Politecnico di Torino

Master of Science Degree in Mechatronics Engineering

MSc Thesis

Quadrotor UAV 3D Path Planning with Optical-Flow-based Obstacle Avoidance

Advisors:

Prof. Alessandro Rizzo

Dr. Ing. Kimon P. Valavanis

Candidate:

Giancarlo Allasia

Academic Year 2019/2020

Abstract

This thesis is the result of a six-months research activity conducted at University of Denver (DU) Unmanned Systems Research Institute (DU2SRI), Colorado, USA. The objective of the thesis is the design of a bio-inspired 3D local path planning strategy for obstacle avoidance based on the optical flow field obtained by a frontal monocular camera mounted on a quadrotor UAV, with the requirement of being real-time implementable and runnable by onboard embedded hardware. Assuming the quadrotor is controlled in position and its path is defined in term of a list of waypoints, the strategy consists in generating an intermediate waypoint in order to avoid obstacles based on optical flow horizontal and vertical unbalance for horizontal and vertical avoidance, while for frontally approaching obstacles avoidance the concept of expansion of optical flow field is used. The algorithm is implemented in Robotic Operative System (ROS) as a ROS node, where aforementioned quantities are generated by means of OpenCV Python library. Everything is tested in a simulated environment run by Gazebo in three different scenarios representing horizontal, vertical and frontal avoidance and both software-in-the-loop and hardware-in-the-loop tests are carried out. Performance of the algorithm is measured with software profiling and in terms of execution time for two embedded computer boards such as NVIDIA Jetson TX1 and RaspberryPi 4 by running ROS node on them. The tests show the effectiveness of the algorithm in avoiding obstacles in all scenarios and the capability of the algorithm to be run in real time at a frequency of at least 5 Hz on onboard mountable hardware.

Dedication

To my mother Mariella and my father Franco.

Declaration

I declare that this thesis presented for the degree of Master of Science in Mechatronics Engineering was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

Acknowledgements

Being able to do my master thesis in the United States of America is a dream I had since a long time. But being able to spend six months in the US completing this job and also having the opportunity to explore the country in the meanwhile during one of the toughest world pandemics caused by COVID-19, is certainly something that cannot be achieved without luck, stubborness and most of all the support of some people. In this part, I want to thank those people that made this possible and those who supported me.
I want to thank professor Rizzo, who put me in touch with professor Valavanis from Denver University and started the collaboration, giving me this remarkable opportunity. Thanks to Kimon, not only for the opportunity he gave me, but also for his typical mediterranean hospitality and kindness that really made me and Giuseppe feel at home during our stay in Denver. My parents, Franco and Mariella, deserve special thanks for the support, both financial and emotional, guessing how concerning it was to have son far from home in a foreign country during a pandemic. Thanks to Giuseppe, roommate, thesis program colleague and friend, whose companionship has been precious for my stay in Denver, the road trip we had to California and back and the last weeks in New York City. Finally, thanks to all my friends that during these college years and especially during the pandemic with video calls always supported me and have made these years special.

Audaces fortuna iuvat.

Contents

Abbreviations

DOF	Degrees Of Freedom
ENAC	Ente Nazionale dell'Aviazione Civile
EKF	Extended Kalman Filter
EOF	Expansion of Optical Flow
FAA	Federal Aviation Administration
FOE	Focus Of Expansion
FOV	Field of view
GPS	Global Positioning System
GUI	Graphical User Interface
HIL	Hardware-In-the-Loop
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging o Laser Imaging Detection and Ranging
MMF	Moving Mean Filter
NAS	National Airspace System
NED	North-East-Down
OF	Optical Flow
OFOA	Optical Flow based Obstacle Avoidance

UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
POV	Point of view
PID	Proportional Integrative Derivative
RF	Reference Frame
ROI	Region Of Interest
ROS	Robotic Operative System
SIL	Software-In-the-Loop
TTC	Time-To-Contact or Time-To-Collision
WCET	Worst Case Execution Time

Chapter 1

Introduction

1.1 Introduction and rationale

In the last decades Unmanned Aerial Systems (UAS), systems which include multiple components such as an Unmanned Aerial Vehicle (UAV), a ground-based controller, and a system of communications between the two, have seen a huge increase in popularity. These systems have seen a growing interest because they allow to carry out a huge variety of missions and tasks that usually require manned flight with vehicles such as planes and helicopters, but at a fraction of the cost, and they have the potential of being employed to make humans more productive or to do new tasks that are too dangerous or impossible for unaided humans. Such types of tasks range through defense, monitoring of territories and environment, aerial photography, precision agriculture, waste management, urban surveillance, delivery (as an example, a further boost in delivery has been given in August 2020 by the approval by Federal Aviation Administration (FAA) to Amazon for its drone delivery service Amazon Prime Air [1]), just to name a few.
Depending on the application, some types of UAVs can be more suitable than others: when wide areas must be covered at high speed and for longer times, fixed-wing type is usually the addressed type; when the application deals with an indoor environment or hovering capability and more flexibility are required, multirotor vehicles are the way to go (examples in Figure 1.1).

Multirotors have also become very popular in last years on consumer market: while in the past they were seen just as expensive robots used in research and intended for professional use, nowadays they can be found in toy shops as a sort of cheap flying cameras, available to a broader public of customers.
Due to this pervasiveness, main institutions regularizing flight in different countries, such as FAA for USA or the italian ENAC (Ente Nazionale dell'Aviazione Civile), had to set up rules, in order to better mark the difference between amateur use from professional applications and discourage improper use. A classification of UAVs has been then required and different ones are proposed depending on considered parameters such as takeoff weight, size, operating altitude, and airspeed; as an example, the U.S. Department of Defense (D.o.D) categorizes UAVs in the five

(a)

(b)

Figure 1.1: Examples of fixed-wing type and multirotor type UAVs. Figure 1.1a shows the General Atomics MQ-1 Predator, a military grade fixed-wing type UAV used by United States Air Force (USAF) and Central Intelligence Agency (CIA), mounting a set of sensors, a camera and designed to carry missiles; it entered in service in 1995 and was used in many conflicts in Middle East. Figure 1.1b shows instead a multirotor type of UAV, for instance an octacopter produced by the italian Italdron, here shown with a cinema camera as a payload for taking professional aerial shots.

different groups shown in Figure 1.2.

UAS Group	Maximum weight (lb) (MGTOW)	Nominal operating altitude (ft)	Speed (kn)	Representative UAS
Group 1	0–20	< 1,200 AGL	100	RQ-11 Raven, WASP,
Group 2	21–55	< 3,500 AGL	< 250	ScanEagle, Flexrotor, SIC5
Group 3	< 1,320	< FL 180		RQ-7B Shadow, RQ-21 Blackjack, Navmar RQ-23 Tigershark, Arcturus-UAV Jump 20, Arcturus T-20, AATI Resolute Eagle, SIC25
Group 4	> 1,320		Any airspeed	MQ-8B Fire Scout, MQ-1A/B Predator, MQ-1C Gray Eagle
Group 5		> FL 180		MQ-9 Reaper, RQ-4 Global Hawk, MQ-4C Triton

Figure 1.2: UAV classification according to U.S. Department of Defense, from [2]

Another classification, the one proposed by ENAC, groups UAVs in three categories based on the type of application, Certified, Specific and Open, better explained in Figure 1.3.

The regularization of UAV flight is also motivated by estimations about market evolution: in November 2019, Teal Group, a company into aerospace and defense industry market analysis, predicts in its 2019/2020 UAV Market Profile an increase from current worldwide military UAV production of 7.3 B$ annually in 2019 to 10.2 B$ in 2029, totaling 98.9 B$ in the next ten years [3].
Hence a safe integration of UAVs in the National Airspace System (NAS) is required and one of the main issues preventing that is obstacle and collision avoidance capability. As a PhD student from MIT's CSAIL said, "Everyone is building drones these days, but nobody knows how to get them to stop running into things", highlighting the necessity of finding solutions for obstacle avoidance implementation.
This is a hot research topic and a huge amount of proposals has been made, each one with its own peculiarities and applicability, not to mention cost. Obstacle avoidance

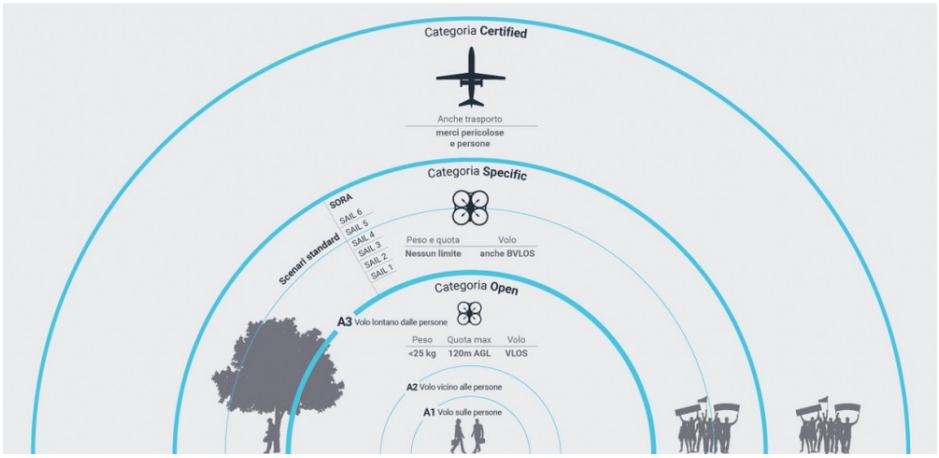


Figure 1.3: UAV classification according to italian ENAC.

requires an UAV to be aware of its surroundings so that in case of imminent collision it can take action to get out of a dangerous situation, for people in the area and then for itself. When it comes to environment perception for avoidance in robotics, most popular sensors are sonars, radars, laser scanners (e.g. LIDAR) and cameras. The most complete information for mapping the environment and then taking action to follow an obstacle-free path is coming from LIDAR, that is still very expensive and bulky, even if prices has lowered recently. Cameras, otherwise, are cheap and can sample a big amount of information that, of course, needs some elaboration to retrieve useful data depending on the application.
Vision on the other hand is undoubtedly a sense that allowed living beings endowed with it to retrieve information about surrounding world to navigate the environment, spotting dangerous situations and ultimately survive. Natural selection and evolution refined over time the working principle of vision techniques, hence the engineering field finds a lot of useful hints to cope with many problems by taking inspiration from nature. The study of flying insects' or birds' behaviour in navigation and interaction with the environment, for example, has revealed interesting facts: they navigate safely in the three-dimensional world by using cues derived more from image motion rather than complex stereo mechanisms such as we human beings usually do (even if we also use this visual motion perception in a lot of situations). In fact, the distance from their eyes is sometimes very small compared to human eyes, so possessing inferior spatial acuity the precision with which insects could estimate range through binocular stereo vision would be much poorer and restricted to relatively small distances; in other cases, eyes are not even point in the same direction, making impossible to retrieve depth data and hence finding perception from optical motion more effective objects in terms of apparent speed of motion of objects' images. Consequently, flight speed is regulated so that the average image velocity as seen by both the eyes is kept constant. For instance, flight speed is automatically lowered in cluttered environments, and thrust is adjusted to compensate for headwinds and tailwinds. As an example, bees landing on a horizontal surface hold the image velocity of the surface constant as they approach it, thus automatically

ensuring a smooth touchdown [4].
This optical movement in the field of view of the subject due to the relative motion between the object and the subject has been named *optical flow* and this concept was first in introduced by American psychologist James Jerome Gibson in the 1940's when describing how animals perceive their environments to be able to move around freely without collisions [5]. Optical flow of course attracted the computer vision research community and one of the first algorithms was proposed by Horn and Schunck in 1980 [6], that given two subsequent frames sampled by a monocular camera was able to estimate pixels motion among frames. A lot of popular applications in technology are based on this biological principle, for instance video compression codecs to interpolate between key frames (e.g. MPEG) or optical flow sensors (e.g. the one used in optical mice), just to name a couple.
This thesis aims hence at proposing a cost-effective solution for the obstacle avoidance problem exploiting this principle inspired by biological systems.

1.2 Thesis organization

This thesis is organized as following.
Chapter 2 presents the literature review carried out in the most comprehensive way on the most recent scientific literature produced about optical flow based obstacle avoidance for UAVs. At the of literature review, a classification of the covered papers is proposed, spotting their main limitations and, from them, a more specific goal for this thesis is set to overcome found limitations.
In Chapter 3 the stated problem is explained in depth. First, the kinematic and dynamic nonlinear model of a quadrotor is obtained, showing the whole procedure; the flight controller with position reference is covered too. Then, theory about optical flow and its mathematical model, existing algorithms and their common implementation is covered and concepts such as focus of expansion, time-to-contact and expansion of optical flow are introduced and their computation formalized. Finally, after reporting used software tools and motivating them, test scenarios and performance metrics, both for obstacle avoidance algorithm and software execution time, are defined.
Chapter 4 contains the explanation of my proposed solution to the problem, explaining its working principle both in a more abstract level and then detailing it in equation and algorithmic implementation.
Then, the proposed obstacle avoidance strategy is tested ans results are collected and commented in Chapter 5, where plots about avoidance effectiveness and statistics and measurements about software real-time execution, tested also on real embedded hardware that can be integrated onboard, are reported.
Finally, in Chapter 6 conclusions are drawn and possible future development is proposed.

Chapter 2

Literature review

A comprehensive literature review of the most recent research about monocular optical-flow-based obstacle avoidance strategies is carried out; among different methods, those based on optical flow will be considered. This topic has been covered by multiple researchers since the first mathematical formalization of this bio-inspired method showed up in the academic community.

In the following, most recent papers are covered and for each of them the main method proposed is summarized, reporting a meaningful insight, then its main limitations are pointed out, if found, and consequent further possible research path. The results are then collected and shown clearly in a table, where main features are pointed out.

2.1 Monocular optical-flow-based obstacle avoidance: literature review

In [7] (2015), Richards et al. present the development of an obstacle avoidance system on a small fixed-wing UAV by means of sparse optical flow algorithm (pyramidal Lucas-Kanade applied on feature points tracking obtained by Shi-Tomasi method) applied with monocular camera video stream. Stream is processed through a Intel NUC board, where the algorithm produces a probability matrix that calculates the safest region in the image the airplane can fly to. From this information, an avoidance command is sent via UART to an Arduino UNO that sends suitable PWM commands to the ArduPilot Mega 2.5. Mechanical relays are used to switch from manual command to autonomous obstacle avoidance. Further development can be identified in fusing multiple sensors (e.g. LiDAR, multiple cameras for expanding field of view) or implementing obstacle avoidance commands as intermediate waypoints sent over MAVLink to the flight controller).

In [8] (2019), Blumenkamp et al. propose a machine learning approach to obstacle avoidance based on optical flow, implemented on a toy race car. The images are sampled from a monocular camera connected to a RaspberryPi compressing its video stream with H.264/AVC that is then decoded by onboard GPU in real time,

where motion vector stream is made available through an API and used as optical flow field. Then this optical flow stream is the input of a deep neural network which aim is to classify obstacles and from it the corresponding steering command for obstacle avoidance is computed. The proposed approach has several limitations given by the experimental framework and the algorithm is developed to work only in 2D. Also, this work is more focused on the deep learning side and requires a heavy computation capability that is not bearable by RaspberryPi and requires off-board computing on a remote desktop, making the implementation sensitive to wireless communication performance.

In [9] (2018), Shankar et al. propose a novel algorithm for obstacle avoidance based on optical flow called Pyramid Histogram of Oriented Optical Flow (p-HOOF), where the sparse optical flow vector field is transformed by this descriptor in an histogram that holds orientation information of the field. Then a Support Vector Machine (SVM) classifier using p-HOOF is trained and used for real-time obstacle classification. To avoid obstacles, a behaviour-based collision avoidance adaptive mechanism is designed that dinamically updates the robot's response sensitivity using a prior update method. Experiments with a differential drive mobile robot controlled by RaspberryPi has been carried out, where frame capture rate is around 20 FPS and computation of optical flow vectors and p-HOOF descriptors takes 550-700 ms, so runs at a rate of 1.4 - 1.8 Hz. However, this algorithm is only tested in 2D with a wheeled robot.

In [10] (2018), Miao et al. use the classic Lucas-Kanade Optical Flow algorithm and the visual velocity field obtained as output is used to compute a visual artificial potential field, from which corresponding force field is computed and subsequently the yaw angle to offer a collision-free path. The potential field is composed by the superposition of an attractive part toward the goal and a repulsive part given by the obstacles and obtained by combining gradient vectors and Time-to-collision (TTC) that represents depth in terms of time. The obtained yaw angle is fed as input to the quadrotor control system implemented as a cascaded PID. Simulations are carried out on Matlab using Virtual Reality Modeling Language (VRML) to prove its effectiveness. However, this method only works in 2D without changing drone height.

In [11] (2017), Gao et al. propose an obstacle avoidance method based on optical flow using Farnebäck's dense optical flow. A comparison analysis is first done with EpicFlow, a novel optical flow approach targeted at large displacements with significant occlusions, concluding after experiments that is an order of magnitude slower than Farnebäck's dense optical flow, that is hence chosen for the proposed approach. Then, flow field produced is processed, eliminating flow where magnitude is too large, getting a greyscale image of magnitudes and then appropriately thresholding to distinguish obstacles from free space.Finally the target (i.e. obstacle) is detected with minimum rectangle bounding and its velocity is computed by summing all its pixels' flow values. The algorithm is tested with real experiments using ROS package *ardrone_autonomy* for controlling an AR.Drone Parrot and results state a success rate of 95% for drone speed below 1.5 m/s and the algorithm

works at a rate of 10 Hz.

In [12] (2005), Zufferey et al. present a control strategy enabling obstacle avoidance and altitude control based on a 1D simplification of optical flow and gyroscopic information. The approach first considers classical 2D optical flow behaviour by splitting in translation and rotation contributions, noticing that rotational one can be compensated by knowing gyroscopic rates, so that only useful translation component can be kept. Than, given the strict constraints of implementation on a 30-grams plane UAV, simplifies optical flow information from 2D to 1D in order to use a light linear CCD array, implementing a simplified computation of 1D version of divergence. This lets understand by thresholding if an obstacle is approaching and where to turn by flow unbalancing. Similar logic is applied for altitude control. Tests of this algorithm are conducted first on a small wheeled robot, then on the 30-gram fixed-wing UAV. The algorithm here applied is indeed way simplified for an effective application in the scope of this thesis.

In [13] (2014), Agrawal et al. propose an optical flow based composite guidance strategy for UAV navigation in unknown outdoor environments while seeking for a predefined goal point. The proposed guidance logic is based on the relative optical flow in the right, left and central regions of the image. With appropriate thresholding of average magnitudes of optical flow in the three aforesaid regions, inverse (i.e. heading rate proportional to the inverse of difference between right and left average flow), balancing (i.e. heading rate proportional to mentioned difference) and goal seeking strategies are appropriately selected. Simulations run with Virtual Reality Modeling Languages (VRML) toolbox from MATLAB validate the effectiveness of this composite approach. However, this approach is only working in a 2D fashion.

In [14] (2012), Eresen et al. propose an optical flow based autonomous flight algorithm for reaching a goal position by means of navigation in virtual urban environment. The UAV is a quadrotor type whose dynamics are modeled in Matlab, with the addition of PID controller for the attitude, and simulation is carried out by linking it to Google Maps virtual urban environment. From Google Maps images of streets and junctions in the city are processed and the optical flow field is computed with Horn and Schunk method. Only an horizontal strip of the image is considered and by means of a template the minimum flow magnitude region is identified in order to avoid collisions with buildings and a correcting yaw movement is computed. Also, junctions are recognized by noticing the different flow magnitudes on the right and left sides of the image and the yaw correction that minimizes distance to goal position is applied. Simulations show the effectiveness of the method. This method however only works in 2D at fixed altitude and its structure makes it most suitable for urban environment.

In [15] (2019) , Cho et al. propose an online optical-flow-based 3D obstacle avoidance strategy applied to a quadrotor vehicle able to cope with both lateral, vertical and frontal obstacles. The strategy deals with horizontal avoidance by considering the flow balance between the right and left half-planes of a considered horizontal strip, using it to compute by means of a PD controller the heading rate control signal, adequately compensated with terms that take into account the waypoint

guidance (as the difference of current heading and heading towards the next waypoint), avoidance of frontal obstacles and the compensation of yaw rate generated optical flow. As far as the vertical avoidance is concerned, similar strategy is used to generate altitude control, where another PD control is implemented based on both optical flow balance from lower and upper half-plane in the considered vertical strip and the waypoint guidance strategy. Frontal obstacles are detected by computing the divergence of the field from the focus of expansion (FOE) and from a geometrical approach time to contact (TTC) is evaluated for each point and by evaluating their balance the heading rate term in the PD control is evaluated. The different strategies then are integrated in a hybrid approach by weighting them with appropriate coefficients computed online according to the current situation, so that obstacle avoidance part can predominate over waypoint guidance part when obstacles are detected by optical flow conditions. Simulations are conducted by means of RotorS ROS/Gazebo simulator to prove the effectiveness of the method. Real test are carried out too with indoor Optitrack motion tracking by using an NVIDIA Jetson TK1 companion board for OpenCV computation at 8 Hz and set points generation, which are sent to the Pixhawk flight control board at 1 Hz frequency for obstacle avoidance. A limitation that can be found in this approach is the lack of compensation for pitch rate in the generated optical flow and further improvement can be made by adding a pre-processing phase where also feature and color detection are considered.

In [16] (2011), Yoo et al. propose a 2D optical-flow-based obstacle avoidance strategy for quadrotor tested in urban environments simulation. The approach is based on classical Lucas-Kanade optical-flow algorithm, where the optical flow field is used to apply a flow balance strategy: avoidance command will be a left turn if the sum of optical flow magnitudes on right half-plane is greater than the left one and vice versa. The approach is tested in MATLAB virtual reality environment, showing basic effectiveness in avoiding static obstacles and moving along a corridor environment. The algorithm runs at computation rate of 10 Hz. The proposed approach however is pretty limited due to its 2D nature and simple strategy, plus is tested only in basic simulated urban environment with static obstacles.

In [17] (2019), Aguilar et al. propose a monocular optical-flow-based method for dynamic obstacles detection and a fuzzy logic controller based avoidance strategy. First, two consecutive frames are sampled from the monocular camera, their absolute difference is computed, thresholded with respect to mean value of difference pixels, noise filtered and passed through Canny edge detector. Then the identified moving object is encapsulated by a single rectangle, within which features points are exctracted by Shi-Tomasi method and their motion is estimated as a vector field by means of Lucas-Kanade optical flow algorithm. Velocity in terms of pixels per second is computed and, with the assumption that moving points belong to the same objects, points with too different velocities with respect to their mean velocity are discarded. Then, template area evolution over the past 15 frames is approximated with a linear regression and the slope of the obtained line's equation is used to discriminate its growth rate, hence if the obstacle is approaching or moving away.

Finally, a fuzzy logic controller with as fuzzy inputs the area of the moving object and the approaching rate is designed by defining fuzzy rules in order to output a suitable speed control command along x-axis with the aim of mantaining the moving object far from the UAV, letting the control being strong when object is approaching too fast and softer when it approaches moderately. Algorithm is tested with different moving objects at different distances, but test details are not clearly pointed out, as well as algorithm effectiveness. The computation is implemented offboard on a gorund control station at real-time rates. The proposed approach has several limitations, being the control output a 1D control command (representing just the strenght of the avoidance control along one axis) computed offboard. Furthermore, this approach seems not to be able to track and handle multiple obstacles and direct tests of UAV obstacle avoidance doesn't seem to have been conducted.

In [18] (2015), Wang et al. propose an obstacle avoidance algorithm using an improved method based optical flow. Based on the assumption of a Kalman filter stabilized quadrotor to minimize vibrations, the captured frames are divided into a 9 regions grid, where on the 5 laying on the cross cells are evaluated with dense optical flow algorithm, while the 4 in the corners are given less computational burden by applying sparse optical flow algorithm and used as a reference. The final control command along both vertical and horizontal directions is computed as the contribution of two techniques: the improved balance strategy and the time-to-contact (TTC) strategy. The former, considers horizontal and vertical unbalances in optical flow magnitudes in considered cross areas, adding a term relative to respective corner areas flow. The latter, computes the focus of expansion (FOE), filters its position by means of moving mean (MM) method in order to reduce the high-frequency noise and for each pixel computes its TTC based on its distance from FOE and OF magnitude in that point, producing a qualitative relative-depth estimation of the scene. The improved balance method is then abstracted by training, where manual control avoidance maneuvers are performed and respective OF conditions are compared and plotted, so that a classification function with 3 different regions in which OF measurements fall into can be designed and used to produce appropriate control command. The algorithm is tested with real experiments by means of Parrot AR.Drone controlled by ROS in indoor environment, showing a success rate of 75%. The proposed method however does not consider frontal obstacles and doesn't provide avoidance strategies for that case.

2.1.1 Classification

In order to have a more systematic overview of optical flow based obstacle avoidance methods proposed up to now, the reviewed papers are classified according to different parameters, as can be seen in table 2.1.

Table 2.1: Classification of developed monocular optical-flow based obstacle avoidance methods. The columns represent the referenced work with its publication year (Ref.), the type of robot intended for algorithm application (Robot), the optical flow algorithm used (OF Method), the dimensionality of obstacle avoidance control commands produced (2D/3D), validation of achieved results (Result) and main limitations (Limitations) classified as follows. A: 2D Obstacle avoidance (left/right control commands), B: Offboard computation, C: Narrow applicability (e.g. specifically structured environment, intrinsically limited), D: Non real-time implementable (or not specified), E: Doesn't deal with frontal obstacles.

