Simulations using Flightmare

Andreas Andersen Kjernlie
Faculty of Technology, Art and Design
Oslo Metropolitan University
Oslo, Norway
s351665@oslomet.no

Hamideh Azarmanesh
Faculty of Technology, Art and Design
Oslo Metropolitan University
Oslo, Norway
s351666@oslomet.no

Johanna Bersås Eggen
Faculty of Technology, Art and Design
Oslo Metropolitan University
Oslo, Norway
s351668@oslomet.no

Abstract:
This research report will take you through the group project in ACIT4820, where the task was to make a simulation using ROS/Gazebo. In the first phase of the project, the project topic was chosen. The group went with Flightmare, a quadrotor simulator described in the research paper "Flightmare: A Flexible Quadrotor Simulator". The simulator was composed by two main components: a configurable rendering engine built on Unity, and a flexible physics engine for dynamics simulation. These two components could run independently of each other, which made the simulator very fast. During the implementation of Flightmare, VMWare, Ubuntu and Gazebo were installed on the computers. After successfully installing and running the systems, the simulation in Flightmare could begin. By using the official Flightmare code from the "Flightmare: A Flexible Quadrotor Simulator" research paper, testing was done and different implementations by the group were tried out. The final code resulted in using a PID-controller moving the quadrotor over a wall, flying close to the ground and hovering between the first wall and another wall, before flying upwards again, over the next wall and then landing. The research paper "Flightmare: A Flexible Quadrotor Simulator" can be found in the webside link below, together with the final code developed and used in this project.

Webside: https://uzh-rpg.github.io/flightmare
Code: https://github.com/Hamidehazar/Flightmare

Keywords: drone simulator, flightmare, gazebo, photo-realistic rendering, quadrotor simulator, ROS, ubuntu, vmware

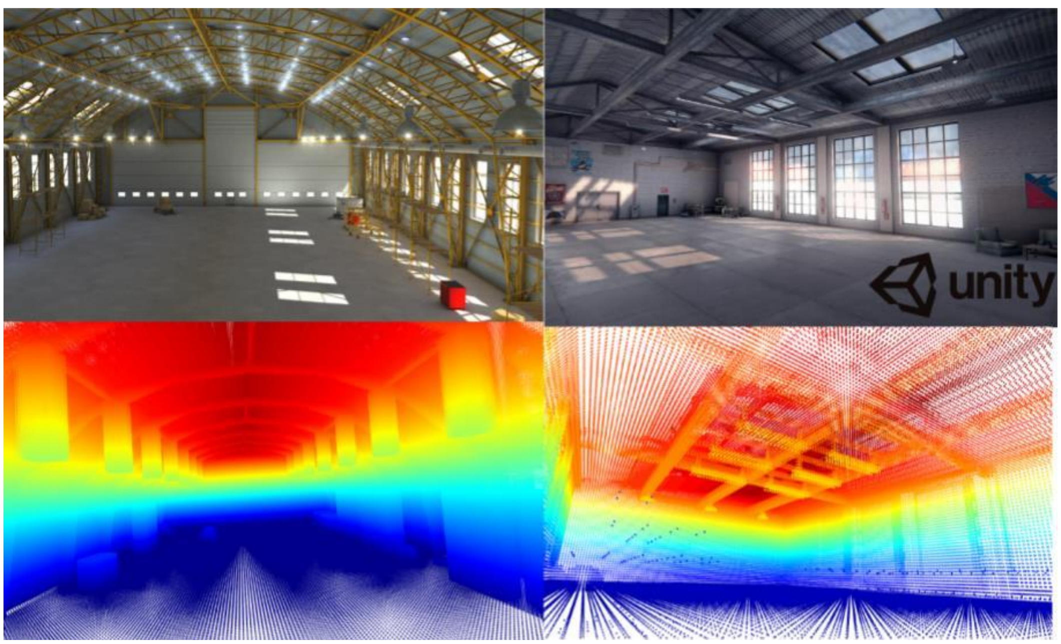
Table of content

1. Introduction

Simulators are very valuable tools for robotics research, due to their ability to develop and test algorithms in a safe and affordable environment. Instead of using expensive equipment to carry out testing and prototyping, it can all be done faster and easier with robot simulators as Gazebo or Webot.

In this group project, we will use what we have learned throughout the ACIT4820 course to successfully present a simulation using ROS/Gazebo. The goal of the project is to gain hands on experience with the course materials and successfully implementing robot simulations to describe and demonstrate in the final project report.

Our group went with Flightmare, a quadrotor simulator developed by five students from the University of Zurich (http://rpg.ifi.uzh.ch/docs/CoRL20_Yunlong.pdf). With the Flightmare simulator, the code developed by the five founders will be used as the ground builder in this final simulation, with some later on experiments and implementations done by our group. The final result of the code will be presented throughout this report, by the end of November 25th.



A demonstration of the visual possibilities with the Flightmare Simulator
https://arxiv.org/pdf/2009.00563.pdf

2. Theory

In this chapter, the theory regarding the systems used in this project will be described. This with the intention to get a better understanding of the work done by our group later on in the report.

2.1VMware Workstation 16 Player

VMware Workstation Player is a desktop virtualization application, first released in December 2005. The purpose of using VMware is that the application can run another operating system on the same computer without rebooting. VMware provides a simple user interface, unmatched operating system support, and portability across the VMware ecosystem. (https://docs.vmware.com/en/VMware-Workstation-Player/16/rn/VMware-Workstation-16-Player-Release-Notes.html)

The VMware Workstation 16 Player is the latest version of VMware, released in November 2020. The upgrade of this version provides better performances within file transfer speeds, virtual machine shutdown time and virtual storage performances.
https://docs.vmware.com/en/VMware-Workstation-Player/16/rn/VMware-Workstation-16-Player-Release-Notes.html

https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html

2.2 Ubuntu 18.04

Ubuntu was first released in 2004 and is an open-source operating system based on the Debian GNU/Linux distribution. The software includes all the features of a Unix OS with an added customizable GUI, which makes it popular in universities and research organizations. Ubuntu is primarily designed to be used on personal computers, but server editions also exists.
https://www.techopedia.com/definition/3307/ubuntu

ubuntu core

https://meterpreter.org/ubuntu-18-04-2-lts-bionic-beaver-released/

Ubuntu is sponsored by Canonical Ltd., a company that generates income by selling support and services to complement Ubuntu. Canonical releases a new version of Ubuntu every six months and provides support for the specific version up to 18 months later, in the form of patches and security performances. Ubuntu 18.04 was released on April 26, 2018 and is the second latest version of Ubuntu after the 20.04 version released in august 2020.

2.3 ROS

ROS stands for Robot Operating System and is an open-source, meta-operating system for robots. It provides libraries and tools to help software developers create robot applications, with the possibility to obtain, build and run code across multiple computers. This means hardware abstraction, device drivers, libraries, visualizers, message-passing, package management and more. http://wiki.ros.org/

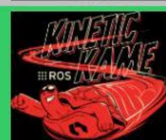
ROS currently only runs on Unix-based platforms, where the software primarily is tested on Ubuntu and Mac OS X systems. It is worth mentioning that the ROS community has contributed support from Fedora, Gentoo, Arch Linux and other Linux platforms. http://wiki.ros.org/ROS/Introduction

2.3.1 ROS Melodic

ROS Melodic Morenia is the twelfth ROS distribution release, released in May 2018. A ROS distribution means being a versioned set of ROS packages. These are related to Linux distributions, as Ubuntu. ROS Melodic is primarily targeted at the Ubuntu 18.04 (Bionic) release. http://wiki.ros.org/melodic

http://wiki.ros.org/melodic

There are many different types of robots with various needs, as seen in a cut out overview of ROS distributions below. The distributions with green color (as ROS Melodic), is releases that are still supported.

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			May, 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
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)

http://wiki.ros.org/Distributions

2.3.2 Catkin

Catkin is the official build system of ROS, and successor to the previous system called rosbuild. The reason why catkin is now the standard of all ROS systems is that it is more conventional and easier to use cross platforms. It has better package distribution, cross compiling support and portability.

The function of catkin is to transform the raw source code into packages, and each package will consist of one or several targets when built.

