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: autopilot, drone simulator, flightmare, gazebo, photo-realistic rendering, quadrotor simulator, ROS, simulator, 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. The solutions provided by the project will be described and demonstrated in this final project report.

Our group went with Flightmare, a quadrotor simulator developed by five students from the University of Zurich (1). 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 made by our project group. The final result will be presented throughout this report, by the end of November 25th.

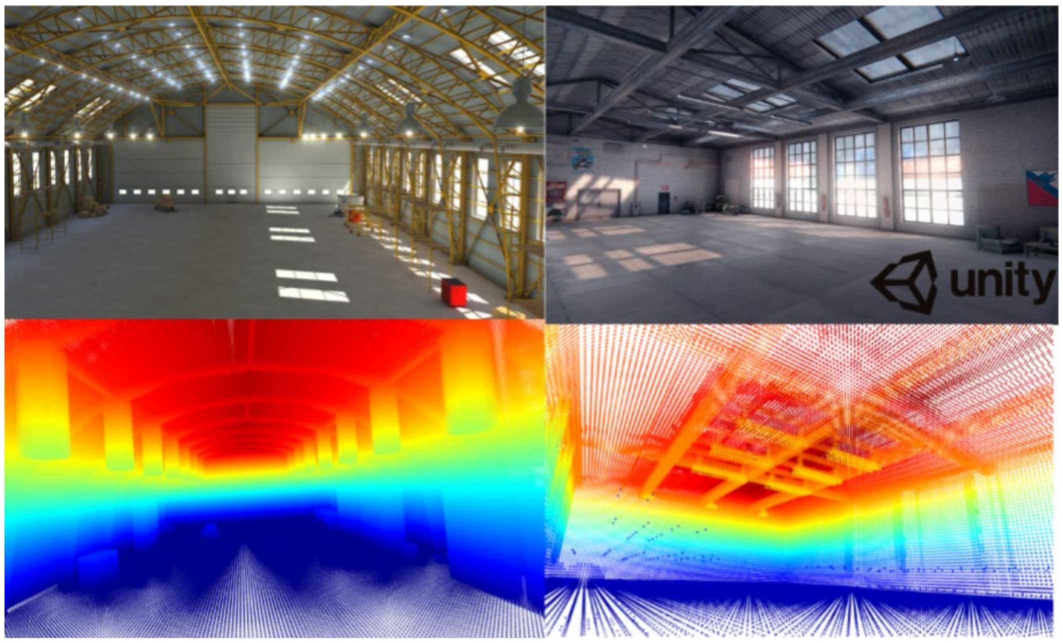


FIGURE 1: A DEMONSTRATION OF THE VISUAL POSSIBILITIES WITH THE FLIGHTMARE SIMULATOR (1)

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 in this project later on in the report.

2.1 VMware 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 (2).

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 (2).

VMware Workstation 16 Player
Player
Home
Welcome to VMware Workstation 16 Player
Create a New Virtual Machine
Create a new virtual machine, which will then be added to the top of your library.
Open a Virtual Machine
Open an existing virtual machine, which will then be added to the top of your library.
Download a Virtual Appliance
Download a virtual appliance from the marketplace. You can then open it in Player.
Help
View online help.
vmware
WORKSTATION
PLAYER™
16

FIGURE 2: VMWARE WORKSTATION 16 PLAYER (2)

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 (3).

FIGURE 3: UBUNTU (4)

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 (5).

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 (6).

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 (7).

FIGURE 4: ROS MELODIC MORENIA (7)

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 are releases that are still supported, such as ROS Melodic Morenia (8).

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)

FIGURE 5: ROS DISTRIBUTIONS (8)

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 across 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 (9).

2.4 Gazebo

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 (10). The latest version of Gazebo is the 11.0.0 version, and it was released in January 2019.

FIGURE 6: GAZEBO (11)

Gazebo has the following main components (12):

- 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.

2.4.1 RotorS

RotorS is a MAV gazebo simulator that provides a series of multirotor models, such as:

- AscTec Hummingbird
- AscTec Pelican
- AscTec Firefly

Further information about these multirotors can be found in reference (13), and it is important to mention that the simulator is not limited to these three multirotors. The sensors included with the simulator is an IMU, a generic odometry sensor and a Visual-Internal (VI) sensor. More information about the VI-sensor can also be found in reference (14).

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 (15). Both the rendering engine and the dynamics simulation are flexible by design, as further described in the chapters below.

2.5.1 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. It is also possible to simulate sensor noise, motion blur, environment dynamics, wind, lens distortions etc.

2.5.2 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

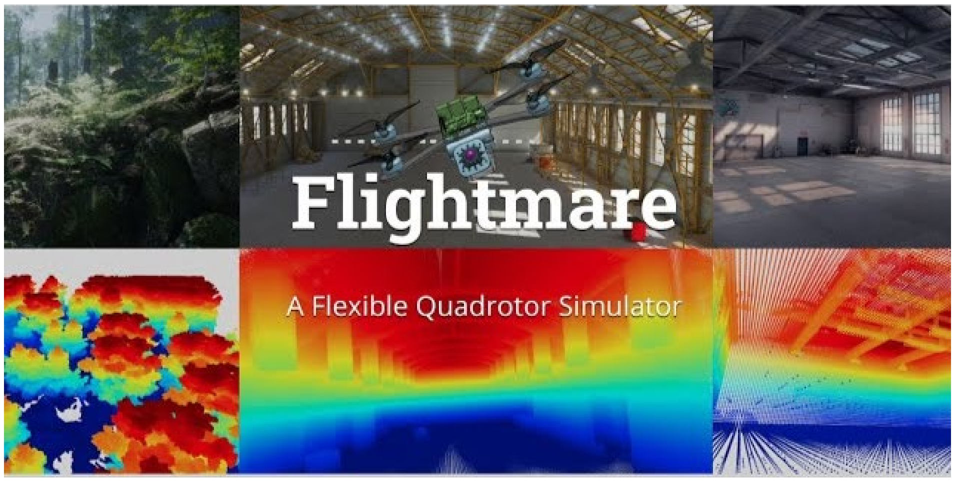


FIGURE 7: FLIGHTMARE (16)

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

2.5.3 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
- ROS integration, including interface to the popular Gazebo-based MAV simulator (RotorS)
- Interface to Model-based quadrotor control

2.5.4 Application areas

Some of Flightmares 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

Flightmare provides interfaces to the 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 (1).

3. Methods

3.1 Choosing a project

The first step in the project was choosing a project topic. In Canvas, there were presented several project ideas, where our group found the drone simulator in Flightmare interesting. The quadrotor simulator seemed like a project that would fit our competence level, where everyone would be able to work and participate together as a group. But most important it also seemed like an interesting topic to write about, which was the key factor with the mindset of completing and handling in an interesting project report at the end of the semester.

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

After choosing the project topic using the simulator Flightmare, the next step was to install the necessities. This as a general overview included VMware, Ubuntu, Gazebo and Flightmare. The process of the different installments was a challenging task in itself, due to continuous struggles with different errors throughout the steps.