Ref.	**Robot**	**OF Method**	**2D/ 3D**	**Results**	**Limitations**				
					A	B	C	D	E
[7] (2015)	Fixed-wing	Sparse Pyramidal OF	2D	SIM, EXP	✗				✗
[8] (2019)	Wheeled	H.264/AVC, Deep Neural Network	2D	EXP	✗	✗			
[9] (2018)	Wheeled	Sparse Pyramidal OF, p-HOOF, SVM classifier	2D	SIM, EXP	✗				
[10] (2018)	Quadrotor	Sparse Pyramidal OF, Visual APF	2D	SIM	✗	n/a		n/a	✗
[11] (2017)	Quadrotor	Dense Farnebäck OF	-	EXP	n/a	✗			✗
[12] (2005)	Wheeled/ Fixed-wing	Simplified 1D OF	2D	EXP	✗		✗		✗
[13] (2014)	Generic	Horn and Schunck OF	2D	SIM	✗	n/a	✗	✗	✗
[14] (2012)	Quadrotor	Horn and Schunck OF	2D	SIM	✗	n/a	✗	n/a	✗
[15] (2019)	Quadrotor	Horn and Schunck OF	3D	SIM, EXP					

Ref.	Robot	OF Method	2D/ 3D	Results	Limitations				
					A	B	C	D	E
[16] (2011)	Quadrotor	Lukas-Kanade OF	2D	SIM	✗		✗		✗
[17] (2019)	Quadrotor	Sparse Lukas-Kanade OF	1D	EXP	✗	✗		✗	✗
[18] (2015)	Quadrotor	Mixed dense and sparse OF	3D	EXP		✗			✗

2.2 Conclusions

As far as monocular optical-flow-based obstacle avoidance for UAV is concerned, five main limitations are spotted: obstacle avoidance carried out in 2D control commands at fixed height, computation carried out on offboard hardware (hence strongly relying on a good link with ground control station), intrinsic limitations in the method (i.e. strong environment assumptions, specific hardware based, very specific context application), non-real-time implementability (due to slow algorithm processing or not specifically addressed as an issue) and lack of dealing with frontally approaching obstacles.
What emerges from the set of reviewed articles is that the main two limitations most of them share is the 2D obstacle avoidance strategy, that supposes a fixed height flight and mainly moves in a plane parallel to ground to avoid lateral obstacles, hence this technique does not allow to avoid floating obstacles unless a particular controller for changing flight height is implemented, and frontal obstacles approaching to the vehicle are not taken into account, just considering lateral obstacles and their horizontal unbalanced contribution to overall optical flow map.
The other identified limitations are better tackled by some studies, while some others still suffer of them. For instance, as far as concerns narrow applicability, some proposed methods assume a urban structured environment or consider motion in a corridor [13, 14, 16]; others use a specific hardware sensor [12]. Computational limitation, both for being offboard or non-real-time, is an issue that not every proposed method shows, but still a part of them doesn't solve.
A further note has to be done about the computational complexity of proposed algorithms: only a few studies report measurements of execution time, while, at the best of my knowledge, no one of them carries out an analysis about computational cost.

Hence, this thesis aims at the attempt of overcoming aforementioned limitations by proposing a monocular optical-flow-based 3D obstacle avoidance strategy that is real-time implementable by onboard hardware.

Chapter 3

Problem statement

As emerges from conclusions drawn from literature review, most recurrent limitations are the lack of a 3D path planning by just considering motion in a 2D plane parallel to ground and not taking into account frontally approaching obstacles. Furthermore, a priori assumptions on the environment's structure are made and not always a computational cost analysis is carried out.
This thesis aims at solving previously mentioned limitations by developing a real-time 3D local path planning strategy based on the optical flow field generated by an onboard frontal monocular camera, without making a priori assumptions about environment's structure.
In order to develop such an algorithm and test it, simulations must be carried out, hence the kinematic and dynamic equations of motion of the quadrotor must be obtained, as well as quadrotor flight control system, insight about how optical flow works is presented and the most appropriate set of simulation software and metrics for measuring software performance must be identified.

3.1 Quadrotor kinematic and dynamic model

In this section the derivation of a kinematic and dynamic model for the quadcopter is presented. A mathematical model is required for both path planning and simulation purposes.

3.1.1 Reference frames

The quadrotor is considered as a rigid body with 6 DOF (3 DOF for translations and 3 DOF for rotations) and in order to represent all its motions and physical quantities a set of different reference frames (RF) must be defined in a suitable way. As pointed out in [19] , a few reasons for defining different RF are:

- Newton-Euler's equations of motion are defined in the RF attached to the body of the quadrotor;
- Thrust produced by motors and propellers is applied in the body RF;

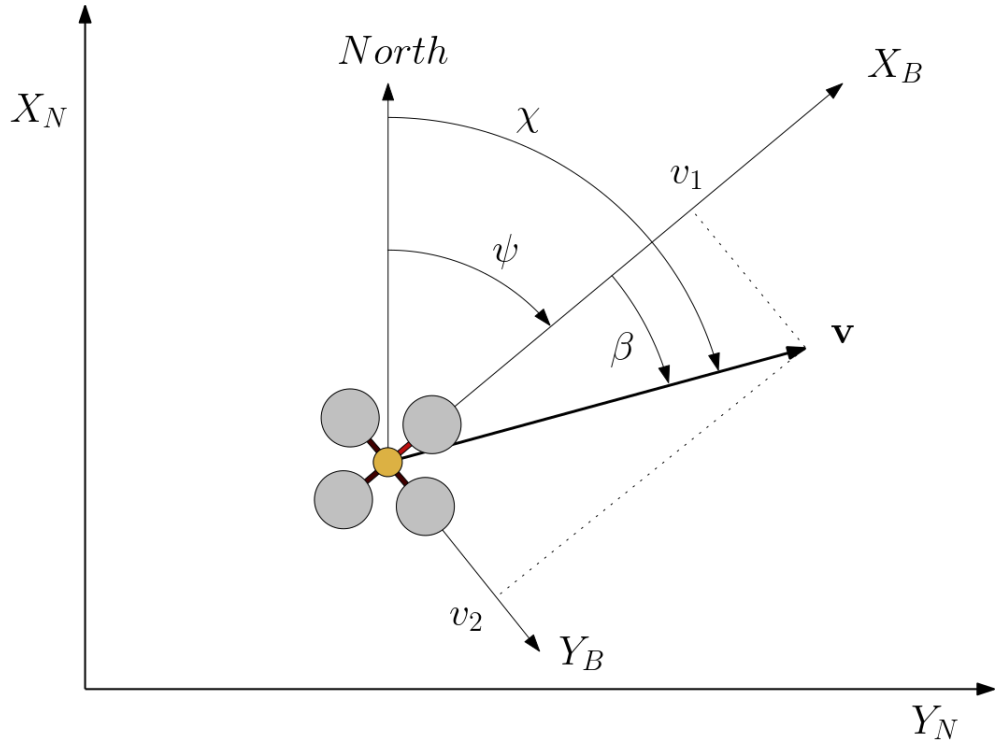


Figure 3.1: In this image different angles are shown. Heading angle ψ is computed as the clockwise deviation from North. Course angle χ refers to actual motion direction (velocity direction $\mathbf{v}$) of the vehicle due to lateral wind perturbation. The angle difference between course and heading is β and is called crab angle or side-slip angle or drift angle.

- Some measurements provided by sensors, such as accelerations by accelerometers and angular rates by gyroscopes, are defined in the body RF. Other quantities such as position (GPS), heading angle (magnetometer), course angle (GPS) and ground speed (GPS) are defined in the inertial RF (Figure 3.1);

- Most flight trajectories in missions are defined in the inertial reference frame, as map information.

In order to provide a consistent representation of all the physical quantities involved in modelling, the following reference frames are defined, trying to be compliant to scientific literature in aerospace field.

Inertial RF $\mathcal{R}_i$ or $\mathcal{R}_{NED}$

The inertial reference frame is defined with the NED (North-East-Down) known in literature, where x-axis is positive pointing North, y-axis is positive pointing East and z-axis is positive pointing toward the center of the Earth. The origin is placed in whatever position has been defined as home position.

The vehicle RF $\mathcal{R}_v$

This frame has axes aligned with the inertial frame but the origin is centered in UAV center of mass (as shown in fig. 3.2).

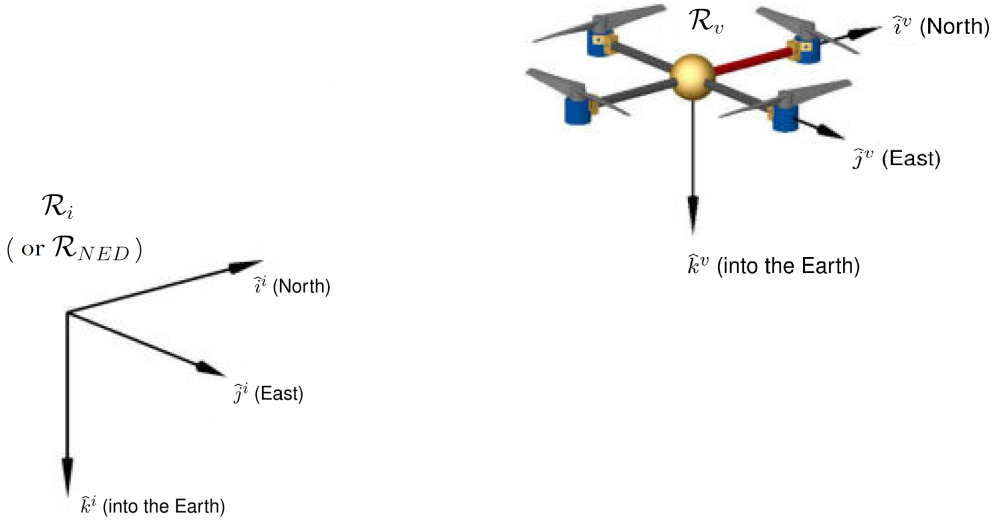


Figure 3.2: The world and the vehicle reference frames.

The body RF $\mathcal{R}_b$

In order to move from the vehicle RF to the body RF representation, attitude of the UAV is obtained by subsequent rotations using RPY angles (and respective rotation matrices concatenation). Intermediate RFs as $\mathcal{R}_{v1}$ and $\mathcal{R}_{v2}$ are defined for defining rotations. $\mathcal{R}_{v1}$ is rotated by yaw angle ψ around $\hat{k}_v$, $\mathcal{R}_{v2}$ is rotated by pitch angle θ around $\hat{j}_{v1}$ and $\mathcal{R}_b$ is rotated by roll angle ϕ around $\hat{i}_{v2}$. Multiplying in the right order obtained rotation matrices the complete RPY rotation matrix is obtained:

$$\begin{aligned} R_v^b(\phi,\ \theta,\ \psi) &= R_{v2}^b(\phi)\, R_{v1}^{v2}(\theta)\, R_v^{v1}(\psi) \\ &= \begin{pmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\theta s_\phi \\ c_\phi s_\theta c_\psi + s_\phi s_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi c_\theta \end{pmatrix} \end{aligned} \tag{3.1}$$

where $c_\theta = \cos\theta$ and $s_\theta = \sin\theta$.
Notice that since from reference frame $\mathcal{R}_i$ to $\mathcal{R}_v$ there is only a translation, following relation holds:

$$R_b^v = R_b^i$$

3.1.2 Quadrotor rotations conventions

The quadrotor is supposed to fly in "+ mode", i.e. with one arm aligned with the motion direction. Numbering convention for rotors and respective speed and moment directions are defined as showed in fig. 3.3 taken from [20]:

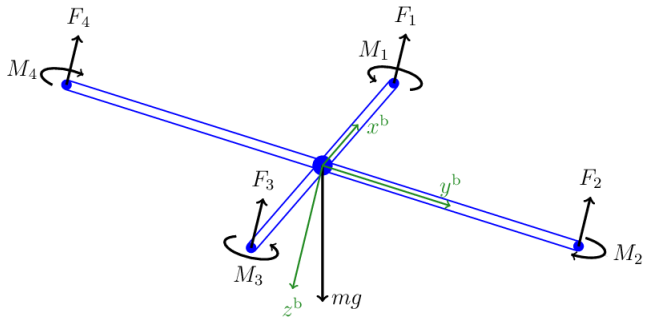


Figure 3.3: Rotations conventions on quadrotor vehicles.

Forces produced by motors thrusts are parallel and opposite to $\hat{k}_b$.
Each motor produces also a moment which direction is opposite to propeller speed's one. Rotors 1 and 3 rotate positive according to $\hat{k}_b$, while rotors 2 and 4 rotate in direction $-\hat{k}_b$. So rotors 1 and 3 will produce a moment along $-\hat{k}_b$ and rotors 2 and 4 will produce a moment along $\hat{k}_b$.

3.1.3 Kinematic and dynamic nonlinear model

State variables of the quadrotor

A quadrotor has 12 states (position, linear velocity, attitude, angular velocity of 3 dimensions each) listed and specified in table 3.1.

Notice that the following relation holds:

$$\mathbf{r_i} = \begin{pmatrix} p_n \\ p_e \\ -h \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

States are represented graphically in figure 3.4.

Kinematic model

For what concerns linear part, position $\mathbf{r} = (p_n, p_e, -h)^T = (x, y, z)^T$ is a state defined in the inertial frame and its variation is linked to state $\mathbf{v} = (u, v, w)^T$ defined in the body frame through the rotation matrix from $\mathcal{R}_b$ to $\mathcal{R}_i$.
As derived in [19]:

Table 3.1: States of the quadrotor

$\mathbf{r_i} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$	inertial (north) position of the quadrotor along $\hat{i}_i$ in $\mathcal{R}_i$, inertial (east) position of the quadrotor along $\hat{j}_i$ in $\mathcal{R}_i$, altitude of the aircraft measured along $-\hat{k}_i$ in $\mathcal{R}_i$,
$\mathbf{v_b} = \begin{pmatrix} u \\ v \\ \omega \end{pmatrix}$	body frame velocity measured along $\hat{i}_b$ in $\mathcal{R}_b$ body frame velocity measured along $\hat{j}_b$ in $\mathcal{R}_b$ body frame velocity measured along $\hat{k}_b$ in $\mathcal{R}_b$
$\alpha = \begin{pmatrix} \phi \\ \theta \\ \psi \end{pmatrix}$	roll angle defined with respect to $\mathcal{R}_{v2}$, pitch angle defined with respect to $\mathcal{R}_{v1}$, yaw angle defined with respect to $\mathcal{R}_v$,
$\omega_\mathbf{b} = \begin{pmatrix} p \\ q \\ r \end{pmatrix}$	angular velocity component along $\hat{i}_b$ in $\mathcal{R}_b$ angular velocity component along $\hat{j}_b$ in $\mathcal{R}_b$ angular velocity component along $\hat{k}_b$ in $\mathcal{R}_b$

$$\begin{aligned} \dot{\mathbf{r}}_\mathbf{i} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} \dot{p_n} \\ \dot{p_e} \\ -\dot{h} \end{pmatrix} = R_b^v \begin{pmatrix} u \\ v \\ w \end{pmatrix} = (R_v^b)^T \begin{pmatrix} u \\ v \\ w \end{pmatrix} \\ = \begin{pmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & c_\theta s_\phi & c_\phi c_\theta \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} \end{aligned} \tag{3.2}$$

For what concerns angular part, the relationship between RPY absolute angles and the angular rates p, q and r is complicated by the fact that these quantities are defined in different RF.

RPY angles are defined in three different frames, so the contribution of the variation rate of each to the angular velocity ω is obtained by considering consequent RF rotations and, after ding computations, the final matrix relationship is:

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \tag{3.3}$$

Around hovering position, given that angles can be considered very small, so $\cos x \approx \sec x \approx 1$ and $\sin x \approx \tan x \approx 0$, the matrix above can be approximated with the identity matrix.

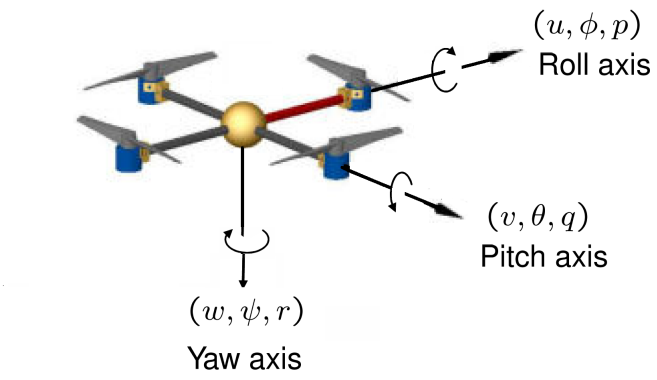


Figure 3.4: States reported on a quadrotor UAV.

Rigid body dynamic model

In order to derive the dynamic equations of the quadrotor as a rigid body, for the linear part the Newton's equation is used, while for the angular part the Euler's equation is used.

Newton's equation

On the quadrotor are acting the gravity force, measured in the inertial frame, and the forces deriving by the four motor thrusts, measured in the body RF.

$$m\ddot{\mathbf{r}} = -\ R_b^i \mathbf{F} + m\mathbf{g}$$

$$mR_b^i \begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{pmatrix} = -R_b^i \begin{pmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{pmatrix} + m \begin{pmatrix} 0 \\ 0 \\ G \end{pmatrix}$$

Defining the forces vector $(F_1,\ F_2, F_3, F_4)^T$ and multiplying both sides by R_i^b and by $\frac{1}{m}$, the dynamic equation can be rewritten as:

$$\begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{pmatrix} = -\frac{1}{m} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} + R_i^b \begin{pmatrix} 0 \\ 0 \\ G \end{pmatrix}$$

Plus, we can define the first input of the system u_1 as:

$$u_1 = \sum_i F_i$$

Euler's equation

The inertia matrix of the quadrotor Γ is assumed as known. Given that the quadrotor is essentially symmetric about all three axes and assuming it is computed along principal axes, the inertia matrix will be assumed diagonal. So it will have this form:

$$\Gamma = \begin{pmatrix} \Gamma_x & 0 & 0 \\ 0 & \Gamma_y & 0 \\ 0 & 0 & \Gamma_z \end{pmatrix}, \quad \Gamma^{-1} = \begin{pmatrix} \frac{1}{\Gamma_x} & 0 & 0 \\ 0 & \frac{1}{\Gamma_y} & 0 \\ 0 & 0 & \frac{1}{\Gamma_z} \end{pmatrix}$$

In addition to forces, each motor produces a moment which direction is opposite to propeller speed's one. Rotors 1 and 3 rotate positive according to $\hat{k}_b$, while rotors 2 and 4 rotate in direction $-\hat{k}_b$. So rotors 1 and 3 will produce a moment along $-\hat{k}_b$ and rotors 2 and 4 will produce a moment along $\hat{k}_b$.

The obtained Euler's equation is:

$$\Gamma \begin{pmatrix} \dot{p} \\ q \\ \dot{r} \end{pmatrix} = \begin{pmatrix} L(F_4 - F_2) \\ L(F_1 - F_3) \\ M_2 + M_4 - M_1 - M_3 \end{pmatrix} - \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \Gamma \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

Where L is the quadrotor arm length. By representing $M_i = \gamma F_i$, where $\gamma = \frac{k_M}{k_F}$, we can write the above equation as a function of forces vector:

$$\Gamma \begin{pmatrix} \dot{p} \\ q \\ \dot{r} \end{pmatrix} = \begin{pmatrix} 0 & -L & 0 & L \\ L & 0 & -L & 0 \\ -\gamma & \gamma & -\gamma & \gamma \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} - \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \Gamma \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

Multiplying by Γ^{-1} both sides:

$$\begin{pmatrix} \dot{p} \\ q \\ \dot{r} \end{pmatrix} = \Gamma^{-1} \left[\begin{pmatrix} 0 & -L & 0 & L \\ L & 0 & -L & 0 \\ -\gamma & \gamma & -\gamma & \gamma \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} - \begin{pmatrix} p \\ q \\ r \end{pmatrix} \times \Gamma \begin{pmatrix} p \\ q \\ r \end{pmatrix} \right]$$

Where:

$$\begin{pmatrix} u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 0 & -L & 0 & L \\ L & 0 & -L & 0 \\ -\gamma & \gamma & -\gamma & \gamma \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix}$$

So the complete input vector of the system is the vector $(u_1, u_2, u_3, u_4)^T$, where u_1 is the vertical thrust force and u_2, u_3, u_4 are the torques generated around $\hat{i}_b$, $\hat{j}_b$, $\hat{k}_b$ axes, respectively.

Complete nonlinear quadrotor model

All previously derived equations are here grouped in table 3.2.

Table 3.2: Complete nonlinear kinematic and dynamic model of quadrotor.

Linear Kinematic equations
$\dot{x} = (c_\theta c_\psi)u + (s_\phi s_\theta c_\psi - c_\phi s_\psi)v + (c_\phi s_\theta c_\psi + s_\phi s_\psi)w$
$\dot{y} = (c_\theta s_\psi)u + (s_\phi s_\theta s_\psi + c_\phi c_\psi)v + (c_\phi s_\theta s_\psi - s_\phi c_\psi)w$
$\dot{z} = -s_\theta u + (c_\theta s_\phi)v + (c_\phi c_\theta)w$
Angular Kinematic equations
$\dot{\phi} = p + (\sin\theta \tan\theta)q + (\cos\phi \tan\theta)r$
$\dot{\theta} = \cos\phi q - \sin\phi q$
$\dot{\psi} = (\sin\phi \sec\theta)q + (\cos\phi \sec\theta)r$
Linear Dynamic equations
$\dot{u} = -\sin\theta G$
$\dot{v} = -\cos\theta \sin\phi G$
$\dot{w} = -\frac{1}{m}(F_1 + F_2 + F_3 + F_4) + \cos\phi \cos\theta G$
Angular Dynamic equations
$\dot{p} = \frac{L}{\Gamma_x}(F_4 - F_2) + \frac{\Gamma_y - \Gamma_z}{\Gamma_x}qr$
$\dot{q} = \frac{L}{\Gamma_y}(F_1 - F_3) + \frac{\Gamma_z - \Gamma_x}{\Gamma_y}pr$
$\dot{r} = \frac{\gamma}{\Gamma_z}(F_2 + F_4 - F_1 - F_3) + \frac{\Gamma_x - \Gamma_y}{\Gamma_z}pq$

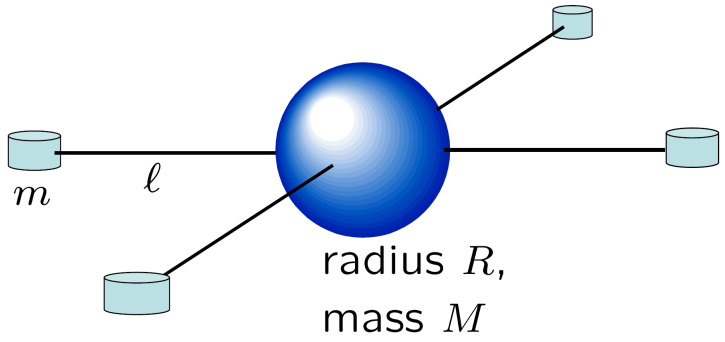


Figure 3.5: Simplified quadrotor model for inertia tensor values computation.

Inertia matrix computation

In Euler's equations rotational inertia is taken into account in inertia tensor $\mathbf{\Gamma}$, which general form is:

$$\mathbf{\Gamma} = \begin{pmatrix} \int(y^2+z^2)dm & -\int(xy)dm & -\int(xz)dm \\ -\int(xy)dm & \int(x^2+z^2)dm & -\int(yz)dm \\ -\int(xz)dm & -\int(yz)dm & \int(x^2+y^2)dm \end{pmatrix} \triangleq \begin{pmatrix} \Gamma_x & -\Gamma_{xy} & -\Gamma_{xz} \\ -\Gamma_{xy} & \Gamma_y & -\Gamma_{yz} \\ -\Gamma_{xz} & -\Gamma_{yz} & \Gamma_z \end{pmatrix}$$

where x, y, and z are coordinates of points on the object relative to the center of mass. In order to simplify computation, the quadrotor is modeled as a spherical dense center with mass M and radius R, and point masses of mass m located at a distance of l from the center, as shown in Figure 3.5.

As can be stated, quadrotor is essentially symmetric about all thee axes, hence $\Gamma_{xy} = \Gamma_{xz} = \Gamma_{yz} = 0$, which leads to a diagonal inertia matrix in the form:

$$\mathbf{\Gamma} = \begin{pmatrix} \Gamma_x & 0 & 0 \\ 0 & \Gamma_y & 0 \\ 0 & 0 & \Gamma_z \end{pmatrix}$$

Therefore its inverse is easily obtained as:

$$\mathbf{\Gamma^{-1}} = \begin{pmatrix} \frac{1}{\Gamma_x} & 0 & 0 \\ 0 & \frac{1}{\Gamma_y} & 0 \\ 0 & 0 & \frac{1}{\Gamma_z} \end{pmatrix}$$

A sphere of mass M and radius R has an inertia of $\frac{2MR^2}{5}$ along all axes, while a point mass m at a distance l from the axis of rotation has a rotational inertia of l^2m. Given the linearity property of integral operator, the values in the inertia tensor are given by the following:

Figure 3.6: Gazebo model of quadrotor UAV implemented in *hector_quadrotor* package. A Hokuyo UTM-30LX laser scanner is mounted below the main body in this image, but this sensor is not used in this thesis, hence a modified version without it is used.

$$\Gamma_x = \frac{2MR^2}{5} + 2l^2m$$
$$\Gamma_y = \frac{2MR^2}{5} + 2l^2m$$
$$\Gamma_z = \frac{2MR^2}{5} + 4l^2m$$

hector_quadrotor package UAV parameters

The quadrotor used in this thesis comes from a ROS package called *hector_quadrotor*, which simulates the physics and control of a quadrotor UAV. The UAV can be seen in Figure 3.6.
In Table 3.3 main quantities defining *hector_quadrotor* model are shown.
Inertia matrix of *hector_quadrotor* is:

$$\mathbf{\Gamma} = \begin{pmatrix} 0.01152 & 0 & 0 \\ 0 & 0.01152 & 0 \\ 0 & 0 & 0.0218 \end{pmatrix}$$

3.2 Quadrotor state estimation and control

For the aim of this thesis, a required assumption is that quadrotors are controlled in position, given in the inertial reference frame. Therefore, suitable commands must

Table 3.3: *hector_quadrotor* parameters.