2.4Gazebo

Gazebo is a well-designed 3D simulator, that offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It is described as a robust physics engine with high-quality graphics and convenient programmatic and graphical interfaces (http://gazebosim.org/). The latest version of Gazebo is the 11.0.0 version, and it was released in January 2019.

https://blog.generationrobots.com/en/robotic-simulation-scenarios-with-gazebo-and-ros/

Gazebo has the following main components:

- World files: Contains all the elements in a simulation, including robots, lights, sensors, and static objects.
- Models – Represent individual elements. The three robots and the object in front of them are models.
- gzserver – Called the "work horse" Gazebo program. It reads the world file in order to generate and simulate a world
- gzclient – This client connects itself to the pzserver in order to visualize the elements. Both the gzserver and the gzclient are listed in the shell output, beneath "NODES".
- gzweb – This is a web version of gzclient, using WebGL.

https://www.theconstructsim.com/ros-5-mins-028-gazebo-simulation/

2.5 Flightmare

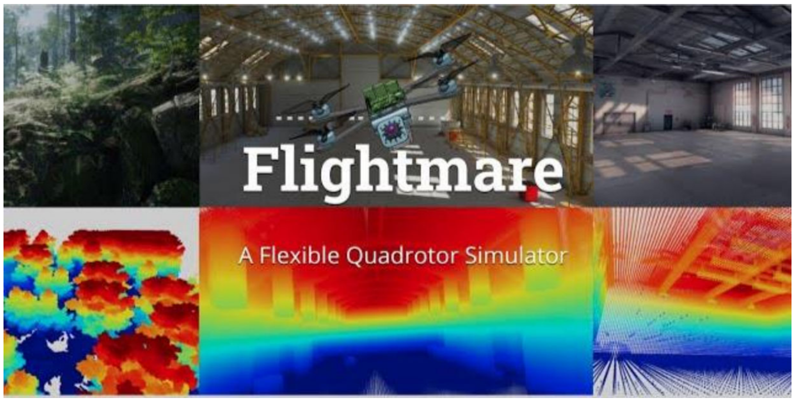
Flightmare is a flexible quadrotor simulator, developed by five students at the University of Zurich. The simulator is composed with photo-realistic rendering engine and a fast quadrotor dynamics simulation. The simulator is set up by two main blocks: a rendering engine, and a physics model. These two blocks are separate and can run independently of each other using parallel programming. Both the rendering engine and the dynamics simulation are flexible by design, as further described in the chapters below.

1.1.1. Rendering engine

The rendering engine can be used within a wide range of 3D realistic environments and generate visual information from low to high vision (http://rpg.ifi.uzh.ch/docs/CoRL20_Yunlong.pdf). It is also possible to simulate sensor noise, motion blur, environment dynamics, wind, lens distortions etc. (A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange. Unity: A general platform for intelligent agents. arXiv e-prints, 2018).

2.5.1 Dynamic Modeling

The dynamic model offers full control to the user in the terms of desired robot dynamics and associated sensing, which means that the user easily can switch between a basic noise-free quadrotor model and a more advanced rigid-body dynamic. The same goes for the inertial sensing and motor encoders, which from the physics model also can be either noise-free or include different degrees of noise (F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. Rotors—a modular gazebo mav simulator framework. In Robot Operating System (ROS), pages 595–625. Springer, 2016).



http://rpg.ifi.uzh.ch/research_drone_racing.html

Flightmare can be used for various applications, including path-planning, reinforcement learning, visual-inertial odometry, deep learning, human-robot interaction, etc.

2.5.2 Features

Some of the main features includes:

- Flexible sensor suite, including RGB images, IMU, depth, segmentation, etc.
- Point Cloud Extractor
- Parallel computing for multi-agents simulation.
- OpenAI Gym-style python wrapper.
- Model-free Reinforcement Learning baselines (stable-baselines).
- ROS integration, including interface to the popular Gazebo-based MAV simulator (RotorS).
- Interface to Model-based quadrotor control.

2.5.3 Application areas

And some of the main applications areas are:

- Reinforcement Learning, Deep Learning
- Path Planning, Model-based Control
- Visual-inertial Odometry, Simultaneous Localization and Mapping
- Virtual-Reality, Human-robot Interaction

https://uzh-rpg.github.io/flightmare/

Flightmare provides interfaces to the popular robotics simulator Gazebo, and a study shows that this simulator can achieve speeds up to 230 Hz for the rendering block and up to 200,000 Hz for the dynamics block with a multi-core laptop CPU (http://rpg.ifi.uzh.ch/docs/CoRL20_Yunlong.pdf).

2.6 MPC controller for quadrotors

Write general theory about MPC controllers here, if we still need to include this.

3. Methods

3.1 Choosing a project

The first thing we had to do in this project, was to choose a project field/topic. In Canvas, there were presented several project ideas, where our group found the drone simulator in Flightmare interesting. The drone simulator seemed like a project that would fit our competence level and a project we all would be able to work and participate on. Most important it seemed like an interesting topic to write about, with the mindset that we in the end of the project should be able to present a 10 000 – 15 000 words report.

3.2 Installing VMware or Linux, Ubuntu 18.04, Gazebo and Flightmare

After choosing a project, the next step was to install VMware, Ubuntu, Gazebo and Flightmare. This was a difficult task in itself, due to continuously struggles with different errors throughout the steps.

The Flightmare GitHub page https://github.com/uzh-rpg/flightmare/wiki/Install-with-ROS was very helpful with providing all the steps for how to get Flightmare up and running.

3.2.1 Install VMware

With the installment of VMware, the only step was downloading the application from the VMware Workstation 16 Player website. https://www.vmware.com/products/workstation-player.html Everything went according to plan, and we could proceed to the next steps.

3.2.2 Install Prerequisites packages

The next step was to install the prerequisites packages. All the steps further below including the prerequisites was done with following the Flightmare installation descriptions in GitHub. https://github.com/uzh-rpg/flightmare/wiki/Install-with-ROS

```
sudo apt-get update && apt-get install -y --no-install-recommends \
   build-essential \
   cmake \
   libzmqpp-dev \
   libopencv-dev
```

https://github.com/uzh-rpg/flightmare/wiki/Install-with-ROS

3.2.3 Ubuntu install of ROS Melodic

To be able to use ROS, it was important to install the right distribution version of it. Since Ubuntu was the operating system used in this project, Melodic Morenia was the only ROS version that would be suited.

ROS Kinetic Kame
Released May, 2016
LTS, supported until April, 2021
This version isn't recommended for new installs.

ROS Melodic Morenia
Released May, 2018
LTS, supported until May, 2023
Recommended for Ubuntu 18.04

ROS Noetic Ninjemys
Released May, 2020
Latest LTS, supported until May, 2025
Recommended for Ubuntu 20.04

http://wiki.ros.org/ROS/Installation

After choosing to install ROS Melodic Morenia, we followed the steps provided in the link: http://wiki.ros.org/melodic/Installation/Ubuntu

3.2.4 Gazebo

Since ROS Melodic was used in this project, Gazebo version 9.x was installed. The figure below shows the decision for this, which makes sense since ROS Melodic is the newest distribution version of ROS that could be used with the newest version of Gazebo.

ROS Melodic and newer: use Gazebo version 9.x `sudo apt-get install gazebo9`

ROS Kinetic and newer: use Gazebo version 7.x `sudo apt-get install gazebo7`

ROS Indigo: use Gazebo version 2.x `sudo apt-get install gazebo2`

https://github.com/uzh-rpg/flightmare/wiki/Install-with-ROS

3.2.5 ROS Dependencies

To install both the system and the ROS dependencies, the following code was used in Ubuntu, where "ROS_DISTRO-octomap-ros" was replaced with "ros-melodic_DISTRO-octomap-ros" and "ROS_DISTRO-joy" was replaced with "ros-melodic-joy".

```
sudo apt-get install libgoogle-glog-dev protobuf-compiler ros-$ROS_DISTRO-octomap-msgs ros-$ROS_DISTRO-octomap-ros ros-$ROS_DISTRO-joy python-vcstool
```

3.2.6 Get catkin tools

To get the catkin tools, the following commands was used:

```
sudo apt-get install python-pip
sudo pip install catkin-tools
```

https://github.com/uzh-rpg/flightmare/wiki/Install-with-ROS

3.2.7 Create a catkin workspace

Create a catkin workspace with the following commands by replacing <ROS VERSION> with the actual version of ROS you installed:

cd
mkdir -p catkin_ws/src
cd catkin_ws
catkin config --init --mkdirs --extend /opt/ros/melodic_DISTRO --merge-devel --cmake-args -DCMAKE_BUILD_TYPE=Release

3.2.8 Install Flightmare

Clone the repository :

cd ~/catkin_ws/src
git clone https://github.com/uzh-rpg/flightmare.git
Clone dependencies:
vcs-import < flightmare/flightros/dependencies.yaml
Build:
catkin build
Add sourcing of your catkin workspace and **FLIGHTMARE_PATH** environment variable to your .bashrc file:
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
echo "export FLIGHTMARE_PATH=~/catkin_ws/src/flightmare" >> ~/.bashrc
source ~/.bashrc

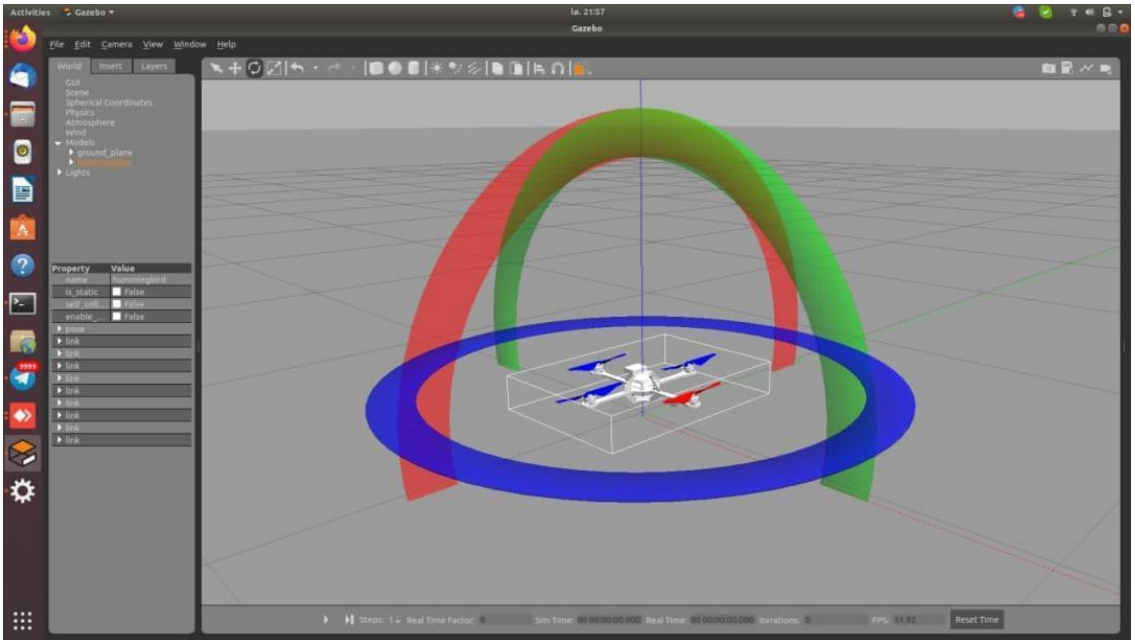
3.2.9 Download Flightmare Unity Binary

Download the Flightmare Unity Binary **RPG_Flightmare.tar.xz** for rendering from the Releases and extract it into the */path/to/flightmare/flightrender*.