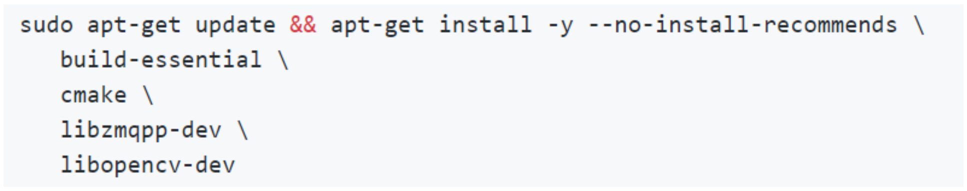
The Flightmare GitHub page, collected from the Flightmare research paper (1), was very helpful with providing all the steps to how to get Flightmare up and running.

3.2.1 Installing VMware

With the installment of VMware, the only step was downloading the application from the VMware Workstation 16 Player website (2), which succeeded with no struggles.

3.2.2 Installing 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 their GitHub page.

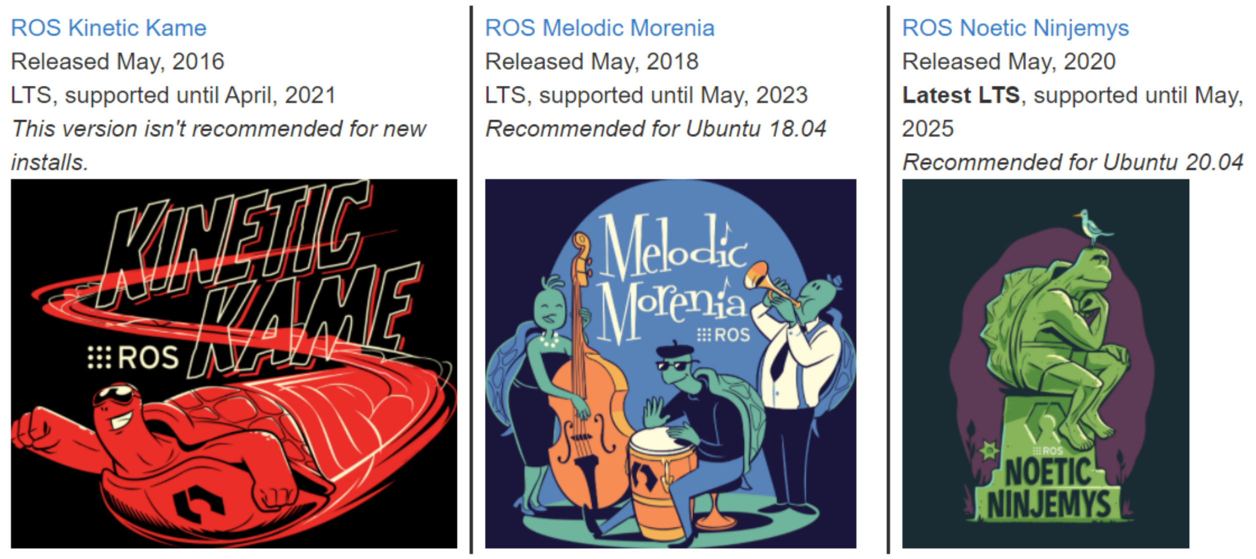


```
sudo apt-get update && apt-get install -y --no-install-recommends \
   build-essential \
   cmake \
   libzmqpp-dev \
   libopencv-dev
```

FIGURE 8: INSTALLMENT OF PREREQUISITES PACKAGES

3.2.3 Ubuntu installing 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 for Flightmare (17).



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

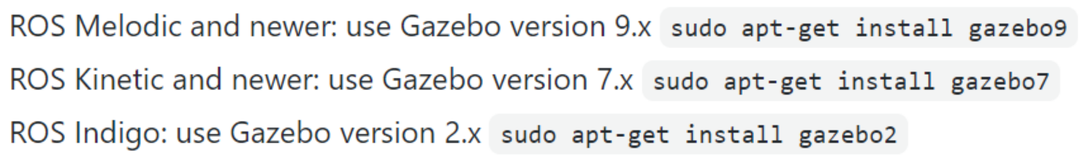
ROS Noetic Ninjemys
Released May, 2020
Latest LTS, supported until May, 2025
Recommended for Ubuntu 20.04

FIGURE 9: ROS DISTRIBUTION VERSIONS

After choosing to install ROS Melodic Morenia, the steps provided in (18) was further followed to complete the installment.

3.2.4 Gazebo

Since ROS Melodic was used in this project, Gazebo version 9.x was the correct version to be 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.

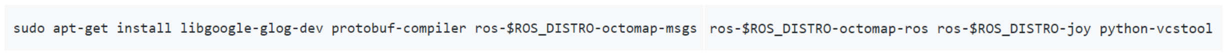


ROS Melodic and newer: use Gazebo version 9.x `sudo apt-get install gazebo9`
ROS Kinetic and newer: use Gazebo version 7.x `sudo apt-get install gazebo7`
ROS Indigo: use Gazebo version 2.x `sudo apt-get install gazebo2`

FIGURE 10: INSTALLING GAZEBO VERSION 9.X

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" (17).

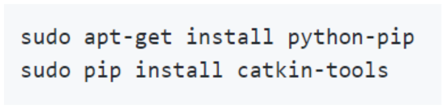


```
sudo apt-get install libgoogle-glog-dev protobuf-compiler ros-$ROS_DISTRO-octomap-msgs ros-$ROS_DISTRO-octomap-ros ros-$ROS_DISTRO-joy python-vcstool
```

FIGURE 11: INSTALLING ROS AND DEPENDENCIES

3.2.6 Getting catkin tools

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

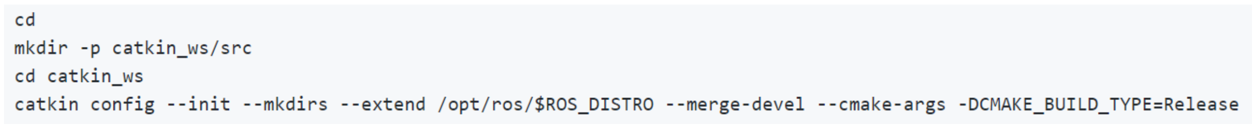


```
sudo apt-get install python-pip
sudo pip install catkin-tools
```

FIGURE 12: CATKIN TOOLS

3.2.7 Creating a catkin workspace

The catkin workspace was created with the following commands shown in Figure 13.

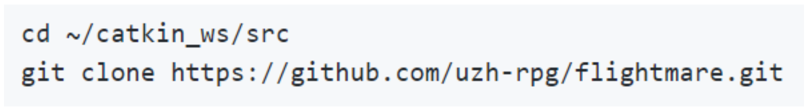


```
cd
mkdir -p catkin_ws/src
cd catkin_ws
catkin config --init --mkdirs --extend /opt/ros/$ROS_DISTRO --merge-devel --cmake-args -DCMAKE_BUILD_TYPE=Release
```

FIGURE 13: CATKIN WORKSPACE

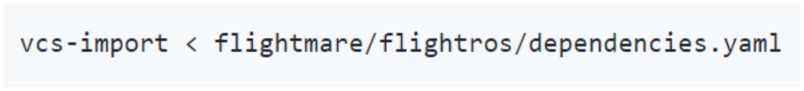
3.2.8 Installing Flightmare

With the installment of Flightmare, the repository and dependencies was cloned.



```
cd ~/catkin_ws/src
git clone https://github.com/uzh-rpg/flightmare.git
```

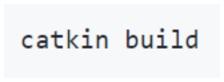
FIGURE 14: CLONING THE REPOSITORY



```
vcs-import < flightmare/flightros/dependencies.yaml
```

