Implementing the Equivariant Visual Inertial Odometry Algorithm onto ArduPilot, an Open-Source Drone Autopilot

A thesis submitted in part fulfilment of the degree of
Bachelor of Engineering (Honours)

by
Liam Fisher
U6670095

Supervisor: Yixiao Ge
CoSupervisor: Prof. Robert Mahony
Examiner: Dr Pieter van Goor

College of Engineering and Computer Science
The Australian National University

November 2022

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of the author's knowledge, it contains no material previously published or written by another person, except where due reference is made in the text.

Liam Fisher
13 November 2022

Acknowledgements

I would like to start by thanking my supervisors Mr Yixiao Ge and Prof. Robert Mahony for their guidance and support throughout this project. I have learned a huge amount while completing this project so thank you for giving me the opportunity. I am also very grateful to Dr Pieter van Goor and Dr Andrew Tridgell for their continued technical support throughout the project. I would also like to thank all the members of the STR group for their input and help.

Finally a special thanks goes to my parents and sister for all their encouragement and support which made this all possible.

Abstract

Visual Inertial Odometry (VIO) algorithms form a class of algorithms which use a camera and Inertial Measurement Unit (IMU) to estimate the trajectory of a vehicle. This thesis concerns the real time implementation of the Equivariant Visual Inertial Odometry (EqVIO) algorithm, with specific emphasis on an ArduPilot quadcopter implementation. The algorithm was successfully implemented onto a Raspberry Pi allowing for an accurate estimation of the state to be generated in real time. Additional work investigated if vehicle simulators such as AirSim provided sufficiently accurate data to allow for the testing of VIO algorithms. While the EqVIO achieved less accurate results using simulated data, predictions of real world behaviour were able to be made. Future work is required to finalise the EqVIO implementation with ArduPilot and the inclusion of features such as improved initialisation and the addition of sensors such as barometers is recommended.

Contents

List of Figures

List of Tables

Nomenclature

API	Application Programming Interface
CPU	Central Processing Unit
EKF	Extended Kalman Filter
ENU	East North Up
EqF	Equivariant Filter
EqVIO	Equivariant Visual Inertial Odometry
FIFO	First in First out
GCS	Ground Control Station
GPS	Global Positioning System
GPU	Graphics Processing Unit
IEKF	Invariant Extended Kalman Filter
IMU	Inertial Measurement Unit
IP	Internet Protocol
KF	Kalman Filter
MEKF	Multiplicative Extended Kalman Filter
NED	North East Down
OKVIS	Open Keyframe-based Visual-Inertial SLAM
RAM	Random Access Memory
RMSE	Root Mean Square Error
SITL	Software in the Loop
SLAM	Simultaneous Localization and Mapping

SSD . . . . . Solid State Drive

ToF . . . . . Time-of-Flight

UDP . . . . . User Datagram Protocol

UKF . . . . . Unscented Kalman Filter

VIO . . . . . Visual Inertial Odometry

VO . . . . . Visual Odometry

VRAM . . . . . Video RAM

Introduction

For a robot to interact and navigate in an environment, its position and orientation, or pose within space must be known. Odometry algorithms form a general family of algorithms that try to identify the pose of a robot. While many classes of algorithms exist that use a variety of different sensors, in recent years Visual Odometry (VO), Visual Inertial Odometry (VIO) and visual Simultaneous Localization and Mapping (SLAM) have become increasingly popular. These algorithms use either a camera or a camera and an Inertial Measurement Unit (IMU) to track the pose of a robot.

These algorithms have a significant number of applications as they do not require the use of Global Positioning System (GPS). This makes them ideal for working in environments where GPS is unreliable or unavailable, such as around buildings or underground. For example, in large cities drones could be used to transport medicines or other time essential supplies from one hospital to another significantly faster than an ambulance. As hospital workers want the drones to land as close as possible to the doors of the hospital, GPS cannot be used to automatically land a drone as the close proximity of the building severally reduces GPS accuracy. The implementation of a VIO algorithm could enable this application as it would allow for this landing procedure to occur automatically even when GPS signals are unavailable.

This project will aim to continue the implementation of a new type of VIO algorithm called the Equivariant Visual Inertial Odometry (EqVIO) algorithm. The theory and initial implementation for the EqVIO were developed by Dr Pieter van Goor [1] while further work has been completed by Mr Yixiao Ge. This project will expand on the previous work with the aim of implementing the EqVIO onto a Raspberry Pi which will sit onboard a drone controlled by an ArduPilot flight controller. This will allow for the EqVIO to provide an estimate of the state of the system to the ArduPilot flight controller with minimal latency, allowing for accurate navigation without GPS.

ArduPilot is a commonly used autopilot for both research and industrial applications. This is due to it being open source and having an advanced feature set allowing for each user to customise their individual implementation as required. Additionally, ArduPilot has an extremely active community so by integrating the EqVIO with ArduPilot it is made available to a much larger user base.

1.1 Research objectives

The primary research question of this project is:

> *Can the EqVIO provide a robust estimate of the pose of the robot in real time on vehicles with low computational power?*

Secondary research questions were also created to provide stepping stones within the project. These included:

- Is using simulations to generate VIO training data a viable alternative to real world data collection?
- How is the EqVIOs accuracy effected by different movements of a drone?
- What level of accuracy can the EqVIO provide?

1.2 Methodology

Several steps were completed prior to attempting the real time EqVIO implementation. To gain familiarity with the EqVIO, data was initially generated through simulation in AirSim. Simulation was also used to investigate the EqVIOs performance due to certain movements and situations. By using simulated data in these early stages, experiments could be conducted rapidly with no risks associated with crashing the drone or time delays due to poor weather. Analysis of the results from AirSim also suggested that improvements could be made to the EqVIO algorithm by adding an attitude innovation.

Real world testing was completed on the drone shown in figure 1.1. This drone was equipped with a RTK GPS that provided centimeter accurate ground truth data, a 720p global shutter gray scale camera and a Cube Orange flight controller running ArduPilot. The main computer onboard the drone which ran the EqVIO was a Raspberry Pi. The Raspberry Pi was used as it is a widely accessible computer with low cost and low weight making it ideal for this application.

Prior to any experiments being conducted, a camera mount was designed and 3D printed that angled the camera at a 60 degree downwards angle. This change in camera position necessitated re-calibration of the camera parameters which was done using Kalibr. A real time version of the EqVIO called the APVIO was then implemented and tested onboard the drone. Testing of the APVIO was initially completed by walking around with the drone held at chest height. This allowed for multiple tests to be conducted very quickly allowing for rapid debugging. Final validation of the APVIO was conducted by flying the drone at the Australian National University's (ANU) Spring Valley farm.

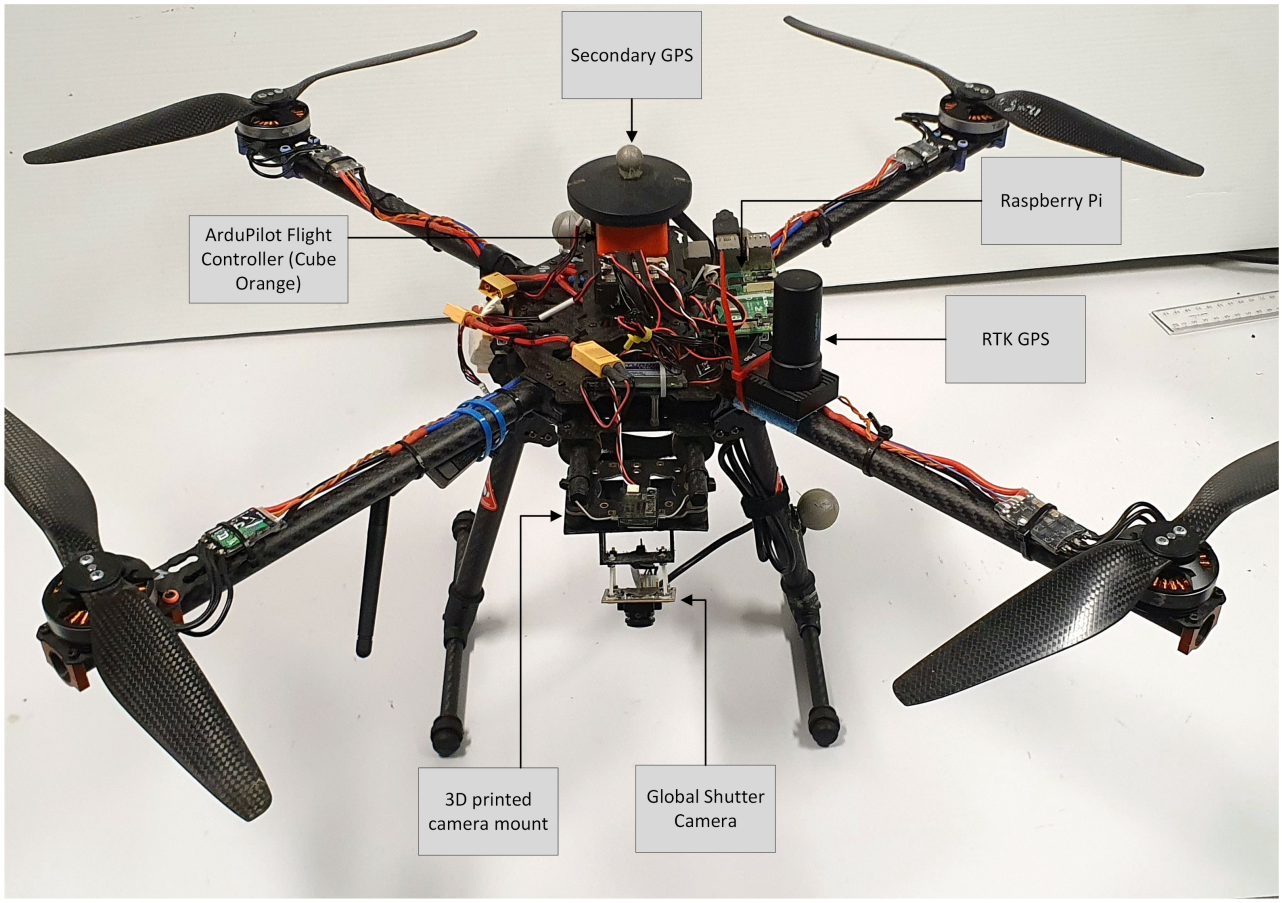


Figure 1.1: Drone used for real world testing

Literature Review

VIO and visual SLAM algorithms currently form a very active area of research as both cameras and IMUs are relative cheap sensors. This allows for these types of systems to have wider spread use as the cost of producing them is much lower compared to systems containing expensive sensors such as Lidar. Current uses of VIO and visual SLAM include augmented reality and self driving cars [2]. The use case of VIO that was focused on within this project was allowing drones to navigate when no GPS is available. A well publicised example of this use case is in NASAs Ingenuity helicopter [3][4] which made the first flight of a man made vehicle on Mars. A VIO algorithm is required as there are no GPS satellites to access around Mars and the delay for human intervention within a flight is much to large. The particular VIO algorithm used was a Extended Kalman Filter (EKF) based algorithm called MAVeN [5]. A particular emphasis was placed on the robustness of this algorithm rather than absolute accuracy as the position of the helicopter could be corrected after each flight by mission control on earth. The drone was equipped with a 640×480 gray scale camera, an IMU and a laser range finder which measured the distance to the ground.

2.1 Additional sensors used in VIO

From VO algorithms we know that a camera can be used to track the pose of a vehicle up to some scale factor. Adding an IMU allows for this scale factor to be calculated and provides additional information helping to stabilise the system. For this reason all VIO algorithms contain at least a monocular camera and an IMU however other sensors can be added to improve the accuracy and reliability of the system.

RGBD cameras have an extra pixel value which gives the distance to the object in that pixel. These make them an attractive addition as many VIOs use a parameterisation of landmarks given by a bearing and a distance to the landmark, the latter of which RGBD camera can directly provide measurements of. A common method of implementing this is through stereo cameras which is a system of two cameras with some baseline separating them. By then considering the pixel shift between a feature present in both cameras the distance to the object can be calculated. There have been a number of VIO algorithms developed to use stereo cameras in recent years [6][7][8] however stereo

cameras do have disadvantages. When compared to a monocular camera they are more expensive as you require an additional camera sensor and significantly more calibration. Stereo cameras are also only effective at close ranges as at large distances the pixel shift between the two cameras is below the noise threshold.

A sensor used as part on the NASA Ingenuity helicopter was a laser range finder [3]. This sensor falls under the broader category of Time-of-Flight (ToF) sensors which measure distance to a target based on how long it takes some emitted signal to rebound and return to the sensor. Onboard the Ingenuity helicopter the ToF sensor was used to measure the distance to the ground, however ToF sensors could be pointed in some other direction such as parallel to the camera giving the distance to some point within the camera's image.

Drones often carry additional sensors such as barometer and magnetometer which are used in navigation. While directly measuring the height above the ground using a barometer is difficult, differential pressure has been integrated into some VIOs [9] to improve the estimate of the height above the ground. Magnetometers have also been integrated into VIOs to provide a global attitude reference [10] which in some cases has improved performance.

2.2 Existing VIO algorithms

There are two major approaches to implementing VIOs. These are filter based VIOs and optimisation based VIOs. The major difference between these methods is that for a filter based VIO, past poses and information about the system are marginalised out into a probability distribution [11]. In contrast, optimisation based methods store a small number of keyframes which are used to generate a cost function dependent of the pose of the system over time. Minimising this cost function then allows for a trajectory with minimised error to be found. While in theory optimisation based methods give the best performance per unit of computational time [11] they are in general more computationally intensive [12]. Several popular VIOs are discussed in this section to give further context to the current state of the VIO problem.

Open Keyframe-based Visual-Inertial SLAM (OKVIS) uses a non-linear optimisation approach. By generating an IMU error term and a landmark reprojection error, a non-linear cost function can be created. Optimising this non-linear cost function results in the state error being minimised [13]. To achieve real time operation, keyframes are used to ensure that the optimisation window remains bounded. A keyframe method marginalises out many of the previous frames and only keeps a few crucial frames based on a given criteria [11] that contain the majority of the information. This method reduces the computational complexity of the problem while attempting to retain the majority of the prior information about the state.

VINS-Mono is probably the highest performance VIO algorithm to date [12]. It uses a

nonlinear optimisation approach but incorporates additional features such as loop closure which make it significantly more accurate. VINS-Mono starts by completing an initialisation procedure that ensures that a accurate initial frame of reference is selected [14]. The optimisation method then accounts for additional variables such as camera-IMU extrinsic calibration and IMU bias correction that can be difficult to measure. While VINS-Mono can be run without loop closure, significantly better results are obtained while using it. The disadvantage of using loop closure is that it is more computationally intensive and requires more memory [12].

ROVIO is a high performance EKF based VIO algorithm. It uses a robocentric frame of reference where the location of all landmarks are calculated with respect to the current location of the camera [15]. Additionally, landmarks are parameterised through a bearing vector and a distance parameter rather then through Cartesian coordinates. Using this parameterisation Bloesch *et al.* claim that no initialisation procedure is required [15].

2.3 Filtering methods

A standard method for estimating the state of a non-linear dynamical system is using the EKF. This is a two-step algorithm where the dynamics of a system are first linearised and then the Kalman Filter (KF) is applied to the linearised system [16]. The EKF has been extremely common since its development in the 1960s and has been successfully used by major organisations such as NASA on their Gemini and Apollo programs [5]. The disadvantage of using the EKF is that it can be difficult to implement and tune and can become unstable when the linearisation error becomes too large. Additionally, the EKF requires the computation of a Jacobian matrix which can be difficult to derive and in some cases may not exist [16].

The defining characteristics of EKFs are that they represent the state of the system using Gaussian distributions. A Gaussian distribution can be described with two variables, the mean of the distribution μ and the standard deviation σ. For multi variable systems of k variables we can then represent the system using a vector $\boldsymbol{\mu} \in \mathbb{R}^k$ to represent the mean of each variable and $\Sigma \in \mathbb{R}^{k \times k}$ which is called the covariance matrix.

Many different algorithms have been proposed to improve the performance of the EKF. The Multiplicative Extended Kalman Filter (MEKF) was proposed to avoid the singularities and discontinuities that arise when using a three-parameter representation of rotation [17]. By using a four-parameter quaternion representation the discontinuities can be avoided. The Unscented Kalman Filter (UKF) [18] used a quaternion error approach to ensure that the quaternions within the filter remained normalised. Other modern approaches such as the Invariant Extended Kalman Filter (IEKF) have tried to account for geometric properties to improve the overall performance of the system [19].

The Equivariant Filter (EqF) [20] takes advantage of similar geometric properties to the

IEKF but extends it to a more general theory by using the fact that many robotics problems are posed on a homogeneous space. Van Goor *et al.* proved that such a system can be extended to a larger system which is equivariant under a symmetry action resulting in good convergent properties.

2.4 EqVIO theory

2.4.1 Formalisation of common groups

$\mathbb{R}^N$ is a N dimensional real space where an element $x \in \mathbb{R}^N$ is represented by some vector $x = (x_1, x_2, ..., x_N)$. The group $\mathbf{SO}(3)$ is called the 3D rotation group. An element of this group will rotate an element of $\mathbb{R}^3$ about the origin. Members of this group are normally represented as a 3×3 matrix as shown below.

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Members of $\mathbf{SO}(3)$ have several properties. For some $R \in \mathbf{SO}(3)$ we can find that $R^{-1} = R^T \implies R^T R = RR^T = I$. Additionally for $R_1, R_2 \in \mathbf{SO}(3)$ we find that $R_3 = R_1 R_2 \in \mathbf{SO}(3)$.

The scaled orthogonal group $\mathbf{SOT}(3)$ is an equivalent representation of a point $p \in \mathbb{R}^3$ [1]. Instead of representing it as a point in Cartesian coordinates, it is represented as a unit vector rotated about the origin by R and then scaled to some point by some $c > 0$. It is defined as:

$$\mathbf{SOT}(3) := \{ \begin{pmatrix} R & 0 \\ 0 & c \end{pmatrix} \Big| R \in \mathbf{SO}(3), c \in \mathbb{R}_{>0} \}$$

Another common group is the special euclidean group $\mathbf{SE}(3)$. This group is often used to either represent pose or to move between two different frames of reference in 3D space. This is because it contains both a translation and a rotation element allowing it to represent a position and an orientation. $\mathbf{SE}(3)$ is defined as:

$$\mathbf{SE}(3) := \{ \begin{pmatrix} R & x \\ 0 & 1 \end{pmatrix} \Big| R \in \mathbf{SO}(3), x \in \mathbb{R}^3 \}$$

The special euclidean group $\mathbf{SE}(3)$ can be extended to allow for modelling of pose and a velocity in one group. This is called the extended special euclidean group $\mathbf{SE}_2(3)$ and is defined as:

$$\mathbf{SE}_2(3) := \{ \begin{pmatrix} R & x & v \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Big| R \in \mathbf{SO}(3), x, v \in \mathbb{R}^3 \}$$

2.4.2 Problem definition

The parameters we are attempting to model in VIO are shown in figure 2.1. We define an initial reference frame $\{0\}$, a body fixed frame $\{B\}$ and a camera frame $\{C\}$ [1].

