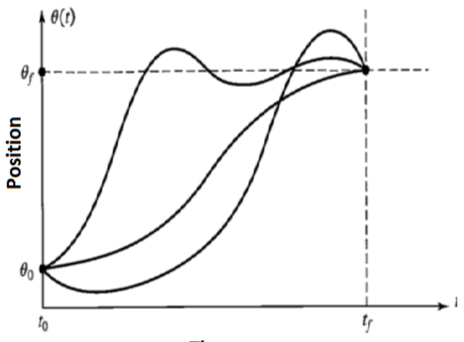
UTA Research Institute
Professor Dr. Frank Lewis

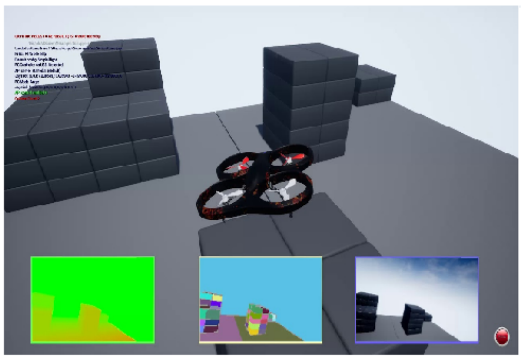
Thesis Substitute Project

# UAV: Trajectory Generation and Simulation

Neha Bhushan



Possible Trajectories



AirSim Simulator

Date of hand in: May, 2019
Supervisor: Dr. Frank Lewis
Patrik Kolaric

# Abstract

Unmanned Aerial Vehicles (UAVs) have gained mass popularity in fields such as Defense, Agriculture, Sport and Photography etc. Their wide adoption in such research projects have opened new horizons for opportunities. However, UAVs operate in a rather dynamic environment since various factors such as airspeed, temperature, pressure, turbulence etc. are capable of hampering the flight operation. There is a need for rather robust trajectory generation algorithm and system such that it provides an easy user interface to specify the path of UAV through space, alongside making the UAV capable enough to maneuver itself from the initial to final position. This project focuses on two aspects of UAV control, the first part focuses on trajectory generation using various mathematically modeled techniques for the position, velocity and acceleration expressed as a function of time. Their comparison is done based on the constraint-based optimization techniques followed by some discussions. The second part focuses on UAV simulators and lays out the foundations and requirements for them, followed by some proposed simulators and their comparison, which could be used not only for flight control simulation, but also have potential to integrate Reinforcement Learning techniques for Autonomous flight control and Autopilot systems. Finally, in-depth analysis of AirSim simulator is carried out along with its environment setup process which may be used as an Interfacing, Installation and set-up documentation for future work.

**Title page image:** First image demonstrates Potential Trajectory functions. Second image is a sample simulated AirSim Blocks environment.

# Acknowledgement

I, Neha Bhushan, Masters Student in Electrical Engineering at the University of Texas, Arlington would like to extend my deepest appreciation and acknowledgement to Dr. Frank Lewis and Patrik Kolaric associated with the University of Texas Arlington and UTA Research Institute. This project would not have been possible to accomplish without their continuous support and feedback. Special appreciation to Vatsal Joshi and Mohit Kalra who provided assistance with work stations and IT infrastructure. Last but not the least, many thanks to my University for all the resources and its Graduate advisors for guiding me throughout my course curriculum and thesis project.

# Contents

**1. Introduction** **1**
1.1. UAV: Overview . . . . 1
1.2. Dynamic Environment . . . . 1
1.3. Trajectory . . . . 1

**2. Motivation** **3**
2.1. Control Strategies . . . . 3
2.2. Application Areas . . . . 3
2.2.1. Research Platform . . . . 3
2.2.2. Military and Law Enforcement . . . . 3
2.2.3. Photography . . . . 4
2.2.4. Drone-Delivery . . . . 4
2.2.5. Industrial Site Inspection . . . . 4
2.2.6. Agriculture and Forests . . . . 4
2.2.7. Archeology . . . . 5
2.2.8. Weather and Storm Analysis . . . . 5
2.2.9. Emergency Response . . . . 5
2.3. Mechanical Structure . . . . 5
2.4. Flight Dynamics . . . . 6

**3. Trajectory Generation** **8**
3.1. Overview . . . . 8
3.2. Proposed Methods . . . . 9
3.2.1. Cubic Polynomials . . . . 9
3.2.2. Linear blend . . . . 10
3.3. Trigonometric Functions . . . . 10
3.3.1. Arc Tangent Function . . . . 10
3.3.2. Sine, Cos Function . . . . 12
3.4. Implementation of Trajectory Generator . . . . 12

**4. Simulator** **14**
4.1. Introduction . . . . 14
4.2. UAV Systems . . . . 14
4.3. Simulator Requirements . . . . 15
4.3.1. Physical Modelling . . . . 15

4.3.2. Environment . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.3.3. Reinforcement Learning . . . . . . . . . . . . . . . . . . 16
4.3.4. Long Term Support . . . . . . . . . . . . . . . . . . . . . 17
4.3.5. Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**5. Choice of Simulators** **19**
5.1. Gazebo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.2. Hector Quadrotor . . . . . . . . . . . . . . . . . . . . . . . . 19
5.3. TUM Simulator . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.4. GymFC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
5.5. AirSim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**6. AirSim** **23**
6.1. Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
6.2. Mathematical Model . . . . . . . . . . . . . . . . . . . . . . . 24
6.2.1. Vehicle Model . . . . . . . . . . . . . . . . . . . . . . . 24
6.2.2. Gravity . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6.2.3. Magnetic Field . . . . . . . . . . . . . . . . . . . . . . . 25
6.2.4. Accelerations . . . . . . . . . . . . . . . . . . . . . . . . 25
6.3. Flight Controller . . . . . . . . . . . . . . . . . . . . . . . . 26
6.3.1. Hardware-in-Loop . . . . . . . . . . . . . . . . . . . . . . 26
6.3.2. Software-in-Loop . . . . . . . . . . . . . . . . . . . . . . 26
6.3.3. simple_flight . . . . . . . . . . . . . . . . . . . . . . . . 26
6.3.4. PX4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**7. Conclusion** **30**

**A. (Appendix) AirSim: Documentation and Installation Instructions** **31**
A.1. Code Structure . . . . . . . . . . . . . . . . . . . . . . . . . . 31
A.1.1. AirLib . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
A.1.2. Unreal/Plugins/AirSim . . . . . . . . . . . . . . . . . . . 31
A.1.3. MavLinkCom . . . . . . . . . . . . . . . . . . . . . . . . . 32
A.1.4. Unreal Framework . . . . . . . . . . . . . . . . . . . . . . 32
A.2. System Requirements . . . . . . . . . . . . . . . . . . . . . . . 32
A.2.1. Hardware Requirements . . . . . . . . . . . . . . . . . . . 32
A.2.2. Software Requirements . . . . . . . . . . . . . . . . . . . 32
A.3. Installation Instructions . . . . . . . . . . . . . . . . . . . . 33
A.3.1. Option 1: Download pre-compiled Binaries . . . . . . . . . 33
A.3.2. Option 2: Build AirSim . . . . . . . . . . . . . . . . . . . 33
A.4. Setup Instructions . . . . . . . . . . . . . . . . . . . . . . . . 33
A.4.1. Unreal Environment . . . . . . . . . . . . . . . . . . . . . 33
A.4.2. AirSim Environment: Blocks . . . . . . . . . . . . . . . . 34
A.5. Programmatic Control . . . . . . . . . . . . . . . . . . . . . . 34
A.6. Application Programming Interfaces (APIs) . . . . . . . . . . . 34

A.7. APIs for Multimotor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
A.8. APIs Explained . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
A.8.1. getMultirotorState . . . . . . . . . . . . . . . . . . . . . . . . . . 36
A.8.2. Async methods, duration and max_wait_seconds . . . . . . . . . . . 36
A.8.3. Drivetrain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
A.8.4. yaw_mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
A.8.5. lookahead and adaptive_lookahead . . . . . . . . . . . . . . . . . . 37

**Bibliography** **43**

# 1. Introduction

## 1.1. UAV: Overview

Unmanned Aerial Vehicles (UAVs) have been gaining momentum in their popularity in wide application fields such as surveillance and monitoring, aerial imaging, cargo delivery, photography etc [23] [9]. Being Aerial, UAVs are independent of terrain unlike Unmanned Ground Vehicles (UGVs) where the application use cases might be hindered by unfavorable terrain contours for which the system might not be designed for. In fact, UAVs can easily fly and reach inspection points, record surveillance data and transmit the information to the base station wirelessly. Being Airborne brings its own set of challenges, limitations. Take aerial traffic for example, just like vehicles running on the ground, UAVs may collide with each other in the air especially for densely populated areas where geographically practical use case of UAV is higher. In residential or commercial places this collision could be disastrous. Furthermore, in a modern city of the present, we have many high-rise buildings and skyscrapers acting as objects of hinderance leaving limited room for maneuverability.

## 1.2. Dynamic Environment

UAVs operate in a rather dynamic environment; another type of challenge is the physical characteristics of Airflow in the environment. Factors such as airspeed, air pressure, temperature, turbulence etc. [17] have the potential to disrupt flight operation and prove fatal for the UAV. Among several open problems related to UAV, trajectory generation is of immense significance which pertains generating a feasible yet optimal or optimally closest flight trajectory in respect to all limitations and constraints such as but not limited to minimum and maximum maneuvering radius, minimum and maximum speed, obstacle and terrain collision avoidance [4].

## 1.3. Trajectory

Trajectory refers to change in position, velocity, and acceleration of UAV with respect to time for the motion of a UAV. In this project, trajectory generation deals with describing the desired motion of UAV in a way that it is easy for user to specify the path of UAV through space rather than calculating its complicated functions in space and time. For the purpose of this project, it is required to generate trajectory for desired initial position, final position and time to reach that position and leave it to the system to decide on the exact shape of the path to get there and calculate the respective the velocity profile.

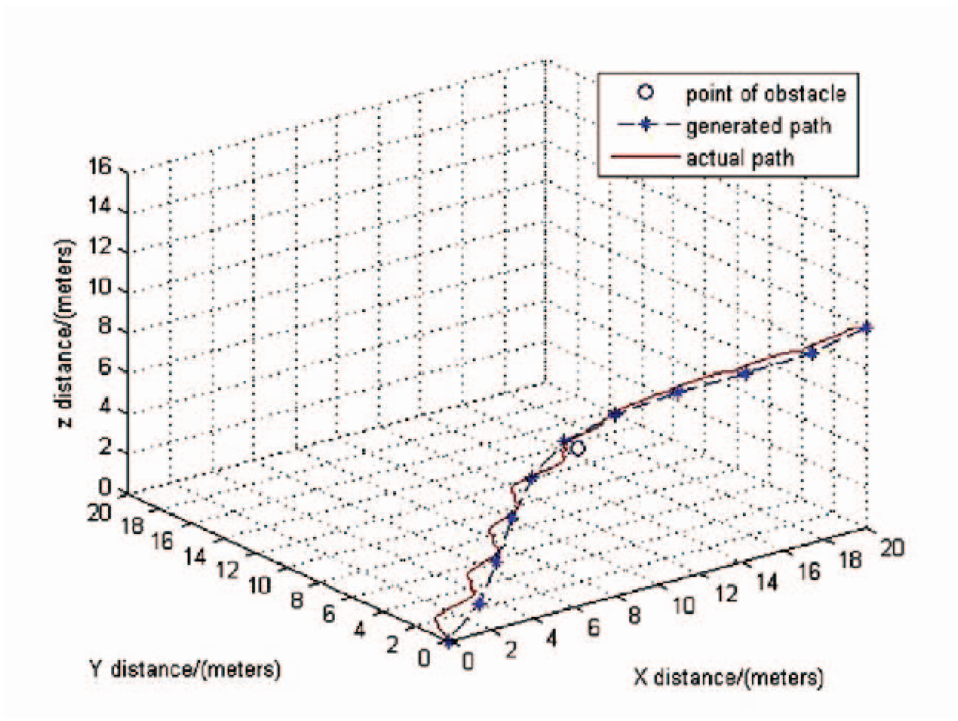


Figure 1.1.: Example of Generated and Actual path

As seen in Fig. 1.1 [7], ], for a typical flight controller, the actual path might differ from the generated path due to presence of obstacles and non-linear characteristics of the controller. In case of UAV flight, it is desirable for the motion to be smooth so that the jerks don't cause the UAV to have erratic behavior like flip in the middle of flight. For a function to be smooth it is desired to be continuous and have a continuous first and second derivatives.

# 2. Motivation

UAVs have become increasingly popular due to their versatile nature and potential to solve real-life challenges in a wide horizon of industry. Environments which are mostly inaccessible to other types of aircrafts such as low-altitude flying in urban areas, carrying out tasks where situation is hazardous for humans prove to be the most generating markets for them. Since UAVs operate without a human on board pilot, there are mainly two control strategies: Remote Control or Autonomous Control. [4]

## 2.1. Control Strategies

In Remote Control, the onboard flight controller receives commands from the base station regarding the trajectory, initial final position, and other flight-critical parameters. At the base station a human pilot flies through the remote controller communicating with Radio Frequency Link. With recent advances in navigation systems, embedded systems, modern microcontroller architectures are getting computationally powerful yet consuming less and less power. This coupled with optimization techniques and digital vision systems have broadened the degree of reliability and autonomy making modern-day UAVs extremely versatile.

## 2.2. Application Areas

Some of the potential use cases of UAVs are as follows.

### 2.2.1. Research Platform

Quadcopters are an extremely useful tool and development platform for University Students and Researchers to validate various control strategies and Algorithms in a real-life scenario to study the flight dynamics and impact of environmental factors such as airspeed, temperature, humidity, turbulence etc. Algorithms relating to flight navigation can also be tested on small inexpensive platforms before deploying them on expensive hardware.

### 2.2.2. Military and Law Enforcement

UAVs are used for surveillance by Military and defense organizations wherever needed for the sake of National Security. These UAVs are generally equipped with Multispectral image sensors, such as RGBD cameras, Infrared based thermal sensing for night-sight vision. Some UAVs depending on application might also be capable of occlusion motions flow sensors and motion detectors for critical areas like border crossings and Secured test sites.

### 2.2.3. Photography

Initially used by professional cinematographers, drones equipped with cameras are gaining mass popularity among millennial travelers for photography and videography. Massive progress has been made towards portable and foldable quadcopters design with advanced technology such as Optical Image Stabilization and Object following technology making its way into commercial photography drones.

