An application data fusion technique in indoor flight of drones

Tianfang Sun

August 28, 2018

Abstract

In order to measure the position of an indoor unmanned aerial vehicle (UAV), A combined position estimate system is designed and tested in this thesis. Two sensors, an optical flow sensor (Px4flow) and a ultrasonic sensor (Marvelmind), are employed to provide the raw measurement data. The extended Kalman filter (EKF) is used as the main data fusion method and some algorithms are built based on this, including the attitude estimation, angular rate compensation and the position estimation. A single board computer (Raspberry Pi) is employed and plays a role as the central controller. Matlab, C language and Python are used to produce all the programs. Some simulation and on-board tests are performed and will be discussed in the thesis as well.

By analysing the test result it can be proved that this system is capable of providing an accurate position estimate (with standard deviation of 0.0275 meter) at a high update frequency (100 Hz).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and background

As the development of electronic technology, algorithms need heavy compute capability which used to performed on a professional workstation can now be carried out on a compact embedded system. This encourages lots applications to be minimized in order to fit some special usages. Among these, the indoor Unmanned Aerial Vehicle (UAV) is one of the most investigated area in the past decade. Nowadays indoor UAV is not only the high-tech toys, but also employed in civilian or military field, such as delivery, search and rescue, explore and investigate, etc.

Evolve from traditional unmanned aerial vehicles, the indoor UAV can use the navigation mechanism inherited from those studied well in the traditional ones. However, the complex working environment of the indoor UAV requires a much higher navigation accuracy and update frequency. Compare to the out door UAV, which relies heavily on Global Navigation Satellite System (GNSS) signal to perform a locating, the indoor UAV suffers from the attenuation of signal[14]. Moreover, the accuracy of GNSS, which is around several meters, can not satisfy the high obstacle density of an indoor environment. A navigation method, with an accuracy of several centimetres should be used for the indoor UAV. Many studies on high accuracy indoor-navigations based on different sensors have been carried out these years, as we will see shortly in section 1.2.

The purpose of this thesis is to build a data fusion system, which employs several navigation sensors and merge their outputs together, in order to produce a position estimate with high accuracy and output frequency. Two sensors, an ultrasonic beacon sensor called Marvelmind and an optical-flow sensor called Px4flow are used in this project. The data from these two sensors will be fused together using the extended Kalman filter (EKF). The capability of this system will be examined by comparing the position estimate from the data fusion system to the original output from the two sensors.

1.2 Summary of UAV

The history of Unmanned Aerial Vehicle can trace back to the World War I. In 1916, the Ruston Proctor Aerial Target, using radio control techniques, is recorded as the first non-pilot aircraft[17]. During the period of World War II, UAV has been considerably well studied since the ability of remote control makes it the idea weapon for attacking important targets. For instance, the Fritz X, which is a German guided glide bomb, can also be considered as a UAV, although they are one-time used and not designed to be recoverable. However the majority usage of the UAV in this period is used as the dummy target. Radioplane OQ-2, manufactured by the Radioplane Company in the United States, was a massive produced UAV during World War II. It is used by the army to train the pilot's shooting skills and over 9,400 of them have been built in the war.

However the UAV in nowadays is much different with their early parents. The UAV in modern days is capable of delivery, search, news reporter and many other usages. And they are not only remote controlled, but also autonomous. In [4], UAV is classified by four criterion: size, mean take off weight (MTOW), operation altitude and autonomy. Figure 1.1 shows a brief category of this. On

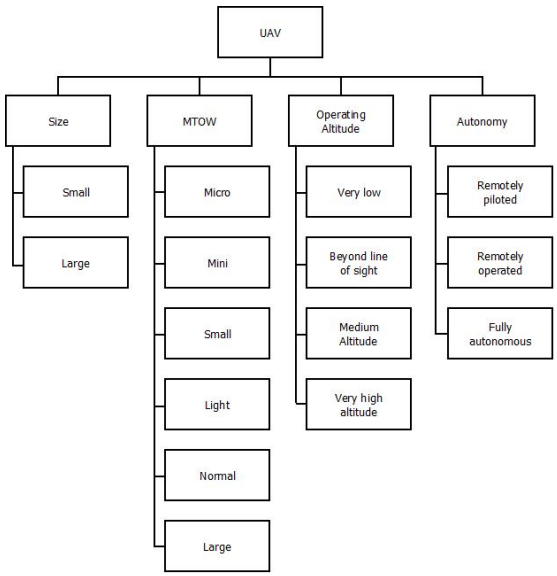


Figure 1.1: Classification of UAV

the other hand, the UAV can also be classified into fixed-wing and rotary-wing by their flying principles[6]. Generally speaking, the fixed-wing UAV has simpler structure, longer flight duration and higher speed compare to the rotary-wing UAV. In contrast, the rotary-wing UAV is capable of performing complex flight actions such as hovering, high-G rotation, vertical takeoff and landing (VTOL),

etc.

Following above classifications, the indoor UAV could usually be categorized into a small sized, micro to small MTOW, very low altitude UAV, and usually is developed into a specific subclass of the rotary-wing UAV, the quadrotor helicopter or quadrotor for short. However the problems we are facing to develop a small sized UAV do not minimized together with the size. In contrast, indoor UAV faced some special challenges due to the complex flight environment.

1.2.1 Quadrotor architecture and indoor challenges

Two majority problems on developing a indoor UAV is the controlling and the navigation. Firstly, the flight principle of a quadrotor UAV is much different with the traditional fixed-wing vehicle or helicopter. The quadrotor has four motors with propellers as its driver, which is shown in figure 1.2.[16] The lift force from propeller and the torque from motor are both used as a parameter to control the speed and attitude of the UAV, so there are total of eight control parameters in a quadrotor. As a consequence, the mathematical model for the control system, such as a PID controller, will be complex and requires heavy computation payload.

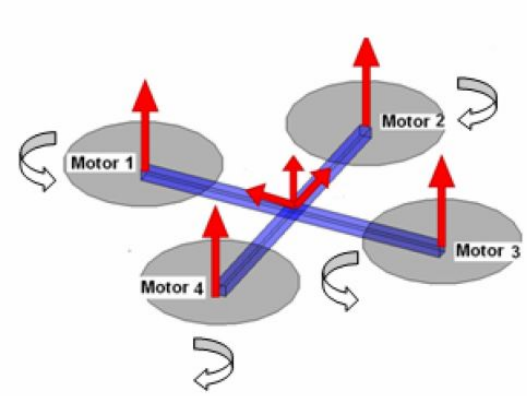


Figure 1.2: Illustration of Quadrotor

Secondly, since the indoor UAV usually working in a environment of high obstacle density, together with some unexpectable situations such as human activities, other UAV movement, occasionally open/close of the door and windows, the requirement of the navigation system is essential. The GPS signal, which is a common locating method appears in outdoor UAV, can hardly fulfil such requirements. This is because, first, the indoor environment will usually cause an attenuation of the GPS signal. This attenuation is caused by the severe multi-path effect due to the walls or roofs (20-30 dB compared to outdoor situation).[13] Second, the accuracy of the GPS, which is about several meters,

is not accurate enough for the indoor environment. In a practical application, an accuracy of several centimetres is usually expected. As a result, it is obligate to find an accurate and high update frequency navigation architecture which suitable for the complex flight environment of the indoor UAV.