FIGURE 15: CLONING THE DEPENDENCIES

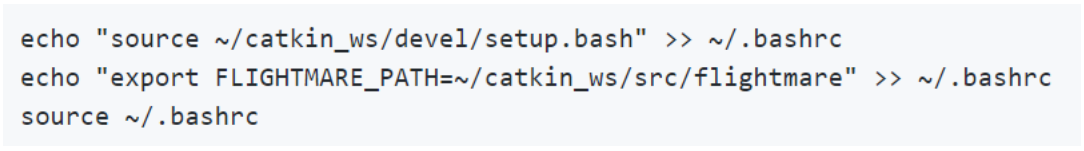
Then the packages were built in the catkin workspace, as shown is Figure 16. A problem that occurred with using the catkin build command is that it is different and more extensive than the catkin make command. Since the ROS environment was already setup from the ROS melodic installation procedure with cmake, the workspace needed to be deleted prior to using catkin build for the flightmare files to be used. The build failed several times after using *catkin clear* due to a missing package. Googling eventually revealed that using the command *pip3 install catkin_pkg* to fetch the missing package and upgrading to *Python3-pip* with the *sudo apt intall* command would solve the build issue.



```
catkin build
```

FIGURE 16: BUILDING PACKAGES TO THE CATKIN WORKSPACE

At last the sourcing of the catkin workspace and the FLIGHTMARE_PATH environment variable was added to the *.bashrc* file.



```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
echo "export FLIGHTMARE_PATH=~/catkin_ws/src/flightmare" >> ~/.bashrc
source ~/.bashrc
```

FIGURE 17: ADDING CATKIN_WS AND FLIGHTMARE_PATH TO THE .BASHRC FILE

3.2.9 Downloading Flightmare Unity Binary

The Flightmare Unity Binary **RPG_Flightmare.tar.xz** was downloaded for rendering from the Releases and extracted into the */path/to/flightmare/flightrender*.

The steps regarding the installment of ROS above caused a lot of errors and was very timeconsuming. For example, there was a repeating issue with the unity connection timing out when launchin gazebo, and resulted in the program shutting down without warning. After googling error after error, and helping each other in the group, the installments was completed and Flightmare could finally be launched.

3.2.10 Launching Flightmare

As seen in Figure 18, Flightmare gets launched for the image rendering, where “RotorS” is the MAV gazebo simulator for the quadrotor dynamics modelling. RotorS provides the multirotor model AscTec Hummingbird for this project (13).

```
roslaunch flightros rotors_gazebo.launch
```

FIGURE 18: LAUNCHING FLIGHTMARE

After successfully launching Flightmare, the simulation started running as displayed in the figure below.

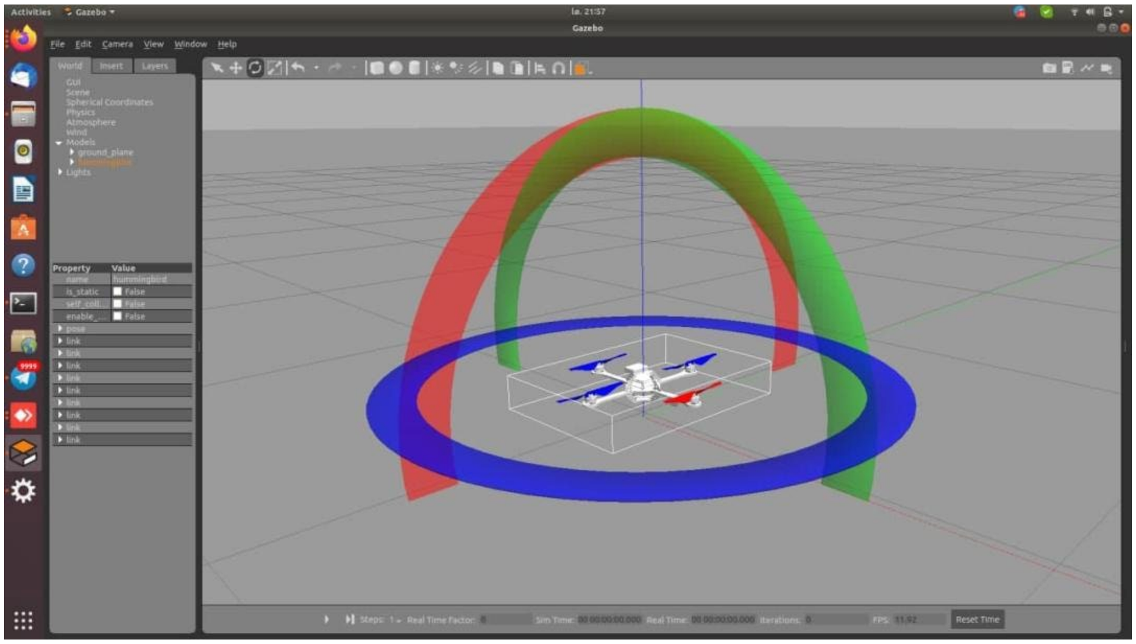
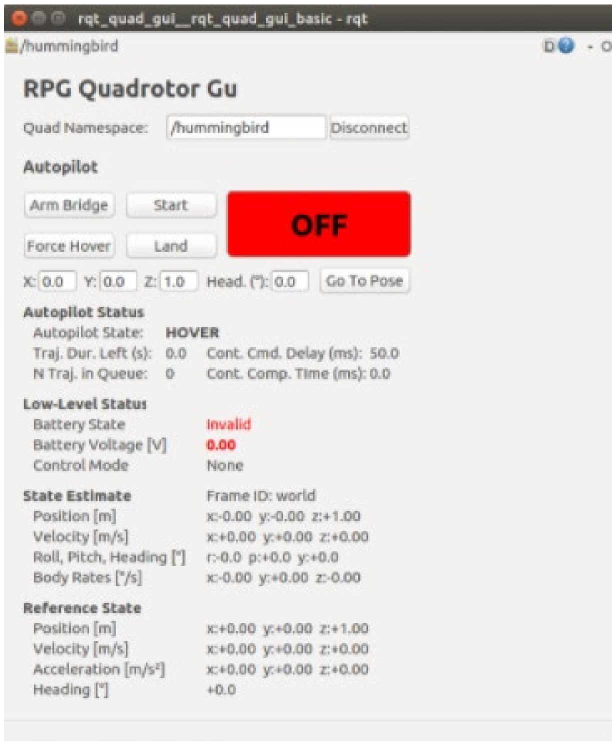


FIGURE 19: THE DISPLAY AFTER LAUNCHING FLIGHTMARE

Figure 19 shows the Hummingbird in the simulated area, and Figure 20 is the quadrotors manual controls.



```
RPG Quadrotor Gu
Quad Namespace: /hummingbird
Autopilot
X: 0.0  Y: 0.0  Z: 1.0  Head. (°): 0.0
Autopilot Status
  Autopilot State:    HOVER
  Traj. Dur. Left (s):  0.0   Cont. Cmd. Delay (ms): 50.0
  N Traj. in Queue:     0     Cont. Comp. Time (ms): 0.0
Low-Level Status
  Battery State              Invalid
  Battery Voltage [V]        0.00
  Control Mode               None
State Estimate               Frame ID: world
  Position [m]               x:-0.00  y:-0.00  z:+1.00
  Velocity [m/s]             x:+0.00  y:+0.00  z:+0.00
  Roll, Pitch, Heading [°]   r:-0.0  p:+0.0  y:+0.0
  Body Rates [°/s]           x:-0.00  y:+0.00  z:-0.00
Reference State
  Position [m]               x:+0.00  y:+0.00  z:+1.00
  Velocity [m/s]             x:+0.00  y:+0.00  z:+0.00
  Acceleration [m/s²]        x:+0.00  y:+0.00  z:+0.00
  Heading [°]                +0.0
```

FIGURE 20: HUMMINGBIRDS MANUAL CONTROLLER

To take off with the quadrotor, the connect/disconnect button had to be used to connect to the GUI. Then the "Arm Bridge" button armed the quadrotor in order to then press "Start" to take off. The bridge is an interface between the autopilot and the low-level controller for the Hummingbird. When arming the bridge, the control output from the autopilot are sent as messages though a node called SBUS bridgde, to the first-person view flight controller that runs the low-level controller. The X, Y, Z and "Head" fields could be used to tell the quadrotor what inputs to run.