### 2.2.4. Drone-Delivery

Many companies such as Deutsche Post and Amazon prime have been experimenting to attempt home delivery of parcels and couriers using drones. These solutions may ultimately prove to be revolutionary for the shipping and logistics industry as they solve the problem of last mile delivery connectivity situation, thereby enabling faster deliveries in downtown metropolitan areas and among high-rise residential. Moreover some food startups are trying to utilize drone potential to make deliveries for home-orders which increases the quality of service since it drastically reduces the time-to-home.

### 2.2.5. Industrial Site Inspection

This is one particular area that has highly impactful real-life use case for the UAVs. Very often in or nearby industrial sites, such as chemical plants, Thermal Power plants as well as Nuclear Plants, site inspection is mandated to ensure safe functionality of these plants. Due to their sheer size, physically it is not possible for humans to inspect all the aspects of them. Drones are gaining mass popularity in these use cases. When equipped with multispectral camera sensors and gas sensors, these can be used for monitoring health of a boiler, reactor or furnace exhaust. With onboard integrated sensors, they collect valuable data which might be helpful in detecting or preventing industrial leakage which might be catastrophic.

### 2.2.6. Agriculture and Forests

There are quite a number of civil applications of UAVs. Recently, many governmental organizations such as the forest department and municipalities have been using drones for Aerial 3D mapping of a locality along with Aerial surveys for crops, inspection of power lines / pipelines, counting and analyzing wildfires, detection of illegal hunting, providing medical support in areas which are otherwise not accessible etc.

### 2.2.7. Archeology

In Latin America, drones have been increasingly replacing the use of small planes, kites and helium balloons as the latter have proven to be expensive and difficult to operate as compared to UAVs. Smaller UAVs have been capable to producing 3D models of Peruvian sites instead of flat-maps in a relatively shorter period of time.

### 2.2.8. Weather and Storm Analysis

UAVs are especially useful in accessing areas that are hazardous for manned vehicles. Situations such as hurricanes, tornadoes and cyclones are some of the typical example where usually manned aircrafts are warned to stay away from. In such cases inexpensive UAVs can be used for in-depth analysis of such calamities to gather crucial data, which helps in making plans for emergency response and evacuations if need be. Such systems are equipped with necessary barometric and temperature sensors which transmit the data wirelessly.

### 2.2.9. Emergency Response

Even in post-calamity conditions, drones have proven to be extremely effective in search and rescue operations alongside property damage assessment. Typical payloads include optical sensors, radars to capture imagery and analyze terrain situation. Similarly, when equipped with gas sensors, they can be used to analyze the pollutions levels and severity especially for cities like New Delhi and Beijing with critical pollution levels.

## 2.3. Mechanical Structure

For a typical quadcopter construction, there are these following components.

1. Frame
2. Propellers (fixed pitch or variable pitch)
3. Electric Motors
4. Battery
5. Flight Controller and Electronics

Generally, to obtain optimum performance and simple control system, the motors and propellers are placed equidistant from each other. To improve the power to weight ratio, exotic materials such are carbon fibre are used in the chassis and mechanical structure construction as these offer exceptionally well, strength to weight ratio, thus enabling UAVs to drastically be lighter. This not only improves the maneuverability but also enhances flight time as battery could be used more efficiently.

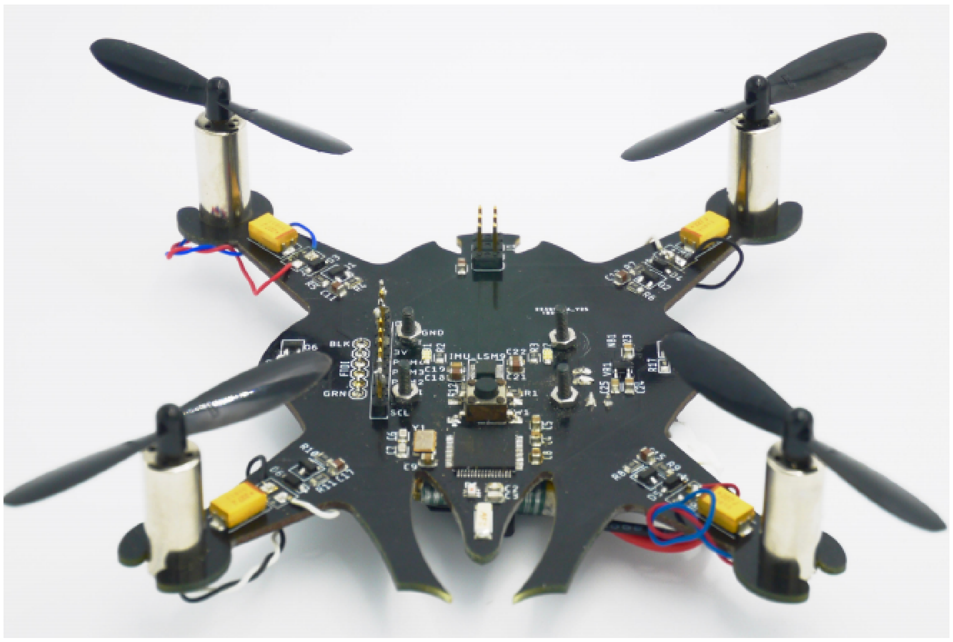


Figure 2.1.: Quadcopter Design [21]

## 2.4. Flight Dynamics

Flight dynamics of a UAV are quite complex and explode exponentially as the number of rotors increase such as for a hexacopter and an octocopter. To break the dynamics down in simpler forms, it is easier to model and understand for a four-rotor design.

A quadcopter has two pairs of rotors in cross-configuration which are capable of rotating at different angular velocities (rads/sec) This differential velocities enable the quadcopter to achieve rotational and translation motion. Detailed dynamics can be found in the words of *Ali et. al [3]*. The following equations briefly summarize the dynamics. the total thrust u is described as [6].

$$u = f_1 + f_2 + f_3 + f_4$$

where $f_1$ $f_2$ $f_3$ $f_4$ denote the upwards thrust generated by each rotor.

$$f_i = k_i \omega_i^2, i = 1, ..., 4$$

where $k_i > 0$ is a constant and $\omega$ is the angular speed of the respective motor, m is the mass of the quadcopter and g is the gravitational acceleration.

$$m\ddot{x} = -u \sin \theta$$

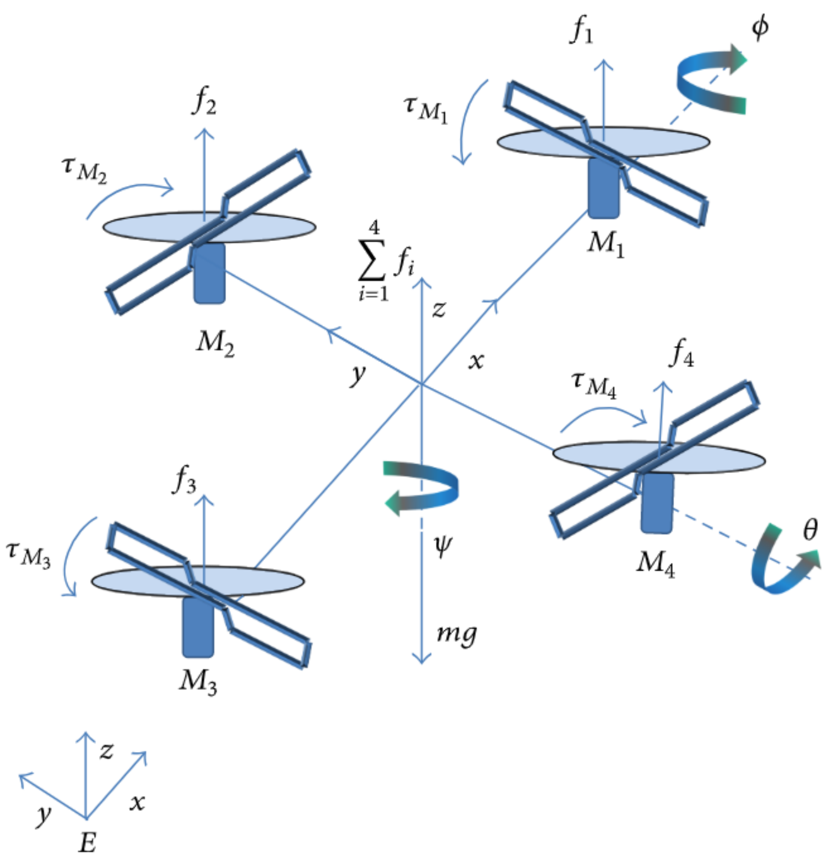


Figure 2.2.: Quadcopter Dynamics [19]

$$m\ddot{y} = u\cos\theta\sin\phi$$

$$m\ddot{z} = u\cos\theta\cos\phi - mg$$

$$\ddot{\psi} = \tau_\psi$$

$$\ddot{\theta} = \tau_\theta$$

$$\ddot{\phi} = \tau_\phi$$

$\tau_\psi \tau_\theta \tau_\phi$ denote the control inputs for yaw, pitch and roll respectively.

# 3. Trajectory Generation

## 3.1. Overview

Our functional requirements can be formulated as follows. Since there are n number of possible permutations and combination for a UAV to travel from initial to the final positions. We apply constraints of limited time window along-side having jerk-free motion.

Let $p(t)$ denote the position as a function of time. Let $p(0)$,$p(f)$ denote the initial and final positions of the UAV respectively. Let the timing constraints for the UAV to reach from initial to final position be given by $t_0$ and $t_f$ respectively. Figure 3.1 shows the possible smooth function trajectories which might be feasible. Of these, the one with most optimal trajectory should be used.

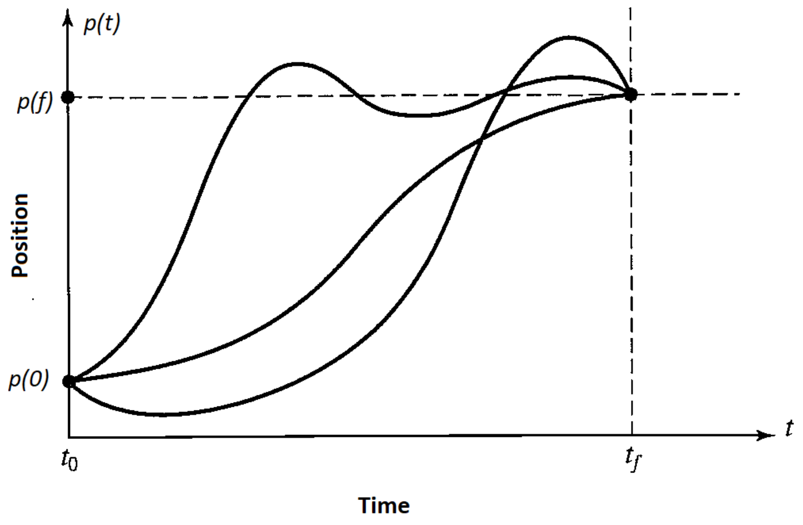


Figure 3.1.: Example of Possible smooth trajectories

In making a smooth motion, at least four constraints on $p(t)$ are evident. To derive simpler equations, the characteristics are considered to be one dimensional rather than the actual 3-dimensional model for the physical system. Two constraints on the function's value come from the selection of initial and final values:

$$tf = T$$

$$p(0) = P_0$$

$$p(tf) = P_f$$

Additional two constraints are that the velocity function be continuous as the UAV starts and stops at the final position, which in this case means that the initial and final velocity on UAV are zero:

$$\dot{p}(0) = 0$$

$$\dot{p}(tf) = 0$$

## 3.2. Proposed Methods

Trajectory generation problems in robotics are generally solved using Cubic Polynomials and Linear function with parabolic blends. The solutions explored are briefly described below:

### 3.2.1. Cubic Polynomials

$$p(t) = x_0 + x_1 t + x_2 t^2 + x_3 t^3$$
$$p(0) = x_0 = P_0$$

$$p(t) = P_0 + x_1 t + x_2 t^2 + x_3 t^3$$

$$v(0) = \dot{p}(0) = x_1 = 0$$

$$v(t) = \dot{p}(t) = x_1 + 2x_2 t + 3x_3 t^2$$

$$a(0) = \dot{v}(0) = 2x_2 = 0$$

$$v(t) = \dot{p}(t) = 2x_2 + 6x_3 t$$

Using this method accelerations can't be zero as it will make all the coefficients zero and for smooth trajectory, these accelerations are desired to be minimized which in turn affects its efficiency in term of time to reach goal position. To have the desired trajectory, we will need higher order polynomials which makes the equation more complex. To follow the constraints, we will have to solve the trajectory problem for third order derivative i.e. jerk and fourth order derivative i.e. snap.

### 3.2.2. Linear blend

In case of linear blend, the motion from the initial to the final position is simply linearly interpolated as seen in Fig. 3.2. However, if such a trajectory is desired, it would result in velocity to be discontinuous as the system would demand abrupt changes in infinitesimally small time from 0 to $v_{final}$ in the starting phase and $v_{final}$ to 0 in the final phase thereby resulting in infinite acceleration in both.To create a smooth path with continuous position and velocity we add parabolic terminators at both these phases as shown in Fig. 3.2. As the acceleration increases, the length of the blend region becomes shorter and shorter. In the limit, with infinite acceleration, we are back to the simple linear-interpolation case.

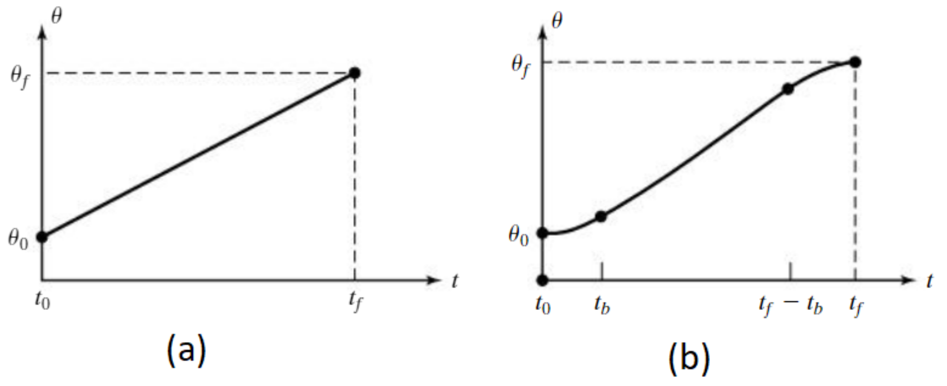


Figure 3.2.: Linear Interpolation with (a) Infinite acceleration (b) Parabolic Blends

Fig. 3.3 (a) shows a case where the value of acceleration is chosen to be quite high. In such cases, we quickly move from initial to the final position, this results in nearly constant velocity with noticeable changes only during the beginning and the end phase. For (b) we see that acceleration is lower as compared to the previous case hence the linear section disappears and the curve appears parabolic.