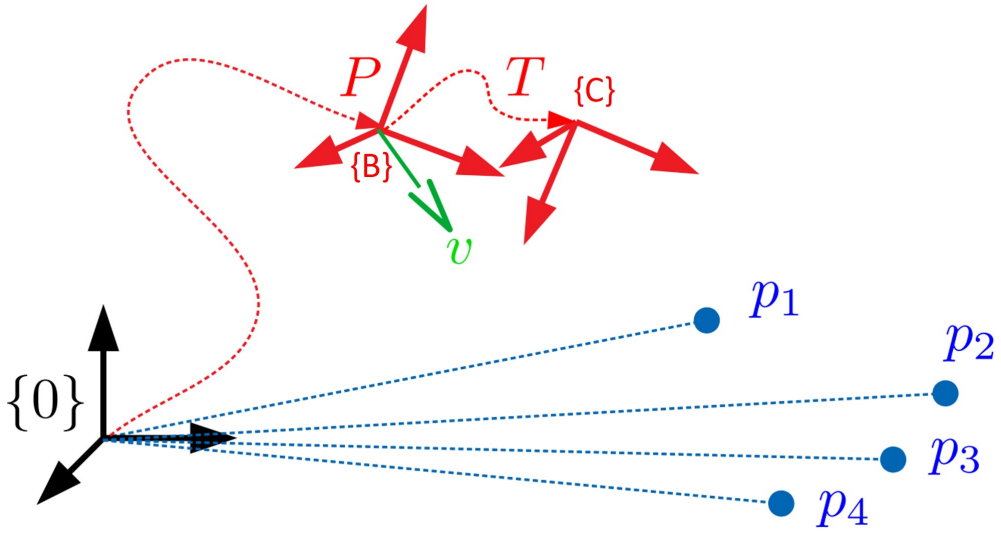


Figure 2.1: Variables estimated for VIO problem [1]

We define the variables:

- $P \in \mathbf{SE}(3)$ - As the pose of the IMU in $\{B\}$ with respect to the reference frame $\{0\}$
- $v \in \mathbb{R}^3$ - As the linear velocity of the IMU in $\{0\}$
- $b \in \mathbb{R}^6$ - As the bias of each axis of the accelerometer and gyroscope
- $p_i \in \mathbb{R}^{3N}$ - As the position of each landmark i in $\{0\}$
- $T \in \mathbf{SE}(3)$ - As the pose of the camera $\{C\}$ with respect to $\{B\}$

2.4.3 EqVIO structure

The structure of the EqVIO can be split into the four stages as discussed below [1].

- Feature Tracking: In this stage GIFT [21] is used to identify features and track them through multiple images. A minimum number of features is set by the user and if the number of tracked features falls below this number, new features will be identified.
- Preprocessing: Features that were previously tracked but have left the frame of view as well as features identified as outliers are removed from the state. New features identified are added to the state.
- Propagation: IMU messages are added to an internal queue to wait for the update step
- Update: This stages occurs when an image has finished feature tracking and pre-processing. First, all IMU messages with time stamps earlier then the current

image time stamp are used to propagate the system forwards in time. The features identified from the image are then used to correct the state of the system.

2.4.4 Differences between EqVIO and general EKF based VIO

Using the EqF rather then an EKF gives the EqVIO several advantages over other filter based VIOs. This stems from the symmetries provided by the EqF. An example of this is that as the EqF considers the state to be part of a lie group rather than in the state space [20] better IMU dynamics are achieved as no linearisation error is present for bias free IMU propagation.

Additional improvements are made by considering the parameterise used for the landmarks. While in the problem definition landmarks are parameterised through $\mathbb{R}^3$ an equivalent parameterisation is through $\mathbf{SOT}(3)$. This can be shown to have much lower output linearisation error due to equivariance in the group action [1].

Van Goor *et al.* [1] additionally define a group which our observer will sit on called the VI-SLAM group. This group accounts for the symmetries fundamental to VIO allowing for us to take advantage of the low IMU linearisation error and better landmark parameterisation. It is defined as:

$$\mathbf{SLAM}_n^{VI}(3) := \mathbf{SE}_2(3) \times \mathbf{SOT}(3)^n$$

This accounts for the n landmarks as well as the position, attitude and velocity of the vehicle. The IMU bias and and camera extrinsics are often desirable to estimate so van Goor *et al.* [1] defined the extended direct product group $\mathbf{G}$ which the observer for the EqVIO will sit on.

$$\mathbf{G} := \mathbf{SLAM}_n^{VI}(3) \times \mathbb{R}^6 \times \mathbf{SE}(3)$$

Sensor Calibration

While modern procedures allow for highly repeatable manufacturing of sensors, calibration is still required to understand the specific internal behaviour of individual sensors. Without calibration the exact behaviour of a sensor can only be known from a data sheet up to some margin of error. Additionally data sheets may not provide calibration data in the desired metric so in all cases sensors should be calibrated prior to use within VIO.

3.1 Identifying noise within IMUs

Several metrics exist for quantifying noise within an IMU. For the EqVIO noise is quantified through a rapidly fluctuating random walk and a more slowly changing bias. The random walk is responsible for accounting for high frequency noise and is modelled as a white noise process [22]. The bias represents a constant offset between the true and measured values and the bias stability is modelled as a Brownian motion process [22].

These values can be measured through the Allan Variance (Deviation) which is a commonly used method for IMUs [23]. This process works by taking a set of N successive data points and dividing it into K different clusters each of length n giving the relationship $K = N/n$. The mean of each cluster $\overline{a}_k$ for $k = 1, 2, ..., K$ is then calculated with the Allan variance being defined as [24]:

$$\sigma_A^2(\tau) = \frac{1}{2(K-1)} \sum_{k=1}^{K-1} (\overline{a}_{k+1} - \overline{a}_k)^2$$

Where $\tau = n/f$ is the averaging time for sampling frequency f. Plotting $\sigma_A^2(\tau)$ onto a log-log graph allows for the identification of the random walk and bias when particular gradients occur within the graph. This is described in figure 3.1.

In general the random walk noise is measured at $\tau = 1s$ while the bias stability is measured as the minimal value of the Allan variance. An existing implementation of the above process was used [25] to generate Allan variance plots for the IMU used onboard the drone. To record the necessary data the drone was placed on level ground and IMU data was recorded for approximately one hour. The plots generated are shown in figure 3.2.

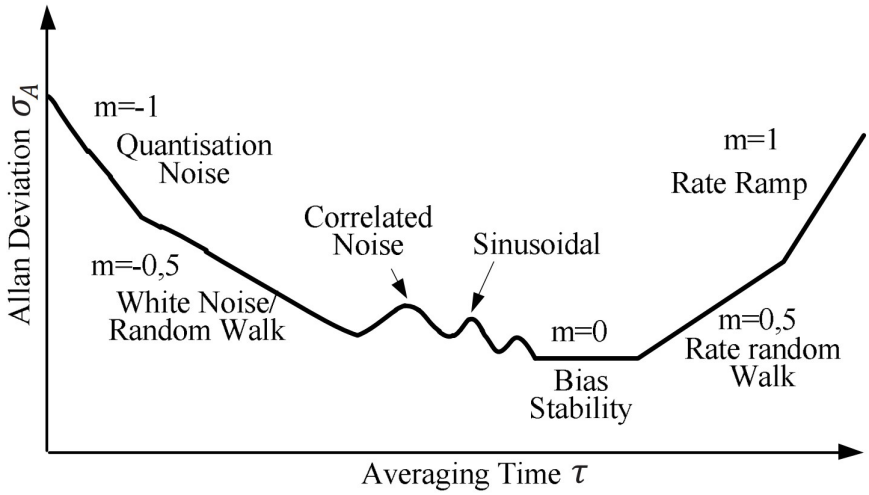


Figure 3.1: Typical graph of Allan variance for IMU [24]

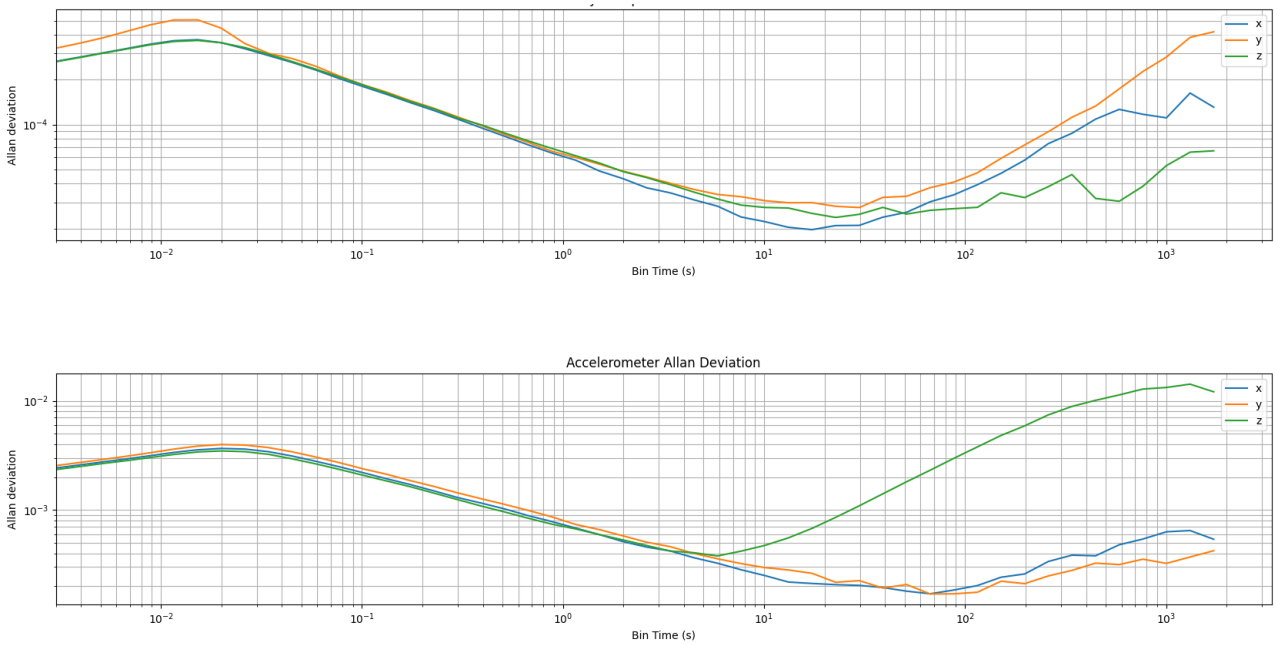


Figure 3.2: Allan variance measured on drone IMU

From these graphs we can find that the accelerometer has random walk noise of 7.8×10^{-4} $m/s/\sqrt{s}$ and bias instability of 2.2×10^{-4} $m/s/s$. We can also determine for the gyroscope that the random walk noise is 6.5×10^{-5} $rad/\sqrt{s}$ and the bias instability is 1.8×10^{-5} rad/s.

3.2 Camera calibration using Kalibr

Camera calibration is an important step for obtaining accurate results within VIO. Without calibration we would not know how to track features across multiple images as we would not understand the internal behaviour of the camera. Additionally, without calibration effects like fish-eye lenses can distort the image making features move in unexpected ways. Cameras should be re-calibrated any time their mounting position is moved or if

the resolution of the camera is changed. As a new camera mount was 3D printed for the drone to improve the position of the camera and the resolution was down scaled from 720p to 640×420, re-calibration of the camera was required.

The intrinsics of a camera relate where a feature is located in 3D to where it appears on the 2D image plane. The pinhole model is very common model for the intrinsics of a camera and can be defined by the following matrix.

$$K = \begin{bmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where f_x and f_y represent the focal length in the x and y direction and p_x and p_y are the coordinates of the centre of the image plane.

The extrinsics of a camera relate the pose of a camera to the body fixed frame of the drone. For VIO we generally treat the IMU as the origin of the body fixed frame meaning that the extrinsics relate the pose of the camera with respect to the IMU. The extrinsics matrix is given by $P \in \mathbf{SE}(3)$.

Distortion models help to remove the effects of fish-eye lenses. An equidistant distortion model was used which given image coordinates $[a, b]$ uses the coefficients $[k_1, k_2, k_3, k_4]$ to transform to the distorted coordinates $[x', y']$ by the given operation [26][27].

$$r^2 = a^2 + b^2$$

$$\theta = \arctan r$$

Create fish-eye distortion:

$$\theta_d = \theta(1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8)$$

Calculate distorted coefficients:

$$x' = (\theta_d/r)a$$

$$y' = (\theta_d/r)b$$

To identify the intrinsics, extrinsics and distortion parameters a tool called Kalibr [28] is used. Kalibr uses a data set containing IMU measurements and camera images from a camera moving around a calibration target to identify optimal values for the parameters. The calibration target used is called an Aprilgrid [29] and an example is shown in figure 3.3.

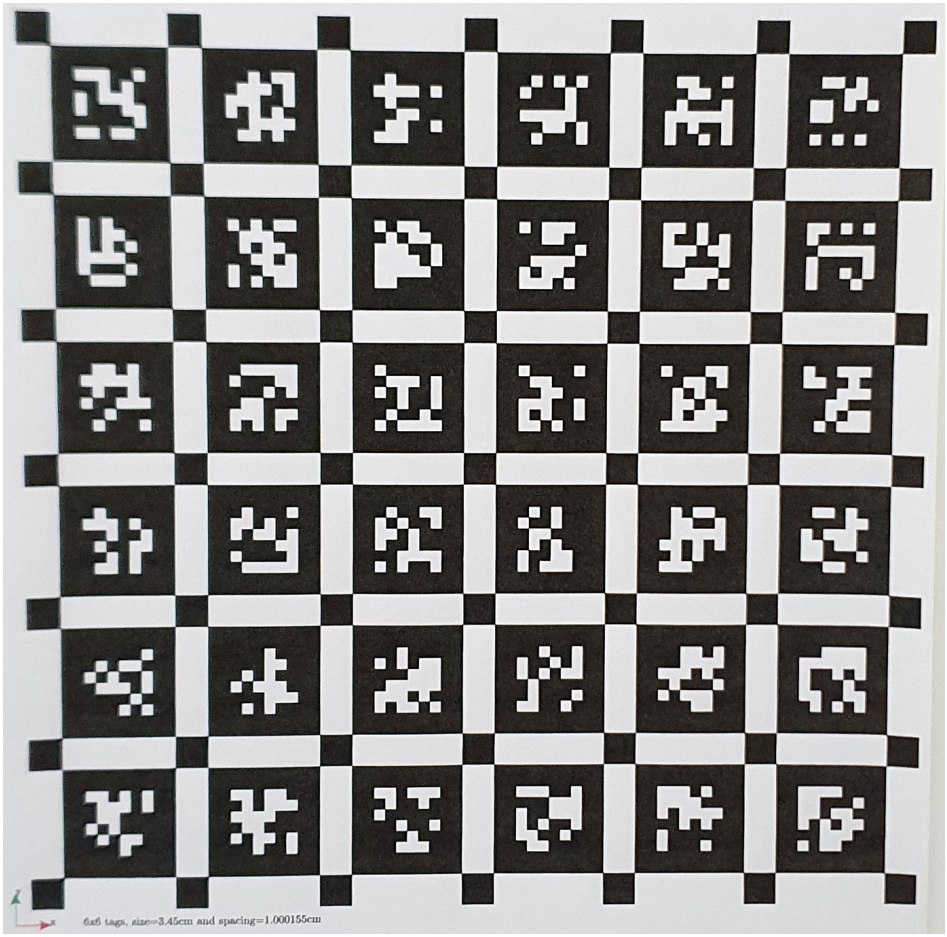


Figure 3.3: Example of 6x6 Aprilgrid calibration target

Data was recorded using the same script used to record offline VIO data sets[1]. The program is run using the command:

```
$ python3 record_vio.py -i
```

Where the -i is removed if the data set is not being recorded inside. To convert the data set to a rosbag which Kalibr can consume, use the command:

```
$ python mav2ROS.py mav_imu.csv  frames/  cam.csv
```

When recording the data set, all axis of the IMU should be stimulated while keeping the calibration target in the view of the camera. Additionally ensure that the edges and corners of the camera see the calibration target so that they can be undistorted. To calculate our parameters we first add Kalibr to the path via:

```
$ source ~/kalibr_workspace/devel/setup.bash
```

The camera intrinsics and distortion coefficients can then be obtained through the command:

```
$ kalibr_calibrate_cameras --bag mav_vio.bag --topics /cam --models
↪  pinhole-equi --target April_6x6.yaml
```

[1] https://github.com/liam327/u6670095_thesis

The extrinsics are found using the command:

```
$ kalibr_calibrate_imu_camera --bag mav_vio.bag --cam
↪  camchain-mav_vio.yaml --imu imu_param.yaml --target April_6x6.yaml
```

Where the file "*imu_param.yaml*" is related to the internal noise characteristics of the IMU. Using this method the camera intrinsics, extrinsics and distortion parameters were identified for our gray scale camera running at 640×420. The results for the intrinsics, extrinsics and distortion parameters are shown below:

$$K = \begin{bmatrix} 373.99 & 0 & 421.04 \\ 0 & 372.91 & 250.59 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} -0.053 & -0.839 & 0.542 & 0.097 \\ 0.998 & -0.067 & -0.007 & 0.0038 \\ 0.042 & 0.54 & 0.841 & 0.19 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\theta_d = \begin{bmatrix} -0.021 & -0.0995 & 0.153 & -0.083 \end{bmatrix}$$

Kalibr also provides graphs showing the reprojection error of the camera after calibration which is useful for determining the quality of a calibration. An example of the reprojection error for the calibration parameters above is shown in figure 3.4. We can see that we are getting approximately 2 pixels worth of reprojection error after calibration. While this amount of error is higher than desirable and could be reduced further, acceptable results were obtained from the EqVIO using this calibration.

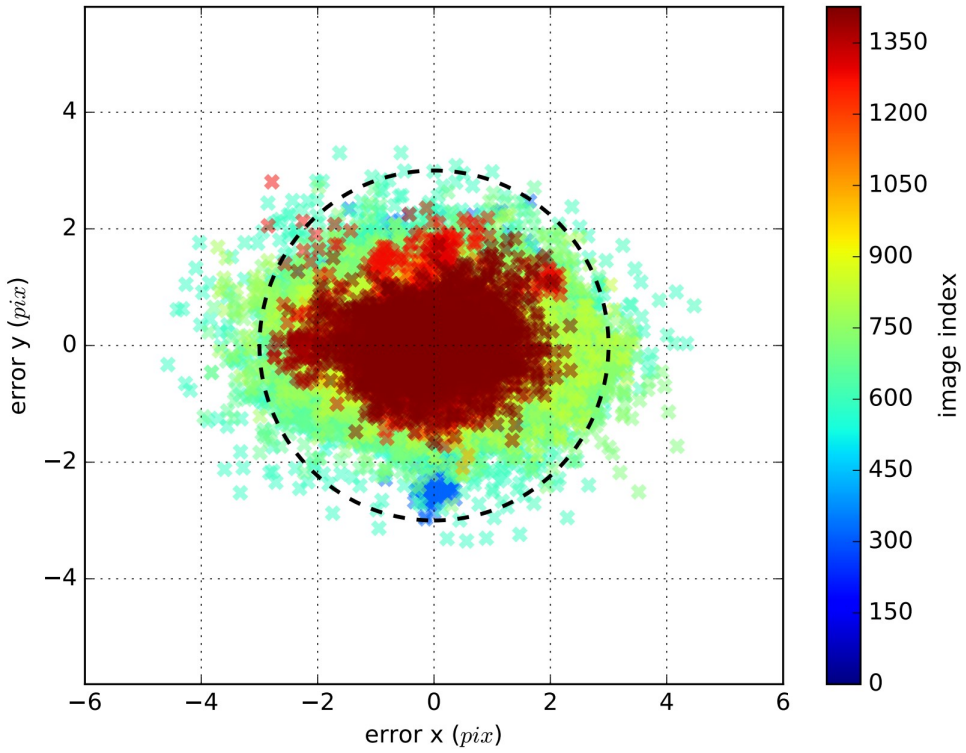


Figure 3.4: Reprojection error of camera after calibration

Simulation Using AirSim

AirSim is a powerful drone and vehicle simulator developed by Microsoft which runs within the Unreal Engine environment. Its main use case is "as a platform for AI research to experiment with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles" [30]. AirSim is able to be connected to a number of different popular flight controllers such as ArduPilot and PX4. Developers can also use the Application Programming Interface (API) to control the vehicle and collect data if a flight controller is not being used. Due to AirSim using Unreal Engine there are also a large number of different environments which can be downloaded from the Unreal Engine store. This can allow for a VIO to be trained in areas that would not be possible to access in the real world such as large cities or remote regions.