1.2.2 Navigation sensor investigation

In the past several years, lots of studies have been performed on the indoor navigation area. Table 1.2 shows a brief summary of the past five years.[12] Based on the measurement method, the navigation sensors used in 1.2 can be catalogued into four classes: ultrasonic, inertial, optical flow, and Lidar. If we consider the estimate principle, then these sensors can be classified into two types: position estimation and velocity estimation. Here we shall give a comparison between them.

We will begin with the catalogue on the measurement method. The ultrasonic sensor sends ultrasonic pulse towards a specific direction and measure the reflected pulse. By compare the time interval between the sending and receiving beams, this type of sensor can estimate the distance of an obstacle before it. Some ultrasonic sensors compare the phase shift between the two pulses and by employ the Doppler effect they can estimate the target velocity at the direction of the ultrasonic beams. However in the indoor UAV area, most ultrasonic sensors are the former type, which is directly measure the absolute distance between the sensor and an obstacle. The advantages of this kind of sensor are light-weighted ($10g$), low power consumption ($\sim 50mW$) and low costs. On the other hand, the working range of ultrasonic sensors is typically small. The minimum measuring range is about $30cm$ and the maximum range is usually several tens of meters. This minimum range rises a restriction when using the ultrasonic sensors to measure the altitude of the UAV since a main feature of the indoor quadrotor UAV is their VTOL capability and the limitation of this minimum measurement range makes the ultrasonic sensor useless in this scenario.

The majority used inertial sensor in the UAV area is the Inertial Measurement Unit (IMU), which is also a very widely used type of sensor in the aero/navy/satellites fields. The IMU employs the accelerometer and gyroscope to measure the acceleration and angular rate. Some IMUs also integrates a magnetometer (compass) in order to measure the heading of the sensor, although a compass does not follow the word "inertial". The advantage of the IMU is the measurement does not reply on any outside input (despite of the compass), and they typically have a high output frequency. However, the IMU is mostly used to estimate the attitude of the UAV, rather than using it to measure a position. In order to measure a position, we have to double integral the acceleration measured by the IMU, which will cause a significant drift error. So in practical, the IMU is used together with some other types of sensors, which require an attitude data for position estimate.

Next we take a look at the optical flow sensor. The optical flow sensor can be regarded as a low pixel digital camera, which keeps taking images from the direction of it's facing. By comparing the difference between the images, the

Team	Institution	Navigation Architecture	Year
Sungsik Huh	KAIST	IMU/Camera/Laser scanner	2013
Nils Gageik	University of Würzburg	IMU/Infrared/Optical flow(ADNS3080)	2013
Korbinian Schmid	DLR	IMU (Analog Devices ADIS16407) /Stereo Camera (PointGrey Firefly FMVU-03MTM)	2014
Jin Q. Cui	National University of Singapore	IMU/Lidar (UTM-30LX&URG-04LX) /Camera	2015
Dong Ki Kim	Cornell University	IMU/Ultrasonic altimeter/Optical flow/Camera (Parrot Bebop)	2015
Shaima Al Habsi	UAE University	IMU/Ultrasonic altimeter/Vicon Mo-cap	2015
Chong Shen	North University of China	IMU (MPU6050) /Optical flow + Ultrasonic (PX4Flow)/Magnetometer (HMC3883L)	2016
Yu Zhang	NUAA, China	IMU/Optical flow + Ultrasonic (PX4Flow) /Camera (UI-1221LE)	2016
Kang Li	Chinese Academy of Sciences	IMU/UBW (DW1000) /laser scanner	2016
Aiden Morrison	Norwegian Defence	IMU (MPU9250/Mti-100) /Optical flow + Ultrasonic (PX4Flow) /Vision (Sony IMX250)	2017
Kimberly McGuire	TU Delft	IMU/Optical flow /Stereo camera	2017
Elena López	Alcalá University	IMU/Scanner/Monocular camera	2017
Zhou Qiang	Shanghai Jiao Tong University	IMU/Ultrasound/Stereo camera/Motion Tracking	2018
Mattew A Copper	Air Force Institute of Technology	IMU/Scanner/motion tracking	2018

Table 1.2: Recent research on indoor UAV navigation architecture

sensor output how much pixels flow-by during the time interval between two images, which is called "optical flow". Combining some distance measurement sensor, this optical flow data can be converted into the ground velocity. The most common example of an optical flow sensor is the optical mouse which is used as an input device for computer. Some modern optical flow sensor employs a 3D reconstruction algorithm so the sensor can also provide an estimate in the azimuth direction.[10] The optical flow sensor attracts much research attentions in recent years but it has some disadvantages for now. Firstly the algorithm used to compute the optical flow data, either 2D or 3D, requires a high computational capabilities. This usually leads to a high power consumption and higher unit costs. Secondly as the optical flow sensor based on the digital camera, the working environment should be illuminated well. Although some recent research employs more sensitive camera which reduces the illumination requirement[9], the limitation still exists.

The Lidar sensor, or Light Detection and Ranging sensor, use laser beams to measure the distance between the sensor and surrounding obstacles, which follows similar mechanism as the ultrasonic sensor. In the indoor UAV area, since a laser emitter usually comes with a large size, the infra-red (IR) focused LED emitter is often employed. Focused IR LED can produced a highly directional IR beam which fulfils the accuracy requirement of the indoor UAV and keeps the size of the sensor sufficiently small. Lidar sensor used in the indoor UAV area an be divided into two types: rangefinder and scanner. The rangefinder type is usually used as an advanced replacement for the ultrasonic sensor, since they have better working range (up to $100m$) and higher accuracy. The scanner type place the Lidar sensor on a wheel which driven by a motor. While the wheel is rotating, the sensor keeps scan the surrounding environment and produces a 2D/3D point cloud data. This data can be used as a map for the indoor UAV to avoid the obstacles in its flight path. However the scanner type sensor is expensive than the rangefinder, which could cost from server thousands euro up to $30,000€$. As a conclusion, table 1.4 shows a brief comparison between different sensors.