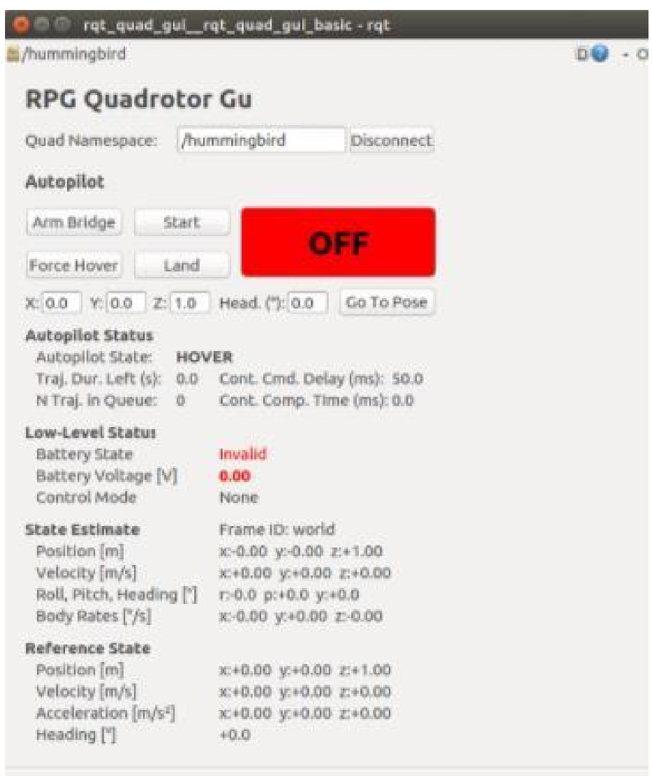
Now, we can move to Basic Usage with ROS and run the example.

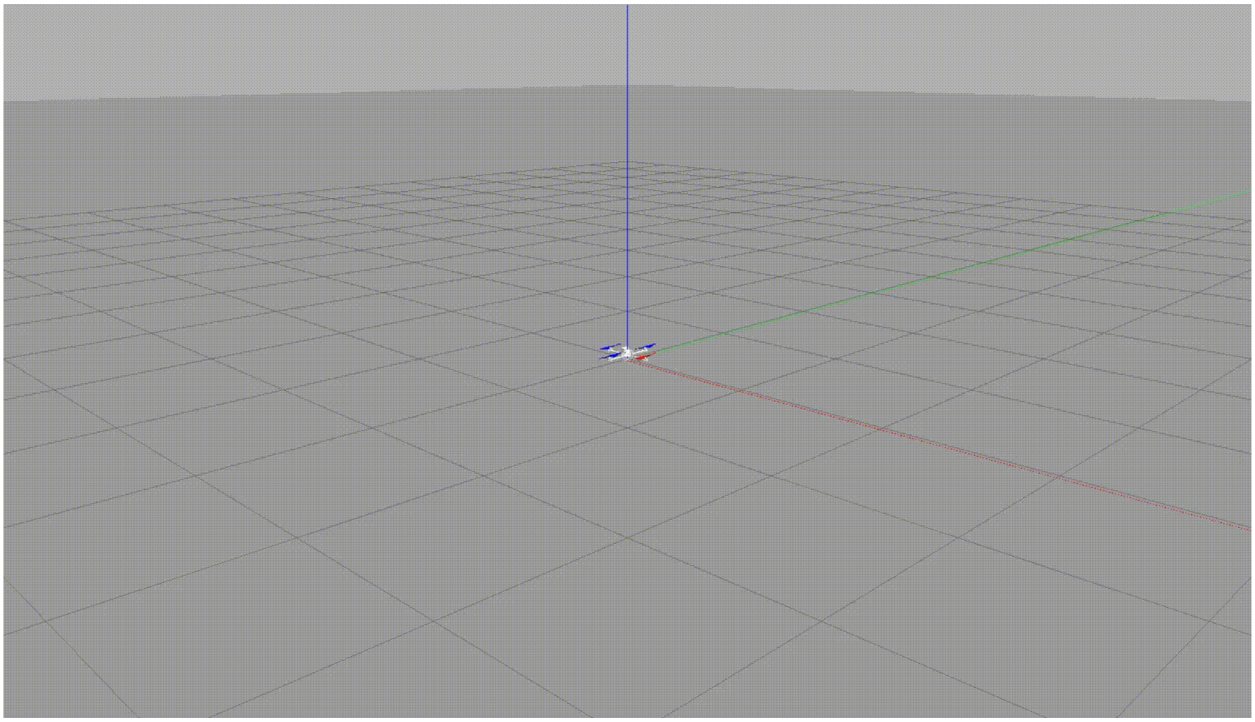
3.2.10 Launch Flightmare (use gazebo-based dynamics)

In this example, we show how to use the RotorS for the quadrotor dynamics modelling, rpg_quadrotor_control for model-based controller, and **Flightmare** for image rendering.
roslaunch flightros rotors_gazebo.launch



Flightmare can be used with other multirotor models that comes with RotorS such as AscTec Hummingbird, the AscTec Pelican, or the AscTec Firefly. The default controller in rpg_quadrotor_control is a PID controller.





It is with manual controller and it is not autopilot. To fly the quadrotor first press the Connect button to connect the GUI, then the Arm Bridge button to enable sending commands to the vehicle and then Start.

3.2.11 9-RPG Quadrotor Control

This repository contains a complete framework for flying quadrotors based on control algorithms developed by the Robotics and Perception Group. They also provide an interface to the RotorS Gazebo plugins to use our algorithms in simulation. Together with the provided simple trajectory generation library, this can be used to test our sofware entirely in simulation. They also provide some utility to command a quadrotor with a gamepad through their framework as well as some calibration routines to compensate for varying battery voltage. Finally, They provide an interface to communicate with flight controllers used for First-Person-View racing.

3.3 Our Goal

3.3.1 Change the environment of project

The project has the basic world that was empty world. We added two walls with wood material on environment to path planning the quadrotor.

3.3.2 Autopiloting the quadrotor

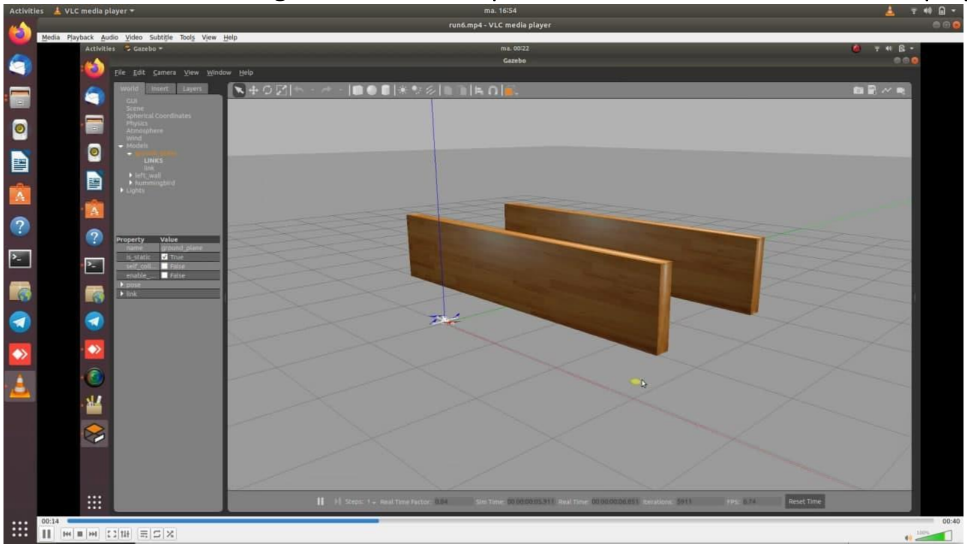
We find some autopilots code and implement on our project to auto start and auto flying the quadrotors.

