POLITECNICO DI TORINO

Master of Science in Mechatronic Engineering

Master's Degree Thesis

Sensor fusion techniques for service robotic positioning and flight in GNSS denied environments using UWB technology

Politecnico di Torino

Supervisors
Prof. Marcello Chiaberge
Dott. Ing. Giovanni Fantin

Candidate
Dott. Ing. Cosimo Conte

December 2021

Abstract

Drones are usually designed to navigate outdoor spaces, they are often equipped with cameras and other sensors, making them a powerful tool for surveying large areas. In these environments a Global Navigation Satellite System (GNSS) is combined with an Inertial Measurement Unit (IMU) to achieve precise positioning, allowing successful navigation in a 3D open space. During the past years users started to use small size drones in challenging environments, indoor places, inside caves or near bridges, where a GNSS is not always reliable or reachable.

Classical positioning techniques are no longer efficient in these cases, so it is necessary to develop new systems to adapt in these situations. Ultra WideBand (UWB) sensors are used to enhance the positioning in closed environments. These sensors allow two tags to exchange signals at high frequency in order to retrieve the distance between each other. It is possible to fuse this information with classical positioning methods to resolve the positioning problem in any scenario.

The goal of this thesis is to design a system that allows drones to flight in both GNSS enabled and denied environments using UWB tags. This work explores the main positioning techniques used in literature and introduces several sensor fusion algorithms, aiming at comparing them in terms of both accuracy and precision. The final intent is to achieve reliable flight in any situation without disruption of service: referred as seamless flight.

The capabilities of the proposed method are measured in a simulated environment and then validated on a real quadcopter. In order to easily implement the algorithms, all proposed codes are integrated in the PX4-Autopilot Robotics API. This allows to test the same instances of the system both on the companion computer, a Raspberry Pi 4 on the drone, and on the Gazebo simulation environment on a desktop computer, without changing a single line of code.

The results of this work show the outstanding capabilities of Kalman filters to fuse sensor's information, with a particular focus on their nonlinear variant. Thanks to these methods is possible to obtain a reliable pose estimate using the raw UWB ranging data and augmenting it with predicted velocity estimate, that is vital in achieving stable control.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work aims to search different sensor fusion techniques for robotic positioning and flight in Global Navigation Satellite System (GNSS) denied environments using Ultra WideBand (UWB) technology. This is an experimental work, where the test bench to compare the performance of the algorithms includes the utilization of quadcopters, both simulated and real.

Quadcopters and the majority flying vehicles are completely dependent from GNSS services, and they are unable to locate themselves and flight in a closed environment, where the GNSS signals are attenuated or missing, but the recent trend to use robots in the industrial world requires methods to achieve stable and precise flight in challenging scenarios, despite the availability of GNSS coverage.

The solution to achieve indoor positioning foresees the usage of UWB ranging sensors, capable to measure the distance between themselves, using their unique transmission properties. This technology allows the drone to receive the ranging measurements from UWB anchors placed in the environment to the UWB tag mounted on the quadcopter's chassis. These anchors are used like GNSS satellites to achieve passive positioning.

The goal of this thesis is to design a plug-in estimator, that coupled with a pre-existent flight stack software, is capable to instruct it with reliable positioning feedback. The estimator is capable to use UWB ranging measurement to estimate the drone's state when a GNSS is not available. The system proposed integrate both sources of data, UWB and GNSS to achieve centimeters like precision. Moreover, it allows to flight from a UWB enabled environment to a GNSS one, and vice versa, without any disruption of service.

This work aims to produce a complete algorithm to be used in a real scenario, the final system is able to achieve a reliable navigation solution in any situation, even if the anchors are not georeferenced and placed by hand with dozens of centimeter of placing error. The anchors must also be lightweight and consume as less power as possible, a precise and expensive solution cannot be realistically integrated, so the trade-off is working with ranging measurement that are noisy and contains outliers.

In Chapter 2 it is described a particular type of Unmanned Aerial Vehicle (UAV), quadcopters, on which all the algorithms are presented, simulated and tested. Quadcopters have high adaptability and low encumbrance, making them optimized for indoor flight and able to support a wide spectrum of sensors. The focus of this chapter is to define the dynamics of the quadcopter and to describe the architecture of the flight stack in use, in order to define a way to integrate the estimator accounting for the dynamics and software properties of the drone. In any case the methods are abstracted to work on a wide spectrum of robots and on different software interfaces.

Chapter 3 provides a theoretical background on the sensors typically used for positioning, then it explores the classical positioning techniques that are known in the literature, using mainly GNSS and Inertial Measurement Unit (IMU) information. Then it also introduces the UWB technology and describe the methods available to calculate the position of a tag from the ranging measurement.

The core of the thesis is presented in Chapter 4, where different methods to extrapolate the drone's state are proposed. UWB data is filtered and fused to the classical sensors in order to achieve the expected results. The theory is matched with the filter implementation, using the Python library *FilterPy* and Robot Operating System (ROS2) development environment. Then the estimate is feed to the drone flight stack in order to evaluate each algorithm performance.

The first mean of evaluating the presented methods is by Software In The Loop (SITL) simulations, thanks to PX4 Autopilot middleware it is possible to connect the flight stack to ROS2 nodes, that implement the filters, in a simulated environment provided by Gazebo. A plugin for Gazebo it has been developed during this work in order to correctly simulate the UWB sensor and its noise model. Both the ROS2 nodes and plugin code are listed in the Appendix A. All the scenario simulated and the measured performances of the filters are indicated with meaningful plots in Chapter 5.

The final part of this work consists of setting up an experimental analysis to prove the correct functionality of the proposed solution, see Chapter 6. Using a real quadcopter allows connecting PX4 Autopilot to a Raspberry Pi 4 that runs the filter ROS2 node. Then a manual command to disable GNSS reception is sent during the tests to reproduce a GNSS denied environment. Different transition between GNSS and UWB enabled environments have been tested in a controlled space, so the methods are reliable enough to be implemented in a complete estimator stack.

Chapter 2

Unmanned Aerial Vehicles

A Unmanned Aerial Vehicle (UAV), commonly known as drone, is an aircraft without a pilot, controlled from the ground or by a computer on board [16]. UAVs are designed for missions where the human presence could be dangerous, unnecessary or expensive. They need to have the circuitry on board to power themselves and to be controlled remotely or to navigate autonomously with a preconfigured mission.

This work focus on a particular type of Vertical Take-Off and Landing (VTOL) vehicles: quadcopters, that provide a suitable test bench for experimenting with Global Navigation Satellite System (GNSS) denied environments. Quadcopters are capable to withstand indoor flight, the principal space that does not provide GNSS positioning, thanks to their high speed dynamics, low inertia and dimensions.

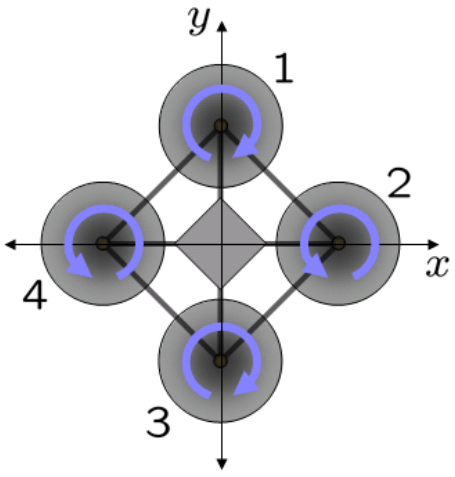

Figure 2.1. Schematic of reaction torque on each motor of a quadcopter [6].

2.1 Quadcopters

Quadcopters are UAVs with four rotors, two spinning clockwise and two counter-clockwise, the rotors are directed upwards and placed in a rectangular formation with equal distance from the center as seen in Figure 2.1.

Each rotor produce thrust and torque about is center of rotation. By changing the rotational speed of the rotors it is possible to achieve precise flight control, moreover their small size and low inertia make this type of vehicle simpler to control and to use in challenging scenarios, like surveillance, search and rescue, construction inspection and others [12].

2.1.1 Dynamics

The dynamical model of the quadcopter is the starting point to understand its properties and to construct a suitable model to use in the estimation of the drone's state.

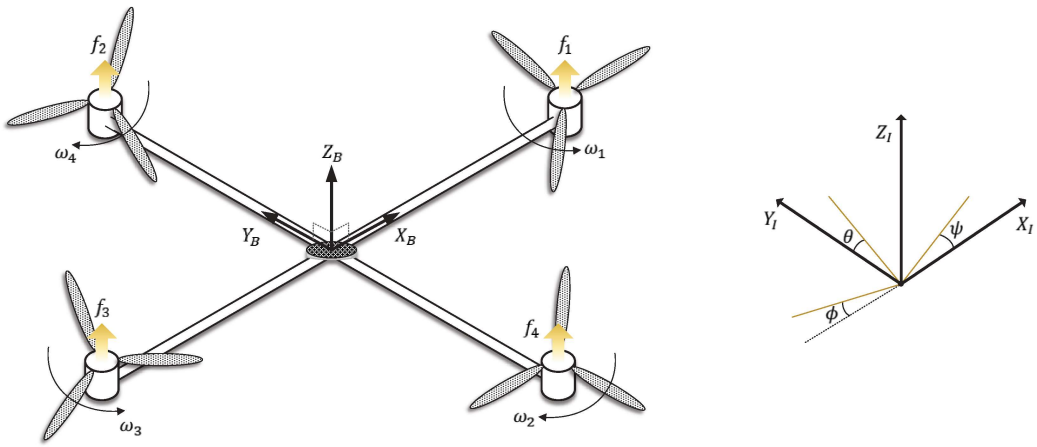

Figure 2.2. Quadcopter rigid body scheme. [1]

The scheme in Figure 2.2 represents the rigid body scheme of a standard quadcopter with the principal quantities. All the reference frames used in this chapter are coherent with general literature [5].

The state of the quadcopter is described by its absolute position in the inertial navigation frame $\boldsymbol{p^n}$; the attitude is defined by the Euler angles $\boldsymbol{\eta}$ representing the orientation of the body frame with the respect of the navigation one:

$$\boldsymbol{p^n} = \begin{bmatrix} x^n \\ y^n \\ z^n \end{bmatrix}, \qquad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \tag{2.1}$$

The origin of the body frame is in the center of mass of the quadcopter, its linear velocities $\boldsymbol{v^b}$ and angular rates $\boldsymbol{\nu}$ in the body frame are vital to control the drone's flight:

$$\boldsymbol{v^b} = \begin{bmatrix} v_x^b \\ v_y^b \\ v_z^b \end{bmatrix}, \qquad \boldsymbol{\nu} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}. \tag{2.2}$$

The quadcopter is assumed to have symmetric structure with the arms aligned with the x and y axis, so the inertia matrix $\boldsymbol{I}$ is diagonal with $I_{xx} = I_{yy}$:

$$\boldsymbol{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}. \tag{2.3}$$

2.1.2 Rotors dynamics

Each rotor i produces a force f_i in the direction of the motor axis and a torque τ_{Mi} around it. Both quantities depend on the angular velocity of the motor ω_i:

$$f_i = k\omega_i^2, \qquad \tau_{Mi} = b\omega_i^2 + I_m\dot{\omega}_i; \tag{2.4}$$

with k the lift constant, b the drag constant and I_m the motor inertia. These 3 parameters depends on the rotor, usually the effect of $\dot{\omega}_i$ are omitted because I_m is very small.

The combined action of the 4 rotors produces a thrust T_z along the body z-axis and a torque $\boldsymbol{\tau^b}$ in the direction of the corresponding body angles:

$$T_z = \sum_{i=1}^{4} f_i = k\sum_{i=1}^{4} \omega_i^2, \tag{2.5}$$

$$\boldsymbol{T^b} = \begin{bmatrix} 0 \\ 0 \\ T_z \end{bmatrix}, \tag{2.6}$$

$$\boldsymbol{\tau^b} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk(-\omega_2^2 + \omega_4^2) \\ lk(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^{4} \tau_{Mi} \end{bmatrix}; \tag{2.7}$$

where l is the distance between each rotor and the center of mass.

It is possible to control the quadcopter rotors and achieving attitude and thrust control by controlling the four angular velocities ω_i. This operation is usually done by a mixer that converts the thrust and orientation control commands in signals sent to the rotors Electronic Speed Control (ESC).

2.1.3 Newton-Euler equations

The rigid body dynamics are described by Newton-Euler equations in the inertial navigation frame:

$$m\ddot{\boldsymbol{p}}^{\boldsymbol{n}} = \boldsymbol{g} + \boldsymbol{R}\boldsymbol{T}^{\boldsymbol{b}}, \tag{2.8}$$

where $\boldsymbol{g}$ is the local gravity vector simplified as $[0, 0, -g]^T$ and $\boldsymbol{R}$ is the rotation matrix from the body frame to the navigation frame. Instead, in the body frame, the system is described by:

$$m\boldsymbol{v}^b + \nu \times (m\boldsymbol{v}^b) = \boldsymbol{R}^T\boldsymbol{g} + \boldsymbol{T}^b \tag{2.9}$$

The drone state mainly depends on the gravity effects and on the torque applied by the rotors, not accounting for aerodynamic drag that only become significant at high velocities. These effects are more difficult to measure with the respect of the body frame acceleration and rotational rates, in fact the latter are measured directly with a Inertial Measurement Unit (IMU).

From IMU data it is possible to have a direct feedback on the drone's state, then integration is sufficient to extract the complete set of state variables, this is explained in Section 3.2. Even if with IMU it is possible to predict the positioning, a direct feedback is always needed to compensate for the accumulating error in the integrator.

2.2 PX4-Autopilot

PX4 is the Professional Autopilot. Developed by world-class developers from industry and academia, and supported by an active worldwide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles. [4]

PX4-Autopilot provides an open source flight controller software stack that connects with sensors and the Radio Controller (RC) in order to control the motors. Moreover, it provides a set of middlewares to implement complex features.

2.2.1 Flight stack architecture

The flight stack is the core of the system: it consists of a collection of guidance, navigation and control algorithms useful to drive different types of drones, as well

as a set of estimation filters for attitude and position. In Figure 2.3 it is shown the connection between the building boxes of the stack.

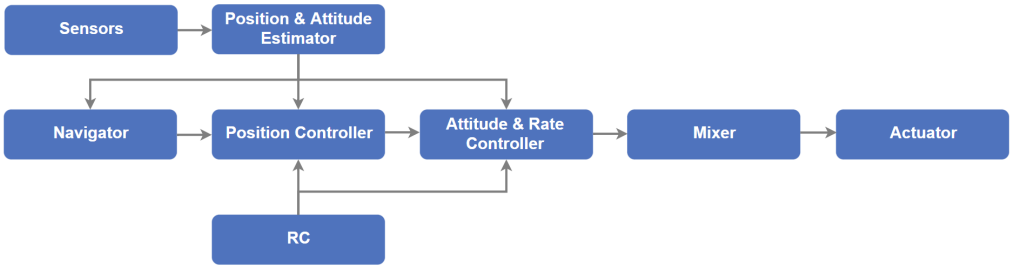

Figure 2.3. PX4 high level flight stack [4].

The estimator is the part of the stack that this work proposes to enhance in order to work in a GNSS denied environment, adding the support of Ultra WideBand (UWB) ranging sensors. Chapter 4 explain how it works.

The controllers take the estimation provided and calculate the control input to reach the desired position and attitude set point, it is essential that the estimate is accurate and stable to obtain precise flight.

The mixer takes high level commands and translate them to individual motor commands, ensuring that limits are not exceeded. It heavily depends on the UAV dynamic characteristics.

2.2.2 Control stack

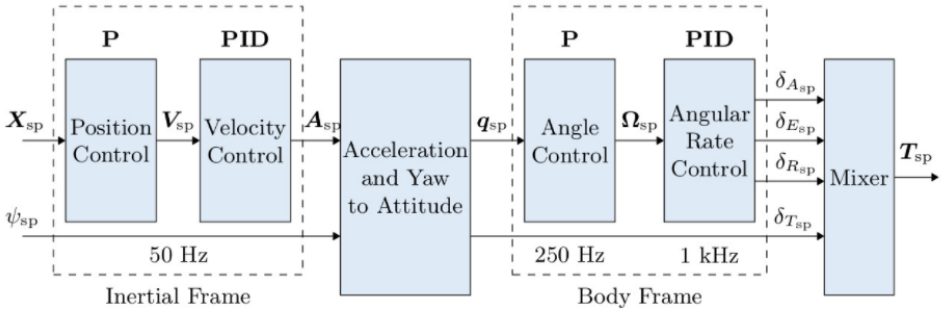

Figure 2.4. PX4 control stack. [4]

The control stack is the part of the system that allows the drone to flight, as stated before it depends heavily on the correct estimation of the drone' state. In Figure 2.4 it is possible to acknowledge the capabilities of the controller, that is

able to allow multiple sources of control for the user. The positioning control is the most complex and require a complete knowledge of the feedback for all the block in the control stack: position, velocity, acceleration, attitude and angular rate.