Before we leaving this section, another comparison between the position estimation and the velocity estimation should be made. A position estimation sensor, such as the ultrasonic sensor, provide a straight forward estimate for position. However this usually requires the interact with surrounding environment, e.g. the ultrasonic pulse need to reflect on the obstacle surface and propagate back to the sensor. Any interference between the sensor and the obstacle will introduce a noise to the final estimation. So the absolute position estimation is usually "jamming", the output data could jump from one value to a distanced one. As a contrast, the velocity estimation sensors, e.g. the optical flow sensor, integral the velocity data to get a position estimation. This requires additional processing on the data but will usually provide a "smooth" result. However, since an integral must be done for this kind of sensor, the drift error rises. The goal for this thesis is trying to merge a position estimation sensor with a velocity estimation one, in order to produce an overall better estimate result.

	ultrasonic	inertial	optical flow	Lidar
pros	low cost, small size	high output frequency, do not rely on outside inputs	high output frequency	high accuracy
cons	low working range	drift error	high computational payloads, requires illumination	expensive

Table 1.4: Comparison between varies sensors for indoor navigation

1.3 Data fusion methods

The term "data fusion" is a very generalized and widespread concept. It can be expressed as a very low-level signal processing method such as a digital filter, but also as a high-level conceptual treatment, for example a classification algorithm or a decision strategy. There are many definitions for data fusion from different research fields. Here we use a well known one, which also appropriately fits the theme of this thesis, that is[7]:

> *data fusion techniques combine data from multiple sensors and related information from associated databases to achieve improved accuracy and more specific inferences than could be achieved by the use of a single sensor alone.*

Following this definition, our purpose to employ a data fusion method is to provide a better information, given the data from multiple sensors. The word "better" here shall not simply be expressed as "more accurate", "faster" or so. It can be expressed by any feature that the designer expects it to have. For example, a 3D recorder typically use two ordinary cameras taking images simultaneously and provide a three dimensional movie. In this scenario, it can hardly say either the result movie is more "accurate" of "faster" than the images from both cameras. But the 3D movie will be still regarded as a "better" one since it provides the feature of 3D vision that the designer wish it to have.

There are many ways to classify the available data fusion techniques. From [3], the classification methods have been summarized into five types:

- following the relations of the input data sources.
- classified by the input/output data types.
- depending on an abstraction level of the employed data.

- based on the different data fusion levels defined by the Joint Directors of Laboratories (JDL)
- according to the architecture type

Since this thesis will focus on one data fusion method, rather than comparing the different data fusion types, we will not provide a comprehensive review on all the classification methods. Here we just discuss the classification method following the relations of the input data sources.

According to [5], the data fusion techniques can be divided into three types according to the relations of the data sources:

complementary sensors provide information from different aspects of a target, then combine together and provide a generalized view of the target.

redundant sensors measure the same information of a target, by merging the data, the final information will be more confident than either sensors.

cooperative the information from different sensors is processed into a new type of information, which will be typically more complex than the original data.

To give a clearer illustration, here are some examples for each type. For the complementary data fusion type, suppose we have a three-axis accelerometer which can measure the direction of the earth gravity, and a three-axis compass which can measure the earth magnetic field. If we put the two sensors on a UAV, then the accelerometer can provide the pitch and roll angles by comparing the gravity on each of its axis. However the accelerometer cannot provide any information about the yaw angle since no matter how the UAV changes its heading at the yaw angle, the earth gravity will remain the same value at all the three axis of the accelerometer. Therefore we need another data source to provide the information about the yaw angle, i.e. the compass. Combining the data from accelerometer and compass, a comprehensive attitude information on pitch, roll and yaw angles will be provided. This the so called complementary data fusion.

For the redundant type, we will directly use the main content in this thesis as an example. Suppose we have an ultrasonic sensor and an optical flow sensor which are discussed in section 1.2. The ultrasonic sensor will directly measure the position of a UAV while the optical flow sensor will measure the velocity. However, in this example, we would integral the data from the optical flow sensor, so the optical flow sensor by its own will also provide a position information of the UAV. In this case, two sensors will both provide data about the position of the vehicle. By some data fusion algorithm, we can merge the two data together and produce an overall more accurate position estimate than each of the single sensor. In this specific example, the redundant information of the position improves our confidence of the position estimate.

As an simple example for the cooperative type, we consider the 3D recorder example again. Each of the ordinary camera in the 3D recorder will just produce

a 2D image. But by processing the images from different cameras, a 3D vision has been provided. This result is much different from any of the images since it contains information of an extra dimension. And this result cannot be achieved by any of the single ordinary camera. Two cameras must "cooperatively" work together to produce the final 3D vision.

We will see in the following chapters that, the data fusion type we choose in this thesis, i.e. the Kalman filter, is a redundant data fusion method. From the point of view of the Kalman filter, which will be discussed in section 3.3.1, one data source will be regarded as an "estimator" and the other one is the "observer". The estimator will keeps estimating a target information, and whenever the data from the observer is available, the data from the estimator will be corrected by the data from observer.

1.4 Outline of this thesis

As the last section of the introduction, the outline of this thesis will be drawn here. This thesis illustrates a combined position estimate system, which employs a ultrasonic sensor Marvelmind and a optical flow sensor Px4flow. The extended Kalman filter is used as the main data fusion algorithm and all the computing is performed on a single-board computer Raspberry Pi. The goal of this system is to improve the overall position estimate accuracy by fusion the data from two sensors. Figure 1.3 shows a simplified general structure of the system.

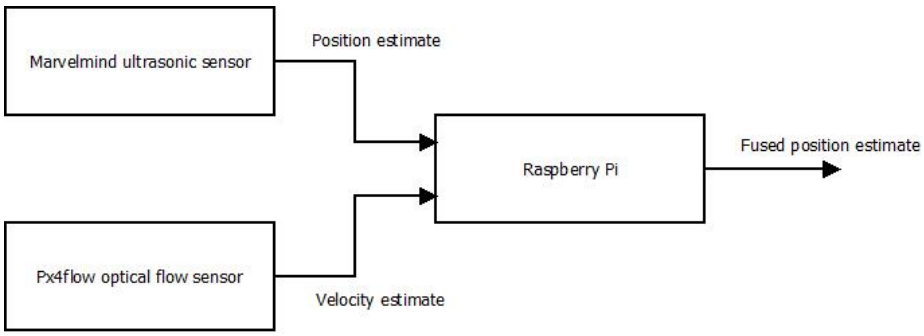


Figure 1.3: Simplified structure of the combined position estimate system

A comprehensive introduction to all the hardware used in the system will be presented in chapter 2, including the electronic characteristics, communication protocol, noise performance etc.

The algorithm used in the system will be illustrated in chapter 3. This chapter will introduce not only the Kalman filter itself, but also some preprocess algorithm used before feeding the data into the filter, i.e. the attitude compensation and the angular rate compensation for the optical flow sensor.

In chapter 4, the details on how to realize the algorithm on the Raspberry Pi are introduced. After that, the on-board test of all the algorithm will be included and discussed.

At last, a general conclusion and some discussion on the future works will be presented in chapter 5.

Chapter 2

Hardware

2.1 Overall Hardware set-up

As it is said in 1.4, the combined position estimate system employs two sensors: the Marvelmind, which is an ultrasonic sensor; and the Px4flow, which is an optical flow sensor. The Kalman filter is used to merge the data from two sensors together. The single board computer Raspberry Pi is in charge of performing all the algorithm. This system is under a redundant data fusion structure, which means we expect the two sensors to measure the position individually before we feed their data into the Raspberry Pi to perform the Kalman filter process. The Kalman filter is designed based on the system model, so it is essential to fully understand the mechanisms of the sensors. So in the chapter, we will focus on the hardware characteristic and working principle of the two sensors.