Name	Value
Width	0.79 m
Length	0.79 m
Height	0.22 m
Mass	1.477 kg
Γ_x	0.01152 $kg \cdot m^2$
Γ_y	0.01152 $kg \cdot m^2$
Γ_z	0.0218 $kg \cdot m^2$

be produced to track reference positions.
From a control perspective, given the defined system inputs and by spotting interdependencies of states in the equations, the dynamical model of the quadrotor can be decomposed into two subsystems: a translation one and an angle one, shown in figure 3.7.

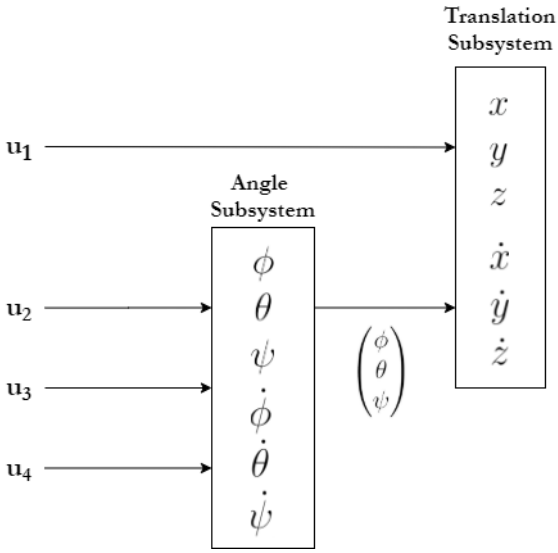


Figure 3.7: Quadrtor's dynamic model decomposition in two subsystems.

In order to control quadrotor flight, these system inputs are generated by a cascaded PID controller. The ROS package used for simulation, *hector_quadrotor* [21], can implement controllers in velocity and in position by using a set of packages called *ros_control* [22], which includes controller interfaces, controller managers, transmissions and hardware interfaces, letting ROS developers setup real time controllers for controlling real hardware by means of an asynchronous architecture like ROS.

The cascaded PID control of Hector Quadrotor is organized as following. An inner loop controls pitch θ, roll ϕ, yaw rate $\dot{\psi}$ and vertical velocity $\dot{z}$. An outer loop controls the horizontal velocity $(\dot{x}, \dot{y})$, the heading ψ and the altitude z. A further

PID control is implemented in order to control the horizontal position (x, y) in the inertial reference frame. A block diagram scheme of the quadrotor PID position control is shown in figure 3.8.

The cascaded PID controllers produce as output commands the input forces of the system $(U_1, U_2, U_3, U_4)^T$ (vertical thrust and torques) that are then translated in terms of electric motors voltages $(V_1, V_2, V_3, V_4)^T$ either by using a static mixture matrix (which in the quadrotor case depends on the so called + or X flight configurations) or by feeding them into an inverted model of the propulsion system [21].
In order to correctly track reference signals, quadrotor states must be known, hence they are estimated bu means of an Extended Kalman Filter (EKF) to fuse all available measurements to a single navigation solution containing the orientation, position and velocity of the vehicle as well as observable error states like the IMU bias errors. This approach is usually referred to as integrated navigation. The EKF is implemented in ROS package *hector_pose_estimation_core*.
PID controllers are therefore tuned such that quadrotor's flight is smooth, with limited overshoot and with non-aggressive manouvers. This is carried out by a trial and error procedure and final parameters for ROS implementation of controllers are reported in table 3.4. Here controllers are presented as their software implementation. Parameters are set by modifying file *params/controller.yaml* inside package *hector_quadrotor_controllers*.

In ROS, pose commands are sent as messages on topic *command/pose* of message type *geometry_msgs/PoseStamped*.

Table 3.4: PID controllers parameters.

Position controller			
	x, y	z	ψ
P	0.8	0.8	5.0
I	0.0	0.0	0.0
D	0.0	0.0	3.0
Velocity controller			
	u, v	w	
P	5.0	5.0	
I	1.0	1.0	
D	0.0	0.0	
Input limit	0.5 m/s	0.5 m/s	
Attitude controller			
	ϕ, θ	$\dot{\psi}$	
P	5.0	5.0	
I	1.0	1.0	
D	0.0	0.0	
Input limit	$\pi/4$ rad	π rad/s	

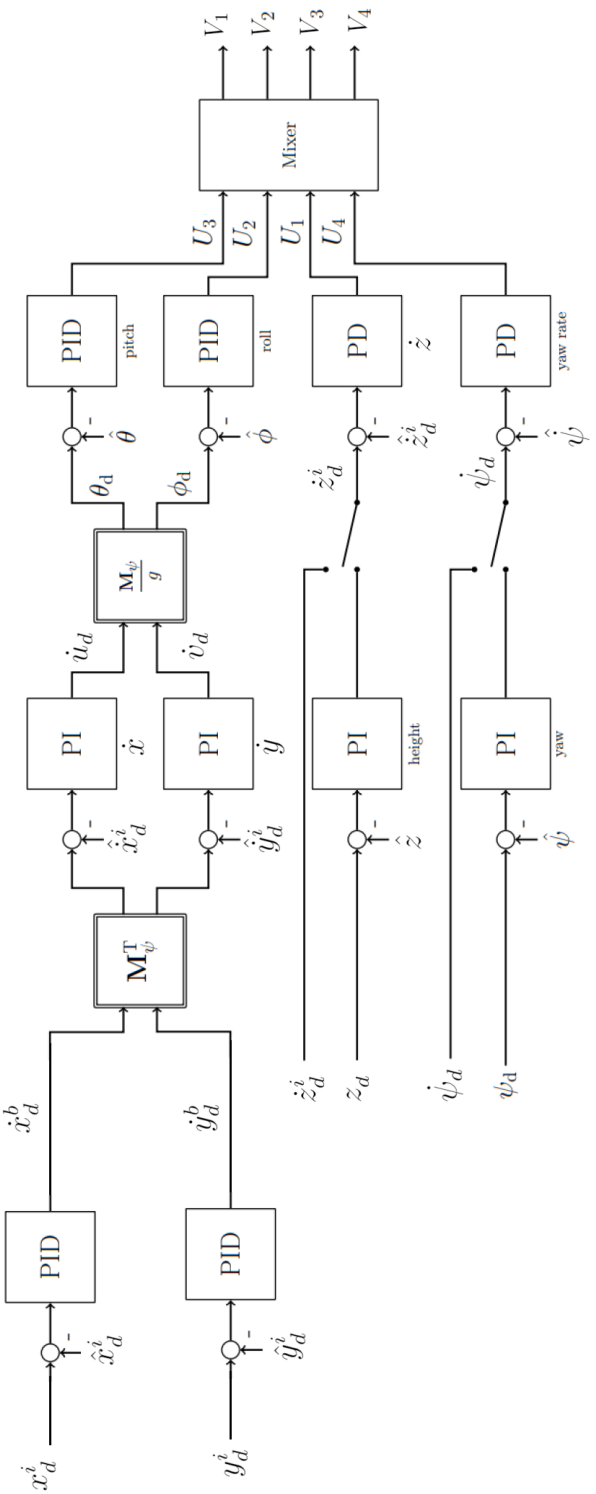


Figure 3.8: Cascaded PID controllers for position control of the quadrotor in the inertial coordinate frame.

3.3 Optical Flow theory

Nature has always been a fascinating source of ideas for finding solutions to practical problems, nonetheless engineering problems. After all, millions of years of evolution have refined biological systems to survive and adapt, finding the most curious ways to overcome challenging environments or reduced capabilities. Bio-inspired algorithms in robotics constitute a huge branch of research that provides several solutions to a wide variety of problems, such as path-planning, image analysis and so on.
Optical flow is one of those bio-inspired algorithms and is inspired by how birds and insects use motion estimation in their field of view to navigate the environment. The concept was first introduced by American psychologist James Jerome Gibson in the 1940's when describing how animals perceive their environments to be able to move around freely without collisions [5]. Humans too constantly make use of optical flow to help complete tasks, such as estimation of self-motion, distinguishing moving and static object in the environment, determining the depth of different objects, estimating for time-to-contact.
This method has been a great inspiration for a lot of applications, both in robotics and in other fields of technology: motion and structure-from-motion estimation, video compression codecs to interpolate between key frames (e.g. MPEG), optical flow sensors (e.g. the one used in optical mice), visual odometry in robot navigation, obstacle avoidance, car driving assistance to detect other cars or pedestrians, creation of additional frames in fast high resolution displays are just some examples of use cases of this technique.

Optical flow in computer vision can be defined in an high-level manner as the motion of objects between consecutive frames of sequence, caused by the relative movement between the object and camera. To use a more precise definition given by Horn and Schunck [6]: "The optical flow is a velocity field in the image that transforms one image into the next image in a sequence. As such it is not uniquely determined; the motion field, on the other hand, is a purely geometric concept, without any ambiguity — it is the projection into the image of three-dimensional (3-D) motion vectors." So what optical flow algorithm produce as an output given two subsequent frames as input is a velocity field. In this thesis the abbreviation OF for optical flow will refer to the 2D vector field generated by one of these methods. A visual example of a produced OF vector field from subsequent frames is shown in Figure 3.9.

3.3.1 Classical mathematical formulation of Optical Flow

This 2D vector field representing optical motion is computed in general by solving an optimization problem that relies on the following three key assumptions about the sampled images:

1. *Object brightness invariance*: local changes in brightness are caused only by

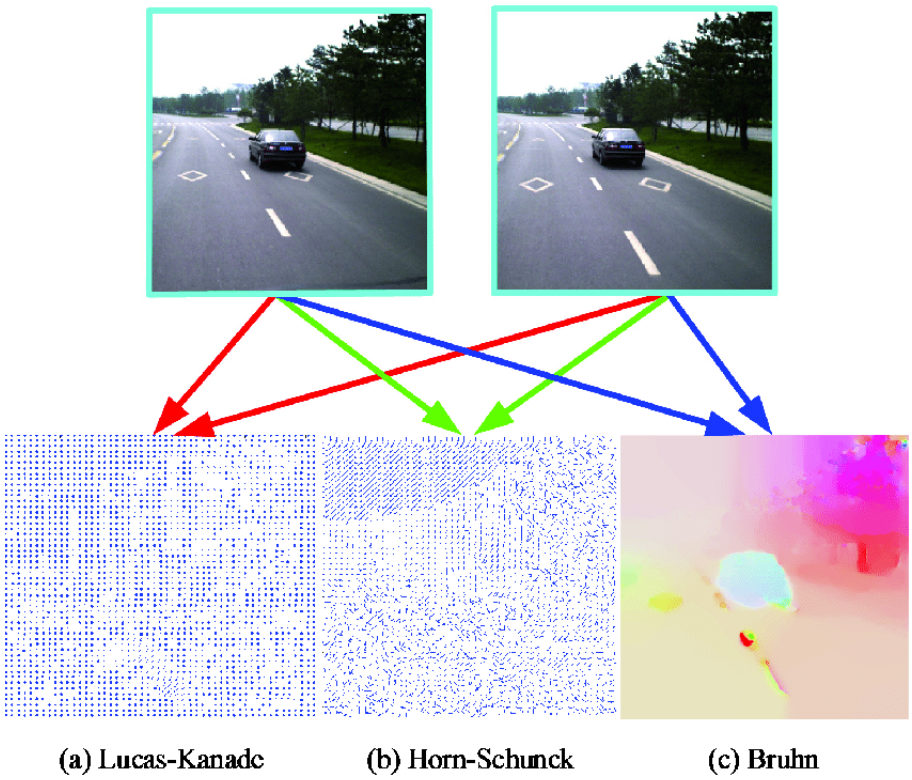


Figure 3.9: The produced optical flow 2D vector field describes the motion of each pixel among subsequent frames. In the picture, taken from [23], results produced from three different algorithms are shown. First two are sparse OF fields while the last one is a dense OF field.

the motion of a certain object with respect to the camera;

2. *Spatial coherence*: the motion is uniform in a small area around a pixel;

3. *Temporal persistence*: In a patch image motion does not change abruptly but gradually over time.

Assuming $I(x, y, t)$ is the gray scale image, containing the brightness of pixel (x, y) at time t (see Figure 3.10), from constant brightness between consecutive frames the following equation can be written:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$$

Then the first order Taylor series approximation of the right-hand side of the equation can be computed, obtaining:

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\delta I}{\delta x}\delta x + \frac{\delta I}{\delta y}\delta y + \frac{\delta I}{\delta t}\delta t + \kappa$$

where κ are higher order terms of the Taylor approximation. Terms $\delta x, \delta y$ and δt are small variations in x, y and t. Higher order terms κ can be neglected because

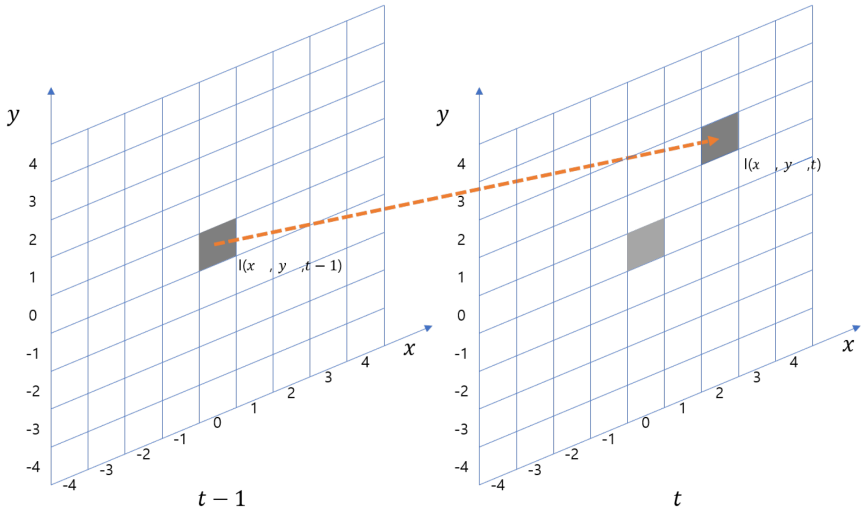


Figure 3.10: The displacement of a pixel obtained among subsequent frame defines the optical flow vector for that specific pixel.

they are very small in magnitude. Furthermore, from previous identity based on brightness constancy assumption, common terms can be deleted, leading to the equation:

$$\frac{\delta I}{\delta x}\delta x + \frac{\delta I}{\delta y}\delta y + \frac{\delta I}{\delta t}\delta t = 0$$

Dividing both sides for dt:

$$\frac{\delta I}{\delta x}u + \frac{\delta I}{\delta y}v + \frac{\delta I}{\delta t} = 0 \tag{3.4}$$

where $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$. Another popular form of the optical flow equation is:

$$I_t = -\nabla I \cdot V, \quad \nabla I = \begin{bmatrix} I_x \\ I_y \end{bmatrix}, \quad V = \begin{bmatrix} u \\ v \end{bmatrix}$$

Terms $I_x = \frac{\delta I}{\delta x}, I_y = \frac{\delta I}{\delta y}$ and $I_t = \frac{\delta I}{\delta t}$ are respectively the horizontal gradient, the vertical gradient and the variation of brightness over time.
The obtained equation (3.4) is called optical flow equation. This equation has two variables (u and v), so it's under-constrained and needs more constraints in order to find feasible solutions to the problem. This situation of analytically undetermined algebraic system is also known as the *aperture problem*, shown in a most intuitive way by the "barber's pole" example in Figure 3.11, where the real motion of the cylinder is circular and the perceived optical flow is a vertical motion field when the real one is horizontal. Another example of this phenomena coming from daily

life is when we are standing in a stationary train observing another adjacent train and we have the feeling that we are moving when instead it is the other that is moving. This happens because we have a limited opening from the window and we don't have precise references to decide which of the two trains is really in motion. [24]

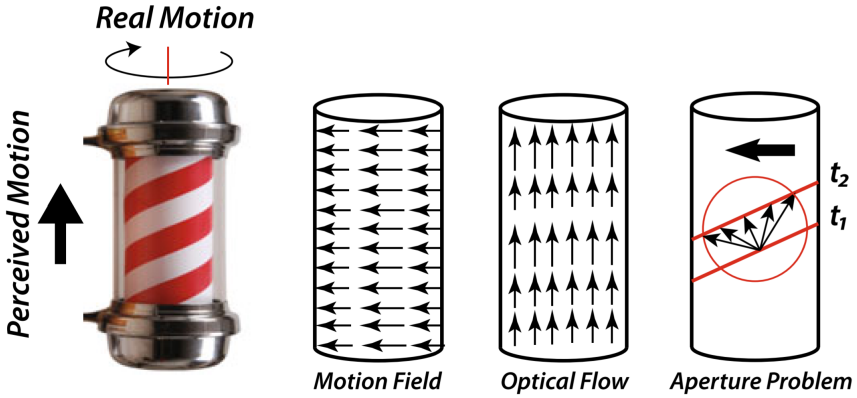


Figure 3.11: Barber's pole illusion. Barber's pole is a panel used in the MiddleAges by barbers. The white cylinder, with the red ribbon wrapped helically, rotates clockwise but the stripes are perceived to move vertically upwards. The perceived optical flow does not correspond to the real motion field, which is horizontal from right to left. This illusion is caused by the aperture problem or by the ambiguity of finding the correct correspondence of points on the edge of the tape (in the central area when observed at different times) since the direction of motion of these points is not determined uniquely by the brain. (Hans Wallach in 1935, a psychologist, discovered that the illusion is less if the cylinder is shorter and wider, and the perceived motion is correctly lateral. The illusion is also solved if the texture is present on the tape.) [24]

Depending on which assumptions and hence constraints are added, different OF techniques are proposed.

3.3.2 OF algorithms classifications

Different classifications are possible for OF algorithms; for instance, according to [25], the following four broad classes are spotted:

1. *Gradient-based*: These methods try to solve the OF equation (3.4) typically tackle this problem by including some constraints—usually based on some form of spatial or temporal coherence—in the algebraic system of equations to be solved. Four main algorithm of this class are *gradient algorithm* [26], *Lukas-Kanade* [27], *Horn and Schunck* [6], *Proesmans* [28].
   The *Gradient algorithm* assumes that OF is constant within a certain spatial and temporal neighborhood of a pixel and assembles an overdetermined system of equations and solves it using Least Square method, by discarding computations over areas in which derivatives are too small or too similar (e.g. due to lack of motion or because there are not distinguishable features) hence

improving computation. *Lukas-Kanade* method is similar with the difference that uses weights to the neighborhoods as a function of their distances from the considered point. *Horn and Schunck* combines equation (3.4) with a global smoothness term λ with the goal to constrain the estimated velocity. *Proesmans* method uses the minimization of an energy functional, similarly to *Horn and Schunck*, but also takes into account the bias in the direction of motion due to correlations in the finite difference approximation.
Algorithms of this class have the advantage of being computationally efficient, but they suffer of the aperture problem and calculation of spatial and temporal derivatives is prone to errors.

2. *Phase-based*: First proposal of this method was made by *Fleet and Jepson* [29], which main idea is that 2D image velocity can be modeled as the phase behavior of a band-pass filter output.
These methods are known to work well for relatively slow motion, however, they are not reliable when trying to estimate fast motion.

3. *Region-Matching-based*: This class of algorithms computes the OF vector for a given pixel by finding the displacement of the region around that pixel between two consecutive frames. This is accomplished by means of a minimization of a predefined function of the differences between two templates.
The advantage of these algorithms is that they behave better than gradient-based ones in situations with fast motion, but they have the drawback of having a high computational burden (in particular, computation time increases quadratically with the maximum allowed object displacement).

4. *Feature-Matching-based*: These algorithms compute OF component by considering the displacement of certain image features between two consecutive frames, detected by a feature-detection algorithm and later associated by a feature-matching algorithm. Examples of feature-detection methods are Harris corner detector [30], scale-invariant feature transform (SIFT) [31] and Shi-Tomasi corner detector [32]. The methods of this class rely on the assumption that the same detected feature over an image are consistently detectable and trackable over subsequent image frames. The advantage is that this doesn't imply a maximum displacement limits between same feature among different frames. Drawbacks are that these features belong to an evenly spaced grid, precision is strongly dependent on performances of associated matching algorithm and they are computationally demanding (but parallelizable).

Another classification of OF techniques divides them in two categories:

1. *Sparse Optical Flow*: Sparse OF gives the flow vectors of just a set of selected pixel, for instance those identified as corresponding to feature detected by previously mentioned corner detecting methods. In this way, computation is carried out for a reduced set of pixel, so it is computationally lighter, but can lack in accuracy and is not suitable for all situations.

2. *Dense Optical Flow*: Dense OF computes flow vectors for all the pixels composing the frame. It has higher accuracy at the cost of being slow/computationally expensive.

An example of the difference between the two categories can be seen in Figure 3.12. The OF algorithm applied to the considered pixels (a subset or the whole frame), can then be any of the previously mentioned algorithms.

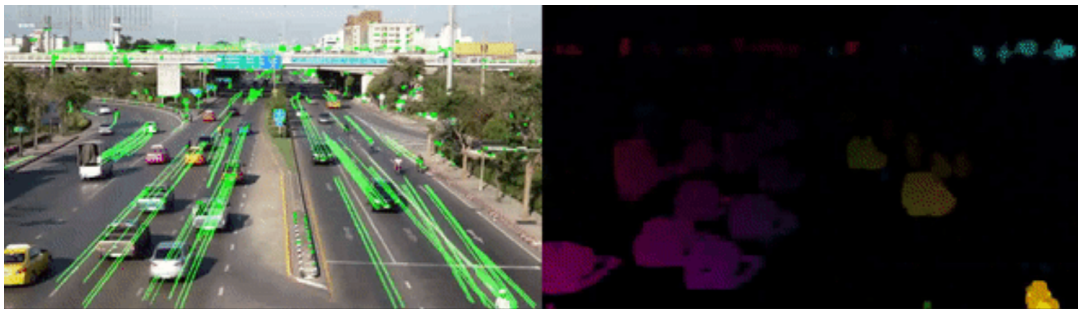


Figure 3.12: Comparison between sparse and dens OF. On the left, OF is computed only for specific featured detected by feature-detection algorithm and tracked in following frames by means of OF algorithm. On the right, a dense OF field is shown, where OF is computed for each pixel of the frame and vector information is color-coded: for each pixel, brightness (value in HSV) is the magnitude of the associated motion vector while color (hue in HSV) is vector angle.

Gunnar Farnebäck Dense OF method

In order to implement obstacle avoidance, an analysis of the whole frame's OF is required, so a dense OF method is used. In particular, Farnebäck's method is chosen [33]. Farnebäck's method follows a different reasoning from previously seen techniques. Its first step is to approximate each pixel's neighborhood in both frames by quadratic polynomials, which can be done efficiently using the polynomial expansion transform, in the local reference frame given by the form:

$$f(\mathbf{x}) \approx \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$$

where $\mathbf{A}$ is a symmetric matrix, $\mathbf{b}$ is a vector and c is a scalar.

Afterwards, considering how an exact polynomial transforms under translation, a method to estimate displacement fields from the polynomial expansion coefficients is derived and after a series of refinements leads to a robust algorithm.

It is known that OF algorithms can misbehave when large movement are present in the frame, hence sometimes those movements are not detected. A solution for this problem is the use of pyramids (see Figure 3.13); both Lukas-Kanade and Farnebäck's methods use this solution to add robustness.

The idea is that an image pyramid is generated, where each level has a lower resolution compared to the previous level. When a pyramid level greater than 1 is

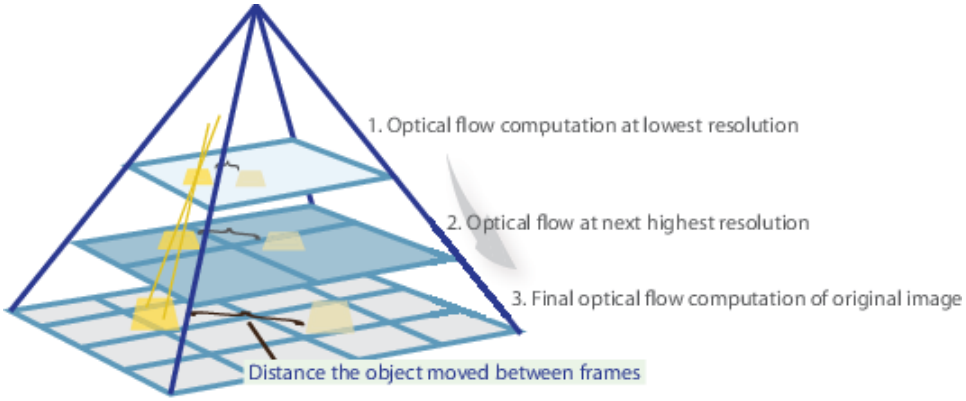


Figure 3.13: Visual representation of pyramid method used to detect larger movement and improve robustness. [34]

selected, the algorithm can track the points at multiple levels of resolution, starting at the lowest level. For each level L the image is shrunk L times resulting in a 2L times smaller image. This shrinking process is called binning. By binning an image an area of pixels in the original image is combined into a single pixel for the output image. A 2x2 binning on an image combines 4 pixels into one pixel and thereby reduces the total number of pixels by a factor 4. The flow speed is then decreased with a factor of 2. Increasing the number of pyramid levels enables the algorithm to handle larger displacements of points between frames. However, the number of computations also increases. The tracking begins in the lowest resolution level, and continues until convergence. The point locations detected at a level are propagated as keypoints for the succeeding level. In this way, the algorithm refines the tracking with each level. The pyramid decomposition enables the algorithm to handle large pixel motions, which can be distances greater than the neighborhood size. [34].

Farnebäck's method is not only more accurate, as proved by results in [33] from a test on Yosemite sequence, but it gets faster on smaller images as tested in [35], where with an increasing binning factor a decreasing execution time with respect to classical Lukas-Kanade method is proved. This becomes important when pyramids method is applied.

3.3.3 Focus of Expansion (FOE), Time-To-Contact (TTC) and Expansion of Optical Flow (EOF)

From the 2D vector field obtained by the algorithm, further information can be retrieved by analyzing it. Even if OF has intrinsically limited information, given that a 3D motion is casted into a 2D plane,

Focus of Expansion

The Focus of Expansion (FOE) depicts the projection center of the environment. Is the point in the image through which, at least theoretically, all the OF vectors pass, so ideally can be found by the intersection of any two vectors in the field. Usually, in ideal situations, it indicates where the drone is pointing to.
In order to compute its coordinates for a given 2D vector field, a possible mathematical reasoning is that FOE is the point in the image that has the smallest cross product with all the vectors of the field.