This work focus on giving the correct feedback to the control stack in any situation, even in a GNSS denied scenario, then it let the stack deal with the actuation of the rotors to reach correct and stable positioning.

2.2.3 Middleware

PX4 provides a set of middlewares to communicate with embedded sensors and with the external world. One of them is the interface with the companion computer, explained in Chapter 6, the other is the simulation layer that allows PX4 flight code to run on a desktop computer, see Chapter 5.

The middleware functions are supported by the internal messaging API used for inter-process navigation: *uORB*. The modules of the autopilot subscribe and publish on different topics in order to exchange messages.

Thanks to the *microRTPS Bridge* there is an external interface to directly connect to *uORB* topics. The bridge can run in a simulated environment, allowing to run the entire stack in a computer without changing a single line of code. Moreover, it is possible to use a companion computer running Robot Operating System (ROS2) and exchange messages between ROS2 topics and PX4 ones using the same bridge, allowing a more flexible prototyping environment.

Chapter 3

Positioning techniques

This chapter explains the sensors and techniques used in the positioning estimation problem. The focus is the classical methods involving Global Navigation Satellite System (GNSS) and Inertial Measurement Unit (IMU) along with the ones that involve Ultra WideBand (UWB) sensors' ranging. In chapter 4 is presented the possibility to fuse these sources of information in order to achieve a better estimate.

3.1 Global Navigation Satellite System

Global Navigation Satellite System (GNSS) is a general term describing any satellite constellation that provides positioning, navigation and timing services on a global or regional basis [14]. Different systems are actually fully functional:

- Global Positioning System (GPS),
- Global'naya Navigatsionnaya Sputnikovaya Sistema (GLONASS),
- BeiDou,
- Galileo.

Others have regional scope or augment the capabilities of the previous ones.

3.1.1 GNSS architecture

Each GNSS is composed by a constellation of satellites orbiting the Earth. At least 24 satellites are required to achieve global coverage. Each satellite require all the instrumentation to power itself with solar panels and to broadcast ranging codes and navigation data messages on several frequencies. They are also equipped with an atomic clock to preserve stable time reference.

A GNSS is also composed of ground stations to monitor and control the satellites orbit and provide necessary corrections and maneuvering. Monitoring stations are

also used to calibrate the satellites clock, instead control stations are able to plan major satellite relocation in the event of failure.

At least the GNSS user equipment receive the signals from the constellation with a receiver, calculates the ranging data from the antenna to each satellite, then the navigation processor compute a position, velocity and time solution. Different user equipments have more functionalities with increasing cost: power supply, user interface, the capability to fuse ranging information with other sensors and multiple constellations support.

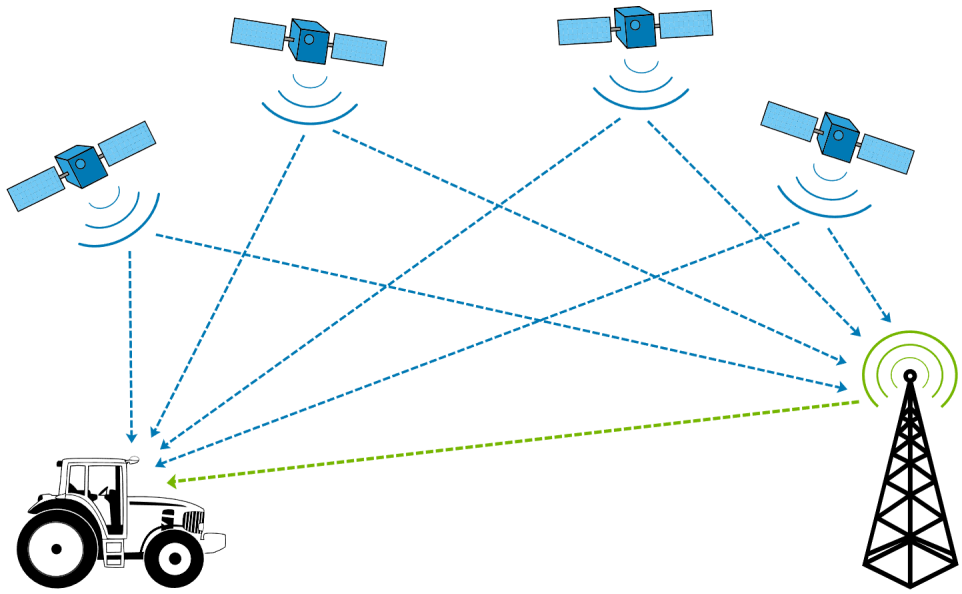

Figure 3.1. GNSS connection scheme.

Figure 3.1 represent the connection between the space segment composed by the satellites, the ground segment that controls them and a GNSS equipment mounted on a tractor.

3.1.2 Ranging measurement

All GNSS signals are in the L-band of the frequency spectrum. Because L-band waves penetrate clouds, fog, rain, storms, and vegetation, GNSS units can receive accurate data in all weather conditions, day or night. There are circumstances in which GNSS units may not receive signals accurately, such as inside concrete buildings or under heavy forest canopies [15].

A GNSS signal is a combination of a carrier with ranging codes and sometime navigation data messages. This work does not explain the complete signal transmission but focus on the transformation from the ranging data to the final estimate. The formulas and methods presented in this section are extrapolated from [5].

The main information derived by the user equipment is the raw GNSS pseudo-range measurement ρ_R from satellite s to antenna a. It is calculated as the difference between the times of signal arrival sa and signal transmission st; multiplied by the speed of light c:

$$\rho^s_{a,R} = (t^s_{sa,a} - t^s_{st,a})c, \tag{3.1}$$

neglecting all error sources.

3.1.3 Ranging error sources

Ranging data are affected by multiple error sources, some of them can be corrected with the help of navigation data messages, others can be only smoothed. The main sources of error are:

- satellite ephemeris and clock errors;
- ionosphere and troposphere propagation errors;
- tracking errors;
- multipath interference and Not Line Of Sight (NLOS) reception.

Satellite errors depend on the drift of the clock mounted on the satellites and on their broadcasted orbit around the Earth. Both errors are corrected by the control station via parameters passed at the receivers, so they are more accurate with better clock modelling and precise orbit monitoring. These errors vary in the order of hours or even days and contribute to a positioning error of 0.5 meters with the respect of the GNSS constellation. [11]

Ionosphere and troposphere reflect the signal adding a considerable amount of delay. Satellites placed with a higher elevation angle are affected more and often removed from the positioning solution. This delay can be calculated in various ways and change slowly over time, so its contribution to the positioning error is not higher than 0.1 meters for the ionosphere and 0.2 meters for the troposphere.

Tracking errors occurs every time the receiving signal is attenuated, this can be due to the receiver thermal noise, Radio Frequency (RF) interference and also others GNSS transmitting on the same frequency. Tracking errors are correlated over less than a second and cannot be corrected, but only smoothed.

Multipath interference and NLOS are also correlated in a few seconds, but they can be partially corrected with reflection and signal penetration models. This type of errors depends heavily on the environment, the presence of natural and artificial obstacles impact the final measurement. A closed space is able to shield the receiver from receiving any signal.

3.1.4 Positioning

Position solution is determined using passive ranging in three spatial dimensions. With three ranging measurements it is possible to restrict the user location to only two points, one of them usually not viable. The receiver is not synchronized with the constellation, so the measurements are made as pseudo-ranges that depend on both antenna and satellite clock biases:

$$\rho^s_{a,R} = r_{as} + (\delta t^a_c - \delta t^s_c)c, \tag{3.2}$$

where r_{as} is the true range.

The satellite clock bias δt^s_c is measured by the control stations and provided in the navigation messages by the satellite. The antenna clock bias δt^a_c is unknown but common to all simultaneous pseudo-ranges. This means that the positioning problem for the GNSS user equipment is four dimensional, so it requires at least four measurements

The corrected pseudo-range measurement by means of the correlated ranging errors are expressed in terms of satellite position r^i_s, user antenna position r^i_a and receiver clock bias $\delta\rho^a_c$:

$$\rho^s_{a,C} = \sqrt{(\boldsymbol{r}^i_s - \boldsymbol{r}^i_a)^T(\boldsymbol{r}^i_s - \boldsymbol{r}^i_a)} + \delta\rho^a_c. \tag{3.3}$$

The position of the satellite is broadcasted in the navigation messages that describes the satellite orbit, know as ephemeris.

3.1.5 Single-epoch navigation solution

The positioning solution cannot be obtained analytically from the equation 3.3, having a set of pseudo-ranges, because it is nonlinear. It needs to be linearized by performing a Taylor expansion of the first order about the predicted user position r^{i-}_{ia} and clock offset δ^{a-}_c. Naturally the predicted values given to the linearization module are the one of the previous iteration of the algorithm. For $m > 4$ measurement the linearized form is:

$$\begin{bmatrix} \rho^1_{a,C} - \rho^{1-}_{a,C} \\ \rho^2_{a,C} - \rho^{2-}_{a,C} \\ \cdots \\ \rho^m_{a,C} - \rho^{m-}_{a,C} \end{bmatrix} = \boldsymbol{H^i_G} \begin{bmatrix} \boldsymbol{r^{i+}_a} - \boldsymbol{r^{i-}_a} \\ \delta\rho^{a+}_c - \delta\rho^{a-}_c \end{bmatrix} + \begin{bmatrix} \delta\rho^{1+}_{c,\epsilon} \\ \delta\rho^{2+}_{c,\epsilon} \\ \cdots \\ \delta\rho^{m+}_{c,\epsilon} \end{bmatrix}. \tag{3.4}$$

The linearization errors are included in the residuals $\delta\rho^+_{c,\epsilon}$, and the measurement matrix is:

$$\boldsymbol{H^i_G} = \begin{bmatrix} \frac{\partial\rho^1_a}{\partial x^i_a} & \frac{\partial\rho^1_a}{\partial y^i_a} & \frac{\partial\rho^1_a}{\partial z^i_a} & \frac{\partial\rho^1_a}{\partial\rho^a_c} \\ \frac{\partial\rho^2_a}{\partial x^i_a} & \frac{\partial\rho^2_a}{\partial y^i_a} & \frac{\partial\rho^2_a}{\partial z^i_a} & \frac{\partial\rho^2_a}{\partial\rho^a_c} \\ & \vdots & & \\ \frac{\partial\rho^m_a}{\partial x^i_a} & \frac{\partial\rho^m_a}{\partial y^i_a} & \frac{\partial\rho^m_a}{\partial z^i_a} & \frac{\partial\rho^m_a}{\partial\rho^a_c} \end{bmatrix}. \tag{3.5}$$

Each time there are at least 4 simultaneous pseudo-ranges it is possible to estimate the position and clock offset by means of the iterated least-squares algorithm:

$$\begin{bmatrix} \boldsymbol{r_a^{i+}} \\ \delta\rho_c^{a+} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r_a^{i-}} \\ \delta\rho_c^{a-} \end{bmatrix} + (\boldsymbol{H_G^i}^T \boldsymbol{H_G^i})^{-1} \boldsymbol{H_G^i}^T \begin{bmatrix} \rho_{a,C}^1 - \rho_{a,C}^{1-} \\ \rho_{a,C}^2 - \rho_{a,C}^{2-} \\ \cdots \\ \rho_{a,C}^m - \rho_{a,C}^{m-} \end{bmatrix}. \tag{3.6}$$

The presence of multiple pseudo-ranges allows this algorithm to reduce the measurement error and to provide a better estimate, but this solution does not make complete usage of the previous predicted state and depends on a correct initial estimate to correctly converge. Moreover, the linearization and the assumption of simultaneous measurement add remarkable noise to the method.

Most of these problems are fixed using a filtered solution, allowing the previous pseudo-ranges to be helpful in the estimation. Then the usage of nonlinear methods is preferred to cancel any form of approximation.

3.1.6 Signal geometry

The accuracy of a GNSS solution does not depend only on the ranging errors and on the navigation method used, but also on the signal geometry. When the signals come only from satellites on the same direction or that share a plane in the Three Dimensional (3D) space, the rows in the measurement matrix became more linear dependent. In the worse scenario the matrix can become singular and positioning is no longer possible. A spread satellites formation, like the one in Figure 3.2, produce a better estimate because each pseudo-range contribute more to the position solution.

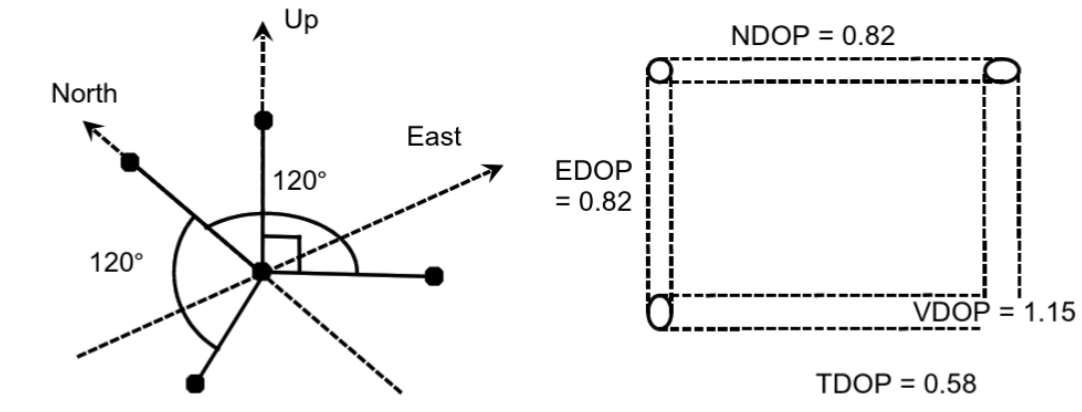

Figure 3.2. Optimal four satellite geometry [5].

The effect of the signal geometry is quantified by the Dilution of precision (DOP), the lower, the better. The positioning error is linearly dependent on the DOP value and the ranging error, so some navigation equipment are able to estimate the positioning accuracy by means of geometry and noise.

3.2 Inertial Measurement Unit

The Inertial Measurement Unit (IMU) combines different inertial sensors in order to produce an accurate measure of specific force and angular rate of the body on which they are mounted. Inertial sensors are typically accelerometers that measure the specific force and gyroscopes for angular rates, that do not need an external reference frame.

