_d_potential(:,1,1))
hold on
plot(bagselect_velocity_matching.MessageList.Time,v_d_matching(:,1,1))
hold on
plot(bagselect.MessageList.Time(1:sample:bagselect.NumMessages),v_d(:,1,1))
legend('command','obstacle','potential','matching','real')
title('Drone velocity along x axis over time')
xlabel('time (s)')
ylabel('V_x (m/s)')
grid on
subplot(3,1,2)
plot(bagselect_velocity_command.MessageList.Time,v_d_command(:,2,1))
hold on
plot(bagselect_velocity_obstacle.MessageList.Time,v_d_obstacle(:,2,1))
hold on
plot(bagselect_velocity_potential.MessageList.Time,v_d_potential(:,2,1))
hold on
plot(bagselect_velocity_matching.MessageList.Time,v_d_matching(:,2,1))
hold on
plot(bagselect.MessageList.Time(1:sample:bagselect.NumMessages),v_d(:,2,1))
legend('command','obstacle','potential','matching','real')
title('Drone real velocity along y axis over time')
xlabel('time (s)')
ylabel('V_y (m/s)')
grid on
subplot(3,1,3)
plot(bagselect_velocity_command.MessageList.Time,v_d_command(:,3,1))
hold on
plot(bagselect_velocity_obstacle.MessageList.Time,v_d_obstacle(:,3,1))
hold on
plot(bagselect_velocity_potential.MessageList.Time,v_d_potential(:,3,1))
hold on
plot(bagselect_velocity_matching.MessageList.Time,v_d_matching(:,3,1))
hold on
plot(bagselect.MessageList.Time(1:sample:bagselect.NumMessages),v_d(:,3,1))
legend('command','obstacle','potential','matching','real')
title('Drone real velocity along z axis over time')
xlabel('time (s)')
ylabel('V_z (m/s)')
grid on
```

8.2 Code of Olfati Saber Flocking with Obstacle Avoidance Algorithm

```
import numpy as np

gain = {
  # Coeff for velocity matching
   'c_vm'   : 5,

   'a'      : 1,
   'b'      : 5,
   'd'      : 2.0,
   'delta'  : 0.2,
   'k'      : 1,
   'r'      : 100,

  # Velocity of migration    replace the velocity matching and it uses the
  # same gain P.c_cm
   'v_migration' : 2.0,

```

```
18    # Obstacles parameters
19    'r0' : 4,
20    'lambda' : 1,
21    'c_pm_obs' : 10,
22  }
23  (positions=rel_pos,
24                                                    my_velocity=my_vel,
25                                                    velocities=rel_vel,
26                                                    migration=migration,
27                                                    obs_positions=rel_obs_pos
      ,
28                                                    obs_velocities=
      rel_obs_vel,
29                                                    model_params=self.
      params_olfati_saber,
30                                                    sim_params=self.
      params_simulation,intru_pos=intru_pos,my_pos=my_pos)
31
32  gain['c']     = (gain['b']-gain['a'])/(2*np.sqrt(gain['a']*gain['b']))
33  gain['c_vm_obs'] = gain['c_vm']
34
35
36  def olfati_saber(positions, my_velocity, velocities=None, migration = None,
      obs_positions=None, obs_velocities=None, model_params=None, sim_params=
      None, intru_pos=None, my_pos=None):
37    """Olfati Saber flocking with obstacle avoidance algorithm for multiple
      agents
38
39    Args:
40      positions: Relative positions of each alpha agent to other alpha agents.
41      my_velocity: velocity of alpha agent
42      velocities: Relative velocities of each alpha agent to other alpha agents
      .
43      migration: unit vector towards direction from flocking center to the
      target point
44      obs_positions: Relative position of each alpha-agent from each obstacles
45      obs_velocities: Relative velocity of each alpha-agent from each obstacles
46      model_params: List of Olfati Saber parameters
47        - c_vm
48        - a
49        - b
50        - c
51        - delta
52        - k
53        - d
54        - r0
55        - lambda
56        - c_pm_obs
57      sim_params : List of simulation parameters
58        - perception_radius : Scalar metric distance
59        - max_agents : Maximum number of neighbors to include
60      my_pos: Position of alpha agent
61      intru_pos: Position of obstacle
62
63    Returns:
64      command: Acceleration command.
65
66    """
67    # Init variables
68    acc_potential = np.array([0, 0, 0])
69    acc_vel_matching = np.array([0, 0, 0])
```

