الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي

Research scientific and Education
Higher of Ministry
المدرسة الوطنية العليا للتكنولوجيا
National Higher School of Technology

Department of Electrical Engineering And Industrial Computing

---

Final Year Project to Obtain the Diploma of
Engineering

Field: Electronics

Speciality: EmbeddedSystems

- Subject -

Virtual Realization of a Quadcopter - Manipulator Under ROS/Gazebo Environment

by

IDER Abdelaziz

Examination Committee:

Mme BOUCHAMA Samira	Chair
M. REZOUG Amar	Supervisor
M.BAIZID Khelifa	Supervisor
Mme REBAI Karima	Examiner
M. KHELOUAT Samir	Examiner

Algiers, the 14/09/2023

Academic year 2022 –2023

Dedication

“

I dedicate this work to my very dear parents, whose sacrifice, tenderness, love, patience, support, help, and encouragement are essential to my success. Without them, I would not be at this stage today.

To my brothers and sisters for their continued support throughout my journey.

To my big family. And to all my friends

”

IDER Abdelaziz

Acknowledgements

We thank Almighty God for giving us the strength and courage to carry out this work.

This work was done under the supervision of Mr. Rezoug Amar, a professor at the National Higher School of Technology ENST in Dergana, and Mr. Khelifa Baizid, a professor at the Italian Institute of Technology IIT in Italy, who closely followed this work for their educational orientation in the creation of this memory.

We would like to express our gratitude to Mme BOUCHAMA Samira, a professor at the National Higher School of Technology, for the honor of chairing the jury for our defense.

M. KHELOUAT Samir and Mme REBAI Karima, professors at the National Higher School of Technology, be assured of our heartfelt gratitude for agreeing to review and critique this work.

I would also like to thank my supervisors, Mr. Rezoug Amar and Mr. Khelifa, for guiding me through the completion of this project.

We'd like to thank everyone who helped us put this memory together, especially Taibaoui Wassel, Hamouche Walid, Hariri Smail and his family ,Saou Mohamed Lamine, and Yous Hichem.

Finally, we want to thank everyone who has contributed to our success during our academic journey.

Abstract

This project describes the work done to develop a test platform for control algorithms in quadcopters with robotic arm attached to it (Unmanned Aerial Vehicle Manipulator System). The quadcopter chosen was a custom drone modeled and inspired from DJI F450 Drone using Solidworks, therefore, all the model parameters used. In order to implement a low-level controller, an available architecture drone package was reformulated in order to assign the forces applied directly to the drone instead of the motor velocities, due to simulation conditions. Moreover, a controller was implemented doing the time synchronization between ROS and Gazebo. The algorithm chosen was a Cascaded PID due to its simplicity compared to other robust control algorithms, The controller, attached to the model, shows a very good result being able to perform a stable hovering at given position and altitude and moving the robot arm to desired configuration.

Keywords : ROS ,Gazebo , Quadcopter, PID,Controller ,SolidWorks.

Résumé

Cette mémoire décrit les travaux réalisés pour développer une plateforme de test d'algorithmes de contrôle dans des quadricoptères auxquels est attaché un bras robotique (Unmanned Aerial Vehicle Manipulator Système). Le quadricoptère choisi était un drone personnalisé modélisé et inspiré du DJI F450 Drone utilisant Solidworks, donc tous les paramètres du modèle utilisés. Afin de mettre en œuvre un contrôleur de bas niveau, une architecture de drone disponible a été reformulée afin d'attribuer les forces appliquées directement au drone au lieu des vitesses du moteur, en raison des conditions de simulation. De plus, un contrôleur a été implémenté pour effectuer la synchronisation du temps entre ROS et Gazebo. L'algorithme choisi était un PID en cascade en raison de sa simplicité par rapport à d'autres algorithmes de contrôle robustes. Le contrôleur, attaché au modèle, montre un très bon résultat en étant capable d'effectuer un vol stationnaire stable à une position et une altitude données et de déplacer le bras du robot vers la position souhaitée. configuration.

Mots clés : ROS, Gazebo, Quadcopter, PID, Contrôleur, SolidWorks.

ملخص

تصف هذه الأطروحة العمل المنجز لتطوير منصة اختبار لخوارزميات التحكم في المروحيات الرباعية المزودة بذراع آلية متصلة بها (المركبات الجوية بدون طيارالمزودة بذراع الي). الطائرة الرباعية التي تم اختيارها عبارة عن طائرة بدون طيار مخصصة ومستوحاة من طائرة Drone F450 DJI تم تصميمها باستخدام ،Solidworks وبالتالي، تم استخدام جميع معلومات النموذج. من أجل تنفيذ وحدة تحكم منخفضة المستوى، تمت إعادة صياغة حزمة البرمجيات للطائرات بدون طيار المتاحة من أجل التحكم في القوى المطبقة مباشرة على الطائرة بدون طيار بدلاً من سرعات المحرك، وذلك بسبب ظروف المحاكاة. علاوة على ذلك، تم تنفيذ وحدة تحكم للقيام بمزامنة الوقت بين ROS و .Gazebo الخوارزمية التي تم اختيارها عبارة عن PID متتالي نظرًا لبساطتها مقارنة بخوارزميات التحكم القوية الأخرى، وتُظهر وحدة التحكم المرفقة بالنموذج نتيجة جيدة جدًا حيث تكون الطائرة قادرة على أن تحوم ارتفاع ثابت والقيام بتحريك ذراع الروبوت إلى المستوى المطلوب.

كلمات مفتاحية :

ROS ، جازيبو ، كوادكوبتر ، PID ، وحدة تحكم ، SolidWorks .

Contents

List of Figures / Tables / abbreviations and acronyms IX

General Introduction 1

1 Background 3
1.1 Unmanned Aerial Vehicles History 4
1.2 State of the Art of UAV and UAM 5
1.2.1 Manipulative robots 5
1.2.2 Unmanned aerial vehicle 6
1.2.3 Unmanned Aerial Manipulator 8
1.3 Control Strategies for Quadrotors 11
1.3.1 Control System 11
1.3.2 PID Control 12

2 Robotics Operational Systems (ROS) and Setup 13
2.1 Robot Operating system 14
2.1.1 Definitions 14
2.1.2 Nodes 15
2.1.3 Topics 15
2.1.4 Messages 15
2.1.5 Services 15
2.1.6 Bags 15
2.1.7 Packages 15
2.2 Setup 16
2.2.1 Ros Noetic installation 16
2.3 Other Packages : Moveit , Rviz , RQT , Plotjuggler 17
2.3.1 Moveit 17
2.3.2 Rviz 18
2.3.3 RQT Console 19
2.3.4 Plotjuggler 19
2.4 ROS Distro: Noetic 20

3 Quadcopter-Manipulator Virtual Design 23
3.1 SolidWorks Modeling Process 24
3.1.1 Drone Assembly and Components 24
3.1.2 Propellers 26
3.1.3 A2212 Brushless Motor 26
3.1.4 Drone Base Arms (4 Parts) 28

3.1.5 Base Top and Base Bottom . . . 29
3.1.6 Landing Stand (Top and Bottom Parts) . . . 31
3.1.7 Robotic Arm Components (Part 1 to Part 5) . . . 33
3.1.8 End Effector (Part 6) . . . 38
3.1.9 System Configuration: . . . 41

4 SolidWorks Design to ROS Package . . . 44
4.1 SWtoURDF Exporter Plugin . . . 45
4.1.1 Ros package architecture . . . 46
4.2 Prepare the package . . . 46
4.2.1 Setting Up a Catkin Workspace . . . 46
4.2.2 Organize our Package . . . 47
4.2.3 Apply necessary modification to achieve our needs . . . 53
4.2.4 URDF File Result: full robot description . . . 57
4.2.5 Creating a new Manipulator Package to control the Robotic Arm URDF using Moveit . . . 60
4.3 ROS Package Validation: RViz and Gazebo . . . 63
4.3.1 Visulize the Model in both Rviz and Gazebo . . . 64
4.3.2 Conclusion . . . 65

5 Control of the Quadcopter-Manipulator in ROS/Gazebo Environment 66
5.1 Gazebo Simulator . . . 67
5.1.1 Sensors provided by Gazebo . . . 69
5.1.2 Computational Power . . . 71
5.2 Control Strategy . . . 73
5.2.1 Control Scheme: Position Controller . . . 74
5.2.2 Control Scheme: Attitude Controller . . . 74
5.2.3 Control Algorithm using Python scripts . . . 75
5.2.4 Control Parameters for Cascaded PID Position and Attitude Control 79
5.2.5 RQT Graph . . . 80
5.2.6 Tuning Methodology . . . 82

6 Solar Panels field inspection using Quadcopter-Manipulator . . . 84
6.1 Computational power . . . 85
6.2 System Usage Guide . . . 85
6.2.1 Launching the simulation . . . 85
6.2.2 Testing our drone: sensors, commands 87
6.2.3 Sending commands to test the motors . . . 89
6.2.4 Testing the created Moveit Robot Arm poses . . . 89
6.2.5 Make a simple hovering test at fixed altitude 3m . . . 89
6.3 Robotic Tasks . . . 91
6.3.1 Creating a 3d world for simulation . . . 91
6.3.2 Solar Panel: Trajectory Tracking and Inspection . . . 91

Conclusion et perspectives . . . 95

Bibliography . . . 98

List of Figures

1.1 Ryan 147, AQM-34 "Lightning Bug" and the family of Teledyne Ryan Aeronautical. . . . 5
1.2 The configurations of a manipulator robot. . . . 6
1.3 Mugin-5 Pro 5000mm Carbon Fiber UAV Platform – Mugin UAV. . . . 7
1.4 Superfox6 Rotary-wing drone by Hélicéo Company. . . . 8
1.5 An Example of a Drone equipped with 2 robotic arm by ProDrone Company. 8
1.6 Drone with Cables. . . . 9
1.7 Robotic Gripper . . . 10
1.8 jaco by Robotics Kinova. . . . 10
1.9 Closed Loop Control. . . . 11

2.1 Examples of Operating Systems: Windows, macOS, Android, Linux. . . . 14
2.2 ROS Communication Using Nodes Under a Topic. . . . 16
2.3 MoveIt by PickNik Robotics. . . . 17
2.4 Image shows Rviz menu. . . . 18
2.5 Image inside Rqt Console: image viewer on the left and topic monitor on the right. . . . 19
2.6 Image inside Plotjuggler . . . 20
2.7 Some Ros 1 distributions. . . . 21

3.1 Final Assembly 3d view. . . . 25
3.2 Landing gear opened view. . . . 25
3.3 propeller 3d view. . . . 26
3.4 A2212 Brushless motor 3d view. . . . 27
3.5 Motors Physical Details. . . . 27
3.6 Base Arm 3d view. . . . 28
3.7 Another Base arm 3d view. . . . 29
3.8 Base bottom sketch. . . . 29
3.9 Base Bottom 3d view. . . . 30
3.10 Base Top sketch. . . . 30
3.11 Base Top 3d view. . . . 31
3.12 Landing bottom sketch. . . . 31
3.13 Landing bottom 3d view. . . . 32
3.14 Landing Top sketch. . . . 32
3.15 Landing Top 3d view. . . . 32
3.16 Robotic Arm Structure 3d view. . . . 33
3.17 Base Link Sketch. . . . 33
3.18 Base Link 3d view. . . . 34

3.19 Another Base Link 3d view. . . . 34
3.20 Link 1 Sketch. . . . 35
3.21 Link 1 3d view. . . . 35
3.22 Link 2 Sketch. . . . 36
3.23 Link 2 3d view. . . . 36
3.24 Link 3 Sketch. . . . 36
3.25 Link 3 3d view. . . . 37
3.26 Link 4 Sketch. . . . 37
3.27 Link 4 3d view. . . . 38
3.28 Another Link 4 3d view. . . . 38
3.29 Camera 3d view. . . . 39
3.30 Magnetic Gripper 3d view. . . . 39
3.31 Gripper 3d view. . . . 40
3.32 Gripper sketch. . . . 40
3.33 X Configuration. . . . 41
3.34 Forces acting on a quadcopter. . . . 41
3.35 Joints Configuration of the Robotic Arm. . . . 43

4.1 Urdf Exporter Configuration. . . . 45
4.2 Our package directory. . . . 46
4.3 How URDF has been processed by ROS. . . . 49
4.4 Custom messages used in our system 56
4.5 moveit setup assistant menu 61
4.6 custom robot poses 'pick pose etc ...' 62
4.7 joints groups 'arm group etc ...' 62
4.8 kdl kinematic solver plugin. . . . 63
4.9 The view of the system in Gazebo. . . . 64
4.10 Rviz views 64

5.1 Gazebo Simulator. . . . 67
5.2 Gazebo Virtual Environment. . . . 68
5.3 Presents a Gazebo Simulation of LIDAR sensor attached to a mobile robot. 69
5.4 Cascaded control system 73
5.5 Position Controller 74
5.6 Attitude Controller 75
5.7 Rqt graph of the system 81

6.1 Spawning the Drone 86
6.2 Rviz motion planner 86
6.3 Rqt topic Monitor 87
6.4 Gps Data 87
6.5 IMU Data 87
6.6 image from simulated world 88
6.7 Position Controller outputs 88
6.8 Attitude Controller Outputs 89
6.9 Motors spinning at 5 rpm 89
6.10 Poses Examples 90

6.11 Hovering test 90
6.12 Some worlds used in simulation 91
6.13 Following the Given Setpoints 92
6.14 Camera View 93
6.15 Drone hovering and inspecting a solar panel 93
6.16 Simulation results 94

List of Tables

3.2 System Physical Parameters 25
3.4 Propeller Physical Parameters 26
3.6 Brush less Motor Physical Parameters 26
3.8 Base Arm Physical Parameters 28
3.10 Base Physical Parameters 31
3.12 Landing Gear Physical Parameters 32

5.1 Control Parameters for Cascaded PID Position and Attitude Control . . . 79

List of abbreviations and acronyms

AI	*Artificial Intelligence*
DOF	*Degrees of Freedom*
EEF	*End Effector*
IMU	*Inertial Measurement Unit*
ISR	*Intelligence , Surveillance and Reconnaissance*
LIDAR	*Light Detection and Ranging*
MMX	*Motor Mixing Algorithm*
UAM	*Unmanned Aerial Manipulator*
UAV	*Unmanned Aerial Vehicle*
URDF	*Unified Robotics Description Format*
ROS	*Robot Operating System*
RPY	*Roll Pitch Yaw*
RVIZ	*ROS Visualization*
PID	*Proportional Integral Derivative*
PWM	*Pulse Width Modulation*
XML	*Extensible Markup Language*
YAML	*Human-readable Data Serialization Language*

General Introduction

Unmanned Aerial Vehicles (UAVs) have emerged as a focal point of extensive research worldwide and are projected to remain at the forefront of study for years to come. This field harbors substantial potential to enhance the quality of human life by leveraging technology to shorten distances and travel times. However, the deployment of UAVs for both cargo transportation and search-and-rescue missions entails inherent risks. Due to their aerial nature, these vehicles have the potential to cause catastrophic disasters. Controlling them is inherently more challenging, and the consequences of a failure are significantly more perilous than those of ground vehicles.

In light of these considerations, the primary objective of this work is to establish a secure testing environment facilitating the evaluation of control algorithms through simulation tools. This work constitutes a comprehensive engineering project that encompasses the entire spectrum, from model selection to algorithm adaptation and result evaluation. The principal goals are as follows:

Develop a platform for testing control algorithms in quadcopters. Design an architecture capable of directly assigning motor velocities (low-level control). Enable the definition of flight trajectories for quadcopters in space. Execute practical tasks within the simulation environment. Unmanned aerial vehicles have garnered considerable attention, particularly following the proliferation of commercial drones, owing to their rapid problem-solving capabilities. They have revolutionized various domains, simplifying tasks such as monitoring deforestation, combatting fires in remote and inaccessible locations, and mapping terrain with unprecedented ease. Undoubtedly, this technology has reshaped our world.

Throughout the progression of this project, a multitude of challenges surfaced. These encompassed the necessity of employing a stable pre-existing model as a foundational study base due to the immense complexity of developing an entire simulator architecture. Issues arising from operating system versions and package compatibility had to be addressed. Additionally, two other formidable challenges emerged: the unsuitability of the initial package for implementing low-level control and the intricate task of synchronizing time between the Gazebo simulator and the controller. These issues were resolved through a series of strategic measures, including downgrading the Linux Ubuntu version, code adaptation, and optimizing the computational resources of the utilized laptop.

This report offers a broad overview of the encountered challenges and the corresponding solutions during the developmental phases of this project. In-depth explanations of each issue and its resolution will be presented in forthcoming chapters.

The essence of this study pertains to the accessibility of information essential for con-

trol assessment. Accordingly, the study's critical components encompass the number of sensors, aerodynamic factors, and physical parameters essential for constructing a valid model. The UAV integrates an Inertial Measurement Unit (IMU) housing a 3-axis accelerometer, 3-axis gyroscope, and a 3-axis magnetometer. To ascertain position during flight, the UAV relies on the Global Positioning System (GPS). Consequently, leveraging these sensor resources, we can compute the velocity, acceleration in three dimensions (X, Y, Z), and the current flight altitude of the drone.

Following the successful execution of this work, we have overhauled the simulator's architecture to meet our project's specific requirements. This modification has empowered the direct assignment of motor velocities and the effective implementation of a cascade PID controller. This controller has not only stabilized the UAV in flight but also facilitated the tracking of predefined trajectories.

Moreover, the potential for future expansion of this project remains promising. The ROS package employed in this project was originally designed for real-world applications with the DJI F450 Drone. Consequently, all controllers devised and tested within the simulator have the potential for seamless integration with real drones.

The structure of this work unfolds as follows: **Chapter 1** delves into the background of UAVs and their control techniques. **Chapter 2** elucidates the software foundations of the drone and provides crucial system instructions and definitions. **Chapter 3** chronicles the Design of the Quadcopter-Manipulator, commencing from its foundational elements to the addition of a robotic arm. **Chapter 4** shows the tools and frameworks instrumental in the implementation of controllers and simulations.**Chapter 5** highlights the simulations performed by the Quadcopter Manipulator including solar panels inspection, the concluding chapter, along with avenues for future work.

Chapter 1

Background

Introduction

Unmanned Aerial Vehicles, often known as UAVs, are more widely used nowadays than at any other time in mankind's history. The importance of this type of vehicle grews over time as its use became more popular. The ability to avoid putting human life in high-risk circumstances was a fantastic feature that UAVs provided us. Many of these activities were handled by autonomous aircraft systems in the latter decades of human history.

Given the increasing relevance of UAVs in our lives, this chapter is devoted to outlining the history and many varieties of these vehicles, particularly the quadrotor, which is employed in this study. Finally, the project aims and motives are presented.

1.1 Unmanned Aerial Vehicles History

Since its very beginning, UAV have tended to be driven by military applications. As said before, the possibility to perform an attack with no risks to the soldiers of an army grews a huge interest on these type of vehicles. Clearly it is undeniable that the military area had a great impact on the technological field over human history.

One could say that the very first UAV was a stone thrown by a caveman in man history early times, others, a Chinese rocket launched in the thirteenth century. That is why it is very important to have a clear definition for these kind of vehicles. If we define UAV as vehicles that generate aerodynamic lift including a medium control of it, kites would perfectly fit in this definition.

In 1883, Douglas Archibald using an anemometer, to measure the wind velocity, and a camera took hundreds of photographs during the Spanish-American war, which may have been one of the first uses of UAVs in combat

Not much time later, in the World War I, Charles Kettering developed a biplane UAV used by the Armed Signal Corps. It could fly nearly 40 miles per hour and carry 180 lbs of explosives. However, these vehicles did not have sophisticated controls systems and crashes were very commons. The truly successful era for the UAVs usage was not before the Vietnam-War

With the start of the Cold War, UAVs began to be used as Intelligence, Surveillance and Reconnaissance (ISR) systems, the Vietnam War was America's first conflict that counted on an extensive use of pilotless aircraft. There were more than 3000 operational reconnaissance missions between 1964 and 1975 . Due to the extent of enemy anti-aircraft fire in Vietnam, UAVs were often used to take photos from low and high altitude which could in turn be handled to strike plannings

In Figure 1.1 we can observe that the vast range of models that were developed and used in the military missions by the United States of America during this period. Still, in the next decade after the war, the funding for the UAV programme was scaled down, and the development of this kind of vehicles remained in the background for a while.

However, some aircraft already built, to be more precise "the Pioneer", were used during the Persian Gulf operations between 1990 and 1991. "The Pioneer" flew 300+ combat reconnaissance missions and received extensive acclaim for its performance [**1**].

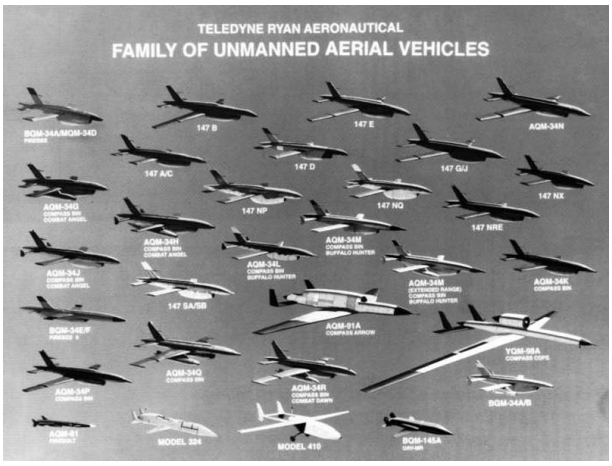


Figure 1.1: Ryan 147, AQM-34 "Lightning Bug" and the family of Teledyne Ryan Aeronautical.

Source: http://blog.udn.com/TUANPETERSON

With the propulsion of this area by the military missions, the UAVs started to be used in the domestic field, and due to this a whole new discussion about safety and privacy was raised. However, as shown before, pilotless aircraft brought to the world a whole new range of possibilities. UAVs used for surveillance, disaster exploration, such as fire, flood, and earthquakes, monitoring leakage of hazardous materials, and search and rescue missions are some of the extent usage of these vehicles nowadays [**2**]. This is one of the reasons why this topic brings a great interest to be studied.

1.2 State of the Art of UAV and UAM

A robot is a machine that can perform a variety of tasks that are typically thought of as risky, difficult, repetitive, or even impossible for humans to perform. This is done in an effort to be as effective as possible.

Robots with fixed bases, such as manipulators, and movable bases, known as mobile robots, are found in mechanical structures. These two categories are the fundamental components of robotics, and this project examines them in a special manner.

In this introduction, we will discuss mobile and manipulative robots and describe the categories of manipulators that are used in the air.

1.2.1 Manipulative robots

The mechanical design of a manipulator robot consists of a set of rigid bodies (links) connected in a specific order by joints. A manipulator is distinguished by a brace that ensures mobility, a grip that confers dexterity, and a goal effector (organe terminal) that completes the task assigned to the robot [**3**].



(a) articulated configuration



(b) Cartesian configuration



(c) Scara I configuration



(d) Scara II configuration

Figure 1.2: The configurations of a manipulator robot.

Source: https://www.packagingstrategies.com

The manipulator robot's movements are supported by joints that may be prismatic, in which case the movement between the two bodies is a translation, or roto-de, in which case the movement between the two bodies is a rotation. Each articulation gives the structure what is known as a degree of freedom (DOF).

To position and align a solid in space, the robot needs six degrees of freedom, three for the object's location and three for its orientation. The robot is regressing from a cinematic standpoint if the number of motorized joints is less than the DOF of the terminal organ.

The three primary articulations' arrangement and kind serve as the basis for categorizing robots Figure:1.2a. There are several types, including the articulated (bras articulated), Scara I, Scara II, and Cartesian. There are two other variants as well, Cylindrical and Spherical, but they are far less frequently used [**4**].

1.2.2 Unmanned aerial vehicle

UAV stands for Unmanned Aerial Vehicle, which is commonly considered to be a drone or an aircraft with no pilot on board. UAVs can be remote-controlled aircraft [**5**]. The unmanned aerial vehicle contains cameras, sensors, communication belonging as well as other payload devices [**6**]. It was created for military usage, and civilian usage to protect the border. UAVs are widely used in the military [**7**]. Unmanned aircraft system manufacturing was started by the United States department of defense (DOD) in 2005 [**8**]. Presently, leading UAV manufacturing countries are the USA, Israel, China, Iran, and Russia [**9**]. Rustom series of Indian UAVs are under development [**10**]. The Building of a

UAV is easier, it is less expensive and it can be created at a reasonable price. UAVs are made of essential components.

There are two categories of UAV. Fixed and Rotatory wing UAV consists of their strength and weakness.

Fixed Wing UAV

Fixed-wing UAV drones have gained popularity in military and defense applications, and fixed-wing has high speed and heavy payload [**11**]. They are not suitable for stationary applications, because fixed-wing UAVs can't do a close inspection.

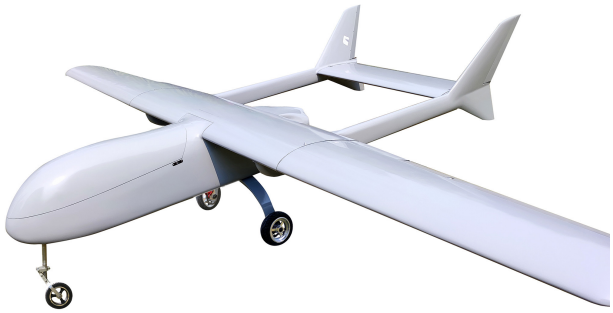


Figure 1.3: Mugin-5 Pro 5000mm Carbon Fiber UAV Platform – Mugin UAV.

Source: https://www.muginuav.com

Fixed-wing UAV drones use a wing to lift themselves in the sky. Fixed-wing drones only use energy to go in a particular direction rather than remaining in one place in the air. The fixed-wing drone can go on long distances and cover larger areas. And fixed-wing UAV drone can spend up to 16 h time in the sky because a fixed-wing UAV drone uses a gas engine rather than using an electric engine.

Rotatory wing UAV

Rotatory-wing UAVs is known as quadcopters, hexacopters, tricopters, and helicopters are the kinds of rotatory wing UAV . they are designed for monitoring the ground situation from the air, for instance, detecting and tracking border state, surveillance of military equipment, and so on. It has been specially created for remote surveillance. Rotatory-wing UAV drones have limited speed and payload rather to fixed-wing. They can stay in a stationary position in the sky. This UAV can do a close inspection. The choice of UAV depends on the uses.



Figure 1.4: Superfox6 Rotary-wing drone by Hélicéo Company.

Source: https://geo-matching.com/products/superfox6-lida

Finally, after several classifications we get to use UAV in this project, the quadcopter, also called quadrotor. These kinds of vehicles are used on missions that require hovering flight. Rotatory wings with four rotors have some unique abilities such as high maneuverability, small size, and easy control. Their most significant applications outside the military field are search-and-rescue and emergency response.

1.2.3 Unmanned Aerial Manipulator

With scientific and technological advancements in sensor technology, UAVs now perform not only passive tasks like surveillance, information gathering, combat operations, etc. but also actively interact with their surroundings through what is known as aerial manipulation.



Figure 1.5: An Example of a Drone equipped with 2 robotic arm by ProDrone Company.

Source: https://uncrate.com/prodrone-robot-arm-drone/

In terms of applications, aerial manipulation includes Grasping, payloads, transport, assembly, disassembly of mechanical parts, measuring equipment, and many other tasks where the object is appropriate for an aerial manipulator.

The UAV should first be outfitted with the necessary tools for performing aerial manipulation tasks, such as cables, grippers, and manipulator arms. They differ mechanically, which affects modeling and control. These manipulation devices are described in the following sections:

Cables

A cable attached to a UAV is typically used in transportation where the connection with the environment requires only extension forces.



Figure 1.6: Drone with Cables.

However, the movement of the charge attached to the cable changes the system's center of mass, which affects both modeling and control.

Gripper

The most commonly used manipulation device is a gripper with a single DDL, which is attached directly to or beneath the UAV. This type of manipulation device has the following advantages: it is simple to build, inexpensive, and easy to model and control.

This type of manipulator has been used in a variety of applications, including the capture of a 150g peluche using a quadrotor equipped with a gripper.