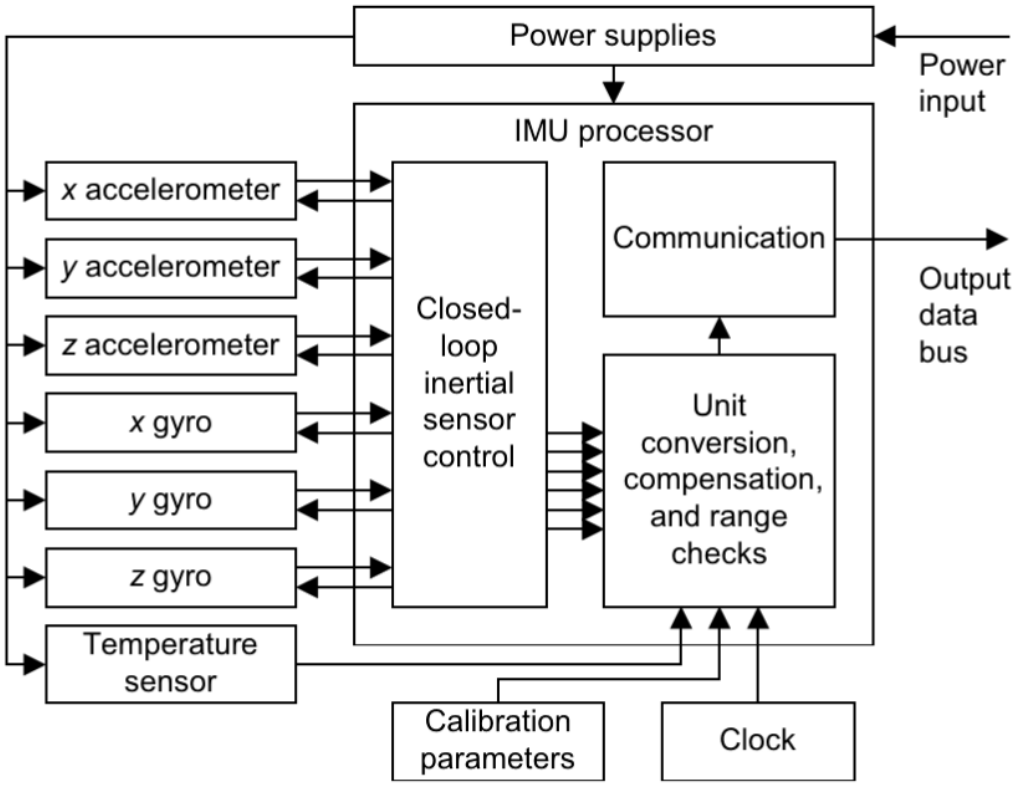

Figure 3.3. Schematic of a IMU. [5]

The information given by the IMU is fundamental for basic flight control because give a reliable rate and acceleration feedback. Moreover, it can give an attitude, velocity and position feedback by integration, but this is less accurate and affected by drift.

In every flight control stack, PX4 included, it is almost impossible to detach the IMU data stream because of its vital role in the flight. Without this information the flight result impossible, even with Radio Controller (RC) manual inputs. This means that the IMU input management is completely in charge of the autopilot, while other sources of information are used to correct the drift of the position and velocity feedback.

Figure 3.3 shows that a IMU is not only in charge of the inertial sensors stream of data, but is a complete computer able to power them, to convert and to process inertial measurements, to provide for sensor compensation and calibration, to interface with a data bus connected to the navigation processor. More expensive IMUs are able to interface with a wide variety of additional sensors.

3.3 Ultra WideBand

Ultra WideBand (UWB) is a radio technology involving generation and transmission of narrow duration pulses that results in very large or wideband transmission bandwidth. The main aspect of UWB that concern this work is its utilization in communication and measurement systems.

The Federal Communication Commission (FCC) on the February 2002 regulated the usage of UWB [2] in the frequency band of 3.1-10.6 GHz, for indoor usage and peer-to-peer operation. UWB is defined by the FCC as any device where the fractional bandwidth this greater than 0.2 or occupies more than 500MHz of the spectrum. The formula for the fractional bandwidth is:

$$FractionalBandwidth = \frac{2(f_h - f_l)}{f_h + f_l} \tag{3.7}$$

where f_h and f_l are the upper and lower frequency of the -10dB emission point.

These regulations permit the unlicensed use of the technology without interfering with other transmission in the same frequency band, requiring sufficiently low emission levels in different applications.

3.3.1 Advantages of UWB

UWB data transmission use Pulse Position Modulation (PPM) or Time Modulation (TM), with a noise-like signal that make interception and detection difficult, combined with its low-power spectral density, it causes very little interference with existing radio systems [8]. The main advantages of UWB are:

- it has a potentially low complexity and cost;
- it has noise-like signal;
- it is resistant to multipath and jamming;

- it has good time domain resolution, allowing location applications.

Its low complexity derive by the nature of the signal transmission: producing a very narrow time domain pulse does not require additional RF stages. At the same time the reverse process of down conversion is not required in the receiver, making it cheaper than other data transmission facilities.

The narrow time domain pulse makes UWB able to offer time precision capabilities more precise than GPS and other radio systems. Together with great material penetration and multipath resistance, UWB is a powerful tool to achieve accurate positioning.

3.3.2 Positioning

The time precision capabilities of UWB are used to estimate with submeter precision the distance travelled by the pulse from the source. Using multiple receivers is possible to achieve position estimation with centimeter precision.

The maximum bandwidth of a UWB signal is 7.5 GHz, this means that the time resolution for a pulse is 133 picoseconds, this translates in a potential 4 cm spatial uncertainty. There are different techniques for positioning in space using UWB time measurement as Time Of Arrival (TOA) or Time difference of arrival (TDOA).

The first way is to directly solving a set of simultaneous equations based on the TDOA measurements. It is necessary to have three measurement between the sensor to be located, called tag, and three sensors with known position, referred as anchors, to solve the problem in the 3D space.

The following equation express the relation between each range ρ from the anchor a and the tag t with the respect of an inertial reference frame:

$$\rho_t^a = \sqrt{(\boldsymbol{r}_a - \boldsymbol{r}_t)^T(\boldsymbol{r}_a - \boldsymbol{r}_t)}. \tag{3.8}$$

A more common solution for an over determined system is to linearize the equations with Taylor expansion and solve the least-square problem. There are different methods to estimate the position with the linearized problem, but they all share the same problems:

- they produce at least two different solutions,
- they require a good initial estimate to produce a correct first linearization.

To avoid these problems is possible to use nonlinear optimization. Different methods can be used: Gauss-Newton, Levenberg-Marquardt, quasi-Newton, DFP formula and BFGS formula. These algorithms are solved iteratively and produce a better estimate the more iteration are scheduled at each epoch.

In this work it is tested and evaluated only the iterative least-square explained in Section 3.1.5 and the Gauss-Newton method to solve the nonlinear least-square problem. This will put a common ground on which more advanced filtered solution are evaluated in Chapter 4.

Chapter 4

Filtering and sensor fusion

The positioning methods cited in Chapter 3 have two major drawback:

- they are single epoch solutions, that cannot utilize previous information to give a more stable and precise estimate;
- they don't offer any capability to fuse information with other sensors.

This chapter unveils a technique to efficiently solve both problems: the Kalman filter, developed by Rudolf E. Kalman [10].

The Kalman filter is a set of mathematical equations that provides an efficient computational recursive means to estimate the state of a process, in a way that minimizes the mean of the squared error [17].

This chapter explores the theory and the implementation of different variants of the Kalman Filter in order to achieve accurate positioning for both Global Navigation Satellite System (GNSS) enabled and denied environments using the ranging measurements of the Ultra WideBand (UWB) technology.

4.1 Kalman filter

The Kalman filter addresses the problem to estimate the state $\boldsymbol{x} \in \mathbb{R}^n$ of a discrete-time controlled process described by the linear state equation:

$$\boldsymbol{x_{k+1}} = \boldsymbol{A}\boldsymbol{x_k} + \boldsymbol{B}\boldsymbol{u_k} + w_k \tag{4.1}$$

where $\boldsymbol{u} \in \mathbb{R}^l$ is the input signal and w is the process noise described by the distribution $p(w) \sim N(0, \boldsymbol{Q})$.

The filter exploit information coming from measurements $\boldsymbol{z} \in \mathbb{R}^m$ described by the equation:

$$\boldsymbol{z_{k+1}} = \boldsymbol{H}\boldsymbol{x_k} + v_k \tag{4.2}$$

where v is the measurement noise described by the distribution $p(v) \sim N(0, \boldsymbol{R})$.

The matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{H}$ are assumed constant in the linear Kalman filter implementation, but they can change at each step if required:

- $\boldsymbol{A}$ is the $n \times n$ state transition matrix that describe the evolution of the state over time;
- $\boldsymbol{B}$ is the $n \times l$ control input matrix that describe the evolution of the system in relation of a control signal;
- $\boldsymbol{H}$ is the $m \times n$ observation matrix that relate the measurements with the state variables.

The Kalman filter estimates a process state using a feedback loop control. The filter predicts the process state for a discrete time and then obtains feedback in the form of measurement. The algorithm is divided in two pieces: the *time update* and *measurement update* equations.

Time update

It projects forward in time the current state and the covariance estimates to obtain the *a priori* estimate $\bar{\boldsymbol{x}}$:

$$\bar{\boldsymbol{x}}_k = \boldsymbol{A}\bar{\boldsymbol{x}}_{k-1} + \boldsymbol{B}\boldsymbol{u}_{k-1}, \tag{4.3}$$

$$\bar{\boldsymbol{P}}_k = \boldsymbol{A}\boldsymbol{P}_{k-1}\boldsymbol{A}^T + \boldsymbol{Q}. \tag{4.4}$$

$\boldsymbol{P}$ represent the error covariance calculated as $\mathbb{E}[(\boldsymbol{x} - \bar{\boldsymbol{x}})(\boldsymbol{x} - \bar{\boldsymbol{x}})^T]$

Measurement update

Represent the feedback: enhancing the *a priori* estimate with measurement data in order to obtain the *a posteriori* estimate x:

$$\boldsymbol{K}_k = \bar{\boldsymbol{P}}_k\boldsymbol{H}^T(\boldsymbol{H}\bar{\boldsymbol{P}}_k\boldsymbol{H}^T + \boldsymbol{R})^{-1}, \tag{4.5}$$

$$\boldsymbol{x}_k = \bar{\boldsymbol{x}}_k + \boldsymbol{K}_k(\boldsymbol{z}_k - \boldsymbol{H}\bar{\boldsymbol{x}}_k), \tag{4.6}$$

$$\boldsymbol{P}_k = (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H})\bar{\boldsymbol{P}}_k. \tag{4.7}$$

The first part compute the Kalman filter gain $\boldsymbol{K_k}$ that, multiplied with the *residual* $\boldsymbol{z}_k - \boldsymbol{H}\bar{\boldsymbol{x}}_k$, is able to correct the estimate. The last step is necessary to update the *a posteriori* error covariance matrix.

4.1.1 Filter design

The design of a Kalman filter consists of defining a suitable linear state equation that describes the system and a suitable measurement equation for each sensor,

that relates it with the state variables. This information composes the matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{H}$.

After that the determination of the process noise covariance $\boldsymbol{Q}$ and of the measurement noise $\boldsymbol{R}$ are generally found via trial and error, or by system identification data analysis. Their magnitude represents the confidence that the filter have on the prediction model and measurement data.

At last the initial conditions influence the filter convergence, choosing a suitable first estimate $\boldsymbol{x}_0$ and initial error covariance $\boldsymbol{P}_0$ is vital to achieve a stable and reliable estimate.

State variables and transition

In order to implement the filter to estimate the position of a flying service robot, as a quadcopter, it is convenient to use the kinematics equations of a particle trajectory in an inertial reference frame. These equations are general to any rigid body moving in a Three Dimensional (3D) space, not depending on the body characteristics. These assumptions make the filter implementation highly reusable on different flying vehicle and even functional for terrestrial ones.

It is also possible to use the equation derived on Chapter 2.1.3, but they require the knowledge of the vehicle parameters and of the control input action. With this system it is possible to achieve better prediction, but the solution is not general and sensible to the drone's parameters variance.

The first step is to choose the state variables: to achieve stable control it is necessary to have at least available the position of the drone and its instantaneous velocity, as seen in Section 2.2.2. The acceleration prediction is needed for enhancing the filter performance during fast change of direction or takeoff/landing phase.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p}_k^n \\ \boldsymbol{v}_k^n \\ \boldsymbol{a}_k^n \end{bmatrix}. \tag{4.8}$$

Once the state variables are fixed it is possible to use the kinematics equations to define relations between them, these relations will compose $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices:

$$\boldsymbol{p}_{k+1}^n = \boldsymbol{p}_k^n + \boldsymbol{v}_k^n \Delta_t + \frac{1}{2}\boldsymbol{a}_k^n \Delta_t^2, \tag{4.9}$$

$$\boldsymbol{v}_{k+1}^n = \boldsymbol{v}_k^n + \boldsymbol{a}_k^n \Delta_t, \tag{4.10}$$

$$\boldsymbol{a}_{k+1}^n = \boldsymbol{a}_k^n, \tag{4.11}$$

with:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{I}\Delta_t & \frac{1}{2}\boldsymbol{I}\Delta_t^2 \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{I}\Delta_t \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \in \mathbb{R}^{9\times 9}, \qquad \boldsymbol{B} = \boldsymbol{0} \in \mathbb{R}^{9\times 0}. \tag{4.12}$$

Measurement matrix

The measurement matrix defines the relation between measurements and state variables. In a linear Kalman filter it is impossible to define a linear equation between pseudo-ranges and positioning, both for GNSS and UWB. The solution is to feed the filter directly with a single-epoch solution $\tilde{\boldsymbol{p}}_{\boldsymbol{k}}^{\boldsymbol{n}}$:

$$\boldsymbol{z} = \tilde{\boldsymbol{p}}_{\boldsymbol{k}}^{\boldsymbol{n}}, \qquad \boldsymbol{H} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \in \mathbb{R}^{3\times 9}. \tag{4.13}$$

The measurement noise matrix $\boldsymbol{R} \in \mathbb{R}^{3\times 3}$ is a constant diagonal matrix with the elements found by trial and errors, but in many cases it will be given by the method used to found the position measurement. Many GNSS receivers are able to produce reasonable values by Dilution of precision (DOP) calculation.

Process noise matrix

The process noise variance derive by the assumption that the acceleration of the system is constant, see Equation 4.11. The acceleration is not actually constant but derives from the forces seen is Section 2.1.3. It is possible to model this noise as continuous time zero mean white noise, assuming that the small changes in velocity average to zero over time. The equation for the discretization of the noise is:

$$\boldsymbol{Q} = \int_0^{\Delta t} \boldsymbol{A}(\boldsymbol{t})\boldsymbol{Q_c}\boldsymbol{A}(\boldsymbol{t})^T dt, \tag{4.14}$$

where the continuous noise is:

$$\boldsymbol{Qc} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \Phi_s \end{bmatrix}. \tag{4.15}$$

The spectral density of the white noise Φ_s is derived by the variance of the acceleration module, there are methods to derive this value, but they require precise modelling and experimental data. Often this factor is a number tuned by trial and errors, so different values will be tested to decide the one that gives better performances.

Another model for the process noise is the piece wise white noise, where the noise is white with zero mean, but only for a single time period. Noise between time periods is uncorrelated:

$$f(x) = \boldsymbol{A}x + \boldsymbol{\Gamma}w \tag{4.16}$$

$$\boldsymbol{\Gamma} = \begin{bmatrix} \frac{1}{2}\Delta_t^2 \\ \Delta_t \\ 1 \end{bmatrix} \tag{4.17}$$

$$\boldsymbol{Q} = \mathbb{E}[\boldsymbol{\Gamma}w(t)w(t)\boldsymbol{\Gamma}^T] = \boldsymbol{\Gamma}\sigma_w^2\boldsymbol{\Gamma}^T \tag{4.18}$$

There is no evidence that this model works better than the previous one, but the tuning parameter σ_w^2 is directly correlated to the amount of acceleration change that we expect from the system.

Initial conditions

Choosing the state variables in the navigation frame allows us to know the initial state of the drone as still in the origin. Putting the state variable initial value as an array of zeros with low initial covariance.

4.1.2 Filter implementation

Filter implementation is handled by the python library FilterPy [9], this library offers a wide set of functions to work with Kalman filters without concerning about the implementation details. The filter will run on a Robot Operating System (ROS2) node, the node code is listed in Appendix A.

4.2 Extended Kalman filter

Working with linear filtering is simple and produces good results, but it comes with some disadvantages:

- it heavily depends on the method performances to calculate the single epoch estimate from the ranging data;
- it is impossible to exploit the information of a single ranging measurement or to exclude a faulty one;
- the linearization adds approximation noise to the estimate.

The Extended Kalman filter (EKF) is the most used approach to work with nonlinear problems, it handles the non linearities by linearizing the system at each time step at the point of current estimate, then the classical Kalman filter equations are used with the linearized system.

In our case non linearities only comes with the measurement model, where ranging data contribute to the system following Equation 3.8. The EKF theory allows also to linearize the system transition equations.

4.2.1 Linearization

In order to find the linearized matrices A and H at each time step, the partial derivatives of the nonlinear functions for the state transition $f(x, u)$ and for the measurement $h(x)$ are used:

$$\boldsymbol{A} = \frac{\partial f(x_t, u_t)}{\partial x}|_{x_t, u_t} \tag{4.19}$$

$$\boldsymbol{H} = \frac{\partial h(x_t)}{\partial x}|_{x_t}. \tag{4.20}$$

Then $\boldsymbol{A}$ and $\boldsymbol{H}$ can be used for the matrix multiplication in the time update and measurement update step.