The Marvelmind ultrasonic sensor, unlike other type of ultrasonic sensors, is used to estimate a position coordinate, rather than just providing the distance from the sensor to an obstacle. A Marvelmind measurement system consists of three or four stationary beacons and one moving unit which is called hedgehog. The distance from the hedgehog to each of the stationary beacons will be measured, and the coordinate of the hedgehog will be estimated based on the distance information. More details on Marvelmind will be illustrated in section 2.2. Even though the working principle of the Marvelmind is complex, the usage of Marvelmind sensor is straight forward: it can directly provide a position estimation so we do not need to modify anything about the data from Marvelmind before we sent it into the filter.

The Px4flow sensor, on the other hand, provide a velocity measurement. In fact, the Px4flow measures the angular rate of a point in its point of view (POV), which we will discuss at length in section 2.3. A distance from the sensor to the ground is required to produce a velocity measurement. In this system, the distance information is also acquired from the Marvelmind. In order to get a position estimate from Px4flow, we need to integral its data before fusion its data

with Marvelmind. Further more, beside the integration, some other processions are also needed to apply to the data from Px4flow. Since the Px4flow measures the velocity, the direction of the velocity it is measuring is also required. After that, the angular rate of the sensor itself will influence its velocity output. As a result, the attitude and angular rate of the Px4flow sensor are needed in order to provide an accurate velocity estimate, which finally yields a position estimation. These processes are called the *pre-process* of the Px4flow and the algorithms of the pre-processes will be comprehensively discussed in 3. For now we just need to know that, to produce the position estimation from the Px4flow, an IMU is also needed. Fortunately there is an IMU integrated inside the Marvelmind which we can employ.

After all, figure 2.1 shows an overall structure of the system. Notice that in the figure, a box with **bold** text means it is a hardware while a box with *italics* text means it is an algorithm. The text with normal type above an arrow stands for a data. Some data have brackets following them, the contents in the brackets indicate the communication protocol they are using.

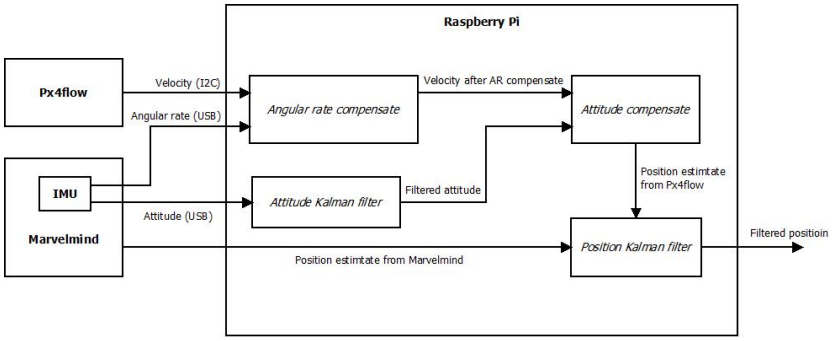


Figure 2.1: Overall structure of the combined position estimate system

2.2 Marvelmind

Marvelmind is an indoor navigation system. The high accuracy ($\pm 2cm$), low power consumption ($0.4W$) and compact size ($55 \times 55 \times 65mm$) make it suitable for all kinds of indoor applications. It employs the ultrasonic ($31kHz$) to perform the measurement and support varies communication protocols such as USB, SPI and UART. The location update rate is about $8Hz$, however this frequency may change depending on the operating environment. The measurement range is around $1000m^2$ and all the components in the Marvelmind system can be powered by a $5V$ power supply. The developer states that the Marvelmind sensor is an indoor GPS system since it can directly output fake GPS format data. So any system requires a GPS format information can integrate with the Marvelmind sensor without much configurations. Moreover, the Marvelmind also has an integrated IMU which can be used to perform attitude

estimations. In this section, the working principle and communication protocol, together with other technical details will be introduced in detail.

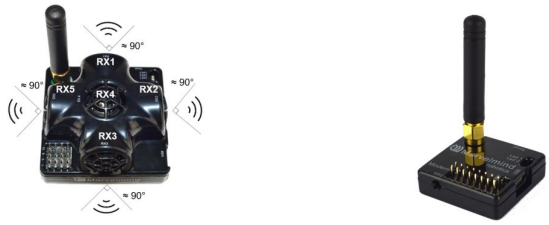
2.2.1 Working mechanism

A complete Marvelmind system consists three components: stationary beacons, hedgehog and a router. Their functions are listed below:

Beacon The beacons are mounted stationary on the walls or roofs. A navigation system requires three or four beacons to perform the position estimate. Beacons will send ultrasonic pulses to the hedgehog and record the time point of sending the pulse. The time point of sending the pulse will be transmitted to the router and compare with the time point the pulse received by the hedgehog. By this, the distance from each beacon to the hedgehog will be estimated. This method of distance measurement is called Time-Of-Flight (TOF) method. Figure 2.2 shows a picture of the beacon.

Hedgehog Technically speaking, the hedgehog and the beacon is identical on hardware. The only difference is their functions. A hedgehog is mounted on the target whose position is desired to be measured. Its duty is to receive the ultrasonic pulses from beacons and measure the time point when a pulse arrives. The time point of receiving the pulse will then be sent to the router and compare with the time point of sending of the same pulse. Since the hardware of a hedgehog and a beacon is the same, their functions can be changed on a PC through a set-up software called dashboard. This means any beacon can be configured into a hedgehog and vice versa.

Router The router plays a central controller role in the Marvelmind system. The time point measured by the beacons and the hedgehog will be sent to the router. The router then compares the interval between the time points and provides a distance estimate from the hedgehog to each of the beacons. Moreover, the router will also perform a calculation that produce a location of the hedgehog in a Cartesian coordinate. The origin point the of coordinate can be freely changed through the set-up software. Notice that although changing the configurations of the system requires connecting the router to a PC through a USB cable, the position estimate itself does not involving any help from the PC. The router will perform all the computation for the position estimate. Thus means one can power the router through an adapter or even a USB power bank and the Marvelmind system will perform its measurement without any problem. Figure 2.2 shows a picture of the router.



(a) Beacon/Hedgehog (b) Router

Figure 2.2: Components of the Marvelmind system

Figure 2.2 shows a image of the beacon/hedgehog and the router. It can be observed that there are five ultrasonic receiver(RX)/transmitter(TX)s on each beacon/hedgehog. Ideally all the RX/TX will be used to perform the measurement. However, if in practical one of the RX/TX is blocked by come component, e.g. the shell of the UAV, this specific RX/TX can be disabled in the set-up software. This will ensure the data from the blocked RX/TX will not influence the measurement.

To successful perform a measurement, a Marvelmind system should contains one router, one hedgehog and three or four beacons. A typical set up of the system will be shown in figure 2.3. The maximum distance between the beacons is $50m$ and it is recommended not to mount the beacons over than $30m$ with each other. Again, the PC is just used for configuration purpose, the measurement does not rely on the presence of a PC.