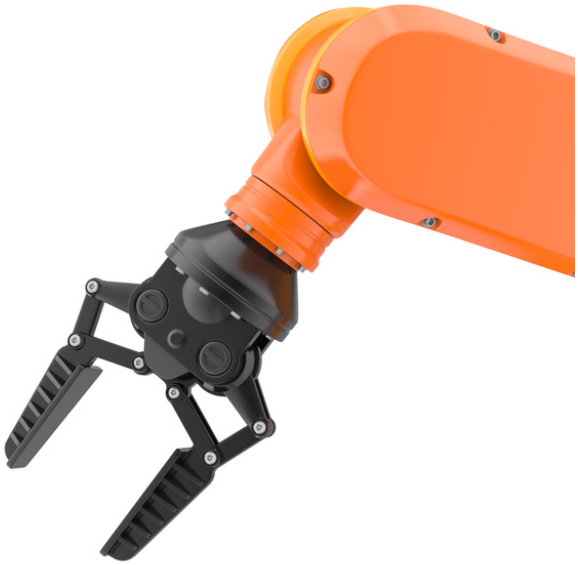


Figure 1.7: Robotic Gripper

These tools, on the other hand, have drawbacks such as limited workspace and the ability to manipulate weights and volumes.

Manipulator bras

To overcome the restrictions of using cables and grippers, a viable approach is to equip an aerial vehicle with a bras manipulator that can delegate environmental contact missions.



Figure 1.8: jaco by Robotics Kinova.

Source: https://assistive.kinovarobotics.com

The design of the manipulator's hand is also important for aerial manipulation; additionally, more capabilities can be added to the aerial platform, and more options can be considered for designing an effective aerial UAM manipulator.

1.3 Control Strategies for Quadrotors

As imagined, each type of UAV has its particular control technique. De- pending on the takeoff way, number of rotors, weight, and so on, several different

approaches specially selected to contemplate these specifications were developed. However, as the focus of this study is quadcopters, the next sections will fall over into a control strategy designed specially for this kind of UAV.

1.3.1 Control System

Firstly, it is important to give a background on this field of study, defining what is a control system and what a control system exactly does. First thing first, a control system is an arrangement or a combination of several different components that work together to arrange or regulate a system to achieve a desired outcome[**12**].

Knowing this basic concept, Figure 1.10 shows the basic structure of control systems. The input signal r(t) is also called a reference signal, which is basically the set-point that you want to achieve in your system. In a control system of this type (closed-loop system), the output signal c(t) is used in order to compare the actual response of the loop to the desired set-point. Therefore, we have the concept of the error signal.

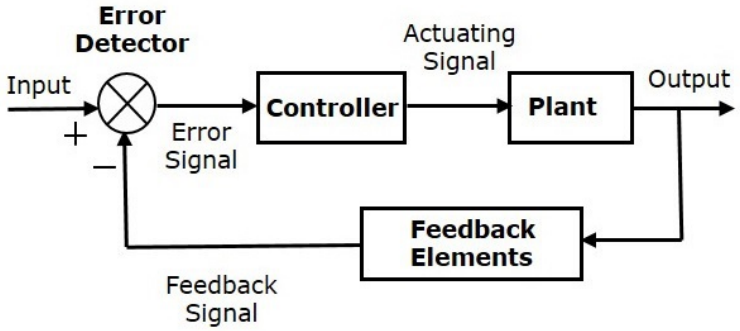


Figure 1.9: Closed Loop Control.

To make a long story short, the main objective of a control system is to get the lowest possible error signal, by matching the output signal c(t) with the reference signal r(t). However, to do that there are several different approaches and techniques, one of them is the well-known PID Control systems Figure : 1.9.

1.3.2 PID Control

The Proportional-Integral-Derivative (PID) control offers the simplest and yet most efficient solution to many real-world control problems [10]. The main idea of this controller is to use not just the error as an input into the system, but also its derivative and integrative response. Being the most used control strategy nowadays, this controller provides us a simple implementation and a good response [**13**]. The basic structure of the PID control is given by the following equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}, \tag{1.1}$$

where:

- u(t) is the control signal at time t.
- e(t) is the error at time t.
- Kp is the proportional gain.
- Ki is the integral gain.
- Kd is the derivative gain.

During the evolution of this study, an analog approach to the PID controller will be used and implemented on the UAV. This approach is called Cascade PID strategy, a controller commonly used to stabilize quadrotors [**13**].

Conclusion

In conclusion, the development and history of unmanned aerial vehicles, or UAVs, show how far technology has come and how it has permeated our daily lives. UAVs have developed in importance since their introduction, providing answers to activities that formerly presented significant threats to human life. Their capabilities and applications have been considerably increased as a result of the current shift to autonomous aircraft systems.

In this chapter, we have examined the extensive history of UAVs, offering insight into the wide variety of emerging vehicles. The quadrotor, which is the subject of our study, stands out among them as a flexible and popular platform with a wide range of potential applications.

Chapter 2

Robotics Operational Systems (ROS) and Setup

Introduction

To completely comprehend the application of Robot Operating System (ROS) in this project, we shall first define What is an Operating System. An Operating System (OS) is a collection of software that manages the raw computational capacity of the computer's hardware, it determines how computer resources, including as the CPU, main memory, input-output devices, communication, and data, are allocated. it also provides a user-friendly interface for running application applications.



Figure 2.1: Examples of Operating Systems: Windows, macOS, Android, Linux.

An operating system (OS) is the machine's manager, defining how the user sees and interacts with it [**14**]. Despite its name, ROS is not an operating system; it does not focus on computer processes and memory allocation, but rather on driver compatibility and previously created state-of-the-art algorithms. In actuality, ROS is a collection of software libraries and tools that aid in the development of robot applications.

2.1 Robot Operating system

ROS is a Linux-based modular framework for robotic applications. It was created by Willow Garage and is now maintained by the Open Source Robotics Foundation.

This is a strong tool because it employs Object-Oriented Programming (OOP), a programming technique that organizes software design around data or objects rather than functions and logic.

As previously said, it operates just like a standard operating system on a personal computer. The distinction is that, rather than apps, ROS programs allow a user to manage the mobile activities of a robot. ROS's combination of tools, libraries, and norms makes it easier to create complicated and robust robot behavior[**15**].

2.1.1 Definitions

ROS software is provided in the form of packages and stacks of packages. A package is a structured directory that can include libraries, executables, configuration files, and so on. To properly grasp this structure, key concepts like as nodes, messages, topics, and bags should be introduced [**15**].

2.1.2 Nodes

A node is, basically, a program (process) being executed. As defined by ROS wiki, "node is a process that performs computation". It is an executable program running inside the application. Nodes are combined into a graph and communicate with each other using ROS messages through topics and services [**15**].

2.1.3 Topics

The concept of the topic is very simple: topics are named buses over which nodes exchange messages. In other words, topics in ROS enable the data to be exchanged between nodes. A topic can have publishers and/or subscribers, where the first sends messages trough the topic and the last one receives those messages [**15**].

2.1.4 Messages

ROS uses a simplified message description language for describing the data values published by a ROS node. Messages are exactly those data values. Message descriptions are stored in files located in a subdirectory of a ROS package [**15**].

2.1.5 Services

The publish/subscribe model is a very flexible communication paradigm, Request/reply is done via a Service, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call.

2.1.6 Bags

A bag is a file format in ROS for storing ROS message data. They offer a convenient way to store all kinds of data during an execution. The main analysis is made using ROS bags stored data [**15**].

2.1.7 Packages

A ROS package is a fundamental organizational unit in the Robot Operating System (ROS), used for bundling and organizing software, scripts, configuration files, and other resources related to a specific robotic application or component. These packages are essential for modularizing and managing the software components of a ROS-based robot or system.

A ROS package typically includes:

- Source Code: The actual code and software libraries that implement specific functionalities or components of a robotic system.
- Configuration Files: Configuration files that define various parameters and settings used by the package.
- Launch Files: Files that specify how nodes (individual software components) within the package should be executed and connected.
- Dependency Information: Information that specifies dependencies on other ROS packages or libraries required for the package to work correctly.
- Documentation: Documentation and information about the package's purpose, usage, and dependencies to help developers and users understand and work with it effectively.

ROS packages are used to organize and distribute software components for robotics projects. They follow a specific directory structure and naming conventions, making it easier for developers to create, share, and integrate software components for building robotic systems.

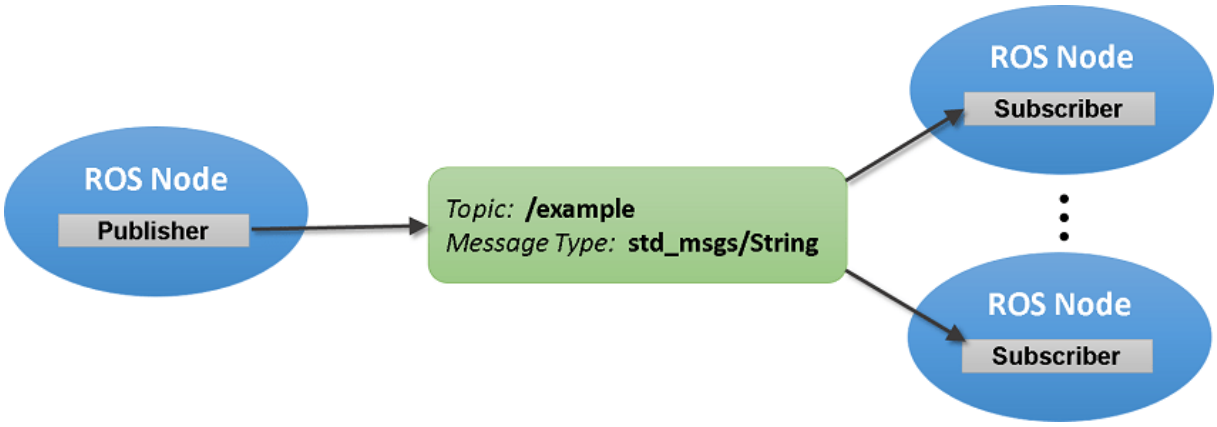


Figure 2.2: ROS Communication Using Nodes Under a Topic.

2.2 Setup

The simulated UAV for this project was a customized DJI f450 drone with a robotic arm attached to it. The architecture design and all tests performed were made on an XMG FUSION 15, personal computer.

2.2.1 Ros Noetic installation

The following line of command allows us to install the latest ROS Noetic Ninjemys on Ubuntu Focal 20.04. Copy and Paste this line of command into the Ubuntu terminal.

```
wget -c https://raw.githubusercontent.com/qboticslabs
/ros_install_noetic/master/ros_install_noetic.sh
&& chmod +x ./ros_install_noetic.sh && ./ros_install_noetic.sh
```

Setup Environment Variable for ROS:

We need to add a Setup.bash script available under the"/opt/ros/noetic/setup.bash" to our system path. The main purpose of this is to set environment variables used by ROS.

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

All the simulations were performed using an XMG FUSION 15. The laptop has a great computational power containing an Intel I7-9750H 2.6 GHz six-core processor and 16 GB of RAM. Also, the computer has a dedicated graphical card, an NVIDIA RTX 2070 with 8GB DDR6. To be able to perform all the simulations with ROS, a Linux Ubunutu 20.04 was installed alongside Windows 10 (dual-boot). To execute all the evaluations and analysis of the results Rqt Console and plotjuggler were used, and all the simulations were done in Gazebo software.

2.3 Other Packages : Moveit , Rviz , RQT , Plotjuggler

2.3.1 Moveit

In this project, we leverage the power of MoveIt, a state-of-the-art motion planning framework in the Robot Operating System (ROS), to facilitate the seamless control of a drone with an attached robotic arm.



Figure 2.3: MoveIt by PickNik Robotics.

Source: https://picknik.ai/news-releases

MoveIt provides comprehensive tools for motion planning, trajectory generation, and collision avoidance, enabling us to efficiently navigate the complexities of the drone-robotic arm system. By defining the drone's kinematics and incorporating the robotic arm's end-effector, we establish a unified control scheme that enables coordinated motion planning for both the drone and the arm. Leveraging MoveIt's intuitive interface and planning algorithms, we develop custom controllers to interface with the hardware, allowing us to execute precise and safe motions. Extensive simulations are conducted to verify the effectiveness and robustness of our approach before deploying it on the physical hardware. Our use of MoveIt empowers us to achieve sophisticated manipulation tasks with the drone and robotic arm, unlocking new possibilities for multi-modal robotics applications and advancing the field of aerial manipulation in dynamic environments.

2.3.2 Rviz

Rviz, an abbreviation for ROS visualization, is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned actions[**15**].

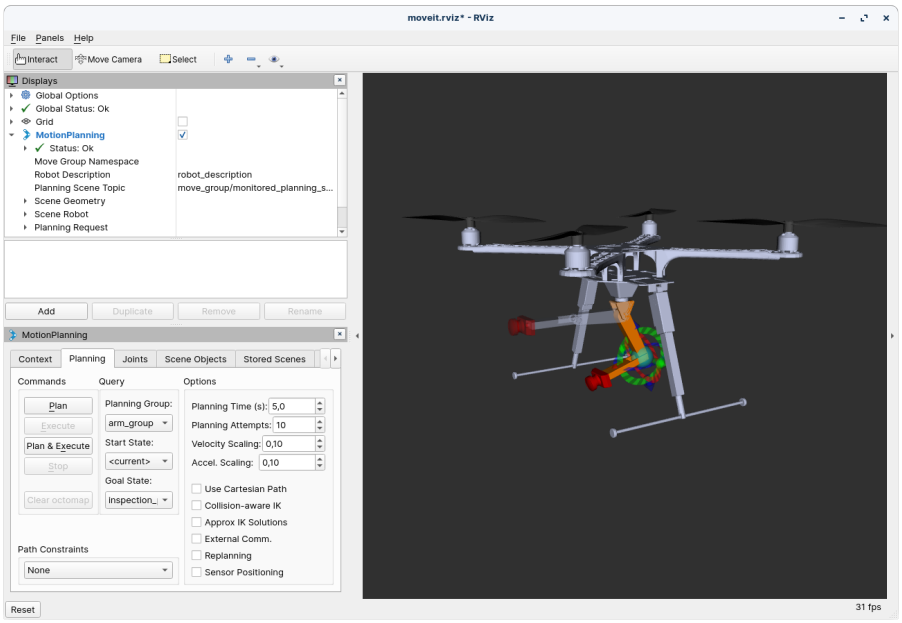


Figure 2.4: Image shows Rviz menu.

Rviz displays 3D sensor data from stereo cameras, lasers, Kinects, and other 3D devices in the form of point clouds or depth images. 2D sensor data from webcams, RGB cameras, and 2D laser rangefinders can be viewed in Rviz as image data.

If an actual robot is communicating with a workstation that is running Rviz, Rviz will display the robot's current configuration on the virtual robot model. ROS topics will be displayed as live representations based on the sensor data published by any cameras, infrared sensors, and laser scanners that are part of the robot's system, this can be useful to develop and debug.

2.3.3 RQT Console

RQT Console is a ROS tool that provides a user-friendly interface for monitoring and debugging the activities of ROS nodes and other components within a robotic system. It displays log messages in real-time, allowing developers and operators to track the system's behavior, diagnose issues, and gather valuable information about the system's operation.

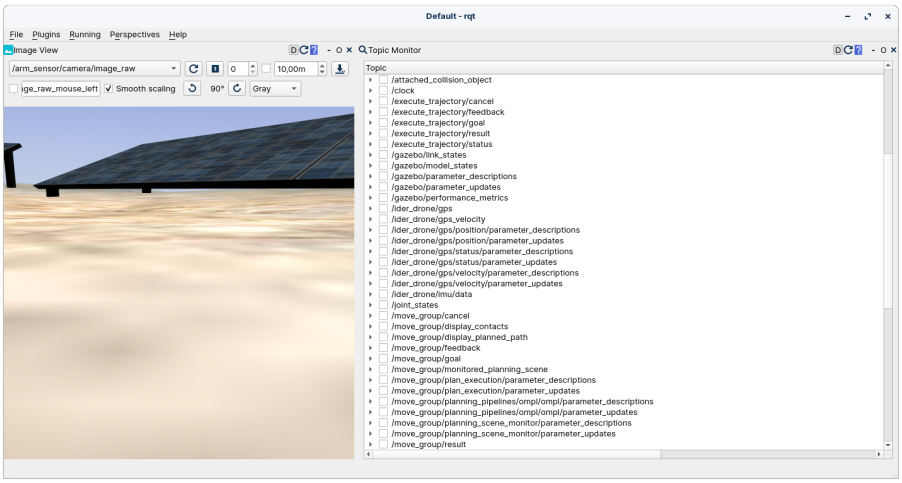


Figure 2.5: Image inside Rqt Console: image viewer on the left and topic monitor on the right.

Key features of the RQT Console may include filtering and sorting capabilities to help users focus on specific types of log messages or events, making it a valuable tool for troubleshooting and debugging ROS-based robotics applications.

2.3.4 Plotjuggler

PlotJuggler is a data visualization and analysis tool that allows users to import, manipulate, and visualize time-series data from multiple sources. It provides a user-friendly interface for creating plots, charts, and graphs to gain insights into the behavior and performance of systems, sensors, and processes over time. PlotJuggler is particularly useful in fields such as robotics, where it helps users analyze sensor data, control signals, and other time-dependent variables to make informed decisions and optimize system performance [**23**].

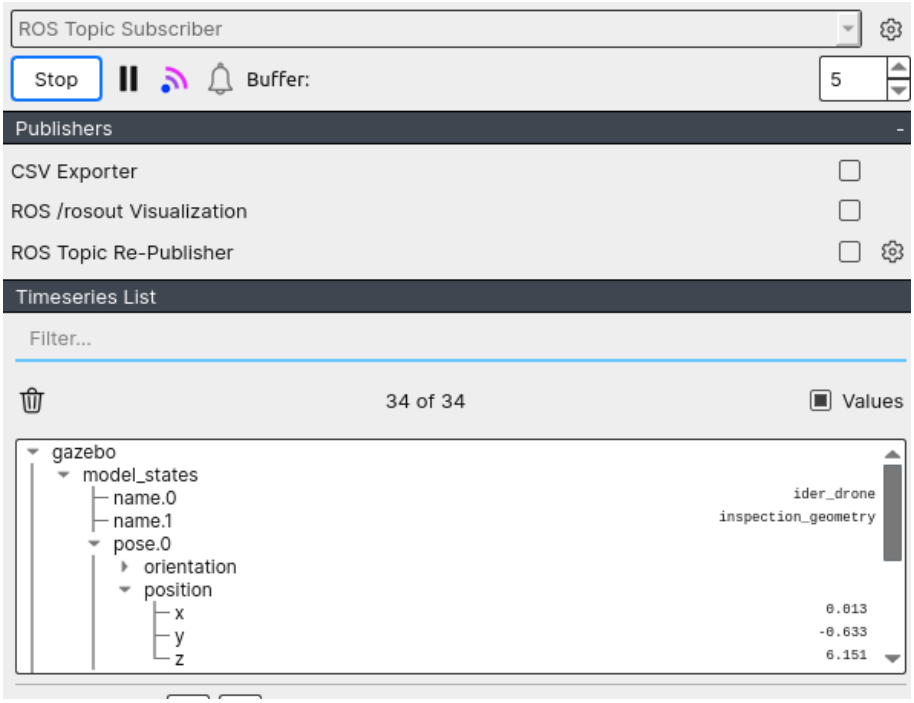


Figure 2.6: Image inside Plotjuggler

Noteworthy features:

- Simple Drag and Drop user interface.
- Load data from a file.
- Connect to live streaming of data.
- Save the visualization layout and configurations to re-use them later.
- Fast OpenGL visualization.
- Can handle thousands of time series and millions of data points.
- Transform your data using a simple editor: derivative, moving average, integral, etc...
- PlotJuggler can be easily extended using plugins.

2.4 ROS Distro: Noetic

ROS Noetic Ninjemys is the thirteenth ROS distribution release. It was released on May 23rd, 2020 [**15**].

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			June 27, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017

Figure 2.7: Some Ros 1 distributions.

ROS Noetic is the updated distribution of ROS Melodic, based on Ubuntu 20.04, and targets Python 3 exclusively. The deprecated Python 2 library was used in the previous ROS distribution, ROS Melodic. ROS Noetic also allows the use of Gazebo 11, whilst ROS Melodic uses Gazebo 9. Working with Python 3 translates into some of the following benefits for users:

- More predictable divisions.
- Advanced iterable unpacking.
- Print is a built-in function.
- Unicode support.

Conclusion

In conclusion, Robotics Operational Systems (ROS) stands as a foundational framework that empowers the development and operation of robotic systems. With its modular and open-source architecture, ROS provides a versatile platform for researchers, engineers, and enthusiasts to create and control robots for various applications. Its widespread adoption in the robotics community and compatibility with various hardware and software components have made it an indispensable tool for the field.

Setting up ROS, especially with the Noetic distribution, offers a robust and up-to-date environment for building and experimenting with robotic systems. Whether you are embarking on educational endeavors, research projects, or industrial applications, a well-configured ROS setup is essential for success.

Additionally, the integration of complementary packages such as MoveIt for motion planning, Rviz for 3D visualization, RQT for graphical tools, and PlotJuggler for data analysis expands the capabilities of ROS, enabling comprehensive development, testing, and analysis of robotic systems.

Chapter 3

Quadcopter-Manipulator Virtual Design

Introduction

This section outlines the comprehensive approach undertaken to model our drone with a robotic arm system, focusing on its integration within the Robot Operating System (ROS). The modeling process, initiated in SolidWorks, allowed us to design each component systematically, culminating in a complete assembly of the robotic system. The transition to ROS further enabled us to define the system's dynamics, kinematics, and relationships through the URDF (Unified Robot Description Format) representation.

Our robotic system comprises a drone equipped with a robotic arm, merging aerial capabilities with versatile manipulation functionalities. The drone's base integrates with precision-engineered motors, and the robotic arm's series of interconnected links culminate in an end effector for dexterous tasks.

3.1 SolidWorks Modeling Process

The modeling process commenced with the identification of all system components, including propellers, brushless motors, drone base arms, structural elements, and the robotic arm's links and end effector. Each component was meticulously designed in SolidWorks, leveraging the software's advanced design tools.

3.1.1 Drone Assembly and Components

The synthesis of meticulous SolidWorks modeling and adaptive design efforts culminated in the seamless assembly of discrete components, uniting to form a singular and awe-inspiring whole: a drone integrated with a sophisticated robotic arm system. From the stable foundation provided by the drone's base arms, equipped with high-performance A2212 brushless motors driving the propellers, to the graceful extension of the robotic arm from the frame, crowned by its versatile adaptive end effector, the assembly represents not only functional excellence but also an enthralling visual spectacle. This collective integration captures the quintessence of modern robotics, where precise engineering and adaptive design harmoniously converge, offering an elegant and transformative solution with the potential to redefine operations across diverse domains.

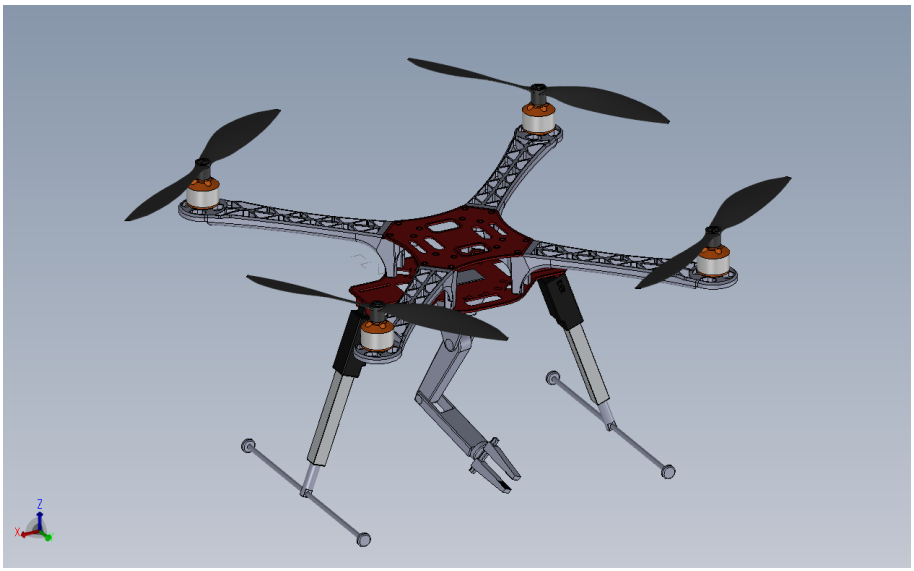


Figure 3.1: Final Assembly 3d view.

Parameter	Value
Total Mass with robotic arm	1.2 [kg]
Drone Mass	1 [kg]
Robotic Arm Length	320 [mm]
Drone Height	290 [mm]
Ixx	0 . 0 0 0 3 4 9 1 [Kg/m2]
Iyy	0 . 0 0 0 3 9 3 [Kg/m2]
Izz	0. 0 0 0 7 2 0 9 2 [Kg/m2]

Table 3.2: System Physical Parameters



Figure 3.2: Landing gear opened view.

3.1.2 Propellers

The propellers were identified as critical components responsible for generating thrust and enabling the drone's flight.

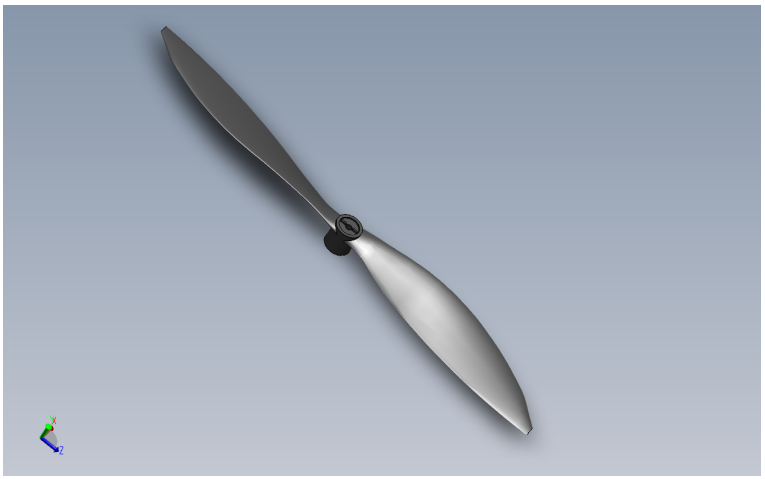


Figure 3.3: propeller 3d view.

Parameter	Value
Mass	38 [g]
Propeller diameter	10 [inch]

Table 3.4: Propeller Physical Parameters

These 10-inch diameter propellers were specifically designed for efficient and stable aerial maneuvers. Characterized by aerodynamics, each propeller was optimized for drone applications, contributing to the system's overall aerodynamic performance.

3.1.3 A2212 Brushless Motor

Central to the drone's propulsion system was the A2212 brushless motor, a robust unit tasked with driving the propellers.

Parameter	Value
Mass	49 [g]
RPM/v	1400 [rpm]

Table 3.6: Brush less Motor Physical Parameters

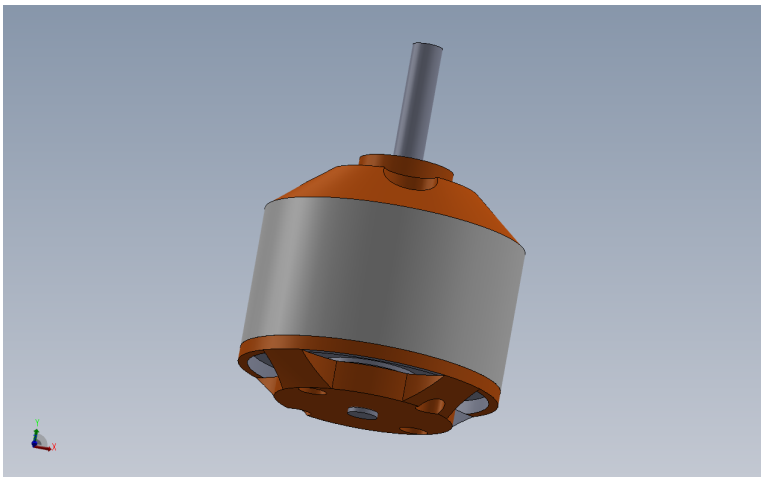


Figure 3.4: A2212 Brushless motor 3d view.