4.2.2 Filter design

Finding the Jacobian for the measurement model is straightforward:

$$h(t) = \sqrt{(x - x_a)^2 + (y - y_a)^2 + (z - z_a)^2}, \tag{4.21}$$

$$\boldsymbol{H} = \begin{bmatrix} \frac{x - x_a}{\sqrt{(x-x_a)^2+(y-y_a)^2+(z-z_a)^2}} \\ \frac{y - y_a}{\sqrt{(x-x_a)^2+(y-y_a)^2+(z-z_a)^2}} \\ \frac{z - z_a}{\sqrt{(x-x_a)^2+(y-y_a)^2+(z-z_a)^2}} \end{bmatrix}, \tag{4.22}$$

where x, y, z are the estimated coordinates of the drone in the navigation frame, x_a, y_a, z_a are the coordinate of the anchor that produced the ranging data.

4.2.3 Detecting bad measurements

The nonlinear implementation of the positioning problem allow us to analyze the incoming ranging one by one and detect the faulty ones. The technique used to exclude bad measurements is called *gating*, where the *gate* is the algorithms that define which measurements are valid.

The gate used in this work is the *mahalanobis* distance, a statistical measure of the standard deviation distance of a point from a distribution. When a measurement mahalanobis distance D_m goes above the 3.0, it is unlikely that that point reside in the distribution and need to be discarded.

$$D_m = \sqrt{(x - \mu)^T \boldsymbol{S}^{-1} (x - \mu)^2}, \tag{4.23}$$

where μ is the mean value of the distribution and $\boldsymbol{S}$ its covariance.

In reality the noises are not really Gaussian, so a higher gating distance need to be used, trying different values with real measurements and comparing performances.

4.3 Unscented Kalman filter

EKF allow us to work directly with ranging, handling the nonlinear measurement function, moreover it provides capabilities to exploit the data of each single UWB range, including the detection of faulty readings.

The problem with EKF is that the Jacobian calculation cannot be always done analytically and, even when possible, it is an approximation of the system that propagates in the state estimation and noise model.

Recently Unscented Kalman Filter (UKF) is raising in popularity, it is an algorithm capable to perform estimation even when the problem is highly nonlinear. The strength of this method consist of not linearizing the initial model, but using Monte Carlo approach to work with distributions.

UKF uses a deterministic sampling technique know as Unscented Transformation (UT) to pick a minimal set of sample points, called *sigma points*, around the mean. The sigma points are then propagated through the nonlinear functions to predict and update the estimate.

4.3.1 Sigma points

Usually Monte Carlo methods relies on a considerable amount of random generated points, that can make the filter too slow to be implemented in an embedded processor. It is necessary to found a restricted set of sigma points that represent accurately the Gaussian distribution of the estimate.

Van der Merwe's Scaled Sigma Point Algorithm

There are many algorithms in literature to select sigma points, but research and industry have settled for the version published by Rudolph Van der Merwe in his 2004 PhD dissertation [13]. This formulation perform well on a great variety of problems and it is tunable via three parameters: α, β and κ.

This formulation generates $2n+1$ sigma points as shown in Figure 4.1. A larger α spread the point further in the distribution and weight the central point more than the other ones. Sigma point computation equation is:

$$\chi_0 = \mu \tag{4.24}$$

$$\chi_i = \begin{cases} \mu + [\sqrt{(n+\lambda)\Sigma}]_i & \text{for i=1..n} \\ \mu - [\sqrt{(n+\lambda)\Sigma}]_{i-n} & \text{for i=(n+1)..2n} \end{cases}, \tag{4.25}$$

where $\lambda = \alpha^2(n+\kappa)$.

The generated points have different weights and, for the central point, the weight used for the mean value is different from the one used for covariance calculations:

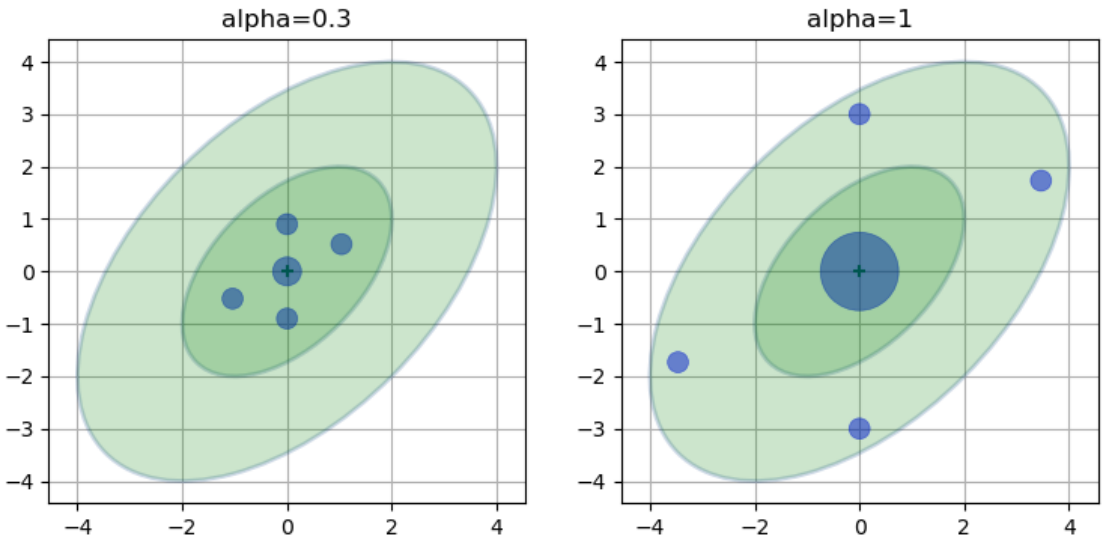

Figure 4.1. Sigma points with different α values.

$$W_0^m = \frac{\lambda}{n+\lambda}, \tag{4.26}$$

$$W_0^c = \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta, \tag{4.27}$$

$$W_i^m = W_i^c = \frac{1}{2(n+\lambda)}. \tag{4.28}$$

4.3.2 Filter math

This section shows the steps in the UKF algorithm, composed by the time update and measurement update step.

Time update

The UKF predict step starts by generating the sigma point from the state variable mean and covariance matrix, as seen in Section 4.3.1. Then the sigma points are passed through the nonlinear process model:

$$\boldsymbol{\Upsilon} = f(\boldsymbol{\chi}, \Delta_t) \tag{4.29}$$

The *a priori* sigma points Υ are used to generate the *a priori* estimate and covariance by the UT:

$$\bar{\boldsymbol{x}} = \sum_{i=0}^{2n} w_i^m \boldsymbol{\Upsilon}_i, \tag{4.30}$$

$$\bar{\boldsymbol{P}} = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\Upsilon}_i - \bar{\boldsymbol{x}})(\boldsymbol{\Upsilon}_i - \bar{\boldsymbol{x}})^T + \boldsymbol{Q}. \tag{4.31}$$

Measurement update

The *a priori* sigma points are update in the measurement space, using the function $h(x)$:

$$\boldsymbol{\zeta} = h(\boldsymbol{\Upsilon}), \tag{4.32}$$

again the UT is used to transform the *a posteriori* sigma point ζ in the *a posteriori* mean and covariance:

$$\boldsymbol{\mu}_z = \sum_{i=0}^{2n} w_i^m \boldsymbol{\zeta}_i, \tag{4.33}$$

$$\boldsymbol{P}_z = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\zeta}_i - \boldsymbol{\mu}_z)(\boldsymbol{\zeta}_i - \boldsymbol{\mu}_z)^T + \boldsymbol{R}. \tag{4.34}$$

To compute the Kalman gain we first need to calculate the cross covariance of the state and measurements:

$$\boldsymbol{P}_{xz} = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\Upsilon}_i - \bar{\boldsymbol{x}})(\boldsymbol{\zeta}_i - \boldsymbol{\mu}_z)^T, \tag{4.35}$$

$$\boldsymbol{K} = \boldsymbol{P}_{xz} \boldsymbol{P}_z^{-1}. \tag{4.36}$$

Once calculated the Kalman gain it is used to compute the *a posteriori* estimate and covariance matrix:

$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \boldsymbol{K}(\boldsymbol{z} - \boldsymbol{\mu}_z) \tag{4.37}$$

$$\boldsymbol{P} = \bar{\boldsymbol{P}} - \boldsymbol{K} \boldsymbol{P}_z \boldsymbol{K}^T \tag{4.38}$$

4.3.3 Filter implementation

The filter implementation is equivalent to the one for EKF, it is possible to use the same matrices and sources of data. The difference is that the measurement update is done directly using the ranging formula.

The critical part is related to the generation and handling of the sigma points, this part is also managed by the library *FilterPy* [9]. The implemention work is reduced to tune the point generator parameters.

Chapter 5

Simulation

The methods proposed and explained in Chapter 4 have been simulated in order to understand their advantages and disadvantages and to compare them. The goal is to find the better method to implement in a real scenario. This chapter focus on the results obtained by the methods in a complete scenario, without explaining in detail the parameters tuning done in hours of simulation.

To simulate a realistic scenario it has been used the Gazebo Software In The Loop (SITL) feature present with the PX4-Autopilot package, already cited in Section 2.2.3. This feature allows the interaction with a simulated vehicle and flight stack with the same characteristics of the real system.

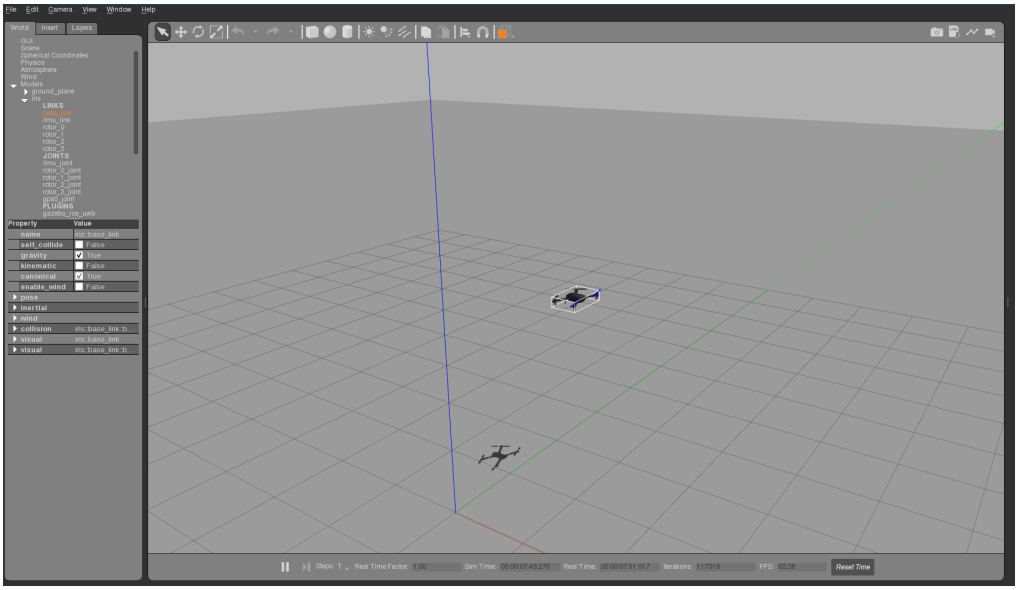

Figure 5.1. Gazebo simulation environment while Iris is hovering.

In Figure 5.1 the graphical environment is visualized with the default configuration while the standard drone, called Iris, is hovering at 2.5 meters. From this view it is possible to keep track of the drone's behavior.

Figure 5.2 is an example of a real time flight plot, in this case a simple takeoff and land is performed. The xyz coordinates are mapped as *North-East-Down* in the inertial navigation frame. The *VehicleLocalPosition* topic indicates the position estimated from PX4, using a noisy Global Navigation Satellite System (GNSS) signal coupled with the Inertial Measurement Unit (IMU) data; the *VehicleLocalPositionGroundtruth* topic indicates the real position of the drone broadcasted by the simulator.

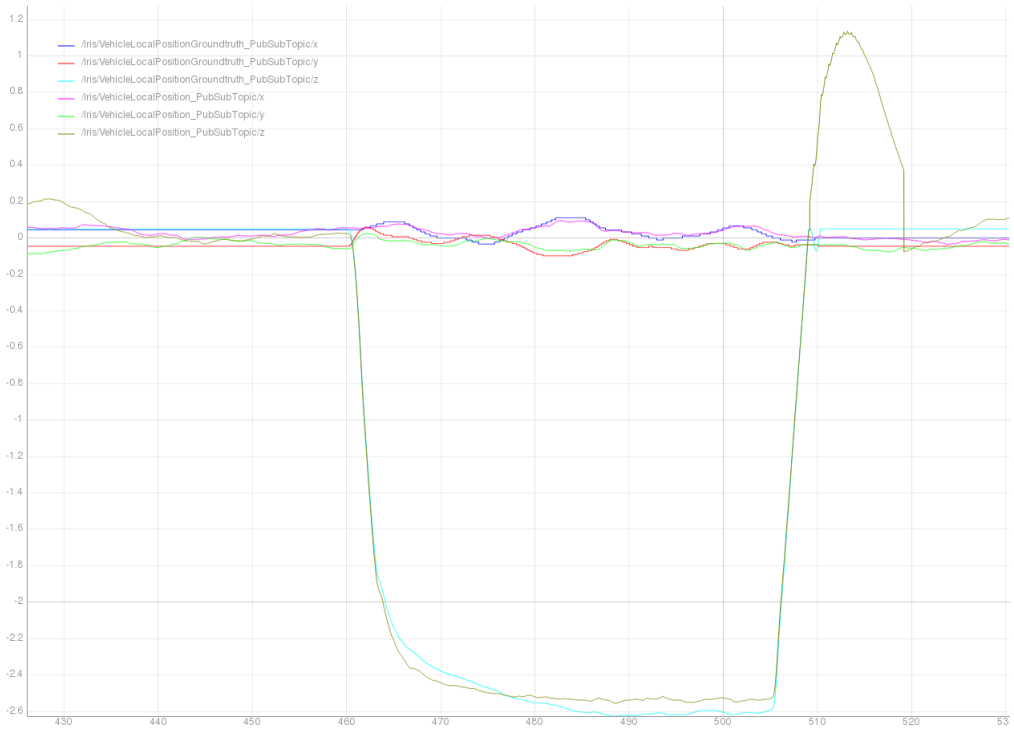

Figure 5.2. Example of a flight in Gazebo.

It is noticeable that the z coordinate is noisier of the other ones, in a real scenario this is caused by the common higher vertical Dilution of precision (DOP) with the respect of the horizontal one. In the simulation this effect is reproduced by doubling the z-axis random walk and noise density of the GNSS simulated receiver.

Then the drone does not recognize landing, the internal estimator of PX4 believes that it is underground, this is not a problem because it happens only after the vehicle is already landed.

5.1 UWB Gazebo plugin

Gazebo simulation environment does not include a Ultra WideBand (UWB) sensor plugin to include anchors and tags. Part of the work for this thesis consisted on developing a Gazebo plugin that allows to place UWB sensors, extract ranging data from them and simulate a noise model. The implementation characteristics of the plugin are unveiled in the Appendix A.

The code implement a *ModelPlugin*: a plugin that need to be linked to a Gazebo model. Embedding any model with the UWB plugin makes it an anchor, and passing it the *PubRange* parameter makes it publish its ranging data, with the respect of any other anchor in the environment, in a Robot Operating System (ROS2) topic.

5.1.1 Implementation

The implementation is simple: every model embedded with the plugin publish its real position in shared a Gazebo topic. Then if an anchor act as a tag, having the *PubRange* parameter, subscribe itself to that topic and start calculating its distance between each anchor, broadcasting it in a ROS2 topic. Other functions of the plugin are:

- setting the publishing rate of each anchor;
- setting the reference frame for the anchors positioning;
- apply Gaussian noise to the ranging measurement with custom mean and variance;
- apply a random offset to the anchor positioning.

In the future it can be added a better noise model exploiting the relative attitude of the anchors and the presence of obstacle in a Not Line Of Sight (NLOS) case.

5.1.2 Usage

The plugin is used taking into account the real UWB anchors characteristics used for the experimental phase, reported in Table 5.1.