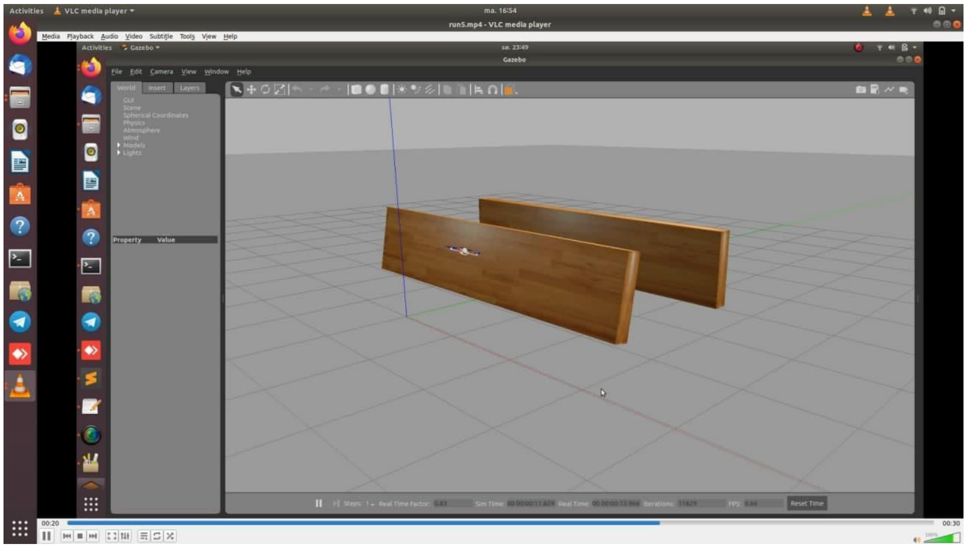
3.3.3 Pathplanning the quadrotor flying

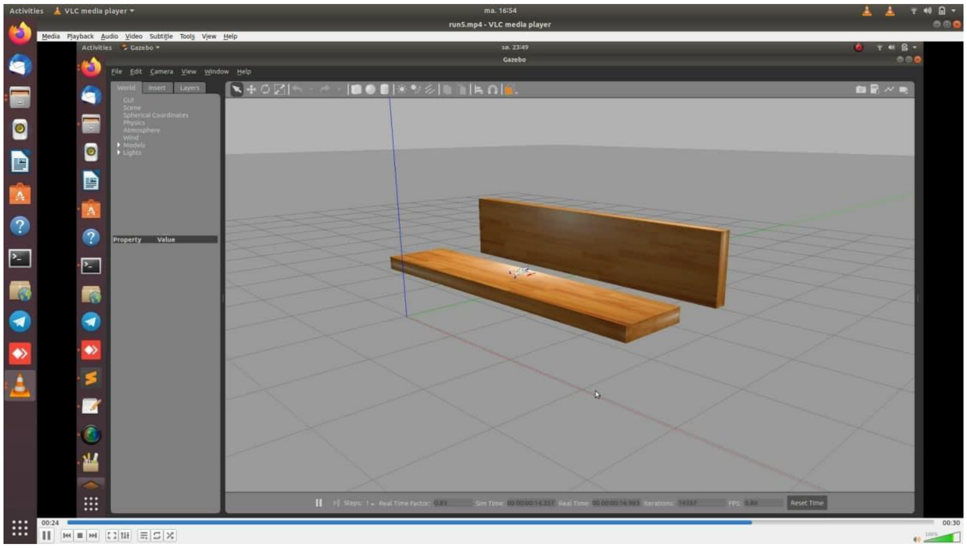
We put 5 position to path planning the quadrotor and some codes to hover it.

3.3.4 Challenges

We have some challenges with that our quadrotor interface the wall and our project failed.







MPC controller for quadrotors

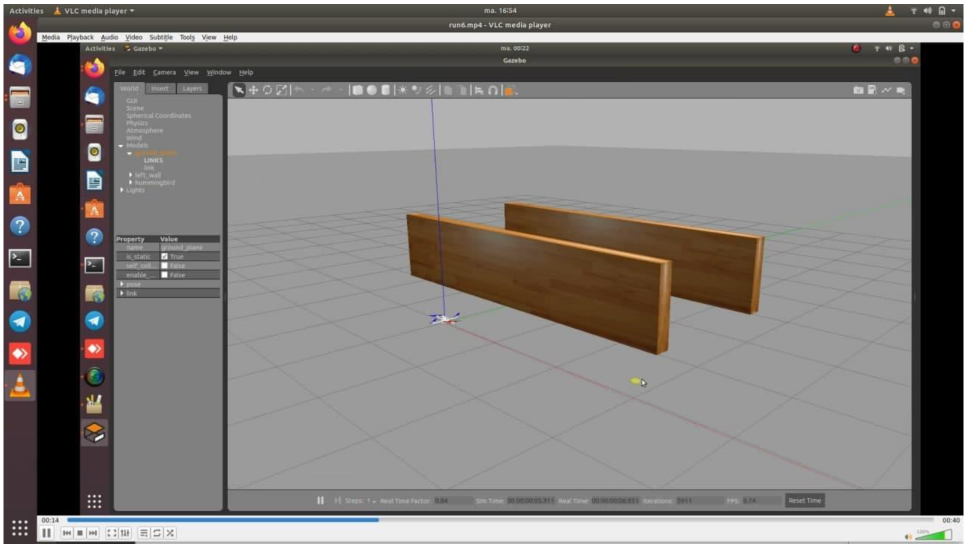
We tried implementing a MPC controller to the Flightmare simulation, with the purpose to enable Flightmare to navigate around obstacles more efficiently. A model predictive controller with perception-aware extension (PAMPS) was integrated into rpg_quadrotor_control, to replace the already built in PID controller that allows the drone to stabilize itself at a fixed angle and coordinate, as well as operate on autopilot. The difference between the already included PID controller and the MPC is that while the PID only has a single input and output (SISO), the MPC has multiple inputs and outputs (MIMO). There are several cool features of the perception-aware extension. For example, the drone can be instructed to hover and circle different objects using its sensors. It can recognize edges and corner of objects and use them as reference points.

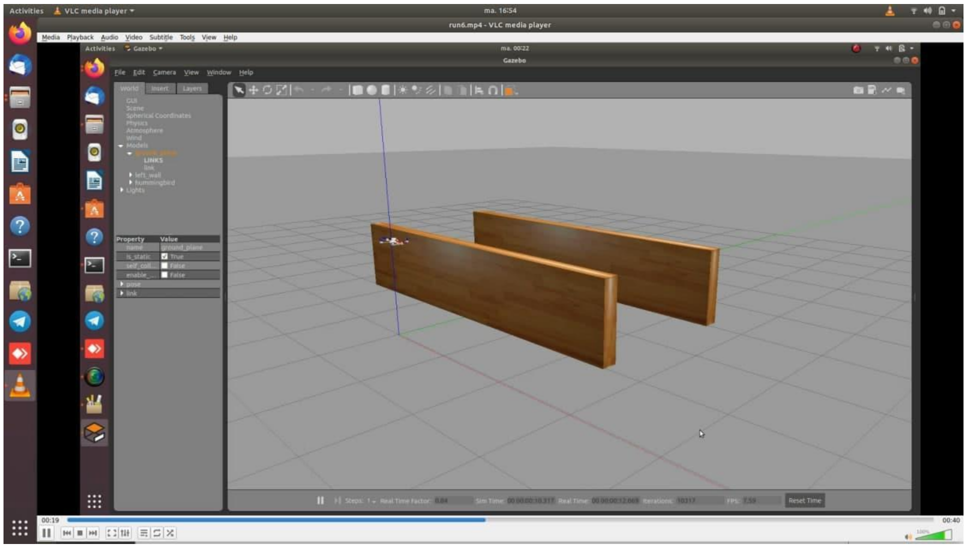
There were issues with loading and running it, however, because there are conflicting buildspaces in ROS. In addition to this, and MPC controller with perception-aware extension was not really needed for our project. The point of using an MPC controller with this extension for drones is, for example, to unify control and planning with perception and action objectives. This makes things a lot more complex, as additional parameters are needed to control the drone. The extension perception-aware model predictive control was published by the following team:
Davide Falanga, Philipp Foehn, Peng Lu, Davide Scaramuzza: **PAMPC: Perception-Aware Model Predictive Control for Quadrotors**, IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS), 2018.

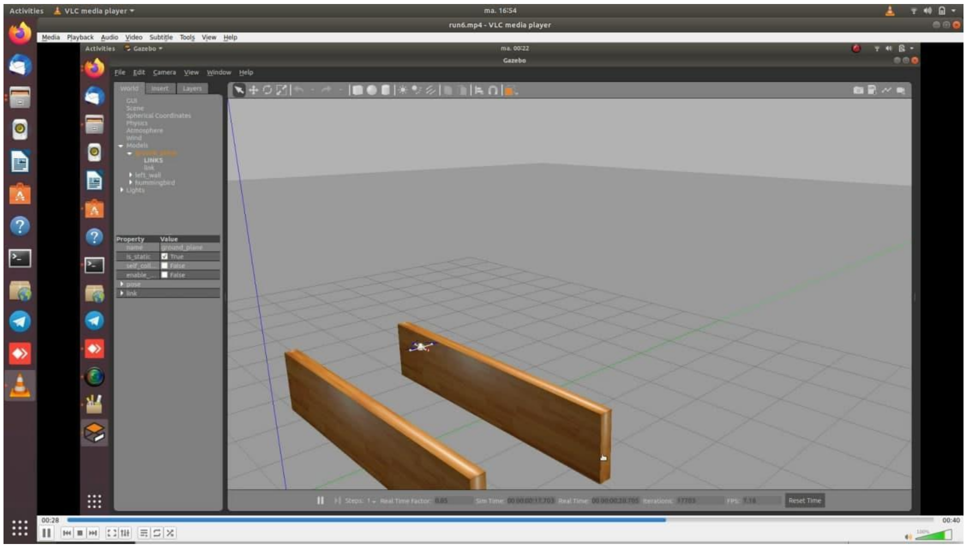
This resulted in going for the method described above with the pathplanning.