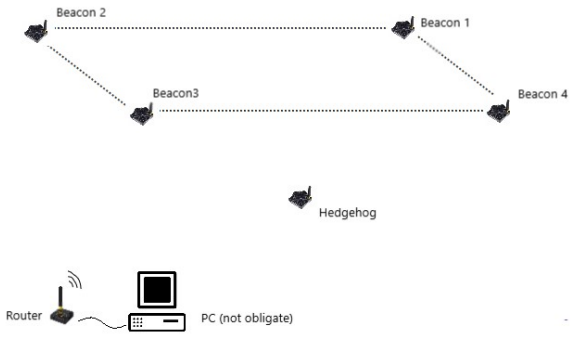


Figure 2.3: An example of a 4-beacon set up of Marvelmind system

After the router finishes the position estimate, the result will be transmitted to the hedgehog. Then the position can be acquired from the hedgehog via varies communication protocols. The details about the protocol will be demonstrated

in the appendix. It is worth to mention that the packet structure shows in the appendix is used in all the protocol supported by Marvelmind, i.e. UART, USB and SPI. However in this thesis, we employs the USB protocol only. The reason to choose USB is not only by it is more easy to use, but also by some hardware restrictions of the Raspberry Pi. We will continue the discussion on this in section 2.4.

2.2.2 Inertial measurement unit

As it is discussed in section 2.1, the estimation of Px4flow requires an attitude information to perform. Luckily there is an inertial measurement unit (IMU) embedded inside the beacon/hedgehog of the Marvelmind system so we do not need any additional device to measure the attitude.

IMU is widely used in guidance and navigation applications. Typically an IMU consists of an accelerometer and a gyroscope. Both the two sensors performs the measurement based on the inertial property itself, without any interaction with surrounding environment. That is where the term "inertial" comes from. When performing the attitude estimation, the accelerometer will in charge of measure the direction of earth gravity while the gyroscope keeps measuring the angular velocity on all the three axis. However the direction of gravity can only provide information on the pitch and roll axis. Figure 2.4 shows a orientation of the three rotation axis. From the figure we can find that no matter how the sensor rotates around the yaw direction, the partial of the gravity on the three axis will remain the same. This means only the gyroscope will be used to measure the rotation on the yaw axis, which introduce a drift error. To solve this, many modern IMU have a compass integrated inside. By the help from the compass, the rotation around the yaw axis can be estimated accurately.

For a better understanding of the structure for the IMU inside the Marvelmind beacon, figure 2.5 shows a brief construction.

Since the IMU is embedded inside the Marvelmind, it also follows the same communication protocol as the Marvelmind. The difference is the IMU using a different payload field from the Marvelmind. More details of the IMU communication protocol can be found in the appendix.

IMU calibration

By analysing the data from IMU, we find all the IMU sensors have sorts of bias outputs. This means the output is not centred at zeros. For example, if we put the Marvelmind at a fixed position without any movement, the output from gyroscope is not zero as we expect. So it is essential to calibrate the bias and cancel them out before any further processing. For the accelerometer and the compass, we can calibrate the bias by the following routine (for clear demonstration the figure 2.6 is used as a reference): For accelerometer:

1. Face the A side of the Marvelmind to the ground, record the data in X axis of accelerometer.

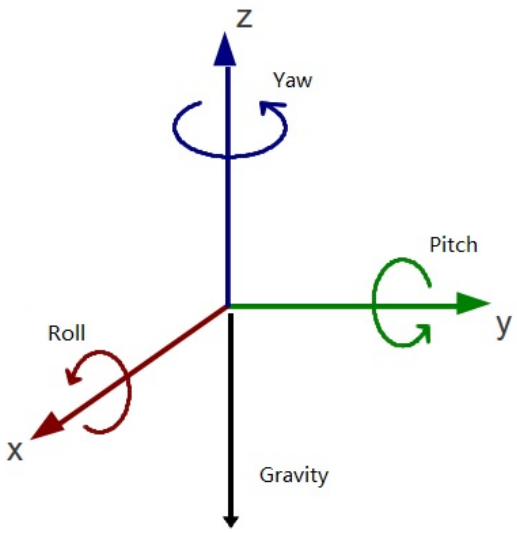


Figure 2.4: Gravity cannot help measuring the change in yaw axis

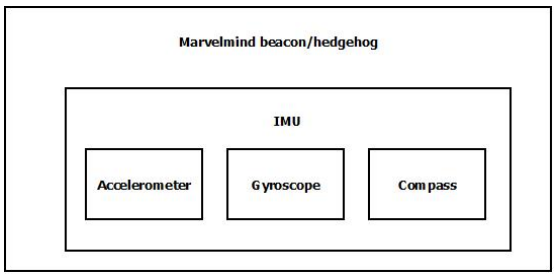


Figure 2.5: The structure of IMU inside Marvelmind beacon

2. Face the B side of the Marvelmind to the ground, record the data in X axis of accelerometer.

3. The mean of the two records is regarded as the bias in the X axis of the accelerometer.

4. Repeat the same routine on C, D sides and top, bottom sides to get the bias in the Y and Z axis.

For compass:

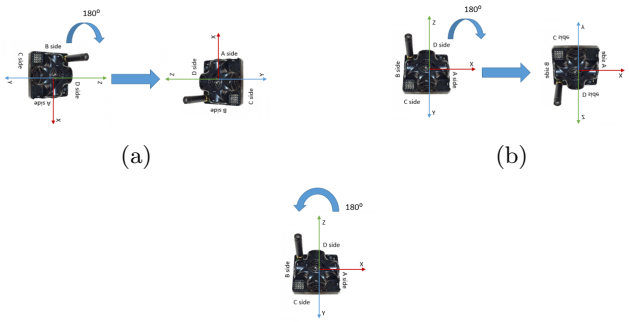
1. Put the Marvelmind at a fixed point, make sure there's no metal objects or high frequency electromagnetic field surrounded.

2. Record the data in X and Y axis of the compass.

3. Turn the Marvelmind 180° around Z axis, record the data in X and Y axis of the compass.

4. The mean of the two records is regarded as the bias in the X and Y axis of the compass.

5. The bias of the Z axis of the compass can be measured using the same method as the Z axis of the accelerometer.



(a) (b) (c)

Figure 2.6: Calibrate the bias of accelerometer and compass

The gyroscope, on the other hand, has a different bias each time the sensor starts working. So we cannot calibrate the gyroscope once and use the bias we measured in another time. The bias of the gyroscope should be measured every time the Marvelmind is powered up. Each time we startup the Marvelmind, the system should be kept stationary without any movement for 20 to 30 seconds and the output from the gyroscope is recorded. After that, the mean value during this period is regarded as the bias of the gyroscope.

2.3 Px4flow

Px4flow is an optical flow sensor which provides velocity estimation. The difference between Px4flow with other optical flow sensors is it has a much higher resolution image sensor. The MT9V034 CMOS image sensor used in Px4flow has a resolution of 752×480 pixels, which makes the Px4flow capable of working without high illumination environment. A 168 MHz Cortex M4F CPU is in charge of performing the optical flow computation on-board at a frequency of $400Hz$. A $16mm$ M12 lens is used which has a $21°$ field of view (FOV). The lens is also equipped with an IR filter to reduce the interference from infrared. The compact size ($45.5mm \times 35mm$) and low power consumption (about $575mW$) makes the Px4flow suitable for the indoor UAV applications. Figure 2.7 shows a picture of the Px4flow sensor. Notice the column object besides the camera is an ultrasonic sensor, it suppose to be used for providing the distance information to the Px4flow. However in the current firmware version of the Px4flow sensor, it has been disabled due to some technical issues. In this thesis, the distance information will be provided by the Marvelmind sensor instead. In this section, the Px4flow sensor will be introduced in detail.