MODEL	KV (rpm/V)	Voltage (V)	Prop	Load Current(A)	Power (W)	Pull (g)	Efficiency (g/W)	Lipo Cell	Weight (g)
A2212	930	11.1	1060	9.8	109	660	6.1	2-4S	52
	1000		1047	15.6	173	885	5.1		
	1400		9050	19.0	210	910	4.3		
	1800		8060	20.8	231	805	3.5		
	2200		6030	21.5	239	732	3.1	2-3S	
	2450		6030	25.2	280	815	2.9		

Figure 3.5: Motors Physical Details.

The motor's design incorporated efficient mechanisms and techniques to ensure consistent performance. Notably, the motor's high RPM capabilities made it an ideal choice for generating the lift necessary for the drone's flight.

3.1.4 Drone Base Arms (4 Parts)

The drone base arms played a pivotal role in providing structural stability and support for the entire system.

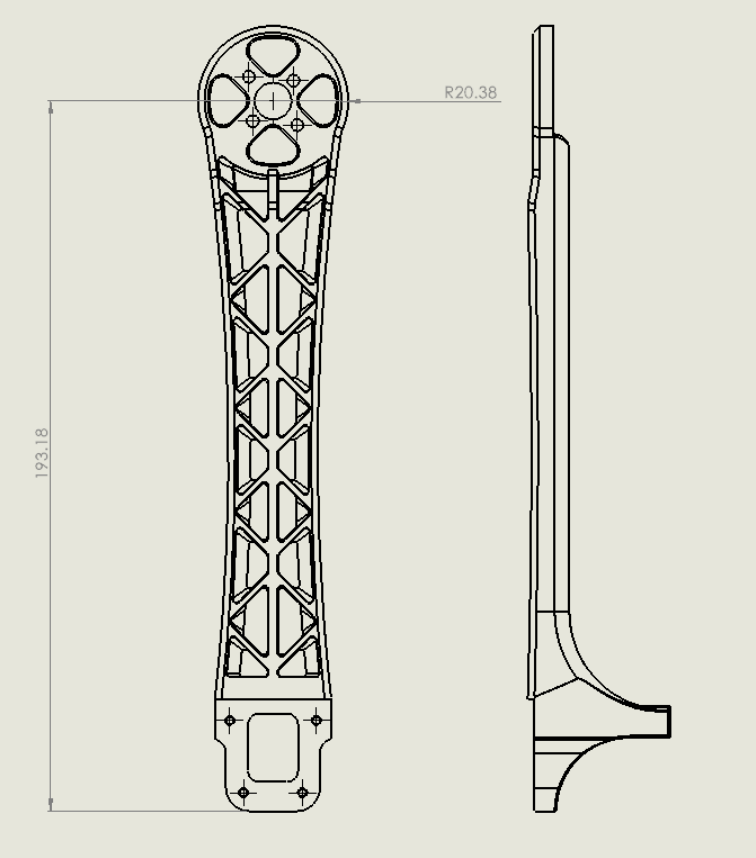


Figure 3.6: Base Arm 3d view.

Parameter	Value
Mass	0.38 [kg]
length between two motors	450 [mm]
length of the arm	190 [mm]

Table 3.8: Base Arm Physical Parameters

Extending from the central drone base, these arms were meticulously designed to withstand the aerodynamic forces encountered during flight.

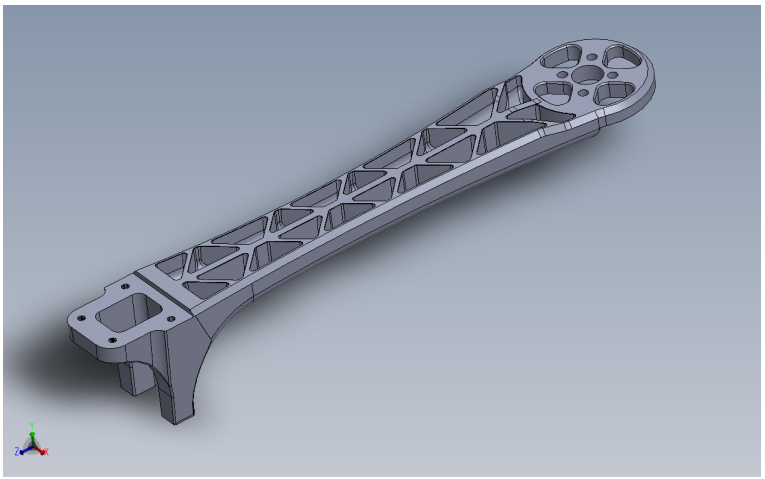


Figure 3.7: Another Base arm 3d view.

The arms not only supported the propellers and motors but also ensured the even distribution of weight across the drone's frame.

3.1.5 Base Top and Base Bottom

The base top and bottom components were integral to the drone's main frame structure.

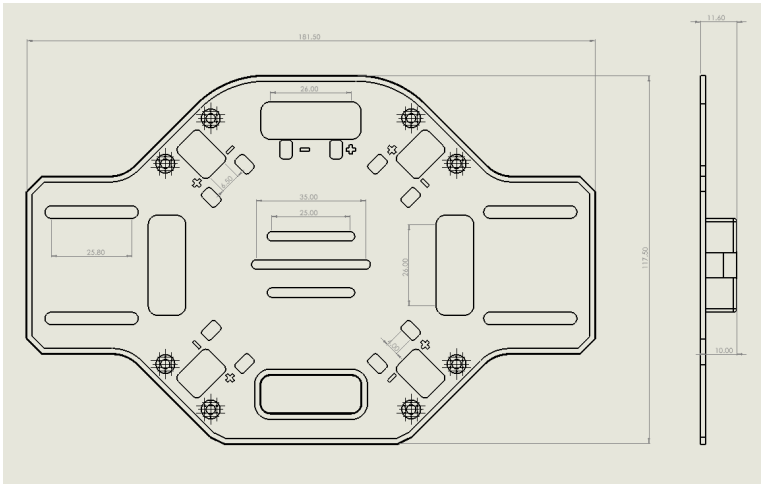


Figure 3.8: Base bottom sketch.

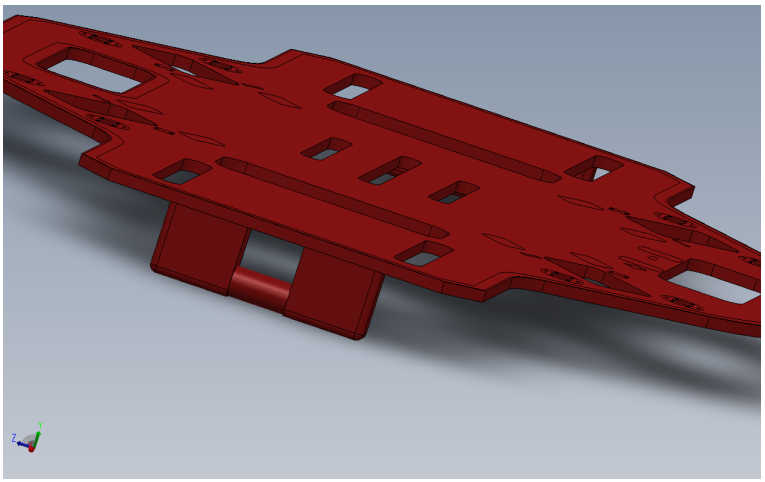


Figure 3.9: Base Bottom 3d view.

The base top served as a secure platform for attaching crucial components such as the flight controller and power distribution units.

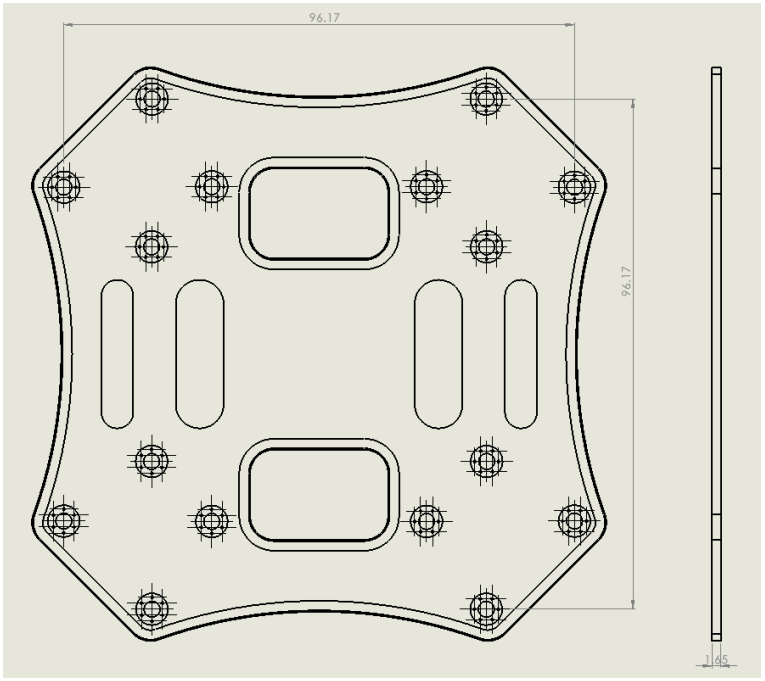


Figure 3.10: Base Top sketch.

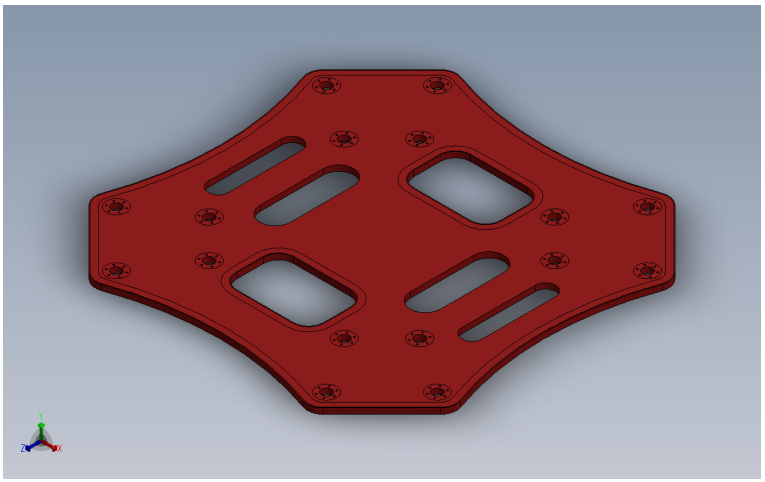


Figure 3.11: Base Top 3d view.

Parameter	Value
Base Top Mass	0.13 [kg]
Base Bottom Mass	0.15 [kg]
Base Bottom Size	38 [cm2]
Base Top Size	45 [cm2]

Table 3.10: Base Physical Parameters

Complementing this, the base bottom component provided structural integrity and rigidity to the drone's overall framework.

3.1.6 Landing Stand (Top and Bottom Parts)

Designed to enhance the drone's functionality during takeoff and landing, the landing stand comprised two parts: the top and bottom components.

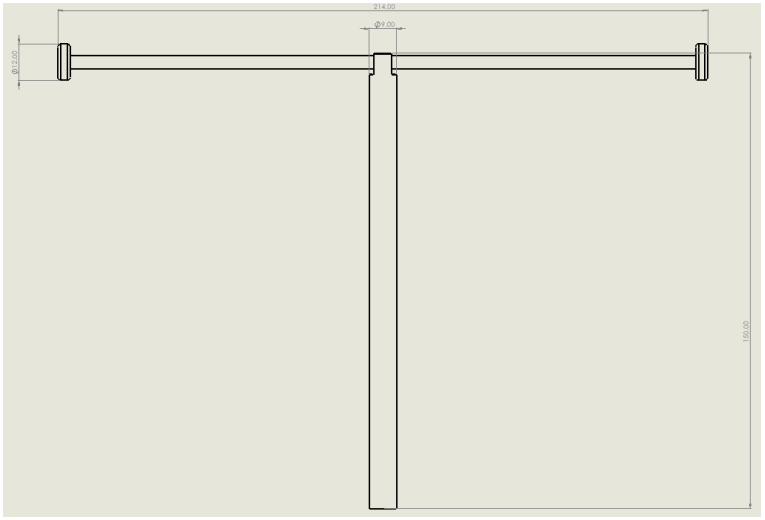


Figure 3.12: Landing bottom sketch.

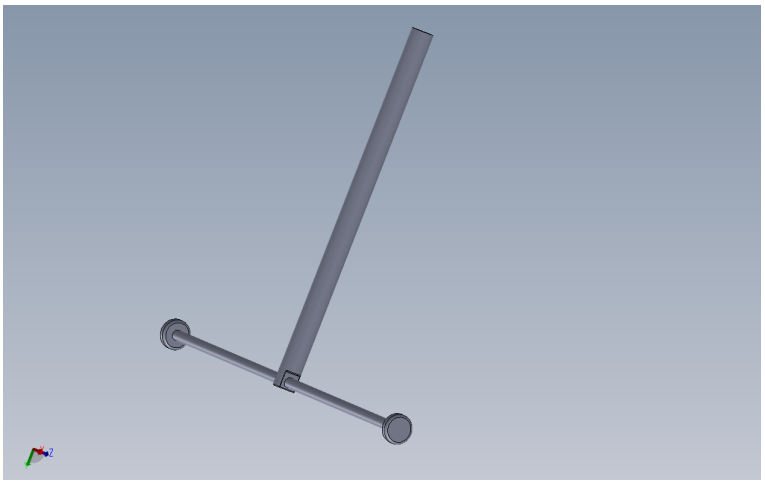


Figure 3.13: Landing bottom 3d view.

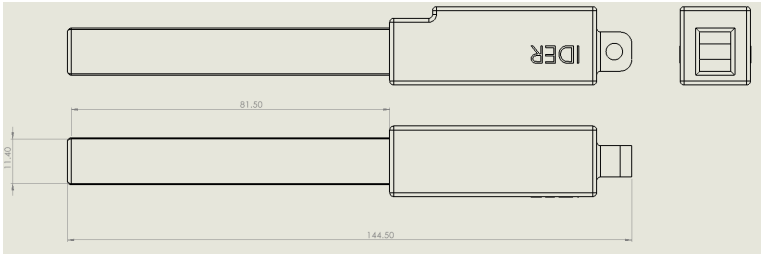


Figure 3.14: Landing Top sketch.

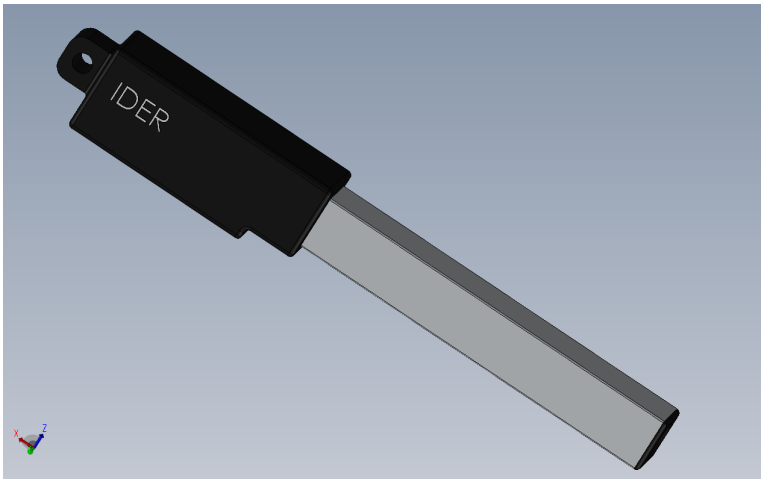


Figure 3.15: Landing Top 3d view.

These components collectively offered a stable platform, safeguarding the drone during these critical phases of flight. Additionally, the landing stand protected the system's components from potential damage when encountering various landing surfaces.

Parameter	Value
Total Mass	0.38 [kg]
Height	175 [mm]

Table 3.12: Landing Gear Physical Parameters

3.1.7 Robotic Arm Components (Part 1 to Part 5)

The link components formed the core structure of the robotic arm. Part 1 served as the foundational arm base, securely attached to the drone base. Following this, parts 2 to 5 formed a sequence of interconnected links, endowing the robotic arm with the ability to articulate and extend. These links were meticulously designed to balance structural integrity with the flexibility needed for the arm's movement.

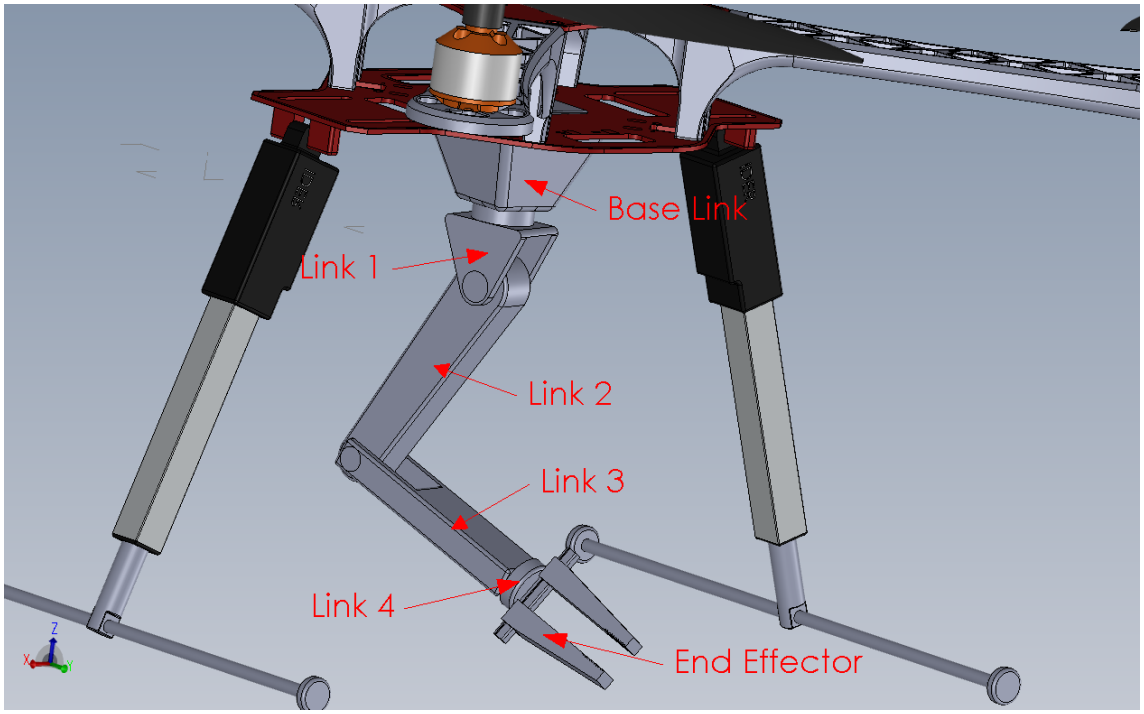


Figure 3.16: Robotic Arm Structure 3d view.

Part 1: Base link

The base link forms the foundation of the robotic arm, providing stability and support for its movements.

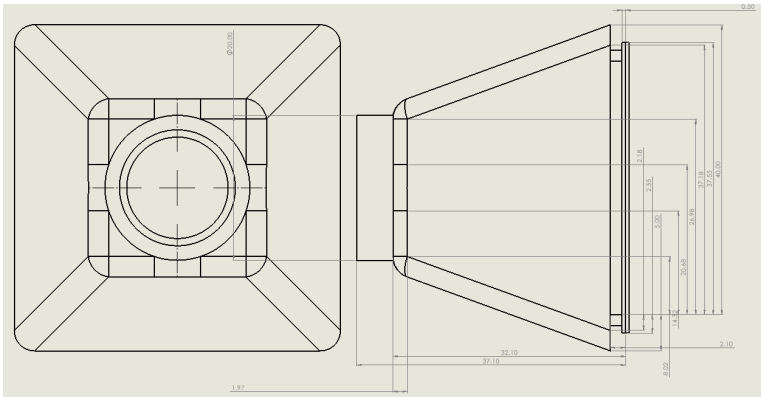


Figure 3.17: Base Link Sketch.

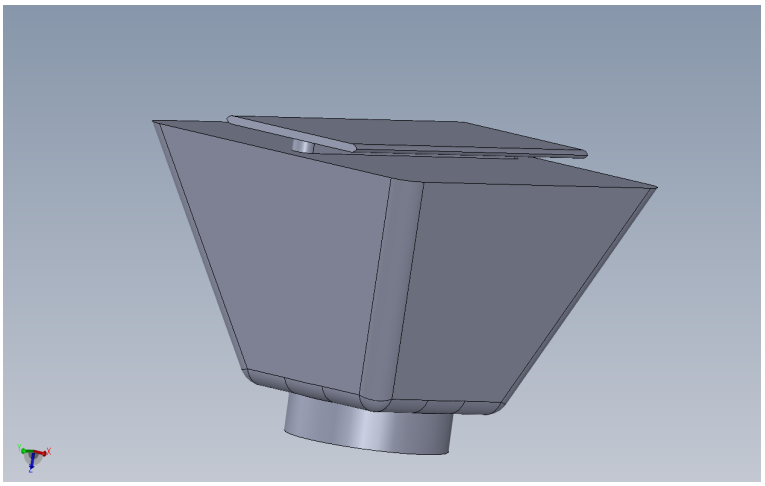


Figure 3.18: Base Link 3d view.

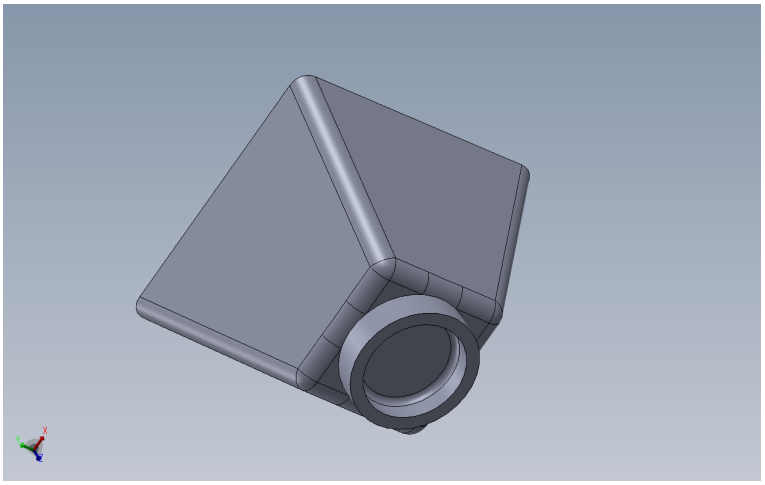


Figure 3.19: Another Base Link 3d view.

Part 2: link 1

Link 1 is the first segment of the robotic arm, extending from the base and allowing the arm to reach outwards.

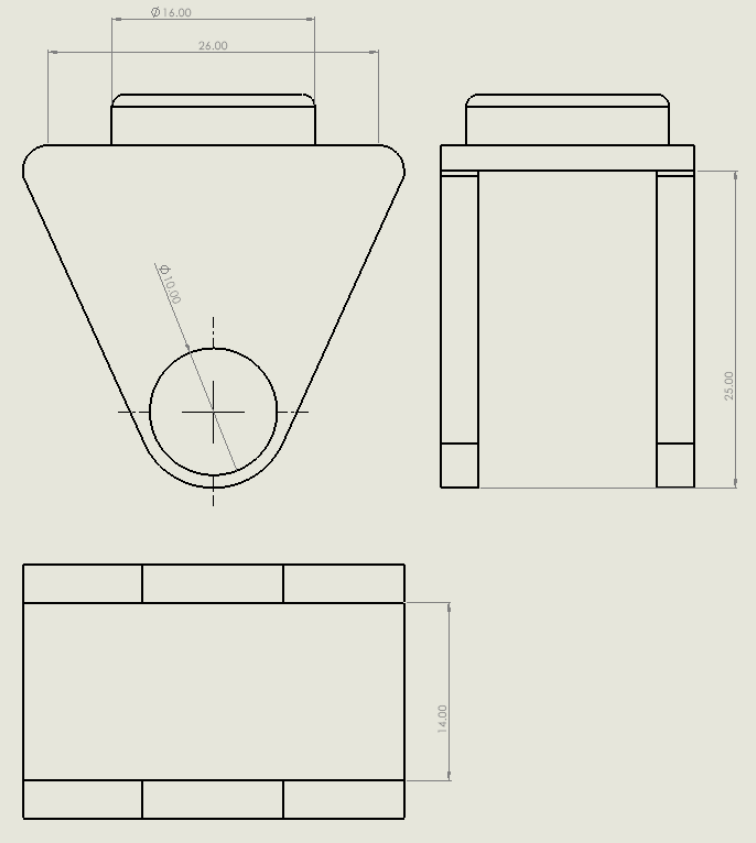


Figure 3.20: Link 1 Sketch.

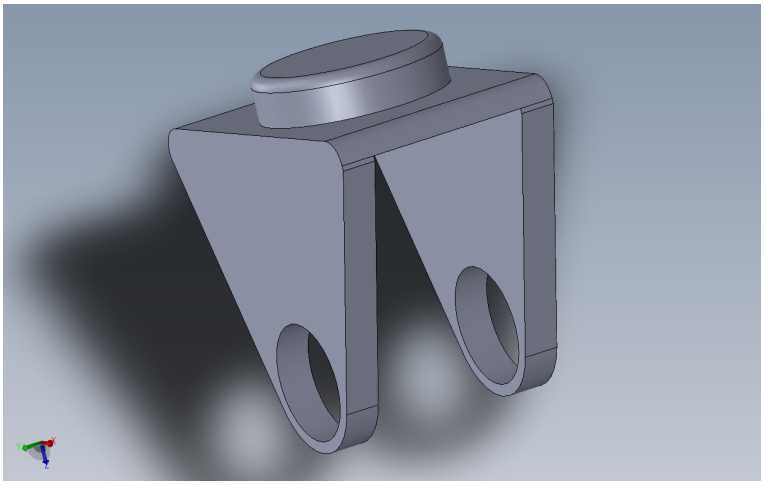


Figure 3.21: Link 1 3d view.

Part 3: link 2

Link 2 follows link 1, extending the arm's reach and flexibility.

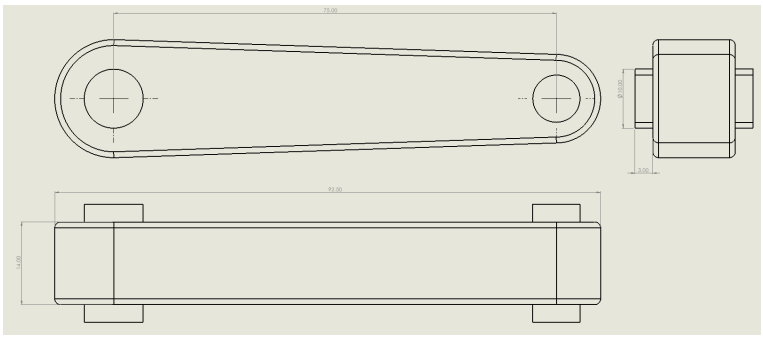


Figure 3.22: Link 2 Sketch.

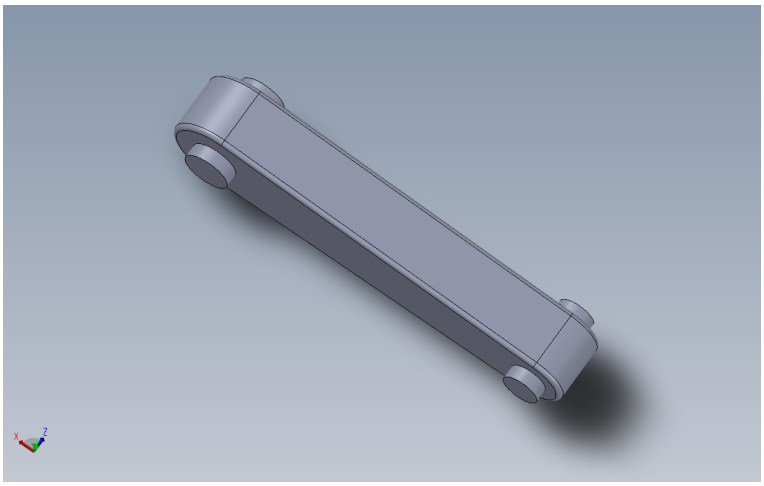


Figure 3.23: Link 2 3d view.

Part 4: link 3

Link 3 is the third segment of the arm, further extending its reach and enabling a wider range of motion.