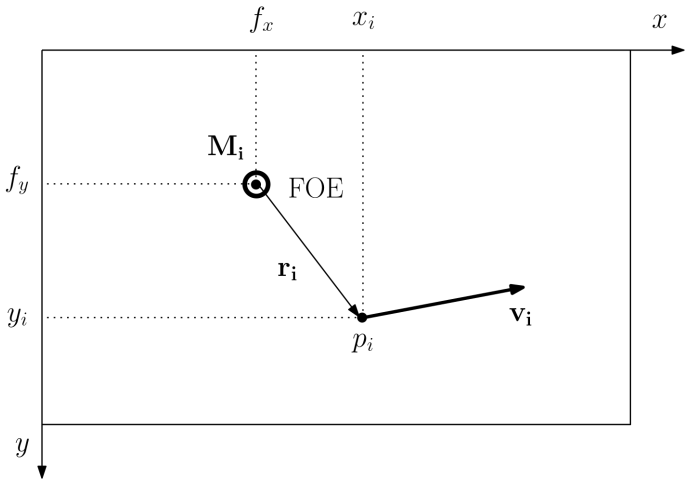


Figure 3.14: Computation of FOE coordinates by minimization of moment $\mathbf{M_i}$ associated to i^{th} pixel.

We know that the OF vector $\mathbf{v_i} = (u_i, v_i)$ represents the displacement of an i^{th} pixel at previous position $p_i = (x_i, y_i)$ to a new location in the next frame. Let's first consider the moment $\mathbf{M_i}$ of the vector $\mathbf{v_i}$ with arm $\mathbf{r_i} = p_i - FOE$ around FOE location $FOE = (f_x, f_y)$, with Figure 3.14 as a reference :

$$\mathbf{M_i} = \mathbf{r_i} \times \mathbf{v_i} = det \begin{bmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ x_i - f_x & y_i - f_y & 0 \\ u & v & 0 \end{bmatrix}$$

Leading to:

$$\mathbf{M_i} = \hat{\mathbf{k}}\,((x_i - f_x)v - (y_i - f_y)u)$$

In order to find FOE coordinates, this moment has to be minimized, so imposing the equation $\mathbf{M_i} = 0$ the following is obtained:

$$\hat{\mathbf{k}}\,((x_i - f_x)v - (y_i - f_y)u) = 0$$

In order to solve the problem for each one of the N pixels in the image, the previous equation is written in matrix form:

$$\begin{bmatrix} -v_1 & u_1 \\ \vdots & \vdots \\ -v_N & u_N \end{bmatrix} \begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} y_1u_1 - x_1v_1 \\ \vdots \\ y_Nu_N - x_Nv_N \end{bmatrix}$$

In a more compact form:

$$\mathbf{A} \cdot FOE = \mathbf{b}$$

That naturally leads to the Least Square problem solution:

$$FOE = \begin{bmatrix} f_x \\ f_y \end{bmatrix} = (\mathbf{A^T A})^{-1}\mathbf{A^T} \cdot \mathbf{b}$$

Time-To-Contact

Knowing the point from which the OF expands, it is possible to extract other useful information to understand how the environment is structured and which obstacles are closer to us. One of those information is Time-to-Contact (TTC) or relative depth. TTC is an estimation of the relative depth as the time that is going to take to collide with that obstacle. It is possible to compute TTC for each pixel in the OF field once FOE has been computed. A visual representation of TTC can be seen in Figure 3.15: pixel brightness is inversely proportional to TTC for that pixel, so the lower the TTC, the higher the brightness, the closer the obstacle.

TTC for the i^{th} pixel is computed with:

$$TTC_i = \frac{\|\mathbf{r_i}\|}{\|\mathbf{OF_i}\|}$$

where $\mathbf{r_i}$ is the distance of i^{th} pixel from FOE and $\mathbf{OF_i}$ is the OF vector in that i^{th} pixel. As can be seen, TTC decreases (i.e. higher risk of collision with obstacles) for high values of $\|\mathbf{OF}\|$ or small values of $\|\mathbf{r_i}\|$.

Expansion of Optical Flow

Another quantity that derives from TTC and is extremely useful for dealing with frontally approaching obstacles is the Expansion of Optical Flow (EOF). When an object is approaching us frontally, it seems for perspective reasons that it is expanding, as shown in Figure 3.16.

Figure 3.15: On the right, video frame with camera moving forward. On the left, respective elaborated image from TTC computation of the recorded scene. The brighter the pixel, the closer the obstacle. Image from [18].

This expansion can be mathematically quantified with the divergence of OF from FOE. The EOF is the sum of all OF divergence components in a considered area of the image (or the whole image if the computation is not restricted to templates), being hence a scalar signal. High values of EOF mean high risk of frontal collision with obstacles.
For the i^{th} pixel, the OF divergence can be computed as the component of $\mathbf{OF_i}$ along the direction that points away from FOE (i.e. along $\mathbf{r_i}$), identified by unit vector $\hat{\mathbf{u}}_{\mathbf{r,i}}$ (see Figure 3.17). This quantity is then divided by the distance from FOE $\|\mathbf{r_i}\|$, so that the farther the considered point (hence the respective OF divergence component) from FOE, the smaller its contribution to frontal collision risk evaluation.

$$OF_{DIV,i} = \mathbf{OF_i} \cdot \hat{\mathbf{u}}_{\mathbf{r,i}}, \quad \hat{\mathbf{u}}_{\mathbf{r,i}} = \frac{\mathbf{r_i}}{\|\mathbf{r_i}\|}$$

$$EOF_i = \frac{OF_{DIV,i}}{\|\mathbf{r_i}\|} = \frac{\mathbf{OF_i} \cdot \mathbf{r_i}}{\|\mathbf{r_i}\|^2}$$

$$EOF = \sum_i^N EOF_i$$

Notice that $OF_{DIV,i} \equiv \frac{1}{TTC_i}$ only when $\mathbf{OF_i}$ lies along $\hat{\mathbf{u}}_{\mathbf{r,i}}$ direction, hence we have a purely expanding OF.

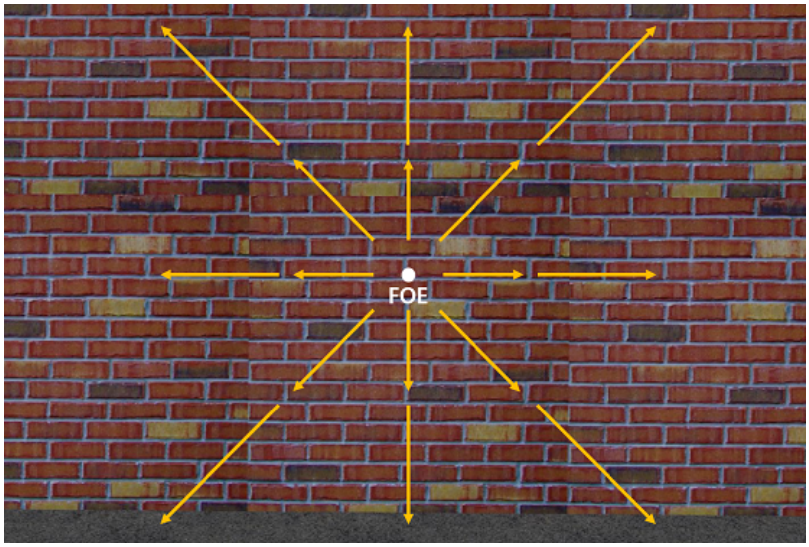


Figure 3.16: An approaching obstacle appears to be expanding in the field of view, hence the optical flow produced by it will ideally have vectors pointing away from the FOE. Image from [15].

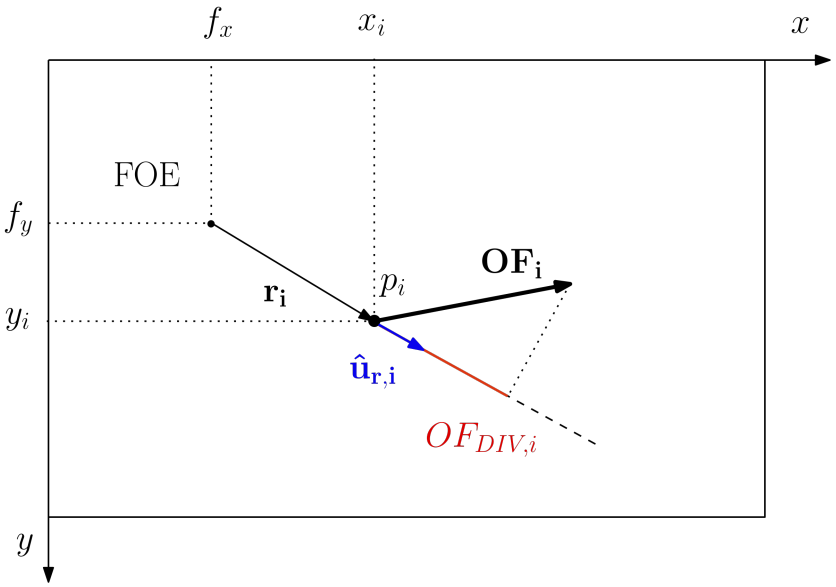


Figure 3.17: Computation of OF divergence component $OF_{DIV,i}$ for the i^{th} pixel.

3.4 Used software

To implement the strategy, apply computer vision algorithms and test the system in a simulated environment, a set of software tools is identified. In the following, software used are introduced and their choice is motivated.

3.4.1 Robotic Operating System (ROS)

In the last years, in robotics research field and lately also in industrial applications [36], ROS [37], which logo can be seen in Figure 3.18a, has become the de facto standard framework for the development of robotics software and plays a key role in the rapid evolution of robotic applications.

(a) Robotic Operating System (ROS) logo.

(b) Gazebo simulator tool logo.

Figure 3.18: ROS and Gazebo logos.

ROS is a collection of software frameworks for robotics software development, a robotics middleware in practice, acting in a distributed fashion and offering advantages 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 [38]. These nodes can exchange data between each other by means of two types of communication paradigm: publish/subscribe, by streaming messages over topics or subscribing to topics, and request/response, by service calls. All the nodes ecosystem is managed by ROS master, the first node that is run and to which nodes register. ROS master that sets up the peer-to-peer communication between nodes and controls updates of a shared database among nodes called parameter server. An example of graph of running nodes obtained by ROS tool *rqt_graph* of ROS demo package *turtlesim* running is shown in Figure 3.19.

Main languages supported for writing ROS nodes are C++, Python and Lisp and client libraries are provided and released under BSD license, and as such are open source software and free for both commercial and research use. The majority of other packages are licensed under a variety of open source licenses. These other packages implement commonly used functionality and applications such as

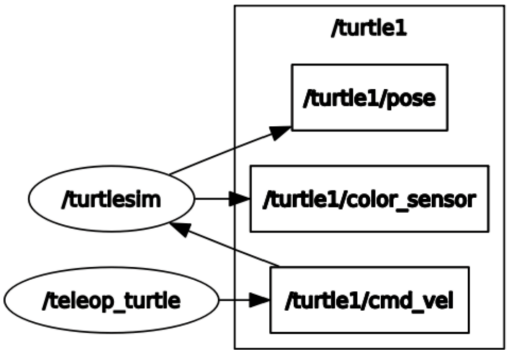


Figure 3.19: Example of graph representing ROS nodes running in a demo application

hardware drivers, robot models, datatypes, planning, perception, simultaneous localization and mapping, simulation tools, and other algorithms, hence promoting software reusability.
An advantage of ROS is that it is conceived as a distributed system, hence nodes that implement computationally heavy activities can be run on a more powerful machine different from the robot itself without the system even perceiving it. Another advantage is the native integration with simulation tools such as Gazebo.
The main ROS client libraries are geared toward a Unix-like system and ROS is provided in distributions following notation-wise the same of Ubuntu distros. For this thesis, distro ROS Melodic Morenia, compatible with Ubuntu 18.04 LTS, is used and Ubuntu distro is run on a virtual machine powered by Oracle VM VirtualBox.

3.4.2 Gazebo

Testing robots algorithms on real robots involves risks and costs: in early releases, bugs can occur and lead to failure, hence damaging the real robot; in case of aerial robots, the consequences of unsuccessful tests become pretty obvious. This is why simulation plays a major role in robotics and in algorithms testing. An advantage of ROS is its native compatibility and integration with the open source simulator tool Gazebo (which logo can be seen in Figure 3.18b), an independent project supported by the same developers of ROS [39].

Gazebo, is a 3D robotics simulator capable of simulating robotics arms, wheeled robots, aerial robots and whatever combination of different types of joints can be described, such as the one shown in Figure 3.20. It integrates high performance physics engine such as ODE, OpenGL rendering and it's also able to support code for sensors simulation and actuators control.

In this thesis, Gazebo 9.0 is used, due to compatibility with ROS Melodic Morenia.

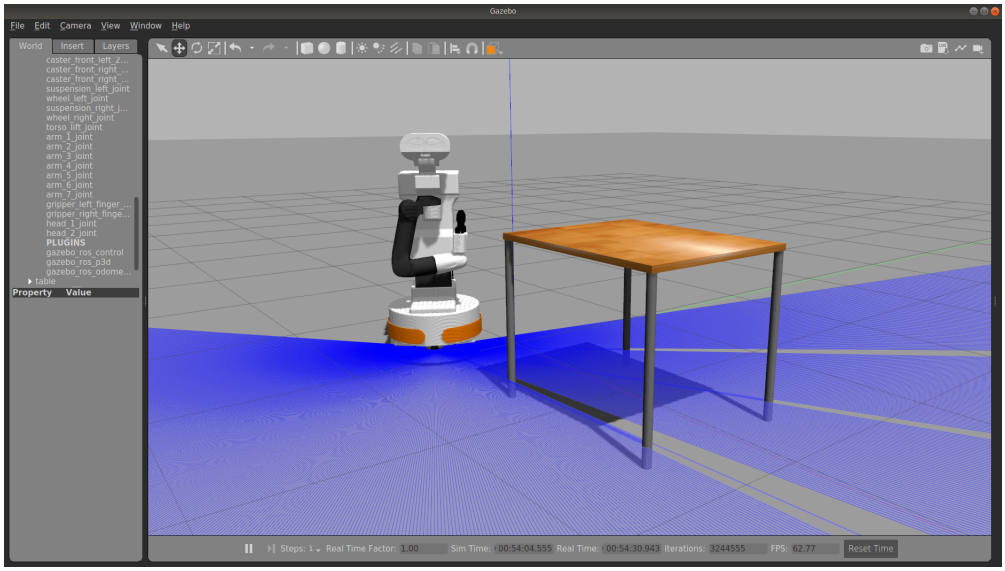


Figure 3.20: A screenshot of Gazebo simulator GUI. Gazebo lets simulate different types of robots in an environment they can interact with. In this case a wheeled robot with a robotic arm mounted on it is shown.

3.4.3 ROS package: *hector_quadrotor*

When it comes to simulating quadrotors in Gazebo, different options are available from other developers made available as open source. The choice fell on ROS package *hector_quadrotor* [40, 41], developed by Team Hector at Technische Universitat in Darmstadt [42], given that a lot of quadrotor Gazebo simulators are based on that package and is popular among researchers. This stack provides packages related to modeling, control and simulation of quadrotor UAV system and in addition enables the simulation of sensors such as sonar, laser scanner, camera and many internal others for state estimation and control. A frame from a demo of the quadrotor during an indoor mapping session by means of a laser scanner is shown in Figure 3.21.

Due to modularity of the package, is possible to mount different controllers on-board, such as attitude, velocity and position controller; the last one is actually the more external one in the nested structure already shown in previous section, letting the quadrotor, for the aim of this thesis, being controlled by position commands (such as waypoints) in the form of *navigation_msgs/PoseStamped* ROS messages.

3.4.4 PlotJuggler

In classical software debugging, the most used tool is an interactive debugger that stops code at inserted breakpoints so that variables at that time instant can be inspected. Embedded software, real-time applications, robotics and distributed systems are software domains where the most effective way to debug the code is through tracing and logging instead: stopping a task implementing a controller acting on a real physical system, such as setting a breakpoint in the middle of the control loop

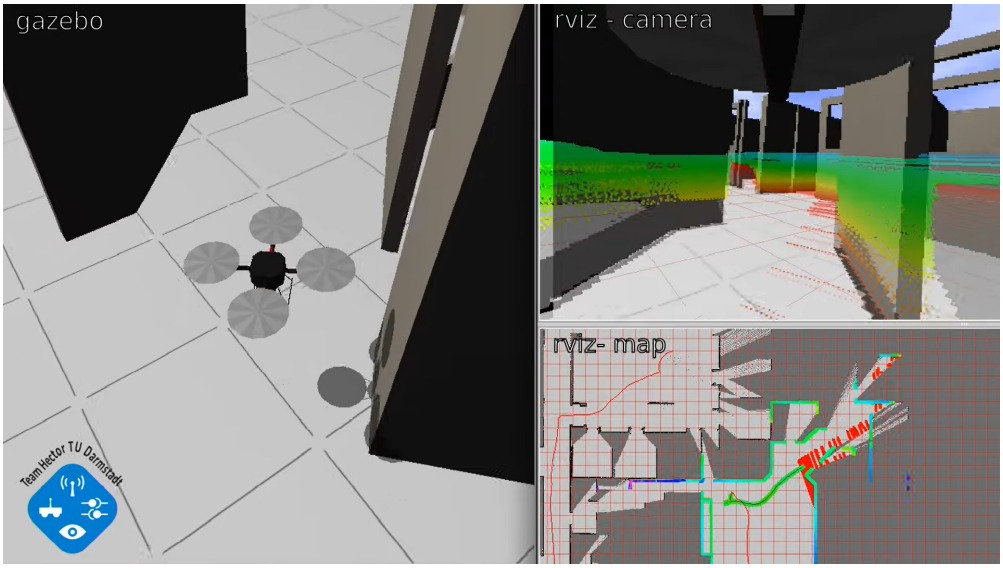


Figure 3.21: In this frame from [43], a demo of *hector_quadrotor* carrying out an indoor mapping session through a laser scanner is shown. The quadrotor mounts a camera onboard, which view can be seen in the upper-right part of the image. In lower-right part the view from Rviz, a tool for visualizing sensors data coming from ROS and Gazebo, is shown.

that stabilizes a drone, i s not possible and would lead to catastrophic consequences. ROS offers a package called *rqt_plot*, which has a rudimental interface and allows a primitive debug, but does not offer a lot of flexibility. A better tool that overcomes these limitations is PlotJuggler [44]. This tool, which interface is shown in Figure 3.22, allows to visualize timeseries in real time by subscribing to topics during while ROS is running or a posteriori by loading ROS bag files recorded previously for analysis. Furthermore, the GUI is pretty flexible, different layouts can be created and basic operations over signals such as time derivatives can be applied.

3.4.5 OpenCV and NumPy

When it comes to computer vision algorithms implementation, the most popular tool is OpenCV (Open Computer Vision Library), an open source computer vision and machine learning software library [45].

(a)

(b)

Figure 3.23: OpenCV and NumPy logos.

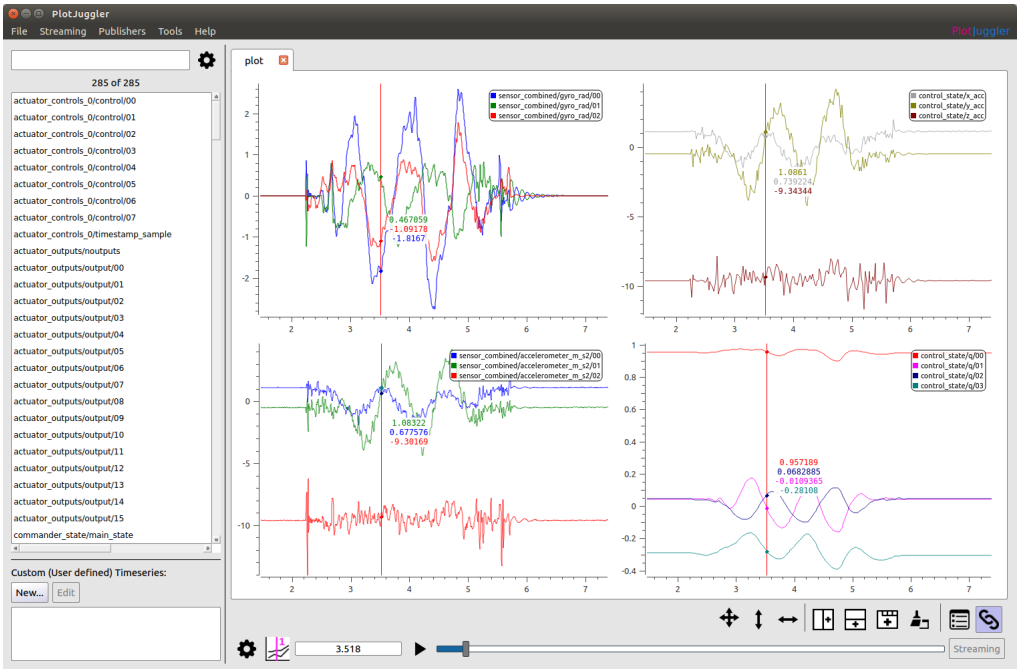


Figure 3.22: Example of PlotJuggler GUI.

This library (logo in Figure 3.23a), natively written in C++ and providing interfaces in C++, Python, Java and MATLAB, provides a huge number of optimized algorithms including both classic and state-of-the-art computer vision and machine learning algorithms and it is distributed under BSD license. These algorithms can be used for tasks such as recognizing faces, identifying objects, extracting features, image manipulation, retrieving 3D point clouds from stereo cameras, recognizing 3D structures, augmented reality and so on.
Among these algorithms, also Farnebäck Dense optical flow algorithm is implemented with pyramids method, which is used for this thesis' purpose. The chosen language is Python, supported also by ROS, because OpenCV Python's library is a light-weight wrapper for optimized compiled C/C++ functions [46]. Hence, the performance loss from C++ to Python can be considered negligible, so that it is possible to combine best features of both the languages, performance of C/C++ and simplicity of Python. Also, OpenCV-Python interface has full support to NumPy (logo in Figure 3.23b), a package for scientific computing in Python with multi-dimensional arrays, that it is also a wrapper around native C code. It is a highly optimized library which supports a wide variety of matrix operations, highly suitable for image processing and machine learning [47]. So combining both OpenCV functions and NumPy functions correctly, the code will be adequately fast.
Version of OpenCV used for this implementation is version 4.2.

CvBridge

Given that the processed images by means of Farnebäck optical flow algorithm implementation in OpenCV are coming from the virtual camera coming from simulated scenario and quadrotor, images must be translated from ROS message of type *sensor_msgs/Image* to OpenCV-NumPy array type. To accomplish this, an ad hoc made ROS package called CvBridge is used [48]. Its working principle is shown in Figure 3.24.

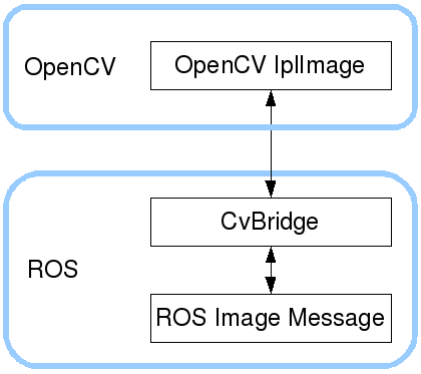


Figure 3.24: CvBridge working principle for translating from ROS Image messages to OpenCV images.

3.5 Test scenarios and performance measurement

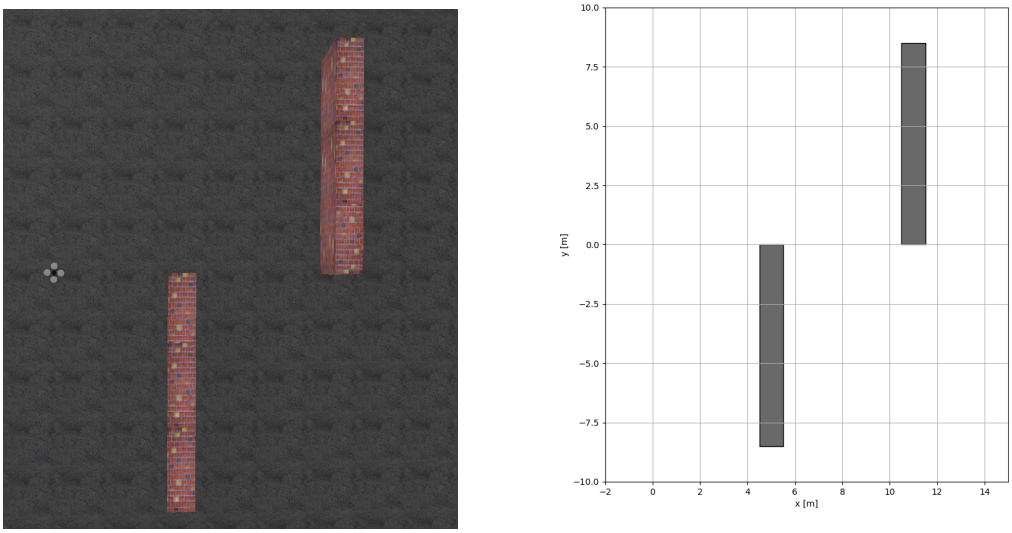
3.5.1 Test scenarios

In order to test the local path planning algorithm for obstacle avoidance, the effectiveness is proved in three scenarios that test horizontal avoidance, vertical avoidance and frontal collision avoidance. Considered Gazebo worlds also are modeled in Python's library Matplotlib in 2D, considering just the plane of interest, i.e. xy plane for horizontal and frontal obstacle avoidance and xz for vertical one. Both Gazebo worlds and Matplotlib maps are shown for each scenario in Figures 3.25, 3.26, 3.27. Notice that given that Gazebo runs high performance physics engines, the kinematic and dynamical model of the quadrotor is taken into account in simulations.