Update rate	20 Hz
Ranging variance	0.0025 m
Anchor offset	0.2 m

Table 5.1. UWB anchors parameters

The anchor offset accounts for the misplacement of an anchor mounted by a human operator; for the tag mounted on the drone chassis it reproduces the offset from the center of mass of the drone.

In a simulation eight anchors will be placed in a box formation, reproducing a typical setup in a rectangular room. NLOS and multipath effect are not directly accounted, it is advised to implement them in a future work.

5.2 Single sensor/epoch solutions

The simulation performed in this section use only one type of sensor to perform a predefined set of maneuvers. As shown in Figure 5.3, the drone takeoff at 3 meters, then perform a series of maneuvers that consist in sudden change of direction and acceleration, at least it lands in the starting location.

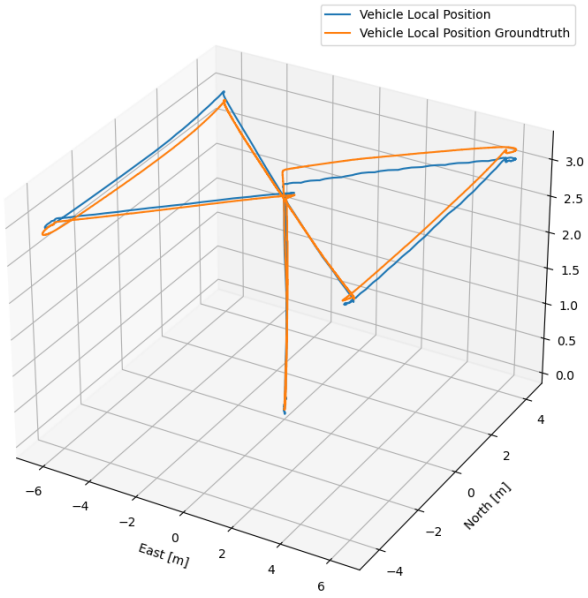

Figure 5.3. Maneuvering sequence used in simulation.

The IMU is always active to allow attitude and angular rate control, completely delegated to PX4 estimator and control stack.

5.2.1 GNSS flight

The first flight is needed to demonstrate the standard path that the following simulations will use and to build a common ground to compare all methods performances.

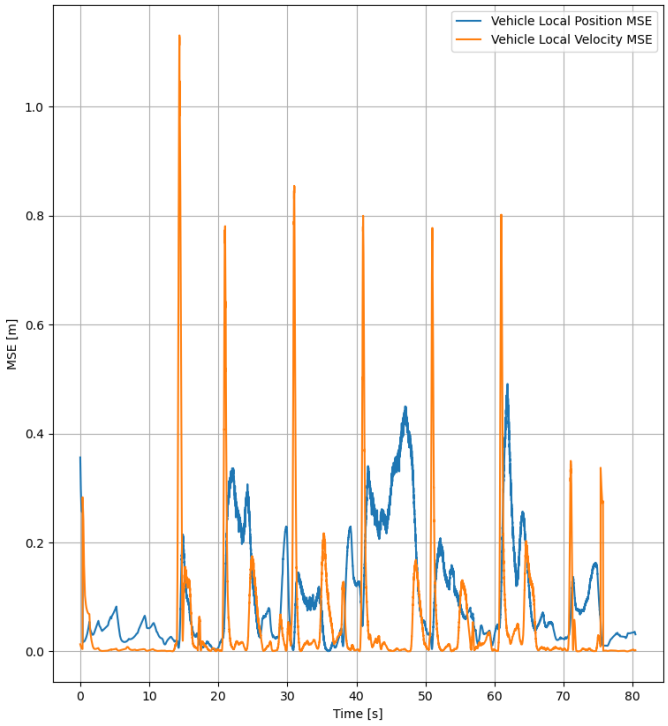

Figure 5.4. MSE plot for GNSS flight.

The performance index used is the Mean Square Error (MSE) computed at each time for the position solution. Then the higher value of MSE during flight will be registered, this value describe the worse point, where the measured method can make the control stack instable by feeding it with wrong data. A stable flight has a higher priority then a precise one for Unmanned Aerial Vehicle (UAV)s, losing control is cause of damage for the vehicle and the surrounding things.

The position and velocity MSE are decoupled to demonstrate the difference between methods that provide a direct measurement on velocity and others that cannot do it. The plot in Figure 5.4 shows a spike of MSE of 0.46 meters for the position, the MSE relative to the velocity reach 1.1 m/s when sudden change of

direction are performed.

The GNSS equipment used provide an already partially filtered solution and provide also a velocity feedback that is broadcasted to the drone's control stack.

5.2.2 UWB flight

In this simulation the UWB solution given by the least-square algorithm is feed to the internal estimator of PX4. This solution does not provide a direct feedback on the velocity estimate, producing a higher error when higher velocities are reached. The positioning error reach values of 5.6 meters of position MSE and 1.0 m/s for velocity one, see Figure 5.5.

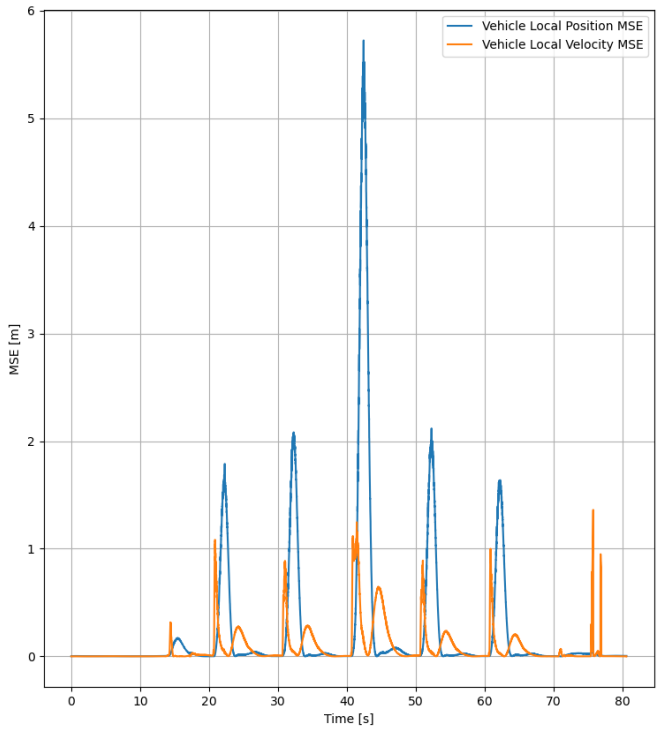

Figure 5.5. MSE plot for UWB flight.

The estimate allows the drone to flight, but the errors weaken the control loop. Figure 5.6 visualize a noise and instable flight, trying to flight faster in this configuration will result in the loss of control, causing damage to the drone. The unfiltered

UWB estimation have much worse performance, with the respect of GNSS one, because the GNSS solution is already filtered by the navigation processor inside the sensor, hence the necessity to filter also the UWB data.

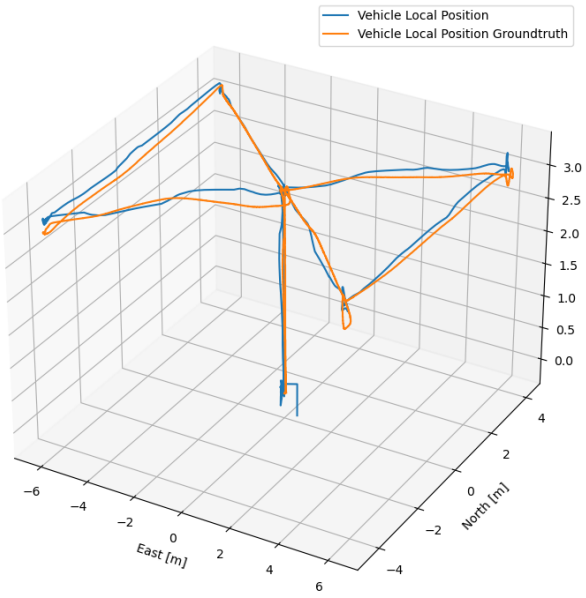

Figure 5.6. 3D navigation using only UWB.

5.3 Filtered solutions

Using more sophisticated single epoch solutions can solve partially this problem, instead, using filtered ones allows the drone to be enhanced with velocity estimate and a smoother position estimate. These methods provide a more precise and stable feedback to the control loop, allowing accurate flight, even at higher speeds.

5.3.1 Linear filtering

Linear filtering is the easier way to enhance the estimate and give excellent result with low velocities. As seen in Figure 5.7 the estimate is even better than the one

of the pre-filtered GNSS solution. This result demonstrates that the UWB anchor can be more precise than GNSS if correctly filtered.

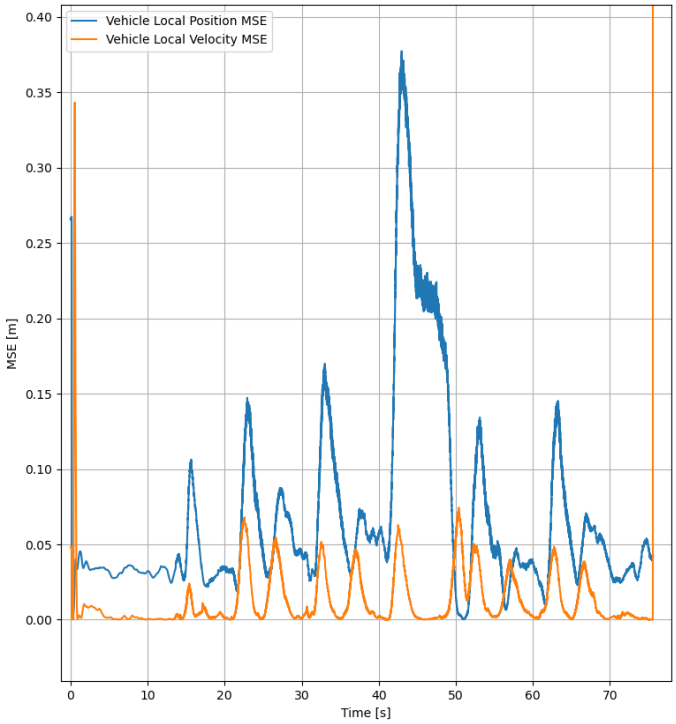

Figure 5.7. MSE plot for Kalman flight, using only UWB.

The position MSE spikes at 0.37 meters, the velocity one at 0.07 m/s. The plot indicated that the filter is not able to fully converge even if the drone is still. This happens because the filter smooth only the position estimate using past data, but does not act on the single ranging measurement to compensate for the ranging offset.

5.3.2 Nonlinear filtering

Nonlinear filtering use directly the ranging data to find the position and velocity estimate, this allows to fully compensate for the anchors offset. It is also possible to use the gating techniques to filter out faulty measurements. Unscented Kalman Filter (UKF) gives the best results, even compared with is famous counterpart: Extended Kalman filter (EKF), but the difference is almost negligible.

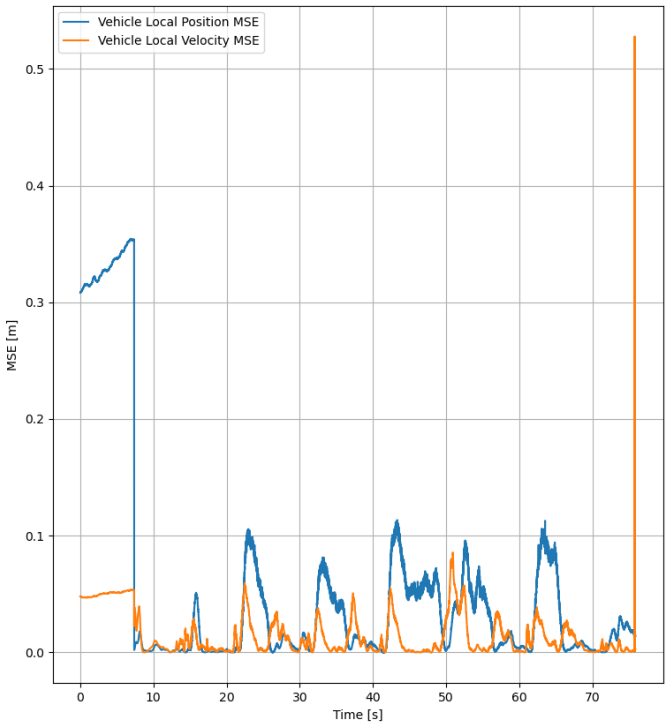

Figure 5.8. MSE plot for UKF flight, using only UWB.

With correct tuning is possible to reach a precision of the order of centimeters, see Figure 5.8. The first part of the graph represents the estimate when the filter is *warming up*, after that the error drops and the drone starts flying.

The position MSE barely cross the 10 centimeters and the velocity one is always under 8 centimeters, even at the maximum flight speed. Figure 5.9 represent a flight that is precise and stable enough to be used safely in a closed environment.

5.3.3 Sensor fusion

The filter is able to fuse also the GNSS data when possible, allowing the transition between GNSS enabled and denied environments. The final goal is to have a coherent estimate in both environments and in all transitions between them. The following simulation tries to reproduce all the meaningful transitions in a dynamic setup: first the drone start in a GNSS only space, then it moves towards the anchors. Inside the anchors space the GNSS is completely shutdown, only to be

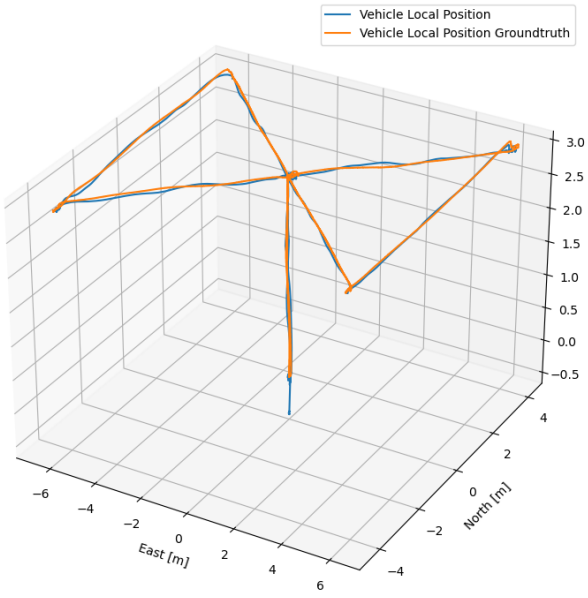

Figure 5.9. 3D navigation using only UKF filtered UWB.

enabled when there are no anchors visible.

Figure 5.10 shows stable flight in every state, the estimation oscillates slightly on entering the UWB enable space, this happens because the UWB reference frame is not aligned with the GNSS one and offsets need to be estimated. Exiting the UWB space the GNSS is enabled only when there are no anchors visible, so the drone is working with less than 3 range for little time period.

Sensor state value	Scenario
0	No sensors
1	Only GNSS
2	Only UWB
3	UWB+GNSS sensor fusion

Table 5.2. Sensor state encoding

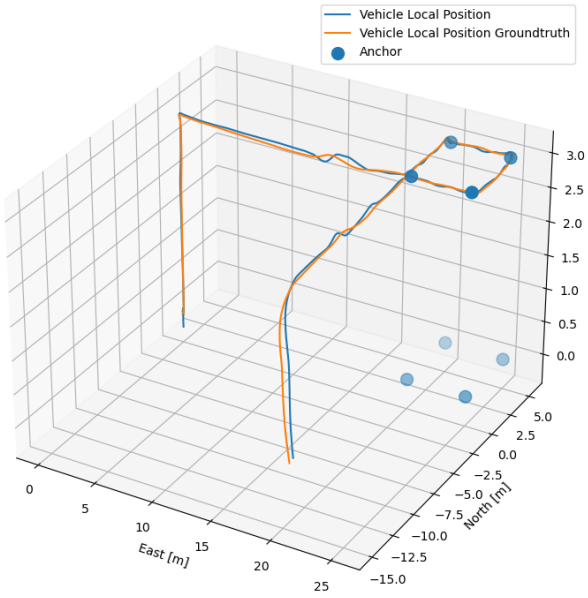

Figure 5.10. 3D navigation in a mixed environment.

A real scenario will never allow the drone to lose both GNSS and UWB signals, but the drone can *wait on the verge* of the two spaces a reliable estimate. This last test represented a stress test for the whole system. Figure 5.11 shows the performance results of this test, *Sensor state* represent the presence of sensors as indicated in Table 5.2.