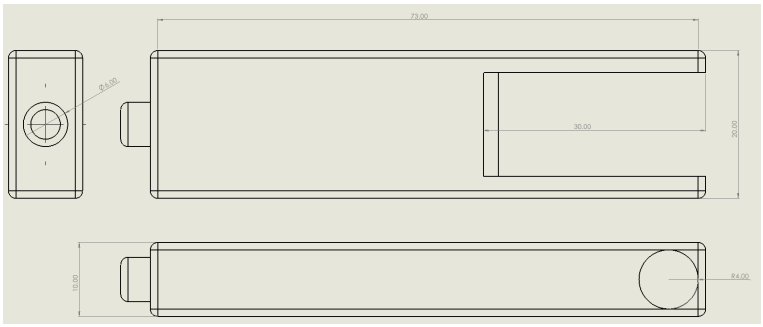


Figure 3.24: Link 3 Sketch.

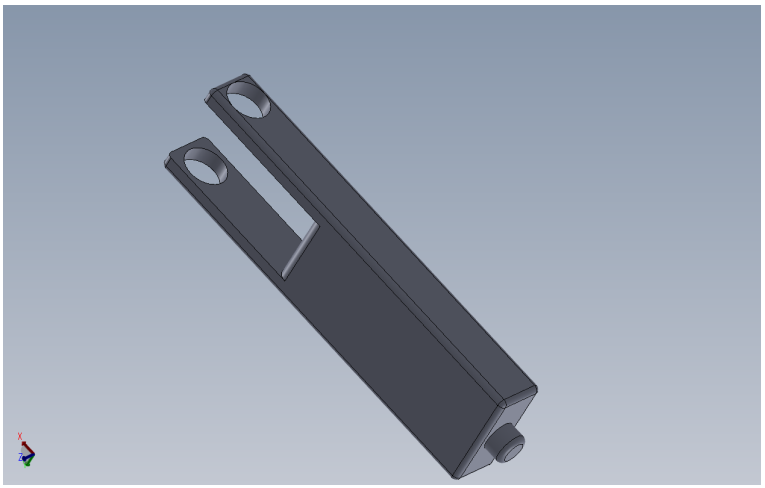


Figure 3.25: Link 3 3d view.

Part 5: link 4

Link 4 is the final segment before the end effector, contributing to the arm's overall length and maneuverability.

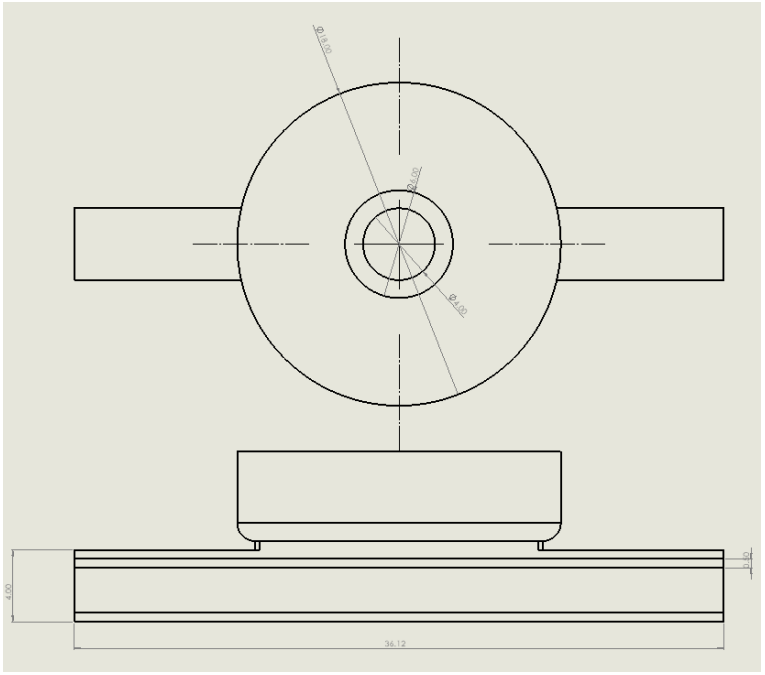


Figure 3.26: Link 4 Sketch.

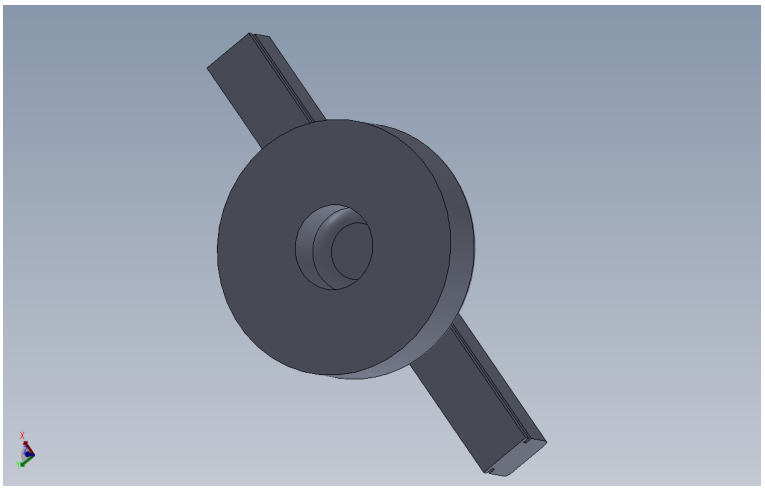


Figure 3.27: Link 4 3d view.

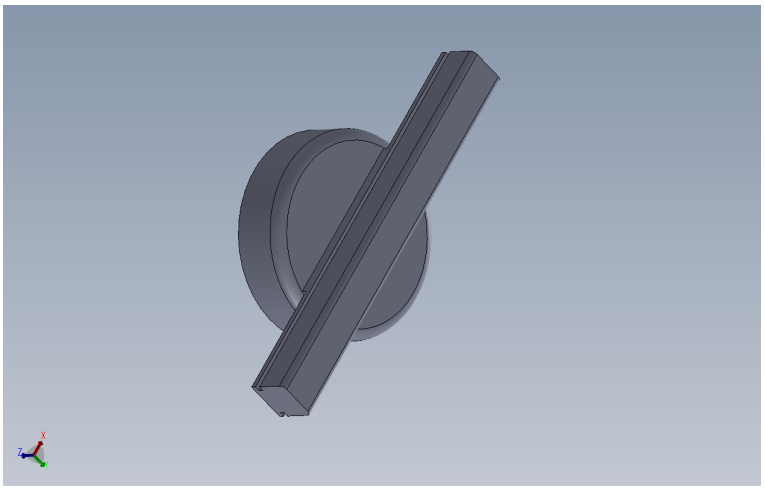


Figure 3.28: Another Link 4 3d view.

3.1.8 End Effector (Part 6)

The end effector, denoted as Part 6, stands as the terminal component of the robotic arm, assuming a crucial role in interfacing with the environment. What sets this end effector apart is its adaptive design, enabling seamless transformation between various functionalities, including a camera, a magnetic gripper, and a pneumatic gripper.

The end effector's adaptability is a key innovation, offering the capability to swiftly interchange its functionalities based on the specific requirements of a task. This adaptability translates into enhanced versatility for the entire robotic system, allowing it to address a range of tasks and scenarios without the need for significant reconfiguration.

- Camera Functionality

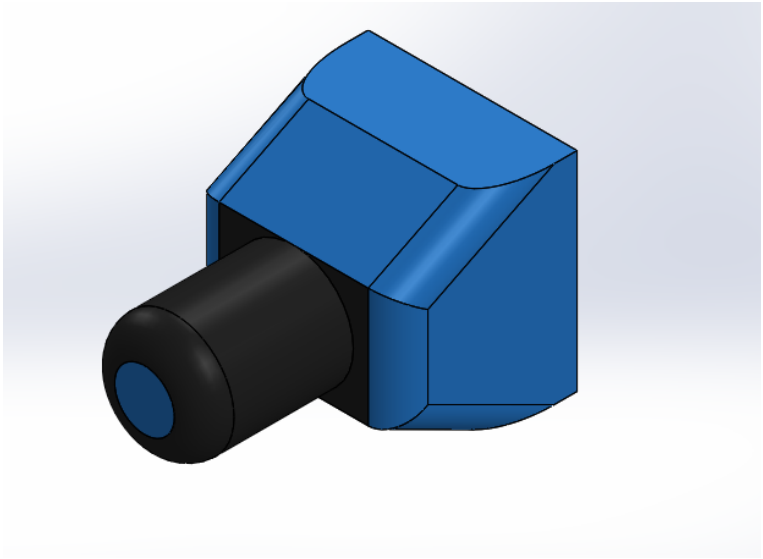


Figure 3.29: Camera 3d view.

- Magnetic Gripper Functionality

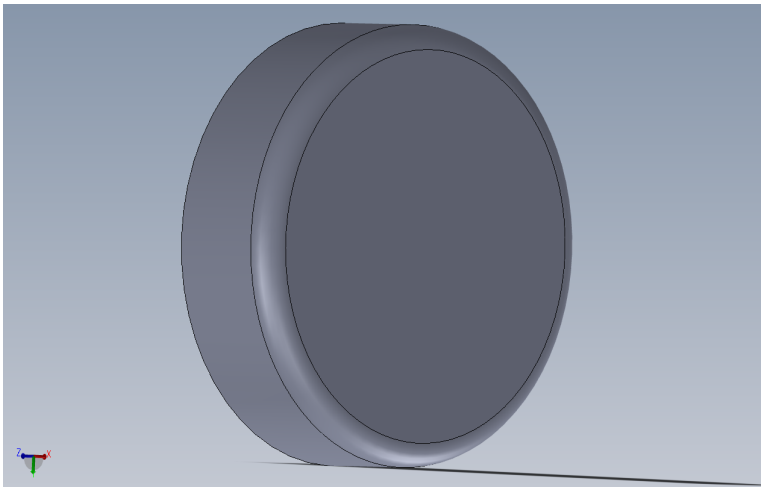


Figure 3.30: Magnetic Gripper 3d view.

- Prismatic Gripper Functionality

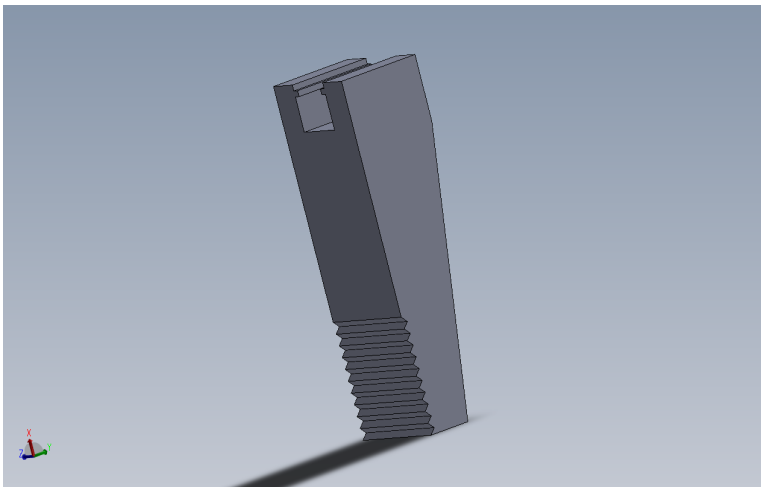


Figure 3.31: Gripper 3d view.

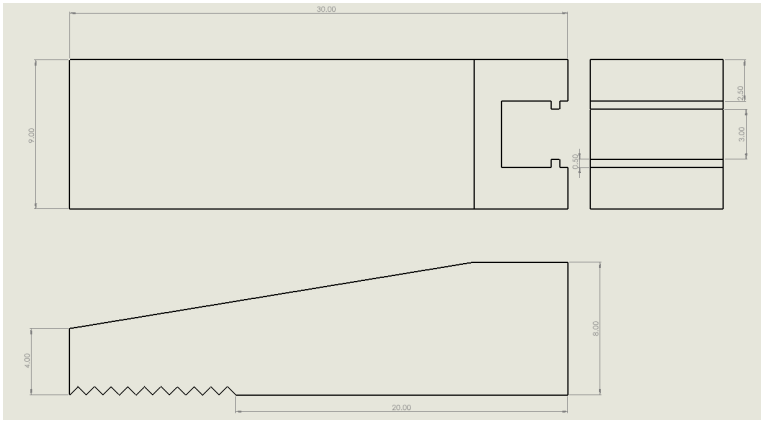


Figure 3.32: Gripper sketch.

Adaptive End Effector with Camera and Gripper Integration

In this study, the development of an adaptive end effector that incorporates both a camera and a gripper significantly enhances the versatility and functionality of the drone-robotic arm system. The integration of a camera provides crucial visual feedback, enabling the robot to perceive its environment and make informed decisions during manipulation tasks. This visual information enhances object detection, recognition, and tracking capabilities, thereby improving the precision and efficiency of the robotic arm's interactions with the

surrounding objects. Furthermore, the inclusion of a changeable gripper allows the system to adapt to various objects and tasks, making it capable of handling diverse shapes, sizes, and materials. By employing MoveIt's motion planning capabilities, the drone-robotic arm system can dynamically switch between different grippers to grasp, manipulate, and release objects with ease and flexibility. The synergistic combination of MoveIt with the adaptive end effector empowers the robot to perform complex manipulation tasks with enhanced dexterity and adaptability, making it a promising solution for a wide range of real-world applications, including search and rescue missions, warehouse automation, and environmental monitoring.

3.1.9 System Configuration:

Drone Configuration

An X-configuration quadcopter is considered to be more stable compared to a + configuration, which is a more acrobatic configuration. Propellers 3 and 4 rotate counter clockwise (CW), and 1 and 2 rotate counter-counter clockwise (CCW).



Figure 3.33: X Configuration.

a quadcopter's motors generate four primary forces that enable its flight. These forces are typically modeled within simulation environments like Gazebo when aerodynamics are restricted, Each motor generates vertical thrust, counteracting the gravitational force on the quadcopter's mass. The sum of the thrust from all four motors determines the quadcopter's ability to ascend, descend, or hover at a certain altitude.

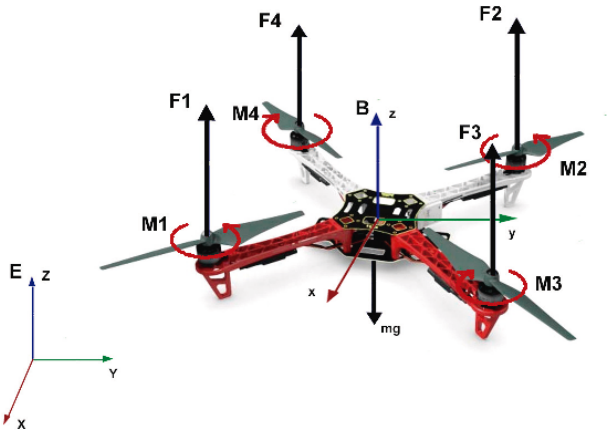


Figure 3.34: Forces acting on a quadcopter.

In Gazebo simulations, these forces can be modeled through plugins that simulate the interactions between the motors, propellers, and the quadcopter's physical properties.

By accurately modeling these forces and their interactions, Gazebo allows us to experiment with various control algorithms and test the quadcopter's behavior in a controlled environment, even when the full complexity of aerodynamics is not considered.

Motor Mixing Algorithm

A motor mixing algorithm, often used in the context of multi-rotor drones or robotic systems, is a set of mathematical rules or calculations that determine the appropriate power distribution to the individual motors or actuators to achieve desired movements or control behavior.

In this simplified example:

$$\begin{aligned}
\text{Thrust_M1} &= \text{T} - \text{P} - \text{R} + \text{Y} \\
\text{Thrust_M2} &= \text{T} - \text{P} + \text{R} - \text{Y} \\
\text{Thrust_M3} &= \text{T} + \text{P} + \text{R} + \text{Y} \\
\text{Thrust_M4} &= \text{T} + \text{P} - \text{R} - \text{Y}
\end{aligned}$$

- ThrustM1, ThrustM2, ThrustM3, and ThrustM4 represent the thrust values for each motor.
- T is the overall throttle command.
- P is the pitch command.
- R is the roll command.
- Y is the yaw command.

In essence, a motor mixing algorithm takes input commands (such as roll, pitch, yaw, and throttle) and calculates how much power each motor should receive to achieve the desired orientation, movement, or stability of the vehicle or robot. These algorithms are essential for maintaining control and stability in complex systems with multiple motors, such as quadcopters, hexacopters, and other multi-rotor platforms [**22**].

The specific motor mixing algorithm used can vary depending on the configuration of the vehicle or robot and the desired control behavior. Different algorithms may be employed for tasks like hovering, maneuvering, or tracking a specific trajectory, with the goal of ensuring that all motors work together harmoniously to achieve the desired outcome.

Robotic Arm Configuration

Robotic arm configuration, in the context of robotics, refers to the arrangement or pose of the individual joints and links that make up a robotic arm Figure : 3.35. It describes the positions and orientations of these components relative to each other and to a reference frame, defining the arm's overall shape and spatial positioning.

joints	type	lower (rad)	upper (rad)	effort	velocity (rad/s)
j_1 (propeller1)	continuous	/	/	300	100
j_2 (propeller2)	continuous	/	/	300	100
j_3 (propeller3)	continuous	/	/	300	100
j_4 (propeller4)	continuous	/	/	300	100
j_5 (link_1)	revolute	0	3.142	300	3
j_6 (link_2)	revolute	0	3.142	300	3
j_7 (link_3)	revolute	-2.09	2.09	300	3
j_8 (link_4)	revolute	0	3.142	300	3
j_9 (eef_1)	prismatic	0	0.01	100	1
j_10 (eef_2)	prismatic	-0.01	0	100	1

Figure 3.35: Joints Configuration of the Robotic Arm.

Key elements of robotic arm configuration include:

- Joint Angles: The angles at which the joints are positioned. These angles determine how the different segments or links of the arm are oriented in relation to each other.
- Link Lengths: The lengths of each link or segment of the arm. These lengths affect the arm's reach and workspace.
- End-Effector Orientation: The orientation of the end effector (tool or payload) is attached to the last link of the arm. It specifies how the end effector is positioned relative to the arm's base.
- Degrees of Freedom (DOF): The number of independent ways the arm can move or change its configuration. Each joint typically contributes to the arm's DOF.

The configuration of a robotic arm is crucial for determining its capabilities, range of motion, and suitability for specific tasks. Robot control systems use configuration information to plan and execute movements, enabling the arm to perform tasks such as picking and placing objects, welding, assembly, or any other operation it is designed for. Different arm configurations can be optimized for different applications, and their design is a critical aspect of robotics engineering.

Conclusion

In conclusion, the systematic exploration of the system description, modeling, and ROS package development has provided invaluable insights into the world of robotics. The journey began with an introduction to the essential components of the process, setting the stage for what lay ahead.

The SolidWorks modeling process emerged as a fundamental step, enabling the creation of precise and detailed representations of the robotic system. This process served as the cornerstone for the subsequent steps, ensuring an accurate and comprehensive understanding of the robot's physical structure.

Chapter 4

SolidWorks Design to ROS Package

Introduction

To seamlessly transition from SolidWorks to ROS, we leveraged the SwtoURDF exporter plugin. This tool facilitated the conversion of our SolidWorks model into URDF format, a standard within ROS for describing robot structures. The plugin enabled the assignment of frames, joints, efforts, velocities, and limits, facilitating precise representation of the system's physical and dynamic characteristics.

4.1 SWtoURDF Exporter Plugin

The SolidWorks to URDF exporter is a SolidWorks add-in that allows for the convenient export of SW Parts and Assemblies into a URDF file. For single SolidWorks parts, the part exporter will pull the material properties and create a single link in the URDF. For assemblies, the exporter will build the links and create a tree based on the SW assembly hierarchy Figure:4.1. The exporter can automatically determine the proper joint type, joint transforms, and axes [**16**].

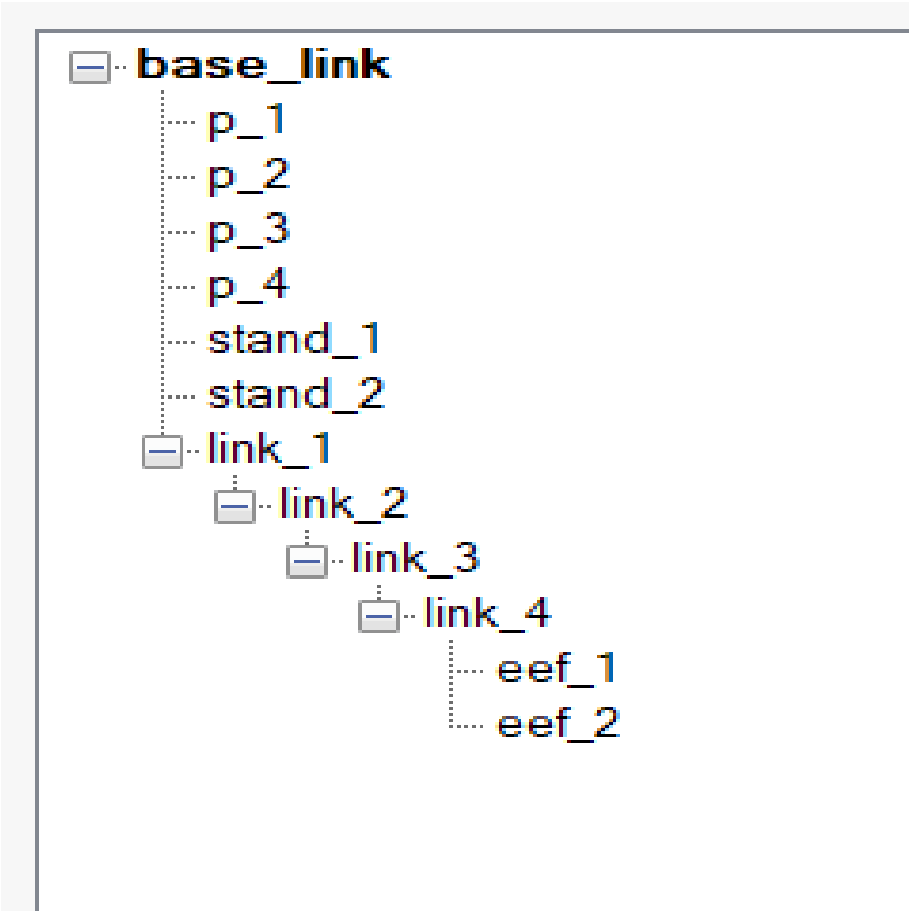


Figure 4.1: Urdf Exporter Configuration.

4.1.1 Ros package architecture

The exporter will create a A ROS-like package that contains a directory for meshes, textures, and robots (URDF files).

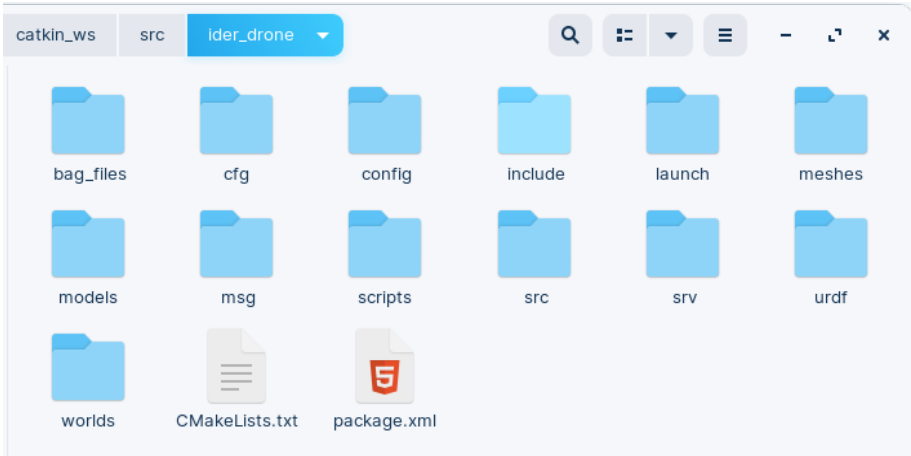


Figure 4.2: Our package directory.

4.2 Prepare the package

One of the most important aspects of working with ROS is setting up a Catkin workspace and creating packages to organize and share our work effectively. Once we have an existing ROS package that contains our results or functionality, integrating it into our Catkin workspace is a straightforward process.

In this section, we'll walk through the process of creating a Catkin workspace and packaging our results.

4.2.1 Setting Up a Catkin Workspace

A Catkin workspace provides a structured environment for ROS development. It keeps our packages organized and streamlines the process of building and deploying ROS nodes.

Create the Workspace Directory:

Begin by creating a dedicated directory for our Catkin workspace. Enter the following command inside your Linux system (Ubuntu ...):

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws
```

Adding our ROS Package:

Start by locating the directory containing the existing ROS package we wish to integrate. Copy the entire package directory to the 'src' directory of your Catkin workspace.

```
cp -r /path/to/existing_results_pkg ~/catkin_ws/src/
```

Build the Workspace:

By returning to the root of our Catkin workspace were able to build the workspace using the 'catkin make ' command.

```
cd ~/catkin_ws
catkin_make
```

Source the Workspace:

Before using the packages source the setup files to inform ROS about their location:

```
source devel/setup.bash
```

4.2.2 Organize our Package

Inside our package directory, we organize our files according to ROS conventions by creating directories like src for source codes, launch for launch files, and config for configuration files.

launch folder

The launch directory holds launch files. Launch files are XML files that define how ROS nodes are executed together to perform a specific task or start a robot's functionality. Launch files enable us to configure and start multiple nodes with a single command. This is especially useful when we want to launch several nodes simultaneously and define their parameters.

```
<launch>
  <include
    file="$(find gazebo_ros)/launch/empty_world.launch" />
  <node
    name="tf_footprint_base"
    pkg="tf"
    type="static_transform_publisher"
    args="0 0 0 0 0 0 base_link base_footprint 40" />
  <node
    name="spawn_model"
    pkg="gazebo_ros"
```

```
    type="spawn_model"
    args="-file $(find drone_arm_pkg)/urdf/
    drone_arm_pkg.urdf -urdf
    -model drone_arm_pkg"
    output="screen" />

</launch>
```

This is an example of a launch file that can spawn our robot model into a gazebo simulation. It launches an empty gazebo world with a node "spawn model" to spawn the URDF model resulting from the modeling process.