3.3 Implementing robot simulations

As seen in Figure 19 above, the original Flightmare simulation consisted of an empty world. To experiment with the simulation, different environmental implementations were done to see how the Hummingbird would navigate. After some testing, two walls was implemented to the simulation as barriers for the quadrotor to use as path planning.

From the Flightmare research paper, we collected inspiration from an autopilot script that could be used to maneuver the Hummingbird across the two walls, by taking off and flying automatically. Five positions were implemented as path planning, with some additional codes to hover the Hummingbird before proceeding with the rest of the flight path.

Before succeeding with our implementations, which will be presented in the result chapter, there were some challenges. Throughout the implementations of the autopilot, there were some complications with the right inputs in order to avoid the obstacles. This can be seen in the figure below, where the Hummingbird failed to climb the wall, due to wrong inputs in the path planning.

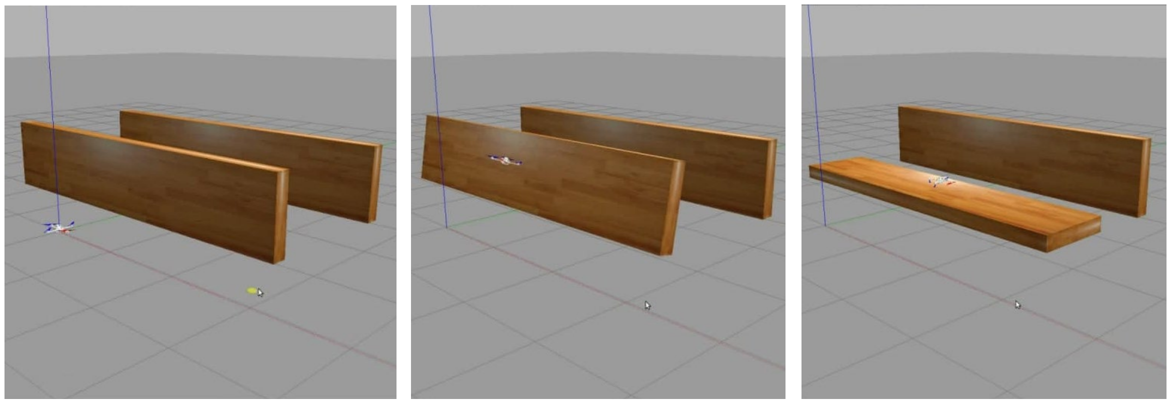


FIGURE 21: AUTOPILOT COMPLICATIONS

3.3.5 Model Predictive Controller for quadrotors (MPC)

There was also an attempt to implement 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 the rpg_quadrotor_control, to replace the already built in PID controller. This controller would allow 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 was that while the PID only had a single input and output (SISO), the MPC provided multiple inputs and outputs (MIMO).

There is worth mentioning that are several cool features regarding the perception-aware extension. To provide an example, the drone can be instructed to hover and circle different objects using the sensors. It can also recognize edges and corner of objects and use them as reference points.

However, there were issues with loading and running the MPC, due to conflicting build spaces 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 MPC was published in the research paper “PAMPC: Perception-Aware Model Predictive Control for Quadrotors (19).

4. Results

After some failed initial tests with getting the drone past the two walls, we managed to get the quadrotor to get from the starting position to the end position, traversing the two walls without colliding. Figure 22 displays the first half of the simulation, with the Hummingbird getting armed in the first picture.

The second picture shows the Hummingbird ascending above the wall, before crossing the first wall on the third picture and descending to a hover position right above the ground between the two walls in the fourth picture.

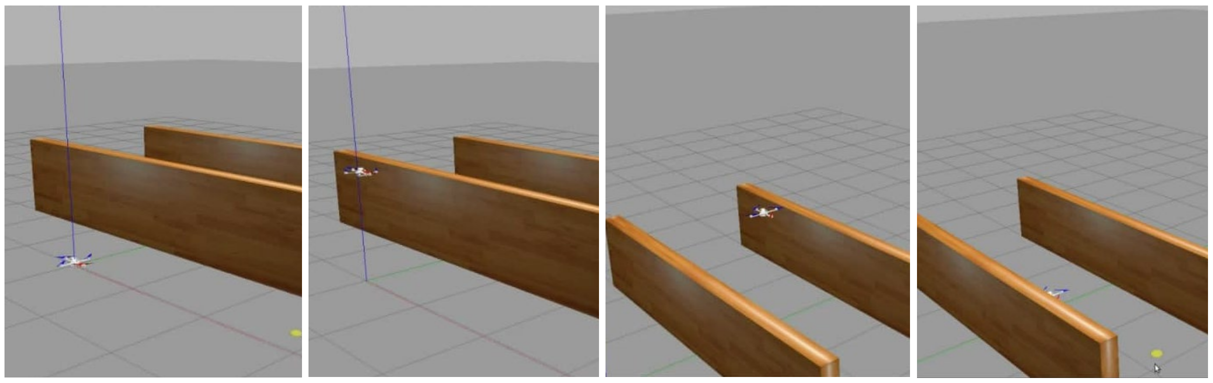


FIGURE 22: THE FIRST HALF OF THE SIMULATION

Then in Figure 23 the Hummingbird ascends again to the prior position, before crossing the second wall and descending again before landing. This was the last part of the simulation with Flightmare done in this project, which can be described as very basic in the form of what Flightmare is possible to achieve. But with our prior experience and time available for the project this simulation successfully displays the founding of what the simulator is capable of achieving with the use of autopilot and path planning.