AirSim was used for several different purposes within this work. Initially AirSim was used to determine if simulation tools provide sufficiently accurate data to allow for VIOs to operate within them. AirSim was also used to explore both the effect of certain movements on the accuracy of the EqVIO and as a testing platform while developing further features for the EqVIO.

4.1 Setting up AirSim

4.1.1 AirSim hardware requirements

AirSim requires reasonably powerful computer hardware to run. The developers recommend that for more complex environments that a Graphics Processing Unit (GPU) with 8GB of Video RAM (VRAM) is used [31]. Additionally it is recommended that an Solid State Drive (SSD) is used rather then a hard drive [32][33] as a hard drive can become a bottle neck due to its lower read and write speeds. Finally a decently powerful Central Processing Unit (CPU) is required and it is desirable to have at minimum 16GB of Random Access Memory (RAM) but more ideally 32GB as high amounts of memory usage were observed. The setup which was used throughout this project is listed below.

- Operating system: Windows 10
- CPU: Ryzen 5 5600X

- GPU: Nvidia RTX 3070
- RAM: 32GB
- Storage: PCIe 3.0 NVMe SSD

4.1.2 Building AirSim in Unreal Engine

There are two major ways to run AirSim. The first is through pre-compiled binary files which allow for AirSim to be run on any computer without the installation of additional software. This means that users only need to download and run the files to get started, significantly reducing initial setup time. There are a number of disadvantages to using the binaries including an inability to modify the pre-generated environment, an inability to modify the source code of AirSim and increased difficulty changing the location of the vehicle within the environment. All of these mean that there is less functionality and flexibility then if you build the project yourself.

The second way to run AirSim is through building from a Unreal Engine project. This has a number of extra requirements over running AirSim from a binary but it is significantly more flexibility in how AirSim can be used. One major benefit of building AirSim from the Unreal Engine environment is that it allows for modifications to the environment. This means unwanted obstacles can be removed or the starting location of the vehicle can be changed very easily. This cannot be done when using the AirSim binaries. Additionally it is sometimes useful to make small changes to the base code of AirSim which can only be done if AirSim is built in Unreal Engine. Finally while a small number of precompiled binaries exist there are a huge number of other environments in the unreal engine store which are only accessible if AirSim is built through the Unreal Engine. Due to this added flexibility AirSim was built through Unreal Engine for this project.

4.2 Flight controllers in AirSim

There are a number of advantages of using a flight controller over a joystick to fly a vehicle in AirSim. The most significant advantage is that it allows for pre-programmed flight routes to be created and saved. These routes can then be flown with high repeatability allowing for accurate comparisons between flights on the same route with different settings. Additionally by using a flight controller you remove much of the difficulty associated with flying a drone using joysticks.

There are several popular flight controllers which have support with AirSim but for this project ArduPilot will be used. This is because it is the flight controller with the widest amount of support for AirSim. There are many different Ground Control Station (GCS) which support ArduPilot but the one with the best compatibility with AirSim is MAVProxy. For this project AirSim is run on a high power desktop computer while MAVProxy is run on a Linux laptop with the two computers connected via ethernet.

4.2.1 Software in the loop

An autopilot system such as ArduPilot normally has two computers which are connected when operating in the real world. One is a flight controller which runs ArduPilot and sits onboard the vehicle. This collects sensor data as well and controls the motors of the vehicle. The other computer is the GCS which serves the interface between the drone and the operator. The GCS allows for the user to command the vehicle as well as displaying the position and sensor values of the vehicle. For a real world drone these systems communicate over radio.

Software in the Loop (SITL) simulation maintains a similar structure of having two computers, except instead of having a flight controller onboard the vehicle we instead have a flight simulator. This flight simulator contains a physics engine to emulate what would actually be happening in the real world as well as a simulated drone that provides all the messages we would normally expect from the flight controller. This structure is described below in figure 4.1. When using SITL, data is transmitted over the User Datagram Protocol (UDP) which is one particular protocol for transmitting data via the internet.

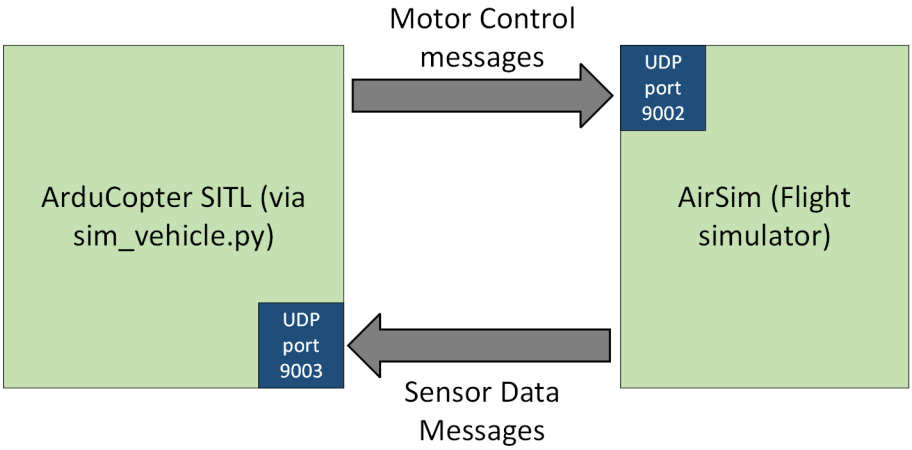


Figure 4.1: SITL architecture for connecting AirSim to ArduPilot

4.2.2 Connecting ArduPilot to AirSim

AirSim is one of many flight simulator but differentiates itself with full 3D world and high fidelity graphics. To allow for AirSim to connect to ArduPilot there are a number of settings which must be adjusted. The majority of these settings are contained within AirSims "*settings.json*" file. This file is extremely important and there are a number of different settings which can and should be modified prior to starting a simulation. An example of a settings file containing many of the common settings modified throughout this project is shown in Appendix A in listing 8.1. The particular settings which are most important for connecting AirSim and ArduPilot are the UdpPort, ControlPort, UdpIp and LocalHostIp.

With two separate computers communicating via the internet each system needs the Internet Protocol (IP) address of the other system to know where to send data packets. The UdpIp port tells AirSim the address of the ArduPilot machine which it needs to send sensor data to [34]. The LocalHostIp is then set to be the IP address of the computer running AirSim. To identify these IP addresses use the commands *ipconfig* or *ifconfig* in either Windows or Linux command line respectively.

There are many different devices which interact with the laptop and desktop via the internet so we need to tell AirSim and ArduPilot which particular channel or port to listen to. This is done by setting the values of UdpPort which is the port number which ArduPilot will receive sensor data sent from AirSim on and ControlPort which tells AirSim which port to listen for messages from ArduPilot on. These values can be set to any free port available however example values for these variables are shown in listing 8.1.

To start a simulation with ArduPilot connected to AirSim, first change the *VehicleType* parameter in "*settings.json*" to be "ArduCopter" rather then the default of "SimpleFlight". AirSim should then be run before ArduPilot, however it will hang when the play button is pressed. This is because it is waiting for ArduPilot to connect to AirSim. ArduPilot should be run through MAVProxy using the linux terminal command shown below:

```
$ sim_vehicle.py -v ArduCopter -f airsim-copter --console --map -A
↪  --sim-address = LocalHostIp -l "47.64146769,-122.14016909,122,40"
```

The command *sim_vehicle.py* is used to start a SITL instance through MAVProxy. The other options are described below:

- "*-v ArduCopter*" tells the simulator that you want to use a Copter vehicle rather then a plane or rover.
- "*-f airsim-copter*" tells the simulator that we want to use an AirSim helicopter
- `--console opens the console`
- `--map opens the map`
- "*-A*" tells the simulator to connect to an external device. The –sim-address then tells ArduPilot the IP address to look for the device at.
- "*-l*" is used to set the starting location. This needs to match the default location in AirSim otherwise it can give unpredictable results.

Once the command is run, AirSim will start running and MAVProxy will display the tabs shown during normal flight allowing for the user to control the quadcopter.

One difference that does exist between flying a drone in the real world versus in SITL is the takeoff procedure. Within SITL a specific process needs to be followed. Initially the drone will be in "*stabilize*" mode. Within this mode the command "*arm throttle*" must be run to arm the motors of the drone. The drone should now be put into "*guided*"

mode using the command "*mode guided*". Sometimes this command will fail if run too quickly after start up due to the GPS not having set its origin. Fixing this is as simple as waiting for the message "*AP: EKF3 IMU0 is using GPS*" to be displayed in the console. Once in guided mode the drone can takeoff using the command *takeoff 20* where 20 is the desired height. Once in the air the drone can be commanded and flown around as normal, or if an auto mission is set up the drone can fly that using the command *mode auto*.

4.2.3 EqVIO data storage format

Data is stored in a specific format for the EqVIO. This is done both for organisational purposes and for consistency in how different scripts interact with the data. The basic structure consists of a folder containing 2 files which must exist for every data set which are "*cam.csv*" and "*mav_imu.csv*". "*cam.csv*" stores the timestamp which each image arrives at as well as the frame number that each image corresponds to. "*mav_imu.csv*" stores all the IMU, GPS and ground truth data as well as timestamps where timestamps for all files are stored in units of seconds. The exact format for "*mav_imu.csv*" uses the structure shown in table 4.1 where GPS estimate is ArduPilots GPS estimate.

Time stamp (s)	Gyroscope (rad/s)			Accelerometer			Mav time stamp (s)	Attitude (rad)			Raw GPS			GPS estimate		
T	X	Y	Z	X	Y	Z	T	Roll	Pitch	Yaw	Lat	Lon	Alt	Lat	Lon	Alt

Table 4.1: File structure for IMU csv

An additional file called "*attitude.csv*" can exist if the attitude innovation is being used. In this case the attitude data is stored in the form of a quaternion with order in the file given by: timestamp, W, X, Y and Z. Finally a folder called "*frames*" exists for all data sets and is contained within the top level folder. This subfolder contains all of the images which were recorded through the flight. Images are stored as *.jpg* and should be named "*frame_i*" where *i* is the frame number which corresponds to the same frame in the "*cam.csv*" file.

4.2.4 Recording data through ArduPilot

The major objective of using AirSim is to allow for testing of the EqVIO in a controlled environment where we can run multiple tests very quickly. To do this both sensor data and camera images must be generated, stored and formatted to allow for the EqVIO to run. Data must be recorded from two separate sources within AirSim. IMU data can be obtained from the ArduPilot SITL instance using the MAVLink communication protocol while images must be recorded from AirSim directly. The structure of the data flow is described in fig 4.2.

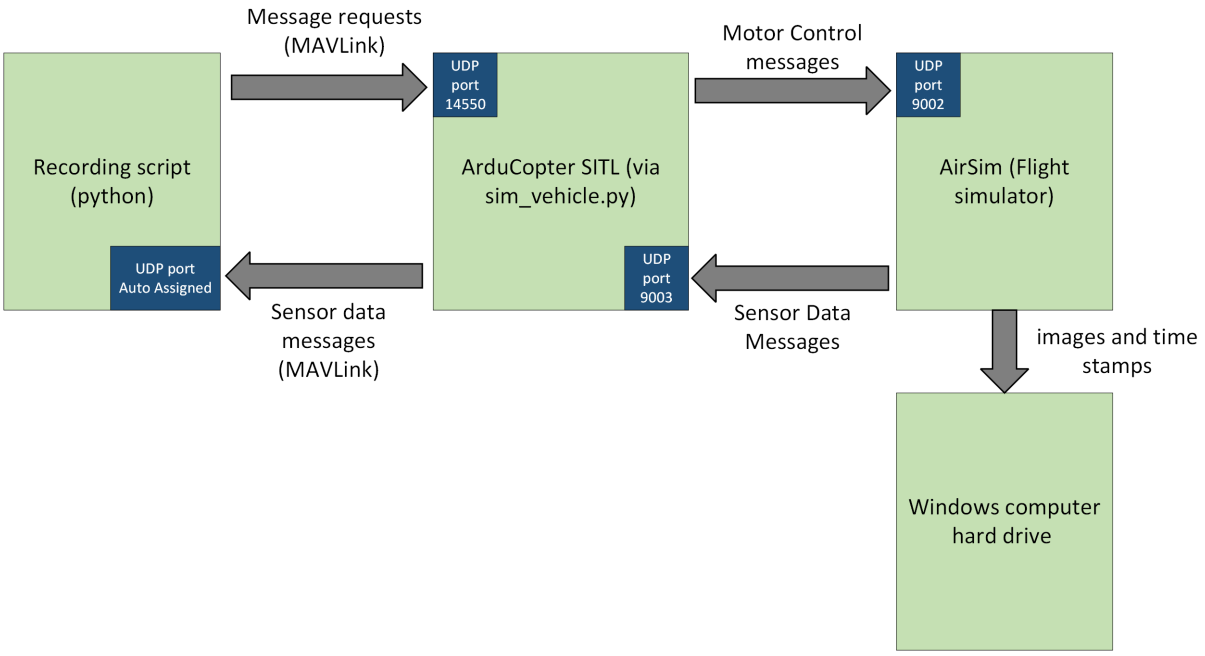


Figure 4.2: SITL architecture for recording AirSim data

The data recording file used to record IMU and GPS data is called "*AirSim_record.py*". This file is nearly identical to that used onboard the drone however the communication protocol used to transfer data from the ArduCopter SITL instance to the recording script is UDP rather then serial. The IMU recording script can otherwise remain unchanged however images should not be recorded through this script. One additional step required when recording AirSim data through ArduPilot is to run the command "*set streamrate -1*" in the ArduPilot SITL command terminal. If this command is not run then data will be recorded at 4Hz rather then at 200Hz.

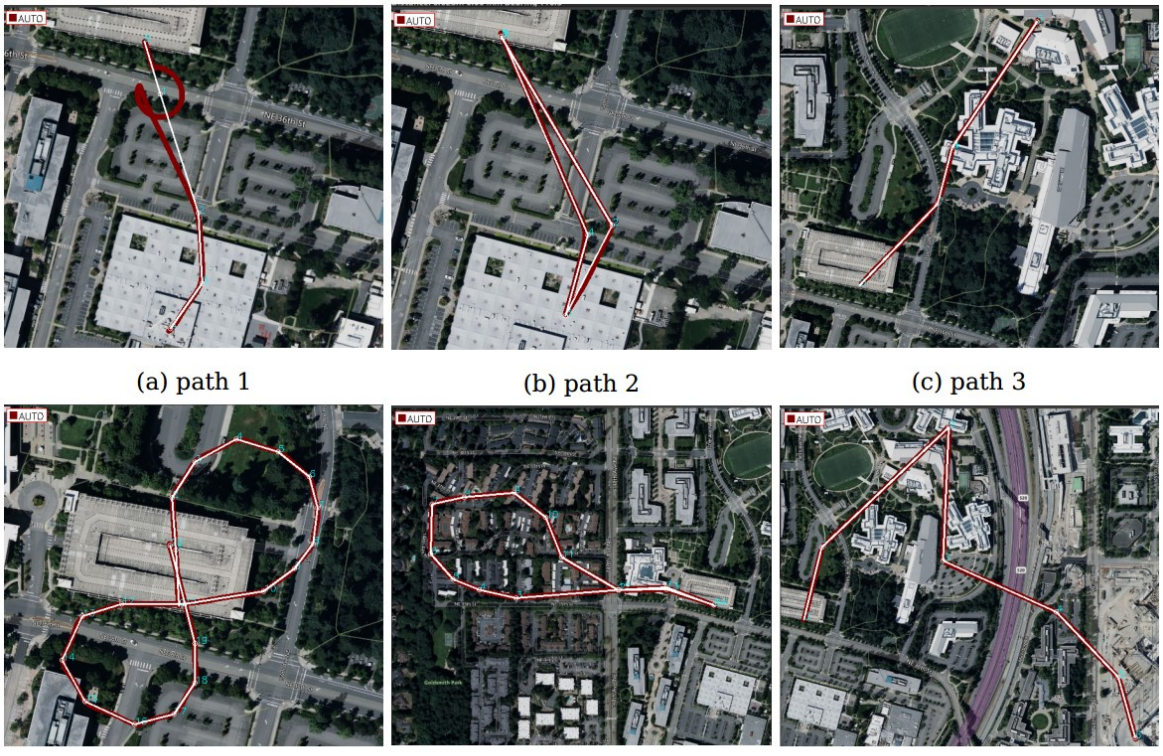
Images are recorded through the inbuilt recording functionality of AirSim as this is the simplest implementation method. Minor changes to the AirSim source code are required to ensure that time stamps associated with an image are in simulation time rather then wall clock time. The difference between simulation time and wall clock time is that wall clock time is the rate of time passing which we observe in the real world and would generally be measured as time since UNIX epoch. Simulation time is the rate of time flow within the simulation and is separated from wall clock time. This is normally done to ensure that the physics of the simulation can occur at a rate which is computationally achievable. As the ArduCopter SITL instance will measure simulation time we need our images to be timestamped in the same way.

To ensure that AirSim and ArduCopters simulation time pass at the same rate a lock step timing approach is used. A result of this is that if AirSim is started before ArduPilot it will hang while it waits for ArduPilot to connect. This effect can be used to generate a shared origin in time which both the ArduCopter SITL instance and AirSim can reference, allowing for synchronisation of the timestamps. Changes made to the AirSim source code ensure that the correct timestamps are used and additionally AirSims file

naming naming conventions to be as close to the EqVIO format as possible. Changes are made to three specific AirSim files where the modified versions can be found on GitHub[1]. Even with these changes to the AirSim source code the data is not in its finalised form so the script "*convert_data_types.py*"[2] is required.

4.3 Flight paths for AirSim

For this project six flight paths were developed within AirSim that aim to test the performance of the EqVIO in different situations. This was done so that the flight paths were consistent between experiments and so that different environments and movements could be tested in a controlled manor. Figure 4.3 shows a top down view of each flight path while table 4.2 provides more detail around the goal of each path. All flights start in the air rather then taking off from the ground to help improve the performance and stability of the EqVIO.



(a) path 1 (b) path 2 (c) path 3

(d) path 4 (e) path 5 (f) path 6

Figure 4.3: Flight paths used within AirSim

[1] https://github.com/liam327/u6670095_thesis/tree/main/AirSim_files/modified_AirSim_source_files
[2] https://github.com/liam327/u6670095_thesis/tree/main/AirSim_files/python_scripts

Flight number	Flight length	Details
1	339m	An easy path which is used to test if the EqVIO is converging and to give a baseline of what good performance should be. The flight has lots of features close to the drone and no sharp turns.
2	422m	An intermediate flight, which while still having many features, has a sharp 180°turn halfway through the flight. This is a more challenging maneuver for VIOs due to the fact that all features are lost during the maneuver.
3	484m	An intermediate flight which aims to test the performance of the EqVIO in a simulated landing and takeoff manoeuvre. Halfway through the flight the drone will land and takeoff in a clearing surrounded by trees. This gives lots of features while descending and reascending. The flight is otherwise very linear making it quite simple.
4	721m	An intermediate flight which aims to test the performance of the EqVIO when the height above the ground is changing through the flight. This is more challenging for VIOs due to the fact the scale factor is continuously changing.
5	1930m	A difficult flight aiming to test the limits of the EqVIO. This flight flies close to a snowy/rocky ground which provides few features and the remaining features are located far away from the drone. Additionally, sharp turns are present through this path.
6	1600m	Another difficult flight aiming to test the limits of the EqVIO. This flight has more features then flight path 5 but still flies over rocky terrain. Additionally the drone flies over a canyon adding an additional challenging feature. Sharp turns are also incorporated to increase the difficulty.