```
<launch>
  <arg
    name="model" />
  <param
    name="robot_description"
    textfile="$(find drone_arm_pkg)/urdf/drone_arm_pkg.urdf" />
  <node
    name="joint_state_publisher_gui"
    pkg="joint_state_publisher_gui"
    type="joint_state_publisher_gui" />
  <node
    name="robot_state_publisher"
    pkg="robot_state_publisher"
    type="robot_state_publisher" />
  <node
    name="rviz"
    pkg="rviz"
    type="rviz"
    args="-d $(find drone_arm_pkg)/urdf.rviz" />
</launch>
```

This is another example of a launch file that can display our robot model in Rviz. It runs three nodes: "robotstatepublisher", "jointstatepublishergui", and "Rviz." The following image explains how they work.

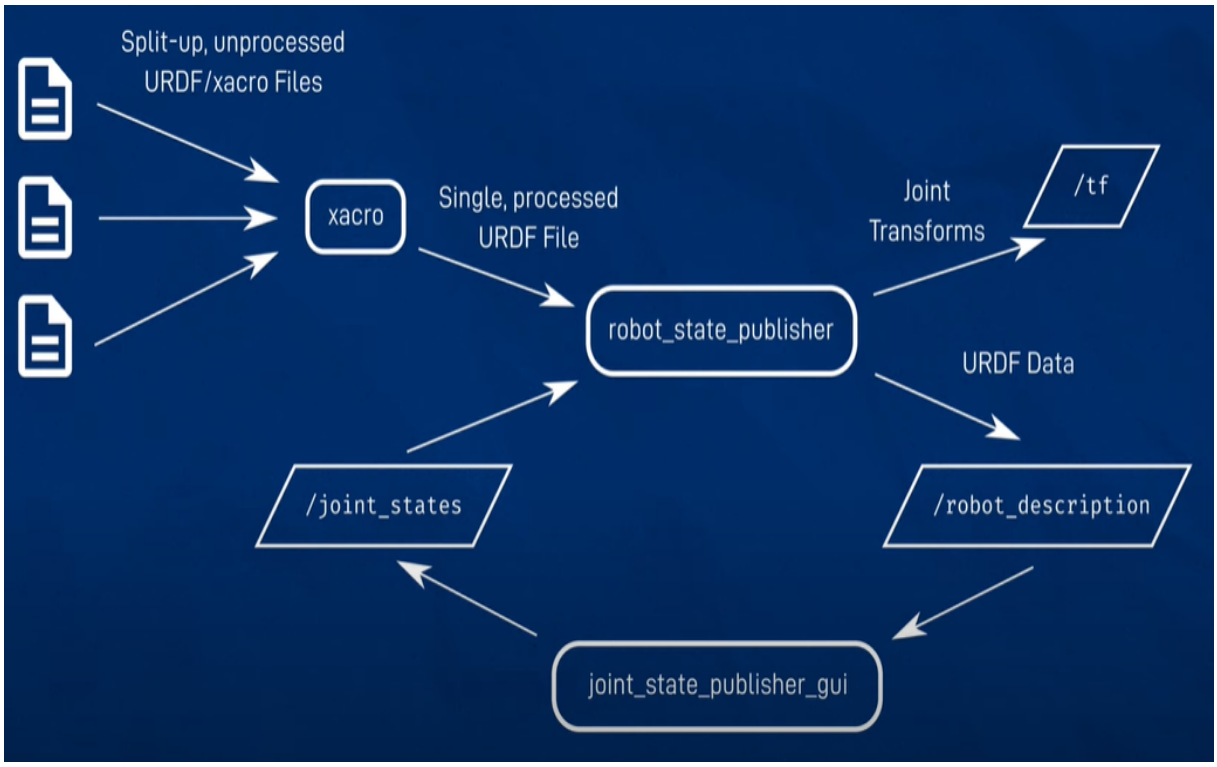


Figure 4.3: How URDF has been processed by ROS.

The important thing is the robot description parameter. This parameter is saved inside the core server of ROS, named master core. It holds all the robot descriptions, from frames to links, relations, etc.

Config

All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and is a common practice to name the folder config to keep the configuration files in it.

Scripts Folder

The scripts directory is where we can store executable scripts written in programming languages like Python or Bash. These scripts can be used as ROS nodes to perform specific tasks, interact with other nodes, or process data.

Messages and Services Folders

The msg and srv directories are used to define custom message and service types for our package. Messages are used to define data structures that ROS nodes can exchange, while services define the request and response structures for communication between nodes.

msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.

srv: an srv file describes a service. It is composed of two parts: a request and a response.

msgs are just simple text files with a field type and field name per line. The field types you can use are:

- int8, int16, int32, int64 (plus uint*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array[] and fixed-length array[C]

urdf folder

The URDF (Unified Robot Description Format) directory is used to store robot description files. A URDF file describes a robot's mechanical structure, visual appearance, and physical properties. This is crucial for visualization, simulation, and control of robotic systems in ROS.

```
<?xml version="1.0" encoding="utf-8"?>

<robot
  name="drone_arm_pkg">
  <link
    name="base_link">
    <inertial>
      <origin
        xyz="-6.47240633018926E-10 -3.75430764806274E-06
        0.0228332219876755"
        rpy="0 0 0" />
      <mass
        value="0.234325224915934" />
      <inertia
        ixx="0.000349102240485739"
        ixy="6.77886440309382E-12"
        ixz="3.25889955687601E-13"
        iyy="0.00039300028629963"
        iyz="-8.8637785450445E-09"
        izz="0.000720919613670048" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
```

```
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://drone_arm_pkg/meshes/
          base_link.STL" />
      </geometry>

    .....
    ...
    .

     <joint
    name="j_10"
    type="continuous">
    <origin
      xyz="-0.0699999999999999 0 0"
      rpy="1.57079632679489 0 1.5707963267949" />
    <parent
      link="link_3" />
    <child
      link="link_4" />
    <axis
      xyz="0 0 -1" />
  </joint>
</robot>
```

This URDF snippet represents a robot's mechanical structure, specifically an arm-like structure for a drone. The <robot> element defines the robot's name. The <link> elements define individual components of the robot, each with its inertial properties, visual appearance, and geometry details. The <joint> elements specify joints that connect the links, detailing their names, types (like continuous or fixed), origins, and axis orientations.

Package.xml and CmakeLists.xml

The package manifest is an XML file called package.xml that must be included with any catkin-compliant package's root folder. This file defines the properties of the package such as the package name, version numbers, authors, maintainers, and dependencies on other Catkin packages. Note the concept is similar to the manifest.xml file used in the legacy rosbuild build system.

```
<package format="2">
  <name>drone_arm_pkg</name>
  <version>1.0.0</version>
  <description>
    <p>URDF Description package for drone_arm_pkg</p>
  </description>
  <author>TODO</author>
```

```
  <maintainer email="TODO@email.com" />
  <license>BSD</license>
  <buildtool_depend>catkin</buildtool_depend>
  <depend>roslaunch</depend>
  <depend>robot_state_publisher</depend>
  <depend>rviz</depend>
  <depend>joint_state_publisher_gui</depend>
  <depend>gazebo</depend>
  <export>
    <architecture_independent />
  </export>
</package>
```

All packages have at least one dependency, These six types of dependencies are specified using the following respective tags: Build Dependencies, Build Export Dependencies, Execution Dependencies, Test Dependencies, Build Tool Dependencies, and Documentation Tool Dependencies.

The file CMakeLists.txt is the input to the CMake build system for building software packages. Any CMake-compliant package contains one or more CMakeLists.txt files that describe how to build the code and where to install it. The CMakeLists.txt file used for a catkin project is a standard vanilla CMakeLists.txt file with a few additional constraints.

```
cmake_minimum_required(VERSION 2.8.3)

project(drone_arm_pkg)

find_package(catkin REQUIRED)

catkin_package()

find_package(roslaunch)

foreach(dir config launch meshes urdf)
        install(DIRECTORY ${dir}/
                DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}/${dir})
endforeach(dir)
```

Our CMakeLists.txt file MUST follow this format otherwise your packages will not build correctly. The order in the configuration DOES count.

- Required CMakeVersion (cmakeminimumrequired)
- Package Name (project())
- Find other CMake/Catkin packages needed for build (findpackage())
- Enable Python module support (catkinpythonsetup())

- Message/Service/Action Generators (addmessagefiles(), addservicefiles(), addactionfiles())
- Invoke message/service/action generation (generatemessages())
- Specify package build info export (catkinpackage())
- Libraries/Executables to build (addlibrary()/addexecutable()/targetlinklibraries())
- Tests to build (catkinaddgtest())
- Install rules (install())

4.2.3 Apply necessary modification to achieve our needs

To tailor the ROS package to our specific needs, We initiate our journey by meticulously incorporating necessary libraries into the CMakeLists.txt file, fortifying the foundation for our package's functionalities. With precision, we outline dependencies to seamlessly blend ROS capabilities with our objectives. Accompanying this, we meticulously prepare the package.xml file, defining our package's distinctive features and requirements.

```
find_package(catkin REQUIRED COMPONENTS
  message_generation
  gazebo_msgs
  roscpp
  rospy
  std_srvs
  geometry_msgs
  sensor_msgs
  nav_msgs
  tf
  tf2_ros
  std_msgs
  gazebo_ros
)

## Generate messages in the 'msg' folder
 add_message_files(
   FILES
   prop_speed.msg
   edrone_cmd.msg
   MarkerData.msg
 )

## Generate services in the 'srv' folder
add_service_files(
  FILES
  Gripper.srv
```

```
)

#####################################
## catkin specific configuration ##
#####################################

catkin_package(
  INCLUDE_DIRS include
  LIBRARIES
        DEPENDS
              roscpp
              gazebo_ros
       std_msgs
       geometry_msgs
       tf
       message_runtime
       dynamic_reconfigure

)

############
## Build ##
############

add_library(gazebo_UAM_propulsion src
/gazebo_UAM_propulsion.cpp)
add_dependencies(gazebo_UAM_propulsion
ider_drone_generate_messages_cpp)
target_link_libraries (gazebo_UAM_propulsion
${catkin_LIBRARIES} ${GAZEBO_LIBRARIES})

add_library(gazebo_ros_gps src/gazebo_ros_gps.cpp)
add_dependencies(gazebo_ros_gps
ider_drone_generate_messages_cpp)
target_link_libraries (gazebo_ros_gps
${catkin_LIBRARIES} ${GAZEBO_LIBRARIES})
```

Delving deeper, we construct two essential Gazebo plugins: gazeborosgps.cpp and gazeboUAMpropulsion.cpp. These plugins, accompanied by their corresponding header files in the src and include folders, are pivotal components that empower our simulation.

```
#include <algorithm>
#include <assert.h>
#include "ider_drone/gazebo_UAM_propulsion.h"

namespace gazebo
```

```
{

  // Register this plugin with the simulator
  GZ_REGISTER_MODEL_PLUGIN(ModelJointControler);

  // Constructor
  ModelJointControler::ModelJointControler()
  {
    this->force_msg_.prop1=0;
    this->force_msg_.prop2=0;
    this->force_msg_.prop3=0;
    this->force_msg_.prop4=0;

    this->temp_msg_.prop1=0;
    this->temp_msg_.prop2=0;
    this->temp_msg_.prop3=0;
    this->temp_msg_.prop4=0;
  }

  ////////////////////////////////////////////////////////////////////////////////

  this->prop1_speed_magn = this->temp_msg_.prop1/10;
    this->prop2_speed_magn = this->temp_msg_.prop2/10;
    this->prop3_speed_magn = this->temp_msg_.prop3/10;
    this->prop4_speed_magn = this->temp_msg_.prop4/10;
    this->force_msg_.prop1 = this->temp_msg_.prop1/146.28;
    this->force_msg_.prop2 = this->temp_msg_.prop2/146.28;
    this->force_msg_.prop3 = this->temp_msg_.prop3/146.28;
    this->force_msg_.prop4 = this->temp_msg_.prop4/146.28;



  }
```

The gazeborosgps plugin breathes life into our model by providing crucial GPS coordinates within the Gazebo environment. On the other hand, the gazeboUAMpropulsion plugin undertakes the responsibility of propelling our drone model, interpreting motor commands and transforming them into the requisite forces and rotations.

```
// Copyright (c) 2012, Johannes Meyer, TU Darmstadt
// All rights reserved.

#include "ider_drone/gazebo_ros_gps.h"
#include <gazebo/physics/physics.hh>

// WGS84 constants
```

```
static const double equatorial_radius = 6378137.0;
static const double flattening = 1.0/298.257223563;
static const double excentrity2 = 2*flattening - flattening*flattening;

// default reference position
static const double DEFAULT_REFERENCE_LATITUDE  = 49.9;
static const double DEFAULT_REFERENCE_LONGITUDE = 8.9;
static const double DEFAULT_REFERENCE_HEADING   = 0.0;
static const double DEFAULT_REFERENCE_ALTITUDE  = 0.0;

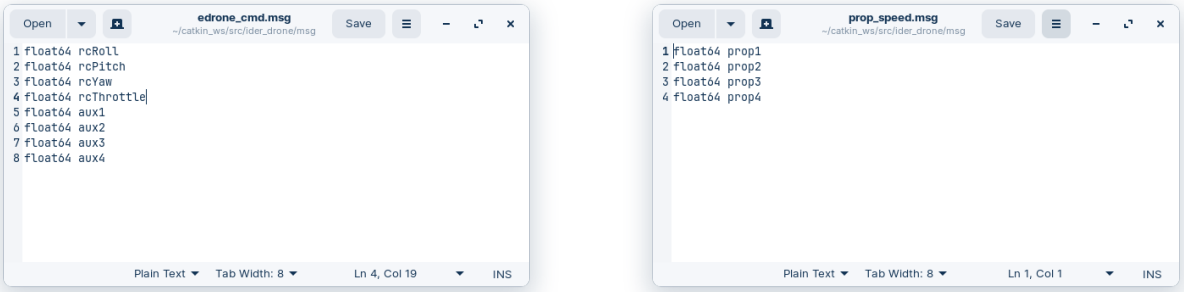
namespace gazebo {

GazeboRosGps::GazeboRosGps()
{
}

////////////////////////////////////////////////////////////////////////////////

} // namespace gazebo
```

To facilitate seamless communication between components, we meticulously craft custom message definitions. These definitions encompass the specialized needs of our project, including dronecmd.msg and propspeed.msg. These messages encapsulate the nuanced parameters required for propeller speed control and precise command execution.



(a) Drone Command message (b) Propeller speed message

Figure 4.4: Custom messages used in our system

Our ROS package's capabilities materialize through meticulously written Python scripts that commandeer ROS nodes.

Adding gazebo propulsion plugin

we added this gazebo plugin to our URDF file, This plugin is designed to apply forces to specific bodies in your drone model and communicate with the drone's control system through PWM (Pulse Width Modulation) signals.

```
<gazebo>
      <plugin name="gazebo_UAM_propulsion" filename=
      "libgazebo_UAM_propulsion.so">
      <namespace_model>/ider_drone</namespace_model>
      <activate_pid_control>no</activate_pid_control>
      <prop_kp> 1 </prop_kp>
      <prop_ki> 0 </prop_ki>
      <prop_kd> 0.0 </prop_kd>
      <robotNamespace>ider_drone</robotNamespace>
      <bodyName_1>prop1</bodyName_1>
      <bodyName_2>prop2</bodyName_2>
      <bodyName_3>prop3</bodyName_3>
      <bodyName_4>prop4</bodyName_4>
      <topicName>pwm</topicName>
    </plugin>
  </gazebo>
```

4.2.4 URDF File Result: full robot description

we provide a comprehensive overview of the robot description and simulation setup used for our project.

body parts URDF example

this is the base frame Description :

```
<link
name="base_frame">
<inertial>
  <origin
    xyz="-6.4724E-10 -3.7543E-06 0.022833"
    rpy="0 0 0" />
  <mass
    value="0.9" />
  <inertia
    ixx="0.0003491"
    ixy="6.7789E-12"
    ixz="3.2589E-13"
    iyy="0.000393"
    iyz="-8.8638E-09"
    izz="0.00072092" />
</inertial>
<visual>
  <origin
    xyz="0 0 0"
    rpy="0 0 0" />
```

```
      <geometry>
        <mesh
          filename="package://ider_drone/meshes/base_frame.STL" />
      </geometry>
      <material
        name="">
        <color
          rgba="0.79216 0.81961 0.93333 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://ider_drone/meshes/base_frame.STL" />
      </geometry>
    </collision>
  </link>
```

Arm Links and Joints URDF example

this is the link2 part of our robotic arm with the attached joint 8

```
      <link
    name="link_2">
    <inertial>
      <origin
        xyz="-0.0339222684392758 -4.93952538593856E-10
        1.87809190139337E-12"
        rpy="0 0 0" />
      <mass
        value="0.001" />
      <inertia
        ixx="9.58071469646005E-07"
        ixy="2.28285313525667E-14"
        ixz="3.56598610678052E-15"
        iyy="1.56516252094845E-05"
        iyz="-1.82753007746996E-15"
        izz="1.57924064468631E-05" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
```

```
      <geometry>
        <mesh
          filename="package://ider_drone_pkg/meshes/link_2.STL" />
      </geometry>
      <material
        name="">
        <color
          rgba="0.792156862745098 0.819607843137255
          0.933333333333333 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://ider_drone_pkg/meshes/link_2.STL" />
      </geometry>
    </collision>
  </link>
  <joint
    name="j_8"
    type="continuous">
    <origin
      xyz="0 0 0.025"
      rpy="-1.5708 -0.033321 0" />
    <parent
      link="link_1" />
    <child
      link="link_2" />
    <axis
      xyz="0 0 -1" />
  </joint>
```

Control URDF example

by adding this the joints are turned into actuators

```
    <transmission name="j_7_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j_7">
      <hardwareInterface>hardware_interface/PositionJointInterface
</hardwareInterface>
      </joint>
      <actuator name="j_7_motor">
```

```
        <hardwareInterface>hardware_interface/PositionJointInterface
 </hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
        </actuator>
</transmission>


    <transmission name="j_8_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="j_8">
        <hardwareInterface>hardware_interface/PositionJointInterface
 </hardwareInterface>
        </joint>
        <actuator name="j_8_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface
 </hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
        </actuator>
</transmission>