3.5.2 Metrics

Minimum distance from obstacles computation

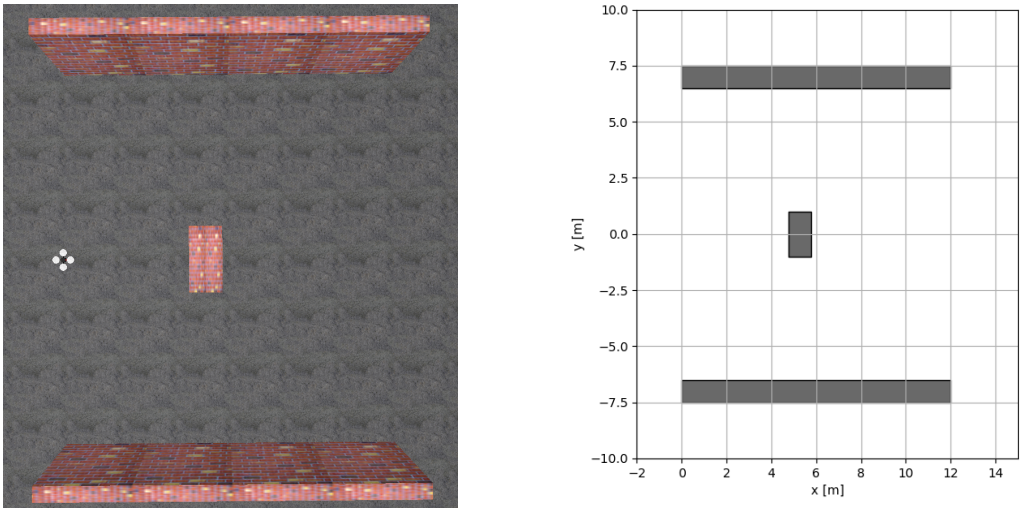
A useful performance indicator to evaluate the obstacle avoidance algorithm is the minimum distance from obstacles in the map. In order to produce plots of minimum distance of the drone from obstacles versus time of a given simulation experiment, this quantity must be evaluated for each position of the drone along its trajectory. Performances are measured afterwards from ROS bag files analysis, so map is known



(a) Gazebo world.

(b) Map of obstacles in xy plane.

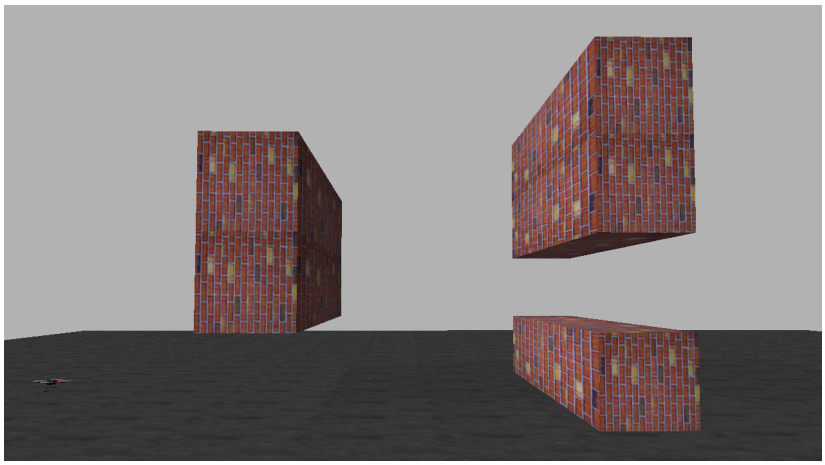
Figure 3.25: Test scenario for horizontal obstacle avoidance.



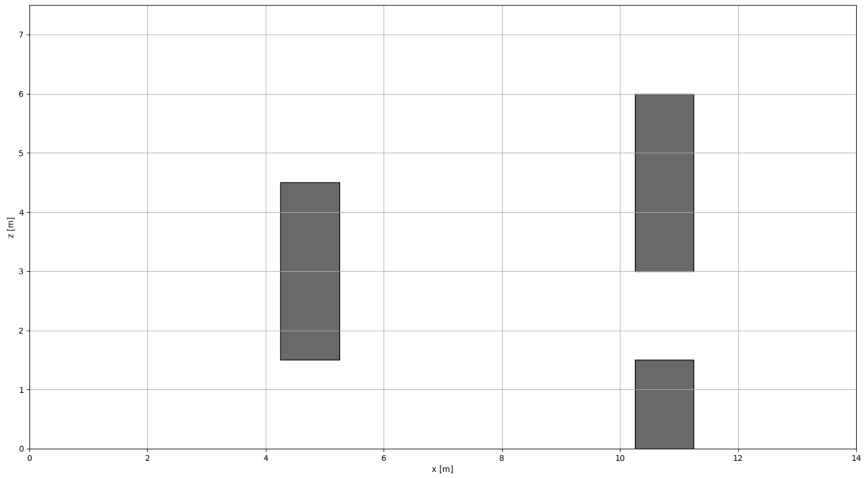
(a) Gazebo world.

(b) Map of obstacles in xy plane.

Figure 3.26: Test scenario for frontal obstacle avoidance.



(a) Gazebo world.



(b) Map of obstacles in xz plane.

Figure 3.27: Test scenario for vertical obstacle avoidance.

a priori; furthermore, depending on simulation scenario and the actual type of test, just a particular 2D plane is of interest (e.g. xy or xz), so obstacles can be represented as polygonal shapes composed of segments (in this particular case rectangles). This simplifies significantly the task of computing the minimum distance in a 2D plane, provided that every obstacle is represented as a polygon with known vertices. The problem is hence casted in the simpler task of computing, for each point in the trajectory, its distances from every segment in the map and choosing the smaller among those as the distance from the nearest obstacle.

A Python script has been made for extracting information from ROS bag files (drone's trajectory from */ground_truth/state* topic and waypoints sent from */waypoints* topic) and from a JSON file that describes the map for each scenario. The core algorithm of this performance analysis can be summed up in Algorithm 1.

Algorithm 1: Computing minimum distance of drone from obstacles

```
Input  : trajectory: Array containing timestamped positions of drone,
         segmentsList: List containing couples of points defining segments
Output: minimumDistanceOverTime: Array containing timestamped
         minimum distance from obstacles
for position in trajectory do
    for segment in segmentsList do
        distance = minDist(position, segment);
        distanceArray.append(distance);
    end
    minimumDistance = min(distanceArray);
    minimumDistanceOverTime.append(minimumDistance)
end
```

Then the obtained values of minimum distances from obstacles over time are plotted by means of *matplotlib.pyplot* Python module.
What emerges from the algorithm is that is crucial to compute correctly the distance from the segment and this is one in function *minDist(position, segment)*, which applies the following geometrical reasoning: given drone's position P and segment's extremities A and B, such as:

$$P = \begin{pmatrix} x_P \\ y_P \end{pmatrix}, \quad A = \begin{pmatrix} x_A \\ y_A \end{pmatrix}, \quad B = \begin{pmatrix} x_B \\ y_B \end{pmatrix}$$

the distance of point P from the line to which segment $\overline{AB}$ belongs and its orthogonal projection P' on it are computed. If this projected point belongs to segment $\overline{AB}$, the returned minimum distance is the one from the line, otherwise the smaller between the two distances from A and B is returned.
In figure 3.28 the geometric objects considered are shown.

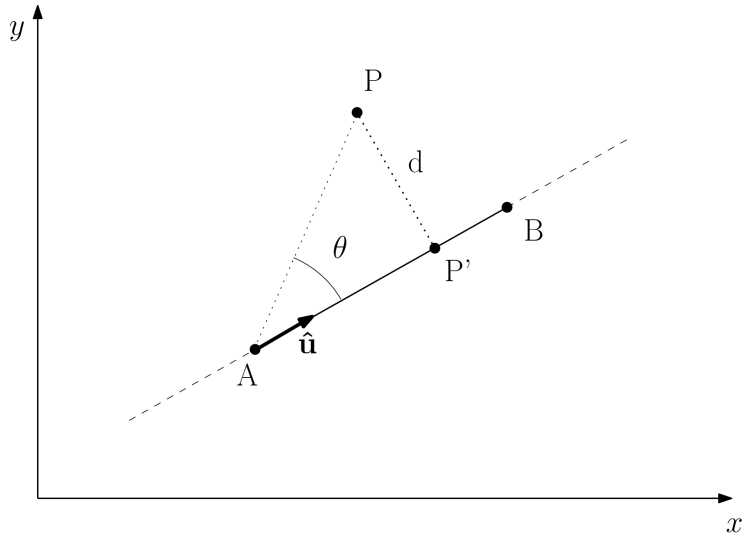


Figure 3.28: Computation of distance of a point from a segment.

Distance d of point P from line given by segment $\overline{AB}$ is first computed as the norm of cross product between the vector $\overrightarrow{AP}$ and the unit vector $\hat{\mathbf{u}}$.

$$d = \|\overrightarrow{AP}\| \sin(\theta) = \|\hat{\mathbf{u}} \times \overrightarrow{AP}\|, \qquad \hat{\mathbf{u}} = \frac{\overrightarrow{AB}}{\|\overrightarrow{AB}\|}$$

Then, in order to find P', the component of $\overrightarrow{AP}$ along $\hat{\mathbf{u}}$ direction is computed with a scalar product

$$\|\overrightarrow{AP'}\| = \overrightarrow{AP} \cdot \hat{\mathbf{u}} = (x_P - x_A)u_x + (y_P - y_A)u_y$$

and point P' can be determined as:

$$P' = A + \|\overrightarrow{AP'}\| = \begin{pmatrix} x_A + \|\overrightarrow{AP'}\|u_x \\ y_A + \|\overrightarrow{AP'}\|u_y \end{pmatrix} = \begin{pmatrix} x'_P \\ y'_P \end{pmatrix}$$

Now that coordinates of P' are known together with segment's extremities, it is possible to discriminate if point $P' \in \overline{AB}$. If this conditions is satisfied, minimum distance from the segment is the same from the line it belongs to d, otherwise it will be the smaller among the ones from the two endpoints A and B:

$$minDist(P, \overline{AB}) = \begin{cases} d & , \quad \text{if } (x_A \leq x_{P'} \leq x_B \quad \vee \quad x_B \leq x_{P'} \leq x_A) \\ & \quad \wedge \quad (y_A \leq y_{P'} \leq y_B \quad \vee \quad y_B \leq y_{P'} \leq y_A) \\ min(\overline{AP}, \overline{BP}) & , \quad \text{otherwise} \end{cases}$$

3.5.3 Performances of software in real-time

Given that real-time execution is another requirement, a measurement as precise as possible of code execution time and performance is necessary. Furthermore, an insight about which parts of the code increase computational burden would be convenient so that code optimization can be operated on computationally heavier functions; hence, using a "stopwatch technique" where just the time interval passed between two points in the execution of the code would not tell the whole story. In order to accomplish this, *software profiling* is what we are looking for.
Software profiling is the act of using software instrumentation (i.e. addition of code instructions to monitor components' performances dynamically, at runtime) to objectively measure the performances of an application, let those be resource use (e.g. CPU, memory), frequency or duration of function calls or wall execution time of part of the application. A software profile outputs then a profile that will be a statistical summary of the execution of functions and that can be interpreted, by means of third-party software, in a graphical way so that bottlenecks in performance can be spotted more easily. However, code instrumentation creates overhead, so it has a performance cost.
In general, software profilers can be divided in two main categories based on operating principle: deterministic (or event-based) and statistical profilers. The former type makes measurement when a certain event happens, such as function call, function leave or exception rise; these profilers hook function *call* and *return* (and more) events to calculate metrics, so they can precisely spot when functions start and end at the cost of adding more overhead. On the other hand, statistical profilers makes their measurements at regular time interval by sampling, by means of operating system's interrupts, the target program's call stack; sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed. More specific profiler types exist, but their application is out of the scope of this thesis.
A variety of software profilers is available for Python language. Actually, Python comes with a built-in module for profiling called cProfile, but this profiler cannot deal with multi-threaded software and asynchronous calls, which is ROS case. Hence, the chosen profiler is Yappi ("Yet Another Python Profiler") [49], which is threaded and coroutine aware. Yappi supports call graphs, which are able to store more information for each function call, because not just how many times the function was called or how long it took is stored, but also the whole call stack, including the function that called the considered one. Call graph format allows to create graphical representations of code performance so that information can be visualized and understood in a more immediate way. There are several call graphs visualizers available for visualizing Valgrind's callgrind format produced by Yappi. The one that has been chosen is KCacheGrind, available for Linux operating systems. An example of its GUI can be seen in Figure 3.29.

The software not only provides a list of called functions with detailed information, but also allows the visualization of time consumption in two formats: treemaps, where each function is represented as a box containing other boxes representing in

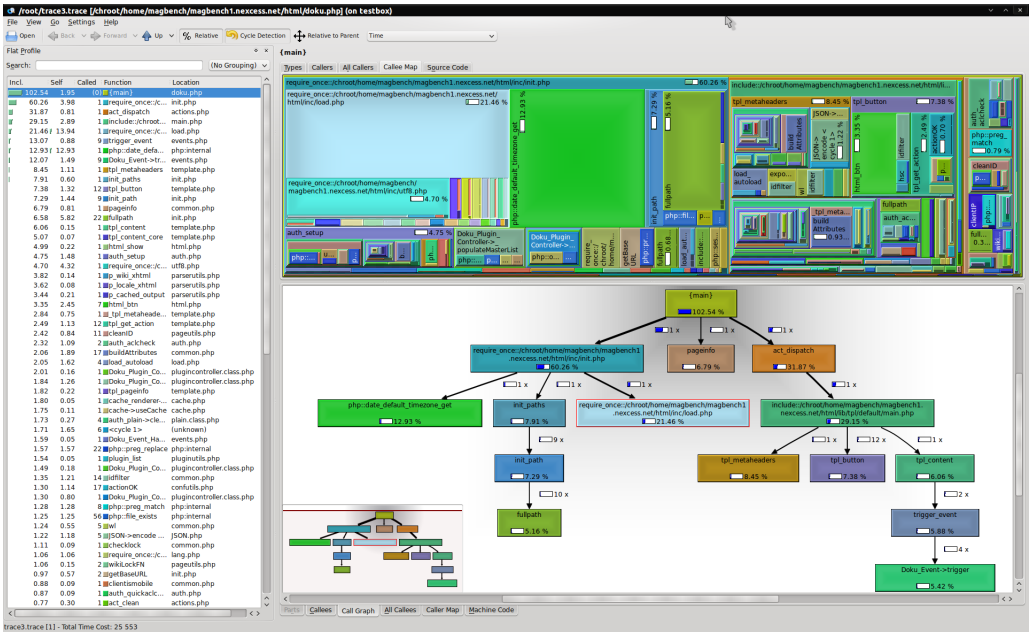


Figure 3.29: KCacheGrind is a GUI based tool made for visualizing software profilers outputs, which format is in the form of call graphs. Here a generic example is shown.

turn its subfunctions and each box's area is proportional to the respective function's execution time, and call graphs, which better highlight relationships between caller function and callee function.

Chapter 4

Proposed solution

Once spotted the main limitations of optical flow (OF) based obstacle avoidance algorithm available in literature, here the main contribution of this thesis is explained in details: a local path planning strategy for obstacle avoidance able to work in 3D, deal with frontal obstacles, that doesn't make a priori assumptions about environment structure, real-time implementable and deployable on onboard embedded systems.

4.1 Optical Flow based Obstacle Avoidance (OFOA) strategy

The proposed algorithm exploiting OF computed from a frontal monocular camera uses OF unbalance principle. Given that the drone is controlled in position, an intermediate waypoint is prepended to the given waypoints list that the drone has to follow, so that the obstacle is avoided successfully. Said intermediate waypoint's coordinates are computed using optical flow unbalances signals and optical flow expansion signal. This principle is also the one spotted in nature in some insects and birds behaviour: for instance, if objects seem to move faster on the right side of the field of view (FOV), that should mean obstacles on the right are at a smaller distance from us, hence to avoid them a movement to the left is required. The same holds for vertical obstacle avoidance: with a floating obstacle we will have faster optical movement in the upper part of our field of view, hence it is required to move downwards to avoid colliding with the obstacle. As far as frontally approaching obstacles are concerned, expansion of optical flow (EOF) signal is computed: as we approach an object frontally, we see it expanding in our FOV (i.e. expanding OF), so we have to stop and head left or right to avoid it.

4.1.1 OF unbalance computation

Optical flow magnitude unbalances must be obtained as signals over time, hence in order to do this, different areas in the image, called templates, must be defined. The defined templates can be seen highlighted in Figure 4.1.

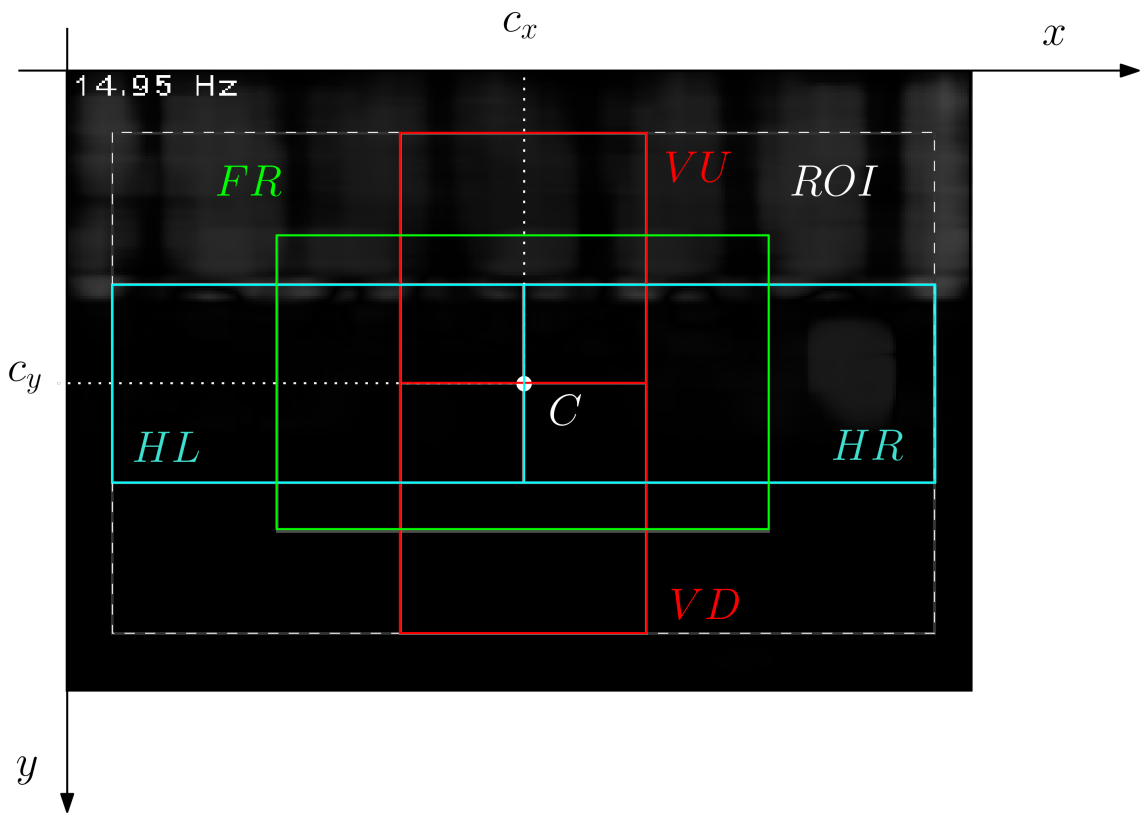


Figure 4.1: Templates geometry notation.

First, the region of interest (ROI) smaller than frame's dimensions is defined, because external parts of the frame contain objects that are not directly colliding with quadrotor. Hence the templates are defined inside ROI. For vertical unbalance, templates vertical-up (VU) and vertical-down (VD) are considered; for horizontal unbalance horizontal-left (HL) and horizontal-right (HR); for frontal avoidance a smaller template (FR) containing just obstacles directly in front of quadrotor is considered. Templates' geometry is defined around ROI's center, that is considered fixed with the frame's center

$$C = \left(\frac{W}{2}, \frac{H}{2}\right) = (c_x, c_y)$$

according to x and y notation in Figure 4.1, where $W = 320, H = 240$ in *hector_quadrotor*'s camera case.

Let's define **OF** vector field as:

$$\mathbf{OF}(x, y, t) = u(x, y, t)\hat{\mathbf{i}} + v(x, y, t)\hat{\mathbf{j}}$$

where (x, y) is the position of the pixel to which OF vector corresponds, computed for iteration at time instant t (discrete time, synchronous with main algorithm

iteration) and u and v are the horizontal and vertical vector components respectively. The field is then adequately compensated for self-motion produced OF, obtaining $\mathbf{OF_C}$ field, more information in subsection 4.1.1. Then, for each template, the respective **OF** vector field sub-region is selected, magnitude and its overall sum are computed for each time instant:

$$\sigma_{VU}(t) = \sum_{(x,y)\in VU} \|\mathbf{OF_C}(x,y,t)\|$$
$$\sigma_{VD}(t) = \sum_{(x,y)\in VD} \|\mathbf{OF_C}(x,y,t)\|$$
$$\sigma_{HL}(t) = \sum_{(x,y)\in HL} \|\mathbf{OF_C}(x,y,t)\|$$
$$\sigma_{HR}(t) = \sum_{(x,y)\in HR} \|\mathbf{OF_C}(x,y,t)\|$$

From magnitude total sum from each template, vertical and horizontal unbalance signals are computed:

$$e_V(t) = \sigma_{VD}(t) - \sigma_{VU}(t)$$
$$e_H(t) = \sigma_{HR}(t) - \sigma_{HL}(t)$$

The unbalance signals are very noisy and sometimes they can show spikes, hence in order to smooth these signals a Moving Mean Filter (MMF) is applied. MMF averages a discrete signal by producing as output at instant t the average of the last M signal's samples. Hence filtered versions of unbalance signals are produced:

$$\overline{e_V}(t) = \frac{1}{M}\sum_{i=0}^{M-1} e_V(t-i), \quad M = 3$$
$$\overline{e_H}(t) = \frac{1}{M}\sum_{i=0}^{M-1} e_H(t-i), \quad M = 3$$

Length of filter's memory M has been set by trial and error procedure.

Self-motion generated OF compensation

Given that the OF field generated is the result of the relative motion between the drone, hence the camera fixed to it, and the surrounding environment, quadrotor's movement is influencing the result. Hence, when the quadrotor is turning around z axis, there will be an OF horizontal component generated by this movement that is considered as a disturb for the aim of our algorithm. In the same way, pitching occurring when quadrotor passes from hovering to motion in order to reach next

waypoint or gaining height are movement that generate an unwanted vertical component of OF field. Given that we are interested just in the OF generated by forward motion to avoid obstacles, these components must be compensated.

The best way to compensate these self-generated OF components is to reduce in magnitude, by multiplying by a time-varying coefficient which value can span between 0 and 1, the whole field, affecting separately vertical and horizontal components with respective factors. Hence, recalling **OF** field is defined as:

$$\mathbf{OF}(x, y, t) = u(x, y, t)\hat{\mathbf{i}} + v(x, y, t)\hat{\mathbf{j}}$$

a compensated version

$$\mathbf{OF_C}(x, y, t) = u_C(x, y, t)\hat{\mathbf{i}} + v_C(x, y, t)\hat{\mathbf{j}}$$

is defined.

The horizontal component is affected just by yaw rate $\dot{\psi}(t)$ at time t, considering that quadrotor will never move sideways. Therefore the compensation is operated by the following equation:

$$u_C(x, y, t) = u(x, y, t)\frac{1}{1 + K_{C,yaw}|\dot{\psi}(t)|}$$

The 1 added at denominator is for avoiding singularity that would occur for very small values of yaw rate, while absolute value is for keeping the coefficient positive and influence the compensation in the same way independently from yaw rotation direction. In a similar way the vertical components of the **OF** field are compensated for the effect induced by climbing rate $\dot{h}(t)$ and pitch rate $q(t)$:

$$v_C(x, y, t) = v(x, y, t)\frac{1}{1 + K_{C,lin\ z}|\dot{h}(t)| + K_{C,pitch}|q(t)|}$$

By means of a trial and error procedure conducted with the help of PlotJuggler, the coefficients have been tuned and their values are reported in Table 4.1.

Table 4.1: Coefficient for self-motion OF compensation.

Coefficient	Value
$K_{C,yaw}$	20
$K_{C,lin\ z}$	8
$K_{C,pitch}$	2

4.1.2 Intermediate waypoint computation

With the obtained signals, both for horizontal of vertical avoidance and for frontally approaching obstacles, the situation can be evaluated and the OFOA strategy can be triggered if necessary. In order to understand that, thresholds are defined and if any of the three signals goes above respective threshold, the OFOA strategy is triggered and the adequate intermediate waypoint is computed. This is done by using the signals in order to compute the polar coordinates in the mobile (vehicle) reference frame and then translating it in cartesian coordinate in the fixed (world) reference frame, as shown in Figure 4.2. Notice that while in previous chapter some reference frames where defined, those were used for deriving the kinematic and dynamic model of quadrotor, while here reference frames used are the ones used in Gazebo simulation environment (with z axis pointing away from Earth's center), given that these equations have the aim of computing waypoints for local path planning to be referenced in the simulator.