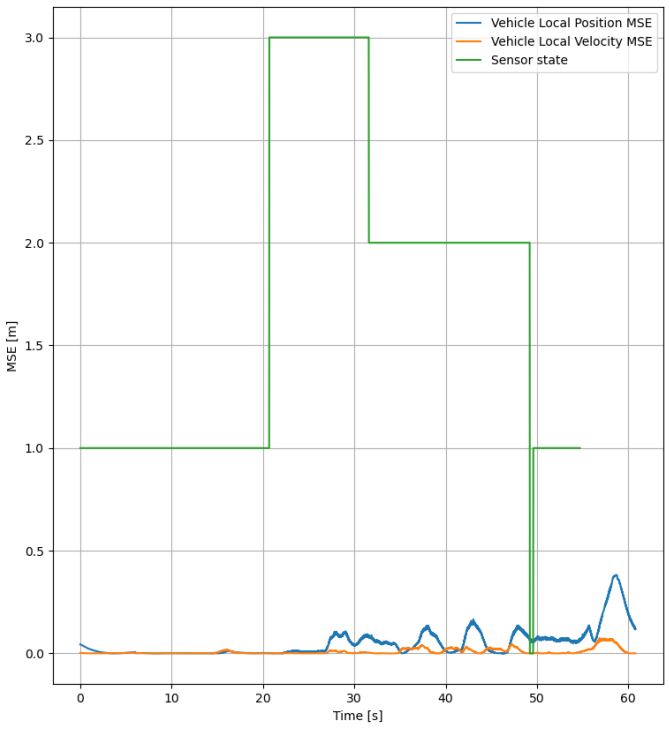

Figure 5.11. MSE plot in a mixed environment.

Chapter 6

Experimental results

This chapter's goal is to translate the methods explained until now and simulated in Chapter 5. First the setup of the drone and of the sensors used are presented, then two different scenarios are built to demonstrate the functionalities of the algorithms in a real scenario.

6.1 Drone setup

The drone used is a Holibro ®X500 frame, shown in Figure 6.1: entirely carbon fiber chassis with 16 mm arms. The kit related with this frame comes with all the necessities to build a functioning vehicle providing space and mounting holes to expand its capabilities. The basic setup is composed of:

- Pixhawk 4 autopilot
- Power Management PM07
- Pixhawk4 GPS
- Motors - 2216 KV880
- BLHeli S ESC 20A
- Propeller 1045
- 433MHz Telemetry Radio / 915MHz Telemetry Radio
- Power and Radio Cables
- Battery Straps

The total dimension of this setup is 410*410*300 mm, the weight is 978 Kg. To this setup it is added a laser sensor to a Li-Po battery 4S and a Raspberry Pi 4 to run Robot Operating System (ROS2) nodes.

Figure 6.1. Holibro ®X500 kit [7].

6.1.1 UWB sensors

The Ultra WideBand (UWB) tag mounted on the drone and the anchors placed on the test environment are provided by the DecaWave EVB1000 evaluation board. The board provide the interface for the DW1000 IEEE802.15.4-2011 UWB-compliant wireless transceiver Integrated Circuit (IC), including a microprocessor, antenna and resident firmware.

The DW1000 IC is a transceiver that enables to develop cost-effective solutions for precise indoor positioning within 10 centimeters. The evaluation board is connected to the Raspberry Pi 4 with a Universal Serial Bus (USB).

6.2 Software stack

The software stack modules are reviewed in Figure 6.3.

The UWB sensor data and the Global Navigation Satellite System (GNSS) ones are parsed by ROS2 nodes that acts as drivers and that can be enabled and disabled with services. The least square UWB algorithm is always running to provide an initial estimate and align the anchor constellation with the respect of the inertial navigation frame. The Unscented Kalman Filter (UKF) node retrieves the sensors' data and perform estimation, the node provide also watchdogs to monitor the

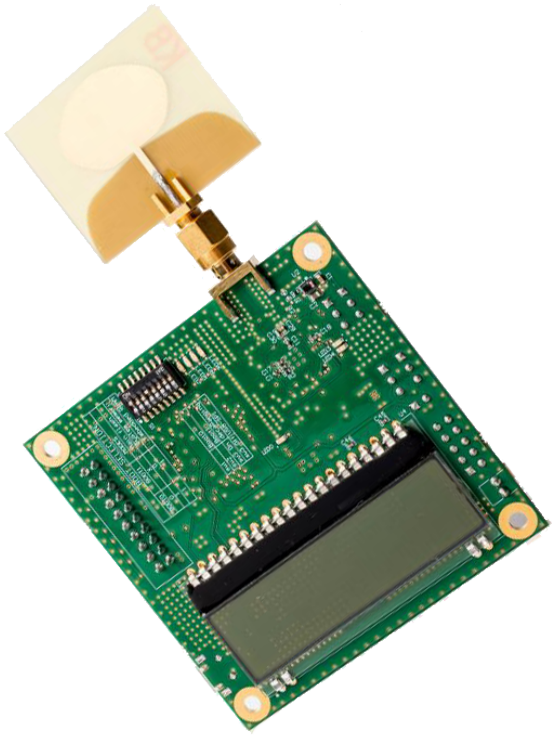

Figure 6.2. DecaWave EVB1000 evaluation board [3].

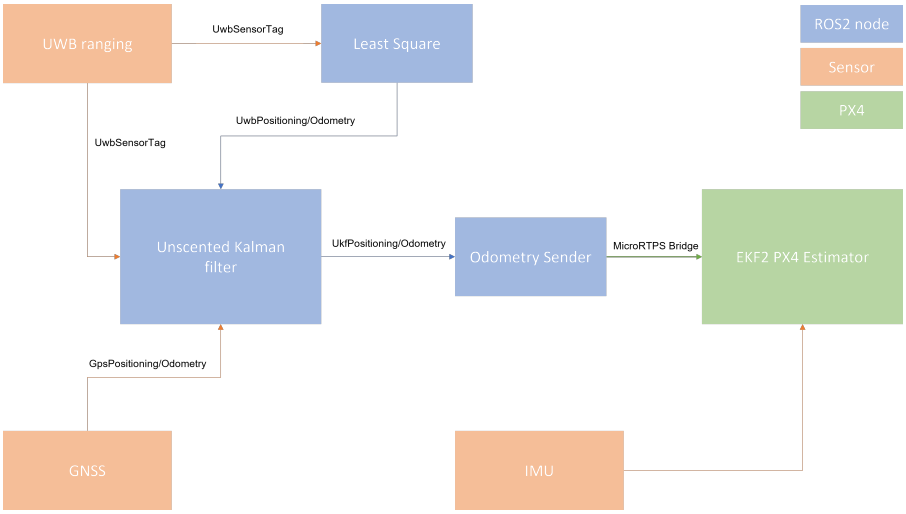

Figure 6.3. Software stack architecture.

health of the sensors and self tests to provide only a reliable estimate. Odometry

sender is a node that perform the transformations needed to send data in the *VehicleVisualOdometry* topic. The internal PX4 estimator is disconnected from all the sensors but Inertial Measurement Unit (IMU) via parameters, the estimate is sent using the external visual odometry interface that PX4 provides. The connection between ROS2 nodes and PX4 is achieved by means of the *microRTPS Bridge*.

This stack provides complete isolation of the tested interfaces to the autopilot. If a ROS2 node, the Raspberry or even the sensors are not available, the absence of data in the visual odometry port allows PX4 to interrupt the positioning flight and switch to a controlled descent. If the autopilot receive data from a Radio Controller (RC) it also detaches the whole system to give the control to the human operator.

6.3 Test environment

Two testing scenarios are presented for evaluating the system performances. The first one in a controlled cage, where the drone cannot hit anything and anyone, mostly utilized for tuning and first flights. The second one is in an open field where more complex flight path can be recreated.

6.3.1 Cage flight

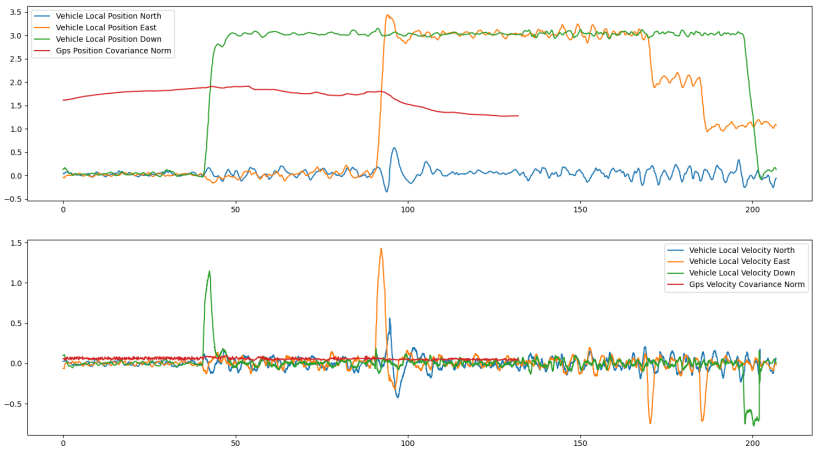

Figure 6.4. Flight maneuvers in the cage.

In the cage 8 anchors are placed, fixing them on the borders of the cage, the drone is placed in the middle. The flight plan consists in a takeoff using GNSS

positioning, then after a maneuver the GNSS feed is interrupted and the UWB one enabled. In Figure 6.4 the interrupted line regarding the Global Positioning System (GPS) covariance norm represent the missing information. Another set of maneuvers are performed in UWB only flight, then the vehicle land.

This flight demonstrates the capabilities of the system to flight in a GNSS enabled and denied environment without losing control and precision. Most of the parameters tuning it has been done in this cage, until a reliable system was reached.

6.3.2 Open space flight

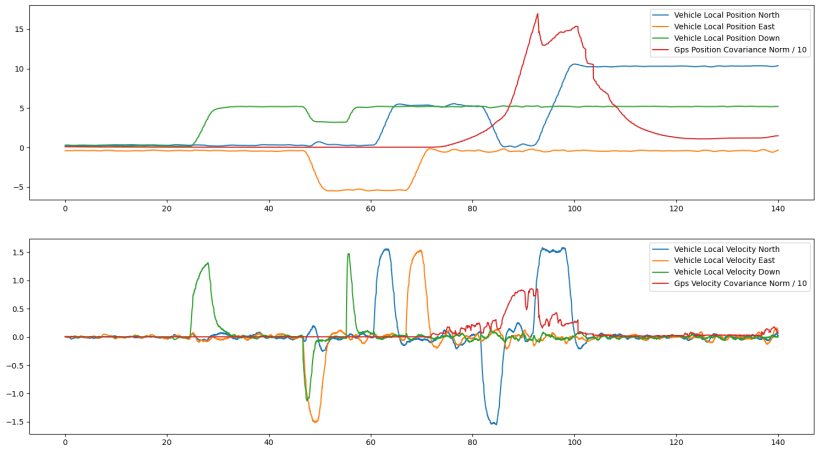

Figure 6.5. Flight maneuvers in an open field.

More complicated tests are conducted in open field, only after the tuning phase is complete. In the field is not possible to place the anchors precisely because of the absence of points of reference. Six anchors are placed using only a 5 meters tape measure, some tripods has been used to have some of them on a different plane.

The flight presented in Figure 6.6 shows the path done by the quadcopter. After landing and some initial movements the GNSS is jammed via software to recreate an indoor environment, in Figure 6.5 is plotted the increasing covariance matrix norm of the GNSS that reached a peak of 170 meters. The system is able to operate without interruption of service, also with corrupted and noisy GNSS navigation data.

At the end of this flight each anchor is shutdown sequentially with the drone still flying. After reaching only 2 active anchors the estimator started to diverge, then the stream of data was interrupted automatically by a threshold on the covariance

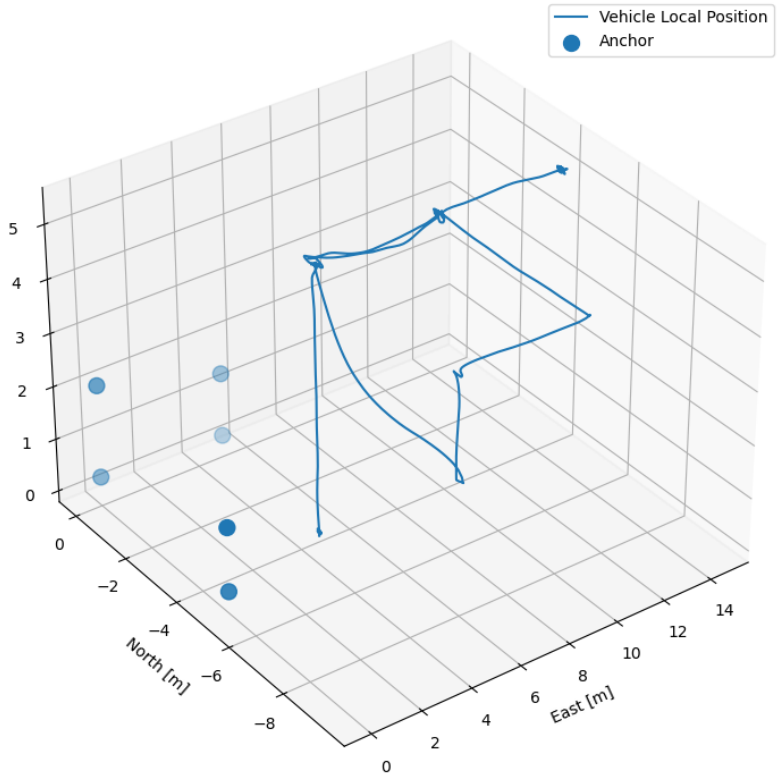

Figure 6.6. 3D flight plot in an open field.

matrix norm. After that the flight control is delegated to PX4 that perform a safe descent.

This flight in particular is the demonstration that transitions between sensors' feed is possible and stable and that only three anchors are enough to not lose control.

Chapter 7

Conclusions

The simulated and experimental results demonstrate that Ultra WideBand (UWB) flight is possible. Not only the anchors provide easy installation and good coverage, but with correct filtering it is possible to obtain better positioning with the respect of classical methods.

Linear filtering is reliable enough to flight at lower speeds with a good anchors constellation, but does not provide facilities to correct the ranging errors or to compensate for a faulty measurement. The advantages of this method are the ease of software integration and the low computing resources utilization.

Nonlinear filtering allows a direct control on the ranging data, this allows to even enhance position estimate with a single anchor when coupled with other positioning system, or to exclude one faulty anchor from the estimate if it does not behave correctly. The simulation results are astounding: centimeter like precision is achieved. In the experimental setup it is possible to flight even when manually covering an anchor to inject noise or by shutting down anchors and GNSS feedback randomly during flight.

This work focused on the Unscented Kalman Filter (UKF) approach, evaluating its performance in various scenarios. The complete setup demonstrate the capabilities of the system to withstand the transition from a GNSS enabled environment to a GNSS denied one where UWB anchors are installed, and vice versa. The drone is able to flight at high speed during the transitions without losing its stability and precision properties.

The work in this field is not over. The applications and scenarios that unveils after this point are countless. First it was not possible to test the estimation algorithm in a real indoor environment, so it is possible to design a path planning algorithm that allows to navigate indoor and allows transitions with the outdoor space, starting from the estimates provided by the methods presented. Moreover, the UWB technology used is mature but the modelling used in this work is simple, it is possible to enhance the UWB plugin capabilities to emulate Not Line Of Sight (NLOS) propagation and signal attenuation.

Appendix A

Code

This appendix present code fragments that allow the reader to explore the implementation details of the algorithms presented in this work.