  <transmission name="j_9_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="j_9">
        <hardwareInterface>hardware_interface/PositionJointInterface
 </hardwareInterface>
        </joint>
        <actuator name="j_9_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface
 </hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
        </actuator>
</transmission>
```

4.2.5 Creating a new Manipulator Package to control the Robotic Arm URDF using Moveit

Moveit is a great open source Robot manipulation tool which can be used to create packages, which can be used to manipulate any Robot in ROS.

We will be using ROS 1 and Moveit 1 for this Moveit is installed by default with the Full Desktop Version of any ROS distribution.

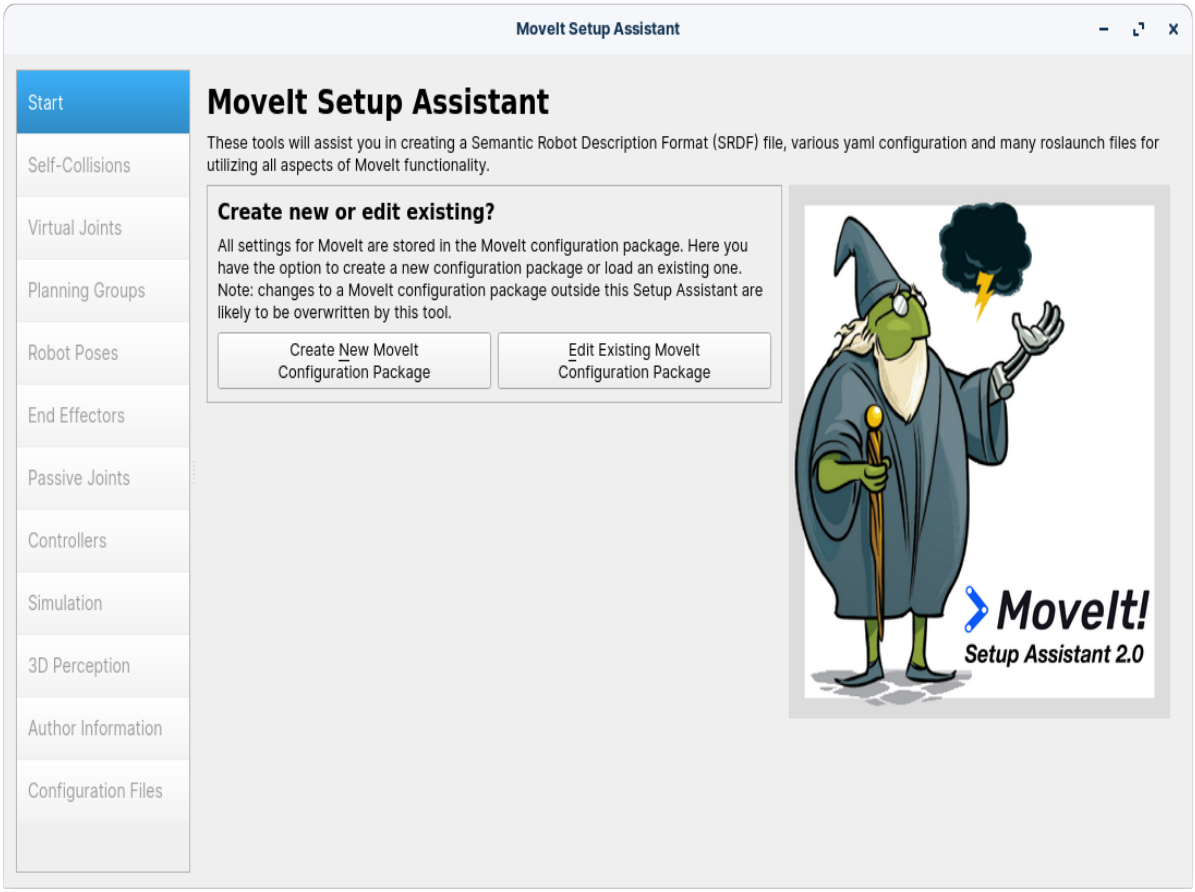


Figure 4.5: moveit setup assistant menu

following these steps we successfully generated our package Moveit with a launch file to spawn and control our robotic arm :

- Creating the Moveit Package using Moveit Setup Assistant to control our URDF file.
- Testing the Moveit Package Generated using Moveit Setup Assistance.
- Configuring the Default Moveit Package to work properly.
- Writing a new ROS Controller for connecting the joint trajectory controller with ROS manipulator or Moveit package.
- Edit the “moveitcontrollermanager.launch.xml” file available in the launch folder of your moveit package.
- Create a new launch file to launch Moveit simulation in Rviz and Gazebo.
- adding robot arm poses and joint groups.

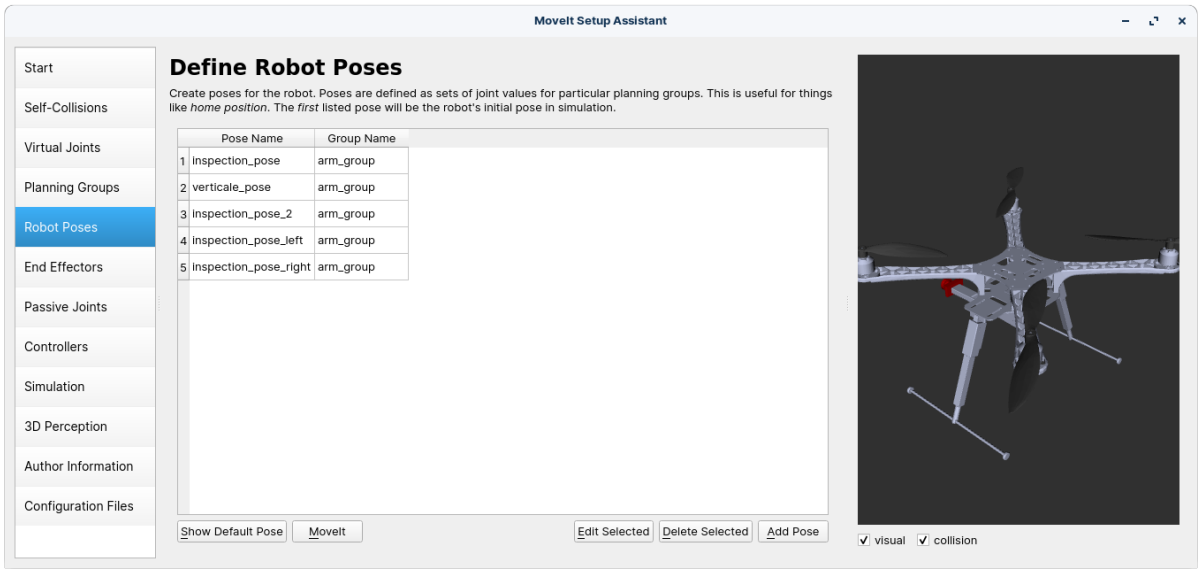


Figure 4.6: custom robot poses 'pick pose etc ...'

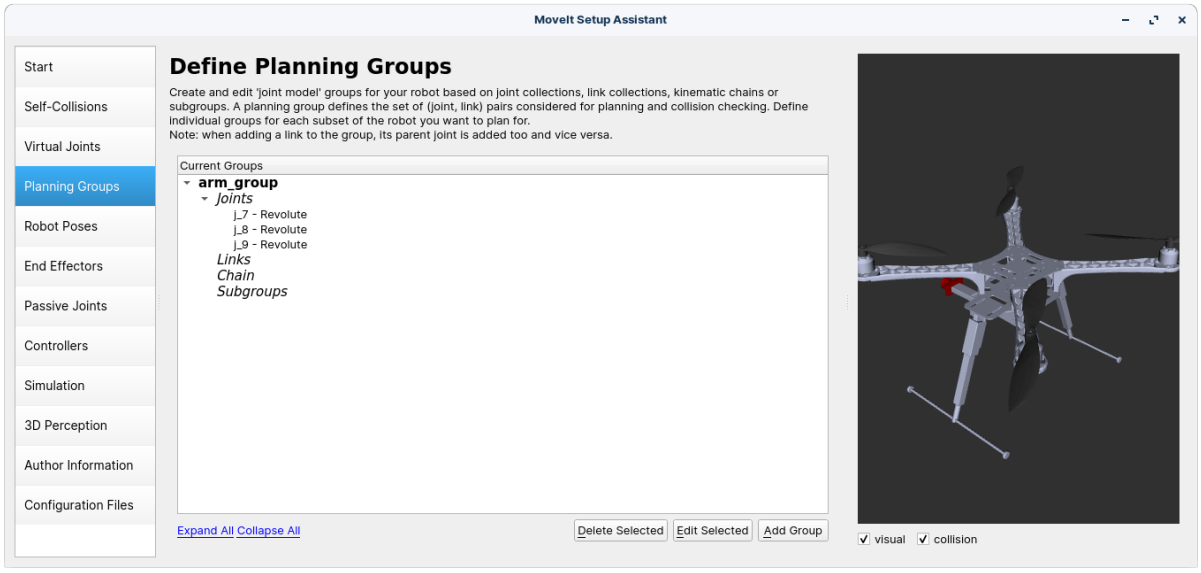


Figure 4.7: joints groups 'arm group etc ...'

- select a kinematic solver for our robot in our case KDL.

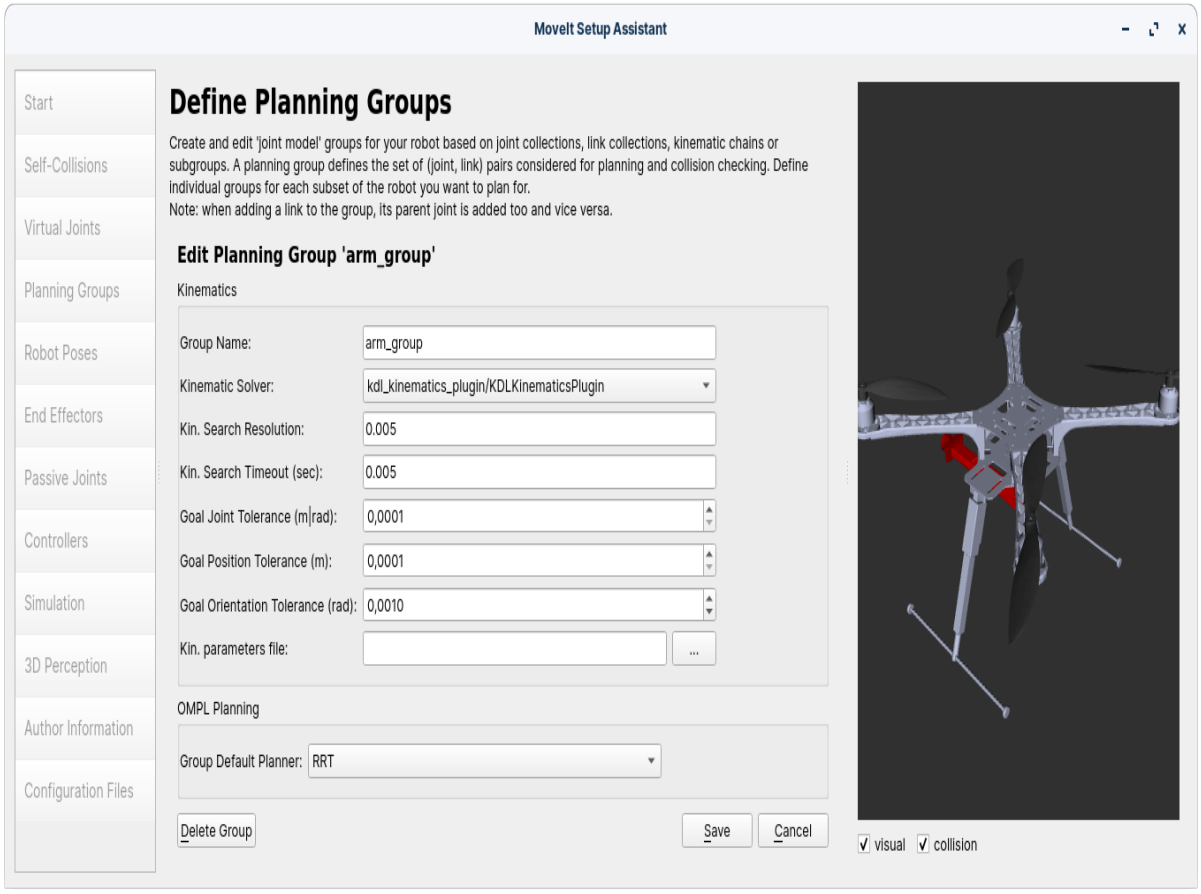


Figure 4.8: kdl kinematic solver plugin.

- Build our catkin workspace.
- the package moveitdrone is ready to use.

4.3 ROS Package Validation: RViz and Gazebo

The culmination of our modeling efforts resulted in the creation of a ROS package. This package includes launch files that enable visualization and simulation of the robotic system in both RViz and Gazebo environments. Through the launch files, stakeholders can observe the system's behavior and interactions, simulating real-world scenarios for analysis and testing.

4.3.1 Visulize the Model in both Rviz and Gazebo

You can see in Figure 4.9 , 4.10 that our model was successfully spawned in Gazebo and Rviz.

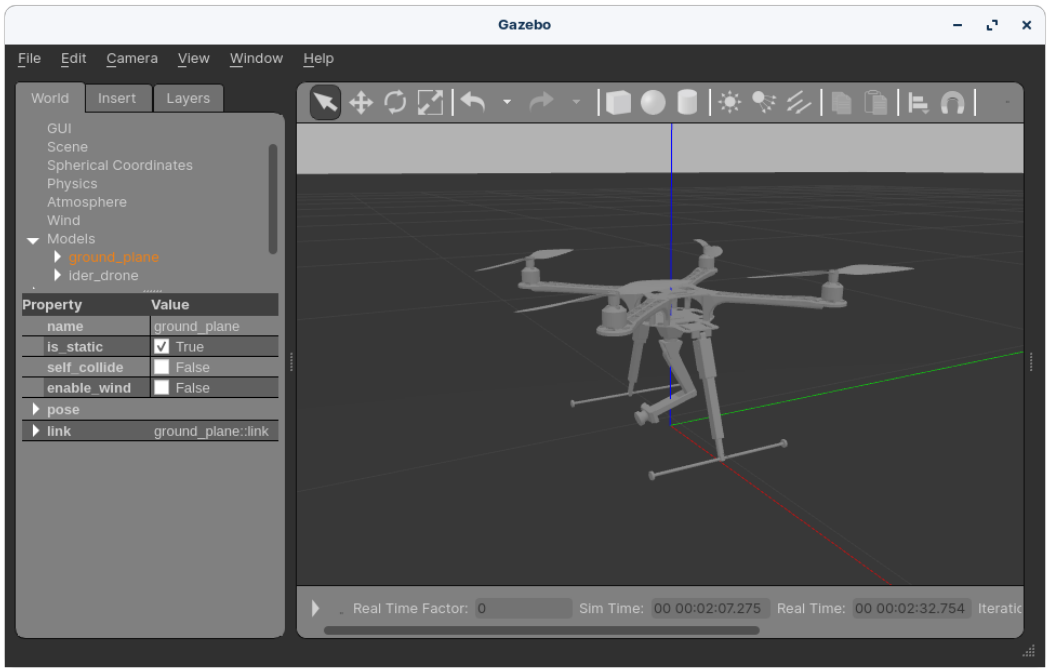
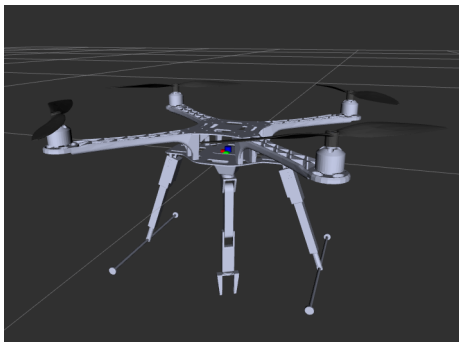
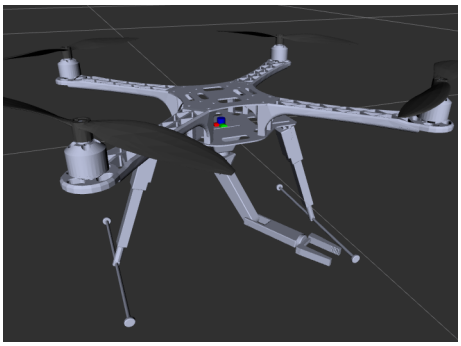


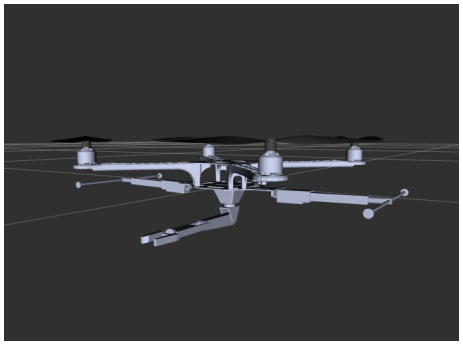
Figure 4.9: The view of the system in Gazebo.



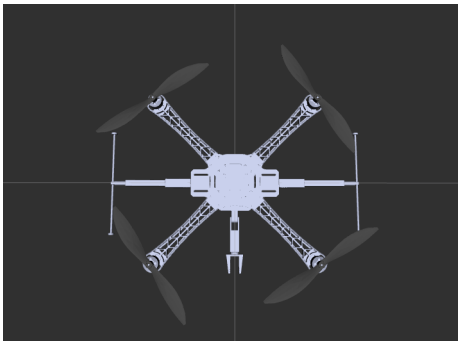
(a) View 1



(b) View 2



(c) Stand Open View



(d) Top view

Figure 4.10: Rviz views

4.3.2 Conclusion

The transition from SolidWorks to ROS, facilitated by the URDF format, marked a pivotal moment in the project. This transformation allowed us to bridge the gap between the physical model and the virtual ROS environment. The careful preparation and packaging of the ROS components ensured that our robot model could seamlessly integrate into the ROS ecosystem.

Furthermore, the validation of our ROS package using RViz and Gazebo provided essential verification steps. These visualization tools allowed us to assess the accuracy of our model, enabling us to identify and rectify any discrepancies.

In essence, this section has illuminated the process of translating a physical robotic system into a digital ROS representation. It underscores the importance of precision, attention to detail, and the critical role that visualization and validation tools play in ensuring the successful integration of a robot into the ROS framework.

Chapter 5

Control of the Quadcopter-Manipulator in ROS/Gazebo Environment

Introduction

With this package, it is now possible to develop both low and high-level controllers for the quadcopter. In order to have a deeper approach to the control strategies used, it is important to differentiate a low and a high-level controller, but first we have to keep in mind the project objectives introduced in Chapter 1. The main three objectives are: to build a platform to test control algorithms; to design an architecture able to assign directly the motor velocities; and to make it possible to define a trajectory to be followed by the UAV. So far, the second objective has already been achieved in Chapter 3. The relationship between the first two objectives and the last one is exactly the relation between a low-level and a high-level controller. While the first focuses on performing all the calculations to stabilize the UAV on air, the second one focuses on giving inputs to the system that will be used by the low-level controller, in order to achieve its objective to follow a given trajectory. In other words, the high-level controller will provide references that would be used by the low-level controller, in order to achieve the desired position in space.

5.1 Gazebo Simulator

Gazebo is a popular and powerful open-source 3D robotics simulator that is widely used in the robotics community. It is often integrated with ROS (Robot Operating System) to create realistic simulations of robots and robotic systems. The combination of Gazebo and ROS provides a powerful platform for developing, testing and evaluating robotics algorithms and systems in a virtual environment before deploying them on real hardware.

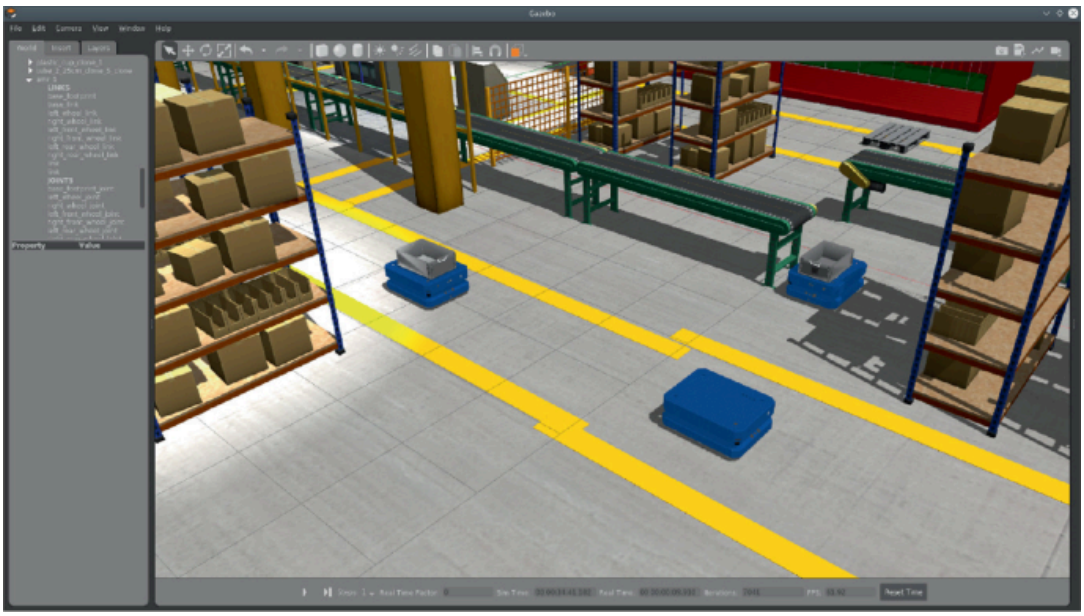


Figure 5.1: Gazebo Simulator.

ROS offers a range of packages and plugins to interface with Gazebo, allowing seamless communication between the simulation environment and ROS nodes. ROS nodes can

publish sensor data (e.g., camera images, lidar scans) and control commands to the simulated robot model in Gazebo. Similarly, Gazebo can simulate the physics and dynamics of the robot, providing feedback to the ROS control system for closed-loop control.

The integration of Gazebo with ROS offers several advantages:

Realistic simulations: Gazebo provides accurate physics-based simulations, allowing developers to test and validate algorithms in a virtual environment that closely resembles real-world scenarios.



Figure 5.2: Gazebo Virtual Environment.

Safety and cost-effectiveness: Running simulations in Gazebo before deploying on real robots reduces the risk of hardware damage and avoids unnecessary costs during the development and testing phase.

Development and testing platform: Gazebo with ROS allows rapid prototyping, testing, and evaluation of various robot behaviors, control algorithms, and perception systems.

Sensor and actuator simulation: Gazebo can simulate various sensors (e.g., cameras, lidars, IMUs) and actuators (e.g., motors, servos), enabling comprehensive testing of the robot's perception and control systems.

Community support: Gazebo and ROS have large and active communities, providing access to a wealth of resources, plugins, and models contributed by the robotics community.

Overall, the combination of Gazebo with ROS enhances the capabilities of the robotics development process, enabling researchers and developers to build sophisticated robotic systems efficiently and effectively.

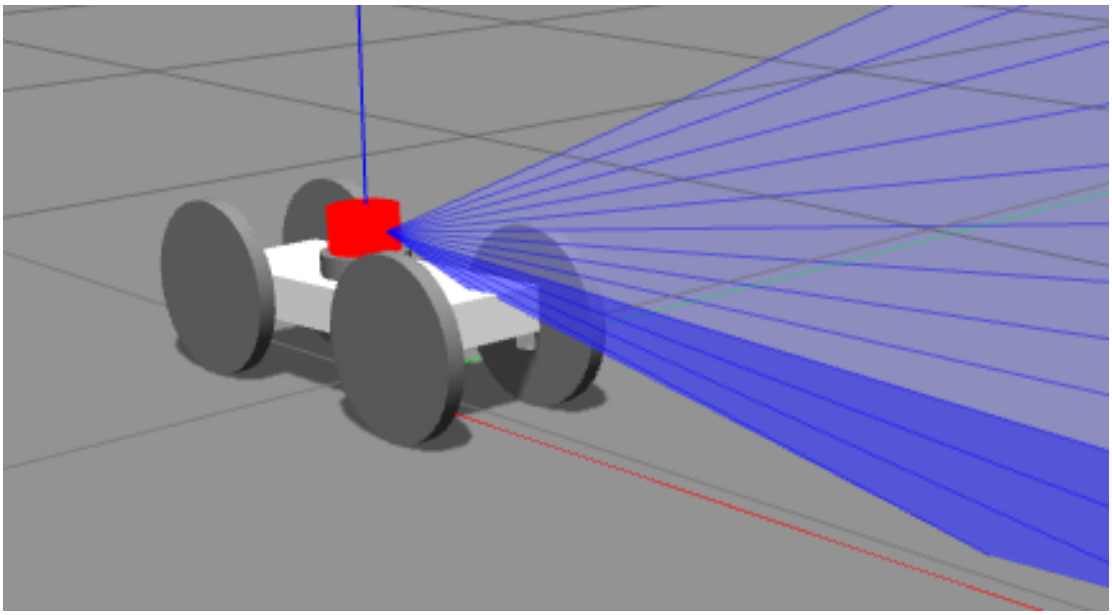


Figure 5.3: Presents a Gazebo Simulation of LIDAR sensor attached to a mobile robot.

5.1.1 Sensors provided by Gazebo

Gazebo offers various types of sensors, including:

- IMU (Inertial Measurement Unit): Simulated IMUs provide information about linear and angular accelerations, as well as angular rates. These are essential for attitude estimation and control.
- GPS Sensor: Gazebo's GPS sensor models simulate position and velocity data, which are valuable for position control and navigation.
- Camera and LiDAR: Simulated cameras and LiDAR sensors offer visual and depth information, enabling perception-based tasks such as object detection, obstacle avoidance, and environment mapping.
- Force and Torque Sensors: These sensors mimic force and torque measurements, facilitating contact detection, collision responses, and force feedback control.

To incorporate sensors into the simulation, we followed these steps:

1. Add Sensor Definitions: Within the robot's URDF file, we added sensor definitions using the appropriate URDF tags. For instance, for an IMU sensor, we used the '<imu>' tag, and for a camera sensor, we used the '<camera>' tag.

2. Plugin Configuration: To enable sensor data exchange between the simulated environment and our control system, we configured plugin elements within the URDF. These plugins establish the communication channels and provide sensor data to the control system.

3. Sensor Properties: We adjusted the sensor properties within the URDF, such as noise levels, update rates, and field of view, to simulate real-world sensor characteristics accurately.

4. Simulation Setup: With the updated URDF, we launched the simulation environment in Gazebo. The sensor plugins in the URDF communicated with Gazebo's physics engine, allowing us to access sensor data in our control algorithms.

Camera Gazebo Plugin :

```
<gazebo reference="link_4_camera">
  <sensor type="camera" name="camera_camera_sensor">
    <update_rate>30.0</update_rate>
    <camera>
      <pose>-0.05 0 0 3.14159265359 1.57079632679 0</pose>

      <image>
        <format>R8G8B8</format>
        <width>1020</width>
        <height>1020</height>
      </image>
      <clip>
        <near>0.01</near>
        <far>100</far>
      </clip>
    </camera>

    <plugin name="camera_camera_controller" filename=
    "libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>arm_sensor/camera</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>link_4_camera</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

we can see more detail about the camera plugin here : https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#Camera

IMU Gazebo Plugin :

```
<gazebo>
    <plugin name="imu_plugin" filename="libgazebo_ros_imu.so">
            <updateRate>60</updateRate>
            <alwaysOn>true</alwaysOn>
            <bodyName>base_frame</bodyName>
            <frameName>base_frame</frameName>
            <topicName>/ider_drone/imu/data</topicName>
            <serviceName>imu_service</serviceName>
            <gaussianNoise>0.0</gaussianNoise>
            <updateRate>20.0</updateRate>
            <xyzOffset>0 0 0</xyzOffset>
            <rpyOffset>0 0 0</rpyOffset>
    </plugin>
</gazebo>
```

we can see more detail about the imu plugin here : https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#IMU(GazeboRosImu)

GPS Gazebo Plugin :

```
<gazebo>
     <plugin name="gps_plugin" filename="libgazebo_ros_gps.so">
            <updateRate>60</updateRate>
            <robotNamespace>ider_drone</robotNamespace>
            <bodyName>base_frame</bodyName>
            <referenceLatitude>19</referenceLatitude>
            <referenceLongitude>72</referenceLongitude>
            <referenceHeading>0</referenceHeading>
            <referenceAltitude>0</referenceAltitude>
            <topicName>gps</topicName>
            <velocityTopicName>gps_velocity</velocityTopicName>
    </plugin>
</gazebo>
```

we can see more detail about more plugins here : https://github.com/tu-darmstadt-ros-pkg by Technische Universität Darmstadt.

5.1.2 Computational Power

In modern control systems, computational resources play a vital role in achieving real-time and accurate control. Our control system, implemented using Gazebo and ROS, demands significant computational power to handle the complex calculations involved in position and attitude control. Here, we discuss the computational resources we utilized for our control system.

Hardware Configuration

Our control setup was executed on a high-end laptop equipped with cutting-edge hardware components. The hardware configuration consisted of:

- **CPU:** Intel Core i7 processor with multiple cores and high clock speed, enabling efficient parallel computation.
- **GPU:** NVIDIA RTX graphics processing unit, which accelerated rendering tasks in Gazebo and offloaded computationally intensive calculations.
- **Memory:** 16Gb RAM capacity to handle data exchange between different components of the control system.

Software Frameworks

Our control system relied on the following software frameworks:

- **Gazebo:** A physics-based simulator for dynamic systems, allowing us to simulate and visualize the behavior of our controlled platform.
- **ROS (Robot Operating System):** A flexible framework for building robotic systems, facilitating communication between various modules and devices.

Computational Demand

The cascaded PID control strategy, coupled with the dynamics of our system, imposed a substantial computational demand. The system required real-time data processing, sensor fusion, and control signal generation. Our high-end hardware configuration, including the powerful CPU and GPU, enabled us to meet these demands while maintaining a responsive control system.

Considerations

While our high-end laptop with its powerful CPU and GPU provided the computational muscle required for our control system, it's essential to consider the feasibility of porting the control solution onto a microcontroller. Microcontrollers offer a compact and energy-efficient platform for control systems, and there are microcontrollers available that can handle ROS and the demands of our cascaded PID control approach.

Utilizing a microcontroller introduces several key considerations:

- Processing Power: Selecting a microcontroller with adequate processing power to handle the control algorithms and ROS communication is crucial. Multi-core microcontrollers with high clock speeds are preferable for real-time control.
- Memory and Storage: Sufficient memory and storage capacity are required to store control parameters, handle data exchange, and accommodate ROS-related tasks.

- ROS Compatibility: Choosing a microcontroller that supports ROS integration is essential. This might involve using ROS-compatible libraries or middleware that enable communication between the microcontroller and the ROS ecosystem.
- I/O Ports: Microcontrollers should offer the necessary I/O ports to connect sensors, actuators, and communication interfaces required by the control system.
- Energy Efficiency: Microcontrollers designed for energy efficiency can prolong the operation of battery-powered systems, making them suitable for applications where power consumption is a concern.
- Code Optimization: Given the limited resources of microcontrollers compared to high-end laptops, efficient code optimization is essential to ensure the control algorithm runs smoothly within the microcontroller's constraints.

Adapting the control system to a microcontroller introduces challenges, but it can lead to more compact and autonomous implementations, especially for applications where portability and energy efficiency are paramount.

5.2 Control Strategy

The aim of this project is to design controllers that will control the Drone orientation as well as position in the Gazebo environment.

The project is divided into 2 sub-tasks:

Task A - Designing attitude controller for the Drone Figure : 5.6

Task B - Designing position controller for the Drone Figure : 5.5

In total 2 PID algorithms were applied to control the flight of the drone. The diagram of the cascaded control system is shown below,

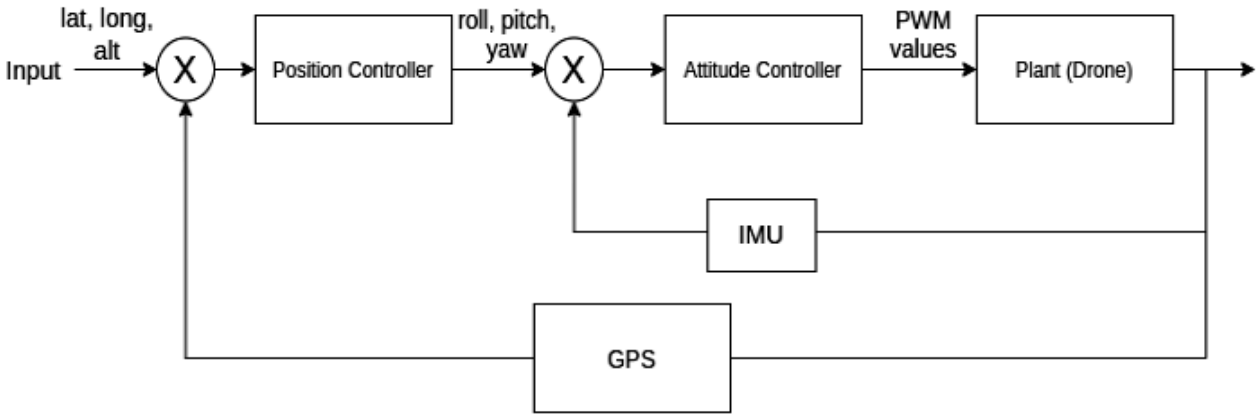


Figure 5.4: Cascaded control system

This project works on concepts of control systems, path planning, image processing, and algorithm development. Tools such as the Robot Operating System, robotics simulator Gazebo, the Python programming language, and many of its libraries are used.

5.2.1 Control Scheme: Position Controller

The main task of the position controller is to give the required drone orientation to reach the required setpoint. The required orientation is calculated using a PID controlled algorithm. The orientation is published in quaternion format which ranges from 1000 to 2000. 1000 corresponds to -10 degrees and 2000 corresponds to 10 degrees and all the angles between -10 and 10 degrees can be found accordingly.

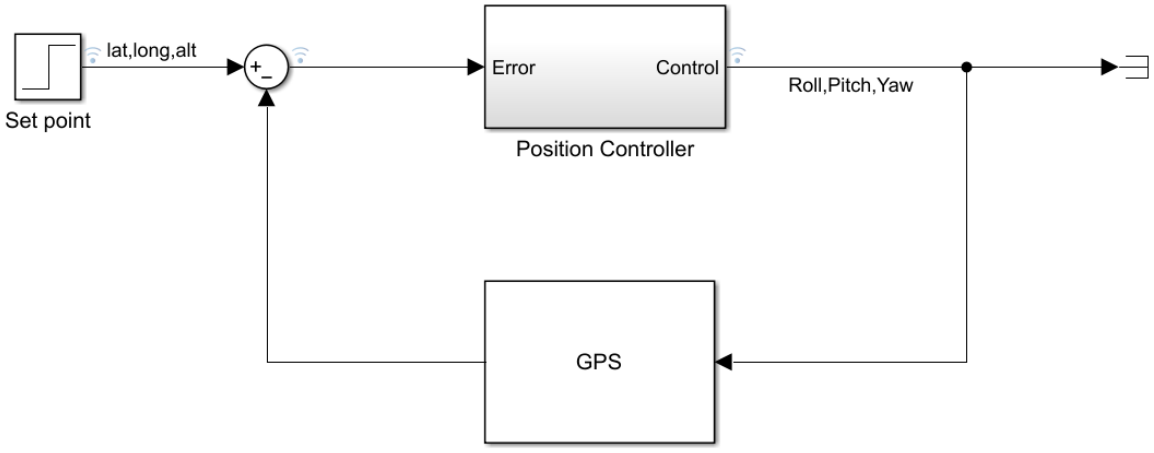


Figure 5.5: Position Controller

5.2.2 Control Scheme: Attitude Controller

The main task of the attitude controller is to keep the drone in the required orientation which is given by the position controller. This is done by another PID-controlled algorithm. After calculating the required PID values throttle is given to each motor of the drone so that it flies in the required orientation. These values are then converted to the required PWM format that could be given to motors.

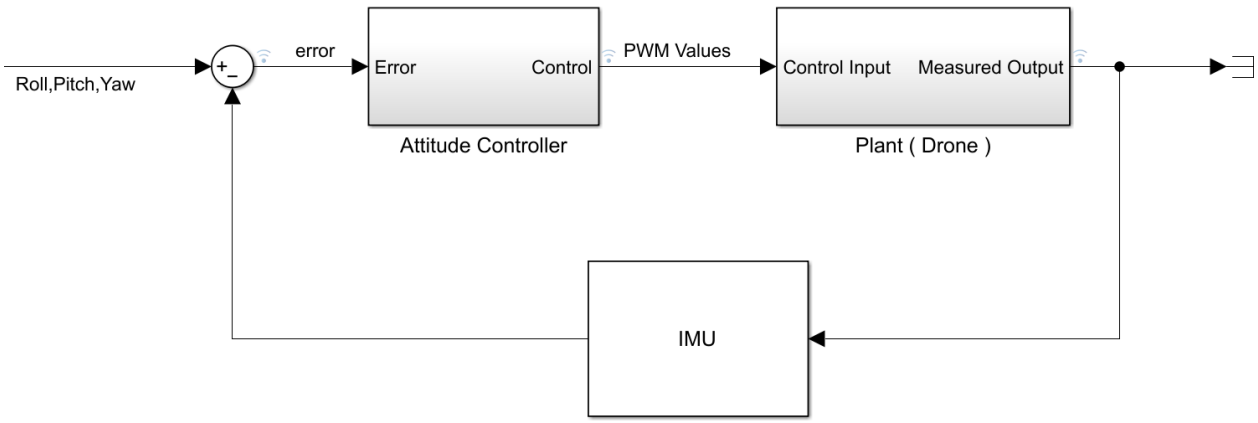


Figure 5.6: Attitude Controller

5.2.3 Control Algorithm using Python scripts

The altitudecontroller.py and positioncontroller.py scripts, fervently executing these nodes, play pivotal roles in our project's execution. The altitudecontroller.py script orchestrates a node that subscribes to sensor data from the Inertial Measurement Unit (IMU) and GPS. The outcome of this computation is then propagated as pulse-width modulation (PWM) values.

```
from ider_drone.msg import *
from pid_tune.msg import PidTune
from sensor_msgs.msg import Imu
from std_msgs.msg import Float32
import math
import rospy
import time
import tf


class Edrone():
    """docstring for Edrone"""
    def __init__(self):
        rospy.init_node('attitude_controller') # initializing
        ros node with name attitude_control

        # This corresponds to your current orientation of
        eDrone in quaternion format. This value must be
        updated each time in your imu callback
        # [x,y,z,w]
        self.drone_orientation_quaternion = [0.0, 0.0, 0.0, 0.0]

        self.lastTime=0.0 # To store the last time

        # This corresponds to your current orientation
```

```
of eDrone converted in euler angles form.
# [r,p,y]
self.drone_orientation_euler = [0.0, 0.0, 0.0]

# This is the setpoint that will be received
from the drone_command in the range from 1000 to 2000
# [r_setpoint, p_setpoint, y_setpoint]
self.setpoint_cmd = [0.0, 0.0, 0.0]
self.setpoint_throttle = 0.0  # Setpoint for
throttle

# The setpoint of orientation in euler angles
at which you want to stabilize the drone
# [r_setpoint, p_psetpoint, y_setpoint]
self.setpoint_euler = [0.0, 0.0, 0.0]

# Declaring pwm_cmd of message type prop_speed
and initializing values

self.pwm_cmd = prop_speed()
self.pwm_cmd.prop1 = 0.0
self.pwm_cmd.prop2 = 0.0
self.pwm_cmd.prop3 = 0.0
self.pwm_cmd.prop4 = 0.0

# Declaring roll_error,pitch_error and yaw_error
of message type Float32 and initializing values

#self.roll_error = Float32()
#self.pitch_error = Float32()
#self.yaw_error = Float32()

#self.roll_error.data = 0.0
#self.pitch_error.data = 0.0
#self.yaw_error.data = 0.0


# initial setting of Kp, Kd and ki for [roll, pitch, yaw]

self.Kp = [0.2, 0.1, 201.0]
self.Ki = [0, 0, 0]
self.Kd = [0.2, 0.2, 0.5]


////////////////////////////////////
```

```
    if __name__ == '__main__':

e_drone = Edrone()
r = rospy.Rate(e_drone.sample_frequency)
# specify rate in Hz based upon your desired
PID sampling time, i.e. if desired sample time
is 33ms specify rate as 30Hz
rospy.loginfo("started")
while not rospy.is_shutdown():
    e_drone.pid()
    r.sleep()
```

This python file runs a ROS-node of name attitude control which controls the roll pitch and yaw angles of the eDrone. This node publishes and subscribes to the following topics:

PUBLICATIONS	SUBSCRIPTIONS
/roll_error	/pid_tuning_altitude
/pitch_error	/pid_tuning_pitch
/yaw_error	/pid_tuning_roll
/ider_drone/PWM	/ider_drone/imu/data
	/drone/drone_command

Simultaneously, the positioncontroller.py script spearheads a node that harmoniously integrates GPS data—position and altitude—into our project. This data is then distilled into a dronecmd message that is communicated via the /dronecommand topic.

```
from ider_drone.msg import edrone_cmd
from std_msgs.msg import Float32
from pid_tune.msg import PidTune
from sensor_msgs.msg import NavSatFix
import math
import rospy
import time


class Edrone():

    def __init__(self):
        rospy.init_node('position_controller')
        # initializing ros node with
        name position_control

        # Coordinates of setpoints stored in a list
        #self.setpoints = [[19.0,72.0,4.05],[19.0,72.0,8.0]
        ,[19.0,71.99995,8.0],
        [19.0,72.0,8.0],[19.0,72.0,4.05]]
```

```
        #self.setpoints = [[19.0,72.0,4.05],[19.0,72.0,7.0]
        ,[18.99990,72.0,7.0],
        [19.0,72.0,7.0],[19.0,72.0,4.05]]

        self.setpoints = [[19.0,72.0,4.05],[19.0,72.0,7.0]
        ,[18.99995,72.0,7.0],
        [18.99995,71.99995,7.0],[19.0,71.99995,7.0],[19.0,72.0,7.0]
        ,[19.0,72.0,4.05]]

        self.lastTime=0.0 # To store the last time

        self.r=0 # To store value of the current index number

        # List for storing current [latitude, longitude, altitude]
        self.current_coord = [0.0,0.0,0.0]

        # Variables for storing errors in latitude
        , longitude, and altitude
        #self.x_error = Float32()
        #self.y_error = Float32()
        #self.z_error = Float32()

        #self.x_error.data = 0.0
        #self.y_error.data = 0.0
        #self.z_error.data = 0.0

        # Declaring drone_orientation of the message type
        edrone_cmd and
        initializing values

        self.drone_orientation = edrone_cmd()

        # initial setting of Kp, Kd, and ki for [latitude,
        longitude, altitude]

        self.Kp = [3000000, 4000000, 120.1]
        self.Ki = [0.0, 0.0, 0.0]
        self.Kd = [30000000, 40000000, 303.5]
////////////////////////////////////////////////
if __name__ == '__main__':

    e_drone = Edrone()
    r = rospy.Rate(e_drone.sample_frequency)  # specify
    the rate in Hz based upon
    your desired PID sampling time, i.e. if desired sample
    time is 33ms specify rate as 30Hz
```

```
rospy.loginfo("started")
i=0 # To pass index for setpoints
while not rospy.is_shutdown():

    i=e_drone.pid(i) # Current index number is passed
    and is returned after
    it is updated

    if(i==len(e_drone.setpoints)): # If index number
    becomes out of range then break
        break

    r.sleep()
```

5.2.4 Control Parameters for Cascaded PID Position and Attitude Control

In the design and implementation of control systems, the selection of appropriate control parameters is a pivotal factor that significantly influences the system's performance, stability, and robustness. we delve into the control parameters utilized in our cascaded Proportional-Integral-Derivative (PID) control setup for both position and attitude control. Specifically, we employ Proportional and Derivative (PD) parameters to achieve effective control.