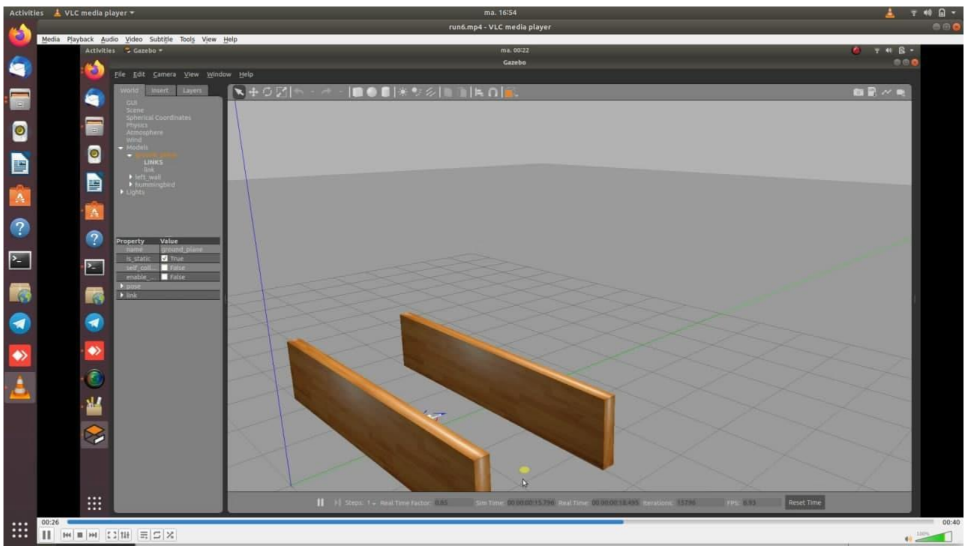
4.Result

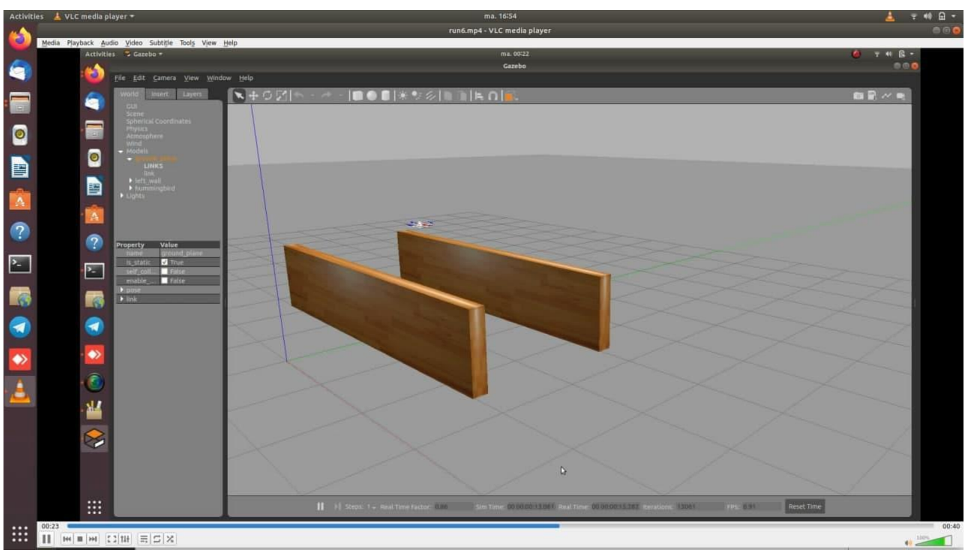
We did it and our quadrotor works properly.

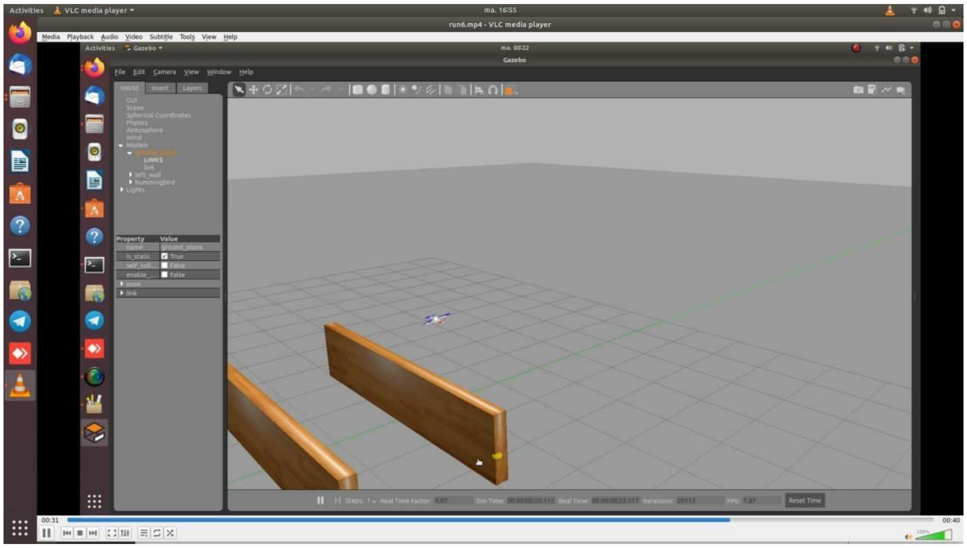

run6.mp4 - VLC media player
Gazebo

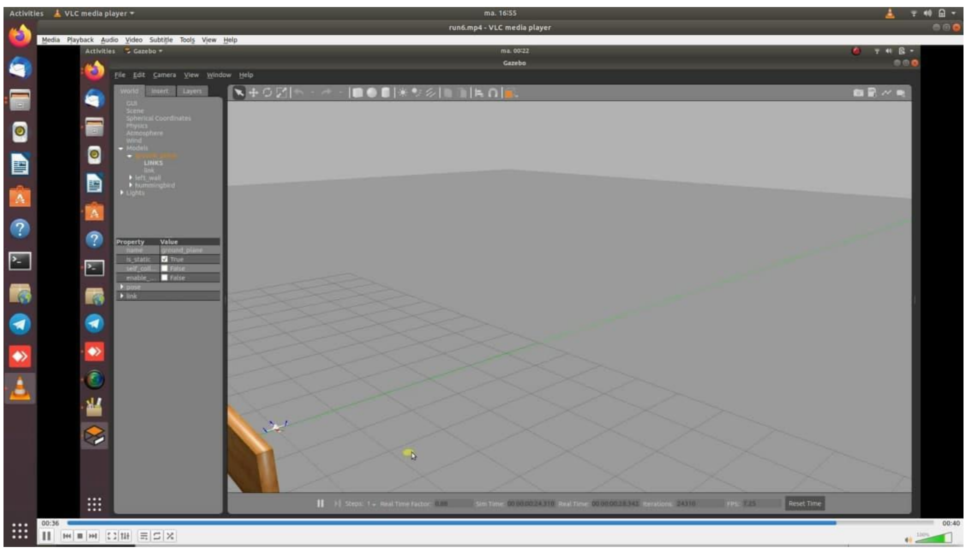

run6.mp4 - VLC media player
Gazebo


Activities
VLC media player
ma. 16:54
run6.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
Gazebo
File Edit Camera View Window Help
Reset Time
00:28
00:40


Activities
VLC media player
ma. 16:54
run6.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
Gazebo
File Edit Camera View Window Help
Reset Time
00:26
00:40


Activities
VLC media player
ma. 16:54
run6.mp4 - VLC media player
Media Playback Audio Video Subtitle Tools View Help
Gazebo
File Edit Camera View Window Help
Reset Time
00:23
00:40





5. Discussion

Flightmare was indeed an interesting simulator to work with, as predicted beforehand. Our own implementations regarding the design and pathplanning went according to the plan, after some struggles. When it came to the change of environment, there were some challenges before settling with the two walls as described above in the result demonstrations. Implementation of diverse environments caused delays in the simulations, which were time-consuming and not easy to work with. This resulted in the decision to go with the two walls, which allowed us to experiment with the code without to many delays interrupting the process. This resulted in a "general" solution for our project, but with the continuous delays and short time as two main factors, this solution was best suited in our situation.