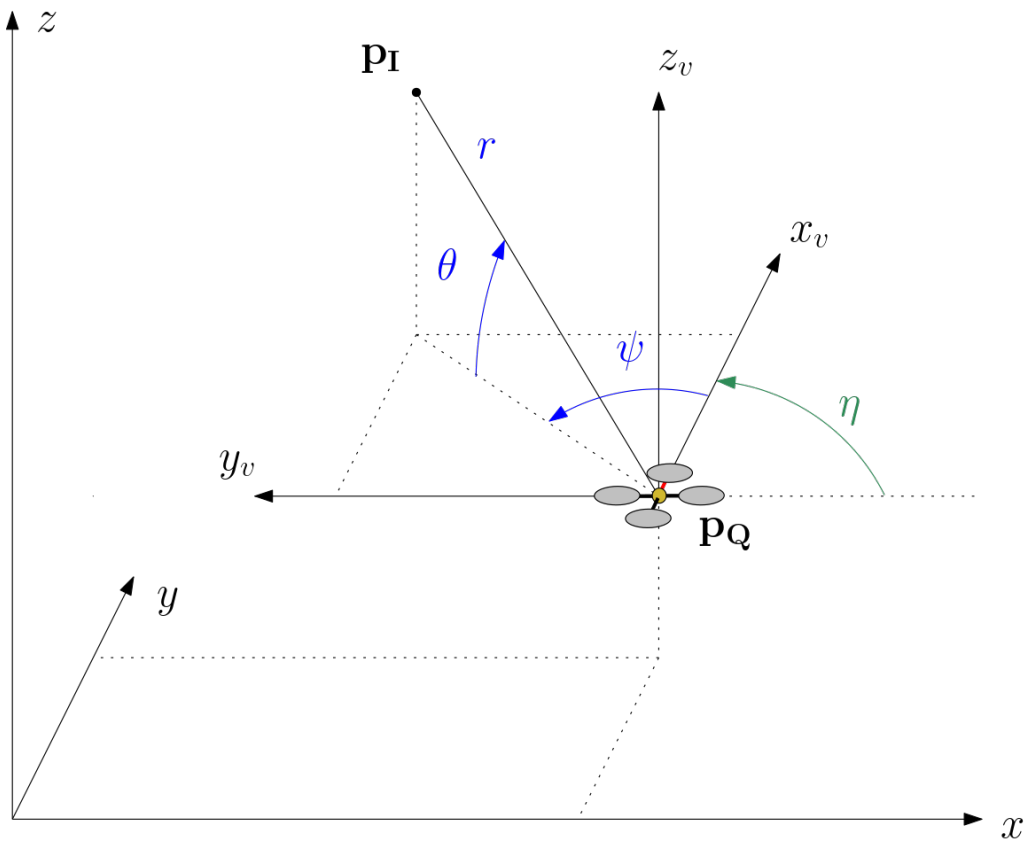


Figure 4.2: Intermediate waypoint computation for obstacle avoidance.

Depending on which avoidance is required, horizontal/vertical or frontal, different equations for computing polar coordinates in vehicle reference frame are used.

For horizontal or vertical avoidance, the following equations for computing r, θ and ψ at time instant t are applied:

$$
\begin{aligned}
&r = r_{V,H} \\
&\theta(t) = \begin{cases} K_{P,V}\ \overline{e_V}(t), & \text{if } \overline{e_V}(t) > \tau_V \quad \wedge \quad \left(-\frac{\pi}{2} \leq K_{P,V}\ \overline{e_V}(t) \leq +\frac{\pi}{2}\right) \\ -\frac{\pi}{2}, & \text{if } \overline{e_V}(t) > \tau_V \quad \wedge \quad K_{P,V}\ \overline{e_V}(t) < -\frac{\pi}{2} \\ +\frac{\pi}{2}, & \text{if } \overline{e_V}(t) > \tau_V \quad \wedge \quad K_{P,V}\ \overline{e_V}(t) > +\frac{\pi}{2} \\ 0, & \text{if } \overline{e_V}(t) \leq \tau_V \end{cases} \\
&\psi(t) = \begin{cases} K_{P,H}\ \overline{e_H}(t), & \text{if } \overline{e_H}(t) > \tau_H \quad \wedge \quad \left(-\frac{\pi}{2} \leq K_{P,H}\ \overline{e_H}(t) \leq +\frac{\pi}{2}\right) \\ -\frac{\pi}{2}, & \text{if } \overline{e_H}(t) > \tau_H \quad \wedge \quad K_{P,H}\ \overline{e_H}(t) < -\frac{\pi}{2} \\ +\frac{\pi}{2}, & \text{if } \overline{e_H}(t) > \tau_H \quad \wedge \quad K_{P,H}\ \overline{e_H}(t) > +\frac{\pi}{2} \\ 0, & \text{if } \overline{e_H}(t) \leq \tau_H \end{cases}
\end{aligned}
$$

Basically, the intermediate waypoint will always be at a specified radius of distance from current quadrotor's position and the computed angles, if the the respective unbalance signals are above the defined thresholds τ_V or τ_H, are proportional to signal magnitude in that time instant t of a gain $K_{P,V}$ or $K_{P,H}$. Then, if the computed values are exceeding $\pm\frac{\pi}{2}$, they are clamped to that value.

For frontal avoidance instead, given that the obstacle cannot just be circumnavigated but requires a right angle turn to be avoided, equations for for computing r, θ and ψ at time instant t assume this form:

$$
\begin{aligned}
&r = r_F \\
&\theta(t) = 0 \\
&\psi(t) = \begin{cases} -\frac{\pi}{2}, & \overline{e_H}(t) \leq 0 \\ +\frac{\pi}{2}, & \overline{e_H}(t) > 0 \end{cases}
\end{aligned}
$$

The values for thresholds, gains and radii has been found by trial and error procedure and are reported in Table 4.2.

Table 4.2: Parameter values used in OFOA strategy

Coefficient	Value
$r_{V,H}$	0.7
r_F	1.7
τ_V	3500
τ_H	5000
τ_F	2300
$K_{P,H}$	4e-5
$K_{P,V}$	2e-4

The frontal avoidance has priority over horizontal and vertical avoidance. Hence if, for instance, both $EOF(t)$ and $\overline{e_V}(t)$ are above the respective thresholds at a certain time instant t, only the frontal avoidance is applied and the second set of equations will be used.

Finally, given that now the polar coordinates in the vehicle reference frame are computed, the translation in cartesian coordinates in the world reference frame is needed in order to feed this point as a reference in position to the quadrotor's controller. Defining $\eta(t)$ as the heading of the quadrotor with respect to world reference frame's x axis, $\mathbf{p_I}$ as the intermediate waypoint in world reference frame and $\mathbf{p_Q}$ as the quadrotor's position, coordinates of $\mathbf{p_I}$ are computed as:

$$\begin{aligned} x_I(t) &= r\cos\theta(t)\cos(\psi(t)+\eta(t)) + x_Q(t) \\ y_I(t) &= r\cos\theta(t)\sin(\psi(t)+\eta(t)) + y_Q(t) \\ z_I(t) &= r\sin\theta(t) + z_Q(t) \end{aligned}$$

The computed waypoint is now prepended to the stored waypoints list and will be used as the next reference in order to avoid the obstacle.

4.2 OFOA Algorithm software implementation

The previously formalized strategy for obstacle avoidance is implemented by means of ROS and tested in Gazebo simulation environment. The algorithm is implemented as a Python class that creates a ROS node called */ofoa_manager* which aim is to run cyclically the algorithm at a fixed frequency (5 Hz) by associating the main algorithm's function to a ROS Timer object. For feeding inputs to the algorithm, the node subscribes to different topics and every time a new message arrives from a subscribed topic the appropriate callback function is called and the retrieved information is stored in the respective lass attribute. Hence, when the main algorithm is called, the inputs used are always the latest received. When algorithm computation is completed, the produced outputs, such as control commands and state information, are published on appropriate topics.
A graph of nodes */gazebo* and */ofoa_manager* and main topics (not all of them) used during a simulation session can be seen in Figure 4.3, while a detail with just inputs and outputs of */ofoa_manager* can be seen in Figure 4.4.

The node subscribes to topics :

- *front_cam/camera/image* for retrieving image frames coming from frontal monocular camera as message type *sensors_msgs/Image*. Images are converted from Image messages to OpenCV format by means of a ROS package called CvBridge;
- *waypoints* for receiving next waypoints produced and sent by a global path planning algorithm as message type *geometry_msgs/Point*, and appending them to the internally stored waypoint's list;

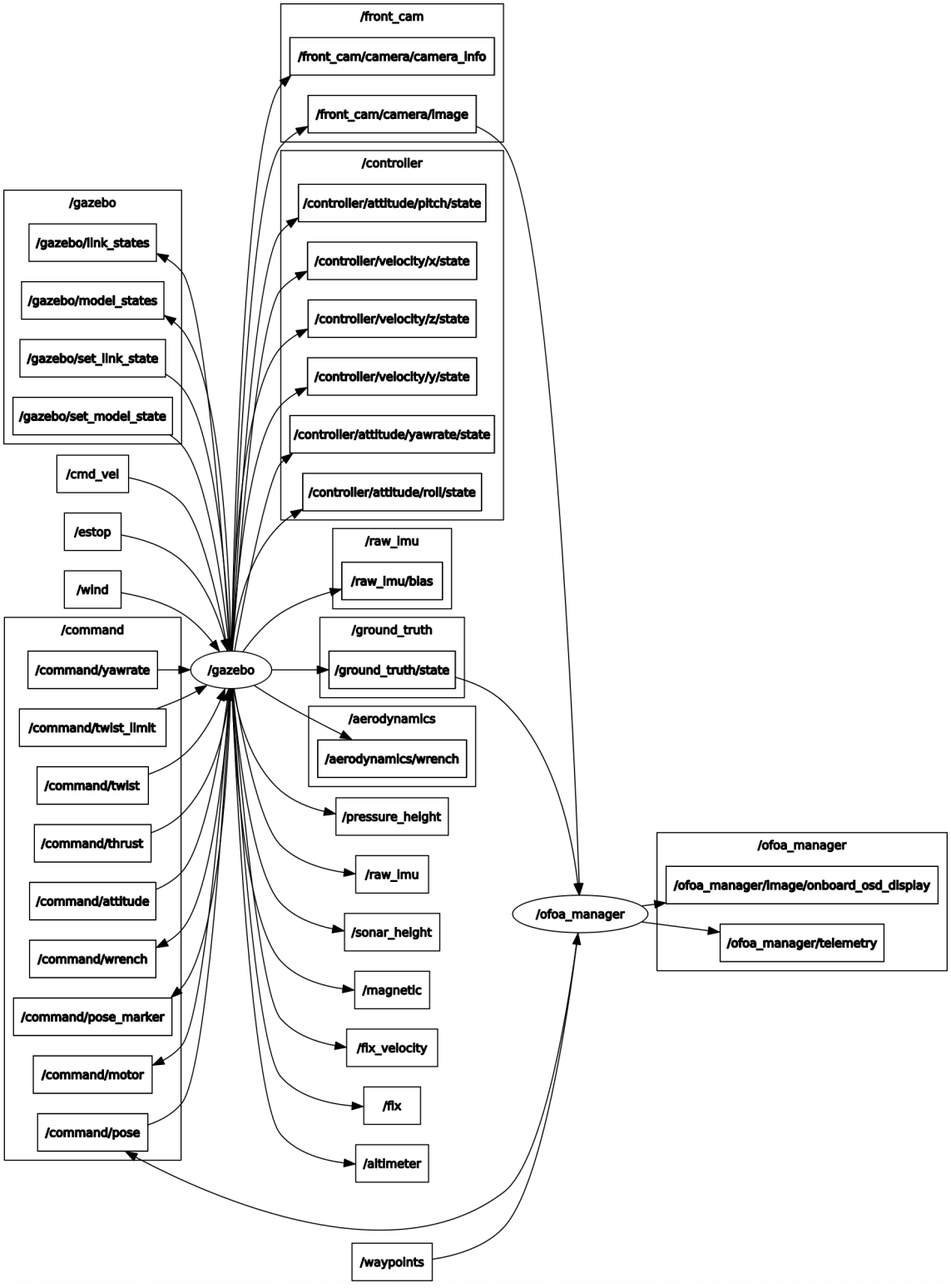


Figure 4.3: Graph representing ROS nodes and topics active during a simulation session.

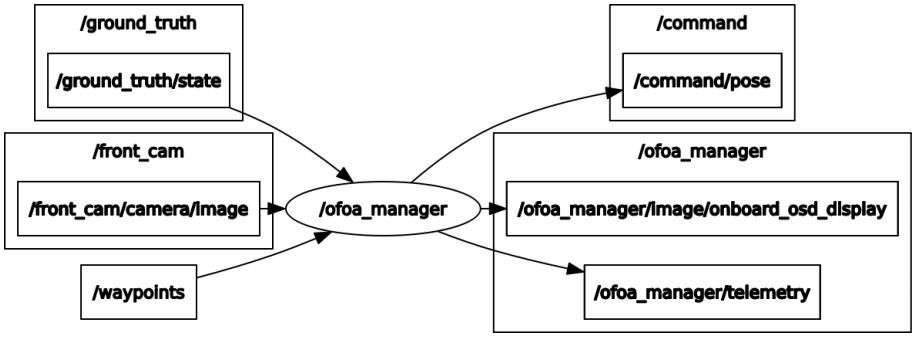


Figure 4.4: Graph of inputs an outputs for node */ofoa_manager*.

- */ground_truth/state* to retrieve its own pose and velocities (linear and angular), as message type *nav_msgs/Odometry*.

The node also publishes commands and information over topics such as:

- *command/pose* for sending position and orientation reference to flight controller in the form of *geometry_msgs/PoseStamped* messages;
- *ofoa_manager/image/onboard_osd_display* for streaming what the OF algorithm sees with some additional information such as type of obstacles detected, frame rate and templates geometry, all as *sensors_msgs/Image* messages. An example of that can be seen in Figure 4.5;
- *ofoa_manager/telemetry* to send telemetry message for retrieving information about the OFOA manager state; telemetry messages are of custom defined type *hector_ofoa/OFOAManagerTelemetry*, useful for real-time debugging by means of PlotJuggler.

The the class method containing the main OFOA algorithm and run at the fixed ROS Timer's frequency follows the working principle shown in the flowchart in Figure 4.6.

The function can be divided mainly in 3 phases: status evaluation, control command computation, produced outputs publishing.
In the status evaluation phase the gathered information stored in class' attributes is used to compute different quantities. From last two frames obtained by frontal camera, OF field is computed and then compensated. Then from each template, overall magnitude sum signals are computed as well as OF unbalance signals; TTC and then EOF are then computed from obtained OF field. Then, the obtained signals are compared with respective thresholds to identify the kind of avoidance is required, also taking into account if the obstacle avoidance is enabled, and respective flags are set or cleared.
In the control command computation the previously evaluated flags are used to decide if and which kind of avoidance is required. First, the algorithm checks if the quadrotor has arrived near the desired waypoint within a certain radius and in that

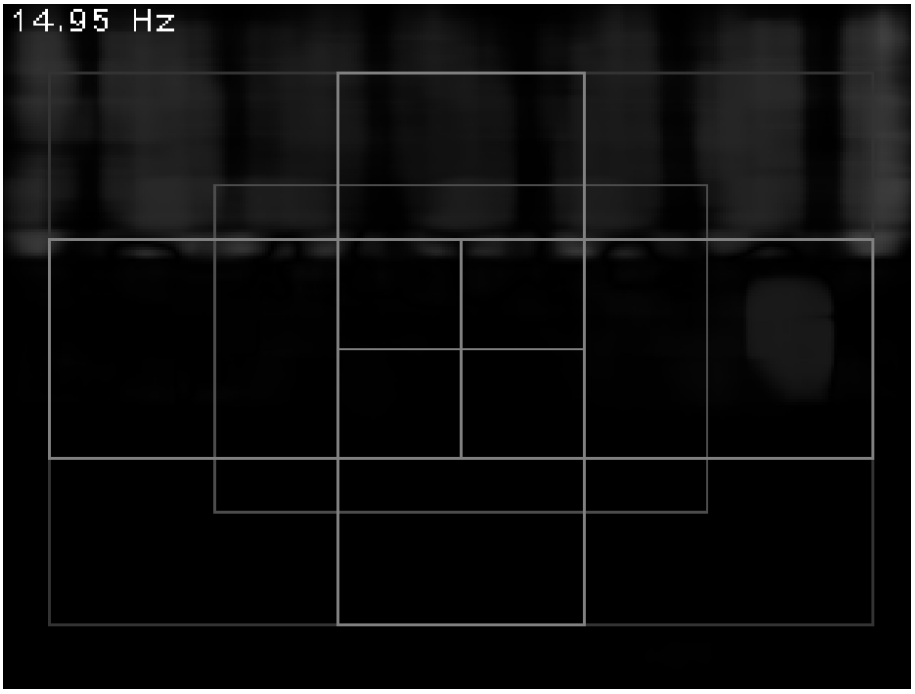


Figure 4.5: Screenshot of OFOA manager node OSD images streamed on topic */ofoa_manager/image/onboard_osd_display*.

case pops the waypoint from the list, moving to the next one (if list is not empty afterwards). Then, if obstacle avoidance is not required or the drone is already heading to a previously computed and appended intermediate waypoint, the quadrotor heads to that point. If otherwise obstacle avoidance is required, the previously stated equations are used to compute correctly the waypoint required depending on which condition occurred, with frontal avoidance having higher priority, and then the waypoint is appended to stored list, so that it can be set as position reference. A flag is then set to know that the next waypoint is an intermediate one, preventing the algorithm to compute a new one at each iteration and appending it to the list. When the quadrotor then sets a reference pose, whichever type of waypoint is moving towards, first hovers in place and turns towards the waypoint, then starts moving once the alignment has been achieved under a certain tolerance.
In the last phase, produced outputs are published over respective topics.
In order to measure actual execution time, a timer is started when the OFOA algorithm function is called and is stopped before publishing outputs, so that it can appropriately being superimposed by OSD function and streamed in telemetry messages. The cycle is then repeated when next call operated by ROS timer object will occur.

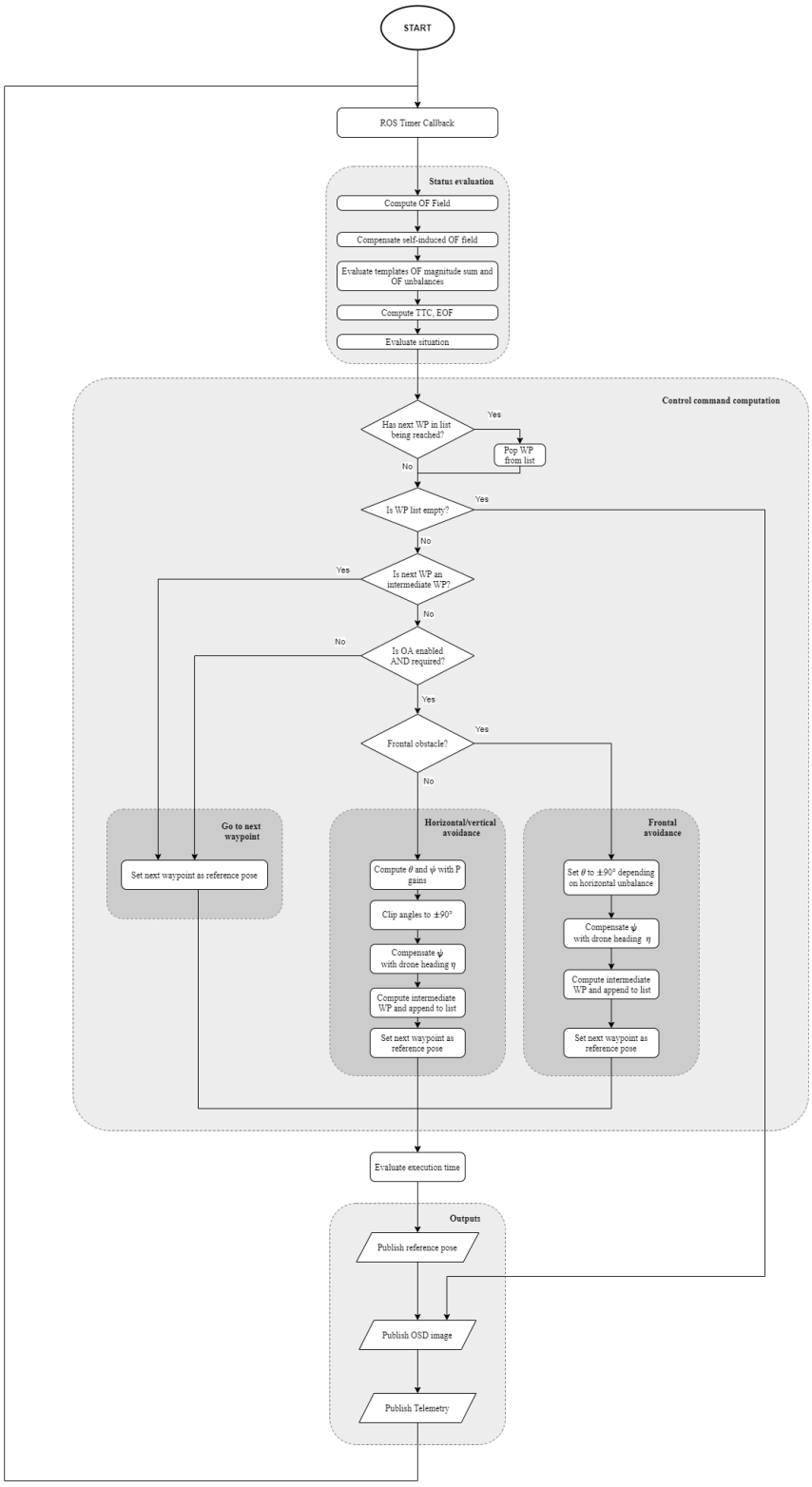


Figure 4.6: Flowchart representing OFOA algorithm Python function, called at a fixed rate in order to solve the obstacle avoidance problem in real-time.

4.2.1 EOF computation

Here the particular software implementation for EOF computation so that optimized operations can be exploited is explained. In order to compute the EOF for frontal obstacle avoidance, the used equations are the ones already introduced in previous chapter, here reported as reference.

$$EOF_i = \frac{OF_{DIV,i}}{\|\mathbf{r_i}\|} = \frac{\mathbf{OF_i} \cdot \mathbf{r_i}}{\|\mathbf{r_i}\|^2}$$
$$EOF = \sum_i^N EOF_i$$

Given that the problem has to be solved for each pixel in the FR template, the most efficient way to implement that in code avoiding the use of for loops is to vectorize the algorithm so that NumPy optimized code can be used. In practice, instead of having a $W \times H \times 2$ tensor OF that holds the OF field information for the frontal template of width W and height H and holds the u_i and v_i vectors components for the i^{th} pixel in the third dimension, the tensor is reshaped to a "column vector" OF_{col} (actually a 2D matrix because each row represents a OF vector in its components u_i and v_i) with shape $N \times 2$, where $N = W \times H$.

$$\mathbf{OF} = \begin{pmatrix} (u,v)_{1,1} & \dots & (u,v)_{1,W} \\ \vdots & \ddots & \\ (u,v)_{H,1} & \dots & (u,v)_{H,W} \end{pmatrix} \quad \Rightarrow \quad \mathbf{OF_{col}} = \begin{pmatrix} u_1 & v_1 \\ \vdots & \vdots \\ u_i & v_i \\ \vdots & \vdots \\ u_N & v_N \end{pmatrix}$$

In this way, operating by rows we compute the result by using optimized operations. The same reasoning holds for each quantity involved in the computation. A $N \times 2$ array containing indices of pixels' positions respective to OF_{col} is built. Then the array $\mathbf{r}$ containing the distance vectors from FOE for each pixel is obtained by subtracting FOE coordinates from the previously obtained array of indices.

$$\mathbf{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_i \\ \vdots \\ r_N \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_i & y_i \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix} - \begin{pmatrix} FOE_x & FOE_y \end{pmatrix}$$

Hence $\|\mathbf{r}\|$ is obtained by applying euclidean norm to each row. Then the dot product $\mathbf{OF} \cdot \mathbf{r}$ is also applied row by row with optimized operations and element-wise division is computed between $\mathbf{OF} \cdot \mathbf{r}$ and $\|\mathbf{r}\|^2$ in order to obtain the EOF_i for each pixel. The total sum is then computed, obtaining the final value of EOF for that time instant.

Chapter 5

Results

In order to evaluate performances of the algorithm proposed as a solution to the stated problem, results from simulated experiments setup are analyzed as far as two aspects are concerned: capability of obstacle avoidance by keeping an adequate distance and real-time software performances.

5.1 OFOA algorithm effectiveness in simulated test scenarios

From previously defined Gazebo test worlds, simulations are run, each one with different starting point, and recorded in ROS bag files so that analysis can be carried out a posteriori.
For each scenario, 20 simulations are run with randomly generated starting points in the plane of interest according to a 2D Gaussian distribution; a further one with the origin $(0, 0, 0)$ as starting point is considered. Normal distribution parameters used for each simulation are reported in respective subsection.
The designed Python script produces two kinds of plots. The first one is a plot of all simulations' trajectories with obstacle profiles and an ellipse which principal semi-axes are 3σ of the randomly distributed starting point displacement along that direction, highlighting hence the 99.7% of the possible starting locations. The second kind of plots shows, for each run, the minimum distance of the drone from the nearest obstacle. In order to compute success rate, the size of the quadrotor is considered: the collision is determined if the nearest obstacle is below a certain distance from the center of the quadrotor. This threshold is set as the half of the size of the quadrotor along x or y in horizontal plane avoidance scenarios or along z in vertical plane avoidance scenarios, incremented of 30% as a sort of safety factor. Hence, in the horizontal plane the threshold for successful obstacle avoidance is set at 0.52 m while in the vertical plane is set at 0.14 m. A dashed line is plotted to denote threshold's value.

In the following subsections, results are quantified by the computation of success

rate, minimum distance from nearest obstacle among all successful runs, average minimum distance among all runs, standard deviation of minimum distance among all runs.

5.1.1 Lateral obstacle avoidance on xy plane

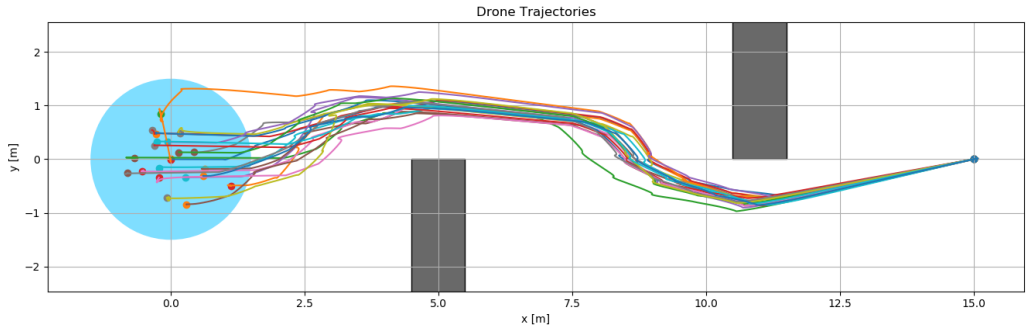
Results obtained in this scenario are plotted in Figure 5.1 and summed up in Table 5.1. As can be seen, the drone successfully reaches commanded waypoint by correctly avoiding lateral obstacles.