Position and Attitude Control Parameters

The position control loop serves as the outer loop in our cascaded architecture and focuses on guiding the system to the desired spatial position. The parameters chosen for the position control loop play a crucial role in shaping the system's response.

Table 5.1: Control Parameters for Cascaded PID Position and Attitude Control

Control Loop	Parameter	Gain	
		Proportional (Kp)	Derivative (Kd)
Position Control	K_latitude	*3000000*	30000000
	K_longitude	*4000000*	40000000
	K_altitude	*120.1*	303.5
Attitude Control	K_roll	*0.2*	0.2
	K_pitch	*0.1*	0.2
	K_yaw	*201*	0.5

Nested within the position control loop, the attitude control loop maintains the desired orientation of the system. Proper selection of control parameters for attitude control is vital to achieving accurate and stable performance.

5.2.5 RQT Graph

In the context of ROS (Robot Operating System), the ”rqt graph” command is used to visualize the communication graph of a ROS system. It provides an overview of the nodes in the ROS system and the topics they are publishing and subscribing to. This visualization can be helpful in understanding how different components of a robotic system are interconnected and communicating with each other. we can see in the next page the full ros graph of our system.

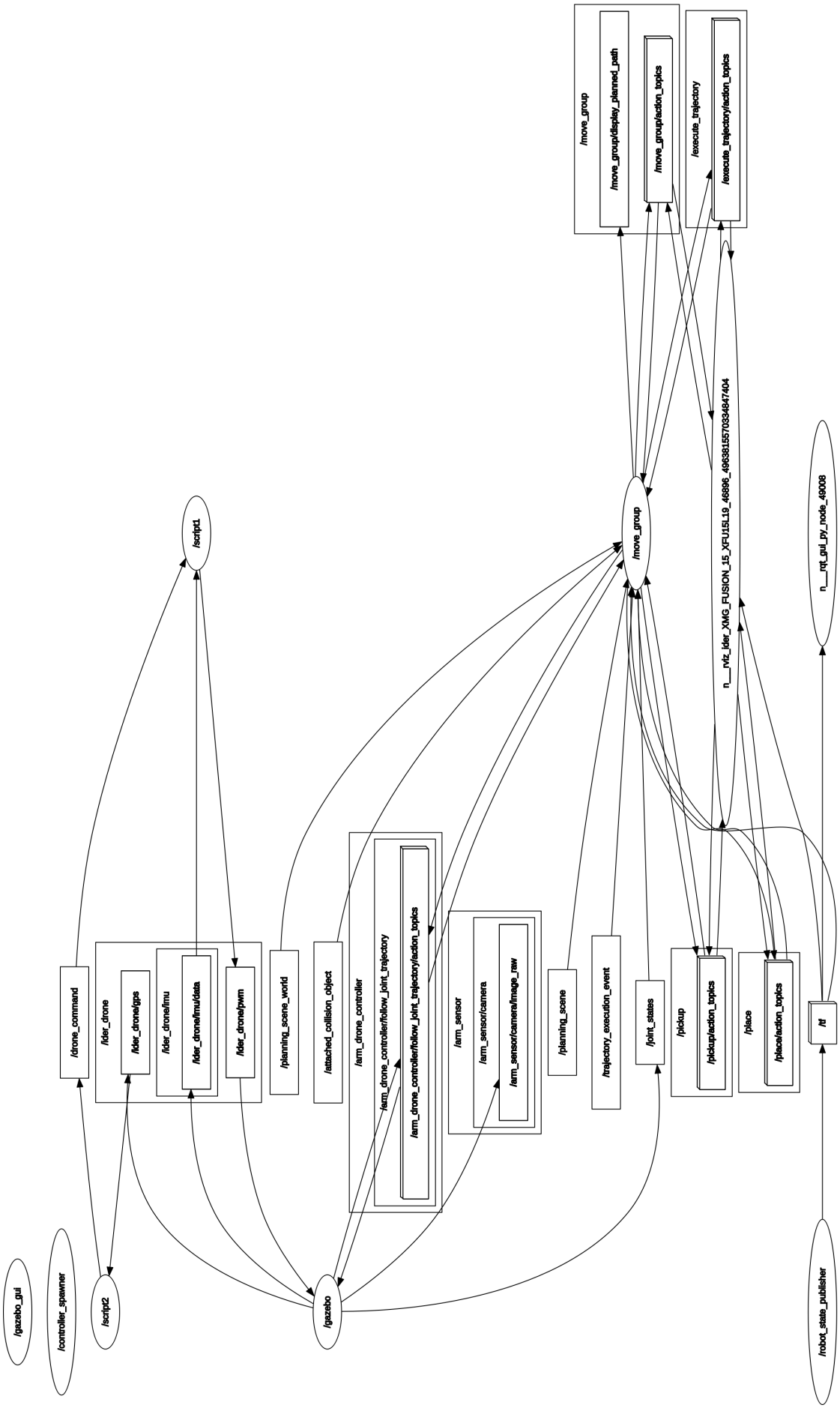


Figure 5.7: Rqt graph of the system

5.2.6 Tuning Methodology

The effectiveness of a control system hinges on the careful selection and tuning of control parameters. In our cascaded PID control setup for position and attitude control, we employed a manual tuning approach, utilizing tools like PlotJuggler to observe error signals and optimize the Proportional and Derivative gains

Manual Tuning Approach

Manual tuning involves iteratively adjusting the control parameters while observing the system's response to various inputs. This process allows us to strike a balance between desired responsiveness and stability. In our case, we used the following steps to manually tune our control parameters:

- **Error Signal Analysis:** We first analyzed the error signals for each control loop, such as roll error, pitch error, yaw error, as well as position error along the x, y, and z axes. These error signals highlight the discrepancies between the desired and actual system states.
- **Initial Parameter Estimation:** We began with initial estimates for the Proportional and Derivative gains. These estimates were based on our understanding of the system dynamics, but they were adjusted conservatively to avoid excessive oscillations or instability.
- **Step Responses:** We subjected the system to step inputs to observe its response. Using PlotJuggler, we visualized the system's behavior in real time, focusing on factors such as overshoot, settling time, and stability.
- **Parameter Adjustments:** Based on the observed responses, we made controlled adjustments to the control parameters. For instance, if we observed excessive overshoot, we reduced the Proportional gain to dampen the response. Conversely, if the system exhibited sluggish behavior, we increased the Proportional gain for greater responsiveness.
- **Fine-Tuning:** We iterated this process, making incremental adjustments to the control parameters while closely monitoring the system's behavior. This fine-tuning phase helped us reach an optimal parameter configuration that balanced stability and performance.

Conclusion

In conclusion, the "Control: ROS and Gazebo" section of this project has been a thorough exploration of the essential components that underlie effective robotic control within a simulated environment. Throughout this journey, we've gained valuable insights into key aspects, including control strategies, software simulations, sensor integration, computational considerations, and control parameter tuning.

Gazebo, as a dynamic simulator, has played a pivotal role in our research, providing a versatile platform for testing and refining control algorithms and strategies. The integration of various sensors, such as the camera, IMU, and GPS Gazebo plugins, has enhanced our simulation environment, allowing for the realistic assessment of sensor data and its impact on control.

Our scrutiny of computational power, encompassing both hardware configuration and software frameworks, has been meticulous. These considerations are paramount in ensuring the efficiency and effectiveness of our control algorithms, operating within the constraints of computational resources.

We've delved into various control strategies, including position and attitude control schemes, demonstrating the versatility and adaptability of these strategies within a simulated context. The implementation of control algorithms using Python scripts has exemplified our ability to customize and fine-tune control methodologies to meet specific requirements.

Moreover, we've highlighted the critical importance of control parameter tuning, a process that significantly influences the robot's behavior and performance. Our exploration of the manual tuning approach underscores the iterative nature of parameter adjustments in achieving optimal control.

In summary, this section has provided a comprehensive understanding of control aspects within the ROS and Gazebo environment. It emphasizes the significance of robust control strategies, seamless sensor integration, computational efficiency, and precise parameter tuning for achieving effective robotic control. These insights will undoubtedly guide and empower future advancements in the field of robotic control systems, contributing to the progress of robotics research and practical applications.

Chapter 6

Solar Panels field inspection using Quadcopter-Manipulator

6.1 Computational power

All the simulations were performed using an XMG FUSION 15. The laptop has great computational power containing an Intel I7-9750H 2.6 GHz six-core processor and 16 GB of RAM. Also, the computer has a dedicated graphical card, an NVIDIA RTX 2070 with 8GB DDR6. To be able to perform all the simulations with ROS, a Linux Ubunutu 20.04 was installed alongside with Windows 10 (dual-boot). To execute all the evaluations and analysis of the results RQT/Plotjuggler was used, and all the simulations were done in Gazebo software.

6.2 System Usage Guide

Before we begin our simulation, we need to illustrate how the package works and how to test it. We do this by echoing any data and instructions delivered using ros definitions such as topics and nodes. To perform the work, we may also utilize programs such as rqt, rqt console, and Plotjuggler[**29**].

6.2.1 Launching the simulation

Before we begin, we ensure that the following requirements are met:

- A functional ROS (Robot Operating System) environment.
- The ”moveitdrone” package is installed in your ROS workspace.

1. Open a terminal and navigate to ROS catkin workspace's root directory:

```
cd /path/to/your/catkin/workspace
```

2. Source the ROS setup script to ensure that workspace packages are properly recognized:

```
source devel/setup.bash
```

3. Launch the ”moveitdrone” package using the provided launch file. Type the following command:

```
roslaunch moveit_drone full_robot.launch
```

We can see that our simulation was successful and that the gazebo simulation and Rviz motion are both operating well.



Figure 6.1: Spawning the Drone

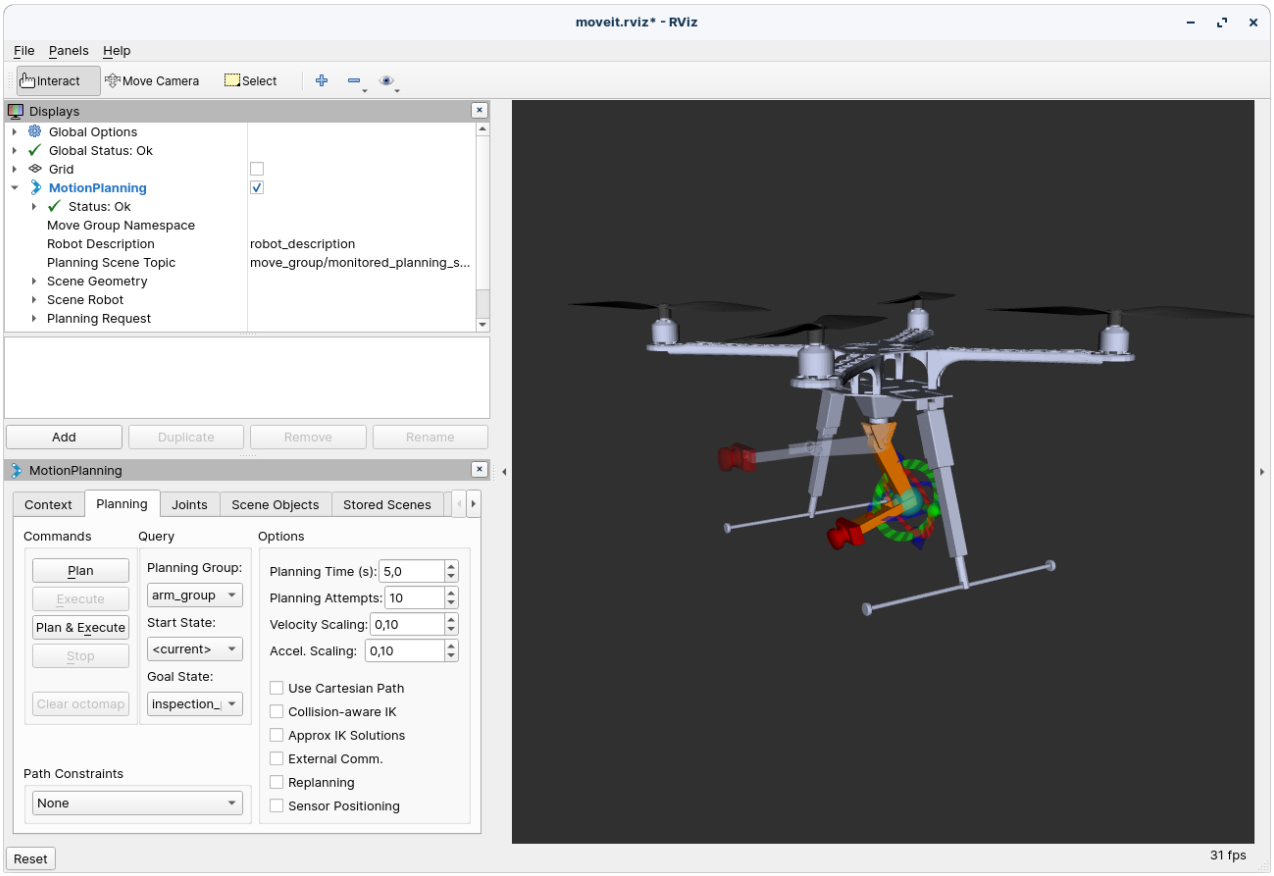


Figure 6.2: Rviz motion planner

6.2.2 Testing our drone: sensors, commands ...

We utilize rqt topic monitor to ensure that all of the drone's components are operational.

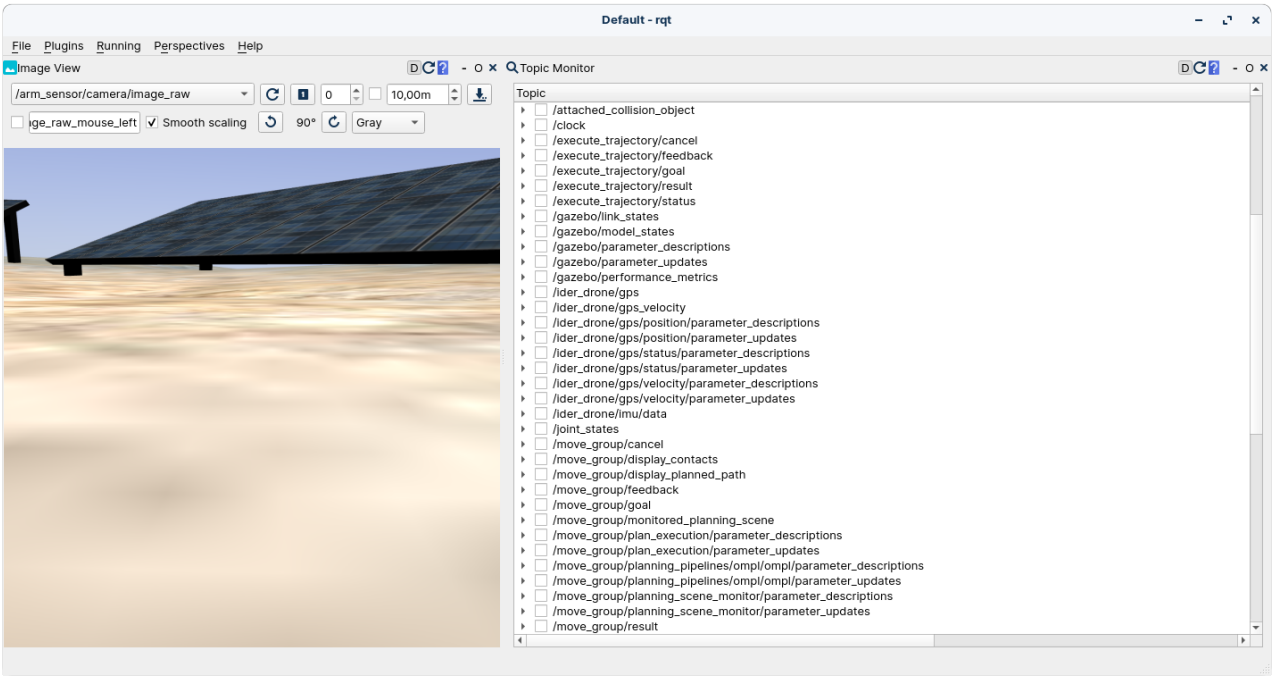


Figure 6.3: Rqt topic Monitor

GPS sensor

The GPS and IMU sensors successfully transmit the data.

/gazebo/performance_metrics	gazebo_msgs/PerformanceMetrics		not monitored
/lider_drone/gps	sensor_msgs/NavSatFix	7.39KB/s	60.55
altitude	float64		6.151138834673414
header	std_msgs/Header		
latitude	float64		19.000000115529875
longitude	float64		72.00000601347266
position_covariance	float64[9]		(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
position_covariance_type	uint8		2
status	sensor_msgs/NavSatStatus		
/lider_drone/gps_velocity	geometry_msgs/Vector3Stamped		not monitored

Figure 6.4: Gps Data

IMU sensor

/lider_drone/imu/data	sensor_msgs/Imu	18.11KB/s	55.56
angular_velocity	geometry_msgs/Vector3		
angular_velocity_covariance	float64[9]		(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
header	std_msgs/Header		
linear_acceleration	geometry_msgs/Vector3		
linear_acceleration_covariance	float64[9]		(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
orientation	geometry_msgs/Quaternion		
w	float64		0.9999996154150621
x	float64		-3.09029820862047 34e-06
y	float64		-8.652500165462027e-06
z	float64		-0.0008769750921891269
orientation_covariance	float64[9]		(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

Figure 6.5: IMU Data

Camera sensor

On the left side of Figure 6.6, we can observe the image published by the camera topic from the simulated environment.

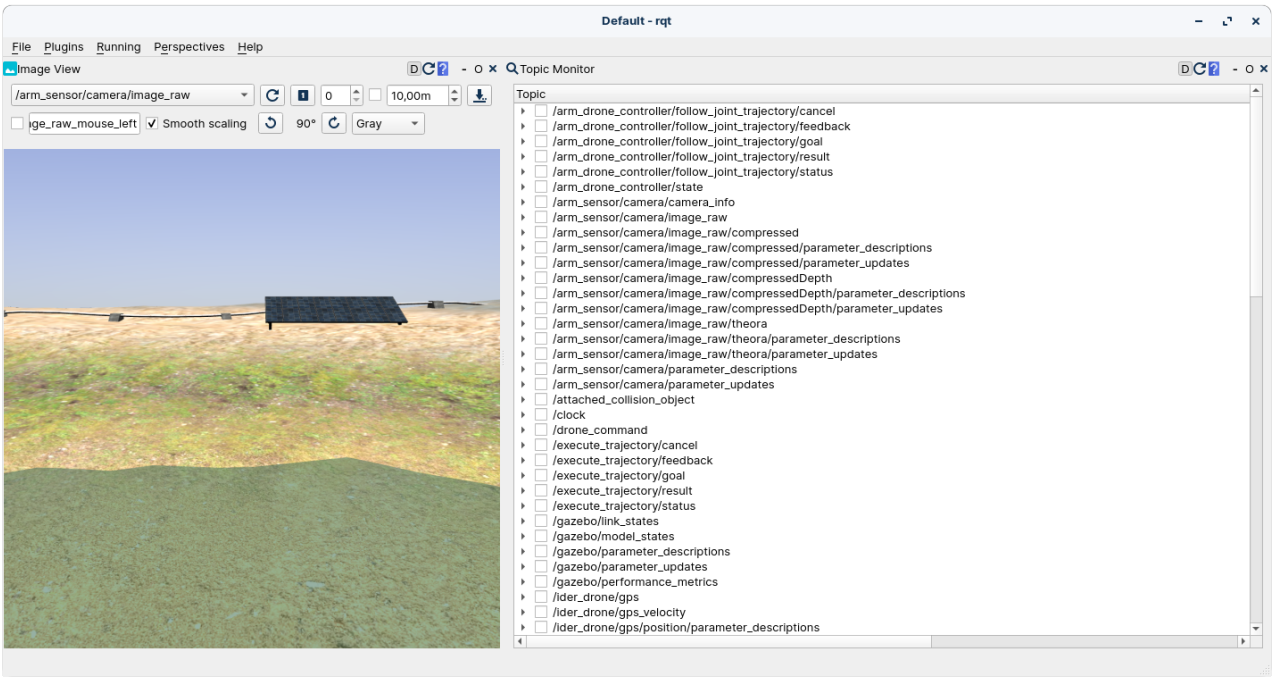


Figure 6.6: image from simulated world

Position Controller outputs

The Rqt Topic visualizer shows that the position controller is functioning as expected; the position controller node is transmitting commands (aux1,aux2,aux3,aux4, rcThrottle,rcRoll,rcPitch,rcYaw).

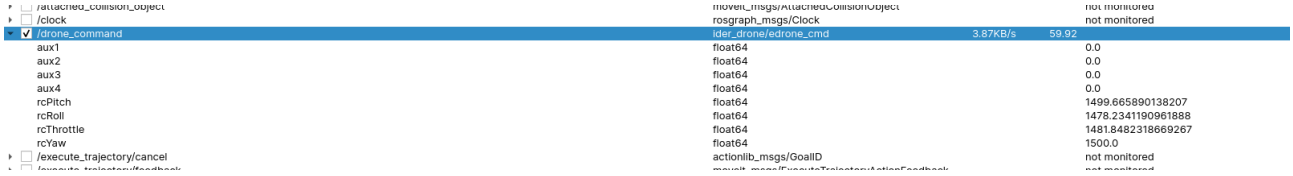


Figure 6.7: Position Controller outputs

Attitude Controller outputs

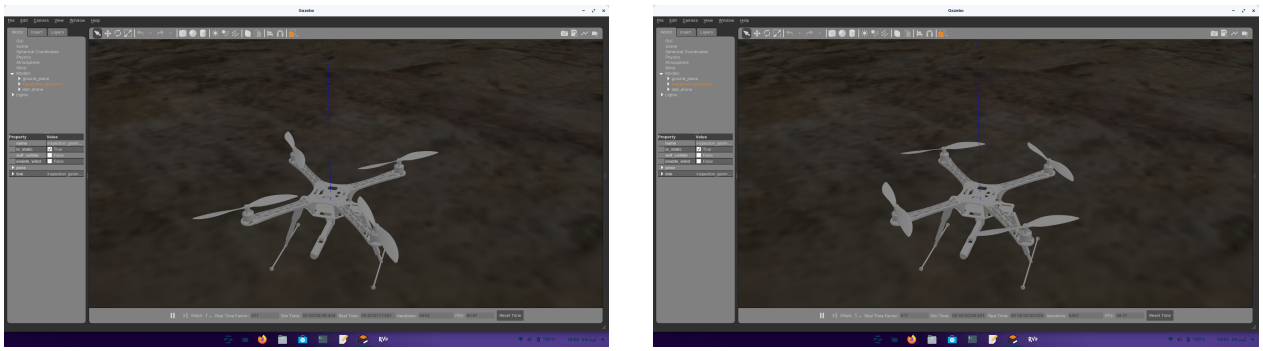
The drone order sent by the position controller node is successfully accepted by the Attitude Controller node, which then sends the calculated motor speed as pwm to the Gazebo propeller plugin.



Figure 6.8: Attitude Controller Outputs

6.2.3 Sending commands to test the motors

In this example, we give the drone a PWM command to test its motors.



(a) motors position 1 (b) motors position 2

Figure 6.9: Motors spinning at 5 rpm

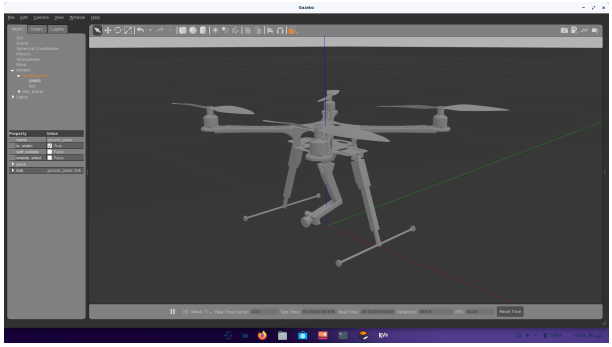
6.2.4 Testing the created Moveit Robot Arm poses

In the Moveit motion planner, we built poses for our robotic arm such as pick, zero, and inspection, Fig. 6.10.