When it came to the groupwork, there was definitely areas of improvement. It was challenging to arrange physical meetings, due to peoples work schedules and other classes. This resulted in a lot of digital communication, which we acknowledge as a factor to why the progression of the project was slow in the implementing phase. Better planning and communication in this area would probably lead to better time with the report and experimentation in the Flightmare simulation. But on the other side, the group dynamic was good, and everyone participated at an equal and engaging level. The cooperation was good, and the group helped each other in the areas where someone had a better understanding and experience within the work being put down in this project.

6. Conclusion

As a conclusion

Flightmare is a flexible modular quadrotor simulator. Flightmare is composed of two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. Flightmare comes with several desirable features: (i) a large multi-modal sensor suite, including an interface to extract the 3D point-cloud of the scene; (ii) an API for reinforcement learning which can simulate hundreds of quadrotors in parallel; and (iii) an integration with a virtual-reality headset for interaction with the simulated environment. Flightmare can be used with other multirotor models that comes with RotorS such as AscTec Hummingbird, the AscTec Pelican, or the AscTec Firefly. The default controller in rpg_quadrotor_control is a PID controller. The RPG Quadrotor Control repository provides packages that are intended to be used with ROS. This repository contains a complete framework for flying quadrotors based on control algorithms and PID controllers. We design and pathplanning the quarotor and also change the environment to show our path planning.

As mentioned in the discussion chapter there were challenges with the time available for the project, due to other assignments that needed first prioritization in the early phase. This, and the struggles with the installation of the programs resulted in the group not having the time to fully experiment with the simulator. We took a decision to mainly focus on the report, which is the reason there obviously were room for further work being put down to make the final simulation more of our own. We found the available code in the Flightmare repository to be a little advanced, and implementing too many changes often led to getting several error messages that ads up. It might have been easier to fix and understand these errors if it had been possible to work in classrooms so different groups could collaborate more on common issues and ask the teacher for help in person, but this was not possible due to the virus situation. Even though the online help on Discord helped a lot, it was not always so practical and efficient. Will write this different in the end...

But with that being said, we found this project were useful and educational in the use of ROS/Gazebo, and conclude that the learning outcomes from this time working on the project has been very successful. We are left with a new insight in the possibilities and utility value of the simulators, and are exited for the further possible research in this area. 😉 🙁

Describing the code:

```cpp
#include "rpg_quadrotor_integration_test/rpg_quadrotor_integration_test.h"

#include <gtest/gtest.h>
#include <vector>

#include <autopilot/autopilot_states.h>
#include <polynomial_trajectories/polynomial_trajectory_settings.h>
#include <quadrotor_common/control_command.h>
#include <quadrotor_common/geometry_eigen_conversions.h>
#include <std_msgs/Bool.h>
#include <trajectory_generation_helper/heading_trajectory_helper.h>
#include <trajectory_generation_helper/polynomial_trajectory_helper.h>
#include <Eigen/Dense>
```

It is the first part of the code and is about import the libraries and functions we need it.

```cpp
namespace rpg_quadrotor_integration_test {

QuadrotorIntegrationTest::QuadrotorIntegrationTest()
    : executing_trajectory_(false),
      sum_position_error_squared_(0.0),
      max_position_error_(0.0),
      sum_thrust_direction_error_squared_(0.0),
      max_thrust_direction_error_(0.0) {
  ros::NodeHandle nh;

  arm_pub_ = nh.advertise<std_msgs::Bool>("bridge/arm", 1);

  measure_tracking_timer_ =
      nh_.createTimer(ros::Duration(1.0 / kExecLoopRate_),
                      &QuadrotorIntegrationTest::measureTracking, this);
}
```

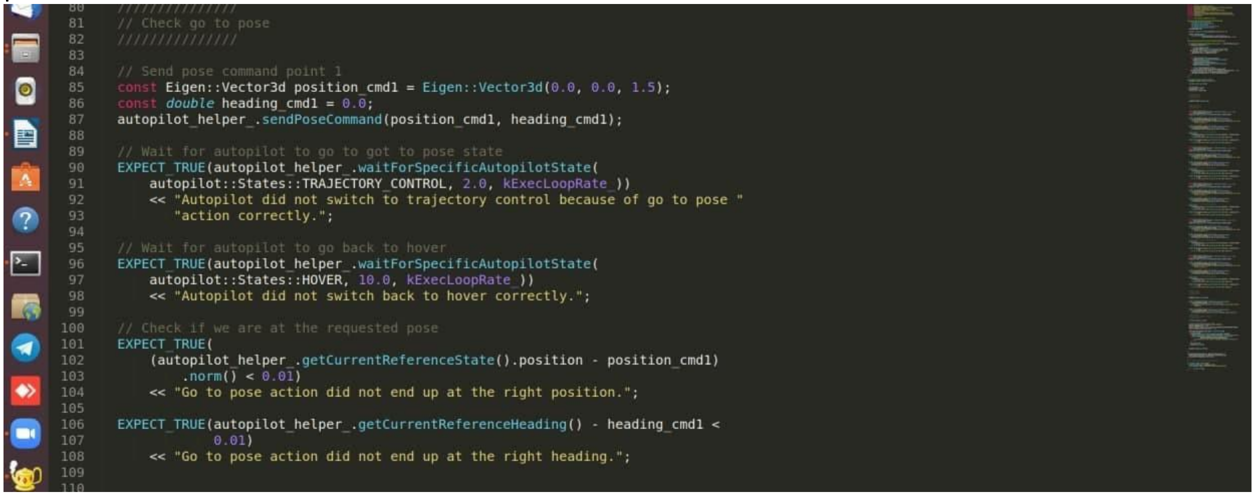
It is about initializing the member of class

```cpp
QuadrotorIntegrationTest::~QuadrotorIntegrationTest() {}

void QuadrotorIntegrationTest::measureTracking(const ros::TimerEvent& time) {
  if (executing_trajectory_) {
    // Position error
    const double position_error =
        autopilot_helper_.getCurrentPositionError().norm();
    sum_position_error_squared_ += pow(position_error, 2.0);
    if (position_error > max_position_error_) {
      max_position_error_ = position_error;
    }

    // Thrust direction error
    const Eigen::Vector3d ref_thrust_direction =
        autopilot_helper_.getCurrentReferenceOrientation() *
        Eigen::Vector3d::UnitZ();
    const Eigen::Vector3d thrust_direction =
        autopilot_helper_.getCurrentOrientationEstimate() *
        Eigen::Vector3d::UnitZ();

    const double thrust_direction_error =
        acos(ref_thrust_direction.dot(thrust_direction));
    sum_thrust_direction_error_squared_ += pow(thrust_direction_error, 2.0);
    if (thrust_direction_error > max_thrust_direction_error_) {
      max_thrust_direction_error_ = thrust_direction_error;
    }
  }
}
```

It is also about position error and thrust direction error and also have measure tracking

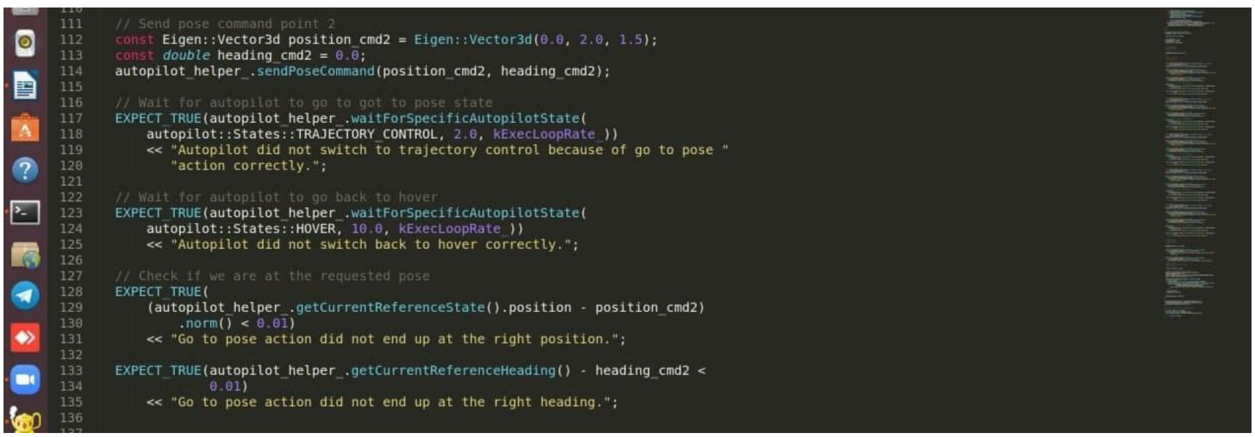
```cpp
    const double thrust_direction_error =
        acos(ref_thrust_direction.dot(thrust_direction));
    sum_thrust_direction_error_squared_ += pow(thrust_direction_error, 2.0);
    if (thrust_direction_error > max_thrust_direction_error_) {
      max_thrust_direction_error_ = thrust_direction_error;
    }
  }
}

void QuadrotorIntegrationTest::run() {
  ros::Rate command_rate(kExecLoopRate_);

  ros::Duration(5.0).sleep();

  // Arm bridge
  std_msgs::Bool arm_msg;
  arm_msg.data = true;
  arm_pub_.publish(arm_msg);

  ///////////////
  // Check take off
  ///////////////

  // Takeoff for real
  autopilot_helper_.sendStart();
```

There is start and run the quadrotor and start the armbridge of quadrotor after 5 seconds, and also we put it to have time to start recorder.