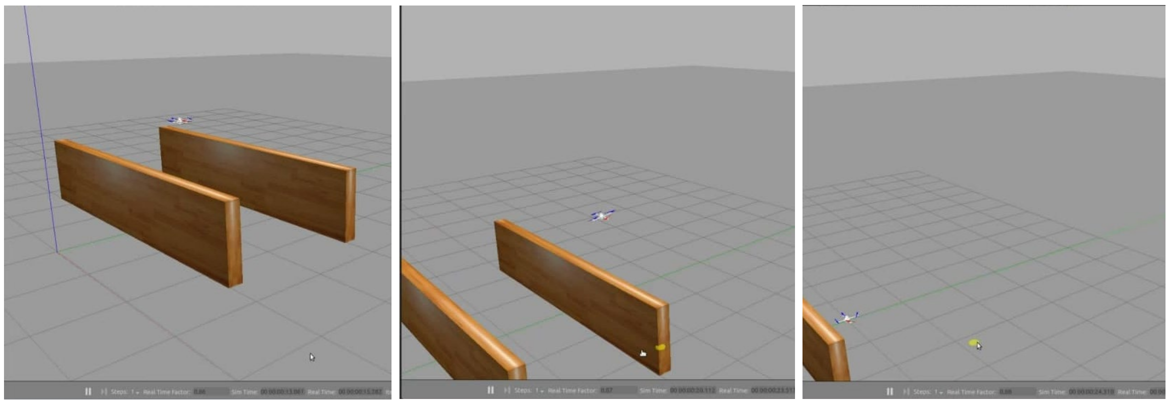


FIGURE 23: THE SECOND HALF OF THE SIMULATION

5. Discussion

With the result provided in the previous chapter, we can confirm that Flightmare indeed was an interesting simulator to work with. Our own implementations regarding the design and pathplanning went according to the plan, after some struggles. When it came to the change of the 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 was the reason for the groups decision to go for the two walls, which allowed us to experiment with the code without to many delays interrupting the process. The end simulation 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.

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. This, and the struggles with the installation of the programs resulted into 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. 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 group helped each other in the areas where someone had a better understanding and experience within the work being put down in this project, which provided a high ceiling for asking questions when needed.

6. Conclusion

We conclude with that the available code in the Flightmare repository was found to be a little advanced, and implementing too many changes often led to getting several error messages that accumulated. 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 obviously 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.

But with that being said, we found this project very 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.

7. Reference List

1. **Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, Davide Scaramuzza.** *Flightmare: A Flexible Quadrotor Simulator.* 2020.

2. **VMware.** VMware. *VMware Workstation 16 Player Release Notes.* [Online] 09 15, 2020. https://docs.vmware.com/en/VMware-Workstation-Player/16/rn/VMware-Workstation-16-Player-Release-Notes.html.

3. **Ubuntu.** TechnoPedia. *Ubuntu.* [Online] 03 14, 2017. https://www.techopedia.com/definition/3307/ubuntu.

4. **Ubuntu design.** Ubuntu design. *Downloads.* [Online] 2019. https://design.ubuntu.com/downloads/.

5. **ROS.org.** ROS.org. *Documentation.* [Online] 06 11, 2020. http://wiki.ros.org/.

6. —. ROS.org. *Introduction.* [Online] 08 08, 2018. http://wiki.ros.org/ROS/Introduction.

7. —. ROS.org. *ROS Melodic Morenia.* [Online] 08 14, 2018. http://wiki.ros.org/melodic.

8. —. ROS.org. *Distributions.* [Online] 06 11, 2020. http://wiki.ros.org/Distributions.

9. —. ROS.org. *Conceptual_overview.* [Online] March 26, 2020. http://wiki.ros.org/catkin/conceptual_overview.

10. **Gazebo.** Gazebosim. *Why Gazebo?* [Online] January 30, 2019. http://gazebosim.org/.

11. **Generation Robots.** Generation Robots. *Robotic simulation scenarios with Gazebo and ROS.* [Online] November 20, 2020. https://blog.generationrobots.com/en/robotic-simulation-scenarios-with-gazebo-and-ros/.

12. **The Construct.** The Construct. *[ROS in 5 mins] 028 - What is Gazebo simulation?* [Online] September 18, 2019. https://www.theconstructsim.com/ros-5-mins-028-gazebo-simulation/.

13. **ethz-asl.** GitHub. *RotorS.* [Online] August 14, 2020. https://github.com/ethz-asl/rotors_simulator.

14. **ROS.org.** ROS.org. *vi_sensor.* [Online] October 16, 2017. http://wiki.ros.org/vi_sensor.

15. **Robotics & Perception Group.** GitHub. *Flightmare.* [Online] 2020. https://uzh-rpg.github.io/flightmare/.

16. **University of Zurich.** ETH zurich. *Drone Racing.* [Online] 2020. http://rpg.ifi.uzh.ch/research_drone_racing.html.

17. **ROS.org.** ROS.org. *ROS Installation Options.* [Online] June 11, 2020. http://wiki.ros.org/ROS/Installation.

18. —. ROS.org. *Ubuntu install of ROS Melodic.* [Online] March 25, 2020. http://wiki.ros.org/melodic/Installation/Ubuntu.

19. **Davide Falanga, Philipp Foehn, Peng Lu, Davide Scaramuzza.** *PAMPC: Perception-Aware Model Predictie Control for Quadrotors.* Madrid : IROS, 2018.

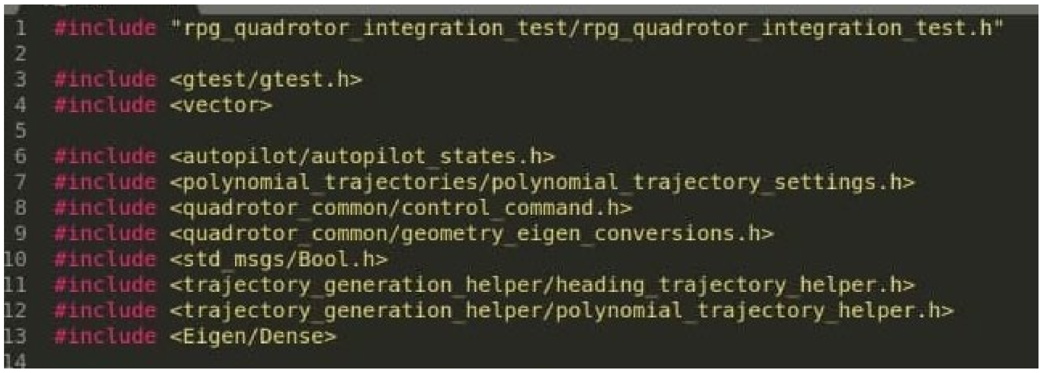
20. **ROS.org.** ROS.org. *Package Summary.* [Online] July 26, 2017. http://wiki.ros.org/catkin.

8. Figure list

9. Appendix 1 (Description of the code)

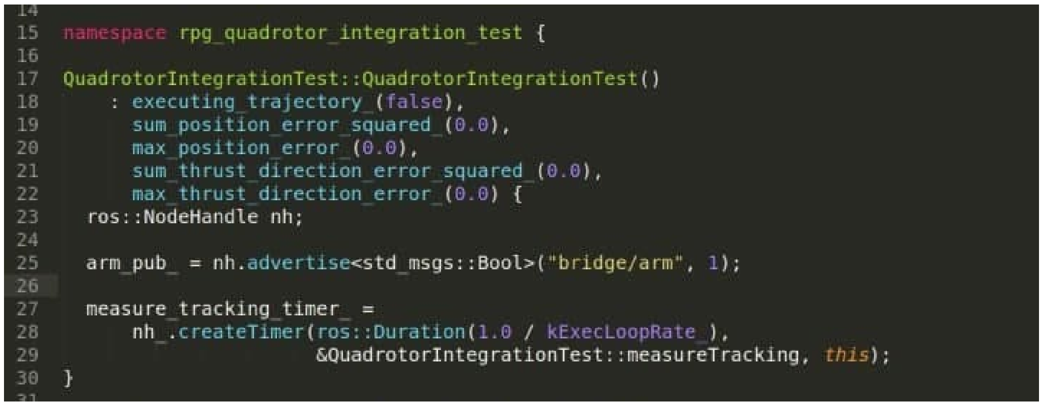
This appendix will give a short description of the code used with the Flightmare simulator, with the descriptions placed above every code section.