A.1 Gazebo UWB plugin

A.1.1 ros2_px4_gazebo_uwb.hpp

```
#ifndef ROS2_PX4_GAZEBO_UWB_HPP_
#define ROS2_PX4_GAZEBO_UWB_HPP_

#include <gazebo/physics/physics.hh>
#include <gazebo/common/Plugin.hh>

namespace gazebo
{

  class RosPx4GazeboUwbPrivate;

  class RosPx4GazeboUwb : public ModelPlugin
  {
  public:
    /// Constructor
    RosPx4GazeboUwb();

    /// Destructor
    virtual ~RosPx4GazeboUwb();

    // Documentation inherited
    virtual void Load(physics::ModelPtr model, sdf::ElementPtr sdf)
    override;

  private:
    /// Private data pointer
```

```
26      std::unique_ptr<RosPx4GazeboUwbPrivate> impl_;
27    };
28
29  } // namespace gazebo
30
31  #endif // ROS2_PX4_GAZEBO_UWB_HPP_
```

A.1.2 ros2_px4_gazebo_uwb.cpp

```
1   #include <ros2_px4_gazebo/ros2_px4_gazebo_uwb.hpp>
2
3   #include <ignition/math/Rand.hh>
4   #include <gazebo_ros/node.hpp>
5   #include <gazebo_ros/utils.hpp>
6   #include <gazebo/transport/transport.hh>
7   #include <ros2_px4_interfaces/msg/uwb_sensor.hpp>
8
9   #include <rclcpp/rclcpp.hpp>
10
11  namespace gazebo
12  {
13    class RosPx4GazeboUwbPrivate
14    {
15    public:
16      /// Callback to be called at every simulation iteration
17      /// \param[in] info Updated simulation info
18      void OnUpdate(const common::UpdateInfo &info);
19
20      /// Callback to be called at every anchor message received by
     the sensor
21      /// \param[in] msg Incoming anchor message
22      void AnchorCallback(ConstPoseStampedPtr &_msg);
23
24      /// Pointer to the link, model and world
25      physics::LinkPtr link_{nullptr};
26      physics::ModelPtr model_{nullptr};
27      physics::WorldPtr world_{nullptr};
28
29      /// Pose of the link
30      ignition::math::Pose3d link_pose_;
31
32      /// The reference model and link to which calculate the pose
33      physics::ModelPtr reference_model_{nullptr};
34      physics::LinkPtr reference_link_{nullptr};
35
36      /// Gazebo node
37      transport::NodePtr gazebo_node_{nullptr};
38
39      /// Gazebo pub/sub to anchor broadcast
40      transport::PublisherPtr anchor_pub_{nullptr};
41      transport::SubscriberPtr anchor_sub_{nullptr};
```

```
42
43    /// Pointer to ROS node
44    gazebo_ros::Node::SharedPtr ros_node_{nullptr};
45
46    /// ROS publisher for sensor ranging data
47    rclcpp::Publisher<ros2_px4_interfaces::msg::UwbSensor>::
     SharedPtr sensor_pub_{nullptr};
48
49    // Topic names
50    std::string anchor_topic_{"/uwb_anchors"};
51    std::string sensor_topic_{"/uwb_sensor_"};
52
53    /// Keep track of the last update time.
54    common::Time last_time_;
55
56    /// Publish rate in Hz.
57    double update_rate_{1.0};
58
59    /// Anchor unique ID
60    std::string anchor_id_;
61
62    /// Gaussian noise
63    double gaussian_noise_;
64    double anchor_offset_[3] = {0, 0, 0};
65
66    /// Pointer to the update event connection
67    event::ConnectionPtr update_connection_{nullptr};
68  };
69
70  RosPx4GazeboUwb::RosPx4GazeboUwb()
71      : impl_(std::make_unique<RosPx4GazeboUwbPrivate>())
72  {
73  }
74
75  RosPx4GazeboUwb::~RosPx4GazeboUwb()
76  {
77    impl_->ros_node_.reset();
78    if (impl_->gazebo_node_)
79    {
80      impl_->gazebo_node_->Fini();
81    }
82    impl_->gazebo_node_.reset();
83  }
84
85  // Load the plugin
86  void RosPx4GazeboUwb::Load(physics::ModelPtr model, sdf::
     ElementPtr sdf)
87  {
88    std::string link_name;
89    std::string reference_model_name;
```

```
90      std::string reference_link_name;
91
92      // Get model, world and time
93      impl_->model_ = model;
94      impl_->world_ = impl_->model_->GetWorld();
95      impl_->last_time_ = impl_->world_->SimTime();
96
97      // Configure the Gazebo node
98      impl_->gazebo_node_ = boost::make_shared<transport::Node>();
99      impl_->gazebo_node_->Init(impl_->world_->Name());
100
101     // <update_rate> is the rate at which publish UWB packets
102     if (!sdf->HasElement("update_rate"))
103     {
104       gzwarn << "UWB plugin missing <update_rate>, defaults to 1.0
    Hz" << std::endl;
105     }
106     else
107     {
108       impl_->update_rate_ = sdf->GetElement("update_rate")->Get<
    double>();
109     }
110
111     // <anchor_id> is an unique ID that represent the UWB tag
112     if (!sdf->HasElement("anchor_id"))
113     {
114       impl_->anchor_id_ = std::to_string(rand());
115       gzdbg << "UWB plugin missing <anchor_id>, assigning random ID
    : " << impl_->anchor_id_ << std::endl;
116     }
117     else
118     {
119       impl_->anchor_id_ = sdf->GetElement("anchor_id")->Get<std::
    string>();
120     }
121
122     // <link_name> is the name of the link where the UWB tag is
    attached
123     if (!sdf->HasElement("link_name"))
124     {
125       gzerr << "Missing <link_name>, cannot proceed" << std::endl;
126       return;
127     }
128     else
129     {
130       link_name = sdf->GetElement("link_name")->Get<std::string>();
131     }
132
133     // Checking if the link exists
134     impl_->link_ = model->GetLink(link_name);
```

```
135     if (!impl_->link_)
136     {
137       gzerr << "link_name: " << link_name << " does not exist" <<
    std::endl;
138       return;
139     }
140 
141     // <reference_model_name> is the model to be taken as reference
     frame, defaults to world
142     if (!sdf->HasElement("reference_model_name"))
143     {
144       gzdbg << "Missing <reference_model_name>, defaults to world"
    << std::endl;
145       reference_model_name = "world";
146     }
147     else
148     {
149       reference_model_name = sdf->GetElement("reference_model_name"
    )->Get<std::string>();
150     }
151 
152     // <reference_link_name> is the link of <reference_model_name>
    to be taken as reference frame
153     if (!sdf->HasElement("reference_link_name"))
154     {
155       gzdbg << "Missing <reference_link_name>, using model as
    reference" << std::endl;
156     }
157     else
158     {
159       reference_link_name = sdf->GetElement("reference_link_name")
    ->Get<std::string>();
160     }
161 
162     // Checking if custom reference frame exists
163     if (reference_model_name != "/world" && reference_model_name !=
     "world" &&
164         reference_model_name != "/map" && reference_model_name != "
    map")
165     {
166       impl_->reference_model_ = impl_->world_->ModelByName(
    reference_model_name);
167       if (!impl_->reference_model_)
168       {
169         gzwarn << "<reference_model_name> [" <<
    reference_model_name << "] does not exist" << std::endl;
170       }
171 
172       impl_->reference_link_ = impl_->reference_model_->GetLink(
    reference_link_name);
```

```
173       if (!impl_->reference_link_)
174       {
175         gzwarn << "<reference_link_name> [" << reference_link_name
    << "] does not exist" << std::endl;
176       }
177     }
178
179     // <anchor_offset> add a constant offset from a uniform
    distribution
180     if (sdf->HasElement("anchor_offset"))
181     {
182       double anchor_offset_std = sdf->GetElement("anchor_offset")->
    Get<double>();
183
184       impl_->anchor_offset_[0] = ignition::math::Rand::DblUniform(-
    anchor_offset_std, anchor_offset_std);
185       impl_->anchor_offset_[1] = ignition::math::Rand::DblUniform(-
    anchor_offset_std, anchor_offset_std);
186       impl_->anchor_offset_[2] = ignition::math::Rand::DblUniform(-
    anchor_offset_std, anchor_offset_std);
187
188       gzdbg << "Adding anchor offset of: " << impl_->anchor_offset_
    [0] << ", " << impl_->anchor_offset_[1] << ", " << impl_->
    anchor_offset_[2] << std::endl;
189     }
190
191     // Setting up the publisher of the anchor pose
192     impl_->anchor_pub_ = impl_->gazebo_node_->Advertise<msgs::
    PoseStamped>(impl_->anchor_topic_, 1);
193
194     // This code is valid only if we want to publish ranging data
195     if (sdf->HasElement("pub_range"))
196     {
197       // Configure the ROS node from the SDF file
198       impl_->ros_node_ = gazebo_ros::Node::Get(sdf);
199
200       // Setting up anchors pose subscriber
201       impl_->anchor_sub_ = impl_->gazebo_node_->Subscribe(impl_->
    anchor_topic_, &RosPx4GazeboUwbPrivate::AnchorCallback, impl_.
    get());
202
203       // Setting up ranges publisher
204       impl_->sensor_pub_ = impl_->ros_node_->create_publisher<
    ros2_px4_interfaces::msg::UwbSensor>(
205           impl_->sensor_topic_ + impl_->anchor_id_,
206           impl_->ros_node_->get_qos().get_publisher_qos(impl_->
    sensor_topic_ + impl_->anchor_id_));
207
208       // <gaussian_noise> is the sigma value of gaussian noise to
    add to range readings
```

```
209       if (!sdf->HasElement("gaussian_noise"))
210       {
211         gzwarn << "Missing <gassian_noise>, defaults to 0.0" << std
    ::endl;
212         impl_->gaussian_noise_ = 0;
213       }
214       else
215       {
216         impl_->gaussian_noise_ = sdf->GetElement("gaussian_noise")
    ->Get<double>();
217       }
218     }
219
220     // Listen to the update event. This event is broadcast every
    simulation iteration
221     impl_->update_connection_ = event::Events::
    ConnectWorldUpdateBegin(
222         std::bind(&RosPx4GazeboUwbPrivate::OnUpdate, impl_.get(),
    std::placeholders::_1));
223   }
224
225   void RosPx4GazeboUwbPrivate::OnUpdate(const common::UpdateInfo &
    info)
226   {
227     // Check if link is destroyed
228     if (!link_)
229     {
230       return;
231     }
232
233     common::Time current_time = info.simTime;
234
235     if (current_time < last_time_)
236     {
237       gzwarn << "Negative update time difference detected" << std::
    endl;
238       last_time_ = current_time;
239     }
240
241     // Rate control
242     if (update_rate_ > 0 &&
243         (current_time - last_time_).Double() < (1.0 / update_rate_)
    )
244     {
245       return;
246     }
247
248     // Get world pose of the linked model
249     link_pose_ = link_->WorldPose();
250
```

```
251     // Get relative pose of the linked model ...
252     if (reference_model_)
253     {
254       // ... with the respect of the reference model ...
255       auto reference_pose = reference_model_->WorldPose();
256       if (reference_link_)
257       {
258         // or with the respect of the reference link
259         reference_pose = reference_link_->WorldPose();
260       }
261 
262       link_pose_.Pos() -= reference_pose.Pos();
263       link_pose_.Pos() = reference_pose.Rot().RotateVectorReverse(
    link_pose_.Pos());
264     }
265 
266     // Fill UWB message
267     msgs::PoseStamped *anchor_msg = new msgs::PoseStamped;
268     msgs::Time *anchor_msg_time = new msgs::Time;
269     msgs::Vector3d *anchor_msg_vec = new msgs::Vector3d;
270     msgs::Quaternion *anchor_msg_qua = new msgs::Quaternion;
271     msgs::Pose *anchor_msg_pose = new msgs::Pose;
272 
273     anchor_msg_time->set_sec(current_time.sec);
274     anchor_msg_time->set_nsec(current_time.nsec);
275     anchor_msg->set_allocated_time(anchor_msg_time);
276 
277     anchor_msg_vec->set_x(link_pose_.Pos().X());
278     anchor_msg_vec->set_y(link_pose_.Pos().Y());
279     anchor_msg_vec->set_z(link_pose_.Pos().Z());
280     anchor_msg_pose->set_allocated_position(anchor_msg_vec);
281 
282     // Sending orientation for a future more complex noise model
283     anchor_msg_qua->set_x(0.0);
284     anchor_msg_qua->set_y(0.0);
285     anchor_msg_qua->set_z(0.0);
286     anchor_msg_qua->set_w(0.0);
287     anchor_msg_pose->set_allocated_orientation(anchor_msg_qua);
288 
289     anchor_msg_pose->set_name(anchor_id_);
290     anchor_msg->set_allocated_pose(anchor_msg_pose);
291 
292     anchor_pub_->Publish(*anchor_msg);
293     last_time_ = current_time;
294   }
295 
296   void RosPx4GazeboUwbPrivate::AnchorCallback(ConstPoseStampedPtr &
    _msg)
297   {
298     double range;
```

```
299
300     // Check if the sensor and the anchor are 2 separate entities
301     if (_msg->pose().name() != anchor_id_)
302     {
303       // Calculate the range between sensor and anchor + gaussian
    noise
304       range = link_pose_.Pos().Distance(_msg->pose().position().x()
    , _msg->pose().position().y(), _msg->pose().position().z());
305
306       if (gaussian_noise_ > 0)
307         range += ignition::math::Rand::DblNormal(0, gaussian_noise_
    );
308
309       // Fill ROS range message
310       ros2_px4_interfaces::msg::UwbSensor sensor_msg;
311
312       sensor_msg.anchor_pose.header.stamp.sec = _msg->time().sec();
     // @todo: Simulate transmission delay anchor->sensor
313       sensor_msg.anchor_pose.header.stamp.nanosec = _msg->time().
    nsec();
314
315       sensor_msg.anchor_pose.header.frame_id = _msg->pose().name();
316
317       sensor_msg.anchor_pose.pose.position.x = _msg->pose().
    position().x() + anchor_offset_[0];
318       sensor_msg.anchor_pose.pose.position.y = _msg->pose().
    position().y() + anchor_offset_[1];
319       sensor_msg.anchor_pose.pose.position.z = _msg->pose().
    position().z() + anchor_offset_[2];
320
321       sensor_msg.anchor_pose.pose.orientation.x = _msg->pose().
    orientation().x();
322       sensor_msg.anchor_pose.pose.orientation.y = _msg->pose().
    orientation().y();
323       sensor_msg.anchor_pose.pose.orientation.z = _msg->pose().
    orientation().z();
324       sensor_msg.anchor_pose.pose.orientation.w = _msg->pose().
    orientation().w();
325
326       sensor_msg.range = range;
327
328       // Publish to ROS
329       sensor_pub_->publish(sensor_msg);
330     }
331   }
332
333   GZ_REGISTER_MODEL_PLUGIN(RosPx4GazeboUwb)
334
335 } // namespace gazebo
```