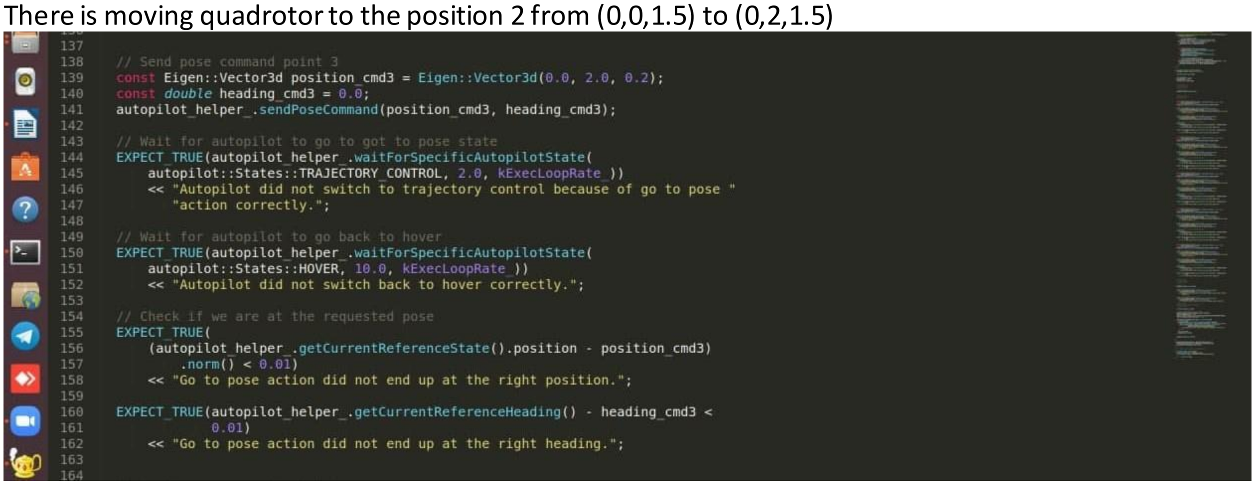
```
// Check go to pose
///////////////

// Send pose command point 1
const Eigen::Vector3d position_cmd1 = Eigen::Vector3d(0.0, 0.0, 1.5);
const double heading_cmd1 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd1, heading_cmd1);

// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";

// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";

// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd1)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";

EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd1 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
```

There is moving quadrotor from the first position from (0,0,0) to (0,0,1.5)



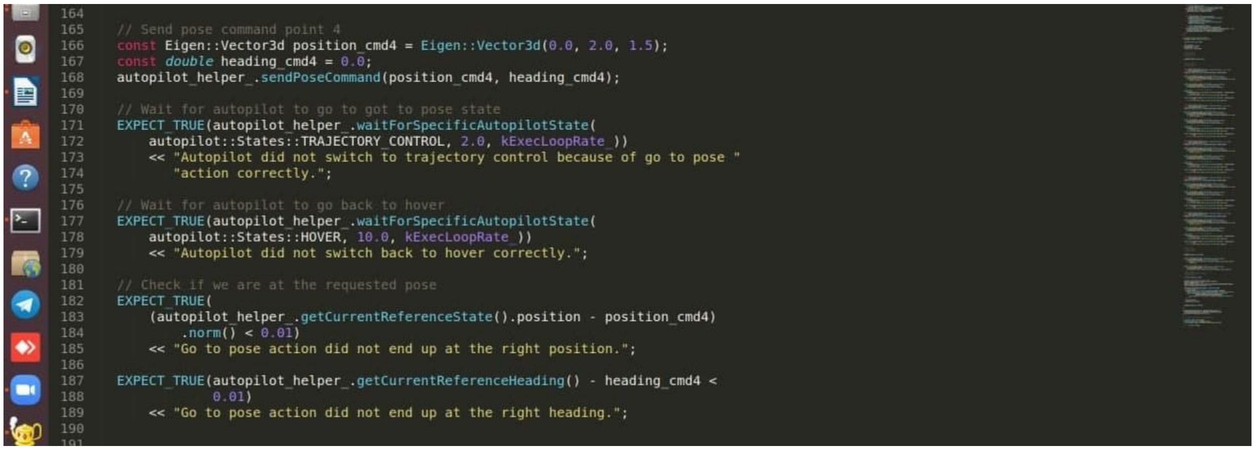
```
// Send pose command point 2
const Eigen::Vector3d position_cmd2 = Eigen::Vector3d(0.0, 2.0, 1.5);
const double heading_cmd2 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd2, heading_cmd2);

// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";

// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";

// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd2)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";

EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd2 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
```

There is moving quadrotor to the position 2 from (0,0,1.5) to (0,2,1.5)



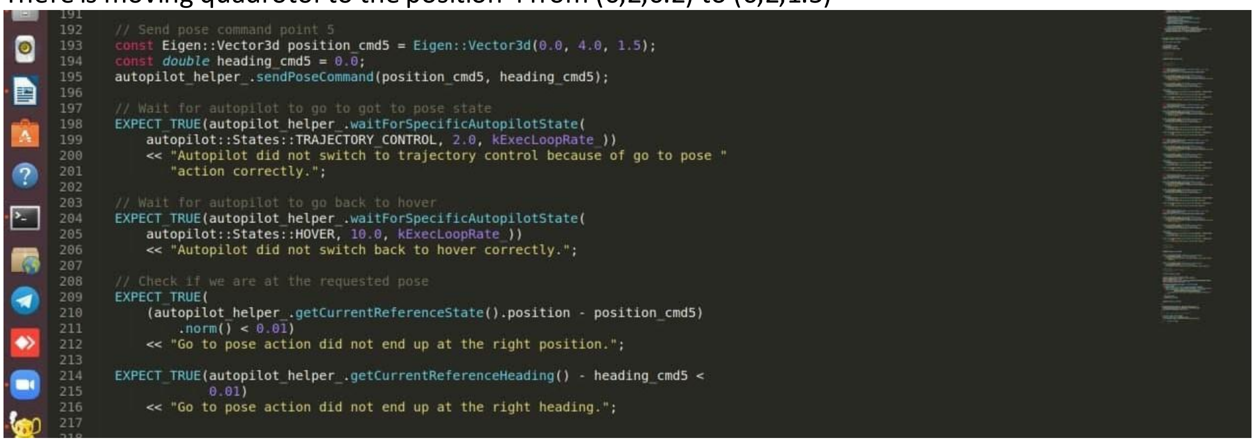
```
// Send pose command point 3
const Eigen::Vector3d position_cmd3 = Eigen::Vector3d(0.0, 2.0, 0.2);
const double heading_cmd3 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd3, heading_cmd3);

// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";

// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";

// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd3)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";

EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd3 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
```

There is moving quadrotor to the position 3 from (0,2,1.5) to (0,2,0.2)



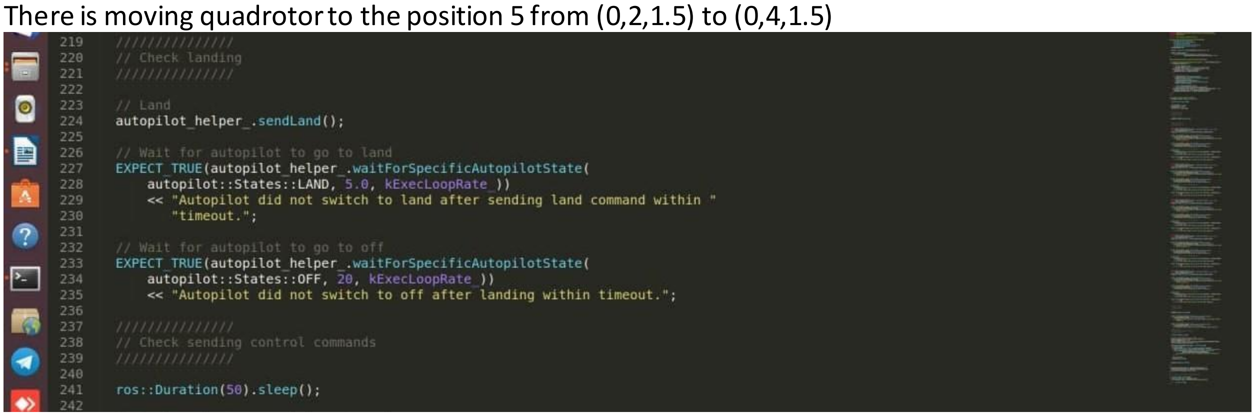
```
// Send pose command point 4
const Eigen::Vector3d position_cmd4 = Eigen::Vector3d(0.0, 2.0, 1.5);
const double heading_cmd4 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd4, heading_cmd4);

// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";

// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";

// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd4)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";

EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd4 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
```

There is moving quadrotor to the position 4 from (0,2,0.2) to (0,2,1.5)



```
// Send pose command point 5
const Eigen::Vector3d position_cmd5 = Eigen::Vector3d(0.0, 4.0, 1.5);
const double heading_cmd5 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd5, heading_cmd5);

// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";

// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";

// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd5)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";

EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd5 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
```