This first section displays the import of the necessary libraries and functions:



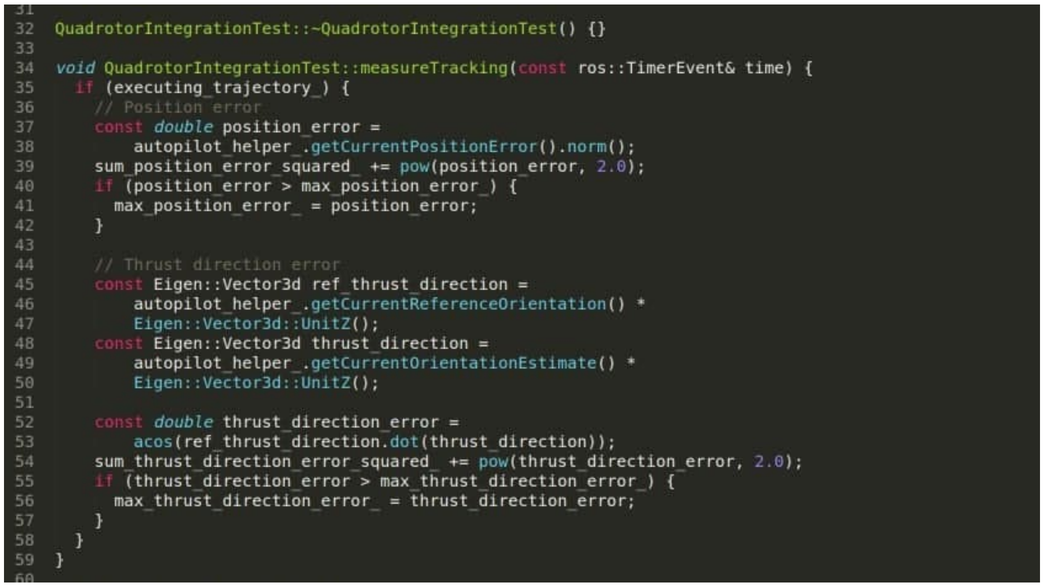
```
#include "rpg_quadrotor_integration_test/rpg_quadrotor_integration_test.h"

#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>

```

Then the initializing of the member of class:



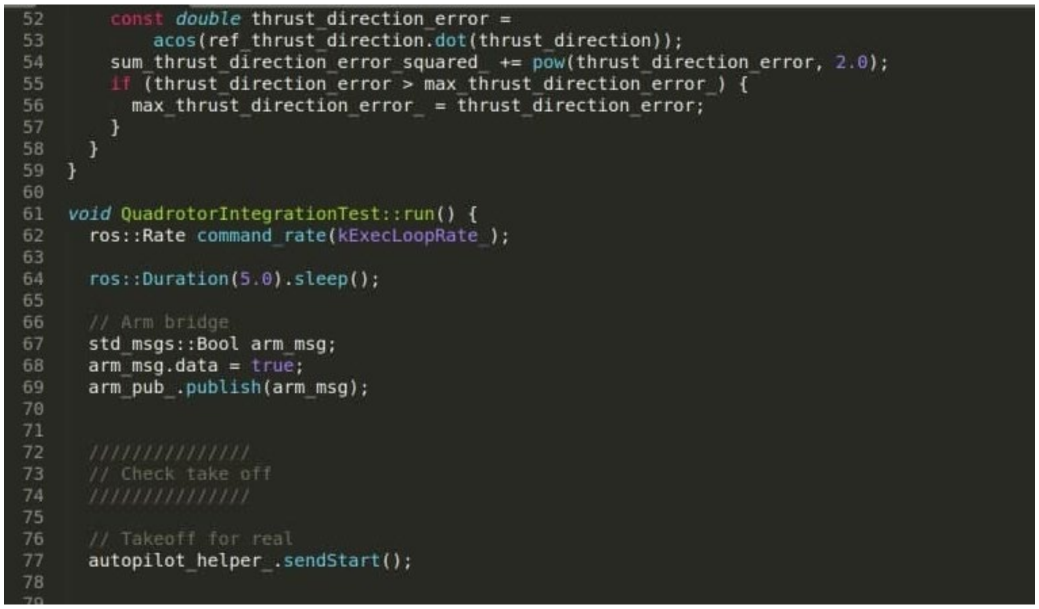
```
namespace rpg_quadrotor_integration_test {

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);
}
```

Implementing the position errors, thrust direction and measure tracking:



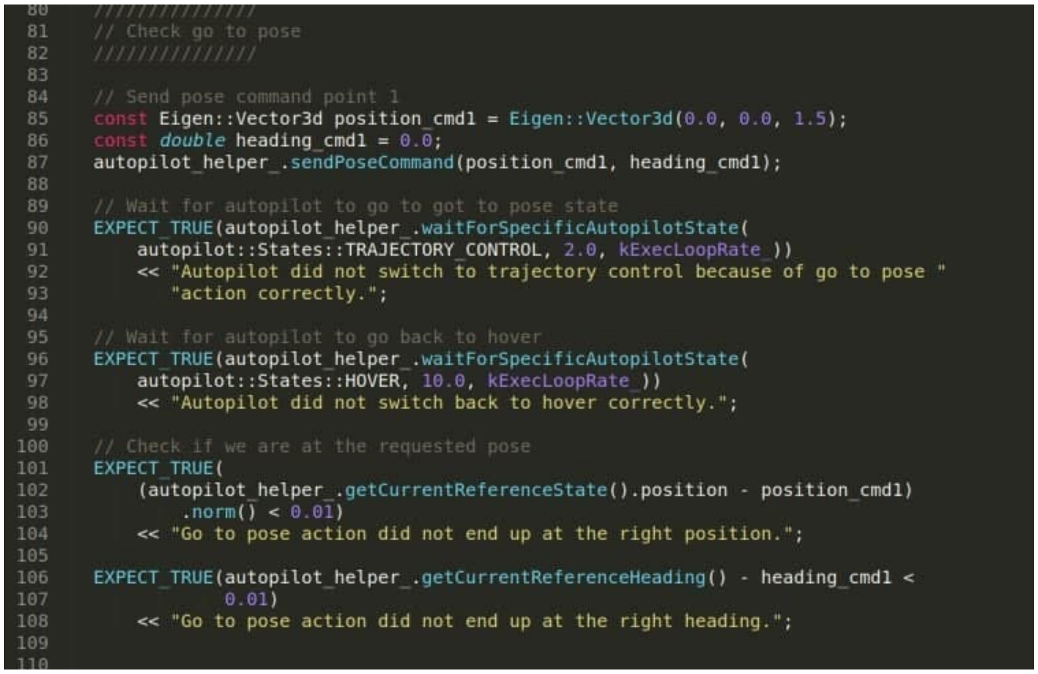
```
QuadrotorIntegrationTest::~QuadrotorIntegrationTest() {}

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;
    }
  }
}
```

Here the arming of the quadrotor and the armbridge of the quadrotor after 5 seconds is initialized:



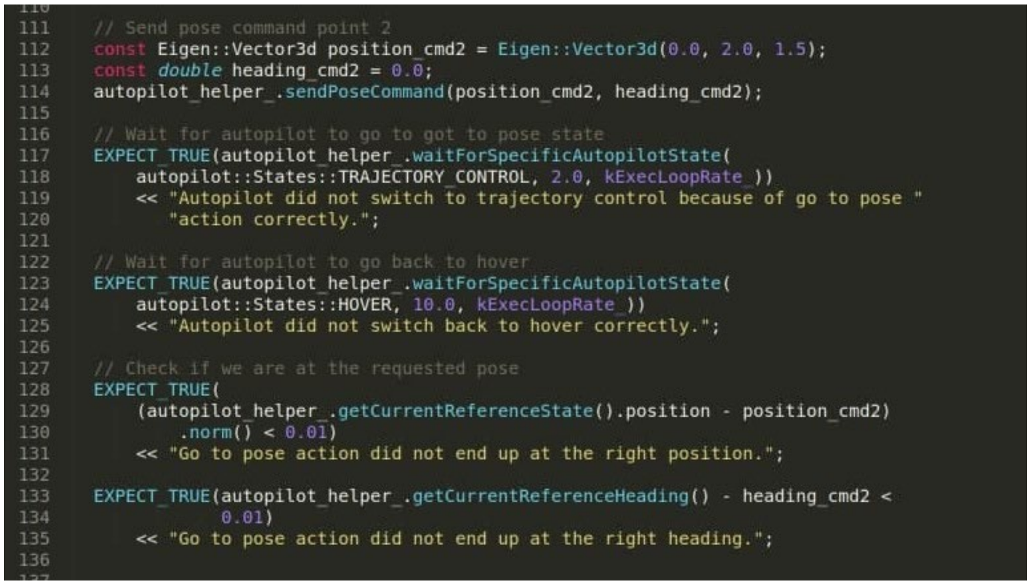
```
    const double thrust_direction_error =
        acos(ref_thrust_direction.dot(thrust_direction));
    sum_thrust_direction_error_squared_ += pow(thrust_direction_error, 2.0);
    if (thrust_direction_error > max_thrust_direction_error_) {
      max_thrust_direction_error_ = thrust_direction_error;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

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();

```

This is where the quadrotor navigates from the first position (0,0,0) to the second position (0,0,1.5):



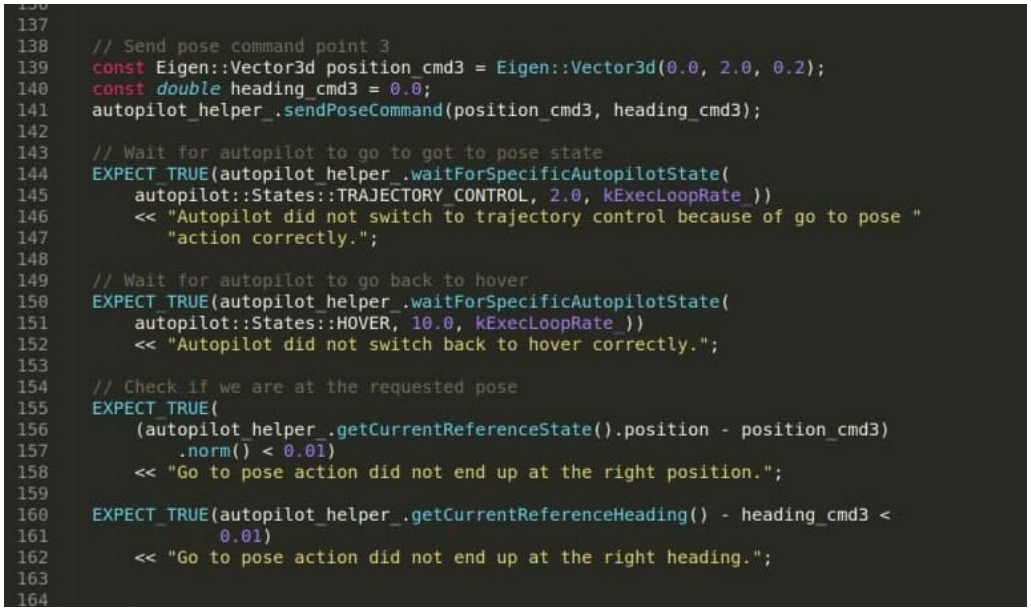
```
///////////////
// Check go to pose
///////////////

// 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.";
```

Here from the second position, to the third position (0,2,1.5):



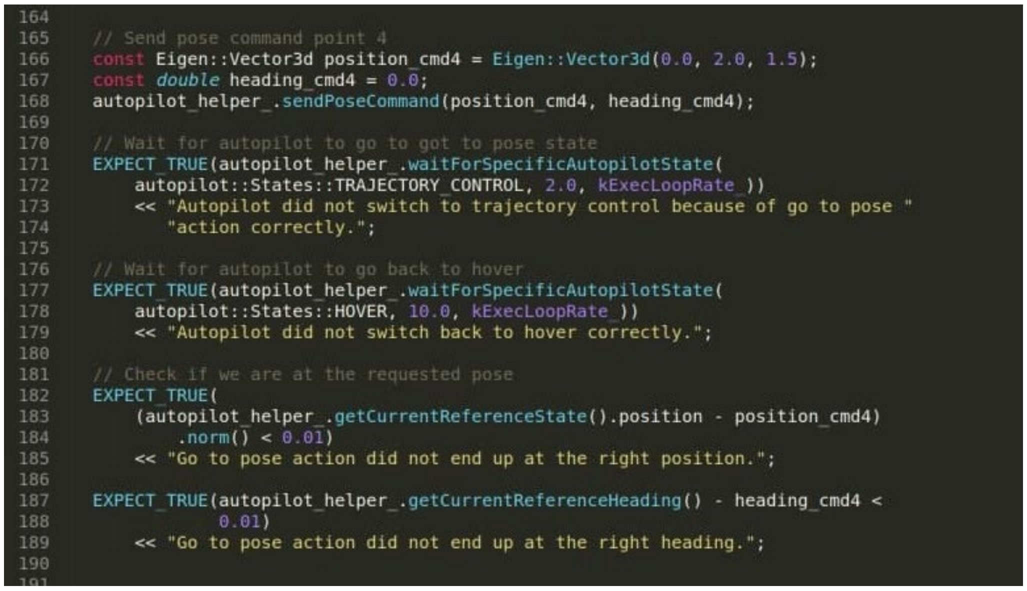
```
// Send pose command point 2
const Eigen::Vector3d position_cmd2 = Eigen::Vector3d(0.0, 2.0, 1.5);
const double heading_cmd2 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd2, heading_cmd2);

// 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.";
```

From the third position to the fourth position (0,2,0.2):