## 3.3. Trigonometric Functions

### 3.3.1. Arc Tangent Function

Due to uneven acceleration values for linear blend, cubic polynomial seems like a better choice. Mathematical functions like exponential and trigonometric functions can be used

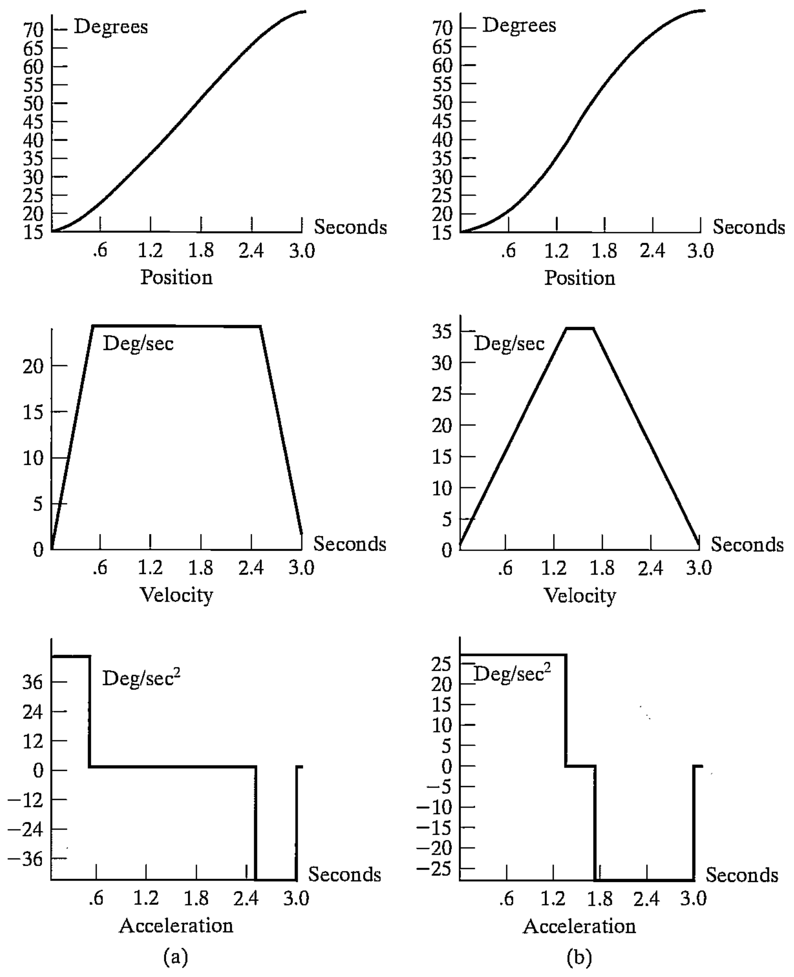


Figure 3.3.: Example curves of position, velocity, acceleration in Linear Interpolation (a) Higher Accleration (b) Lower Acceleration

similarly for trajectory generation as they produce similar results when differentiated. Transformation of these functions and curve fitting methods can be used to produce results for problem at hand. Trigonometric functions like arctan and cos are best suited for the desired position curve. I went ahead with testing the arctan function first. Considering the variables, A, B, C and D for transformation of functions we can start with the equation below.

$$p = C.\arctan(At - B) + D$$

For the given curve of final position can be defined as:

$$p = 0.9 \lim_{t \to \infty} [C.\arctan(At - B) + D]$$

$$p_f = 0.9[C.\frac{\pi}{2} + D]$$

$$P_0 = -C.\arctan(B) + D = 0$$

Considering max velocity $A = \frac{V_{max}}{C}$, Solving these equations did not come up with a very smooth curve as the variables in this equation are not independent.

### 3.3.2. Sine, Cos Function

The other solution is transforming cos functions and limiting them generated more desirable curve. The equation used for position, velocity and acceleration as follows

$$p(t) = -\frac{P_f}{2}\cos(\frac{\pi}{T}.t) + \frac{P_f}{2}$$

$$v(t) = \frac{p_f}{2}.\frac{\pi}{T}\sin(\frac{\pi}{T}.t)$$

$$a(t) = \frac{P_f}{2}.\frac{\pi}{T}.\frac{\pi}{T}\cos(\frac{\pi}{T}.t)$$

The curves generated for these equations are well suited for smooth trajectories and are hence preferred.

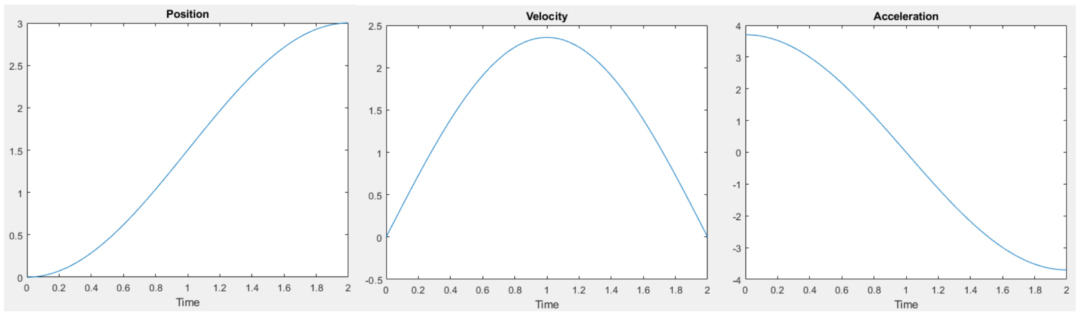


Figure 3.4.: Position, Velocity, Acceleration curves

## 3.4. Implementation of Trajectory Generator

Trajectory Generator only generates a path for the UAV from an initial position to the final position. However during a UAV flight and motion, other aspects such as Hovering, Dynamical Rotor control and actual flight are not directly controlled by the Trajectory Generated.

When actually using the trajectory generator for desired motion of UAV, the trajectory generator must be triggered at the time of defining path or guiding the UAV from starting position to desired final position and after reaching the final position, a threshold of a small value (eg. 10 cm) must be considered to kill the trajectory generator and override by the hover function to make the UAV stop and hover at the final position.

# 4. Simulator

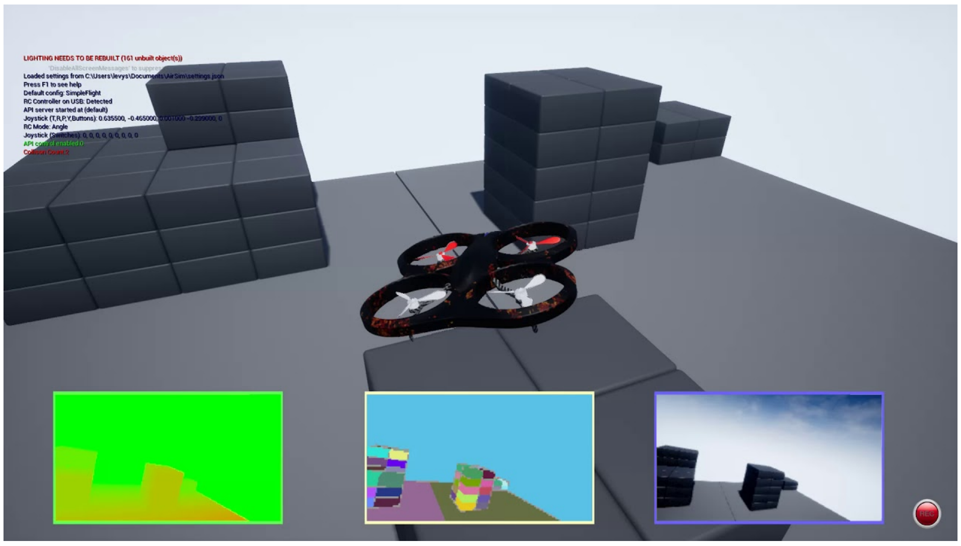
## 4.1. Introduction

Simulation has always been an integral tool in robotics and is a reliable source to generate useful data by saving time and cost. Testing of any robotic hardware or software can be done by simulation by creating number of scenarios and the reaction of robot in these scenarios for example a hundred of seconds of real-world scenario can be enacted in one second of a simulation. Moreover, it lets user to study and perform research in complex missions that may be time-consuming and risky in real-world. Virtually, the mistakes or bugs in simulation cost nothing as the vehicle can crash multiple times and hence the implementation of a concept can be debugged by learning through the simulation of the concept in various conditions. In case of UAVs, developing and testing algorithms in real world can be very expensive and time-consuming. Moreover, utilizing current research and advancement in machine intelligence and deep learning requires collection of huge amounts of training and testing of in a variety of conditions and environments.

## 4.2. UAV Systems

UAV system simulation is desired to train UAV to control and navigate by its own thereby having fully autonomous capabilities. The flight simulator is desired to artificially recreate the UAV flight dynamics and environment in which it flies by simulating aerodynamic model. This is done by replicating the equations that govern the control of UAV, their reaction to external factors such as air density, turbulence, wind shear, cloud, precipitation, etc. The best simulators generally offer a variety of courses, variable weather conditions and realistic physics to keep it as realistic and practically applicable as possible.

However, simulating the real-world is a big challenge. To operate a UAV in the real-world without any hurdles it is important to be able to transfer the learning it does in simulation as the simulation and real-world varies a lot. One of the most difficult components in this aspect is simulating the detailed physics of the environment [22]. For a UAV simulation system, physics includes perception, sensing and the dynamics of the system, ground and atmosphere. Therefore, it is important to develop accurate models of the UAV system dynamics so that simulated behavior enacts the real-world as accurately as possible.



```
LIGHTING NEEDS TO BE REBUILT (161 unbuilt object(s))
        'DisableAllScreenMessages' to suppres
Loaded settings from C:\Users\levys\Documents\AirSim\settings.json
Press F1 to see help
Default config: SimpleFlight
RC Controller on USB: Detected
API server started at (default)
Joystick (T,R,P,Y,Buttons): 0.635500, -0.465000, 0.001000 -0.299000, 0
RC Mode: Angle
Joystick (Switches): 0, 0, 0, 0, 0, 0, 0, 0, 0
API control enabled:0
Collison Count:2
```

Figure 4.1.: Quadcopter Simulation Environment

## 4.3. Simulator Requirements

For a typical UAV simulator we have the following requirements.

1. Good physical model that include aerodynamical effects like wind and gravity.
2. Realistic Environment for better use at time of implementing in real world.
3. Reinforcement Learning integration.
4. Long term support and easy to use for other users.
5. Good interface for system integration and hardware in loop testing.

### 4.3.1. Physical Modelling

For a typical UAV simulator, the quality of the environment and simulating engine depends on how well detailed and complex is the physical model. The environment is more real-life like if it supports the effects of wind gusts, precise gravitational forces, air temperature and pressure. Since these parameters are beyond the control of the system, modelling them enhances the robustness of algorithm and system design. Some even more complex parameters such as aerodynamical effects play a critical role in higher altitude and high-speed maneuvers during a flight. Having such properties is highly desirable in a simulator.

### 4.3.2. Environment

For vision-based systems and simulators supporting trajectory generation and path planning. Detailed modelling of the environment and setting becomes crucial, since the algorithm of the system being tested in the simulated environment might be custom tailored for some custom and specific scenarios. Typical examples of Environment includes, Hill Side, City Life, Parks, Indoors. Depending on the use case and application area, some simulators might have an advantage over the rest. It is highly desirable to have a simulator that supports multiple environments and has extensions or add-ons, where newly modelled environments could be easily integrated.

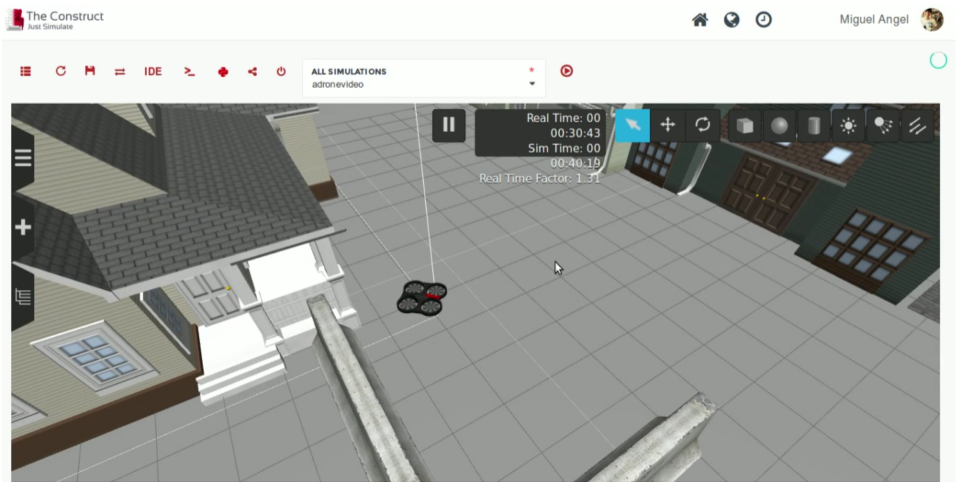


Figure 4.2.: Indoor Simulator Environment [10]

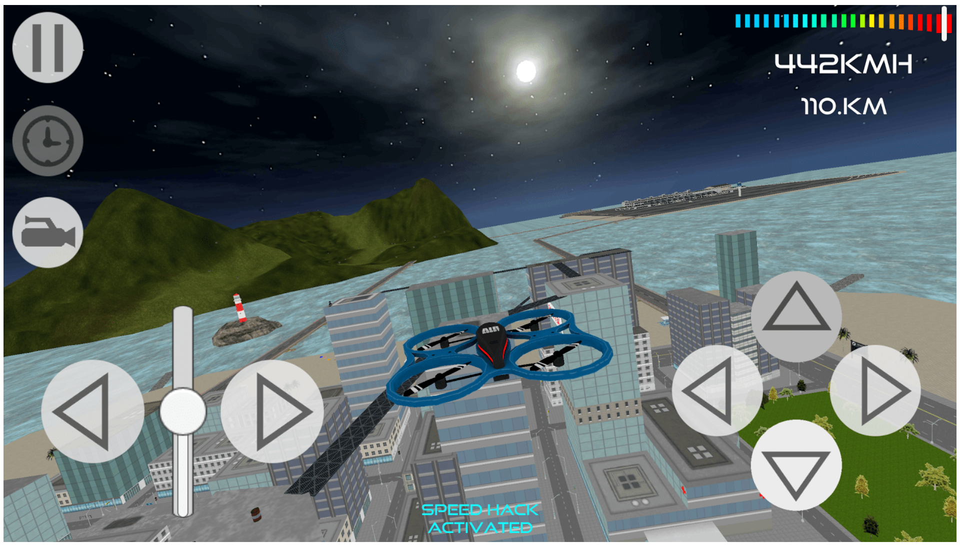


Figure 4.3.: City Simulator Environment [12]

### 4.3.3. Reinforcement Learning

With the advancement of Machine Learning based algorithms and techniques, simulators have gained massive popularity. Certain unsupervised learning techniques such as K-means clustering and Q-Learning based DQN networks have a massive advantage over supervised learning techniques in a way that they do not require labelled dataset, i.e. they do not require for each input label x to have an output label y. This has massive implications in terms of savings in time and resources as manually labelling dataset is quite expensive and time-consuming. Rather by utilizing the techniques of reward and penalty, a cost function could be formulated where, the network can be trained on massive unlabelled data, which learns by optimizing the cost function by minimizing the penalty and maximizing the reward.