Table 4.2: Details for flight paths flown in AirSim

4.4 Error metrics

A set of six error metrics was devised that aim to quantify the overall performance of the EqVIO on each of the flight paths. These metrics will be used throughout this work to maintain consistency. The six metrics are described below:

- Relative position error - This is defined as the Root Mean Square Error (RMSE) of the position divided by the flight length. This is a widely used metric which can be interpreted as the average positional drift of the drone as it flies. Other high performing VIO's tend to have values between 2-5%.
- Maximum position error (m) - This metric is a good indicator on the overall stability of the position. If the maximum error is extremely large but the relative position error is small then this can tell us that while we have low drift we have at some

point had a large positional error indicating instability in the system.

- Horizontal velocity RMSE (m/s) - This tells us about the error in the XY velocity of the drone. This is an important metric for ArduPilot because if accuracy is achieved in the velocity then position can be found by integration. The horizontal and vertical velocity are split as they are often not correlated.
- Vertical velocity RMSE (m/s) - The reasoning behind this metric is identical to that of the vertical velocity.
- Attitude error standard deviation (°) - A high standard deviation on the attitude error would imply that the attitude of the estimate changes significantly throughout the flight compared to the true value. This allows us to loosely interpret this value as the stability of the attitude estimate when compared to the ground truth. A low attitude error standard deviation tells us that the shape of the attitude estimate closely matches the shape of the ground truth attitude.
- Maximum attitude error (°) - Due to the fact that the attitude is aligned with the direction of minimum positional error metrics such as attitude RMSE become redundant as the attitude estimate has been globally rotated. The maximum attitude error can be interpreted alongside the attitude error standard deviation to gain insight into the accuracy of the attitude estimate. Large maximum attitude error with small attitude error standard deviation would indicate potentially good tracking but poor alignment while small maximum attitude error would indicate either good performance or good alignment.

It should be noted that with many of these error metrics, the actual values are not useful in and of themselves. Instead, comparisons to the results of other experiments should be made to gain insight.

4.5 AirSim results

Data was collected for each of the six flight paths and the EqVIO run using this data. Config tuning was completed prior to finalising any results to ensure good performance on as many paths as possible using an existing optimisation script which optimises each parameter one after the other. While this does not ensure a global minimum it should reach a local minimum giving semi-optimal performance.

4.5.1 Noise in GPS data from AirSim

One issue identified with AirSim was that the raw GPS data being received was quite noisy. This data is important as it is used to generate a ground truth position and ground truth body fixed velocity. The magnitude of the noise is small enough that it does not significantly effect the accuracy of the position's ground truth. The noise does have a large impact on the velocity ground truth as it is calculated as the point to point gradient

of the position ground truth. This means that only a small amount of noise is required to caused significant error in the velocity ground truth. An example of this noise is shown in figure 4.4.

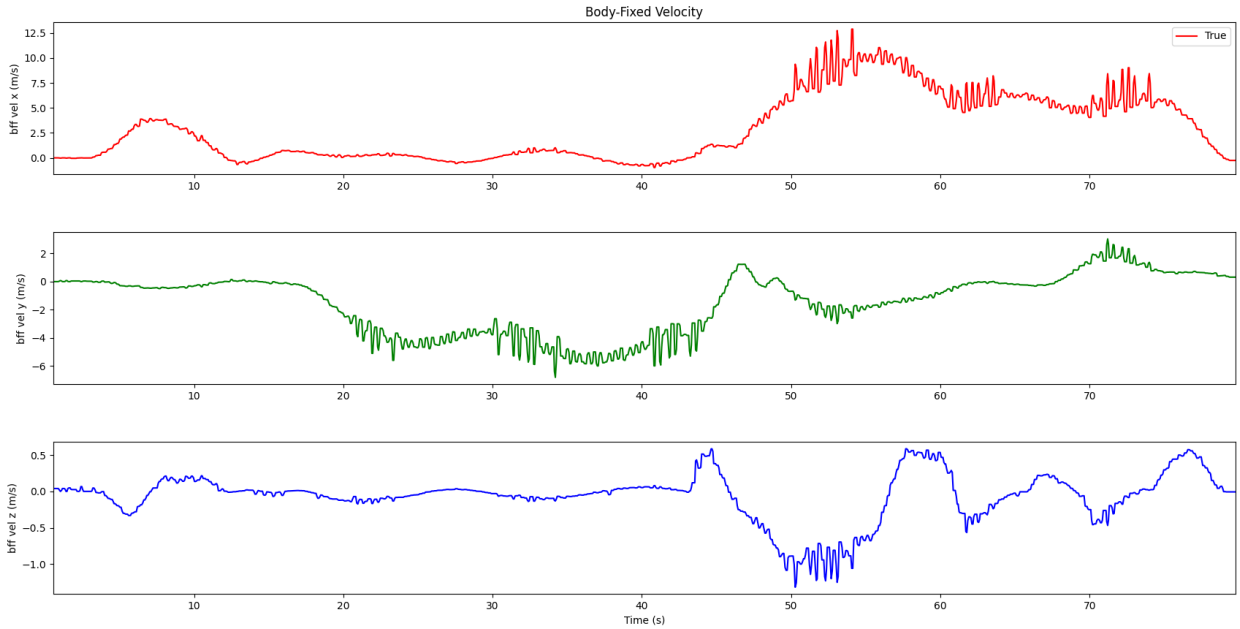


Figure 4.4: Velocity noise in raw GPS signal

This issue can be mitigated by using ArduPilot's internal estimate of the GPS position rather then the raw GPS values. While this does not remove all the noise in the velocity, it does help to reduce the magnitude of the noise and make them more regular. An example of the body fixed velocity given by the ArduPilot GPS estimate for the same flight as figure 4.4 is shown in figure 4.5.

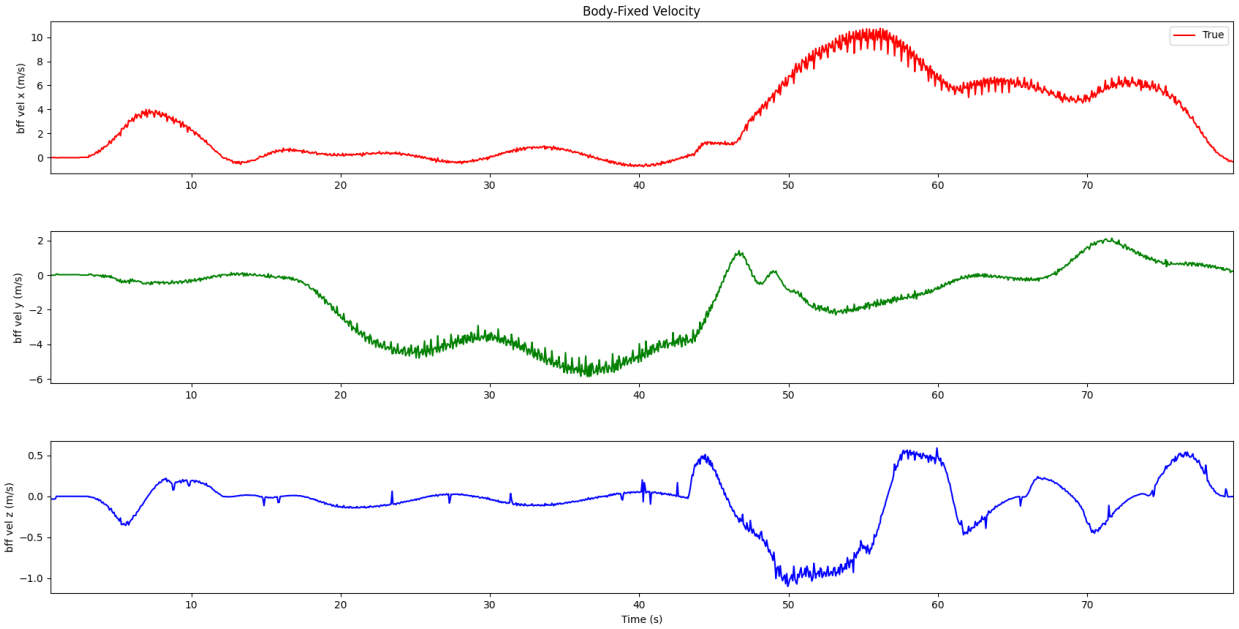


Figure 4.5: Velocity noise in ArduPilot GPS estimate

4.5.2 Results of EqVIO in AirSim

Table 4.3 shows the results of the EqVIO for each of the six flight paths according to the error metrics discussed in section 4.4.

Flight path	Relative position error (%)	Max position error	Horizontal velocity RMSE	Vertical velocity RMSE	Max attitude error (°)	Attitude error standard deviation (°)
1	5.19	47.13	3.69	0.28	67.44	10.43
2	3.31	33.42	1.55	0.16	9.9	1.65
3	7.64	49.46	3.36	0.34	29.86	1.15
4	27.78	617.05	8.67	0.711	85.5	27.6
5	17.4	563.44	8.03	0.82	21.09	3.36
6	11.55	336.69	8.84	1.33	55.5	12.92

Table 4.3: Results of EqVIO AirSim flight paths

We can observe that the EqVIO is doing a good job at estimating the position of the drone for paths 1 and 2 due to the low relative position error. Given that both paths were designed to be relatively easy this is not surprising, however it does show that the EqVIO can accurately predict the position of the drone for simple paths.

An interesting result from flight 1 is that we have a large maximum attitude error. Splitting this attitude error into roll, pitch and yaw shows that the roll and pitch maintain a constant error of 2 and 4 degrees respectively over the length of the flight. The yaw's error is much larger with an average value of approximately 60 degrees and additionally has much higher variance than the roll or pitch. The additional error in the yaw axis is likely due to the fact that the yaw is assigned some arbitrary origin with no physical meaning when the EqVIO is initialised. In contrast the roll and pitch of the drone are initialised from the gravity direction giving them a physical origin. When there is no physical meaning to the origin it is hard to pick an accurate initial direction so a higher error in the yaw axis is likely to be a persistent problem with the EqVIO. Interestingly, path 2 did not experience this yaw error even though it is a very similar path sharing many similar features. This likely indicates that there is an instability or unpredictability in the attitude estimates due to the poor initial estimate of the yaw axis.

Paths 3 and 4 are more advanced paths, both containing a change in height above the ground during the flight. Path 4 performs significantly worse then path 3 with both worse general performance and worse vertical velocity RMSE. The major difference between the two flights is the environment surrounding them when the change in height occurs. During flight 4, the environment around the drone was very rocky meaning that all visible landmarks were located on the ground. In flight 3 the drone had trees surrounding it when the change in height occurred. This indicates that the EqVIO generally struggles with paths that change their height above the ground except when a large number of features are present surrounding the drone which it can use to stabilise the system.

Paths 5 and 6 show similar poor performance to path 4 with very high relative position error and velocity RMSE. The attitude estimate of paths 5 and 6 is slightly better then that of path 4 but is still not particularly accurate. To summarise, on the easier paths low relative position error was achieved with somewhat unpredictable attitude performance. On the challenging paths significantly worse performance was achieved in all metrics. This is not unexpected due the incorporation of sharp turns and far away features. The unpredictability of the attitude error for the easier paths is more surprising so a potential solution for this is discussed in the next chapter.

EqVIO Attitude Innovation

While some of the results from section 4.5 are promising, there is an unpredictable attitude error in most simulations. To help improve the stability of the attitude estimate we can take advantage of the fact that ArduPilot accurately estimates the attitude within its EKF. This is more accurate then the EqVIOs attitude estimate due to the fact that they receive IMU data at approximately 8kHz while the EqVIO samples IMU data at 200Hz. Using an innovation we can measure the value of ArduPilot's attitude estimate and then by treating it as a measurement of the ground truth attitude, incorporate it into the EqVIO. Doing this should reduce the instability in the attitude estimate and improve the systems performance.

5.1 Derivation of attitude innovation equations

The system we are trying to represent within the EqVIO as described in section 2.4.2 is the following:

$$\xi = (P, v, b, p_i, T) \tag{5.1}$$

The attitude of the system is a rotation element $R \in \mathbf{SO}(3)$ which is contained within the pose of the system $P \in \mathbf{SE}(3)$. The exact value of ξ is not known to the observer and instead we make an estimate of the state $\hat{\xi}$ based on measurements of the system. This observer state $\hat{\xi}$ gives an estimate of each of the parameters defined in equation 5.1. It is important to note that this estimate require us to assign some initial origin for the system ξ^0 however the actual choice of the origin is not important.

To improve the estimated states accuracy we are going to include measurements of the ArduPilots attitude estimate within the EqVIO. To do this we define the measurement function $h^R(\hat{\xi}) : \mathcal{M} \longrightarrow \mathbf{SO}(3)$ which maps the manifold the observer sits on $\mathcal{M}$ to $\mathbf{SO}(3)$. This is defined as:

$$h^R(\hat{\xi}) = R_{\hat{P}} = R_{P_0} R_{\hat{A}} \tag{5.2}$$

Where $R_{\hat{P}}$ is the system attitude estimate, R_{P_0} is our origin of the attitude and $R_{\hat{A}}$ is the estimate of our current state with respect to our origin. By then including a measurement

for ArduPilot's attitude estimate R_P we can calculate the error in the attitude R_e. This is defined as:

$$R_e = R_{P_0}^T R_P R_{\hat{A}}^T \tag{5.3}$$

This error in the attitude R_e can be thought of as rotating from our origin by our measured attitude from ArduPilot and then rotating backwards by our attitude estimate. In the case that our attitude estimate is perfect, then this will take us back to the origin while any deviation from the origin will tell us about the error in our estimate. We can now define our innovation as the log error of R_e:

$$\tilde{y} = \log(R_e) = \log(R_{P_0}^T R_P R_{\hat{A}}^T)^\vee \tag{5.4}$$

Where $T^\vee$ represent a variable T going from vector to skew symmetric form and $T^\wedge$ represents a skew symmetric matrix going back to vector form. The skew symmetric form of a vector $T = (t_1, t_2, t_3) \in \mathbb{R}^3$ is defined as:

$$T^\times = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$$

This is useful as for some other variable $p \in \mathbb{R}^3$ we can find that $T^\times p = T \times p$ where $\times$ is the cross product. Another variable which must be calculated is the output matrix C_t^R for the attitude innovation. The values in C_t^R are different to those in C_t which is used in the image update step so it must be recalculated. C_t^R is used to update the covariance matrix Σ_t using the dynamics shown below:

$$\Sigma_{t+1} = \Sigma_t - K_t C_t^R \Sigma_t; \tag{5.5}$$

Where K_t is the Kalman gain. C_t^R can be derived as the Jacobin of the innovation and tells us the rate at which our innovation changes. This is defined as:

$$C_t^R = D_{\epsilon|0}\tilde{y}(\epsilon) \tag{5.6}$$

Where $D_{\epsilon|0}$ represents the derivative of the innovation $\tilde{y}(\epsilon)$ evaluated at $\epsilon = 0$. We now consider multiplying the system by some vector u. This means that our error can be approximated as some small error in the u direction giving us $\epsilon \approx 0 + su$.

$$\implies C_t^R[u] = D_{\epsilon|0}\tilde{y}(\epsilon)[u] \tag{5.7}$$

$$= \frac{d}{ds}\Big|_{s=0} \tilde{y}(0 + su) \tag{5.8}$$

Substituting in our definition for $\tilde{y}$ from equation 5.4 gives:

$$C_t^R[u] = \frac{d}{ds}\Big|_{s=0} \log(R_{P_0}^T R_P(\epsilon) R_{\hat{A}}^T)^\vee \tag{5.9}$$

By rearranging equations 5.4 and 5.3 we can find that:

$$R_e(\epsilon) = \exp(su^\wedge) \tag{5.10}$$

$$R_P(\epsilon) = R_{P_0} R_e(\epsilon) R_{\hat{A}} = R_{P_0} \exp(su^\wedge) R_{\hat{A}} \tag{5.11}$$

Finally by substituting the result of equation 5.11 into equation 5.9 we find:

$$\implies C_t^R[u] = \frac{d}{ds}\Big|_{s=0} \log(R_{P_0}^T (R_{P_0} \exp(su^\wedge) R_{\hat{A}}) R_{\hat{A}}^T)^\vee \tag{5.12}$$

$$= \frac{d}{ds}\Big|_{s=0} \log(\exp(su^\wedge))^\vee \tag{5.13}$$

$$= \frac{d}{ds}\Big|_{s=0} su \tag{5.14}$$

$$= u \tag{5.15}$$

This can only be true if $C_t^R = I$ meaning that the portion of C_t^R that acts on the attitude must be the identity while all other parts of C_t^R must be zero. The final matrix which is required for the EqVIO is the measurement noise matrix Q_t. In this case it will be given by the identify matrix multiplied by some covariance associated with the uncertainty in the attitude estimation provided from ArduPilot.

5.2 Details of attitude innovation implementation

While we have derived that the portion of C_t^R which acts on the attitude must be the identity matrix, the actual position of the attitude within the wider system state ξ is defined by the implementation of the EqVIO. Without considering the landmarks which are added to the end of the state, the implementation for the EqVIO has the states ordered as follows:

$$\xi = \begin{bmatrix} [IMUbias]_{3\times6} & [Attitude]_{3\times3} & [Position]_{3\times3} & [Velocity]_{3\times3} & [CameraPose]_{3\times6} \end{bmatrix}_{3\times21}$$

Therefore C_t^R is given by:

$$C_t^R = \begin{bmatrix} 0_{3\times6} & I_{3\times3} & 0_{3\times12} \end{bmatrix}_{3\times21}$$

One other detail that must be considered is that while ArduPilot uses North East Down (NED) coordinates for attitude, the EqVIO uses East North Up (ENU) coordinates for attitude. Therefore the attitude estimate measurements from ArduPilot must be converted from NED to ENU. To do this we can multiply the attitude measurements from ArduPilot with the following matrix:

$$R_{ned2enu} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

5.3 Initial results from attitude innovation

The results for the EqVIO with the attitude innovation included are shown in table 5.1. This data was recorded on the same paths as discussed in section 4.3.

Flight path	Relative position error (%)	Max position error	Horizontal velocity RMSE	Vertical velocity RMSE	Max attitude error (°)	Attitude error standard deviation (°)
1	2.65	27.97	1.2	0.2	10.37	0.72
2	3.25	24.35	1.73	0.19	6.06	0.51
3	7.44	68.02	2.18	0.31	13.38	0.36
4	4.69	73.33	1.49	0.78	19.4	0.62
5	5.84	160.43	3.87	0.4	25.37	1.14
6	9.83	269.84	4.3	0.48	14.54	0.35

Table 5.1: Initial results for EqVIO with attitude innovation

To quantify how much of an improvement the attitude innovation provides, an average of each error metric can be taken over all 6 paths for both the base EqVIO and the EqVIO with the attitude innovation added. This is shown in table 5.2 along with the relative improvement in each error metric due to the attitude innovation being added.

EqVIO type	Relative position error (%)	Max position error	Horizontal velocity RMSE	Vertical velocity RMSE	Max attitude error (°)	Attitude error standard deviation (°)
Base EqVIO	12.15	274.53	5.69	0.61	44.88	9.52
EqVIO with Attitude Innovation	5.62	103.99	2.46	0.39	14.85	0.62
Relative Improvement (%)	216.23	264.00	231.14	154.28	302.17	1543.51

Table 5.2: Average value of error metrics for EqVIO with attitude innovation

Table 5.2 demonstrates that by including the attitude innovation we improve the performance of the EqVIO in every error metric. In particular the attitude error standard deviation has seen the most significant improvement with a 1543% increase over the base EqVIO. This is not unexpected as the attitude innovation should have the most significant impact on the attitude error given this is what the attitude innovation measures. All other error metrics have seen a 150-300% increase in performance.