There is moving quadrotor to the position 5 from (0,2,1.5) to (0,4,1.5)



```
///////////////
// Check landing
///////////////

// Land
autopilot_helper_.sendLand();

// Wait for autopilot to go to land
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::LAND, 5.0, kExecLoopRate_))
    << "Autopilot did not switch to land after sending land command within "
       "timeout.";

// Wait for autopilot to go to off
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::OFF, 20, kExecLoopRate_))
    << "Autopilot did not switch to off after landing within timeout.";

///////////////
// Check sending control commands
///////////////

ros::Duration(50).sleep();
```

This is for landing option and allow the quadrotor to land after that off it after 20 seconds and after 50 seconds go to sleep

```
  //////////////
  // Check sending control commands
  //////////////

  ros::Duration(50).sleep();

  // Send control command to spin motors
  quadrotor_common::ControlCommand control_command;
  control_command.armed = true;
  control_command.control_mode = quadrotor_common::ControlMode::BODY_RATES;
  control_command.collective_thrust = 10.0;

  ros::Time start_sending_cont_cmds = ros::Time::now();
  while (ros::ok()) {
    autopilot_helper_.sendControlCommandInput(control_command);
    if ((ros::Time::now() - start_sending_cont_cmds) > ros::Duration(0.5)) {
      EXPECT_TRUE((autopilot_helper_.getCurrentAutopilotState() ==
                   autopilot::States::COMMAND_FEEDTHROUGH))
          << "Autopilot did not switch to command feedthrough correctly.";
      break;
    }
    ros::spinOnce();
    command_rate.sleep();
  }

  autopilot_helper_.sendOff();

}
```

This is the control commands to spin motors for quadrotor and off them

```
}

TEST(QuadrotorIntegrationTest, AutopilotFunctionality) {
  QuadrotorIntegrationTest rpg_quadrotor_integration_test;
  rpg_quadrotor_integration_test.run();
}

}  // namespace rpg_quadrotor_integration_test

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  ros::init(argc, argv, "rpg_quadrotor_integration_test");

  return RUN_ALL_TESTS();
}
```

This is the main code that run all of the class and parameters and run the quadrotor.

7. References

[1] Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, Davide Scaramuzza. Flightmare: A Flexible Quadrotor Simulator. Conference on Robot Learning, Cambridge MA, USA. 2020.

[2] Website: https://uzh-rpg.github.io/flightmare

[3] Website: https://github.com/uzh-rpg/flightmare

[4] Website : https://github.com/uzh-rpg/rpg_quadrotor_control

[5] Website : https://stackoverflow.com/

[6] Website : http://wiki.ros.org/

[7] Website : https://github.com/uzh-rpg/rpg_mpc/wiki/Basic-Usage

Codes

```
#include "rpg_quadrotor_integration_test/rpg_quadrotor_integration_test.h"
#include <gtest/gtest.h>
#include <vector>
#include <autopilot/autopilot_states.h>
#include <polynomial_trajectories/polynomial_trajectory_settings.h>
#include <quadrotor_common/control_command.h>
#include <quadrotor_common/geometry_eigen_conversions.h>
```

```
#include <std_msgs/Bool.h>
#include <trajectory_generation_helper/heading_trajectory_helper.h>
#include <trajectory_generation_helper/polynomial_trajectory_helper.h>
#include <Eigen/Dense>
namespace rpg_quadrotor_integration_test {
QuadrotorIntegrationTest::QuadrotorIntegrationTest()
    : executing_trajectory_(false),
      sum_position_error_squared_(0.0),
      max_position_error_(0.0),
      sum_thrust_direction_error_squared_(0.0),
      max_thrust_direction_error_(0.0) {
  ros::NodeHandle nh;
  arm_pub_ = nh.advertise<std_msgs::Bool>("bridge/arm", 1);
  measure_tracking_timer_ =
      nh_.createTimer(ros::Duration(1.0 / kExecLoopRate_),
                      &QuadrotorIntegrationTest::measureTracking, this);
}
QuadrotorIntegrationTest::~QuadrotorIntegrationTest() {}
void QuadrotorIntegrationTest::measureTracking(const ros::TimerEvent& time) {
  if (executing_trajectory_) {
    // Position error
    const double position_error =
        autopilot_helper_.getCurrentPositionError().norm();
    sum_position_error_squared_ += pow(position_error, 2.0);
    if (position_error > max_position_error_) {
      max_position_error_ = position_error;
    }
    // Thrust direction error
    const Eigen::Vector3d ref_thrust_direction =
        autopilot_helper_.getCurrentReferenceOrientation() *
        Eigen::Vector3d::UnitZ();
    const Eigen::Vector3d thrust_direction =
        autopilot_helper_.getCurrentOrientationEstimate() *
        Eigen::Vector3d::UnitZ();
    const double thrust_direction_error =
        acos(ref_thrust_direction.dot(thrust_direction));
    sum_thrust_direction_error_squared_ += pow(thrust_direction_error, 2.0);
    if (thrust_direction_error > max_thrust_direction_error_) {
      max_thrust_direction_error_ = thrust_direction_error;
    }
  }
}
void QuadrotorIntegrationTest::run() {
  ros::Rate command_rate(kExecLoopRate_);
  ros::Duration(5.0).sleep();
  // Arm bridge
  std_msgs::Bool arm_msg;
  arm_msg.data = true;
  arm_pub_.publish(arm_msg);
  ///////////////
  // Check take off
  ///////////////
  // Takeoff for real
  autopilot_helper_.sendStart();
  ///////////////
  // Check go to pose
  ///////////////
  // Send pose command point 1
  const Eigen::Vector3d position_cmd1 = Eigen::Vector3d(0.0, 0.0, 1.5);
  const double heading_cmd1 = 0.0;
  autopilot_helper_.sendPoseCommand(position_cmd1, heading_cmd1);
  // Wait for autopilot to go to got to pose state
  EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
```

```
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";
// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";
// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd1)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";
EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd1 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
// Send pose command point 2
const Eigen::Vector3d position_cmd2 = Eigen::Vector3d(0.0, 2.0, 1.5);
const double heading_cmd2 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd2, heading_cmd2);
// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";
// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";
// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd2)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";
EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd2 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
// Send pose command point 3
const Eigen::Vector3d position_cmd3 = Eigen::Vector3d(0.0, 2.0, 0.2);
const double heading_cmd3 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd3, heading_cmd3);
// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";
// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";
// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd3)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";
EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd3 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
// Send pose command point 4
const Eigen::Vector3d position_cmd4 = Eigen::Vector3d(0.0, 2.0, 1.5);
const double heading_cmd4 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd4, heading_cmd4);
// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
```

```
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";
// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";
// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd4)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";
EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd4 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
// Send pose command point 5
const Eigen::Vector3d position_cmd5 = Eigen::Vector3d(0.0, 4.0, 1.5);
const double heading_cmd5 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd5, heading_cmd5);
// Wait for autopilot to go to got to pose state
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::TRAJECTORY_CONTROL, 2.0, kExecLoopRate_))
    << "Autopilot did not switch to trajectory control because of go to pose "
       "action correctly.";
// Wait for autopilot to go back to hover
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::HOVER, 10.0, kExecLoopRate_))
    << "Autopilot did not switch back to hover correctly.";
// Check if we are at the requested pose
EXPECT_TRUE(
    (autopilot_helper_.getCurrentReferenceState().position - position_cmd5)
        .norm() < 0.01)
    << "Go to pose action did not end up at the right position.";
EXPECT_TRUE(autopilot_helper_.getCurrentReferenceHeading() - heading_cmd5 <
            0.01)
    << "Go to pose action did not end up at the right heading.";
///////////////
// Check landing
///////////////
// Land
autopilot_helper_.sendLand();
// Wait for autopilot to go to land
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::LAND, 5.0, kExecLoopRate_))
    << "Autopilot did not switch to land after sending land command within "
       "timeout.";
// Wait for autopilot to go to off
EXPECT_TRUE(autopilot_helper_.waitForSpecificAutopilotState(
    autopilot::States::OFF, 20, kExecLoopRate_))
    << "Autopilot did not switch to off after landing within timeout.";
///////////////
// Check sending control commands
///////////////
ros::Duration(50).sleep();
// Send control command to spin motors
quadrotor_common::ControlCommand control_command;
control_command.armed = true;
control_command.control_mode = quadrotor_common::ControlMode::BODY_RATES;
control_command.collective_thrust = 10.0;
ros::Time start_sending_cont_cmds = ros::Time::now();
while (ros::ok()) {
  autopilot_helper_.sendControlCommandInput(control_command);
  if ((ros::Time::now() - start_sending_cont_cmds) > ros::Duration(0.5)) {
```

```
      EXPECT_TRUE((autopilot_helper_.getCurrentAutopilotState() ==
                   autopilot::States::COMMAND_FEEDTHROUGH))
          << "Autopilot did not switch to command feedthrough correctly.";
      break;
    }
    ros::spinOnce();
    command_rate.sleep();
  }
  autopilot_helper_.sendOff();
}
TEST(QuadrotorIntegrationTest, AutopilotFunctionality) {
  QuadrotorIntegrationTest rpg_quadrotor_integration_test;
  rpg_quadrotor_integration_test.run();
}
}  // namespace rpg_quadrotor_integration_test
int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  ros::init(argc, argv, "rpg_quadrotor_integration_test");
  return RUN_ALL_TESTS();
}
```