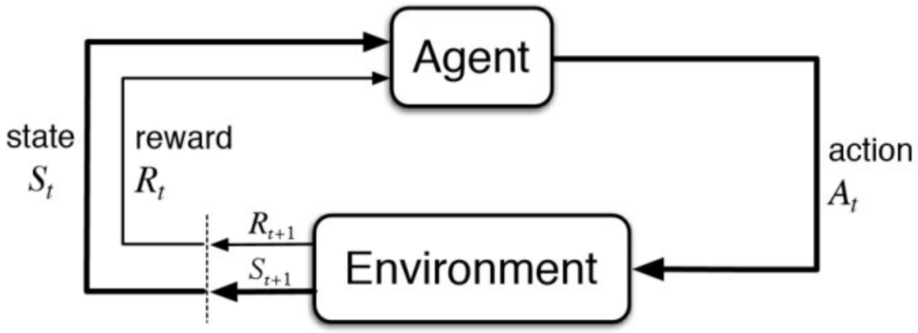


Figure 4.4.: Reinforcement Learning Block Diagram

Such learning frameworks are usually based on Python or R, hence if the simulator is capable to be interfaced by using certain Application Programming Interfaces (APIs) along with TensorFlow, Caffee, is an added advantage. When configured correctly, the system could be initialized by integrating with the simulator and ideally train on infinite amount of simulated data without the need for manual labelling, thereby truly creating an Artificially Intelligent system.

### 4.3.4. Long Term Support

Long Term Support or legacy support is critical from a life-cycle management point of view. Traditional development of new projects tend to take a few years depending on the complexity of the underlying system. Meanwhile, if the simulator is discontinued or is not constantly being updated according to the state-of-the-art standards results in a buggy development environments which is not at all desirable. Long-term support also ensures wider adoptability of a simulator since developers are willing to pay a premium over support, which leads to even wider adoption and hence creates a positive improvement life-cycle.

### 4.3.5. Interface

System integration, APIs, IOs and Hardware-in-loop testing is another crucial characteristic for a simulator. Many a times, after validating the system in a simulated environment, it is important to verify it in actual hardware setting before a test flight. However, due to high degree of uncertainty, there is a high degree of risk associated with actual flight and using expensive hardware for such tests. Hardware-in-loop is an excellent verification methodology which ensures that the algorithm performs at par in performance as compared when tested in a simulated environment. This enables to check the hardware functionality running the actual algorithm yet in a simulated environment, such that even if certain things go hay-wire, no damage is done to expensive hardware. Similar extensive hardware interfacing and integration is desirable since devices supporting various communication protocols could be interfaced.

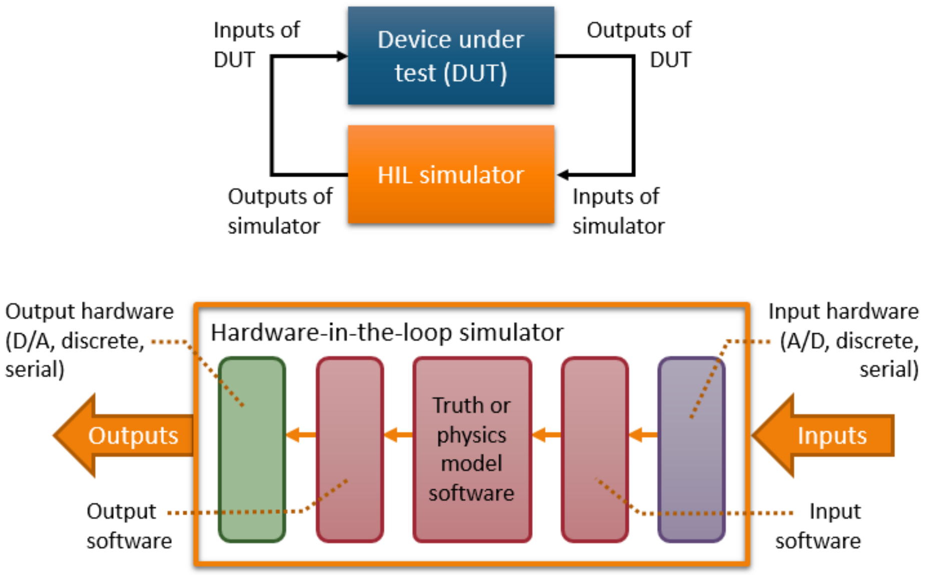


Figure 4.5.: Hardware-in-loop Testing [24]

# 5. Choice of Simulators

There are a bunch of flight simulators out there continuously trying to improve the controllers and make it as realistic as possible. With increased computation power and easily accessible better hardware for computing makes it easier to run learning algorithms as it helps creating realistic test data at lower cost of resources like time and money.

## 5.1. Gazebo

Gazebo is the most popular simulator used in robotics research mainly because of its long-term and is well suited for testing computer vision and object avoidance algorithms. Its access to the control interface of UAV in the simulation enables users to experiment with control dynamics of the UAV providing a good base for further research and analysis. Gazebo provides simulation environment which can integrate different features like support for ROS, SITL (Software in the loop) and HITL (Hardware in the loop) testing and support for various UAVs like Quad (Iris and Solo, Hex (Typhoon H480), Generic quad delta VTOL, Tailsitter, Plane. Gazebo simulation environment lets simulators be customized according to the goal of its mission. Fig. 4.2 is an example indoor environment instance running for a quadcopter.

## 5.2. Hector Quadrotor

Hector Quadrotor is a ROS package based on packages related to modeling, control and simulation of quadrotor UAV systems. Maintained by Johannes Meyer and written by Johannes Meyer, Stefan Kohlbrecher, it offers wind tunnel tuned flight dynamics, sensor models that includes bias drift using Gaussian Markov process and software-in-loop using Orocos Toolchain. Hector lacks support for popular hardware platforms such as Pixhawk and protocols such as MavLink. [13] Hector Quadrotor consists of the following packages:

- hector_quadrotor_description provides a generic quadrotor URDF model as well as variants with various sensors.
- hector_quadrotor_gazebo contains the necessary launch files and dependency information for simulation of the quadrotor model in gazebo.
- hector_quadrotor_teleop contains a node that permits control of the quadrotor using a gamepad.

- hector_quadrotor_gazebo_plugins provides plugins that are specific to the simulation of quadrotor UAVs in gazebo simulation.

## 5.3. TUM Simulator

This package, written by Hongrong Huang and Juergen Sturm is based on hector quadrotor made particularly for implementation of a gazebo simulator for the Ardrone 2.0. The simulator can simulate both the AR.Drone 1.0 and 2.0, the default parameters however are optimized for the AR.Drone 2.0 by now. The AR.Drone2.0 connects with a computer via WIFI, while the user manipulate a joystick which is via USB connecting with the same computer. ROS is running in this computer. [2]

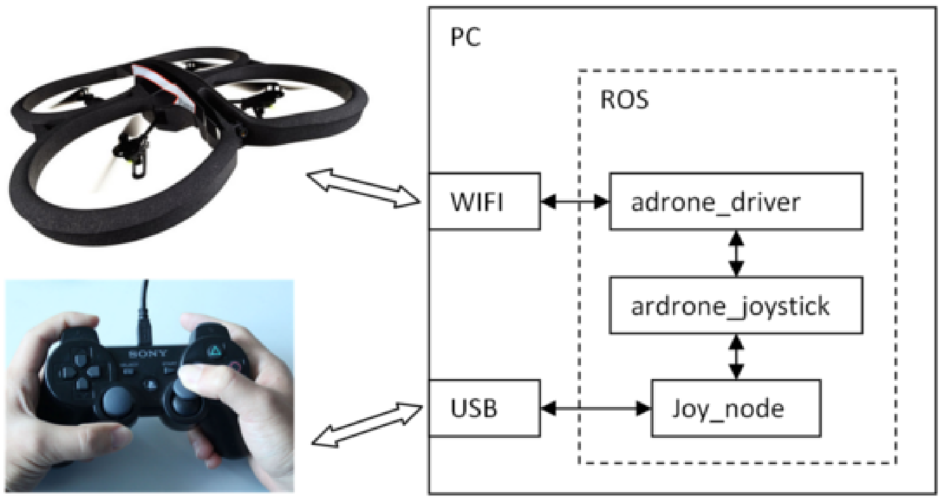


Figure 5.1.: TUM Simulator [2]

## 5.4. GymFC

GymFC [8] is an OpenAI Gym environment specifically designed for developing intelligent flight control systems using reinforcement learning. This environment was customized to progress research of the intelligent flight controller by providing with a tool to access and modify the PID controller. GymFC's primary goal is to train controllers capable of flight in the real world. In order to construct optimal flight controllers, the aircraft used in simulation should closely match the real world aircraft. Therefore the GymFC environment is decoupled from the simulated aircraft. The documentation of GymFC includes an example to verify the environment and get used to it. GymFC is recommended to be run using gzserver in a headless mode but it is generally desired to visually see the aircraft during development and testing for verifying results.

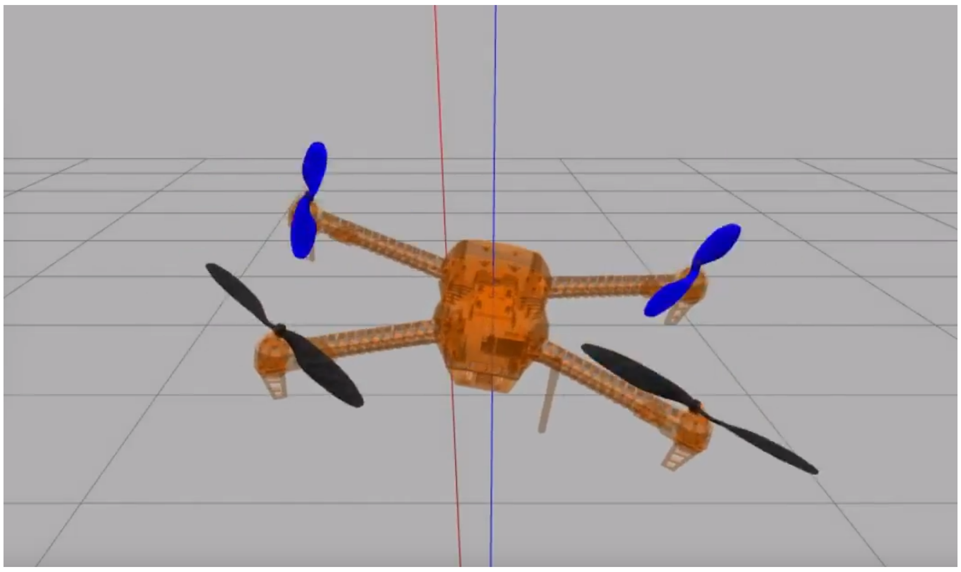


Figure 5.2.: GymFC Environment [8]

## 5.5. AirSim

AirSim [20] is a new open source simulator in market that is still under development. Starting in 2017, this simulator uses Unreal Engine for physically and visually realistic environment and supports cross-platform functionality that makes it easy to interface provides the freedom of using libraries with support of learning algorithms and providing fast computing for logging and testing data. It exposes APIs to interact with the vehicle in the simulation programmatically to retrieve images, get state, control the vehicle etc. It is the best choice of simulator for current research because of its cross-platform functionality to support learning algorithms in Python, support for Windows as well as Linux OS platform, integration with ROS and advanced rendering and detailed environments using Unreal Engine.

It includes a physics engine that can operate at a high frequency for real-time hardware-in-the-loop (HITL) simulations with support for popular robotic communication protocols like MavLink. The simulator is designed to be extensible to accommodate new types of vehicles, hardware platforms and software protocols. In addition, the modular design enables various components to be easily usable independently in other projects. Its development as an Unreal plugin enables it to be dropped into any Unreal environment and support flight controllers such as PX4 for physically and visually realistic simulations.

The most important aspect of Airsim for the project is its integration with learning algorithms and its cross-platform accessibility. Airsim is designed to be used for experimenting with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles, which can be done through the exposed APIs to retrieve data and control vehicles in a platform independent way. Another major advantage of using AirSim is that it was initially developed by Microsoft and then open-sourced to the community. As a result, it has wide adoption across industry and developers thereby ensuring that long-term support would be easily available. At

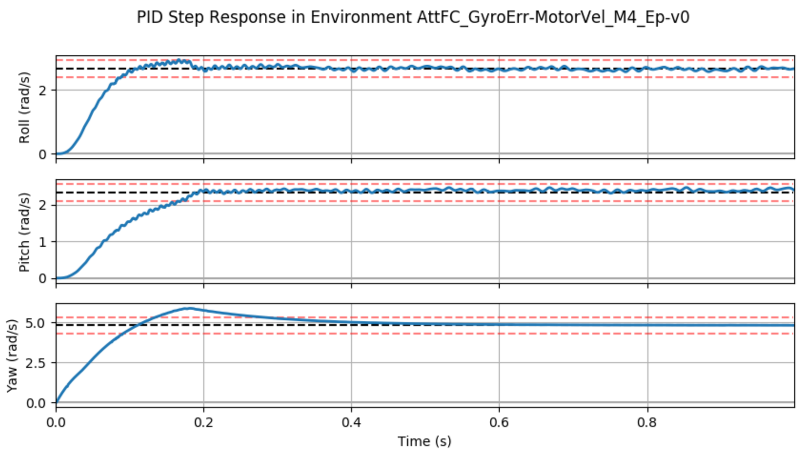


Figure 5.3.: GymFC PID controller response [8]

GitHub [14] , it is available in Free-Open Source License, such that the source code and respective releases are regularly released. At present there are about 100+ contributors to the project with 1700+ commits to the project, making it pretty actively updated. Moreover, the support developers are constantly working to resolve any bug fixes.