While these results from Table 5.2 are very positive, it should be noted that the averages for the base EqVIO are heavily skewed due to the paths which achieved very poor performance. There are instances where the base EqVIO performs equally as well as the EqVIO with the attitude innovation such as for the relative position error of flight 3. Instead what we can observe is that adding the attitude innovation has helped to make the system more constantly accurate and less likely to diverge. This is important as by

increasing the robustness of the system, we increase the number of paths the drone can fly while still achieving a good state estimate.

While the results so far are very positive, by default ArduPilot uses GPS as one of the inputs into the EKF where the attitude measurement is derived from. In general however the EqVIO will be used within ArduPilot when no GPS is available so we need to test the effect of removing GPS from the attitude estimate.

5.4 Attitude innovation without GPS

To test the attitude innovation without GPS, we will want to have a ground truth being recorded with GPS and an attitude estimate available without GPS input. The best way to do this is by running two of ArduPilots EKFs simultaneously in what is called ride along mode. In this mode one EKF will receive GPS data and will fly the simulated drone. The other EKF will operate in ride along mode where it will receive all the same inputs as the main EKF except GPS but will not control the drone. We can then get ground truth data from the main EKF and attitude data without GPS from the ride along EKF.

As this data was recorded through AirSim we can use EKF10 to provide a perfectly accurate estimate for our ground truth. The attitude estimate without GPS will then be supplied by EKF3. Additionally we need to change the MAVLink message which is used to retrieve the attitude estimate as the normal message "*ATTITUDE_QUATERNION*" will retrieve attitude of EKF10. To retrieve the estimate from EKF3 we use the MAVLink message "*AHRS2*" however "*AHRS2*" does not provide a time stamp with the message. To get an estimate of the time stamp the wall clock time was initially used however this induced a significant time scaling issue. To correct this the assumption is made that the rate at which simulation time and wall clock time pass are locally linear so that we can write:

$$t_{sim} = mt_{wall} + b \tag{5.16}$$

Where t_{sim} and t_{wall} are the amount of time passing between time steps for the simulation and wall clock time respectively. m is a scale factor which essentially tells us how long one wall clock second is compared to one simulation second. Finally b is the y-intercept for the wall clock time however as discussed in section 4.2.4 we know that the zero point in time for wall clock time and simulation time is equal meaning that $b = 0$. This means that we can get the equation for our scale factor m as:

$$m = \frac{t_{sim}}{t_{wall}} \tag{5.17}$$

To find the scale factor m we divide the current simulation time by the current wall clock time. This value of the scale factor is then used to convert the next measurement of wall

clock time to simulation time. This results in a loop where we update the scale factor and then correct the next value of the wall clock time. This allows for the scale factor to be updated every time we record IMU data which is important as changes in CPU utilisation on the computer running AirSim could change the rate at which time passes within simulation. A comparison can now be made between the attitude data provided by ArduPilot both when it is consuming GPS data and when it is not. This is shown in figure 5.1.

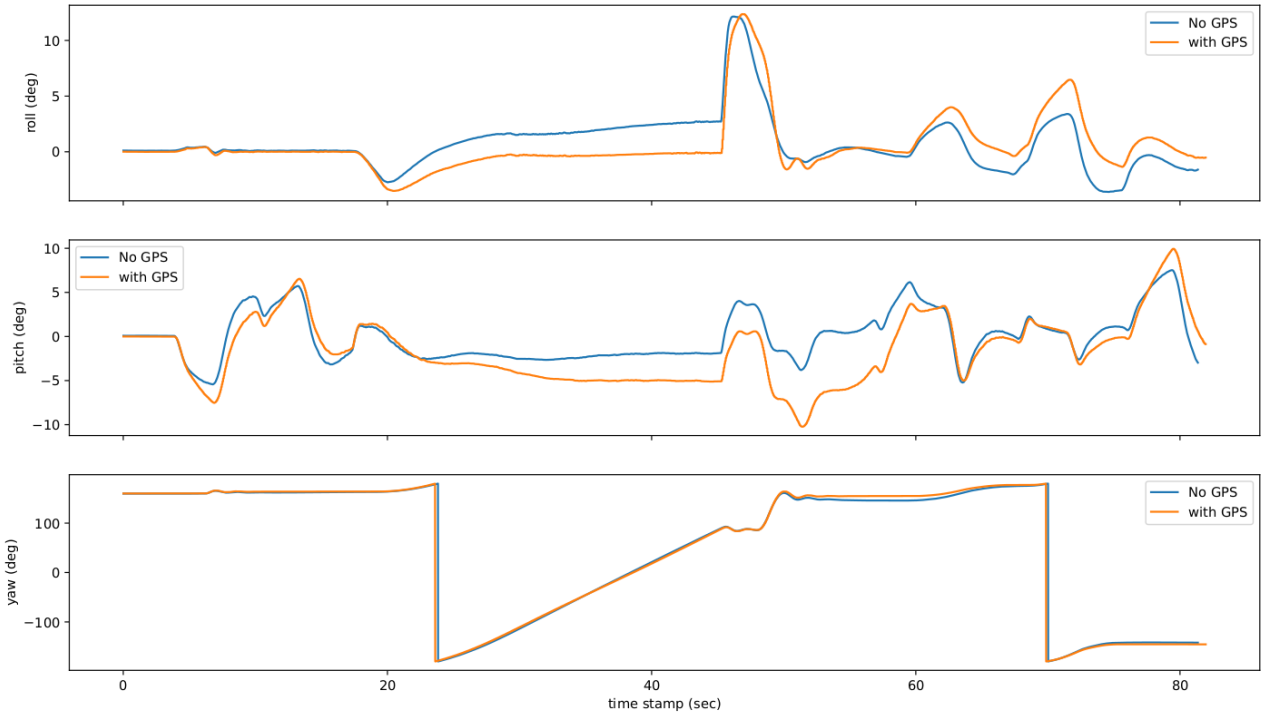


Figure 5.1: Ground truth attitude and attitude estimate without GPS

5.5 Results for attitude innovation without GPS

Table 5.3 shows the results of the EqVIO when using the attitude innovation without GPS.

Flight path	Relative position error (%)	Max position error	Horizontal velocity RMSE	Vertical velocity RMSE	Max attitude error (°)	Attitude error standard deviation (°)
1	15.78	101.28	5.58	0.52	107.92	3.81
2	13.21	88.64	6.7	1.16	47.14	3.82
3	23.89	238.16	8	1.24	56.02	1.61
4	12.77	210	5.32	1.41	74.18	2.12
5	34.8	1099.82	14.11	1.3	35.91	3.35
6	15.98	409.58	12.25	2.98	25.58	2.86

Table 5.3: Results for EqVIO with attitude innovation without GPS

Table 5.3 shows extremely high relative position error even on the easy paths indicating very high levels of drift. The horizontal and vertical velocity RMSE is also very high on almost every path. To further quantify these results table 5.4 shows the average of each error metric over the six flights paths for the base EqVIO and the EqVIO with attitude innovation without GPS. It also shows the relative performance gain or loss compared to the baseline.

EqVIO type	Relative position error (%)	Max position error	Horizontal velocity RMSE	Vertical velocity RMSE	Max attitude error (°)	Attitude error standard deviation (°)
Base EqVIO	12.15	274.53	5.69	0.61	44.88	9.52
EqVIO with Attitude Innovation without GPS	19.41	357.91	8.66	1.44	57.79	2.93
Relative Improvement (%)	62.59	76.70	65.70	42.29	77.66	325.04

Table 5.4: Mean value of error metrics for attitude innovation without GPS

What we observe is that when the attitude innovation without GPS is included, we are only achieving 60-80% of the performance of the base EqVIO in most error metrics. The exception to this is the attitude error standard deviation which did improve implying a more stable attitude estimate. This indicates that in almost all circumstances the attitude innovation should not be used except when ArduPilot is receiving GPS.

To identify why the results for the EqVIO with attitude innovation are so poor when ArduPilot is not recieving GPS, we can run simulations where artificial noise is added to the ground truth attitude. This will allow us to test which types of noise most significantly effect the EqVIO and the impact of the magnitude of this noise.

Figure 5.1 shows that the type of noise we are trying to approximate is not Gaussian noise (White noise) as the mean value of the attitude without GPS does not align with the true value. Instead we need to model a type of noise that will have a different mean value to the ground truth attitude. To do this Brownian noise [35] was used. This creates noise generated from Brownian motion [36] where at each time step, the change in the attitude of the system is decided by a Gaussian distribution with standard deviation proportional to the size of the time step. Changing the magnitude of this Gaussian allows for control of how fast the random walk will move. This Brownian noise is then smoothed to remove some of the high frequency noise. Examples of the Brownian noise generated and the attitude with the noise added are shown in figures 5.2 and 5.3 respectively.

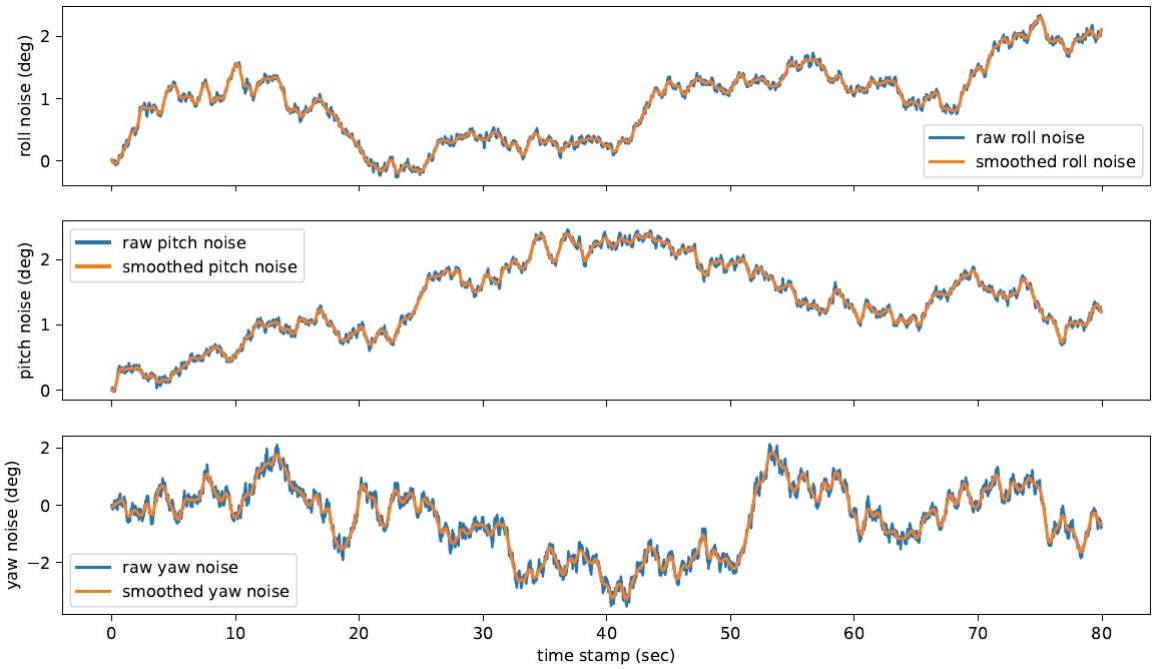


Figure 5.2: Example of Artificial Brownian noise

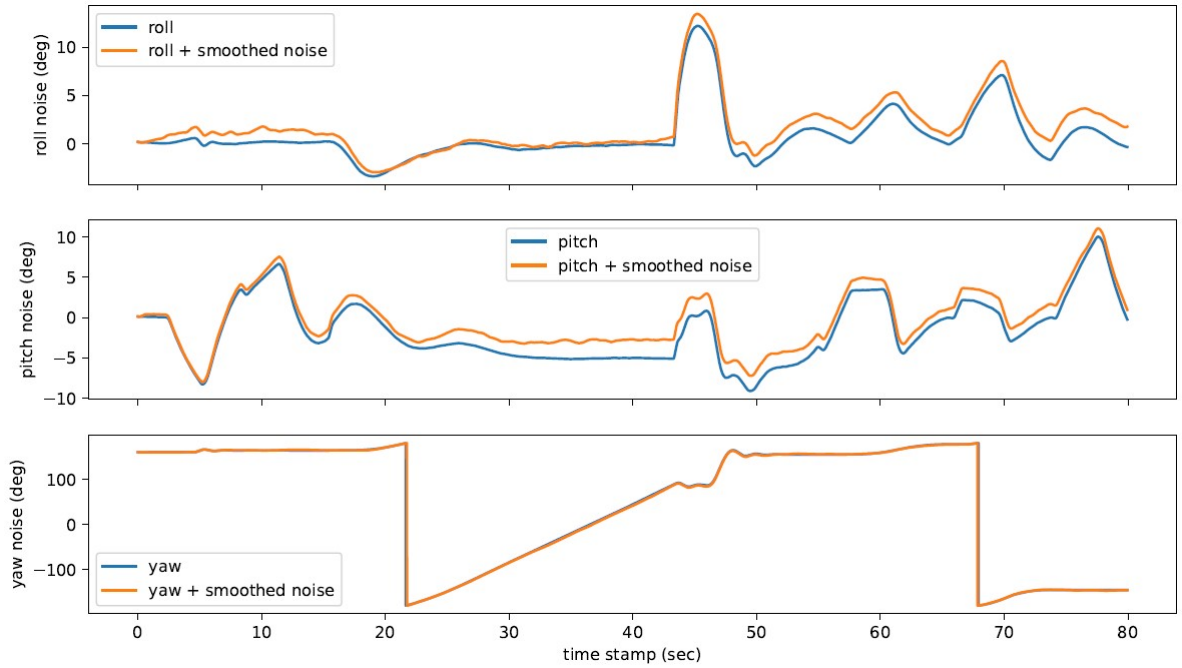


Figure 5.3: Ground truth attitude with Brownian noise added

While not of the exact same form as the noise from figure 5.1 it is close enough to allow for testing of the effect of noise to the attitude data used in the attitude innovation for the EqVIO. Tests were run on the effect of the magnitude of the noise being added as well as which axis (roll, pitch and yaw) were most effected by the noise. Additionally, the effect of adding Gaussian noise to the ground truth is also tested to see if the type of noise being added is the significant factor. The results for this are shown in table 5.5 for data collected on flight path 1.

Performance metric	No Noise added			With simulated noise added				
	Baseline	Attitude with GPS	Attitude without GPS	All axis noise	Yaw noise	Roll and pitch noise	Large roll and pitch noise	Gaussian noise all axis
maximum error in position (m)	47.13	15.42	101.28	156.11	19.47	123.82	132.22	26.96
Relative position error (%)	5.19	2.24	15.78	20.54	2.38	14.39	14.45	2.53
RMSE for velocity (m/s)	3.71	1.14	5.6	2.88	1.16	2.64	6.45	1.21
Standard deviation for attitude (d)	10.42	1.04	3.8	2.23	1.59	2.59	3.58	0.71
Maximum error in attitude (d)	67.43	12.95	107.92	16.11	14.56	23.66	43.62	10.27

Table 5.5: Comparison of attitude without GPS to simulated noise

Table 5.5 shows a couple of interesting trends. The first and most important observation is that when we only add Gaussian noise, we perform significantly better then almost every example of when Brownian noise is added. This strongly indicates that the type of noise found in the ArduPilot estimate of attitude when no GPS is present is what is causing the large loss in performance. Another important observation is that adding Brownian noise to only the yaw does not seem to significantly decrease performance compared to when noise is added to either the roll or pitch.

These observations suggest that the loss in performance is due to the fact that all EKF's model noise as a Gaussian distribution with some mean value and variance. Figure 5.1 shows that there are extended periods where the attitude estimate is a long way from the true value which is not what we expect from a Gaussian distribution. As we take ArduPilot's attitude estimate to be a measurement of the attitude of the system the EqVIO attempts to correct the state estimate to match the incorrect ArduPilot estimate. When this occurs for extended periods of time it has the effect of changing the gravity direction estimate which results in more of the acceleration due to gravity being assigned to the x and y axis rather then the z axis. As acceleration is related to position through a quadratic relationship, a small error in the gravity direction results in a large error in the position. This also explains why the velocity RMSE was so high when this attitude error is present as the velocity shares a linear relationship with the acceleration.

This ultimately means there is no way to improve the performance of the EqVIO using an attitude innovation when we are receiving attitude data without GPS, as the type of noise present is different to the type of noise we are modeling. There is still a use case for the EqVIO with the attitude innovation when we are receiving GPS as it is significantly more accurate then the base EqVIO. This could, for example, be used to initialise the filter while we are receiving GPS data, allowing for highly accurate estimates of our IMU biases and camera pose. When GPS is no longer available we could stop consuming attitude data and freeze the bias and camera pose estimates in a known accurate state. A future project could investigate this use case as well as adding in innovations based on GPS position and velocity to further improve the accuracy of the EqVIO while flying with GPS.

Real Time EqVIO

6.1 Introduction to the APVIO

Previous work has been completed to develop a real time version of the EqVIO called the APVIO, however the APVIO was non functional. This project built on the existing APVIO code base to correct the code and added additional functionality such as logging of data and updating the code to run on the newest version of the EqVIO. Smaller changes like adding the ability to change the camera resolution and frame rate through the config file were also included.

The first step of the APVIO implementation was to get the outputs of the APVIO matching those of the EqVIO when the same inputs were given. If this was the case then the APVIO is implemented identically to the EqVIO but is running in real time. Once this had been completed test flights could be conducted with the APVIO running onboard a raspberry pi attached to the drone. These flights are used to test the performance of the APVIO.

The hardware setup of the drone running the APVIO is shown in figure 6.1. Images are taken at 20Hz while IMU and GPS data are passed through to the Raspberry Pi from the ArduPilot flight controller at 200Hz and 20Hz respectively.

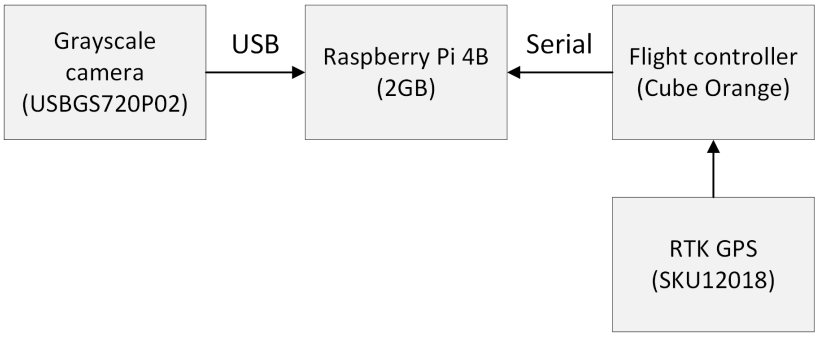


Figure 6.1: Drone hardware structure

For the purpose of this project real time has been taken to mean having a processing time per frame less then 50ms. As camera images arrive at 20Hz we have 50ms to process an image before the next image arrives and must be processed. This concept is used to

provide a hard limit around how long a loop of the processing section of the code can take.

6.2 Data logging in the APVIO

There are two purposes of logging data in the APVIO. The first is to log the state estimate outputted by the APVIO which can then be compared to the ground truth allowing for analysis of the performance of the APVIO. The second is to log image and IMU data which allows for the offline EqVIO to be run on data collected from the APVIO.

Logging in the APVIO needs to be implemented slightly differently to logging within the EqVIO due to the fact that it needs to run in real time. To achieve this we are required to use a threaded system where the main processing thread passes data to separate logging threads so that the reads and writes to the SD card do not interfere with the processing of data. The flow of data is described in Figure 6.2.