A.2 UKF ROS2 node

A.2.1 ukf_positioning.py

```python
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node

import numpy as np
import scipy

from filterpy.kalman import MerweScaledSigmaPoints
from filterpy.kalman import UnscentedKalmanFilter as UKF
from filterpy.common import Q_discrete_white_noise

from nav_msgs.msg import Odometry
from ros2_px4_interfaces.msg import UwbSensor
from px4_msgs.msg import DistanceSensor


QUEUE_SIZE = 10
FILTER_DIM = 9                    # Linear kinematic model
IS_SENSOR_ALIVE_TIMEOUT = 1.      # s
MAHALANOBIS_THRESHOLD = 6.        # sigmas
MAX_ALLOWED_UWB_RANGE = 10.       # meters


class UkfPositioning(Node):
    """UKF Positioning node"""

    def __init__(self):
        super().__init__("UkfPositioning")

        self.declare_parameters("", [
            ("delta_t", 0.02),
            ("q", 0.1),
            ("r_uwb", 0.0025),
            ("r_laser", 0.01)
        ])

        self.params_ = {
            x.name: x.value for x in self.get_parameters(
                ["delta_t", "q", "r_uwb", "r_laser"]
            )
        }

        self.filter_state_ = "Offline"
        self.calibration_counter_ = 0
        self.aligning_counter_ = 0
```

```
47          self.anchor_offset_ = np.zeros(3)
48          self.last_sensor_list_ = []
49          self.sensor_wd_ = {
50              "uwb": 0,
51              "gps": 0,
52              "laser": 0
53          }
54
55          # Sigma points generator
56          sigmas = MerweScaledSigmaPoints(
57              FILTER_DIM, alpha=1e-3, beta=2., kappa=0.,
58              sqrt_method=scipy.linalg.sqrtm
59          )
60
61          def f_(x, dt):
62              f = np.array([
63                  [1., dt, 0.5*dt**2.],
64                  [0., 1.,         dt],
65                  [0., 0.,         1.]
66              ])
67              F = scipy.linalg.block_diag(*[f]*3)
68              return F @ x
69
70          # UKF
71          self.kalman_filter_ = UKF(
72              dim_x=FILTER_DIM,
73              dim_z=FILTER_DIM,
74              dt=self.params_["delta_t"],
75              fx=f_, hx=None,
76              points=sigmas
77          )
78
79          # Initial estimate
80          self.kalman_filter_.x *= 0.
81
82          # Covariance matrix
83          self.kalman_filter_.P *= 1.
84
85          # Process noise
86          self.kalman_filter_.Q = scipy.linalg.block_diag(
87              Q_discrete_white_noise(
88                  dim=3, dt=self.params_["delta_t"], var=self.params_
    ["q"], block_size=3)
89          )
90
91          # Setting up sensors subscribers
92          self.uwb_pos_subscriber_ = self.create_subscription(
93              Odometry, "UwbPositioning/Odometry",
94              self.callback_uwb_pos_subscriber, QUEUE_SIZE
95          )
```

```
96          self.uwb_subscriber_ = self.create_subscription(
97              UwbSensor, "/uwb_sensor_tag_0",
98              self.callback_uwb_subscriber, QUEUE_SIZE
99          )
100         self.gps_subscriber_ = self.create_subscription(
101             Odometry, "GpsPositioning/Odometry",
102             self.callback_gps_subscriber, QUEUE_SIZE
103         )
104         self.laser_subscriber_ = self.create_subscription(
105             DistanceSensor, "DistanceSensor_PubSubTopic",
106             self.callback_laser_subscriber, QUEUE_SIZE
107         )
108
109         # Setting up position and velocity publisher
110         self.odometry_publisher_ = self.create_publisher(
111             Odometry, "~/Odometry", QUEUE_SIZE
112         )
113
114         # Prediction timer
115         self.timer = self.create_timer(
116             self.params_["delta_t"], self.predict_callback)
117
118         self.get_logger().info(f"Node has started: {self.params_}")
119
120     def callback_uwb_pos_subscriber(self, msg):
121         """Measuring UWB position offset estimate wtr of the
    navigation frame
122
123         Args:
124             msg (nav_msgs.msg.Odometry): The UWB odometry
125         """
126
127         # Deriving the anchor offset
128         anchor_offset = np.array([
129             msg.pose.pose.position.x,
130             msg.pose.pose.position.y,
131             msg.pose.pose.position.z
132         ]) - self.kalman_filter_.x[[0, 3, 6]]
133
134         # Mean values until convergence, than this subscriber can
    be destroyed
135         var = (anchor_offset - self.anchor_offset_) / \
136             (self.aligning_counter_ + 1)
137
138         if np.linalg.norm(var) > 1e-3 or self.aligning_counter_ <
    100.:
139             self.anchor_offset_ += var
140             self.aligning_counter_ += 1
141         else:
142             self.get_logger().info(f"""
```

```
143             Anchor reference frame aligned in {self.
    aligning_counter_} iterations:
144             Offset = {self.anchor_offset_}
145             """)
146             self.destroy_subscription(self.uwb_pos_subscriber_)
147             self.aligning_counter_ = -1
148 
149     def callback_uwb_subscriber(self, msg):
150         """Measuring UWB range sensor
151 
152         Args:
153             msg (ros2_px4_interfaces.msg.UwbSensor): The UWB
    message
154         """
155 
156         # Must predict once first and anchor must be aligned
157         if(self.filter_state_ == "Offline" or self.
    aligning_counter_ > -1):
158             return
159 
160         # Storing measurement in a np.array
161         z = np.zeros(FILTER_DIM)
162         z[0] = msg.range
163 
164         if z[0] > MAX_ALLOWED_UWB_RANGE:
165             return
166 
167         if any(np.isnan(z)):
168             self.get_logger().error(f"Invalid UWB data")
169             return
170 
171         # Storing timestamp
172         self.sensor_wd_["uwb"] = self.get_clock().now().nanoseconds
173 
174         # Storing anchor position in a np.array
175         anchor_position = np.array([
176             msg.anchor_pose.pose.position.x,
177             msg.anchor_pose.pose.position.y,
178             msg.anchor_pose.pose.position.z
179         ])
180 
181         # Measurement model for a rotated range sensor
182         def h_uwb(x):
183             h = np.zeros(FILTER_DIM)
184 
185             # Range measurements
186             h[0] = np.linalg.norm(
187                 x[[0, 3, 6]] - (anchor_position - self.
    anchor_offset_))
188             return h
```

```
189
190         # Filter update
191         x = self.kalman_filter_.x.copy()
192         P = self.kalman_filter_.P.copy()
193         self.kalman_filter_.update(z, R=self.params_["r_uwb"], hx=
    h_uwb)
194
195         # Gating
196         if self.kalman_filter_.mahalanobis > MAHALANOBIS_THRESHOLD
    and self.filter_state_ == "Calibrated":
197             self.get_logger().warn(
198                 f"Gating anchor {msg.anchor_pose.header.frame_id} @
    : {self.kalman_filter_.mahalanobis}")
199             self.kalman_filter_.x = x.copy()
200             self.kalman_filter_.P = P.copy()
201
202     def callback_gps_subscriber(self, msg):
203         """Measuring GPS sensor
204
205         Args:
206             msg (nav_msgs.msg.Odometry): The GPS message
207         """
208
209         # Must predict once first
210         if(self.filter_state_ == "Offline"):
211             return
212
213         # Storing measurements in a np.array
214         z = np.zeros(FILTER_DIM)
215         z[0:6] = np.array([
216             msg.pose.pose.position.x,
217             msg.pose.pose.position.y,
218             msg.pose.pose.position.z,
219             msg.twist.twist.linear.x,
220             msg.twist.twist.linear.y,
221             msg.twist.twist.linear.z,
222         ])
223
224         if any(np.isnan(z)):
225             self.get_logger().error(f"Invalid GPS data")
226             return
227
228         # Storing timestamp
229         self.sensor_wd_["gps"] = self.get_clock().now().nanoseconds
230
231         # Measurement model for a GPS sensor
232         def h_gps(x):
233             h = np.zeros(FILTER_DIM)
234             h[0:6] = np.array([x[0], x[3], x[6], x[1], x[4], x[7]])
235             return h
```

```
236
237         # Filter update
238         R = np.eye(FILTER_DIM) * np.array([
239             msg.pose.covariance[0],
240             msg.pose.covariance[7],
241             msg.pose.covariance[15],
242             msg.twist.covariance[0],
243             msg.twist.covariance[7],
244             msg.twist.covariance[15], 1., 1., 1., 1.
245         ])
246
247         self.kalman_filter_.update(z, R, hx=h_gps)
248
249     def callback_laser_subscriber(self, msg):
250         """Measuring laser sensor
251
252         Args:
253             msg (px4_msgs.msg.DistanceSensor): The Laser message
254         """
255
256         # Must predict once first
257         if(self.filter_state_ == "Offline"):
258             return
259
260         # Storing timestamp
261         self.sensor_wd_["laser"] = self.get_clock().now().
    nanoseconds
262
263         # Storing measurements in a np.array
264         z = np.zeros(FILTER_DIM)
265         z[0] = msg.current_distance
266
267         if any(np.isnan(z)):
268             self.get_logger().error(f"Invalid laser data")
269             return
270
271         # Measurement model for laser sensor
272         def h_laser(x):
273             h = np.zeros(FILTER_DIM)
274             h[0] = x[6]
275             return h
276
277         # Filter update
278         x = self.kalman_filter_.x.copy()
279         P = self.kalman_filter_.P.copy()
280         self.kalman_filter_.update(z, R=self.params_["r_laser"], hx
    =h_laser)
281
282         # Gating
```

```
283         if self.kalman_filter_.mahalanobis > MAHALANOBIS_THRESHOLD
    and self.filter_state_ == "Calibrated":
284             self.get_logger().warn(
285                 f"Gating laser @: {self.kalman_filter_.mahalanobis}
    ")
286             self.kalman_filter_.x = x.copy()
287             self.kalman_filter_.P = P.copy()
288 
289     def predict_callback(self):
290         """This callback perform the filter predict and forward the
     current
291         estimate
292         """
293 
294         if (self.filter_state_ == "Calibrating"):
295             if(np.linalg.norm(self.kalman_filter_.P) < 10.):
296                 self.calibration_counter_ += 1
297 
298                 if (self.calibration_counter_ > 100):
299                     self.get_logger().info("Filter calibrated")
300                     self.filter_state_ = "Calibrated"
301                     self.calibration_counter_ = 0
302             else:
303                 self.calibration_counter_ = 0
304 
305         # Check which sensors are working
306         active_sensor_list = []
307         for sensor in self.sensor_wd_:
308             if self.get_clock().now().nanoseconds - self.sensor_wd_
    [sensor] < IS_SENSOR_ALIVE_TIMEOUT*1e9:
309                 active_sensor_list.append(sensor)
310 
311         # Send estimation only if calibrated
312         if(self.filter_state_ == "Calibrated"):
313             # Check covariance norm
314             if(np.linalg.norm(self.kalman_filter_.P) > 100.):
315                 self.filter_state_ = "Diverged"
316                 self.get_logger().error("Filter is diverging")
317                 return
318 
319             # Sending the estimated odometry
320             msg = Odometry()
321             msg.header.frame_id = "UkfPositioning"
322             msg.header.stamp = self.get_clock().now().to_msg()
323 
324             msg.pose.pose.orientation.x = (1. if ("gps" in
    active_sensor_list)
325                                            else 0.) + (2. if ("uwb"
     in active_sensor_list) else 0.)
326 
```

```
327             # Position
328             msg.pose.pose.position.x = self.kalman_filter_.x[0]
329             msg.pose.pose.position.y = self.kalman_filter_.x[3]
330             msg.pose.pose.position.z = self.kalman_filter_.x[6]
331 
332             msg.pose.covariance[0] = self.kalman_filter_.P[0][0]
333             msg.pose.covariance[1] = self.kalman_filter_.P[3][3]
334             msg.pose.covariance[2] = self.kalman_filter_.P[6][6]
335 
336             # Velocity
337             msg.twist.twist.linear.x = self.kalman_filter_.x[1]
338             msg.twist.twist.linear.y = self.kalman_filter_.x[4]
339             msg.twist.twist.linear.z = self.kalman_filter_.x[7]
340 
341             msg.twist.covariance[0] = self.kalman_filter_.P[1][1]
342             msg.twist.covariance[1] = self.kalman_filter_.P[4][4]
343             msg.twist.covariance[2] = self.kalman_filter_.P[7][7]
344 
345             self.odometry_publisher_.publish(msg)
346 
347         # Filter predict only if first iteration or new sensor data
348         if active_sensor_list != [] or self.filter_state_ == "
    Offline":
349             self.kalman_filter_.predict()
350 
351         # Log on sensor list change
352         if (active_sensor_list != self.last_sensor_list_):
353             self.get_logger().info(f"Fusion using: {
    active_sensor_list}")
354             self.last_sensor_list_ = active_sensor_list
355 
356             if (active_sensor_list == [] and self.filter_state_ ==
    "Calibrated"):
357                 self.get_logger().error(f"Data fusion is not
    possible without data ^_^")
358                 self.filter_state_ == "Offline"
359 
360         # First data just arrived
361         if(self.filter_state_ == "Initialized" and
    active_sensor_list != []):
362             self.filter_state_ = "Calibrating"
363             self.get_logger().info("Filter is calibrating")
364 
365         # First predict just happened
366         if(self.filter_state_ == "Offline"):
367             self.filter_state_ = "Initialized"
368             self.get_logger().info("Filter initialized")
369 
370 
371 def main(args=None):
```

```
372     rclpy.init(args=args)
373     node = UkfPositioning()
374     rclpy.spin(node)
375     rclpy.shutdown()
376 
377 
378 if __name__ == "main":
379     main()
```

Acronyms

3D Three Dimensional

DOP Dilution of precision

EKF Extended Kalman filter

ESC Electronic Speed Control

FCC Federal Communication Commission

GLONASS Global'naya Navigatsionnaya Sputnikovaya Sistema

GNSS Global Navigation Satellite System

GPS Global Positioning System

IC Integrated Circuit

IMU Inertial Measurement Unit

MSE Mean Square Error

NLOS Not Line Of Sight

PPM Pulse Position Modulation

RC Radio Controller

RF Radio Frequency

ROS2 Robot Operating System

SITL Software In The Loop

TM Time Modulation

TOA Time Of Arrival

TDOA Time difference of arrival

UAV Unmanned Aerial Vehicle

UKF Unscented Kalman Filter

USB Universal Serial Bus

UT Unscented Transformation

UWB Ultra WideBand

VTOL Vertical Take-Off and Landing

Bibliography

[1] Jaemin Baek and Jinmyung Jung. «A Model-Free Control Scheme for Attitude Stabilization of Quadrotor Systems». eng. In: *Electronics (Basel)* 9.10 (2020), p. 1. ISSN: 2079-9292 (cit. on p. 12).

[2] Federal Communications Commission. *Revision of Part 15 of the Commission's Rules Regarding Ultra-Wideband Transmission Systems*. 2002. URL: https://transition.fcc.gov/Bureaus/Engineering_Technology/Orders/2002/fcc02048.pdf (cit. on p. 23).

[3] DecaWave, ed. *EVK1000 Evaluation Kit*. URL: https://www.decawave.com/product/evk1000-evaluation-kit/ (cit. on p. 49).

[4] The Dronecode Foundation, ed. *PX4 Autopilot User Guide*. 2021. URL: https://docs.px4.io (cit. on pp. 14, 15).

[5] Paul D Groves. *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*. eng. Norwood: Artech House, 2013. ISBN: 1608070050 (cit. on pp. 13, 18, 21, 22).

[6] Gabriel Hoffmann, ed. *Schematic of reaction torques on each motor of a quadrotor aircraft, due to spinning rotors*. 2007. URL: https://commons.wikimedia.org/wiki/File:Quadrotor_yaw_torque.png (cit. on p. 11).

[7] Holybro, ed. *X500 Kit*. URL: https://shop.holybro.com/x500-kit_p1180.html (cit. on p. 48).

[8] Jari Iinatti Ian Oppermann Matti Hämäläinen. *UWB: Theory and Applications*. eng. Chichester: John Wiley & Sons, Incorporated, 2004. ISBN: 0-470-86917-8 (cit. on p. 23).

[9] Roger R. Labbe Jr, ed. *FilterPy - Kalman filters and other optimal and non-optimal estimation filters in Python*. 2015. URL: https://github.com/rlabbe/filterpy (cit. on pp. 29, 33).

[10] R. E Kalman. «A New Approach to Linear Filtering and Prediction Problems». eng. In: *Journal of fluids engineering* 82.1 (1960), pp. 35–45. ISSN: 0098-2202 (cit. on p. 25).

[11] Elliott D Kaplan and Christopher Hegarty. *Understanding GPS/GNSS: Principles and Applications, Third Edition*. eng. Third Edition. Place of publication not identified: Artech House, 2017. ISBN: 1630810584 (cit. on p. 19).

[12] Teppo Luukkonen. «Modelling and control of quadcopter». In: *Independent research project in applied mathematics, Espoo* 22 (2011), p. 22 (cit. on p. 12).

[13] Rudolph Merwe and Eric Wan. «Sigma-Point Kalman Filters for Probabilistic Inference in Dynamic State-Space Models». In: *Proceedings of the Workshop on Advances in Machine Learning* (June 2003) (cit. on p. 31).

[14] Navigation National Coordination Office for Space-Based Positioning and Timing, eds. *Official U.S. government information about the Global Positioning System (GPS) and related topics*. 2021. URL: https://www.gps.gov/systems/gnss (cit. on p. 17).

[15] Clement A Ogaja. *Applied GPS for Engineers and Project Managers*. eng. Reston, VA: American Society of Civil Engineers, 2011. ISBN: 9780784411506 (cit. on p. 18).

[16] Oxford University Press, ed. *Definition of UAV noun from the Oxford Dictionary*. URL: https://www.oxfordlearnersdictionaries.com/definition/english/uav (cit. on p. 11).

[17] Greg Welch, Gary Bishop, et al. «An introduction to the Kalman filter». In: (1995) (cit. on p. 25).