Figure 2.7: An overview of the Px4flow sensor

2.3.1 Working principle

When measuring, Px4flow sensor keeps taking image by its CMOS image sensor at a frequency of $400Hz$. If the Px4flow moves its camera against some reference plane (such as tiles on the ground, or texture of the ceiling), then two images taken at the adjacent time points will be generally identical, with

a small amount of displacement. This slightly differences can be used to justify the velocity of the camera moving against the reference plane. An image comparison algorithm will comparing the difference between two images and estimate who much the displacement is. This is so called the "optical flow", which basically means how many pixels fly by during the time interval of the two images. Figure 2.8 shows a roughly image of this measuring mechanism. Notice that in real situation, the time interval between two images is very short ($2.5ms$) so the according displacement between them is typically small.

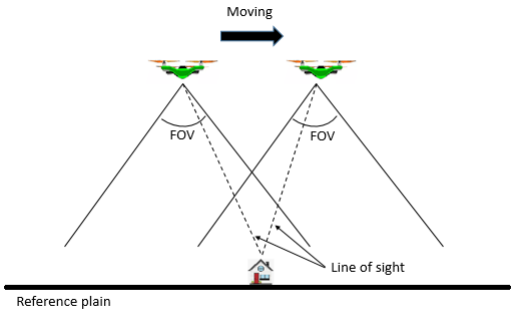


Figure 2.8: The working mechanism of Px4flow

After the optical flow is measured, an angular rate of the target moving inside the FOV of the camera can be computed. For a better demonstration, figure 2.9 shows a image of the light path between the lens and the CMOS image sensor. A reference target is expressed using a star and its image on the CMOS sensor is presented by the dashed star. Suppose the target moving from location A to location B as shown in the figure, its image will move from location A' to B' on the CMOS sensor. The displacement is the optical flow as we discussed above. Then divide the displacement with the focus length of the lens, we can get the angle θ. Then simply divide θ by the time interval between two images, the angular velocity of the target moving inside the FOV of the camera can be computed. Since the θ is typically small, we can employ the approximation as:

$$\theta \approx \tan\theta \approx \frac{D}{F} \tag{2.1}$$

where

- D is the displacement of the image, or equally says the "optical flow".
- F is the focus length of the lens.

Notice the size of one pixel of the camera in Px4flow is $6\mu m$ and the comparison algorithm using a four binning image process. This means one unit of optical flow corresponds to $6 \times 4\mu m$. The focus length of the lens is $12mm$.

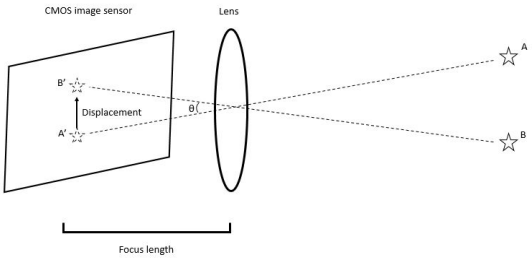


Figure 2.9: The light path inside the camera

As we discussed above, the Px4flow actually providing an estimation of the angular rate of the vehicle. Thus means to convert angular rate to velocity regards to the reference plane, it requires a distance information. In this thesis, this information is gathered from Marvelmind since the Marvelmind can estimate the target position, which also include the altitude information. In our design we will face the Px4flow sensor's camera to the ground, then the altitude, which is the distance between the UAV and the ground, yields the value we need to compute the velocity from the angular rate.

2.3.2 Attitude compensation

In subsection 2.3.1, we have seen that the Px4flow provides an estimate of the vehicle velocity. Then the position can be calculated by integral the velocity. However before merging the position estimated by Px4flow and by Marvelmind, we must put two position estimations in the same coordinate system. Thus, the attitude of the vehicle should be considered.

For a clear illustration, figure 2.11 shows a situation that two UAV has different attitude in a same reference coordinate system. Assuming two identical UAVs mounting the Px4flow sensor, and their position is also measured by the Marvelmind sensor in a Cartesian coordinate system. UAV 1 moves along the Y axis and the UAV 2 moves along the 45° direction between X and Y axis. Suppose they are moving with the same velocity, i.e. $v_1 = v_2$. If we directly integral the velocity measured by the Px4flow to estimate their position, then the two UAVs will have the same position offset. This kind of position estimate is obviously useless for us since we would like to know the position in a coordinate frame as the Marvelmind do.

Moreover, the attitude of the UAV also influence the distance from the camera of Px4flow to the reference plane. Since we are going to employs the altitude data from Marvelmind as the distance from Px4flow to the ground. Suppose the UAV flying with a non-horizontal attitude. Then the real distance to the reference is larger than the altitude of the UAV. We can employ figure 2.10 illustrate this mechanism more clearly. In figure 2.10a, A stands for the altitude of the UAV and R stands for the displacement of the vehicle. If we assume the θ

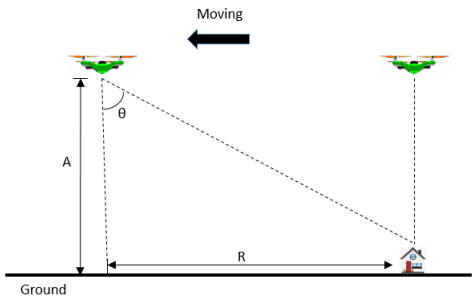
is small, the we can use the following approximate to estimate the displacement:

$$R = A \tan\theta \approx A \cdot \theta \tag{2.2}$$

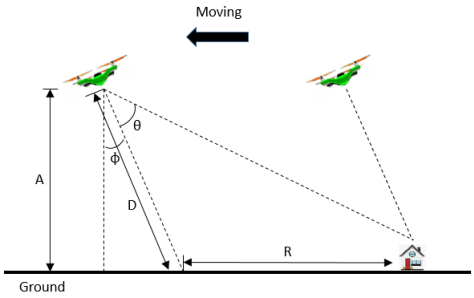
Now we consider the situation in figure 2.10b, obviously Eq. 2.2 is not applicable any more. The distance from the Px4flow to the reference D should be firstly computed by:

$$D = A \cos\phi \tag{2.3}$$

where ϕ is introduced by the attitude of the UAV.



(a)



(b)

Figure 2.10: Attitude influences the displacement estimation

As a result, an attitude information is required so we will know not only how fast the vehicle is moving, but also which direction it is moving towards. In this thesis, we will use the quaternion and direction cosine matrix (DCM) method to estimate the attitude information from the data of IMU. The detailed explanation of the algorithm will be discussed in chapter 3.