5.1.2 Frontal obstacle avoidance on xy plane

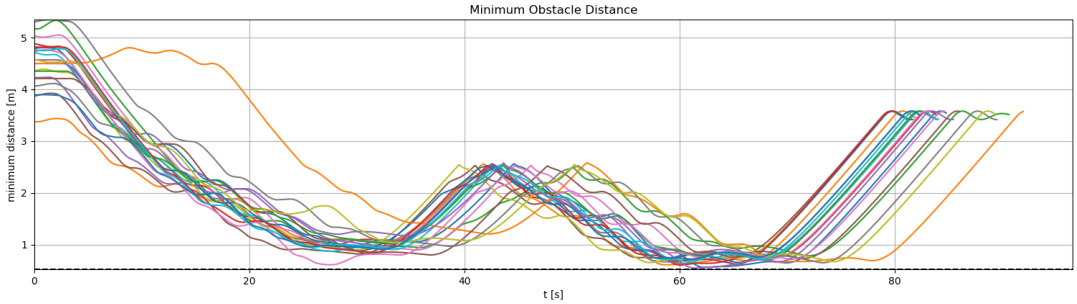
Results obtained in frontal obstacle scenario are plotted in Figure 5.2 and summed up in Table 5.2.
As shown, in this case success rate is not 100% because of the simulation run in which the frontal obstacle wasn't detected as such, leading the drone to avoid it by means of horizontal unbalance, that in the considered situation didn't provide high enough unbalance to produce a sufficiently wide angle, leading it to crash.

5.1.3 Vertical obstacle avoidance (floating obstacle and horizontal slit) on xz plane

Results obtained in vertical obstacle scenario are plotted in Figure 5.3 and summed up in Table 5.3.
This scenario tests the capability of the algorithm to avoid both floating obstacles and ground obstacles which can be climbed over, without crashing in another upper obstacle, hence passing through a slit.



(a) Trajectories of all simulations.

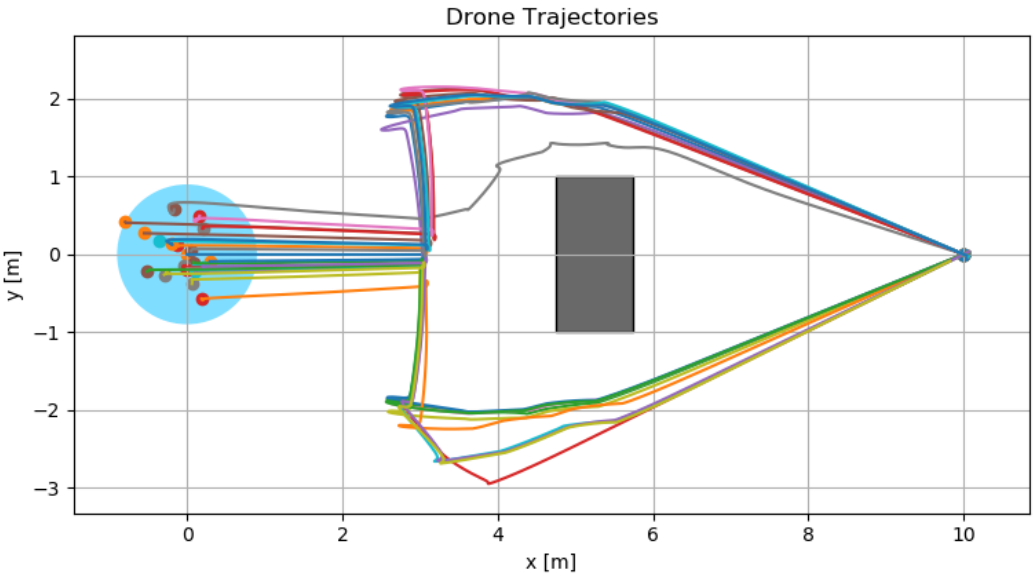


(b) Minimum distance form nearest obstacle.

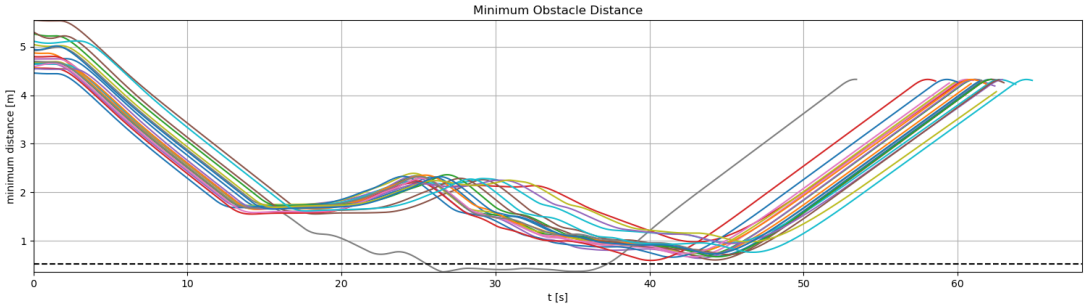
Figure 5.1: Plots of results obtained from simulations run in lateral obstacle avoidance scenario in horizontal plane.

Table 5.1: Summary of results from simulations run in lateral obstacle avoidance scenario.

Starting point	$X \sim \mathcal{N}(0, 0.5^2)$ m
	$Y \sim \mathcal{N}(0, 0.5^2)$ m
Success rate	100 %
Minimum distance	0.54 m
Average minimum distance	0.66 m
Standard deviation	0.06 m



(a) Trajectories of all simulations.

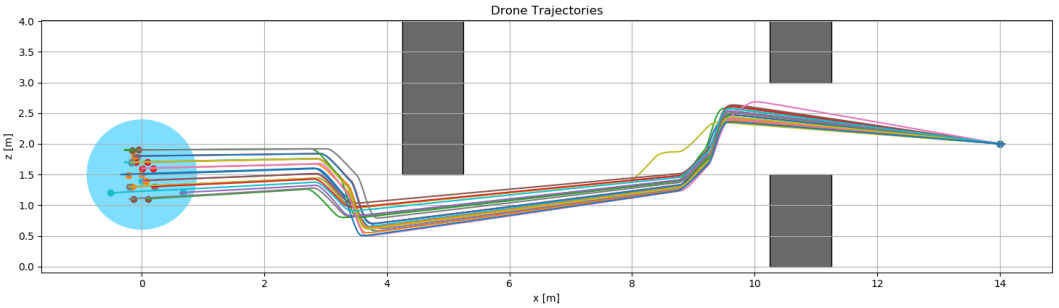


(b) Minimum distance form nearest obstacle.

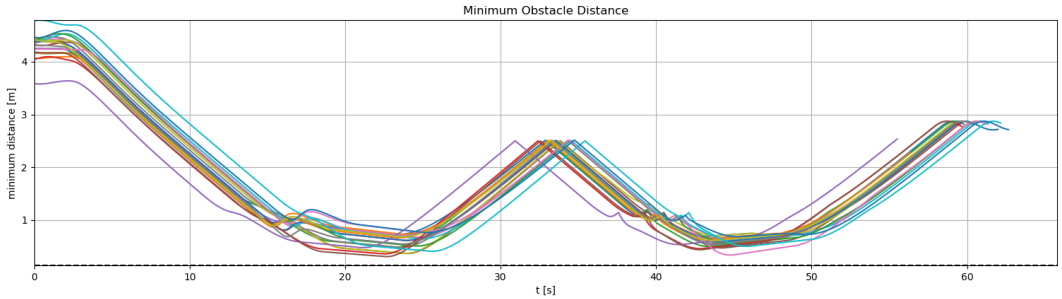
Figure 5.2: Plots of results obtained from simulations run in frontal obstacle avoidance scenario in horizontal plane.

Table 5.2: Summary of results from simulations run in frontal obstacle avoidance scenario.

Starting point	$X \sim \mathcal{N}(0, 0.3^2)$ m
	$Y \sim \mathcal{N}(0, 0.3^2)$ m
Success rate	95.2 %
Minimum distance	0.59 m
Average minimum distance	0.73 m
Standard deviation	0.14 m



(a) Trajectories of all simulations.



(b) Minimum distance from nearest obstacle.

Figure 5.3: Plots of results obtained from simulations run in vertical obstacle avoidance scenario in vertical plane.

Table 5.3: Summary of results from simulations run in vertical obstacle avoidance scenario.

Starting point	$X \sim \mathcal{N}(0, 0.3^2)$ m
	$Z \sim \mathcal{N}(1.5, 0.3^2)$ m
Success rate	100 %
Minimum distance	0.31 m
Average minimum distance	0.53 m
Standard deviation	0.12 m

5.2 Real-time software performances

Given that a requirement that this thesis aims to satisfy is real-time implementability of the algorithm, not only that quadrotor properly avoids obstacles is required, but also a reasonable execution time for the main obstacle avoidance algorithm must be guaranteed.
Software performance analysis is here conducted from two different aspects: code optimization by software profiling and algorithm execution time statistics on different platforms.

5.2.1 Software profiling and code optimization

In order to let the algorithm being real-time implementable, an analysis of where the most part of execution time is spent in the code has to be carried out. The method used to identify bottlenecks and try to optimize slower parts of code is software profiling, which gives an insight about the relative cost of each code part. To accomplish this task, software profiling tools such as Yappi Python library are used, so that code is instrumented in order to measure execution time of each thread and time spent in function calls. Main obstacle avoidance algorithm's ROS node is instrumented with Yappi and run for around 60 s and a profile file in Callgrind format is generated at the end of the experiment. This is afterwards fed to KCacheGrind call graph visualizer in order to see how long does each function takes to execute.
Experiments are carried out by deploying the algorithm's ROS node on the development platform NVIDIA Jetson TX1, shown in Figure 5.4.

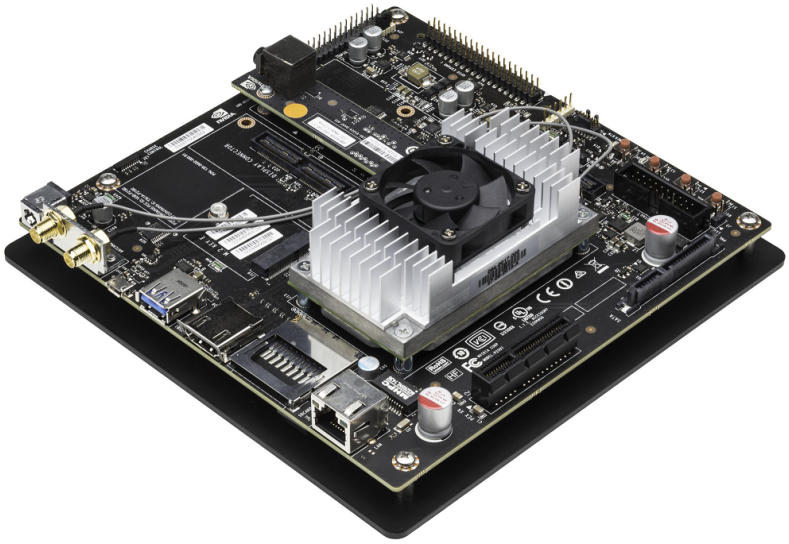


Figure 5.4: NVIDIA Jetson TX1 Development Kit platform for vision computing. The board is hosting the Jetson TX1 module, which offers high computational capabilities with 256 core GPU and low power consumption.

The board is based on Jetson TX1 module, which offers an NVIDIA Maxwell GPU with 256 CUDA cores, a quad-core ARM Cortex-A57 MCore CPU running

at 1.9 GHz, 4 GB LPDDR4 RAM and 16 GB flash storage. The OS running on it is Ubuntu 14.04 and multiple communication interfaces are available, such as WiFi module. The module alone weights 88 g and consumes $6.5 - 15$ W.
Given that ROS is a distributed system by nature, running ROS nodes on different platforms simultaneously letting them communicating with each other is made easy, as long as every platform hosting ROS nodes is connected on the same LAN and every ROS node knows ROS master's IP address and port.

The setup for the hardware-in-the-loop (HIL) simulation is shown in Figure 5.5.

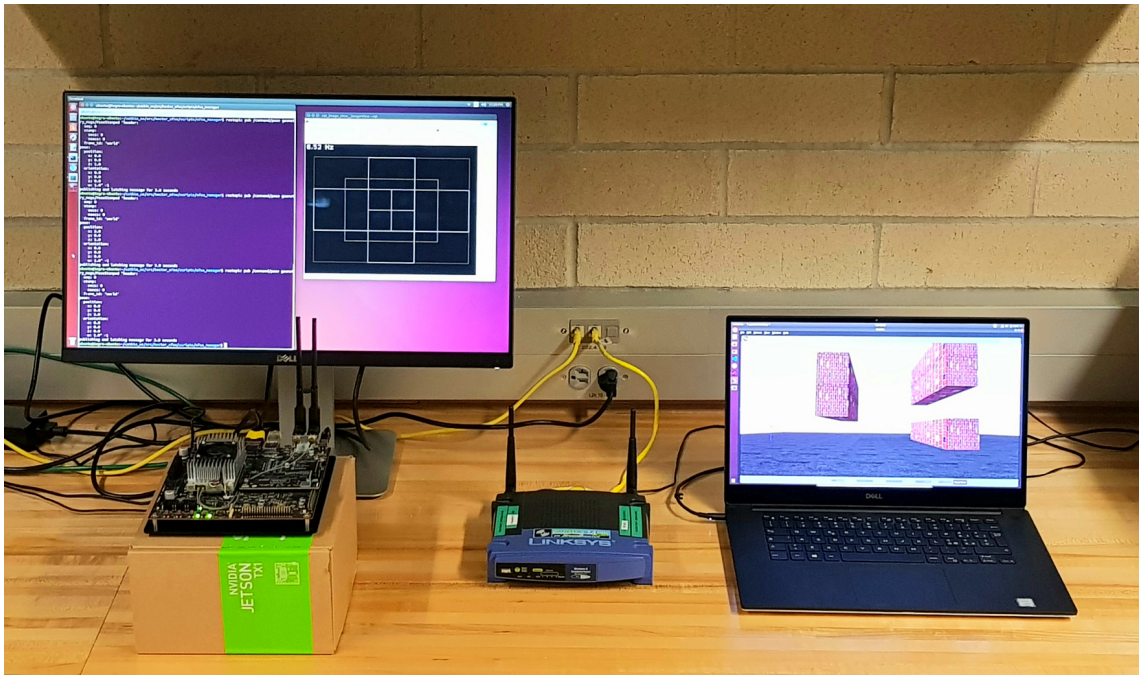
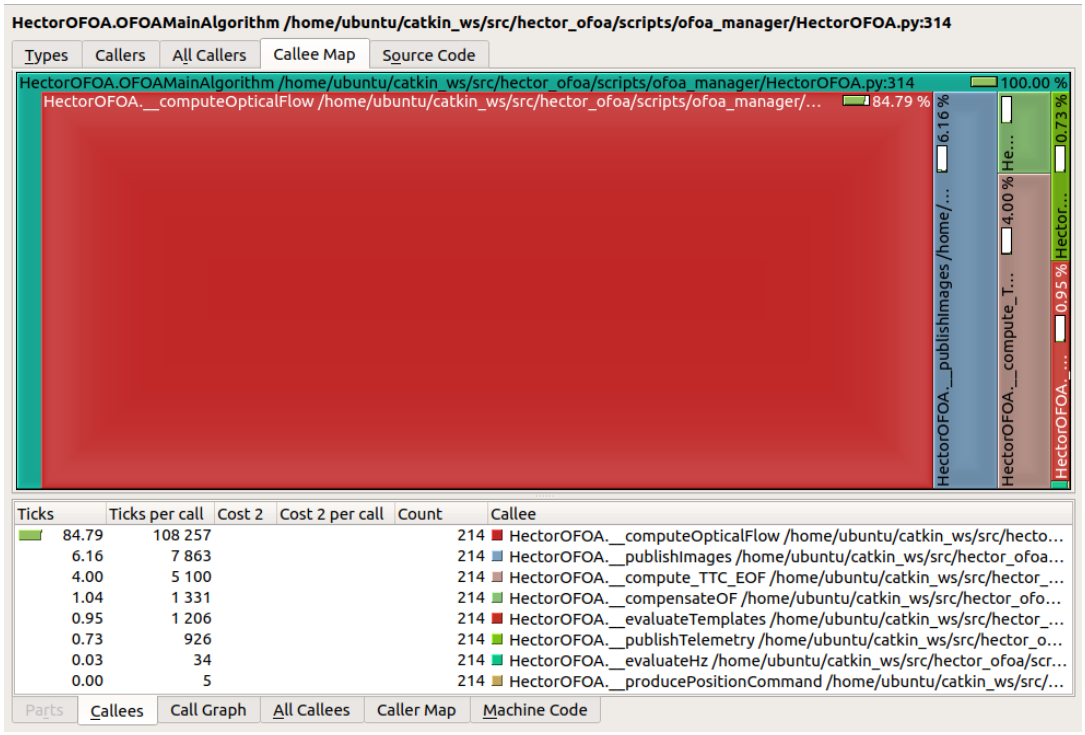
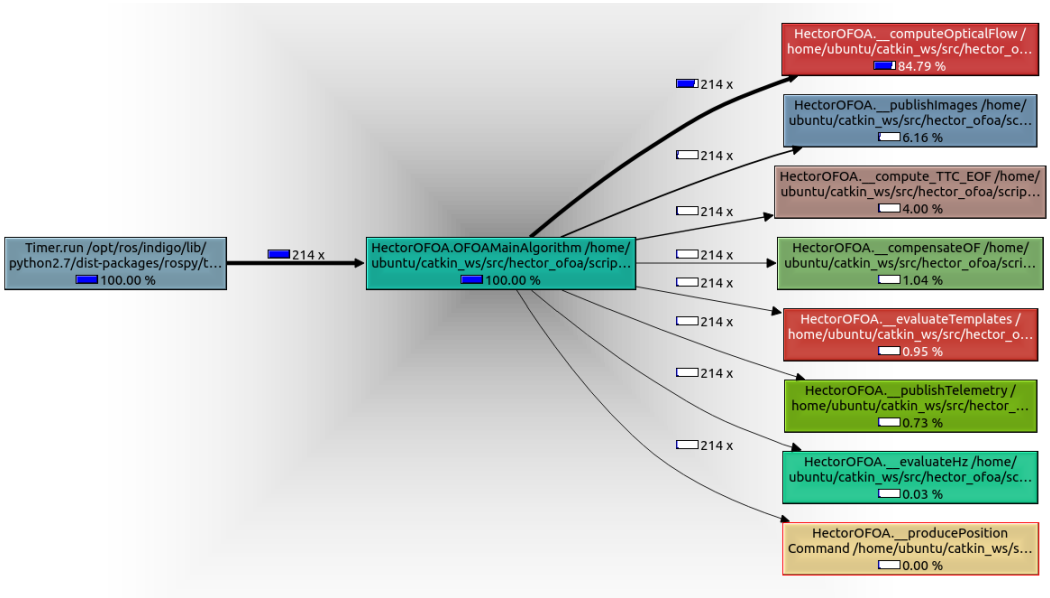


Figure 5.5: Setup for HIL simulation. On the left, the Jetson TX1 Development Kit is connected to a monitor via HDMI cable, running Ubuntu 14.04 and OFOA ROS node. On the right, the laptop with virtualized Ubuntu 18.04 which hosts ROS master node and Gazebo simulator. The Jetson TX1 and the PC are connected by means of the WiFi router acting as a gateway, in the middle.

Profiling visualization of main OFOA algorithm function in the node is shown in Figure 5.6 in both forms of tree map and call graph. Execution time is measured in actual wall time and expressed in terms of CPU ticks.
What can be noticed from Figure 5.6a is that the majority of time is spent in computing the OF field, actually 84.79% of the overall time spent in caller function *OFOAMainAlgorithm*, making it the real bottleneck of the whole algorithm. OF field computation is implemented by an OpenCV function that recalls an optimized compiled algorithm in C++, hence cannot be optimized anymore.
The second most expensive function is the one used to compute TTC and EOF for frontal obstacle avoidance evaluation, with a time cost of 6.16% of main function execution time. The computation of TTC and EOF has already been vectorized with NumPy array in order to exploit the highly optimized compiled modules for array operations, hence the function is already in its optimal form. Other functions



(a)



(b)

Figure 5.6: Treemap (5.6a) and call graph (5.6b) profiling representation by means of KCacheGrind of function *OFOAMainAlgorithm* run on NVIDIA Jetson TX1.

are reported in decreasing execution time percentage order, but with lower overall impact and containing non-optimizable or already optimized operations.

5.2.2 Algorithm execution time statistics and Hardware-in-the-loop (HIL) simulation

Optimizing code is not enough if the algorithm cannot produce results at a reasonable rate in order to be used on real onboard embedded systems and react in real time. Hence, a statistical analysis of main algorithm execution time is carried out here on different platforms.
In order to produce the measurements, a timer starts at the beginning of main algorithm function and it's stopped at the end of its execution, evaluating time elapsed and, by its inverse, the frequency of execution ideally achieved in that iteration. Here operating system time is considered, not simulation time. This information is then published by means of telemetry ROS messages of type *OFOAManagerTelemetry* created on purpose for debugging, where entries *time_delta* and *fps* hold execution time and its inverse respectively. A ROS bag file is recorded for around 60 s of execution (wall time) and finally statistics are computed by a Python script designed for this purpose that retrieves time evolution of these quantities and outputs minimum, maximum, average value and its standard deviation. In case the algorithm is implemented as a process to be scheduled in an operating system in an embedded system, in order to obtain a feasible scheduling the maximum execution time can be considered the worst case execution time (WCET), used in design phase to choose the task period. An histogram to show samples distribution is also produced and reported in following results.
The main algorithm function is associated to a ROS timer object which frequency is set to 5 Hz, hence with period 200 ms, that is a safe enough value in order to let all the operations be executed without excessively compromising obstacle avoidance reactivity.

Measurements are first made on the same platform where simulations are run, i.e. Ubuntu 18.04 run on Virtual Box, and then deployed on real embedded hardware that can be practically mounted onboard on a quadrotor.

Simulation with VM VirtualBox

Previously reported simulations are run on Ubuntu 18.04 LTS virtualized by means of Oracle VM VirtualBox. The PC on which VirtualBox runs is a Dell XPS 15, with an Intel Core i7-9750H 6-cores CPU which run at 2.6 GHz and 16 GB of RAM and an Intel UHD Graphics 630 GPU with 8 GB memory. The VM is set with 3 of the 6 available cores and 8 GB of RAM of the 16 GB available.
In Figure 5.7 the time evolution plot of *OFOAMainAlgorithm* execution time over a 1 minute (wall time) run is shown.

The reported time evolution, after an analysis carried out by a Python script, gives the statistical results reported in Table 5.4.
As can be noticed, the maximum execution time (or WCET) required from the main algorithm to be executed in the conducted experiment is lower than the 200 ms set period.

Hardware-in-the-loop simulation with NVIDIA Jetson TX1

In order to have a more practical idea of how the software would behave in a real scenario, the OFOA manager ROS node is deployed on the already mentioned NVIDIA Jetson TX1 board.
Execution time trend and its distribution in time in histogram form are reported in Figure 5.8. Statistical analysis produced by Python script gives results reported in Table 5.5. The results show that the algorithm can be deployed on an embedded hardware and be able to keep up with required rate with no problems.

Hardware-in-the-loop simulation with RaspberryPi 4

As a further test, the algorithm is tested with the same method on a RaspberryPi 4, a cheaper alternative to Jetson TX1 board, but powerful as well, shown in Figure 5.9. RaspberryPi 4 is a Linux-based computer on a board, mounting a Quad core Cortex-A72 (ARM v8) 64-bit CPU running at 1.5 GHz, 4 GB LPDDR4-3200 SDRAM memory, several connections such as 2.4 GHz and 5.0 GHz WiFi, USB 3.0 and a 40 pins GPIO that let the designer interface the board with actual hardware. The board weights 46 g and power consumption is around $3 - 6.25$ W.