# 6. AirSim

As discussed in the previous chapter, the best choice for simulator for experimenting with reinforcement learning in drones is AirSim because of its cross-platform functionality and long term support. AirSim can be used as a simulator for drones, cars and more, built on Unreal Engine, it provides physically and visually realistic simulations. Its APIs retrieve data and control vehicles in a platform independent way and prove to be very useful in experimenting with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles.

## 6.1. Architecture

Fig. 6.1 shows the block diagram of AirSim. A very important aspect of Airsim is its modular design and its ability to be expanded and changed as needed. Its classification of modules for specific function makes it easy to understand and implement. Higher level overview of the simulator consists of environment model, vehicle model, physics engine, sensor models, rendering interface via Unreal Engine, public API layer and an interface layer for vehicle firmware.

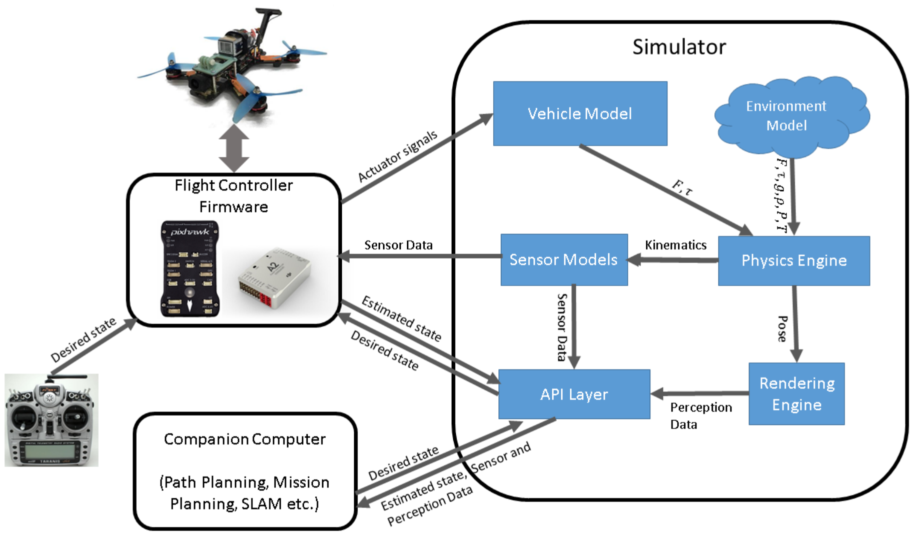


Figure 6.1.: AirSim Block Diagram [20]

Unreal Engine (UE) [5] provides support for computer vision analysis that is transferable to the real world. Some of the features of this architecture enables hardware-in-the-loop (HIL) simulation, where a flight controller such as Pixhawk [11] directly interacts with the simulation environment.

When the simulator is started, the simulator provides all the sensor data from the environment i.e. the simulated world to the flight controller which in return outputs the actuation signals. These actuator signals are fed to the vehicle model as an input component of the simulator engine. The objective of the vehicle model is to calculate the forces and torques which are resultant of the actuation mechanism. Moreover, the engine has to take into account of additional forces such as drag, friction, gravity etc which might be applicable depending on the dynamics of the vehicle. These forces are taken as input arguments by the physics engine to compute the following state of kinematics of the vehicle in the simulated world. The reference of ground truth for this kinematics is provided by the simulated sensor models.

The final input state to the flight controller can be manually entered by using a remote controller or initialization script in autonomous mode. This script may perform high-level computations such as determining next desired waypoint, performing Simultaneous Localization and Mapping (SLAM), calculating desired trajectory etc. The computational complexity and demand for computation power is quite high for the companion computer as it may have to perform large amount of data processing in real-time to process all the data generated by the cameras, LIDARs etc. To leverage such advanced sensor systems, the environment model requires to be rich in details which can be leveraged through the rendering technique used in the Unreal Engine.[5]

The interaction between companion computer and the simulator is accomplished via a set of APIs that enables the companion computer to observe the sensor streams, vehicle state and send commands. These APIs are designed to prevent the companion computer of being aware of its state of running under simulation or in the real world. This feature serves useful in development and testing of algorithms in simulator and its deployment to real vehicle without the need to make additional changes for its implementation. The AirSim code base is implemented as a plugin for the Unreal engine that can be dropped into any Unreal project. The Unreal engine platform offers an elaborate marketplace with hundreds of pre-made detailed environments, many created using photogrammetry techniques to generate reasonably faithful reconstruction of real-world scenes. [16]

## 6.2. Mathematical Model

### 6.2.1. Vehicle Model

AirSim defines the vehicle as a rigid body that may have arbitrary number of actuators [20]. Some of the vehicle parameters are mass, inertia, drag, friction etc. A vehicle is defines as a collection of K vertices at positions $\{r_1, ...r_k\}$ and normals $\{n_1, ...n_k\}$ with control inputs as $\{u_1, ...u_k\}$. Then forces and torques are computed as follows

$$F_i = C_T \rho \omega_{max}^2 D^4 u_i$$

$$\tau_i = \frac{1}{2\pi} C_{pow} \rho \omega_{max}^2 D^5 u_i$$

$$\rho = \frac{P}{R.T}$$

where, $C_T$ and $C_{pow}$ are the thrust and power coefficients, $\rho$ is the air density and D is the propeller diameter $\omega_{max}$ is the maximum angular velocity. P is the air pressure, T is the temperature and R is the specific gas constant

### 6.2.2. Gravity

Gravitational forces are calculated as per the following formula:

$$g = g_0 . \frac{R_e^2}{(R_e + h)^2} \approx g_0 .(1 - 2\frac{h}{R_e})$$

Where $R_e$is the Earth's radius and $g_0 is$the gravitational constant measured at surface

### 6.2.3. Magnetic Field

The Horizontal Field Intensity (H), latitudinal (X) , longitudinal (Y) and vertical field (Z) components of the magnetic field vector are calculated as follows

$$H = |B| \cos \alpha$$

$$Z = |B| \sin \alpha$$

$$X = |H| \cos \beta$$

$$Y = |H| \sin \beta$$

### 6.2.4. Accelerations

The Force and Torque are calculated as follows.

$$F_{net} = \sum_i F_i + F_d$$

$$\tau_{net} = \sum_i |\tau_i + r_i \times F_i| + \tau_d$$

## 6.3. Flight Controller

The flight controller uses desired state and the sensor data as inputs to compute the estimate of current state and output the actuator control signals to achieve the desired state. In case of quadrotors, pitch, roll and yaw angles are desired states and sensor data from accelerometer and gyroscope is also considered to estimate the current angles to compute the motor signals required to achieve the desired angles.

### 6.3.1. Hardware-in-Loop

HITL or HIL (Hardware-in-Loop) means flight controller runs in actual hardware such as Naze32 or Pixhawk chip. The hardware is connected to PC using USB port. Simulator communicates with the device through USB port to retrieve actuator signals and send it simulated sensor data. It requires more steps to set up and is usually hard to debug as compared to SITL, significant problem being simulator clock and device clock running on their own speed and accuracy. Also, USB connection (which is usually only USB 2.0) may not be enough for real-time communication.

### 6.3.2. Software-in-Loop

SITL or SIL (software-in-loop simulation) mode uses the firmware in computer as opposed to separate board. This is generally fine except the fact that code paths specific to device are not hampered. Moreover, none of the code now runs with real-time clock usually provided by specialized hardware board. For well-designed flight controllers with software clock, these are usually not concerning issues.

### 6.3.3. simple_flight

simple_flight is the built-in flight controller in AirSim. It is used by default and does not need any configuration. AirSim also supports PX4 as another flight controller for advanced users. Flight controllers are generally designed to run on actual hardware on vehicles and their support varies for running in simulator. These have complex build usually lacking cross-platform support, therefore, are difficult to configure for non-expert users. simple_flight is designed as a library with clean interface that can work onboard the vehicle as well as simulator. The core principle is that flight controller must not be able to specify special simulation mode and therefore must not be able to differentiate between running under simulation or real vehicle. This way flight controller is simply viewed as collection of algorithms packaged in a library.

Moreover, this code is developed as dependency free header-only pure standard C++11 code requiring no special build to compile simple_flight. To work on any project, the source code is required to be copied to the project that is being worked on. As no additional setup is needed to work with simple_flight, it is advantageous over other flight controllers. Also, simple_flight uses steppable clock which enables the simulation to be paused and still it will not be affected by high variance low precision clock that operating system provides. Further, simple_flight is simple, cross-platform and 100% header-only dependency-free C++ code enabling user to step through from simulator to inside flight controller code within same code base.

#### 6.3.3.1 Control

Vehicle control in simple_flight is done by taking in desired input as angle rate, angle level, velocity or position. Each axis of control can be specified with one of these modes. Internally simple_flight uses cascade of PID controllers to finally generate actuator signals. This means position PID drives velocity PID which drives angle level PID which finally drives angle rate PID.

#### 6.3.3.2 State Estimation

In current release ground truth from simulator is used for state estimation. Work is in progress to add complimentary filter based state estimation for angular velocity and orientation using 2 sensors (gyroscope, accelerometer) and integrate another library to do velocity and position estimation using 4 sensors (gyroscope, accelerometer, magnetometer and barometer) using EKF.

### 6.3.4. PX4

PX4 [18] flight code is used by simulator to control a computer modeled vehicle in a virtual world. PX4 can be integrated with different simulators like Gazebo and AirSim to explore flight controller APIs for various UAV functionalities. All PX4 airframes share a single codebase (this includes other robotic systems like boats, rovers, submarines etc.). The complete system design is reactive. Fig. 6.3 shows the block diagram for PX4

1. The functionality is clustered into reusable and exchangeable components.
2. Asynchronous communication is established with message passing.
3. The system is robustly designed to work with varying workload.

#### 6.3.4.1 Flight Stack

PX4 flight stack includes guidance, navigation and control algorithms for autonomous drones. It consists of controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position. As shown in the figure, the building blocks consist of the full pipeline from sensors, RC input and autonomous flight control (Navigator), down to the motor or servo control (Actuators).

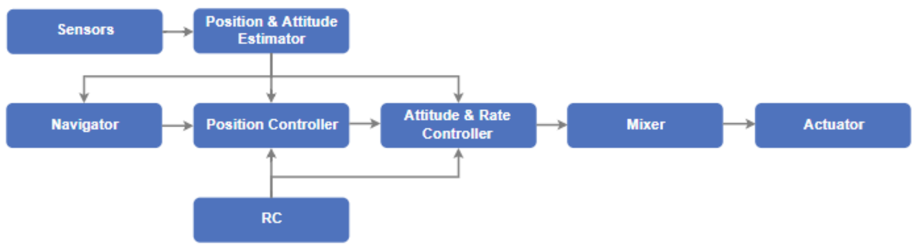


Figure 6.2.: PX4 Flight Stack Block Diagram [18]

- The estimator takes one or more sensor inputs, combines them, and computes a vehicle state (for example the attitude from IMU sensor data).
- The controller takes a setpoint and a measurement or estimated state (process variable) as input. Its goal is to adjust the value of the process variable such that it matches the setpoint. The output is a correction to eventually reach that setpoint (for example the position controller takes position setpoints as inputs, the currently estimated position as the process variable, and attitude and thrust setpoint as output that move the vehicle towards the desired position).
- The mixer takes force commands (e.g. turn right) and translates them into individual motor commands. This translation is specific for a vehicle type and depends on various factors like motor arrangements with respect to the center of gravity, or the vehicle's rotational inertia.

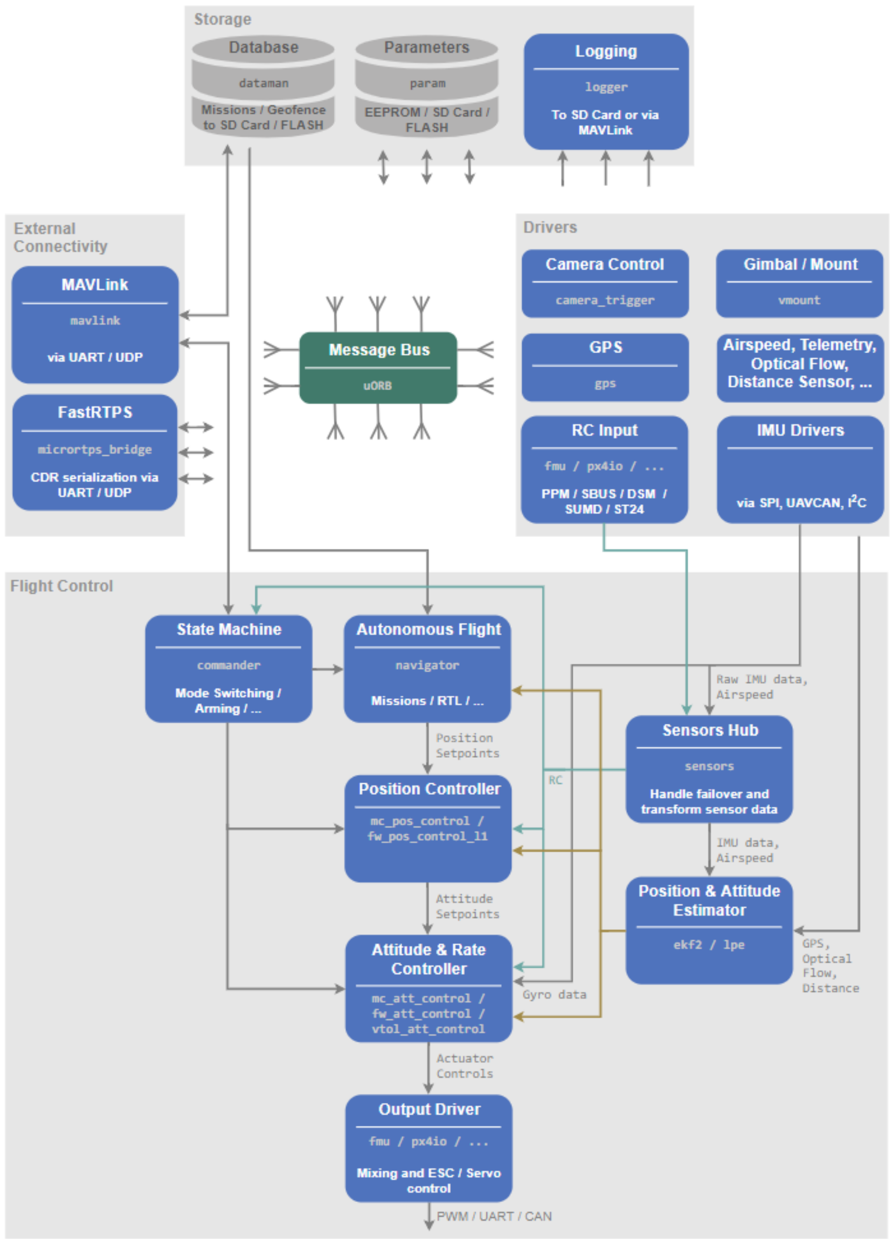


Figure 6.3.: PX4 Block Diagram [18]

# 7. Conclusion

From the discussions in the previous sections, it is clear that technological trends favoring UAVs have been growing exponentially. Cheaper computation power, improving performance per dollar and improving performance per watt have been the major contributing factors working in favour of UAVs. Massive data being generated has a lot of potential to revolutionize the Robotics and UAV industry in the years to come. From the first phase of this report, it can be concluded that UAVs find a wide array of application across industry and one of the major challenges in designing versatile drones is trajectory generation. Some mathematical functions were discussed weighing their pros and cons and trigonometric sine, cosine were selected for their smooth trajectory functions.