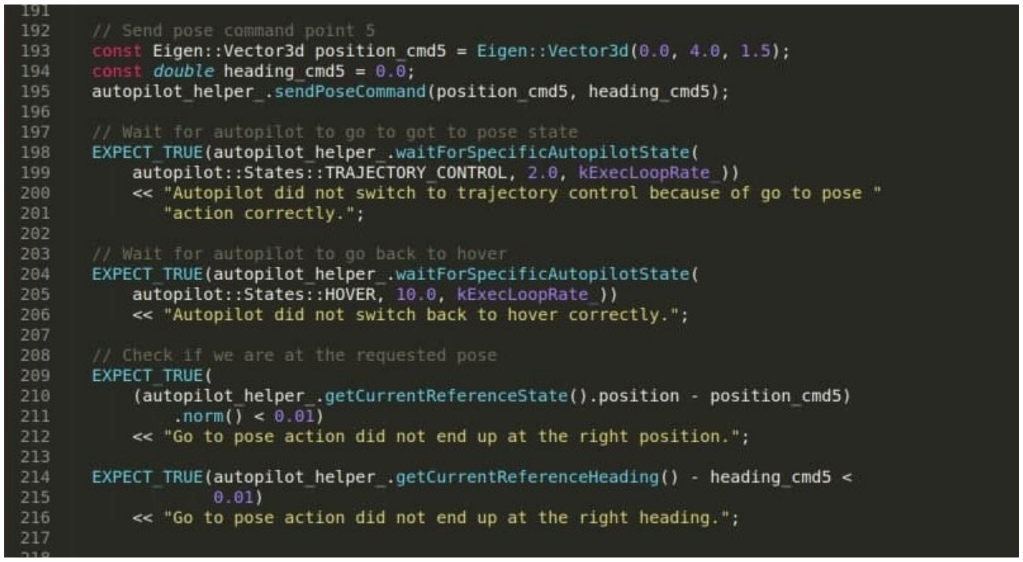
```
  // Send pose command point 3
  const Eigen::Vector3d position_cmd3 = Eigen::Vector3d(0.0, 2.0, 0.2);
  const double heading_cmd3 = 0.0;
  autopilot_helper_.sendPoseCommand(position_cmd3, heading_cmd3);

  // 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.";
```

From the fourth position to the fifth position (0,2,1.5):



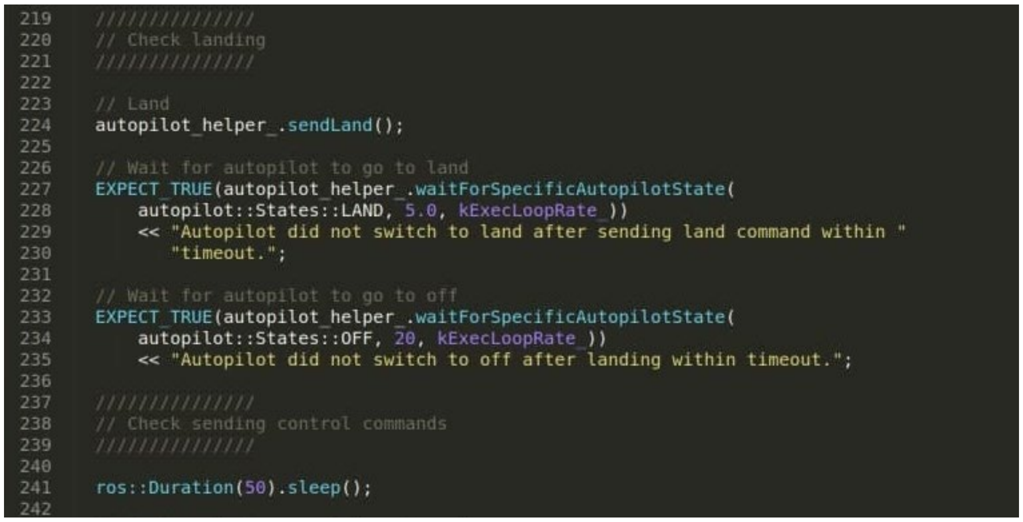
```
  // Send pose command point 4
  const Eigen::Vector3d position_cmd4 = Eigen::Vector3d(0.0, 2.0, 1.5);
  const double heading_cmd4 = 0.0;
  autopilot_helper_.sendPoseCommand(position_cmd4, heading_cmd4);

  // 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.";
```

From position five to the last path position (0,4,1.5):



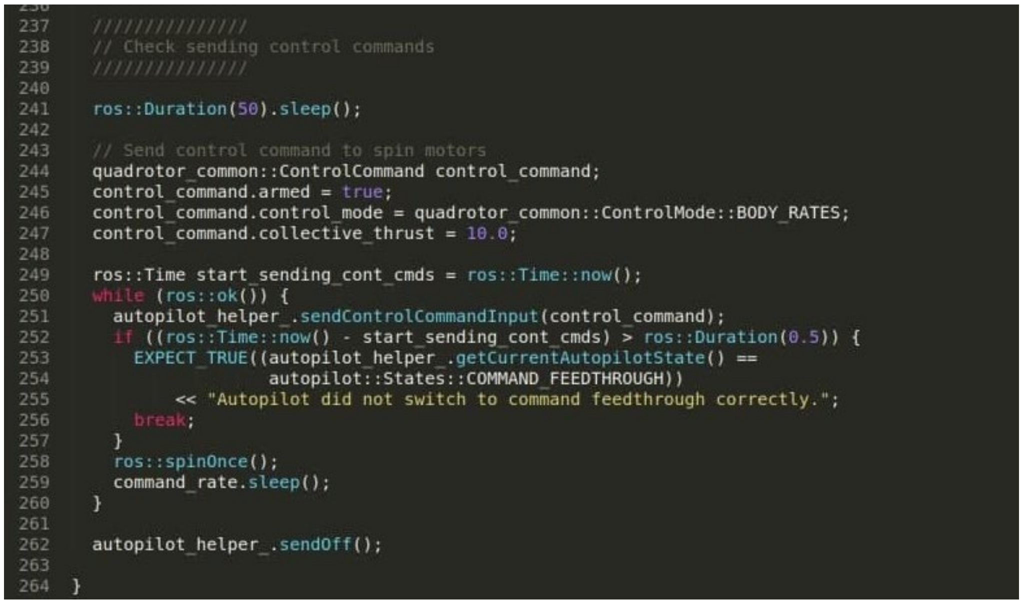
```
// Send pose command point 5
const Eigen::Vector3d position_cmd5 = Eigen::Vector3d(0.0, 4.0, 1.5);
const double heading_cmd5 = 0.0;
autopilot_helper_.sendPoseCommand(position_cmd5, heading_cmd5);

// 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.";
```

After completing the paths above, the quadrotor can land after 20 seconds, before further disarming itself after 50 seconds:



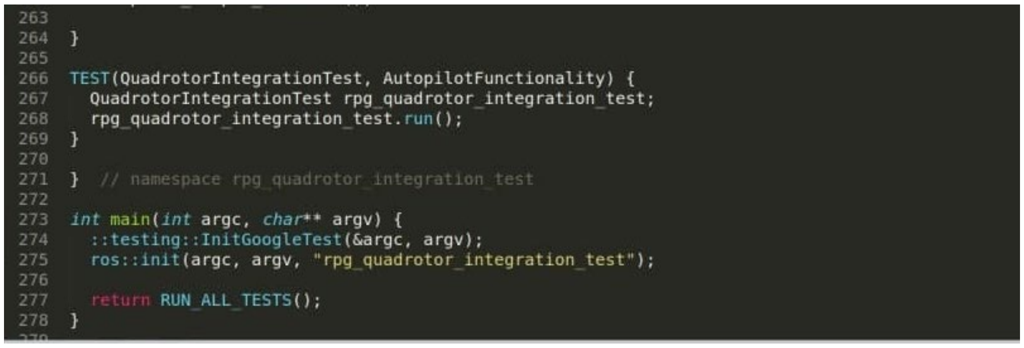
```
///////////////
// Check landing
///////////////

// 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 the control commands that run/control the quadrotors motors:



```
  ///////////////
  // 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();

}
```

And at last the main field that runs the code:



```

}

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();
}
```