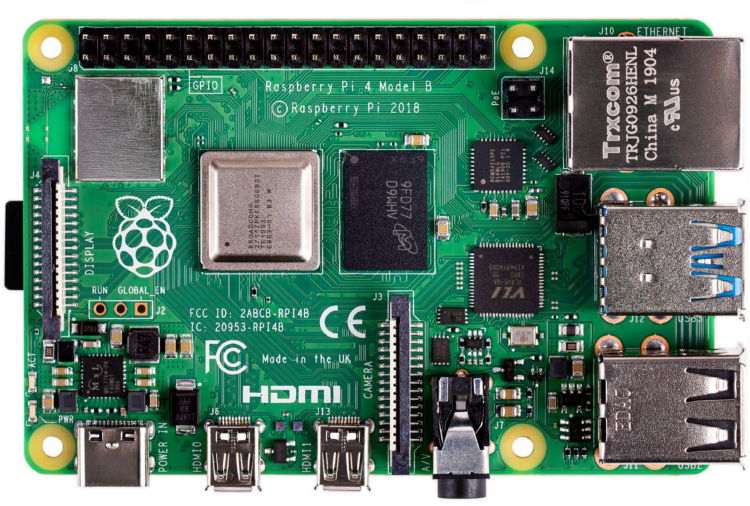
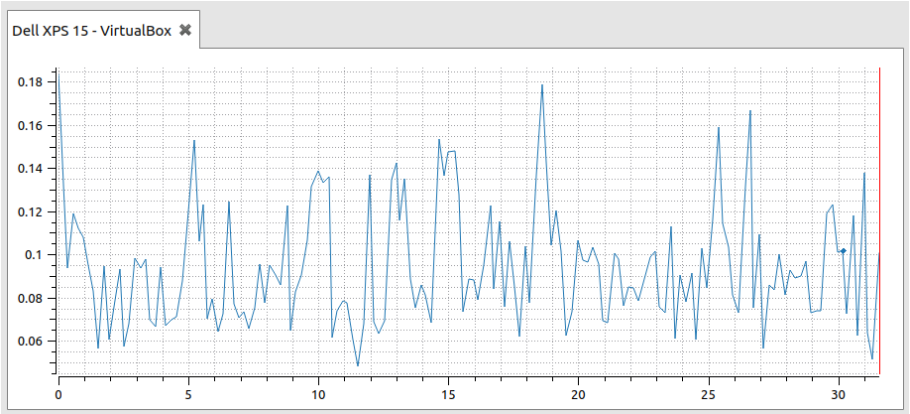
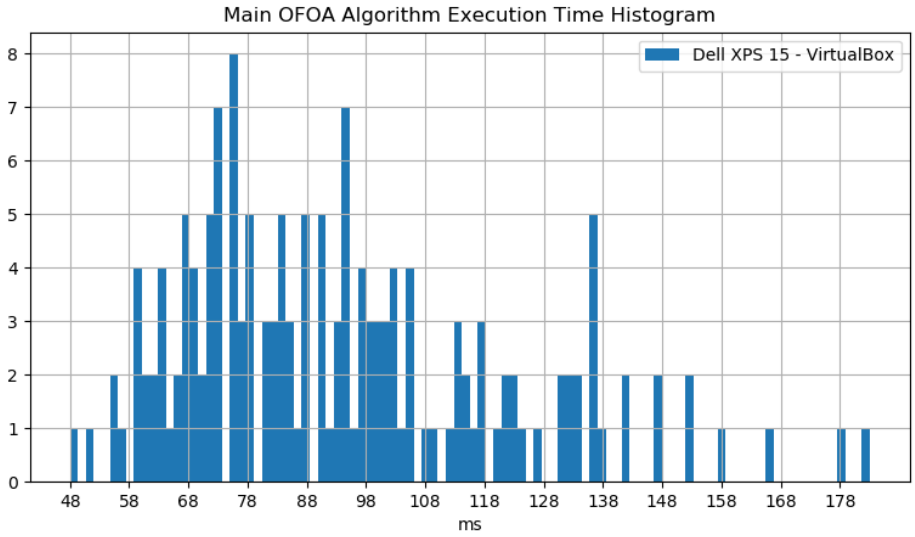


Figure 5.9: RaspberryPi 4 board, a powerful Linux-based embedded system.

The most popular OS used on RaspberryPi 4 is Raspbian OS, but the board is able to run any Linux-based OS (by always keeping an eye on performance requirements). Given that ROS is needed on the RaspberryPi to run the OFOA manager



(a)

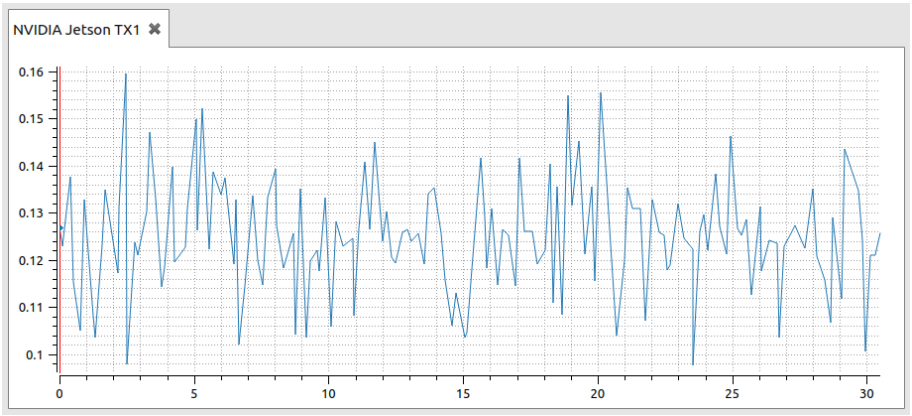


(b)

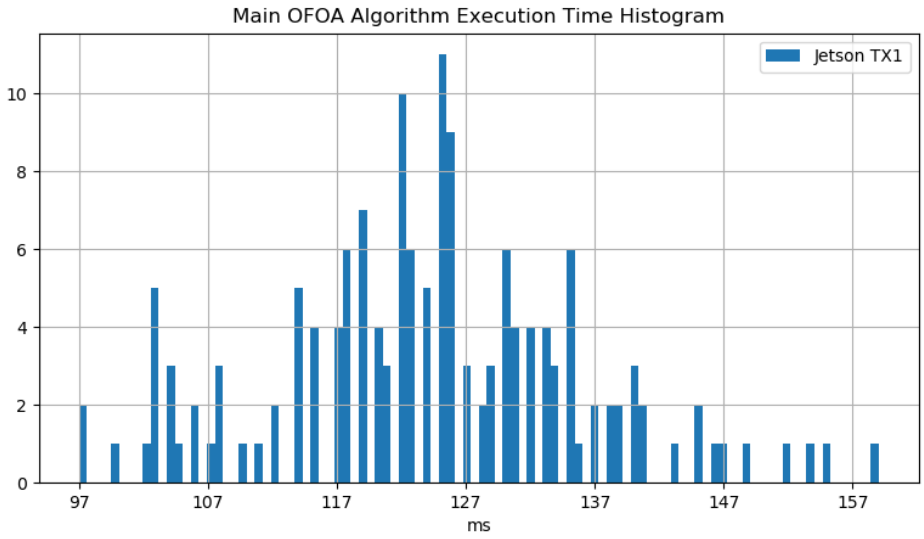
Figure 5.7: Time evolution of *OFOAMainAlgorithm* function execution time in each iteration run on VirtualBox, visualized by means of PlotJuggler (5.7a) and Matplotlib (5.7b) in histogram form.

Table 5.4: Statistics of main algorithm computation time performance run in VirtualBox environment.

	Time [ms]	Frequency [Hz]
Mininum	48.2	5.46
Maximum	183.3	20.73
Average	94.6	11.39
Standard deviation	27.0	3.03



(a)



(b)

Figure 5.8: Time evolution of *OFOAMainAlgorithm* function execution time in each iteration run on Jetson TX1 board, visualized by means of PlotJuggler (5.8a) and Matplotlib (5.8b) in histogram form.

Table 5.5: Statistics of main algorithm computation time performance run on NVIDIA Jetson TX1 embedded system.

	Time [ms]	Frequency [Hz]
Minimum	97.7	6.27
Maximum	159.4	10.24
Average	124.8	8.07
Standard deviation	11.9	0.78

node and ROS is not available for Raspbian OS, the Ubuntu distro *Ubuntu MATE 18.04 LTS* has been flashed on the micro SD card that is used as storage and main boot drive. Then ROS Melodic has been installed and OFOA manager node has been correctly setup to run on it. The experimental setup for HIL is shown in Figure 5.10.

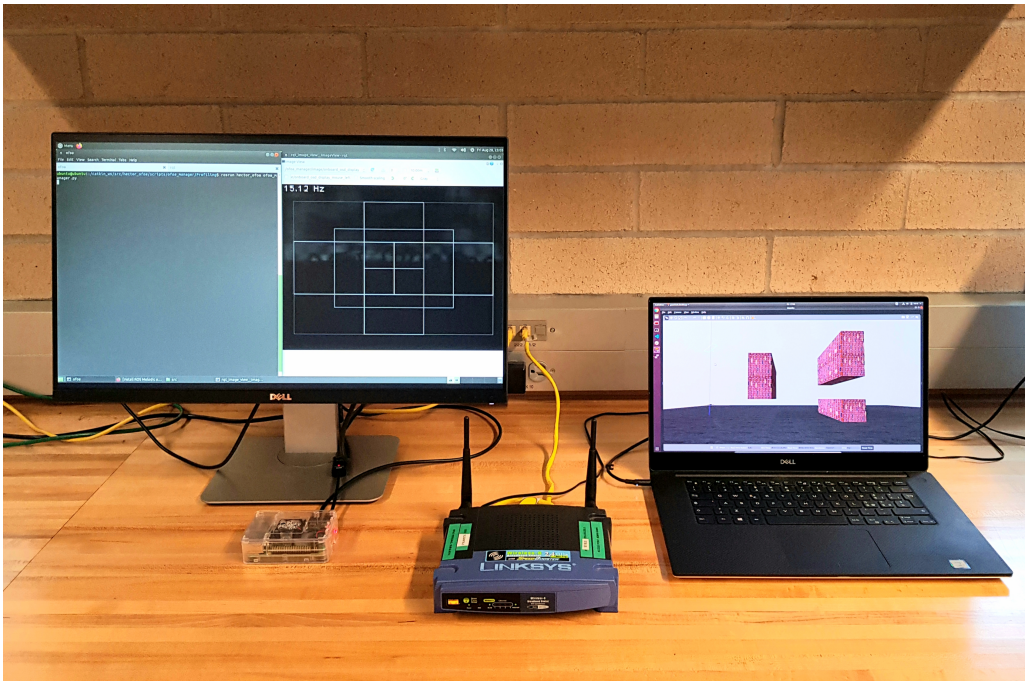
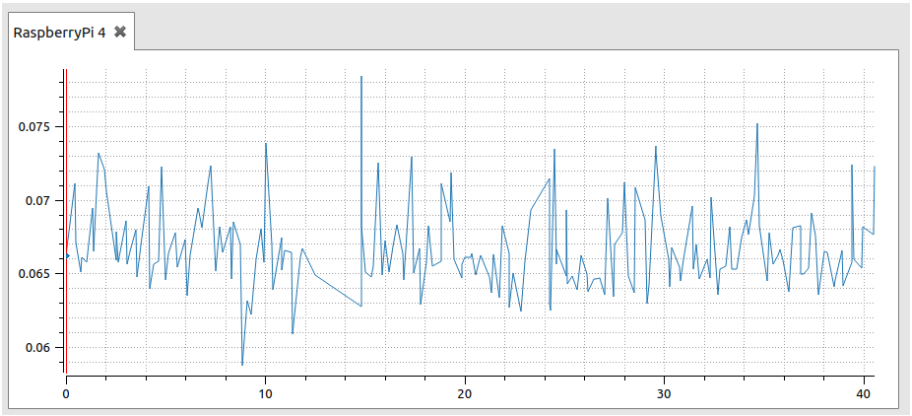
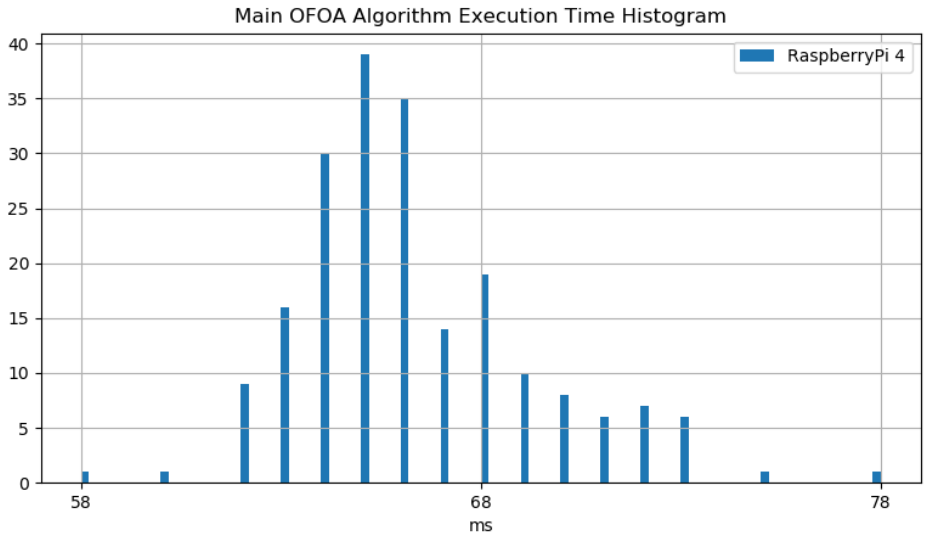


Figure 5.10: Setup for HIL simulation. On the left, the RaspberryPi 4 is connected to a monitor via HDMI cable, running Ubuntu MATE 18.04 and OFOA ROS node. On the right, the laptop with virtualized Ubuntu 18.04 which hosts ROS master node and Gazebo simulator. The RaspberryPi and the PC are connected by means of the WiFi router acting as a gateway, in the middle.

A 60 s long ROS bag recording is done and, as before, algorithm execution time is visualized by PlotJuggler and Python script in Figure 5.11, while its statistics can be seen in Table 5.6.
As results show, the RaspberryPi 4 board can perfectly manage the computations involved, with a minimum frequency of 12.75 Hz, more than double the minimum one performed by Jetson TX1 board, and average frequency is almost double. Furthermore, RaspberryPi 4 weights around 52% of Jetson TX1's weight and its maximum consumption is around 41.7% in comparison. Hence, RaspberryPi 4 not only can handle the real-time requirement better than Jetson TX1 module, but it is more power efficient and lighter in weight, making it suitable to be mounted onboard on a quadrotor.



(a)



(b)

Figure 5.11: Time evolution of *OFOAMainAlgorithm* function execution time in each iteration run on RaspberryPi 4 board, visualized by means of PlotJuggler (5.11a) and Matplotlib (5.11b) in histogram form.

Table 5.6: Statistics of main algorithm computation time performance run on RaspberryPi 4 embedded system.

	Time [ms]	Frequency [Hz]
Minimum	58.7	12.75
Maximum	78.4	17.02
Average	66.8	15.00
Standard deviation	2.9	0.63

Chapter 6

Conclusions and future work

6.1 Summary of results

In this research, first of all, a review of all the most recent scientific literature produced about optical flow based obstacle avoidance strategies for UAVs has been carried out at the best of my knowledge and, after spotting possible main limitations across different studies, a classification of limitations has been proposed. From identified limitations, the objective of obtaining a monocular optical-flow-based 3D obstacle avoidance strategy that is real-time implementable by onboard hardware is stated.
After that, the kinematic and dynamic nonlinear equations of a quadrotor aerial vehicle has been derived together with the inertia matrix of an ideal version of quadrotor. The cascaded PID implemented by used ROS package to control quadrotor's flight in position is then explained and correctly tuned. After delving in optical flow algorithms theory and their classification, the Gunnar Farnebäck Dense OF method is chosen among others for characteristics and performances reasons and is explained, as well as concepts such as focus of expansion (FOE), time-to-contact (TTC) and expansion of flow (EOF) and their mathematical computation. Then the chosen software for algorithm implementation, simulation and performances measurement is shown and motivated. Test scenarios in which avoidance performances of the algorithm are tested have been identified so that effectiveness in different kinds of situations can be proved: one with lateral obstacles for testing horizontal avoidance, one with a frontal wall to test frontal avoidance and one with a floating obstacle and a slit for testing vertical avoidance. Performances measurement metrics, both for obstacle avoidance and for software performance, are then formalized and reported. A solution to the stated problem is then proposed by designing a local path planning strategy, operating by commanding appropriate waypoints to the quadrotor, which working principle is similar to the one used by some insects and birds exploiting generated optical flow from frontally mounted monocular camera to understand where in the frame there is more optical movement by means both of optical flow unbalance principle and moving in the opposite direction to avoid the obstacle and expansion of the optical flow to identify a frontal obstacle and avoiding it, all by computing

an intermediate waypoint based on aforementioned information and prepending it to the already stored list of waypoints.
Finally, the solution is implemented as a ROS node written in Python, tested in the defined scenarios and performances are measured. Results in the three scenarios show a very high success rate, in particular 100% in horizontal and vertical avoidance and 95.2% for frontal avoidance by considering a minimum safe distance from obstacles, showing the effectiveness of the proposed method. The ROS node is run both on the same PC running Gazebo, hence performing a SIL test, and on two embedded hardware platforms such as NVIDIA Jetson TX1 and RaspberryPi4 exploiting the distributed nature of ROS, doing then an HIL test. Software performances are measured both by means of a profiler in order to test code optimization and also by measuring statistical behaviour of algorithm execution time on each platform. Results show that the software is already optimized, with the optical flow field computation as the main bottleneck for performances and execution time wise algorithm can run in real time on both the tested embedded systems, where Jetson TX1 lets it run with a minimum frequency of at least 6.27 Hz and an average one of 8.07 Hz, while the RaspberryPi guarantees a minimum frequency of 12.75 Hz and an average one of 15.00 Hz, but with a weight being 52% and estimated power consumption being 41.7% of the Jetson's ones, making it more suitable for onboard integration.

6.2 Future research

The proposed algorithm, even if proved effective, has some limitations such as dependence on visibility: in dark environments is difficult for camera to obtain sufficiently well exposed images. Environments with poorly textured objects can be a problem too because the optical flow computation algorithm struggles in finding relationships between pixels among two consecutive frames. A possible solution is integrating other types of sensors such as IR cameras or laser scanners, increasing however the cost of the application and not by a small amount, even if laser scanner sensors are becoming cheaper.
A way to improve robustness, given that the frontal position of the camera prevents the quadrotor from being aware of obstacles on the sides such as wall corners due to limited FOV angle, could be to add lateral sonar sensors to spot them and react accordingly; same reasoning holds for obstacles above and below quadrotor.
Another limitation comes from the fact that being the sampled information (the image) coming just from a monocular camera, the information about depth is lost; hence, to make the algorithm more robust, a stereoscopic camera can be used instead at the cost of an heavier computational burden beside economical cost in order to retrieve depth information and applying the optical flow unbalance to this information, being also able to better estimate time-to-contact.
A further development of the method is the integration of this local path planning strategy into a team global path planning strategy, so that more complex tasks can be accomplished and robustness of overall team global path planning can be

improved.

Bibliography

[1] "Amazon wins FAA approval for Prime Air drone delivery fleet". In: *CNBC* (2020). URL: https://www.cnbc.com/2020/08/31/amazon-prime-now-drone-delivery-fleet-gets-faa-approval.html.

[2] *U.S. military UAS groups*. URL: https://en.wikipedia.org/wiki/U.S._military_UAS_groups.

[3] *2019-2020 World Military Unmanned Aerial Systems Market Profile & Forecast by Teal Group*. URL: https://www.tealgroup.com/index.php/pages/press-releases/64-teal-group-predicts-worldwide-military-uav-production-of-almost-99-billion-over-the-next-decade-in-its-2019-2020-uav-market-profile-and-forecast.

[4] Mandyam Srinivasan, Saul Thurrowgood, and Dean Soccol. "Competent vision and navigation systems". In: *Robotics & Automation Magazine, IEEE* 16 (Oct. 2009), pp. 59–71. DOI: 10.1109/MRA.2009.933627.

[5] James Jerome Gibson. "The Perception of the Visual World". In: Jan. 1950. ISBN: 978-1114828087.

[6] Berthold K.P. Horn and Brian G. Schunck. "Determining optical flow". In: *Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.* Vol. 17. 1981, pp. 185–204.

[7] Blin Richards et al. "Obstacle Avoidance System for UAVs using Computer Vision". In: *AIAA Infotech @ Aerospace*. 2015. DOI: 10.2514/6.2015-0986. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2015-0986. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2015-0986.

[8] Jan Blumenkamp. "End to end collision avoidance based on optical flow and neural networks". In: *ArXiv* abs/1911.08582 (2019).

[9] Ajay Shankar, Mayank Vatsa, and P.B. Sujit. "A Low-Cost Monocular Vision-based Obstacle Avoidance using SVM and Optical Flow". In: *Unmanned Systems* 6 (Aug. 2018). DOI: 10.1142/S2301385018500097.

[10] Huiqi Miao and Yan Wang. "Optical Flow Based Obstacle Avoidance and Path Planning for Quadrotor Flight". In: June 2018, pp. 631–638. ISBN: 978-981-10-6444-9. DOI: 10.1007/978-981-10-6445-6_69.

[11] P. Gao et al. "Obstacle avoidance for micro quadrotor based on optical flow". In: *2017 29th Chinese Control And Decision Conference (CCDC)*. 2017, pp. 4033–4037.

[12] J. -. Zufferey and D. Floreano. "Toward 30-gram Autonomous Indoor Aircraft: Vision-based Obstacle Avoidance and Altitude Control". In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 2594–2599.

[13] Pooja Agrawal, Ashwini Ratnoo, and Debasish Ghose. "A Composite Guidance Strategy for Optical Flow based UAV Navigation". In: *IFAC Proceedings Volumes* 47.1 (2014). 3rd International Conference on Advances in Control and Optimization of Dynamical Systems (2014), pp. 1099–1103. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20140313-3-IN-3024.00151. URL: http://www.sciencedirect.com/science/article/pii/S1474667016327914.

[14] Aydin Eresen, Nevrez imamoğlu, and Mehmet Efe. "Autonomous quadrotor flight with vision-based obstacle avoidance in virtual environment". In: *Expert Systems with Applications* 39 (Jan. 2012), pp. 894–905. DOI: 10.1016/j.eswa.2011.07.087.

[15] Gangik Cho, Jongyun Kim, and Hyondong Oh. "Vision-Based Obstacle Avoidance Strategies for MAVs Using Optical Flows in 3-D Textured Environments". In: *Sensors(Basel)* (June 2019). DOI: 10.3390/s19112523.

[16] Dong-Wan Yoo, Dae-Yeon Won, and Min-Jea Tahk. "Optical Flow Based Collision Avoidance of Multi-Rotor UAVs in Urban Environments". In: 2011.

[17] Wilbert Aguilar et al. "Monocular Vision-Based Dynamic Moving Obstacles Detection and Avoidance". In: Aug. 2019, pp. 386–398. ISBN: 978-3-030-27540-2. DOI: 10.1007/978-3-030-27541-9_32.

[18] C. Wang, W. Liu, and M. Q. -. Meng. "Obstacle avoidance for quadrotor using improved method based on optical flow". In: *2015 IEEE International Conference on Information and Automation*. 2015, pp. 1674–1679.

[19] Randall W. Beard. "Quadrotor Dynamics and Control". In: (2008).

[20] Alexander Sendobry. "Control System Theoretic Approach to Model Based Navigation". In: (2014).

[21] Johannes Meyer et al. "Comprehensive Simulation of Quadrotor UAVs using ROS and Gazebo". In: *3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*. 2012, to appear.

[22] Sachin Chitta et al. "ros_control: A generic and simple control framework for ROS". In: *The Journal of Open Source Software* (2017). DOI: 10.21105/joss.00456. URL: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf.

[23] Yaochen Li et al. "Road Scene Simulation Based on Vehicle Sensors: An Intelligent Framework Using Random Walk Detection and Scene Stage Reconstruction". In: *Sensors* 18 (Nov. 2018), p. 3782. DOI: 10.3390/s18113782.

[24] Arcangelo Distante and Cosimo Distante. "Motion Analysis". In: *Handbook of Image Processing and Computer Vision: Volume 3: From Pattern to Object*. Cham: Springer International Publishing, 2020, pp. 479–598. ISBN: 978-3-030-42378-0. DOI: 10.1007/978-3-030-42378-0_6. URL: https://doi.org/10.1007/978-3-030-42378-0_6.

[25] M. Mammarella et al. "Comparing Optical Flow Algorithms Using 6-DOF Motion of Real-World Rigid Objects". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1752–1762.

[26] John Barron, David Fleet, and Steven Beauchemin. "Performance Of Optical Flow Techniques". In: *International Journal of Computer Vision* 12 (Feb. 1994), pp. 43–77. DOI: 10.1007/BF01420984.

[27] Bruce Lucas and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)". In: vol. 81. Apr. 1981.

[28] Marc Proesmans et al. "Determination of optical flow and its discontinuous using non-linear diffusion". In: vol. 2. Apr. 2006, pp. 294–304. DOI: 10.1007/BFb0028362.

[29] David Fleet and Allan Jepson. "Computation of component image velocity from local phase information". In: *International Journal of Computer Vision* 5 (Aug. 1990), pp. 77–104. DOI: 10.1007/BF00056772.

[30] Christopher G. Harris and Mike Stephens. "A Combined Corner and Edge Detector". In: *Alvey Vision Conference*. 1988.

[31] D. G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2.

[32] Jianbo Shi and Tomasi. "Good features to track". In: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 1994, pp. 593–600.

[33] Gunnar Farnebäck. "Two-Frame Motion Estimation Based on Polynomial Expansion". In: vol. 2749. June 2003, pp. 363–370. DOI: 10.1007/3-540-45103-X_50.

[34] *Object for estimating optical flow using Farneback method*. URL: https://www.mathworks.com/help/vision/ref/opticalflowfarneback.html;jsessionid=4b3554d7cb85befae29e886626a9.

[35] Jasper de Boer and Mathieu Kalksma. "Choosing between optical flow algorithms for UAV position change measurement". In: 2015.

[36] *ROS Industrial*. URL: https://rosindustrial.org/.

[37] *Robotic Operating System (ROS)*. URL: https://www.ros.org/.

[38] *Wikipedia page for ROS*. URL: https://en.wikipedia.org/wiki/Robot_Operating_System.

[39] *Gazebo simulator tool*. URL: http://gazebosim.org/.

[40] *Hector quadrotor ROS wiki page*. URL: http://wiki.ros.org/hector_quadrotor.

[41] *Hector quadrotor GitHub repository*. URL: https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor.

[42] Johannes Meyer et al. "Comprehensive Simulation of Quadrotor UAVs using ROS and Gazebo". In: *3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*. 2012.

[43] *Hector quadrotor indoor mapping demo session YouTube video*. URL: https://www.youtube.com/watch?v=IJbJbcZVY28%5C&ab_channel=TeamHectorDarmstadt.

[44] *PlotJuggler: Timeseries visualization tool for ROS*. URL: https://www.plotjuggler.io/.

[45] *OpenCV: an open source computer vision library*. URL: https://opencv.org/.

[46] *OpenCV-Python bindings: wrapper for C++ compiled functions*. URL: https://docs.opencv.org/3.4.9/da/d49/tutorial_py_bindings_basics.html.

[47] *NumPy: The fundamental package for scientific computing with Python*. URL: https://numpy.org/.

[48] *CvBridge ROS package*. URL: http://wiki.ros.org/cv_bridge.

[49] *Yappi Python Profiler*. URL: https://pypi.org/project/yappi/.