In the second phase, Simulators have been discussed from a modelling and specifications point of view, which helps in reducing costs in testing and validation of UAV flight controllers. An in-depth analysis of multiple simulators is carried out at multiple fronts including but not limited to long-term support, integrations, effective physical modelling etc. Of these, AirSim seems to stand out of the rest of them given its highly versatile and powerful APIs, strong community support and robust system model. The in-depth analysis and documentation presented in this study shall prove to be useful for future works, which opens the potential of interfacing Reinforcement Learning with the Simulators to provide a foundation for next level of Autonomy and Artificial Intelligence.

# A. (Appendix) AirSim: Documentation and Installation Instructions

This section and its contents have been quoted from the AirSim official documentation which is available online at [1]. This documentation has in-depth description of AirSim Code Structure, System Requirements, APIs, Vehicle and Environment Models, Setup Installation instructions and Guide to use the simulator.

## A.1. Code Structure

### A.1.1. AirLib

Majority of the code is located in AirLib. This is a self-contained library that can be compiled with any C++11 compiler. AirLib consists of the following components:

1. Physics engine: This is header-only physics engine. It is designed to be fast and extensible to implement different vehicles.
2. Sensor models: This is header-only models for Barometer, IMU, GPS and Magnetometer
3. Vehicle models: This is header-only models for vehicle configurations and models. Currently it has model for a MultiRotor and a configuration for PX4 QuadRotor in the X configuration
4. Control library: This part of AirLib provides abstract base class for our APIs and concrete implementation for specific vehicle platforms such as MavLink. It also has classes for the RPC client and server.

### A.1.2. Unreal/Plugins/AirSim

The only portion of AirSim dependent on Unreal engine is the plugin. It is kept isolated to enable implementation of simulator for other platforms as well. The Unreal code takes advantage of its UObject based classes including Blueprints.

1. SimMode_ classes: To support various simulator modes such as pure Computer Vision mode where there is no drone. The SimMode classes help implement many different modes.
2. VehiclePawnBase: This is the base class for all vehicle pawn visualizations.
3. VehicleBase: This class provides abstract interface to implement a combination of rendering component (i.e. Unreal pawn), physics component (i.e. MultiRotor) and controller (i.e. MavLinkHelper).

### A.1.3. MavLinkCom

This library is developed by Chris Lovett at Microsoft that provides C++ classes to talk to the MavLink devices. This library is stand alone and can be used in any project.

### A.1.4. Unreal Framework

The following picture illustrates how AirSim is loaded and invoked by the Unreal Game Engine:

## A.2. System Requirements

System Requirements as per [20]

### A.2.1. Hardware Requirements

GPUs such as NVidia 1080 or NVidia Titan series with powerful desktop such as one with 64GB RAM, 6+ cores, SSDs and 2-3 displays (ideally 4K) is recommended. The development experience on high-end laptops is generally sub-par compared to powerful desktops however they might be useful in a pinch. Generally laptops with discrete NVidia GPU (at least M2000 or better) with 64GB RAM, SSDs and hopefully 4K display work well. Laptops with only integrated graphics might not work well.

### A.2.2. Software Requirements

- • Unreal Engine
- Windows (Preferable)/ Linux OS
- Anaconda (preferable Python 3.5 or above for APIs)
- ROS

Windows 10 and Visual Studio 2017 are recommended as development environment. This is because the support for other OSes and IDE is not as mature on the Unreal Engine side.

## A.3. Installation Instructions

1. Download the Epic Games Launcher. While the Unreal Engine is open source and free to download, registration is still required.
2. Run the Epic Games Launcher, open the Library tab on the left pane. Click on the Add Versions which should show the option to download Unreal 4.18. If there are multiple versions of Unreal installed then make sure 4.18 is set to current by clicking down arrow next to the Launch button for the version.

AirSim binaries can be installed from releases or compile from the source as given below (Windows, Linux).

### A.3.1. Option 1: Download pre-compiled Binaries

Precompiled binaries can be easily downloaded and run to get started immediately. Download the binaries for the environment of choice from [15]

### A.3.2. Option 2: Build AirSim

- Install Visual Studio 2017.
- Make sure to select VC++ and Windows SDK 8.1 while installing VS 2017.
- Start *x64 Native Tools Command Prompt for VS 2017*.
- Clone the repo: git clone *https://github.com/Microsoft/AirSim.git*, and go the AirSim directory by *cd AirSim*.
- Run *build.cmd* from the command line. This will create ready to use plugin bits in the *Unreal\Plugins* folder that can be dropped into any Unreal project.

## A.4. Setup Instructions

### A.4.1. Unreal Environment

Setting Up the Unreal Project with Built-in Blocks Environment: To get up and running fast, the Blocks project that already comes with AirSim can be used. This is not very highly detailed environment to keep the repo size reasonable but it can be used for various testing and it is the easiest way to start experimenting with the simulator.

### A.4.2. AirSim Environment: Blocks

Blocks environment is available in repo in folder *Unreal/Environments/Blocks* and is designed to be lightweight in size. That means it is very basic but fast.

Here are quick steps to get Blocks environment up and running: (For Windows)

1. Navigate to folder *AirSim\Unreal\Environments\Blocks* and run *update_from_git.bat*.
2. Double click on generated .sln file to open in Visual Studio 2017 or newer.
3. Make sure *Blocks* project is the startup project, build configuration is set to *DebugGame_Editor* and Win64. Hit F5 to run.
4. Press the Play button in Unreal Editor.

## A.5. Programmatic Control

AirSim exposes APIs so user can interact with the vehicle in the simulation programmatically. These APIs are used to retrieve images, get state, control the vehicle and so on. The APIs are exposed through the RPC, and are accessible via a variety of languages, including C++, Python, C# and Java. These APIs are also available as part of a separate, independent cross-platform library, so it can be deployed on a companion computer on a UAV. This way the code can be written and tested in the simulator, and later executed on the real UAV.

To use Python to call AirSim APIs, it is recommended to use Anaconda with Python 3.5 or later versions and install the following package:

```
pip install msgpack-rpc-python
```

## A.6. Application Programming Interfaces (APIs)

The following is a list of some of the common APIs

- *reset*: This resets the vehicle to its original starting state. Note to call *enableApiControl* and *armDisarm* again after the call to reset.
- *confirmConnection*: Checks state of connection every 1 sec and reports it in Console so user can see the progress for connection.
- *enableApiControl*: For safety reasons, by default API control for autonomous vehicle is not enabled and human operator has full control (usually via RC or joystick in simulator). The client must make this call to request control via API. It is likely that human operator of vehicle might have disallowed API control which would mean that *enableApiControl* has no effect. This can be checked by *isApiControlEnabled*.

- *isApiControlEnabled*: Returns true if API control is established. If false (which is default) then API calls would be ignored. After a successful call to *enableApiControl*, the *isApiControlEnabled* should return true.
- *ping*: If connection is established then this call will return true otherwise it will be blocked until timeout.
- *simPrintLogMessage*: Prints the specified message in the simulator's window. If message_param is also supplied then it's printed next to the message and in that case if this API is called with same message value but different message_param again then previous line is overwritten with new line (instead of API creating new line on display). For example, *simPrintLogMessage*("Iteration: ", to_string(i)) keeps updating same line on display when API is called with different values of i. The valid values of severity parameter is 0 to 3 inclusive that corresponds to different colors.
- *simGetObjectPose*, *simSetObjectPose*: Gets and sets the pose of specified object in Unreal environment. Here the object means "actor" in Unreal terminology. They are searched by tag as well as name. Please note that the names shown in UE Editor are auto-generated in each run and are not permanent. So if the actor needs to be referred by name, then its auto-generated name needs to be changed in UE Editor. Alternatively, add a tag to actor which can be done by clicking on that actor in Unreal Editor and then going to Tags property, click "+" sign and add some string value. If multiple actors have same tag then the first match is returned. If no matches are found then *NaN* pose is returned. The returned pose is in NED coordinates in SI units with its origin at Player Start. For *simSetObjectPose*, the specified actor must have Mobility set to Movable or otherwise you will get undefined behavior. The *simSetObjectPose* has parameter *teleport* which means object is moved through other objects in its way and it returns true if move was successful

## A.7. APIs for Multimotor

Multirotor can be controlled by specifying angles, velocity vector, destination position or some combination of these. There are corresponding *move** APIs for this purpose. When doing position control, we need to use some path following algorithm. By default AirSim uses carrot following algorithm. This is often referred to as "high level control" because it specifies high level goal and the firmware takes care of the rest. Currently lowest level control available in AirSim is *moveByAngleThrottleAsync* API.

Python Scripts needs to be generated with the AirSim Simulator to perform different maneuvers, whereupon specifying parameters such as velocity, angle, position etc, API calls to simulator can be made. Regarding control of these parameters, these APIs make it easy, as such parameters can be passed as arguments to function calls in these libraries. (For eg. Velocities in the X,Y,Z directions can be specified as:

*Client.moveByVelocity(vx=x_velocity, vy=y_velocity, vz=z_velocity, duration=duration_of_movement)*

List of built in APIs given below can be found in the Airsim repo:

*https://github.com/Microsoft/AirSim/blob/master/PythonClient/airsim/client.py*

Some of the APIs are shown in the Fig. A.2 A.3 A.4

## A.8. APIs Explained

Some of the common APIs are explained below:

### A.8.1. getMultirotorState

This API returns the state of the vehicle in one call. The state includes, collision, estimated kinematics (i.e. kinematics computed by fusing sensors), and timestamp (nanoseconds since epoch). The kinematics here means 6 quantities: position, orientation, linear and angular velocity, linear and angular acceleration. Please note that simple_slight currently doesn't support state estimator which means estimated and ground truth kinematics values would be same for simple_flight. Estimated kinematics are however available for PX4 except for angular acceleration. All quantities are in NED coordinate system, SI units in world frame except for angular velocity and accelerations which are in body frame.

### A.8.2. Async methods, duration and max_wait_seconds

Many API methods has parameters named *duration* or *max_wait_second*s and they have Async as suffix, for example, *takeoffAsync*. These methods will return immediately after starting the task in AirSim so that your client code can do something else while that task is being executed. If user wants to wait for this task to complete then *waitOnLastTask* should be called like this:

```
//C++
client.takeoffAsync()->waitOnLastTask();
# Python
client.takeoffAsync().join()
```

If another command is started then it automatically cancels the previous task and starts new command. This allows to use pattern where the code continuously does the sensing, computes a new trajectory to follow and issues that path to vehicle in AirSim. Each newly issued trajectory cancels the previous trajectory allowing the code to continuously do the update as new sensor data arrives.

All Async method returns concurrent.futures.Future in Python (std::future in C++). Please note that these future classes currently do not allow to check status or cancel the task; they only allow to wait for task to complete. AirSim does provide API *cancelLastTask*

### A.8.3. Drivetrain

There are two modes to fly UAV: drivetrain parameter is set to *airsim.DrivetrainType.ForwardOnly* or *airsim.DrivetrainType.MaxDegreeOfFreedom*. When *ForwardOnly* is specified, it means that vehicle's front should always point in the direction of travel. So if drone needs to take left turn then it would first rotate so front points to left. This mode is useful when only front camera is available and drone is being operated using FPV view. This is more or less like travelling in car which only has front view. The *MaxDegreeOfFreedom* means it does not matter where the front points to. So when left turn is taken, it just start going left like crab. Quadrotors can go in any direction regardless of where front points to. The *MaxDegreeOfFreedom* enables this mode.

### A.8.4. yaw_mode

*yaw_mode* is a struct *YawMode* with two fields, *yaw_or_rate* and *is_rate*. If is_rate field is *True* then *yaw_or_rate field* is interpreted as angular velocity in degrees/sec which means vehicle should rotate continuously around its axis at that angular velocity while moving. If *is_rate* is *False* then *yaw_or_rate* is interpreted as angle in degrees which means vehicle should rotate to specific angle (i.e. yaw) and keep that angle while moving.

When *yaw_mode.is_rate == true,* the drivetrain parameter shouldn't be set to *ForwardOnly* because it is contradicting by saying that keep front pointing ahead but also rotate continuously. In most cases, it is not desired for yaw to change which can be done by setting yaw rate of 0. The shorthand for this is *airsim.YawMode.Zero()* (or in C++: *YawMode::Zero()).*

### A.8.5. lookahead and adaptive_lookahead

When UAV is needed to follow a path, AirSim uses "carrot following" algorithm. This algorithm operates by looking ahead on path and adjusting its velocity vector. The parameters for this algorithm is specified by *lookahead* and *adaptive_lookahead*. For most of the time it is desired for algorithm to auto-decide the values by simply setting

```
lookahead = −1 and adaptive_lookahead = 0
```

More complex maneuvers can be carried out by combining the given APIs. Following is an example to show the usage of APIs to control the motion of quadrotor as shown in Fig. A.5

*hello_drone.py* uses the RPC client to connect to the RPC server that is automatically started by the AirSim. The RPC server routes all the commands to a class that implements *MultirotorApiBase*. In essence, *MultirotorApiBase* defines our abstract interface for getting data from the quadrotor and sending back commands.

From Reinforcements Learning point of view, this interface can be used by the network to control the UAV, while also getting feedback regarding the state of the drone using Status

APIs which returns the complete state of the UAV including but not limited to the below mentioned. Fig. A.6 shows the reply of status API call

Another important feature is that all this information can be logged to create log files and transcripts for later analysis / verification and also supervised learning techniques if need be in the future.