2.3.3 Angular rate compensation

We have already seen how the attitude of the UAV can influence the position estimation from Px4flow. In this subsection, we would like to discuss another

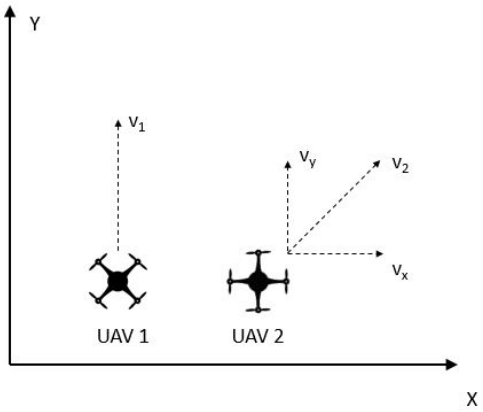


Figure 2.11: Attitude influences the position in the coordinate system

kind of interaction comes from the angular rate of the vehicle.

In subsection 2.3.1 we have seen how the Px4flow measuring the angle change of a target in figure 2.8. However this is the the perfect situation we can imagine. In practical, the sensor will encounter the rotations which will introduce an error to the result.

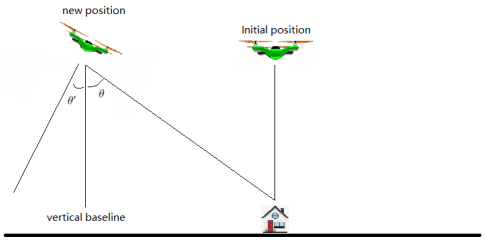


Figure 2.12: Measurement error caused by rotation

Figure 2.12 shows an example. In the figure, $\theta' + \theta$ is the angle that the sensor measures and the θ' part is caused by the rotation of the sensor which we want to cancel out. This rotation may occur in any body axis so the angular rate around each body axis of the UAV (can be measured by the gyroscope) is coupled together. This means, for example, the angular rate around x axis ω_x will not only influence the output of the sensor in y axis, but also affect the output in x axis. To compensate the extra angle introduced by the rotation, the angular rate must be measured. And the quaternion and direction cosine matrix method will be used. Again, the detailed explanation of the algorithm will be discussed in chapter 3.

2.3.4 Illumination effect

The Px4flow employs a 752×480 CMOS image sensor to take images. This image sensor is much more sensitive comparing to other optical sensors, e.g. the sensors in an optical mouse, so it does not heavily rely on illumination sources. However the lighting situation does affect the noise characteristic of the output from px4flow. So it is recommended to test the noise variance in the specific illumination environment and adjust the parameters in Kalman filter according to the test.

One situation that need to pay enough attention is, when using Fluorescent lamp as the illumination source, the output from Px4flow could be disastrous. This is due to the fact that the Fluorescent lamp keeps blinking at a specific frequency. The traditional type of Fluorescent tube blinks at the same frequency as the grid power, which is $50Hz$ or $60Hz$ in most of the countries. Some modern technique Fluorescent tubes work in a higher frequency such as 500 - 2000 Hz. The Px4flow processes the images at a frequency of up to 400 Hz. Because we cannot guarantee the Px4flow and the Fluorescent lamp are synchronize, so even we assume the Fluorescent lamp we are using as lighting source is working at 2000 Hz, it still has the possibility that the sampled images in Px4flow have different brightness. Figure 2.13 shows a test result of Px4flow under different illumination situations. The test is done by locating the Px4flow at a fixed position and facing the lens to the ground. The blue dots indicate the data collected under the natural sunlight while the red ones are under the Fluorescent lamp. The enormous difference can be observed easily and table 2.1 gives some numerical results. Notice that the Fluorescent lighting not only increases the standard variance by over 10 times as the sunlight, but also introduces a bias of about $0.0134rad$. As a comparison the output under sunlight has a mean value of $1.4875E-04rad$, which can be approximately regarded as a unbiased output, while the output under Fluorescent is obviously biased. Another thing for the test is, the test of the output under Fluorescent is done in the daytime. This means it still has the sunlight at the background. If it is in an environment without any natural light, the result could be even worse.

	Standard variance (Unit:rad)	Mean (Unit:rad)
Under sunlight	0.0022	1.4875E-04
Under Fluorescent	0.0308	0.0134

Table 2.1: Noise characteristic of outputs from Px4flow

So as a conclusion in this subsection, it is suggested that avoiding Fluorescent lamps as the lighting source. If this cannot avoid, then the noise characteristic must be measured carefully and the bias of the output should be also considered.

Before we leaving this section, one thing needs to note is the Px4flow sensor

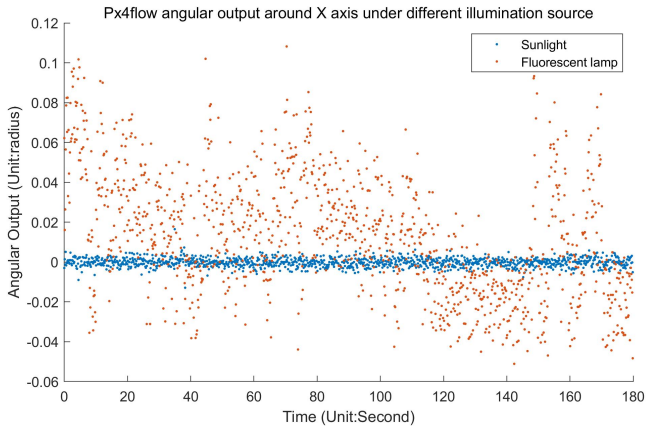


Figure 2.13: Output from Px4flow in different illumination situations

supports three types of communication protocol: UART, USB and I2C. However the USB protocol in Px4flow is mainly used for configuration purpose. Besides the optical flow data, the image taken by the camera will also be transmitted in USB protocol. The image can be used to adjust the focus of the lens. However it will also reduce the update rate of the optical flow data since it has to leave sufficient space for the image transmission. In the thesis, I2C protocol is used to connect the Raspberry Pi with the Px4flow. Choosing I2C over UART is due to some special restrictions of the Raspberry Pi, which will be discussed in section 2.4. More details of the communication protocol of the Px4flow will be introduced in the appendix.

2.4 Raspberry Pi

Raspberry Pi is a low cost single board computer. It is designed to be a cheap, compact PC/develop board, but still rather powerful comparing to its size. Its capability and compact size (85.60mm $\times$ 53.98mm $\times$ 17mm) makes it very popular since its born. The first generation of Raspberry Pi is released in 2012 and sold out nearly 15 million in 2017. It directly runs Linux as its operating system, which makes it easily to expand with customized programs. Figure 2.14 shows a picture of a Raspberry Pi 3 B+ model.

Raspberry Pi is widely used as multiple roles in many applications, such as Internet of Things (IoT), smart-home, robotics, etc. Armed with a monitor, a keyboard and a mouse, the Raspberry Pi can be used as a normal PC which even capable of running some 3D video games. In the mean while, the 40-pin GPIO header makes it ideally for embedded developing purpose. In this thesis, the Raspberry Pi 3 B+ is employed as the main controller, which will be briefly introduced in this chapter.