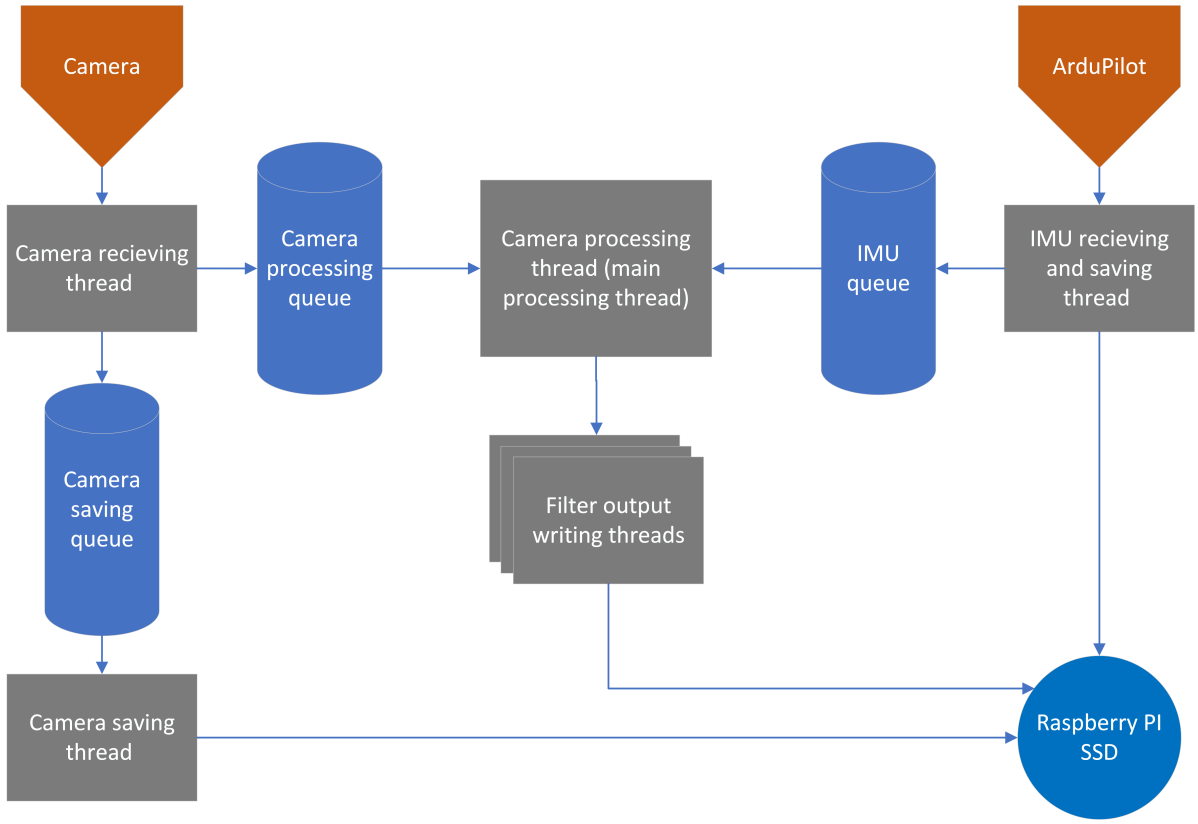


Figure 6.2: Data flow within the APVIO

The camera processing thread is an important thread where the IMU data and images are processed via the EqVIO. It draws data from the camera processing queue and the IMU queue. These First in First out (FIFO) queues are required to push data between threads and also help to keep the data organised by the time received. Once the filter has been processed the output is passed to the filter output thread. These threads allow

for the data saving process to be done outside of the main processing thread which helps to remove potential bottlenecks within the system.

While logging of the APVIO outputs is currently functional, the logging of images and IMU data required to run the EqVIO is not functional. This is due to an issue within the filter output thread that can cause an intermittent delay in processing when logging is turned on. This is due to a issue within the "*aofstream.h*" file where there is conflict between mutex locks while the output is being flushed to the SD card. This issue has not currently been solved and as such logging can not be used reliably.

6.3 Installing and running the APVIO

To install the APVIO use the commands:

```
$ git clone --recursive https://github.com/ShawnGYX/AP_VIO.git
$ cd AP_VIO
$ mkdir build
$ cd build
$ cmake ..
$ make
```

There are several prerequisites which must be installed prior to installing the APVIO including Eigen 3, yaml-cpp and GIFT. Due to the APVIO using C++17 there are some features which are not natively supported by the Raspberry Pi as it uses GCC 8.3. To fix this GCC 10 [37] must be installed. Once installed, two lines of code must be added to the start of the APVIO cmake file to tell it where to find GCC 10 and G++ 10. This code is:

```
set(CMAKE_C_COMPILER "/opt/gcc-10.1.0")
set(CMAKE_CXX_COMPILER "/opt/gcc-10.1.0/bin/g++-10.1")
```

The APVIO can then be run using the command:

```
$ ./build/vio_ap EQVIO_config_simple.yaml /dev/serial0
```

6.4 Configuring the APVIO

There are a number of parameters which must be correctly tuned to accurately run the APVIO. Examples of potential configurations are provided in the repository[1] however details around several key parameters are provided below in table 6.1.

[1] https://github.com/ShawnGYX/AP_VIO

Configuration Parameter	Explanation
equaliseImageHistogram	This tells the image processing if the image should go through histogram equalisation. In general leave this as true.
featureDist	This specifies the minimum distance between features on within an image. Setting this value too small will result in features clustering together resulting in worse performance.
maxFeatures	The maximum number of features to track at any given time.
sceneDepth	This is related to the distance which features are initialised from the camera plane. Setting this to the approximate height above the ground seems to give reasonable results.
cameraOffset	This is the offset from the camera to the IMU (cam to imu matrix within Kalibr). The first three numbers are the XYZ position offset. The next four number is the quaternion rotation with order WXYZ.
initialVariance	These values set the variances when the APVIO is initialised.
processVariance	The variance added each iteration.
velocityNoise	This is the IMU noise parameters which were found using Allan variance as described in section 3.1.
indoorLighting	This is a Boolean value describing if the drone is inside or outside. This sets the initial camera exposure.
camera(X/Y)Resolution	The camera resolution at which images should be taken at. Recommended to run at 480p or no higher then 720p. Cameras should be calibrated at this resolution.
cameraFrameRate	The frame rate desired from the camera.
saveImages	A Boolean value describing if camera images should be saved awhile the APVIO is running. This should only be turned on if the EqVIO will be run offline using these images.

Table 6.1: Configuration paramaters for APVIO

6.5 Comparing the EqVIO and the APVIO

The first experiment which was run while implementing the APVIO was to test that the outputs of the APVIO and EqVIO matched when inputted with the same data. This is very important to test as any major difference here would indicate a fundamental difference in the way the APVIO is running when compared to the EqVIO. This experiment can be conducted on the ground by picking up the drone and walking around with it at chest height. This allows for enough data to be gathered to validate that the outputs of the APVIO and the EqVIO are equivalent.

Figure 6.3 shows an example of a path recorded by walking around in a circle on an oval with the drone held at chest height. Data was recorded on an oval to improve GPS ground truth accuracy. What we can observe is that the APVIO and EqVIO outputs are extremely close giving confidence that the filters are operating identically.

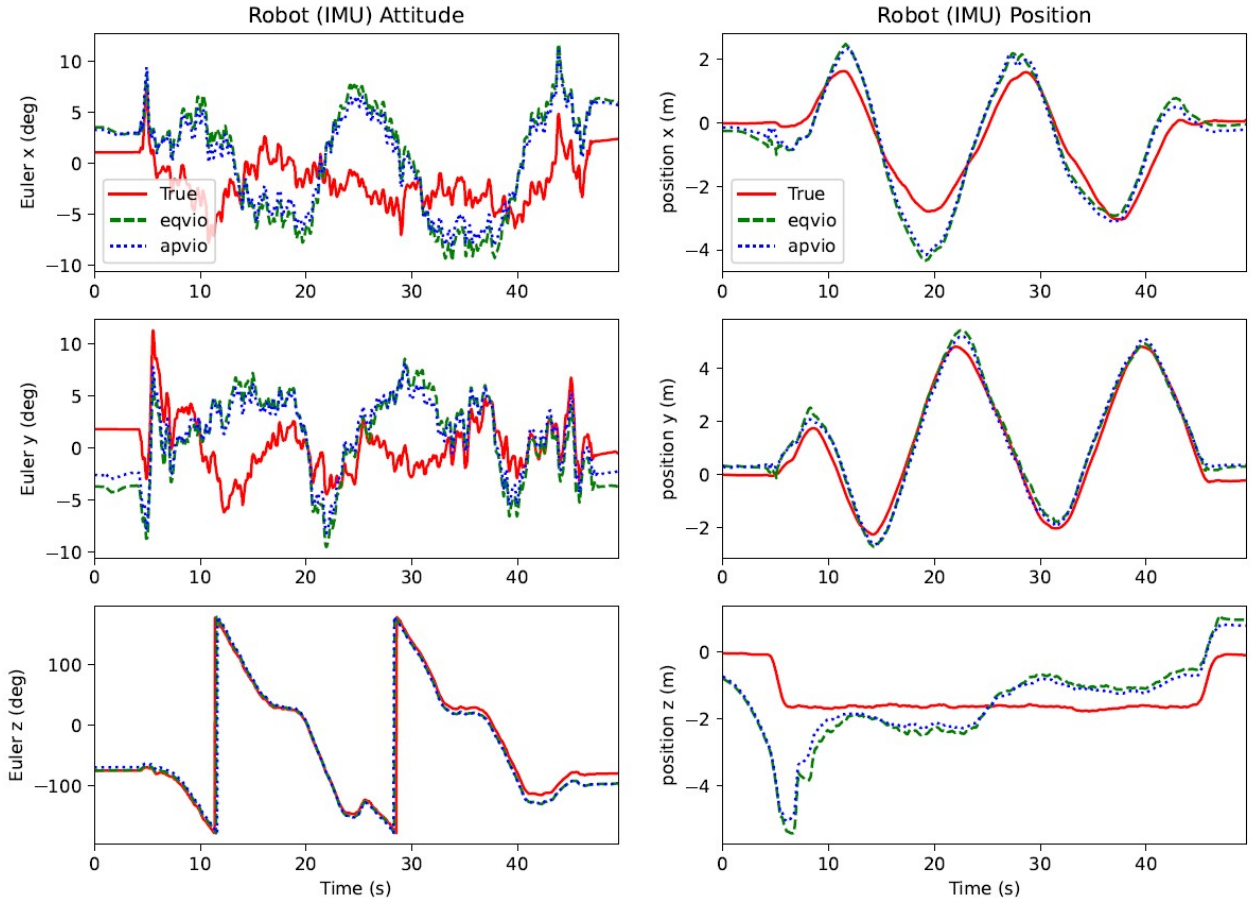


Figure 6.3: EqVIO and APVIO position and attitude comparison

While the results in figure 6.3 shows that the APVIO and EqVIO outputs are nearly identical, looking closely we can note that the APVIO is more slightly accurate then the EqVIO. This is additionally supported by the APVIO having a lower relative position error of 3.04 compared to the EqVIOs relative position error of 3.39. This small difference is likely due to the fact that the images being used by the EqVIO are stored as JPGs which are a compressed image format. This means that there is some small loss in detail within

the images used by the EqVIO compared to the raw images used by the APVIO which would likely result in slightly worse performance. Further config tuning was completed to improve the performance of the system before the drone was flown in the air. The results after this config tuning are shown in figure 6.4.

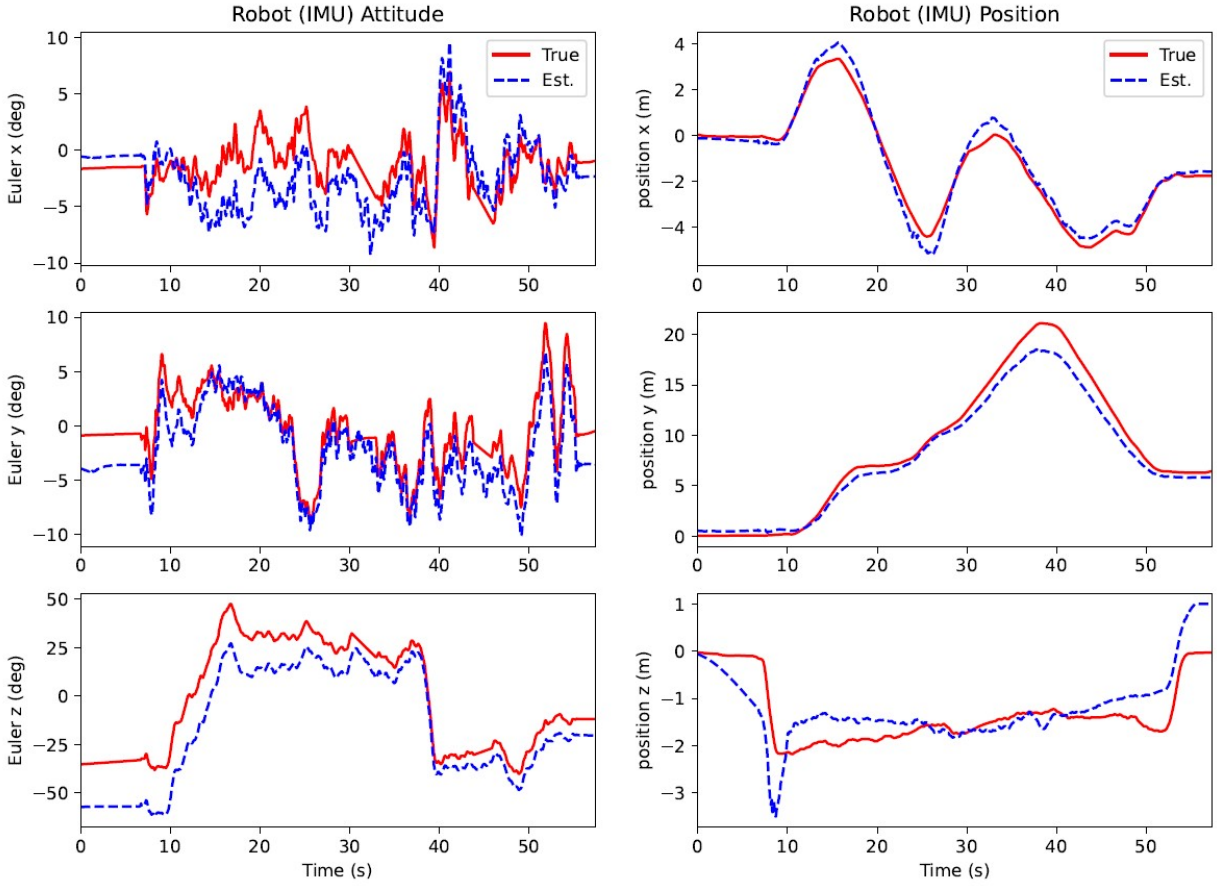


Figure 6.4: Pose results for tuned APVIO

This shows much better attitude performance then the previous test while still maintaining similar positional accuracy. Interestingly the z axis of the position performs worse then the x and y axis indicating that the system might struggle to estimate the height above the ground accurately. It is also interesting to consider the gravity direction and body fixed velocity of our system as these are two variables that are directly observable. This is shown in figure 6.5.

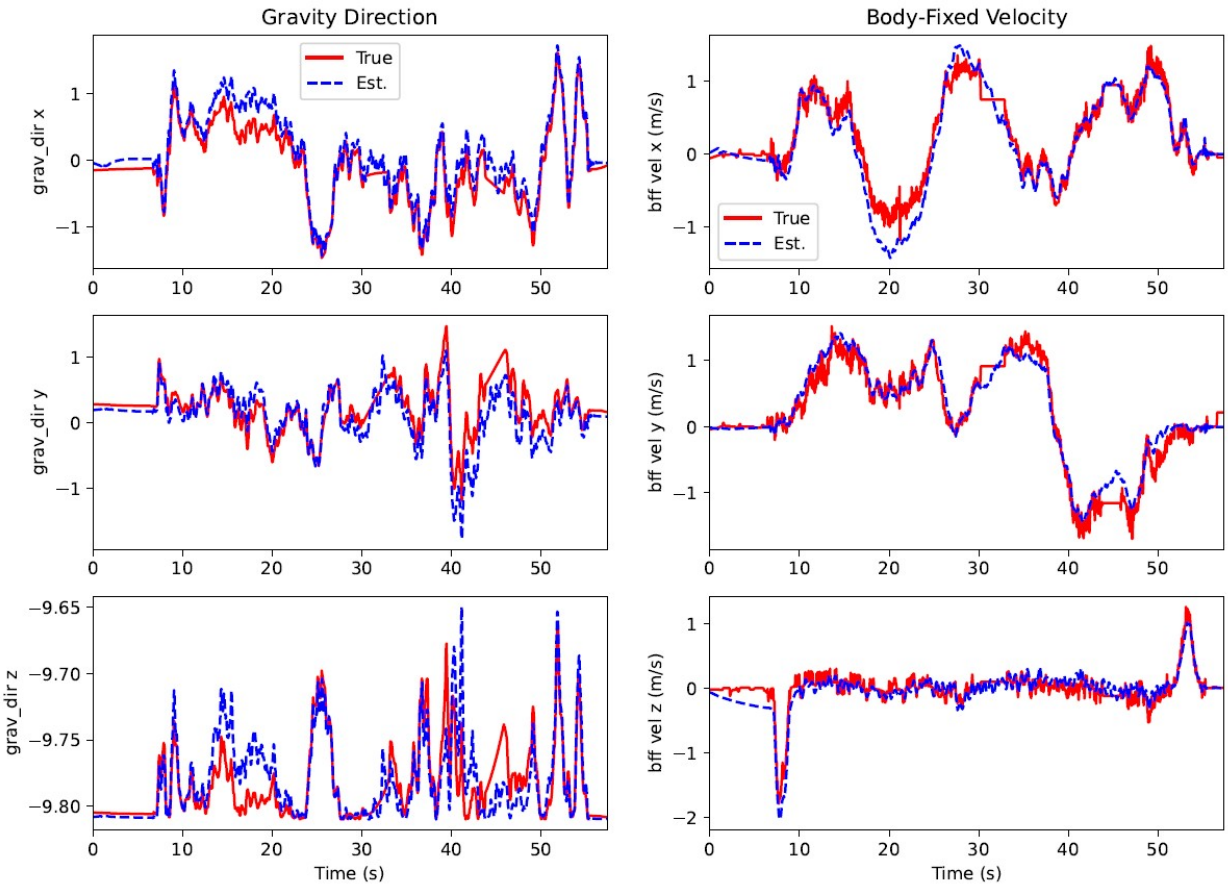


Figure 6.5: Estimate of gravity direction and body fixed velocity for tuned APVIO

We can see that the estimate of the gravity direction is accurately tracking the ground truth with only small deviations. Additionally our body fixed velocity is very accurate except for a couple of regions where the ground truth velocity become horizontal such as between 30 and 33 seconds. This has to do with the GPS signal dropping out due to trees in the area blocking it. These results however perform significantly better than those prior to config tuning which gives us the confidence to run the APVIO onboard the drone while it is in the air.

6.6 Testing the APVIO onboard an airborne drone

Three different experiments were conducted at the ANU Spring Valley farm with the APVIO running onboard an airborne quadcopter. The top down view of the three flights paths is shown in figure 6.6. The square path is designed to be a easy path with no yaw perturbation and simple 90 degree turns. The EqF skywriting is a more difficult path, still with no yaw perturbation but with more complex movements in the XY plane. The Lissajous path is a challenging path with reasonably complex movements in the XY plane as well as perturbation in the roll, pitch and yaw. Additionally it has a changing height above the ground throughout the flight. This makes it significantly more challenging and attempts to mirror AirSim path 4 as described in section 4.3.

(a) Square path (b) Eqf skywriting (c) Lissajous path

Figure 6.6: Flight paths used for real world APVIO testing

With only three flights flown the error metrics used in the AirSim section are less useful due to the fact that there are less data points to compare. Results are instead presented through graphs which compare the state estimate of the APVIO with the ground truth. This is done as the graphs provide a higher level of detail and allow for more meaningful analysis of each individual flight. The graphs for each path will contain the position and attitude in one graph and the gravity direction and body fixed velocity on another.