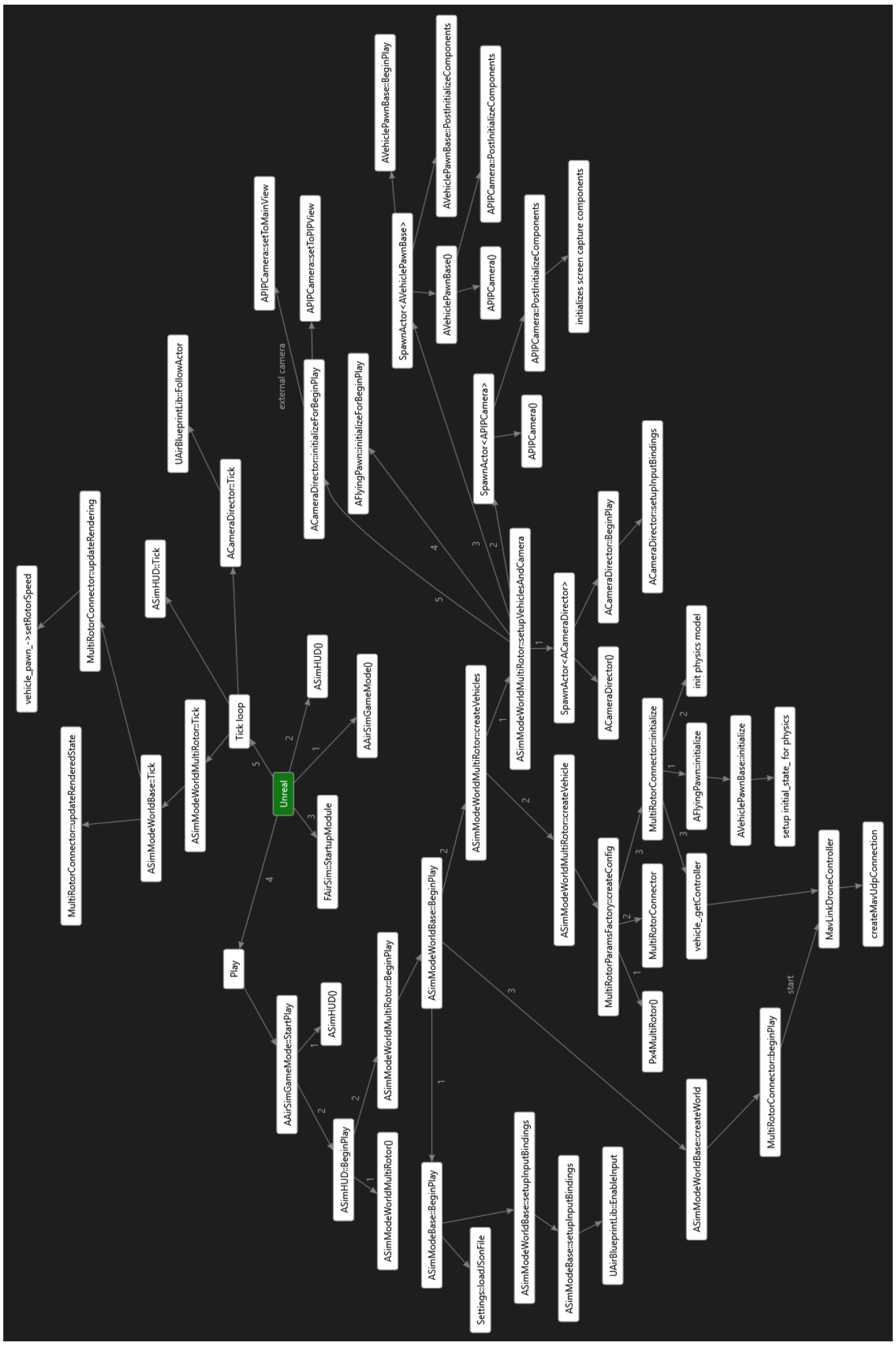
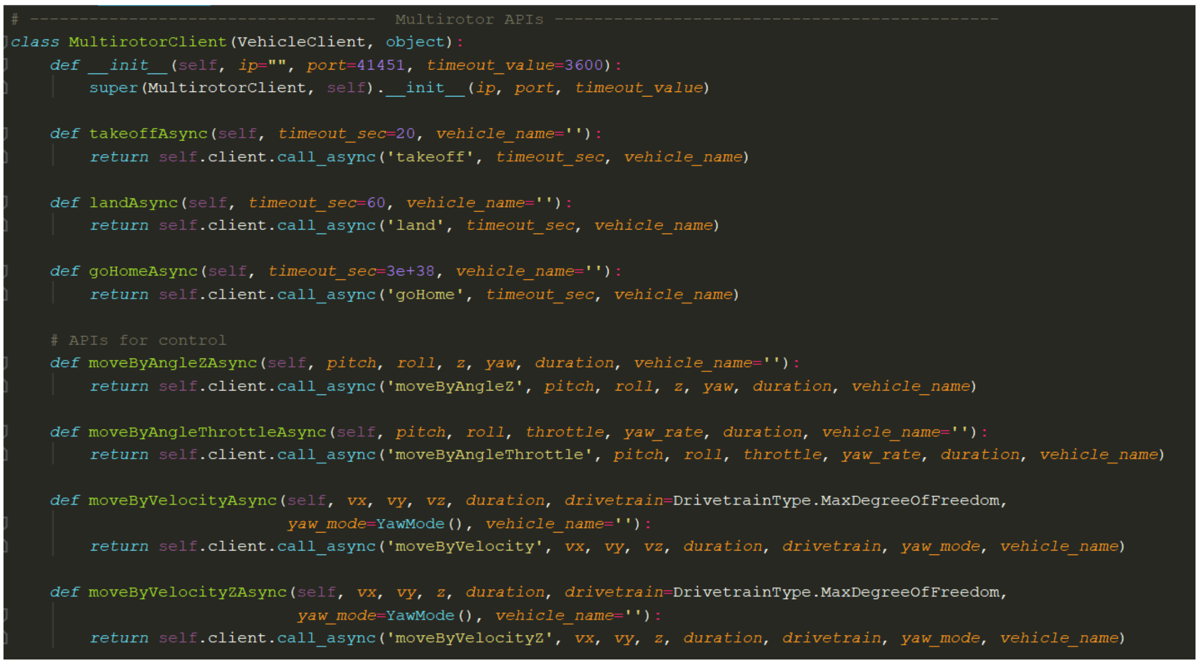
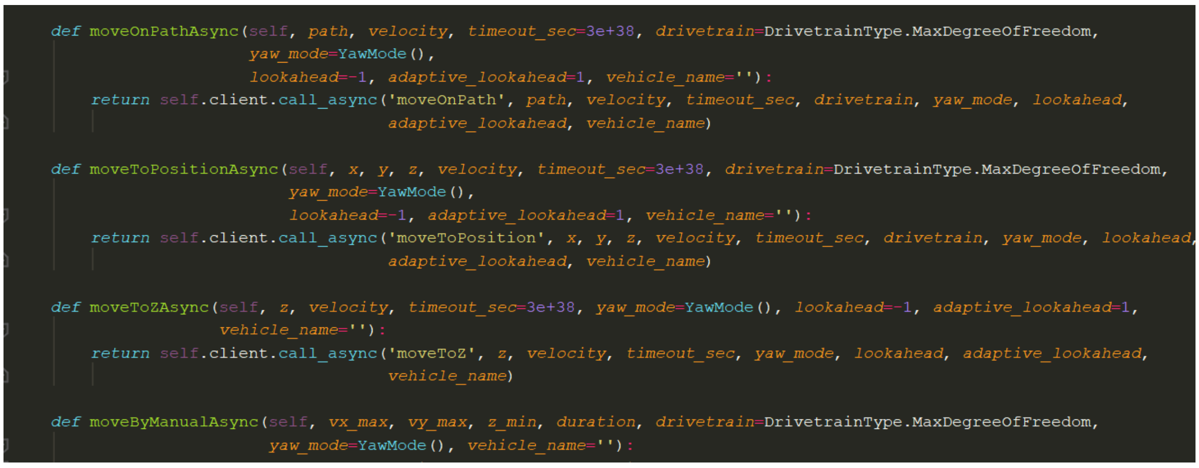

Unreal
Play
Tick loop
AAirSimGameMode()
ASimHUD()
FAirSim::StartupModule
AAirSimGameMode::StartPlay
ASimHUD()
ASimHUD::BeginPlay
ASimModeWorldMultiRotor()
ASimModeWorldMultiRotor::BeginPlay
ASimModeWorldBase::BeginPlay
ASimModeBase::BeginPlay
Settings::loadJSonFile
ASimModeWorldBase::setupInputBindings
ASimModeBase::setupInputBindings
UAirBlueprintLib::EnableInput
ASimModeWorldBase::createWorld
MultiRotorConnector::beginPlay
start
MavLinkDroneController
createMavUdpConnection
ASimModeWorldMultiRotor::createVehicles
ASimModeWorldMultiRotor::createVehicle
MultiRotorParamsFactory::createConfig
MultiRotorConnector
Px4MultiRotor()
MultiRotorConnector::initialize
AFlyingPawn::initialize
AVehiclePawnBase::initialize
setup initial_state_ for physics
init physics model
vehicle_.getController
ASimModeWorldMultiRotor::setupVehiclesAndCamera
SpawnActor&lt;ACameraDirector&gt;
ACameraDirector()
ACameraDirector::BeginPlay
ACameraDirector::setupInputBindings
SpawnActor&lt;APIPCamera&gt;
APIPCamera()
APIPCamera::PostInitializeComponents
initializes screen capture components
SpawnActor&lt;AVehiclePawnBase&gt;
AVehiclePawnBase()
APIPCamera()
APIPCamera::PostInitializeComponents
AVehiclePawnBase::PostInitializeComponents
AVehiclePawnBase::BeginPlay
AFlyingPawn::initializeForBeginPlay
ACameraDirector::initializeForBeginPlay
APIPCamera::setToPIPView
APIPCamera::setToMainView
external camera
ACameraDirector::Tick
UAirBlueprintLib::FollowActor
ASimHUD::Tick
ASimModeWorldMultiRotor::Tick
ASimModeWorldBase::Tick
MultiRotorConnector::updateRenderedState
MultiRotorConnector::updateRendering
vehicle_pawn_->setRotorSpeed

Figure A.1.: Unreal Engine Block Diagram [1]



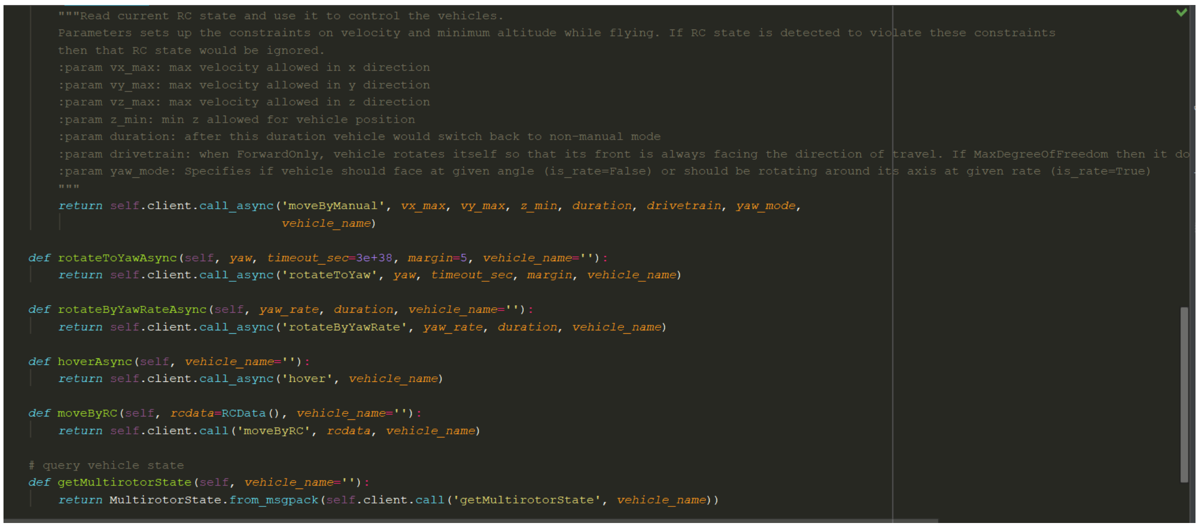
```python
# -----------------------------------  Multirotor APIs ---------------------------------------------
class MultirotorClient(VehicleClient, object):
    def __init__(self, ip="", port=41451, timeout_value=3600):
        super(MultirotorClient, self).__init__(ip, port, timeout_value)

    def takeoffAsync(self, timeout_sec=20, vehicle_name=''):
        return self.client.call_async('takeoff', timeout_sec, vehicle_name)

    def landAsync(self, timeout_sec=60, vehicle_name=''):
        return self.client.call_async('land', timeout_sec, vehicle_name)

    def goHomeAsync(self, timeout_sec=3e+38, vehicle_name=''):
        return self.client.call_async('goHome', timeout_sec, vehicle_name)

    # APIs for control
    def moveByAngleZAsync(self, pitch, roll, z, yaw, duration, vehicle_name=''):
        return self.client.call_async('moveByAngleZ', pitch, roll, z, yaw, duration, vehicle_name)

    def moveByAngleThrottleAsync(self, pitch, roll, throttle, yaw_rate, duration, vehicle_name=''):
        return self.client.call_async('moveByAngleThrottle', pitch, roll, throttle, yaw_rate, duration, vehicle_name)

    def moveByVelocityAsync(self, vx, vy, vz, duration, drivetrain=DrivetrainType.MaxDegreeOfFreedom,
                            yaw_mode=YawMode(), vehicle_name=''):
        return self.client.call_async('moveByVelocity', vx, vy, vz, duration, drivetrain, yaw_mode, vehicle_name)

    def moveByVelocityZAsync(self, vx, vy, z, duration, drivetrain=DrivetrainType.MaxDegreeOfFreedom,
                             yaw_mode=YawMode(), vehicle_name=''):
        return self.client.call_async('moveByVelocityZ', vx, vy, z, duration, drivetrain, yaw_mode, vehicle_name)
```