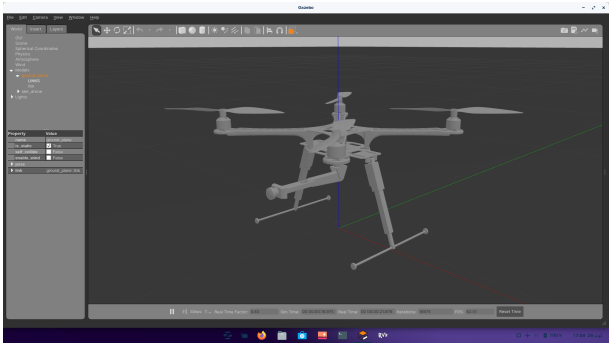
6.2.5 Make a simple hovering test at fixed altitude 3m

By altering the position controller Python script, we can create a hover at a set altitude of 3m, as shown in Figure 6.11, by following these steps:

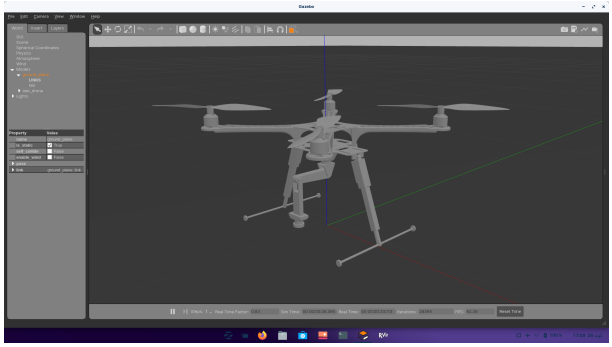
```
roslaunch moveit_drone full_robot.launch
roslaunch iderdrone hovering_task.launch
```



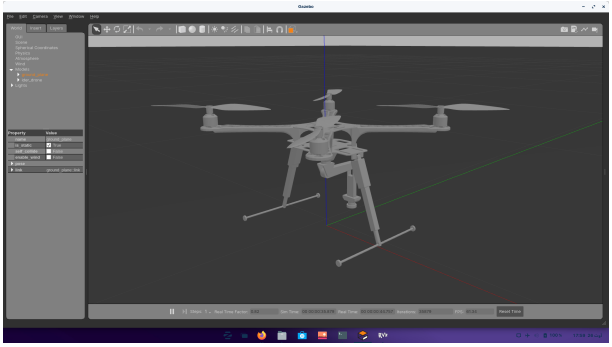
(a) inspection pose



(b) zero pose



(c) pick pose



(d) place pose

Figure 6.10: Poses Examples

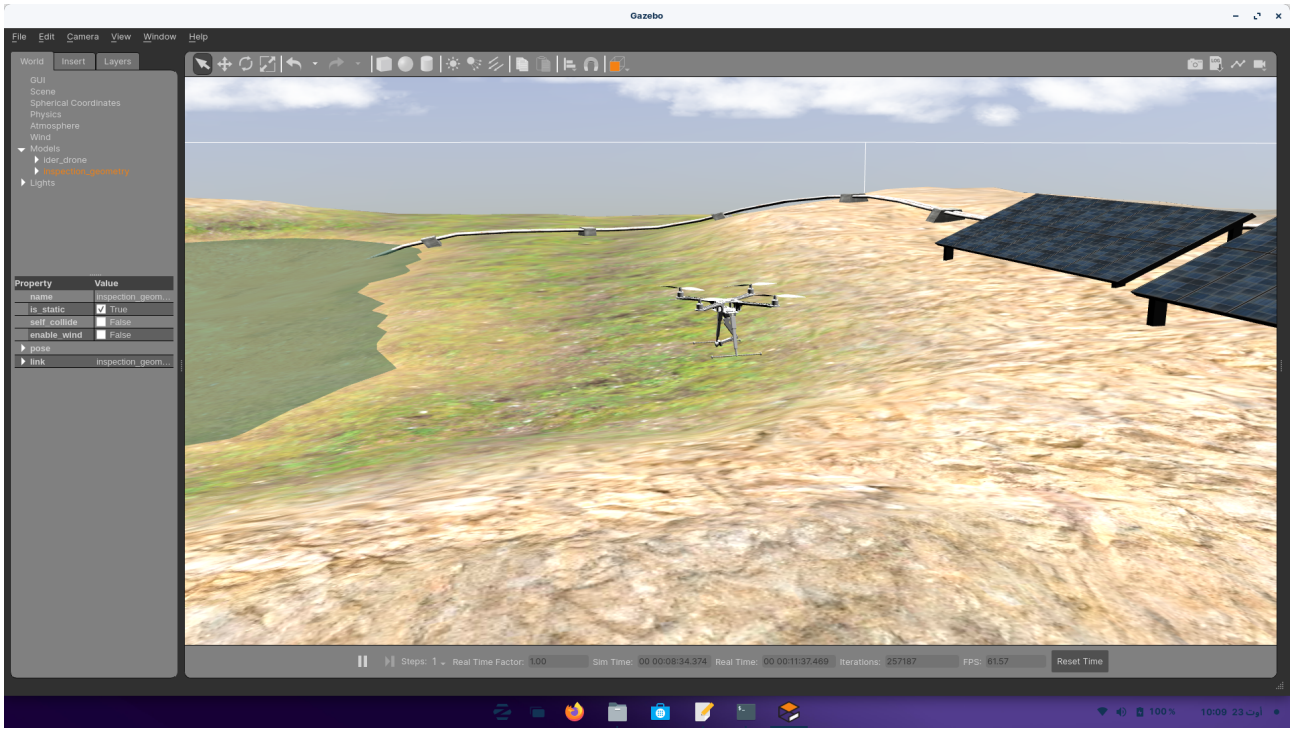


Figure 6.11: Hovering test

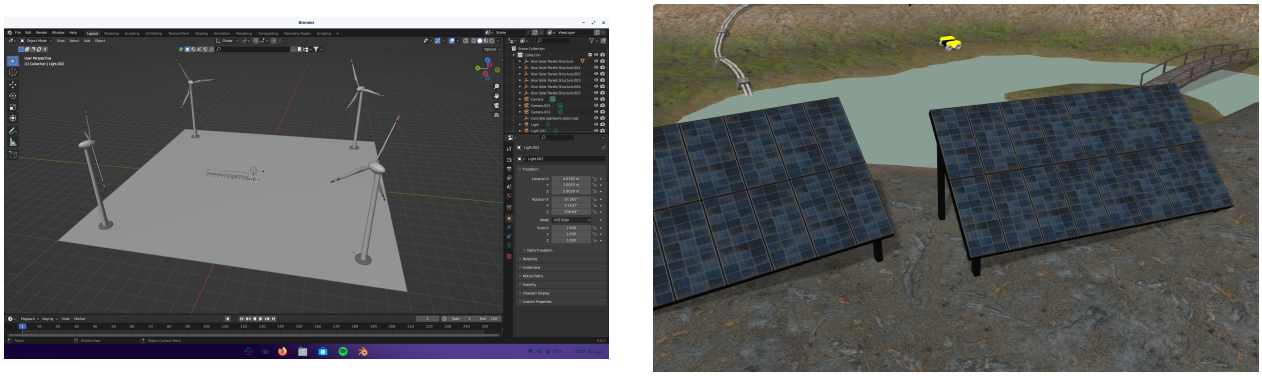
6.3 Robotic Tasks

In this section, we delve into the practical application of our aerial manipulator system within the simulated environment, focusing on two distinct robotic tasks: trajectory following and solar panel inspection.

6.3.1 Creating a 3d world for simulation

As previously said, gazebo employs world models to execute the simulation; we may choose any world that will assist us in performing our simulations; in our tasks, we manage to construct from scratch a basic world with an array of solar panels, as well as some dwellings and wind turbines.

Figure 5.12 depicts a 3D model made with Blender and the Blenderkit[**20**] on the left, and a world developed by clearpath robotics [**21**]on the right.



(a) Custom blender model

(b) inspection world by clearpath robotics

Figure 6.12: Some worlds used in simulation

6.3.2 Solar Panel: Trajectory Tracking and Inspection

The ability of an aerial robotic system to execute precise and dynamic trajectories is a fundamental requirement for various applications, such as surveillance, mapping, and monitoring. Through our research, we have successfully developed and implemented a trajectory following task for our aerial manipulator.

This example shows you the list of **Setpoints** that presumably represent coordinates or waypoints for a robotic task. Each entry in the list seems to consist of three values: latitude, longitude, and altitude :

```
self.setpoints = [[19.0,72.0,0.171],
[19.0,72.0,5.0],[19.0,71.9999,5.0],
[19.0,71.9998,5.0],[19.0,71.9997,5.0],
[19.0,71.9996,5.0],[19.0,71.999585,5.0],
[19.0,71.99957,0.171]]
```

In this task, we defined a series of waypoints and trajectories that the aerial manipulator must follow. Leveraging the control strategies we've meticulously engineered, the robotic system demonstrates its agility and accuracy in adhering to these predefined paths. This task serves as a testament to the effectiveness of our control algorithms in real-world scenarios.

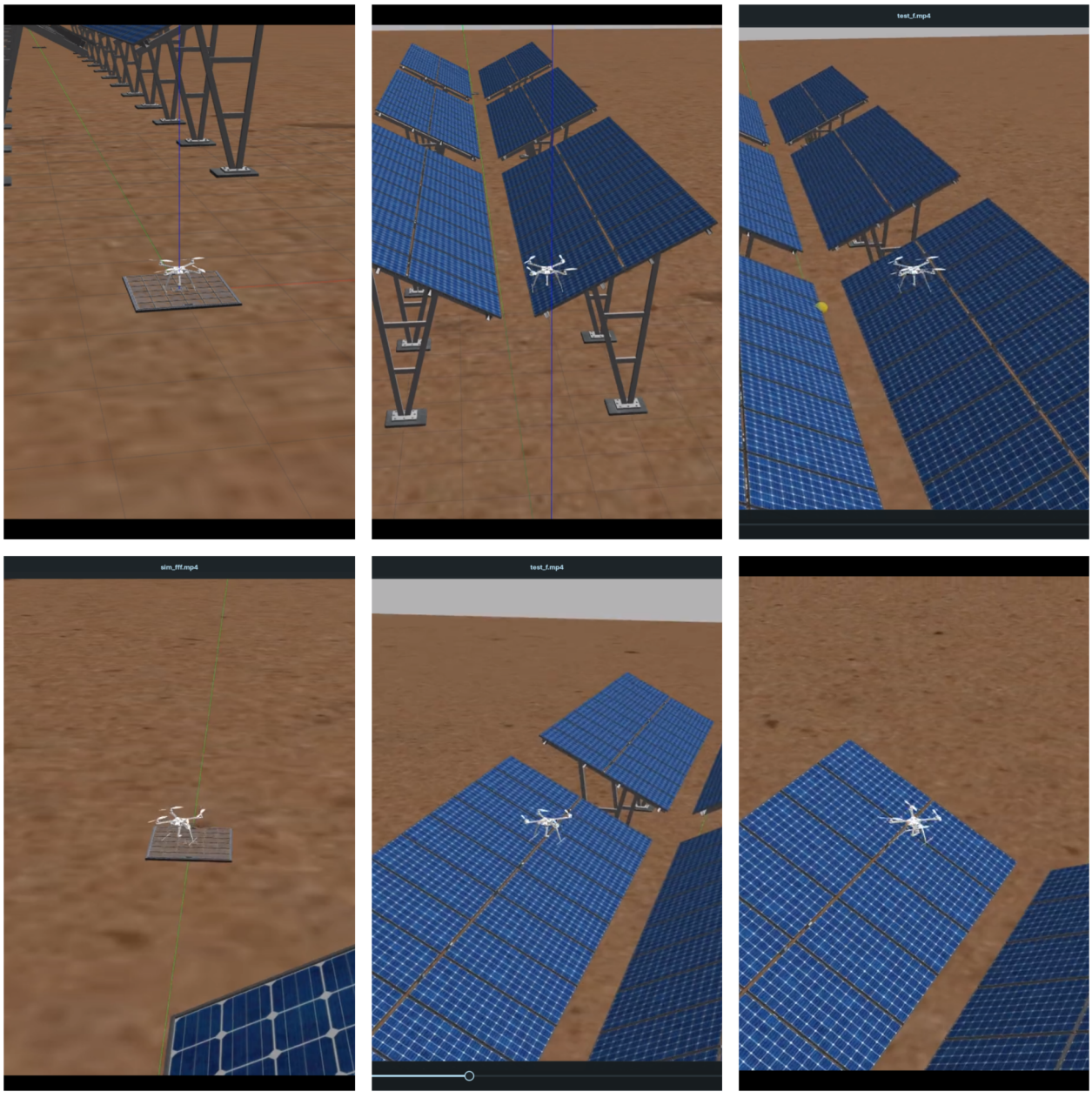


Figure 6.13: Following the Given Setpoints

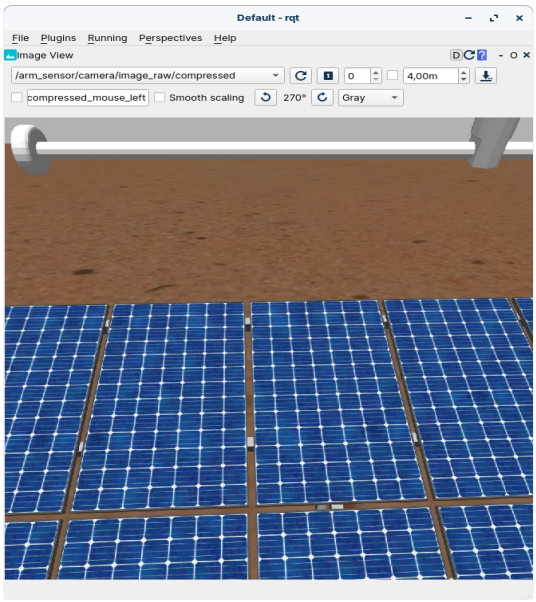


Figure 6.14: Camera View

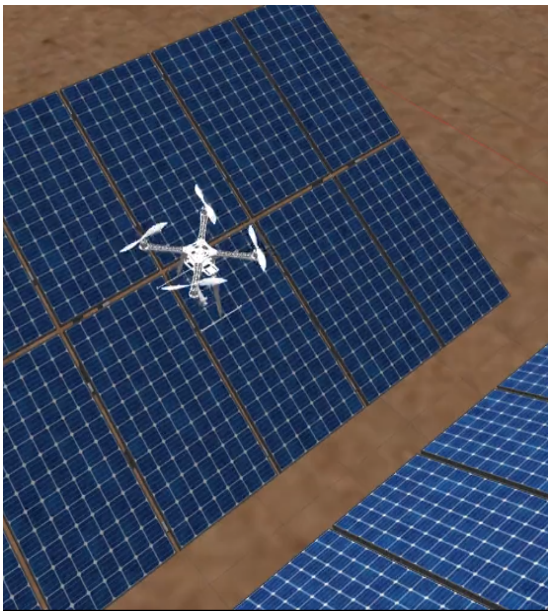


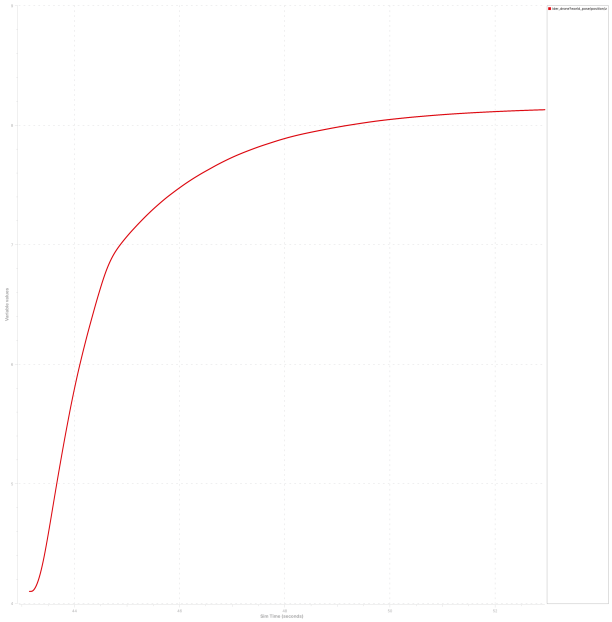
Figure 6.15: Drone hovering and inspecting a solar panel

Within this task, the robotic system autonomously navigates to a designated solar panel array, hovers at a specified altitude, and employs an onboard Camera sensor to inspect individual solar panels for anomalies, damage, or dirt accumulation. The ability to carry out such inspection tasks with precision and efficiency underscores the practical applicability of our research in industries embracing renewable energy sources.

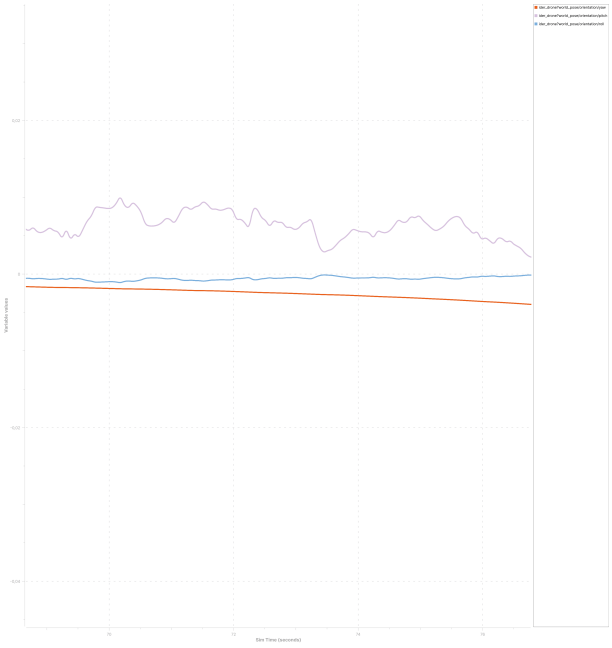
Here are some reactions to our system's cascaded pid loop control, including Figure 5.16.an accurate setting of the system target height and Figure 6.16.b, roll, pitch, and yaw error, which is almost zero (the desired angles).

Because PID control is so straightforward, we had to build the robotic arm that is attached to the drone in the middle to distribute the mass evenly and preserve the system's center of mass in the base drone.

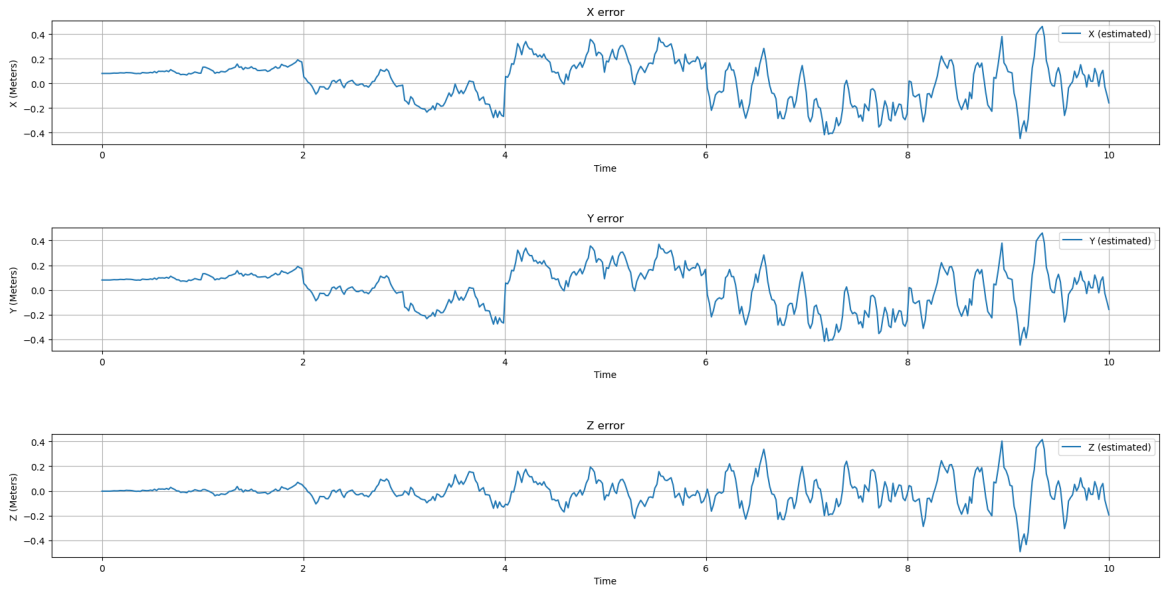
A system like ours requires better control approaches to get better results; in our instance, there are 6 parameters [longitude, latitude, altitude, roll, pitch, yaw] and each one has PID gains that must be carefully adjusted and validated in all simulation scenarios before being used in a Real model.



(a) Z Response: Altitude



(b) Roll, Pitch, Yaw Responses



(c) X Y Z Position errors

Figure 6.16: Simulation results

Conclusion and perspectives

General conclusion

In this project, we have taken a comprehensive journey through the realms of unmanned aerial vehicles (UAVs) and unmanned aerial manipulators (UAMs), delving into the intricate worlds of robotics, control systems, and simulation environments. Our main objective was to investigate how these technologies may be combined to produce an adaptable and powerful aerial robotic system. We have accomplished a number of notable things and made some important contributions to the field through our study.

We commenced our exploration by examining the historical evolution of UAVs, tracing their roots from simple remote-controlled aircraft to the sophisticated, autonomous aerial systems we see today. This historical perspective allowed us to appreciate the rapid advancements that have transformed UAVs into indispensable tools across various domains.

Building upon this historical foundation, we conducted a review of the state-of-the-art in UAV and UAM technology. This review revealed the multifaceted nature of these systems, encompassing everything from manipulative robots to advanced control strategies for quadrotors. Understanding the current landscape was pivotal in shaping the direction of our research.

We then ventured into the world of Robotics Operational Systems (ROS), a robust platform that empowers researchers and engineers to create complex robotic systems with ease. Our in-depth exploration of ROS, including its core components like nodes, topics, and services, provided us with the tools necessary to build a strong foundation for our project.

The heart of our research lies in the development of a versatile robotic system. Using SolidWorks modeling and URDF format conversion, we constructed a detailed and realistic representation of our aerial manipulator. This allowed us to create a ROS package that served as the backbone for control and simulation.

Validation of our ROS package was essential, and we successfully demonstrated the accuracy of our model using RViz and Gazebo. These simulation environments provided a safe and controlled space for testing and development, ensuring that our robotic system would function as intended.

The culmination of our efforts can be observed in the control strategies we implemented. Leveraging Gazebo's capabilities, we devised and fine-tuned control algorithms for position and attitude control, with a focus on stability and precision. This laid the groundwork for the practical application of our aerial manipulator in various scenarios.

In our simulation results and analysis, we provided a user guide for our system, highlighting its versatility and potential applications. We also assessed the computational power required for our simulations and evaluated the performance of our control algorithms under various conditions. The results of these tests demonstrated the robustness and effectiveness of our system.

In conclusion, this work represents a significant step forward in the integration of UAVs and UAMs within the context of ROS and simulation environments. Our contributions encompass modeling, control, and validation, providing a solid foundation for

future research and practical applications in fields such as aerial robotics, automation, and beyond. As technology continues to evolve, we anticipate that our work will serve as a valuable resource for those seeking to push the boundaries of aerial robotic systems.

The journey does not end here. There is ample room for further exploration, refinement, and real-world deployment of the concepts and technologies presented in this work. As we conclude this chapter of research, we look forward to the innovations and advancements that the future holds for aerial robotics and its myriad applications.

Perspectives

Looking ahead, the field of aerial robotics holds immense promise and potential for further exploration and innovation. While this project has laid a strong foundation for the integration of UAVs and UAMs within the context of ROS and simulation environments, there are several exciting avenues for future research and development.

Expanding Application Domains:

Expanding beyond this study, our system can revolutionize diverse robotic applications, including Search and Rescue, Delivery, and Renewable Energy infrastructure maintenance**a related work on Aerial Manipulation using this system**, boosting robotic versatility and addressing real-world challenges effectively.

Platform for Control Algorithm Development:

Building upon the success of our simulation framework, there is a pressing need to create an open and adaptable platform for the development and testing of control algorithms. This platform will serve as a collaborative space for researchers and engineers to refine and innovate upon control strategies for aerial robots.

Transition to Real Hardware Implementation:

The transition from simulation to real-world deployment is a critical milestone. Future endeavors should focus on adapting and refining the technologies and algorithms developed in this virtual environment for practical implementation on real hardware.

In summary, the journey of aerial robotics is far from over, and there are numerous exciting prospects on the horizon. Researchers, engineers, and innovators in this field have the opportunity to shape the future of technology and its applications, with the potential to bring about transformative changes in various industries and sectors. As we look forward, collaboration and interdisciplinary efforts will play a pivotal role in unlocking the full potential of aerial robotic systems.

Bibliography

[1] J. F. Keane and S. S. Carr, "A brief history of early unmanned aircraft," pp.558–571, 2013. 558–571, 2013.

[2] F. S. Barbosa, "4DOF Quadcopter: Development, Modeling and Control," p. 86, 2017.

[3] B. Siciliano, L. Sciavicco, L. Villari and G. Orlio. Robotics Modeling, Planing and Control. Advanced Textbooks in Control and Signal Processing. UK. Springer-Verlag. 2009

[4] Jonatan Alvarez Muñoz, Modélisation et contrôle des véhicules VTOL avec manipulateurs rigides, thèse de doctorat, Université de Grenoble, 2017.

[5] R.Rossi, A.Santamaria-Navarro,J.Andrade-Cetto, andP.Rocco, "Trajectory generationforunmanned aerial manipulators through quadratic programming",IEEE Robotics andAutomation Letters, 2(2):389-396,2017.

[6] H. Lee and H. Kim. "Estimation control and planning for autonomous aerial transportation". IEEE Transactions on Industrial Electronics, vol.64N°4, pp 3369-3379, 2017.

[7] S. Kannan, M. Olivares-Mendes, and H. Voos, "Modeling and control of aerial manipulation vehicle with visual sensor", in 2nd IFAC Workshop on Research, Education and Development of Unmanned Aerial Systems, pp. 303–309, 2013.

[8] G. Antonelli and E. Cataldi, "Adaptive control of arm-equipped quadrotorsTheory and simulations", in 22nd Mediterranean Conference on Control and Automation, Palermo, I, pp. 1446–1451, 2014.

[9] M. Orsag, C. Korpela, S. Bogdan and P. Oh, "Hybrid Adaptive Control for Aerial Manipulation", J. Intelligent and Robotic Systems, 73(1-4): pp. 693-707, 2014.

[10] A.E. Jimenez-Cano, G. Heredia, M. Bejar, K. Kondak, A. Ollero, "Modelling and control of an aerial manipulator consisting of an autonomous helicopter equipped with a multi-link robotic arm", Proc. Inst. Mech. Eng. G 230 (10), pp. 1860–1870, 2013.

[11] Lippiello V. "Grasp the possibilities: anthropomorphic grasp synthesis based on the object dynamic properties", IEEE Rob Autom Mag;22(4):69–79, 2015.

[12] S. N. Sy Salim and M. Zainon, Control Systems Engineering. Penerbit Universiti Teknikal Malaysia Melaka, 2010, 01 2010.

[13] "Pid control system analysis, design, and technology," Control Systems Technology, IEEE Transactions on, vol. 13, pp. 559 – 576, 08 2005.

[14] J. Kerr and K. Nickels, "Robot operating systems: Bridging the gap between human and robot," 44th IEEE Southeastern Symposium on System Theory (SSST), pp. 99–104, 03 2012.

[15] Ros wiki Website : https://wiki.ros.org/ROS.

[16] SwtoURDF Plugin, Authored and maintained by Stephen Brawner. Past supporters include PickNik Consulting, Verb Surgical, Open Robotics, and Willow Garage. URL : https://github.com/ros/solidworks_urdf_exporter .

[17] Suhas Varadaramanujan, Melissa Morris," Design of a Drone with a Robotic End-Effector", 30th Florida Conference on Recent Advances in Robotics, May 11-12-2017. .

[18] Siciliano , Khatib , Springer Handbook of Roborics , , ISBN : 978-3-540-23957-4 .

[19] Oussama Khatib and Krasimir Kolarov ,Lecture Notes (CS223A) , Stanford University , Winter 2016-17.

[20] blenderkit addons , URL:https://www.blenderkit.com/ , Last time Visited : 09/09/2023.

[21] blog: "New Gazebo Worlds", URL: https://clearpathrobotics.com/blog/2020/07/ , Last time Visited : 17/08/2023.

[22] Motor Mixing Algorithm, URL : http://autoquad.org/wiki/wiki/configuring-autoquad-flightcontroller/frame-motor-mixing-table/ , Last time Visited : 28/08/2023.

[23] Plotjuggler github , URL : https://github.com/facontidavide/PlotJuggler , Last time Visited : 10/07/2023.