The position and attitude estimates of the square and EqF skywriting flight paths can be seen in figures 6.7 and 6.8 respectively. There exists some common behaviour between the output of the APVIO for these two paths. In general they are estimating the XY position of the drone to a fairly high level of accuracy in addition to accurately estimating the roll and pitch aside from a constant offset due to alignment of the position estimates after the fact. The yaw output for both paths seems to be relatively inaccurate which aligns with the results of the AirSim section.

Another interesting observation from the EqF flight occurs when the drone begins to descend at 140 seconds. At this point the estimates for all the position and attitude axis becomes very unstable resulting in divergence. This is interesting behaviour and strongly suggests that the APVIO's estimate of the Z axis is only just stable. When we began descending this movement was enough to cause divergent behaviour in the Z height estimate resulting in a poor estimate in the other axis. The APVIO and EqVIO by extension have previously been observed to have a poor Z height estimate, both in testing within AirSim and when testing the APVIO on the ground.

We can also consider the gravity direction and the body fixed velocity for the square and EqF flights. These are shown in figures 6.9 and 6.10 respectively.

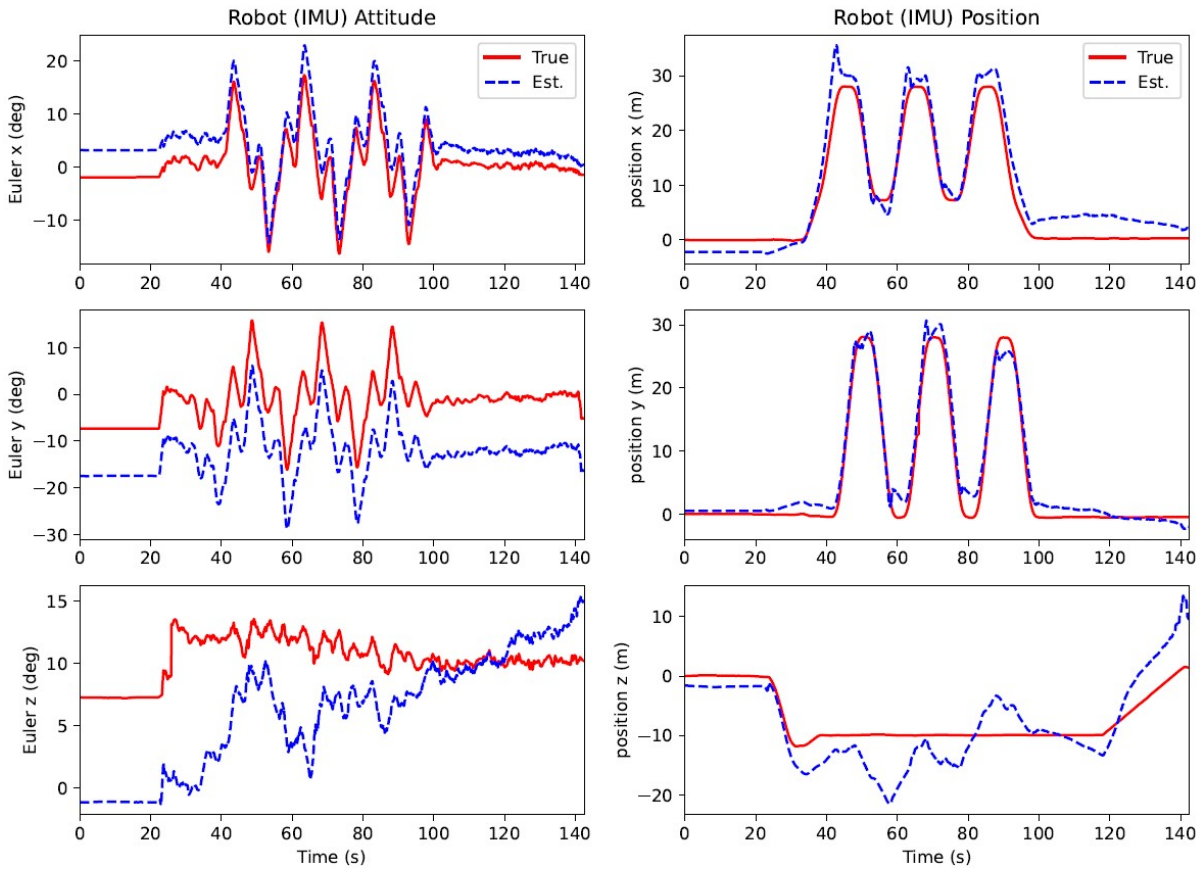


Figure 6.7: Pose estimate of square flight path for APVIO

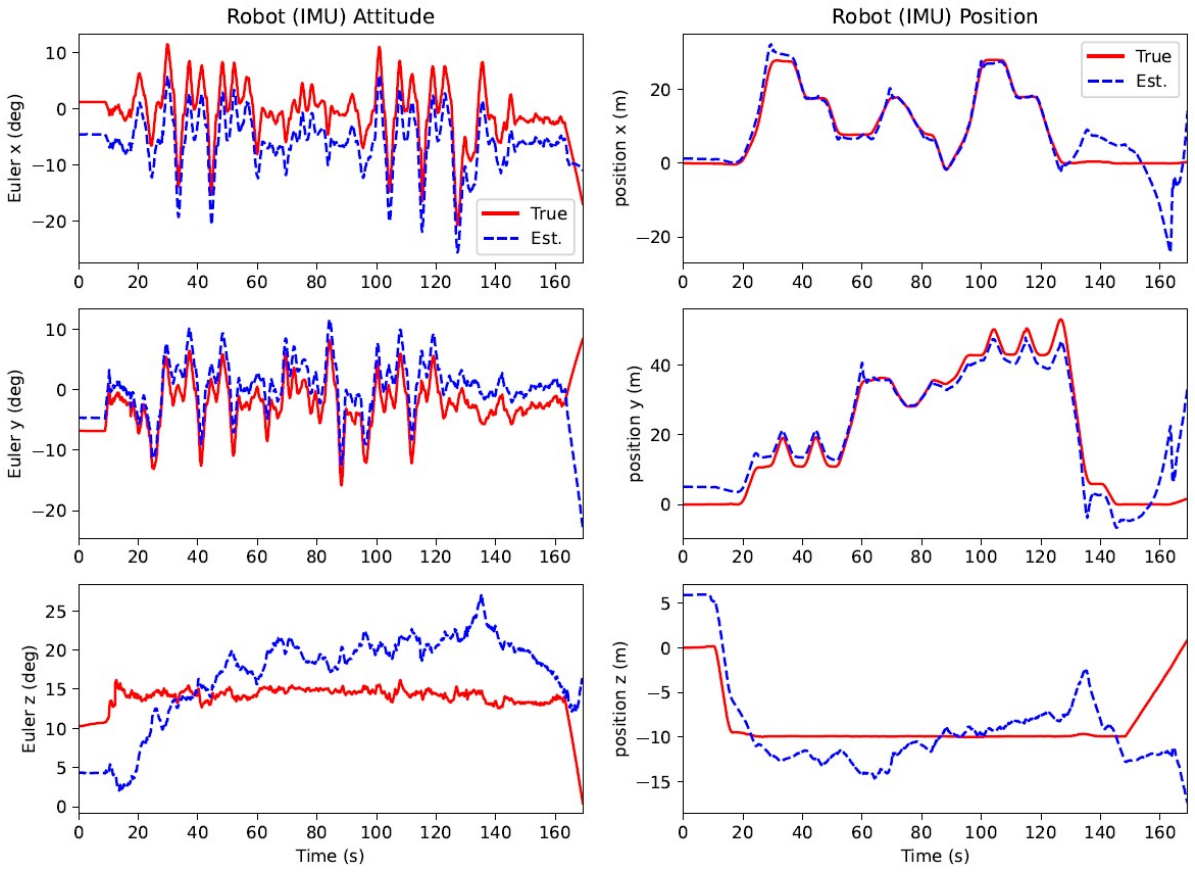


Figure 6.8: Pose estimate of EqF flight path for APVIO

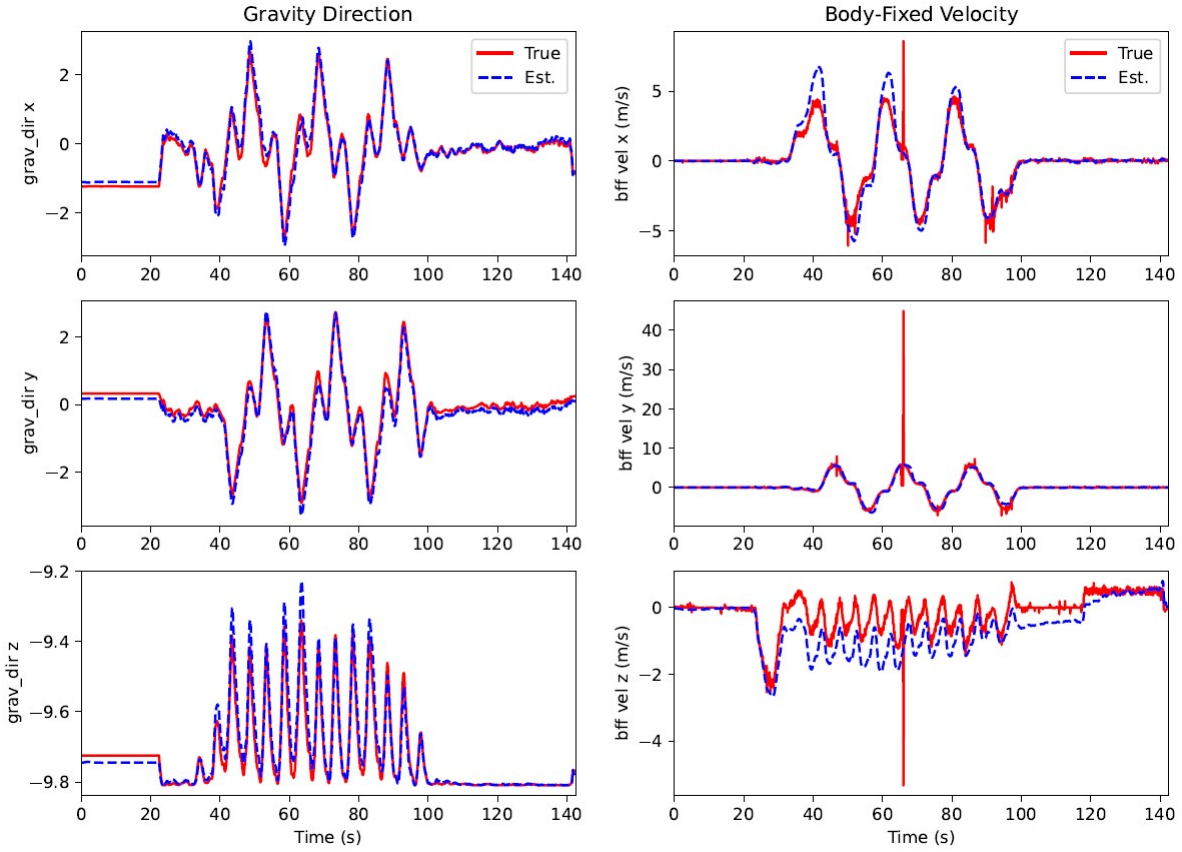


Figure 6.9: Gravity direction estimate of square flight path for APVIO

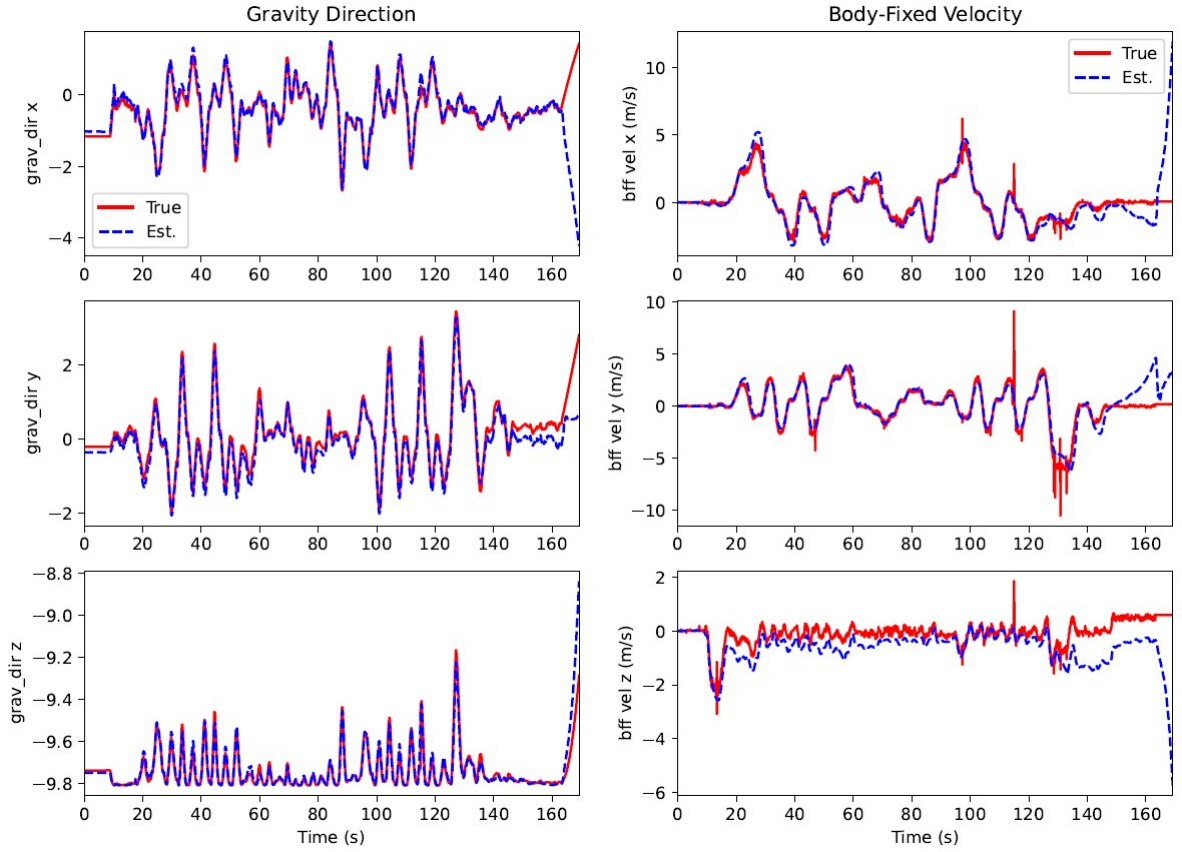


Figure 6.10: Gravity direction estimate of EqF flight path for APVIO

The APVIO seems to be achieving a very good gravity direction estimate for both flights. This is a promising result as the gravity direction is one of the observable parameters so an accurate estimate is expected. It also indicates that the attitude estimate in the roll and pitch axis prior to alignment with the direction of minimum position error was very accurate as the gravity direction is strongly correlated with these axis. The APVIO is also providing a reasonable estimate of the body fixed velocity especially in the XY plane. The Z velocity is significantly less accurate in both cases.

The final flight path which was recorded at Spring Valley was a Lissajous path which has a changing height above the ground. As shown in figures 6.11 and 6.12 the Lissajous path diverged very quickly. The positional axis all show large errors over the entire flight path as does the body fixed velocity which shows an interesting alternating pattern. Due to the position diverging the attitude of the system which is aligned to minimise the positional error has been aligned to some unknown orientation making it very hard to interpret. Given the poor estimate of the gravity direction it is likely that the attitude estimate was also very poor.

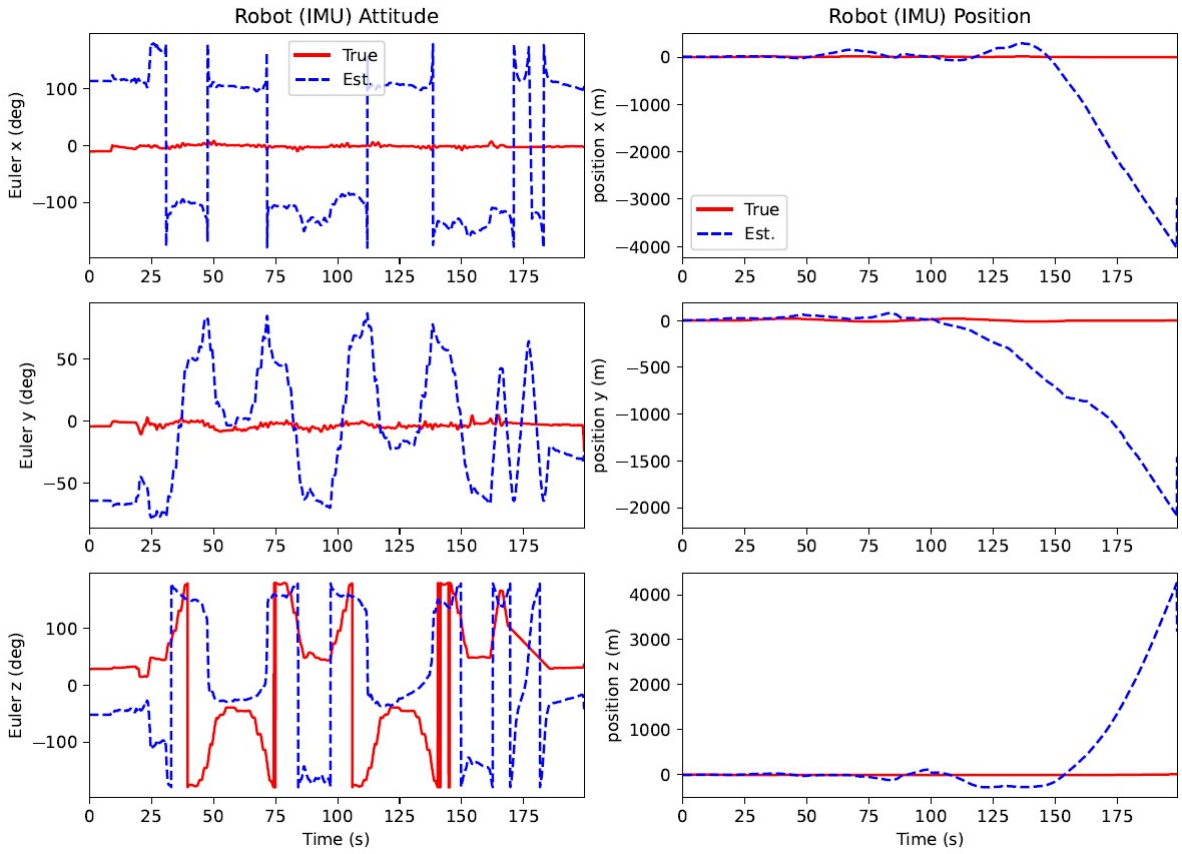


Figure 6.11: Pose estimate of Lissajous flight path for APVIO

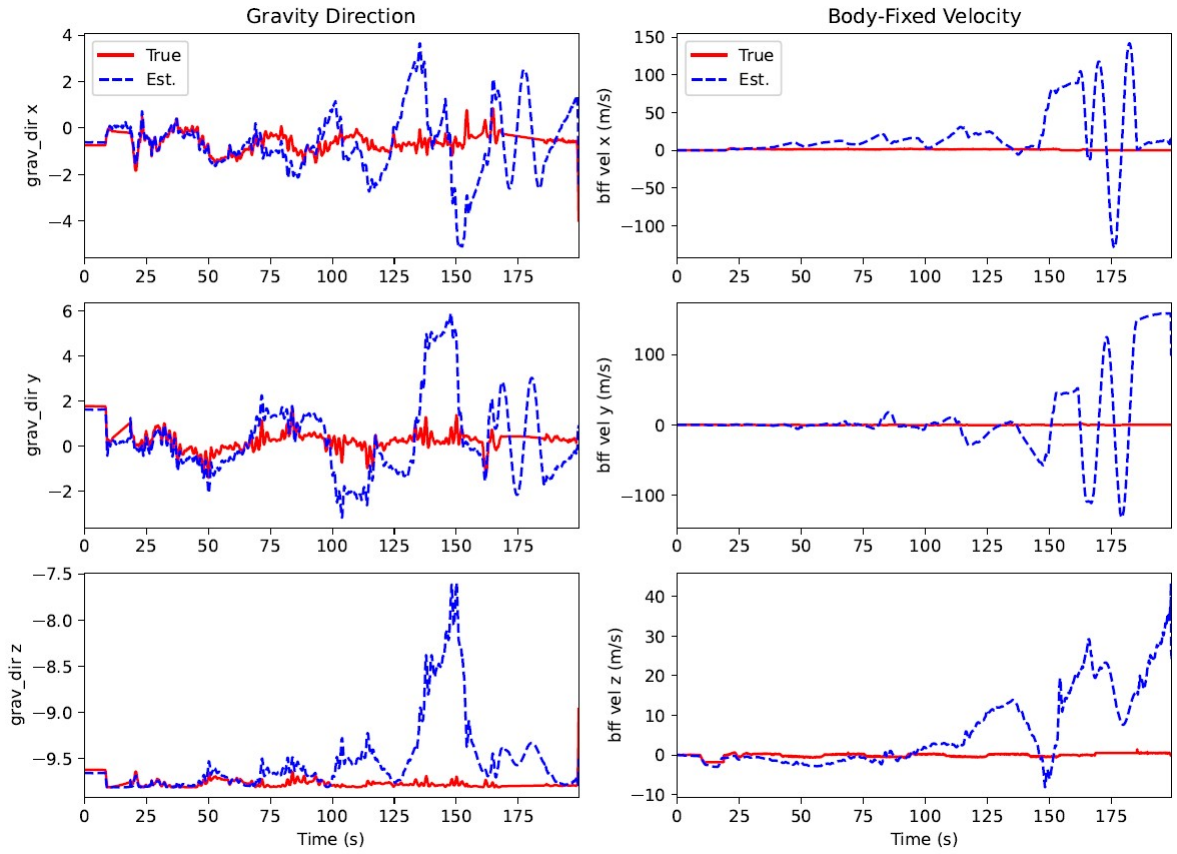


Figure 6.12: Gravity direction estimate of Lissajous flight path for APVIO

While some paths in this section have shown mixed performance the majority of data recorded has shown good accuracy in all of our estimated parameters. Additionally as the output of the APVIO is now matching the output of the EqVIO we can be fairly confident that further work in parameter tuning will result in improved performance. The fact that the square and EqF flight paths which did not have changing heights converged while the Lissajous path with changing height diverged indicates that the APVIO currently struggles to accurately estimate the state of the system when the height above the ground is changing.