```
70    acc_obstacles =np.array([[0., 0., 0.]])
71
72    positions = np.array(positions)
73    if velocities is None:
74      velocities = np.zeros_like(positions)
75    else:
76      velocities = np.array(velocities)
77    num_agents, dims = positions.shape
78
79    indices = np.arange(num_agents)
80    # Filter agents by metric distance
81    distances = np.linalg.norm(positions, axis=1)
82
83    if sim_params['perception_radius'] is not None:
84      indices = distances < sim_params['perception_radius']
85      distances = distances[indices]
86      positions = positions[indices]
87      velocities = velocities[indices]
88
89    # Filter agents by topological distance
90    if sim_params['max_agents'] is not None:
91      indices = distances.argsort()[:sim_params['max_agents']]
92      distances = distances[indices]
93      positions = positions[indices]
94      velocities = velocities[indices]
95
96    # Compute Olfati Saber flocking only if there is an agent in range
97    if len(distances) != 0:
98      unit_vect = (positions.T / distances).T
99      acc_potential = ( unit_vect.T * phi( distances, model_params ) ).T
100
101   # Compute the acceleration for Migration
102   if migration is not None:
103     acc_vel_matching = model_params['c_vm'] * (migration * model_params['
      v_migration'] - my_velocity)
104
105   # Compute acceleration for obstacle avoidance
106   if obs_positions is not None and obs_velocities is not None:
107     obs_positions = np.array(obs_positions)
108     obs_velocities = np.array(obs_velocities)
109     intru_pos=np.array(intru_pos)
110     my_pos=np.array(my_pos)
111     obs_distances = np.linalg.norm(obs_positions, axis=1)
112
113     obs_indices = obs_distances < model_params['r0']
114     obs_distances = obs_distances[obs_indices]
115     obs_velocities = obs_velocities[obs_indices]
116
117     # If at least one obstacle is in range
118     if len(obs_distances) != 0:
119       #print(obs_distances) #1
120       #print(obs_velocities) #3
121
122       res_rho = rho( obs_distances / model_params['r0'], model_params )
123       res_phi = phi( obs_distances-model_params['d'], model_params )
124       unit = (obs_positions.T/obs_distances).T
125
126       gamma_x=intru_pos+model_params['lambda']*migration * model_params['
      v_migration']/np.linalg.norm(migration * model_params['v_migration'])
127       gamma_x=[np.append(gamma_x[:,:2],0.)]
128       d_ag=np.linalg.norm(gamma_x-my_pos)
```

```
129         acc_obstacles += +model_params['c_pm_obs'] * res_rho * res_phi * unit +
        phi(d_ag-model_params['d'],model_params)*(np.append((gamma_x-my_pos)
        [:,:2],0.))/d_ag
130         #acc_obstacles += model_params['c_vm_obs'] * obs_velocities
131
132     #print(acc_potential.sum(axis=0), acc_obstacles.sum(axis=0),
        acc_vel_matching)
133     #Return the acceleration command
134     return acc_potential.sum(axis=0), acc_obstacles.sum(axis=0),
        acc_vel_matching
135
136 def psi(z, P):
137     """Psi function, potential
138
139     Args:
140         z: Point to evaluate the potential function
141         model_params: List of Olfati Saber parameters
142
143     Returns:
144         scalar : potential at given z point
145
146     """
147     return ((P['a'] + P['b']) * (np.sqrt(1 + (z - P['d'] + P['c'])**2) - np.
        sqrt(1 + P['c'] ** 2)) + (P['a'] - P['b']) * (z - P['d'])) / 2
148
149 def psi_der(z, P):
150     """Psi derivative function, derivative potential
151
152     Args:
153         z: Point to evaluate the potential function
154         model_params: List of Olfati Saber parameters
155
156     Returns:
157         scalar : Derivative of the potential at given z point
158
159     """
160     return (P['a'] + P['b']) / 2 * (z - P['d'] + P['c']) / np.sqrt(1 + (z - P['
        d'] + P['c'])**2) + (P['a'] - P['b']) / 2;
161
162 def rho(z, P):
163     """Rho - Function defining the adjacency coefficients
164
165     Args:
166         z: Point to evaluate the function
167         model_params: List of Olfati Saber parameters
168
169     Returns:
170         scalar : Function at given z point
171
172     """
173     #Init return value with zeros
174     adj_coef = np.zeros_like(z)
175     #Apply only on indexes where the condition is match
176     indice_inf = z < P['delta'] * P['r']
177     indice_sup = z < P['r']
178     indice_sup = indice_sup & ~indice_inf
179
180     adj_coef[indice_inf] = 1
181     adj_coef[indice_sup] = (1./2**P['k']) * ( 1+np.cos( np.pi*(z[indice_sup]/P[
        'r'] - P['delta'])/(1 - P['delta']) ) ) ** P['k']
182     return adj_coef
```