Figure A.2.: API I



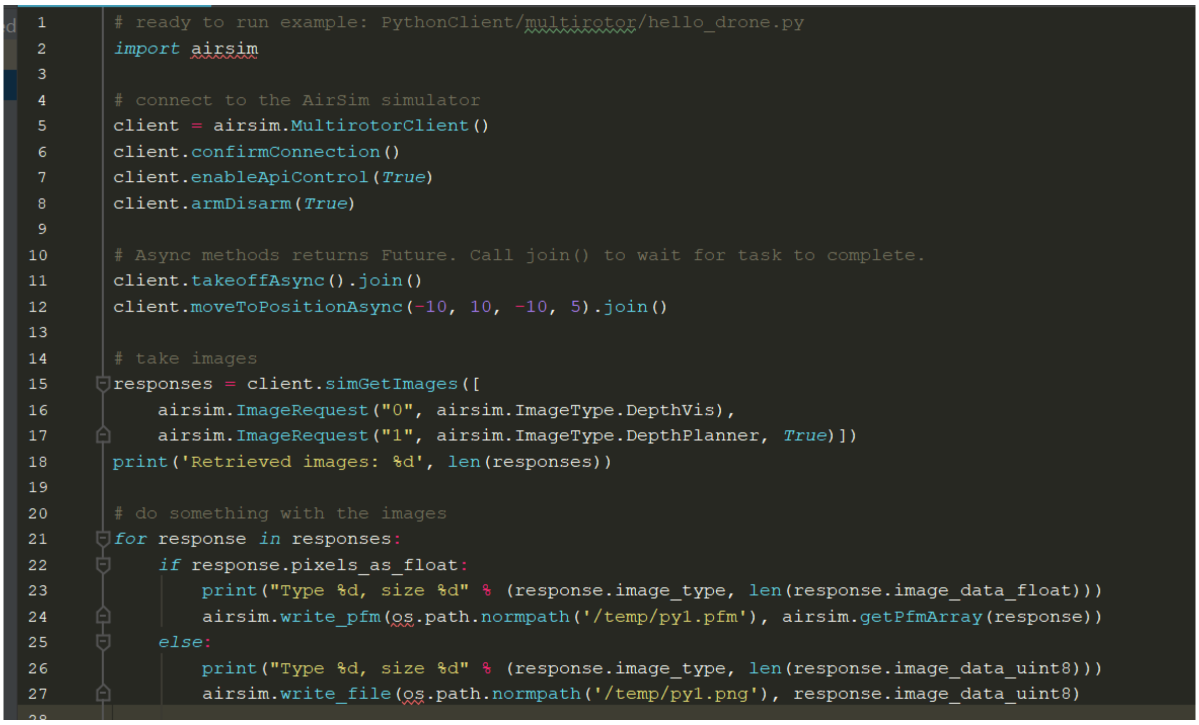
```python
    def moveOnPathAsync(self, path, velocity, timeout_sec=3e+38, drivetrain=DrivetrainType.MaxDegreeOfFreedom,
                        yaw_mode=YawMode(),
                        lookahead=-1, adaptive_lookahead=1, vehicle_name=''):
        return self.client.call_async('moveOnPath', path, velocity, timeout_sec, drivetrain, yaw_mode, lookahead,
                                      adaptive_lookahead, vehicle_name)

    def moveToPositionAsync(self, x, y, z, velocity, timeout_sec=3e+38, drivetrain=DrivetrainType.MaxDegreeOfFreedom,
                            yaw_mode=YawMode(),
                            lookahead=-1, adaptive_lookahead=1, vehicle_name=''):
        return self.client.call_async('moveToPosition', x, y, z, velocity, timeout_sec, drivetrain, yaw_mode, lookahead,
                                      adaptive_lookahead, vehicle_name)

    def moveToZAsync(self, z, velocity, timeout_sec=3e+38, yaw_mode=YawMode(), lookahead=-1, adaptive_lookahead=1,
                     vehicle_name=''):
        return self.client.call_async('moveToZ', z, velocity, timeout_sec, yaw_mode, lookahead, adaptive_lookahead,
                                      vehicle_name)

    def moveByManualAsync(self, vx_max, vy_max, z_min, duration, drivetrain=DrivetrainType.MaxDegreeOfFreedom,
                          yaw_mode=YawMode(), vehicle_name=''):
```

Figure A.3.: API II



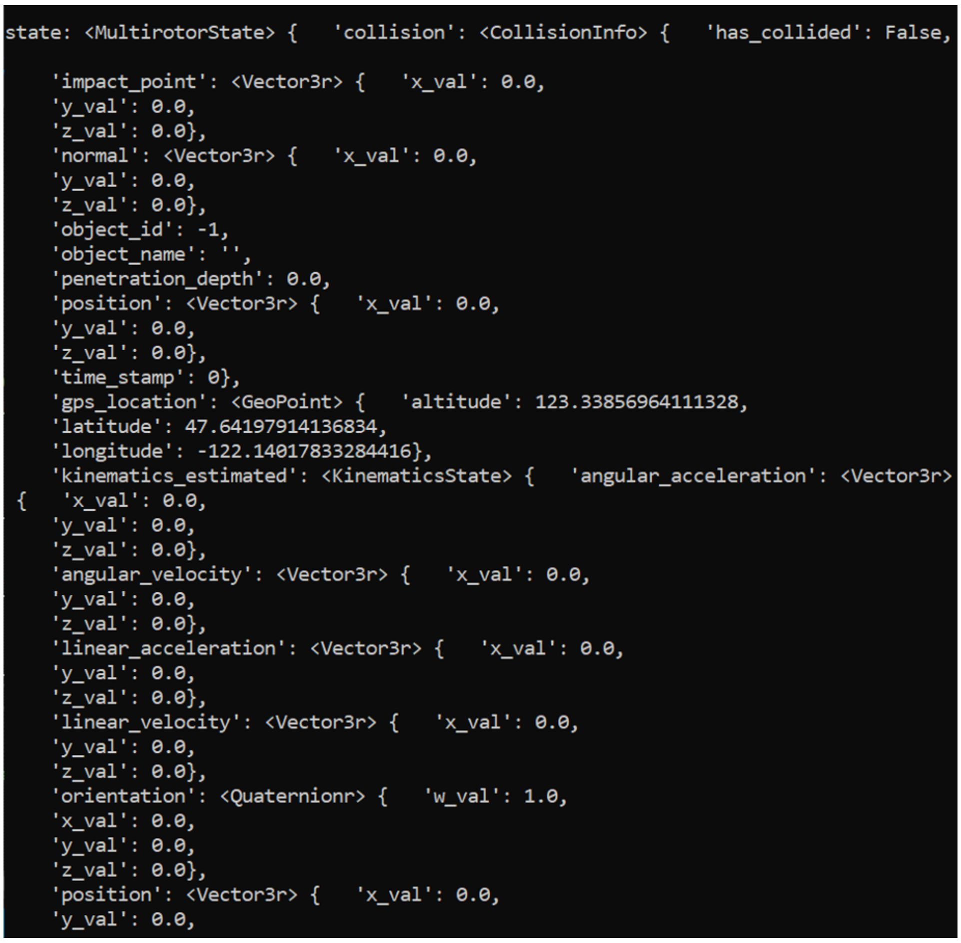
```
        """Read current RC state and use it to control the vehicles.
        Parameters sets up the constraints on velocity and minimum altitude while flying. If RC state is detected to violate these constraints
        then that RC state would be ignored.
        :param vx_max: max velocity allowed in x direction
        :param vy_max: max velocity allowed in y direction
        :param vz_max: max velocity allowed in z direction
        :param z_min: min z allowed for vehicle position
        :param duration: after this duration vehicle would switch back to non-manual mode
        :param drivetrain: when ForwardOnly, vehicle rotates itself so that its front is always facing the direction of travel. If MaxDegreeOfFreedom then it do
        :param yaw_mode: Specifies if vehicle should face at given angle (is_rate=False) or should be rotating around its axis at given rate (is_rate=True)
        """
        return self.client.call_async('moveByManual', vx_max, vy_max, z_min, duration, drivetrain, yaw_mode,
                                      vehicle_name)

    def rotateToYawAsync(self, yaw, timeout_sec=3e+38, margin=5, vehicle_name=''):
        return self.client.call_async('rotateToYaw', yaw, timeout_sec, margin, vehicle_name)

    def rotateByYawRateAsync(self, yaw_rate, duration, vehicle_name=''):
        return self.client.call_async('rotateByYawRate', yaw_rate, duration, vehicle_name)

    def hoverAsync(self, vehicle_name=''):
        return self.client.call_async('hover', vehicle_name)

    def moveByRC(self, rcdata=RCData(), vehicle_name=''):
        return self.client.call('moveByRC', rcdata, vehicle_name)

    # query vehicle state
    def getMultirotorState(self, vehicle_name=''):
        return MultirotorState.from_msgpack(self.client.call('getMultirotorState', vehicle_name))
```

Figure A.4.: API III



```
# ready to run example: PythonClient/multirotor/hello_drone.py
import airsim

# connect to the AirSim simulator
client = airsim.MultirotorClient()
client.confirmConnection()
client.enableApiControl(True)
client.armDisarm(True)

# Async methods returns Future. Call join() to wait for task to complete.
client.takeoffAsync().join()
client.moveToPositionAsync(-10, 10, -10, 5).join()

# take images
responses = client.simGetImages([
    airsim.ImageRequest("0", airsim.ImageType.DepthVis),
    airsim.ImageRequest("1", airsim.ImageType.DepthPlanner, True)])
print('Retrieved images: %d', len(responses))

# do something with the images
for response in responses:
    if response.pixels_as_float:
        print("Type %d, size %d" % (response.image_type, len(response.image_data_float)))
        airsim.write_pfm(os.path.normpath('/temp/py1.pfm'), airsim.getPfmArray(response))
    else:
        print("Type %d, size %d" % (response.image_type, len(response.image_data_uint8)))
        airsim.write_file(os.path.normpath('/temp/py1.png'), response.image_data_uint8)
```

Figure A.5.: hello_drone.py



```
state: <MultirotorState> {   'collision': <CollisionInfo> {   'has_collided': False,

    'impact_point': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'normal': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'object_id': -1,
    'object_name': '',
    'penetration_depth': 0.0,
    'position': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'time_stamp': 0},
    'gps_location': <GeoPoint> {   'altitude': 123.33856964111328,
    'latitude': 47.64197914136834,
    'longitude': -122.14017833284416},
    'kinematics_estimated': <KinematicsState> {   'angular_acceleration': <Vector3r>
 {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'angular_velocity': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'linear_acceleration': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'linear_velocity': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'orientation': <Quaternionr> {   'w_val': 1.0,
    'x_val': 0.0,
    'y_val': 0.0,
    'z_val': 0.0},
    'position': <Vector3r> {   'x_val': 0.0,
    'y_val': 0.0,
```

Figure A.6.: Status API call results

# Bibliography

[1] *AirSim Documentation, Online: https://microsoft.github.io/AirSim/docs/code*$_s$*tructure/.*$(p.\ 31), (p.\ 39)$

[2] *TUM Simulator ROS Online: http://wiki.ros.org/tum*$_s$*imulator.* $(p.\ 20)$

[3] Q. Ali and S. Montenegro. Explicit model following distributed control scheme for formation flying of mini uavs. *IEEE Access*, 4:397–406, 2016. (p. 6)

[4] Alireza Babaei and Amirhossein Karimi. Optimal Trajectory-Planning of UAVs via B-Splines and Disjunctive Programming. (p. 1), (p. 3)

[5] Epic Games Brian Karis. Real shading in unreal engine 4. *Online: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf*, 2013. (p. 23), (p. 24)

[6] P. Castillo, A. Dzul, and R. Lozano. Real-time stabilization and tracking of a four rotor mini-rotorcraft. In *Proc. European Control Conf. (ECC)*, pages 3123–3128, September 2003. (p. 6)

[7] J. A. S. Jayasinghe and A. M. B. G. D. A. Athauda. Smooth trajectory generation algorithm for an unmanned aerial vehicle (UAV) under dynamic constraints: Using a quadratic bezier curve for collision avoidance. In *Proc. Manufacturing Industrial Engineering Symp. (MIES)*, pages 1–6, October 2016. (p. 2)

[8] William Koch, Renato Mancuso, Richard West, and Azer Bestavros. Reinforcement learning for uav attitude control. (p. 20), (p. 21), (p. 22)

[9] D. Lande, V. Andrushchenko, and I. Balagura. Data science in open-access research online resources. In *Proc. IEEE Second Int. Conf. Data Stream Mining Processing (DSMP)*, pages 17–20, August 2018. (p. 1)

[10] Yuhong Lin. Simulating drones with gazebo in the construct. *Online: http://www.theconstructsim.com/simulating-drones-with-gazebo-in-the-construct/*, 2016. (p. 16)

[11] L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys. Pixhawk: A system for autonomous flight using onboard computer vision. In *Proc. IEEE Int. Conf. Robotics and Automation*, pages 2992–2997, May 2011. (p. 24)

[12] The Three Wise Men. Best drone simulators for first time flyers. *Online: http://www.gizmosnack.com/top-lists/simulators/best-drone-simulators-first-time-flyers/*, 2017. (p. 16)

[13] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. *Simulation, Modeling, and Programming for Autonomous Robots*, January 2012. (p. 19)

[14] Microsoft. Airsim. *Online: https://github.com/Microsoft/AirSim*, 2018. (p. 22)

[15] Microsoft. Airsim releases. *Online: https://github.com/Microsoft/AirSim/releases.*, 2019. (p. 33)

[16] Harrison Moore. Creating assets for the open world demo. *Online: https://www.unrealengine.com/en-US/blog/creating-assets-for-open-world-demo*, 2015. (p. 24)

[17] Chung-Kiak Poh, Chung-How Poh, Mei-Ling Yeh, and Tien-Yin Chou. Conceptual design of a tropical cyclone UAV based on the AR-6 Endeavor aircraft. (p. 1)

[18] PX4. Px4 development documentation. *Online: https://dev.px4.io/en/concept/architecture.html*, 2018. (p. 27), (p. 28), (p. 29)

[19] Sergio Montenegro Qasim Ali. Decentralized Control for Scalable Quadcopter Formations. *International Journal of Aerospace Engineering*, 2016. (p. 7)

[20] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. (p. 21), (p. 23), (p. 24), (p. 32)

[21] Steven Swanson. Trial by flyer: Building quadcopters from scratch in a ten-week capstone course. (p. 6)

[22] A. Tallavajhula and A. Kelly. Construction and validation of a high fidelity simulator for a planar range sensor. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, pages 6261–6266, May 2015. (p. 14)

[23] Kimon P. Valavanis and George J. Vachtsevanos. Future of Unmanned Aviation. In *Handbook of Unmanned Aerial Vehicles*. Springer, 2015. (p. 1)

[24] Todd VanGilder. What is hardware-in-the-loop (hil) testing? *Online: https://www.winemantech.com/blog/what-is-hardware-in-the-loop-hil-testing*, 2018. (p. 18)