A future project could investigate integrating sensors such as barometers and magnetometers into the APVIO to improve the state estimate. A barometer would provide a measurement of atmospheric pressure which can be used to either estimate the height above sea level or the difference in pressure could be used to get an approximate change in height. The magnetometer could be used to provide a physical interpretation for yaw as aligning with the direction of magnetic north. While the specific implementation of both sensors would need to be investigated the addition of these sensors would likely improve the Z height and yaw estimates which are currently the weakest elements of the APVIO. Both sensors are also very common and are integrated into many drones already, so the addition of these sensors would not significantly reduce the accessibility of the APVIO to the wider community.

6.6.1 Passing the APVIO estimate to ArduPilot

The long term goal of the APVIO is to pass the current state estimate back to the ArduPilot flight controller to allow navigation when no GPS is present. This can be done through the MAVLink protocol using the "*vision_position_estimate*" and "*vision_speed_estimate*" messages. This is implemented within the APVIO so by looking at the ArduPilot logs we are able to check if ArduPilot is receiving these messages. The VISP and VISV channels of the logs correspond to the respective position and velocity estimates of the APVIO. A plot of the "*vision_position_estimate*" messages is shown in figure 6.13 through the VISP channel.

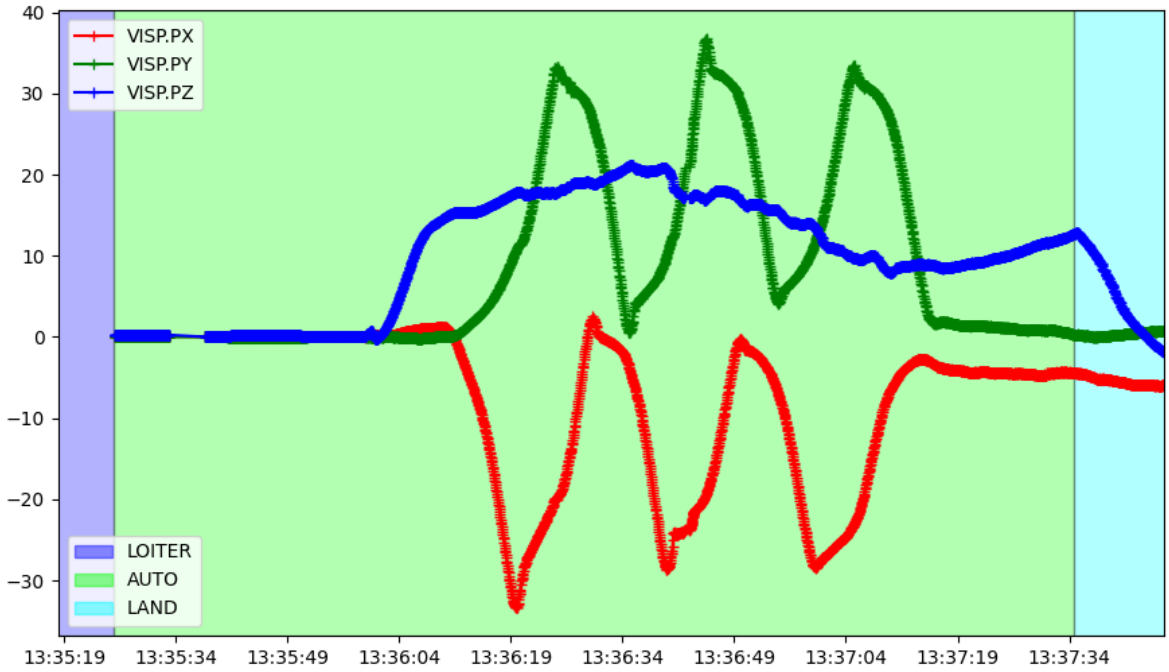


Figure 6.13: APVIO position estimate passed to ArduPilot

We can see a number of issues with the current implementation from figure 6.13. The first is that the results are being passed in the wrong reference frame. This can be noted as the X and Y positions have been swapped and the Z axis is the negative of what it should be. This is due to the fact that ArduPilot uses NED coordinates while the EqVIO uses ENU coordinates.

The other major issue is that given we are flying in a square path we would expect to see a path that only changes in X or Y at any given time. However, we can see that both the X and Y axis are changing at the same time. This might indicate that the attitude within the EqVIO has a different origin to the attitude in ArduPilot resulting in poorly aligned results. A possible fix to this is to initialise the APVIO attitude estimate at ArduPilot's attitude estimate to ensure the same initial orientation of the drone. These result show that while we are able to pass the state of the system to ArduPilot, more work needs to be completed to fully integrate the APVIO with ArduPilot.

Conclusion and Future Work

From chapter 6 we can see that the EqVIO has been successfully implemented in real time on board a drone while running on a Raspberry Pi. On easier paths the performance of the system is very good with it accurately estimating the X-Y position as well as the roll and pitch. On harder paths with changing height above the ground the APVIO diverges. This indicates a lack of robustness for movements where the height above the ground changes supporting results from chapter 4.

Chapter 4 showed that data generated through simulation via AirSim could be used to run the EqVIO. The accuracy of the EqVIO within this section was worse than real time results from section 6 on similarly difficult paths, likely indicating that simulation is not as accurate as real world flights. Using simulation we were able to predict the behaviour of real world flights prior to indicating that some understanding of real world systems can be gained through simulation.

An additional feature was added to the EqVIO in the form of an attitude innovation within chapter 5. The EqVIO showed a significant improvement in performance when using the attitude innovation if GPS was used within ArduPilot's attitude estimate. While the attitude innovation can therefore only be used when GPS is available, a use case was proposed where the attitude innovation is used to initialise the IMU biases and camera position.

While this project has achieved the goals set out by the research questions, a significant amount of work still needs to be completed for the APVIO to be widely used within the ArduPilot community. Future work should include fixing data logging within the APVIO to allow for recording of data sets to be run offline. The performance of the APVIO can then be optimised on these offline data sets much more easily then during real world flights. The initialisation procedure should also be improved to ensure that the APVIO is initialised in the same state as ArduPilots EKF. This could be done by expanding the attitude innovation to include innovations for position and velocity that would run for a short period after start up or when GPS is available. Finally the addition of sensors such as barometers and magnetometers could significantly improve the reliability and overall performance of the system.

Appendix A

8.1 The AirSim settings file

An example of a settings file containing many of the common settings modified throughout this project is shown in listing 8.1.

Listing 8.1: Example of AirSim settings for connecting to ArduPilot

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "ClockSpeed": 1,

  "Recording": {
    "RecordInterval": 0.05,
    "Folder": "",
    "Enabled": false,
    "Cameras": [
      {
        "CameraName": "0",
        "ImageType": 0,
        "PixelsAsFloat": false,
        "VehicleName": "",
        "Compress": true
      }
    ]
  },

  "Wind": {
    "X": 0,
    "Y": 0,
    "Z": 0
  },

  "Vehicles": {
    "Copter": {
      "VehicleType": "ArduCopter",
      "UseSerial": false,
      "LocalHostIp": "10.0.0.128",
      "UdpIp": "10.0.0.1",
      "UdpPort": 9003,
      "ControlPort": 9002
    }
  },

  "CameraDefaults": {
    "CaptureSettings": [
      {
        "ImageType": 0,
        "Width": 1280,
        "Height": 720,
        "FOV_Degrees": 90,
        "AutoExposureSpeed": 100,
        "MotionBlurAmount": 0
      }
    ]
  },

  "OriginGeopoint": {
    "Latitude": 47.641468,
```

```
    "Longitude": -122.140165,
    "Altitude": 122
  }
}
```

Other settings which have not been shown include the ability to modify individual sensor accuracy as well as time of day settings. The type of camera shown in sub windows can also be changed through the settings. While many of the settings are easy to understand or have existing descriptions [38], more detail is provided in table 8.1 around particular settings which were used within this work.

AirSim command	Explanation
SimMode	The type of vehicle being used in the simulation. To start the simulation using a quadcopter use *SimMode = Multirotor*.
ClockSpeed	This determines the rate that times passes in the simulation as compared to a wall clock. If this is set to one then time passes at the same rate in simulation as on a wall clock, while if it is less than one, then time passes slower in the simulation. This is sometimes useful when running on slower computer hardware and can also improve the quality of the simulation.
Recording:RecordInterval	This is the rate at which all data is saved when recording and is the number of miliseconds between two measurements. This applies to both images and IMU data meaning that if two different rates of data are required for example 100Hz IMU data and 20Hz camera images then either an API or flight controller must be used.
Recording:Enabled	This determines if the recording should start as soon as the simulation starts or until *R* is pressed.
Recording:Cameras	This determines which cameras are used to capture images when recording occurs.
Vehicles:Copter	This is the class of vehicle that will be generated in simulation.
Vehicles:Copter: VehicleType	This controls the type of vehicle generated and how it will be controlled. The two common values are *SimpleFlight* which will act like a basic drone controlled by a joy stick or *ArduCopter* which will allow the drone to be controlled by ArduPilot.
CameraDefaults: CaptureSettings	This is the default camera that will be shown and unless otherwise specified recorded with.
CameraDefaults: CaptureSettings: ImageType	This value determines the type of camera used. Possible options can be found at [39].

CameraDefaults: CaptureSettings: Width/Height	This value determines the resolution of the camera.
OriginGeopoint	This determines the starting Geopoint of the simulation. This can be left as a default value unless an environment is being used which represents a real world location.

Table 8.1: Explanation of common AirSim settings

Bibliography

[1] Pieter van Goor and Robert Mahony. *EqVIO: An Equivariant Filter for Visual Inertial Odometry*. 2022. DOI: 10.48550/ARXIV.2205.01980. URL: https://arxiv.org/abs/2205.01980.

[2] Meta. *Powered by AI: Oculus Insight*. 2019. URL: https://ai.facebook.com/blog/powered-by-ai-oculus-insight/.

[3] David Bayard et al. 'Vision-Based Navigation for the NASA Mars Helicopter'. In: Jan. 2019. DOI: 10.2514/6.2019-1411.

[4] Jeff Delaune et al. 'Extended Navigation Capabilities for a Future Mars Science Helicopter Concept'. In: *2020 IEEE Aerospace Conference*. 2020, pp. 1–10. DOI: 10.1109/AERO47225.2020.9172289.

[5] J.R. Carpenter and C.N. D'Souza. *Navigation Filter Best Practices*. Tech. rep. NASA, 2018. URL: https://ntrs.nasa.gov/api/citations/20180003657/downloads/20180003657.pdf.

[6] Feng Zheng et al. 'Trifo-VIO: Robust and Efficient Stereo Visual Inertial Odometry Using Points and Lines'. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 3686–3693. DOI: 10.1109/IROS.2018.8594354.

[7] Guang Yang et al. 'Optimization-Based, Simplified Stereo Visual-Inertial Odometry With High-Accuracy Initialization'. In: *IEEE Access* 7 (2019), pp. 39054–39068. DOI: 10.1109/ACCESS.2019.2902295.

[8] Huanyu Wen, Jindong Tian and Dong Li. 'PLS-VIO: Stereo Vision-inertial Odometry Based on Point and Line Features'. In: *2020 International Conference on High Performance Big Data and Intelligent Systems (HPBDIS)*. 2020, pp. 1–7. DOI: 10.1109/HPBDIS49115.2020.9130571.

[9] Ch. Doer et al. 'Inertial sensor data based motion estimation aided by image processing and differential barometry'. In: *2018 25th Saint Petersburg International Conference on Integrated Navigation Systems (ICINS)*. 2018, pp. 1–10. DOI: 10.23919/ICINS.2018.8405839.

[10] Jingzhe Wang et al. 'VIMO: A Visual-Inertial-Magnetic Navigation System Based on Non-Linear Optimization'. In: *Sensors* 20.16 (2020). ISSN: 1424-8220. DOI: 10.3390/s20164386. URL: https://www.mdpi.com/1424-8220/20/16/4386.

[11] Hauke Strasdat, J. M. M. Montiel and Andrew J. Davison. 'Real-time monocular SLAM: Why filter?' In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 2657–2664. DOI: 10.1109/ROBOT.2010.5509636.

[12] Jeffrey Delmerico and Davide Scaramuzza. ‘A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots’. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 2502–2509. DOI: 10.1109/ICRA.2018.8460664.

[13] Stefan Leutenegger et al. ‘Keyframe-Based Visual-Inertial Odometry Using Nonlinear Optimization’. In: *The International Journal of Robotics Research* 34 (Feb. 2014). DOI: 10.1177/0278364914554813.

[14] Tong Qin, Peiliang Li and Shaojie Shen. ‘VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator’. In: *IEEE Transactions on Robotics* 34.4 (2018), pp. 1004–1020. DOI: 10.1109/TRO.2018.2853729.

[15] Michael Bloesch et al. ‘Robust visual inertial odometry using a direct EKF-based approach’. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 298–304. DOI: 10.1109/IROS.2015.7353389.

[16] S.J. Julier and J.K. Uhlmann. ‘Unscented filtering and nonlinear estimation’. In: *Proceedings of the IEEE* 92.3 (2004), pp. 401–422. DOI: 10.1109/JPROC.2003.823141.

[17] John Crassidis, Landis Markley and Yang Cheng. ‘Survey of Nonlinear Attitude Estimation Methods’. In: *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM* 30 (Jan. 2007), pp. 12–28. DOI: 10.2514/1.22452.

[18] John Crassidis. ‘Unscented Filtering for Spacecraft Attitude Estimation’. In: *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM* 26 (July 2003). DOI: 10.2514/2.5102.

[19] Silvère Bonnable, Philippe Martin and Erwan Salaün. ‘Invariant Extended Kalman Filter: theory and application to a velocity-aided attitude estimation problem’. In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. 2009, pp. 1297–1304. DOI: 10.1109/CDC.2009.5400372.

[20] Pieter van Goor, Tarek Hamel and Robert Mahony. *Equivariant filter (EqF): A general filter design for systems on homogeneous spaces*. 2021. DOI: 10.48550/ARXIV.2107.05193. URL: https://arxiv.org/abs/2107.05193.

[21] Pieter van Goor et al. ‘A Geometric Observer Design for Visual Localisation and Mapping’. In: *2019 IEEE 58th Conference on Decision and Control (CDC)*. 2019, pp. 2543–2549. DOI: 10.1109/CDC40024.2019.9029435.

[22] ETHZ ASL. *IMU Noise Model*. 2022. URL: https://github.com/ethz-asl/kalibr/wiki/IMU-Noise-Model#additive-white-noise.

[23] ‘IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Interferometric Fiber Optic Gyros’. In: *IEEE Std 952-1997* (1998), pp. 1–84. DOI: 10.1109/IEEESTD.1998.86153.

[24] Marin B. Marinov et al. ‘Analysis of Sensors Noise Performance Using Allan Deviation’. In: *2019 IEEE XXVIII International Scientific Conference Electronics (ET)*. 2019, pp. 1–4. DOI: 10.1109/ET.2019.8878552.

[25] Pieter van Goor. *compute$_a$llan$_d$eviation*. 2021. URL: https://github.com/pvangoor/junkcode/blob/master/converters/compute_allan_deviation.py.

[26] OpenCV. *Fisheye camera model*. 2022. URL: https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html#details.

[27] J. Kannala and S.S. Brandt. 'A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.8 (2006), pp. 1335–1340. DOI: 10.1109/TPAMI.2006.153.

[28] ETHZ ASL. *Kalibr*. 2022. URL: https://github.com/ethz-asl/kalibr.

[29] ETHZ ASL. *Kalibr*. 2022. URL: https://github.com/ethz-asl/kalibr/wiki/calibration-targets.

[30] Shital Shah et al. 'AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles'. In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[31] AirSim. *FAQ*. 2021. URL: https://microsoft.github.io/AirSim/faq/#what-computer-do-you-need.

[32] AirSim. *Tips for Busy HDD*. 2021. URL: https://microsoft.github.io/AirSim/hard_drive/.

[33] AirSim. *Development Workflow*. 2022. URL: https://microsoft.github.io/AirSim/dev_workflow/.

[34] ArduPilot. *Using SITL with AirSim*. 2022. URL: https://ardupilot.org/dev/docs/sitl-with-airsim.html.

[35] Wikipedia. *Random walk*. 2022. URL: https://en.wikipedia.org/wiki/Random_walk.

[36] Wikipedia. *Brownian motion*. 2022. URL: https://en.wikipedia.org/wiki/Brownian_motion.

[37] Paul Silisteanu. *Raspberry Pi - Install GCC 10 and compile C++17 programs*. 2017. URL: https://solarianprogrammer.com/2017/12/08/raspberry-pi-raspbian-install-gcc-compile-cpp-17-programs/.

[38] AirSim. *AirSim Settings*. 2022. URL: https://microsoft.github.io/AirSim/settings/.

[39] AirSim. *Image APIs*. 2022. URL: https://microsoft.github.io/AirSim/image_apis/#available-imagetype-values.