```
183
184 def rho_der(z, P):
185   """Derivative of Rho
186
187   Args:
188     z: Point to evaluate the function
189     model_params: List of Olfati Saber parameters
190
191   Returns:
192     scalar : Derivative of the function at given z point
193
194   """
195   #Init return value with zeros
196   adj_coef = np.zeros_like(z)
197   #Apply only on indexes where the condition is match
198   indice_inf = z < P['delta'] * P['r']
199   indices = z < P['r']
200   indices = indices & ~indice_inf
201
202   arg = np.pi*(z[indices]/P['r'] - P['delta'])/(1 - P['delta'])
203   adj_coef[indices] = -np.pi/(1 - P['delta'])*P['k']/(2**P['k'])*(1+np.cos(
      arg))**(P['k']-1)*(np.sin(arg))
204
205   return adj_coef
206
207 def phi(z, P):
208   """Psi function, force defining the attraction/repulsion as function of the
       distance
209
210   Args:
211     z: Point to evaluate the force
212     model_params: List of Olfati Saber parameters
213
214   Returns:
215     scalar : Force at given z point
216
217   """
218   return 1 / P['r'] * rho_der(z, P) * psi(z, P) + rho(z, P) * psi_der(z, P)
```

References

[1] Airframes reference. [Online]. Available: https://docs.px4.io/v1.9.0/en/airframes/airframe_reference.html

[2] Crazyflie 2.0. [Online]. Available: https://www.bitcraze.io/crazyflie-2/

[3] S.-J. C. Daniel Morgan, Giri P Subramanian and F. Y. Hadaegh, “Swarm assignment and trajectory optimization using variable-swarm, distributed auction assignment and sequential convex programming,” *The International Journal of Robotics Research*, vol. 35, 2016.

[4] F. S. Enrica Soria and D. Floreano, “The influence of limited visual sensing on the reynolds flocking algorithm,” 2018.

[5] E. A. Giuseppe Silano and L. Iannelli, “Crazys: a software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter,” *Mediterranean Conference on Control and Automation*, 2019.

[6] G. S. S. James A. Preiss, Wolfgang Honig and N. Ayanian, "Crazyswarm: A large nano-quadcopter swarm," *International Conference on Robotics and Automation (ICRA)*, 2017.

[7] J. Ma and E. M.-K. Lai, "Flock diameter control in a collision-avoiding cucker-smale flocking model," 2017.

[8] J. Noronha, "Development of a swarm control platform foreducational and research applications," *TRANSACTIONS ON ROBOTICS*, 2016.

[9] R. Olfati-Saber, "Flocking with obstacle avoidance: Cooperation with limited information in mobile networks," *Conference on Decision and Control*, 2003.

[10] ——, "Flocking for multi-agent dynamic systems: Algorithms and theory," *TRANSACTIONS ON AUTOMATIC CONTROL*, vol. 51, 2006.

[11] A. S. S.Iovino, "Implementation of a distributed flocking algorithm with obstacle avoidance capability for uav swarming," *AIAA Information Systems*, 2017.

[12] P. D. S. S. Soon-Jo Chung, Aditya Avinash Paranjape and V. Kumar, "A survey on aerial swarm robotics," *TRANSACTIONS ON ROBOTICS*, vol. 34, 2018.

[13] W. W. P. W. P. K. Wojciech Giernacki, Mateusz Skwierczyński, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," 2017.

[14] L. S. T. P. M. B. N. A. Wolfgang Honig, Christina Milanes, "Mixed reality for robotics," *International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[15] Y. W. X. W. Yinbo Xu, Yongwei Zhang, "Physical experimental realization of modified artificial physics method based on uavs formation control," *International Conference on Intelligent Human-Machine Systems and Cybernetics*, 2017.

[16] L. Zlajpa, "Simulation in robotics," *Mathematics and Computers in Simulation*, vol. 79, 2008.