Figure 2.14: A Raspberry Pi 3 B+

2.4.1 Brief introduction

The hardware specifications of Raspberry Pi 3 B+ are listed below:

- main processor: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SOC, 1.4GHz
- 1GB LPDDR2 SDRAM
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
- Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)
- Extended 40-pin GPIO header
- Full-size HDMI
- 4 USB 2.0 ports
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- 4-pole stereo output and composite video port
- Micro SD port for loading your operating system and storing data
- 5V/2.5A DC power input
- Power-over-Ethernet (PoE) support (requires separate PoE HAT)

The power consumption of Raspberry Pi will be vary depending on what kind of tasks its running with. However a typical power usage when it is booted up without any further operation is about 2 watt.

Since the Raspberry Pi uses the Linux as its operating system, developers can use almost any program language to expand its function such as C/C++, Java, Python, etc. This is very convenient comparing to other embedded processors, which only support assembly language or C/C++ to program. However this convenience also comes with a drawback. The utilization of Linux means the Raspberry Pi is not a real-time processor. The kernel of standard Linux will judge which task will be processed first, not the developer. Although some Linux patch (such as Linux RT) makes the system running in a "quasi-realtime" way, it is still hardly to perform some specific tasks with high realtime processing requirement. For example, even though there are 40 GPIO pins in a Raspberry Pi, developers can not use any pin they want as a standard UART serial port. In contrast, a developer using the Arduino Uno can declare any GPIO pins for serial transmissions.

As it is said above, the Raspberry Pi support a wide range of programming language. In this thesis we are using the Python to build the "logic parts" of our algorithm. Logic parts here means all the tasks excluding the filter algorithms, such as opening a serial/I2C port, changing the unit of the data, reading/saving results to a file, etc. Python is an interpreted high-level programming language. It is object-oriented and easy to read. Using Python at the logic parts will save lots of time dealing with miscellaneous treatment (the data type, function pointer, ...).

On the other hand, the C language will be used to process the actual algorithm in this thesis, including the attitude estimation, angular rate compensation, Kalman filter. Since these algorithm involves many mathematical computation, especially some matrix calculations and float computation, using the C language will provide a faster computation on such tasks.

2.4.2 Communication restricts of Raspberry Pi

The Raspberry Pi support varies communication protocols to transmit data with peripheral devices, such as I2C, SPI, UART, USB. However there are some restrictions in the UART port of Raspberry Pi. Table 2.2 shows an overall situation about the available communication protocols in a Raspberry Pi. Notice that the SPI port in Raspberry Pi has two chip select pins, which makes it able to connect two devices.

Protocol	Available ports
I2C	2
SPI	1
UART	1 standard + 1 mini
USB	4

Table 2.2: Supported communication protocol in Raspberry Pi

As we have discussed in subsection 2.4.1, not all the GPIO pins can be used as the serial port since the Linux is not a realtime system. The Raspberry Pi uses the serial port integrated inside the BCM2837B0 processor to perform UART communication tasks. As a result, there is only one standard UART port inside the Raspberry Pi and it is connected to the bluetooth module by default. This means we have to disable the bluetooth module if we want to use the standard UART port. However the Raspberry Pi do have another UART port which is called a mini-UART, and is used for console output by default. But the mini-UART lacks the functions such as parity check, making the use of mini-UART highly restricted. What make things even worse is the standard UART and the mini-UART share the same GPIO pin through a multiplexer, this means we can only reach to one of them at the same time. As a result, we are trying to avoid using a UART protocol in this thesis. The USB protocol is used to connect with the Marvelmind and the I2C protocol is used to connect with the Px4flow.

Chapter 3

Data fusion algorithm

3.1 Introduction

As we have already introduced, the combined position estimate system in this thesis employs the Kalman filter as the main data fusion method. The essential factor to build a proper Kalman filter is the system modelling. Our system consists of three main part: the attitude estimation for the UAV, the angular rate compensation for the Px4flow, and finally the position estimate combining the data from Px4flow and Marvelmind. Mathematical models for these three parts should be carefully built before the Kalman filter processing.

So in this chapter, we will first introduce two important tools which is used in the attitude estimate and the angular rate algorithm. They are the concepts of quaternion and direction cosine matrix (DCM). Then we should focus on the mathematical modelling for our system, including the attitude estimate, angular rate compensate and the position estimate. At last, the details of the Kalman filter will be discussed.

3.2 System modelling

There are three main algorithms for the combined position estimate system: attitude estimation, angular rate compensation and the position estimation. In this section, their mathematical models will be introduced. However, before that, two mathematical tools which is used to present an attitude of the object will be introduced.

3.2.1 Attitude mechanism

There are many ways to express a rotation of an object. Three majority methods used in the UAV field is: Euler angle, direction cosine matrix and quaternion. The Euler angle express an object attitude using the angle between the body axis and the fixed reference axis, which is typically denoted by φ, θ,

ψ. However, there is not a uniform definition on which denotion corresponding to which axis. Further more, different sequences of the rotation also produce different attitude, so it is essential to determine a specific rotation order before using the Euler angle to express a rotation. Another problem of the Euler angle is the gimbal lock, that is, in some specific situation, the expression of Euler angle will lost one degree of freedom, so different attitudes may refer to the same Euler angle sets. A detail discussion of gimbal lock can be found in [8].

The other two method, direction cosine matrix and quaternion avoid the problem of gimbal lock, and have a well defined orientation, so people do not need to clarify their coordinate set-up before employs this two methods. Moreover, since the two methods are mathematically equivalent, so the expression in one of the methods can be easily converted to another. In this section, we will first introduce the direction cosine matrix method, then follows the quaternion theory. At last, the relationship between them will also be introduced.

Direction cosine matrix

In this section, we will use $\hat{x_1}$,$\hat{x_2}$,$\hat{x_3}$ indicate the three unit vectors along the axis of the reference frame and $\hat{e_1}$, $\hat{e_2}$, $\hat{e_3}$ for unit vectors along the axis of the body frame. The body frame will rotate together with the object while the reference frame is a fixed frame. We will also use θ_{ij} represents the angle between the i-th axis in the body frame with the j-th axis in the reference frame. Notice here θ_{ij} and θ_{ji} are two different angles, for example θ_{12} means the angle between $\hat{e_1}$ and $\hat{x_2}$ while θ_{21} means the angle between $\hat{e_2}$ and $\hat{x_1}$. We also suppose the origin point of the two frame is same. Figure 3.1 shows a general idea of this set up.

Suppose at beginning, the reference frame and the body frame coincide together. Then the body frame rotates to a new position which results the situation in figure 3.1. Following the above definitions, we can write out the relationship between the unit vectors in the two frames.

$$\hat{e_1} = cos(\theta_{11})\hat{x_1} + cos(\theta_{12})\hat{x_2} + cos(\theta_{13})\hat{x_3} = a_{11}\hat{x_1} + a_{12}\hat{x_2} + a_{13}\hat{x_3} \tag{3.1}$$

$$\hat{e_2} = cos(\theta_{21})\hat{x_1} + cos(\theta_{22})\hat{x_2} + cos(\theta_{23})\hat{x_3} = a_{21}\hat{x_1} + a_{22}\hat{x_2} + a_{23}\hat{x_3} \tag{3.2}$$

$$\hat{e_3} = cos(\theta_{31})\hat{x_1} + cos(\theta_{32})\hat{x_2} + cos(\theta_{33})\hat{x_3} = a_{31}\hat{x_1} + a_{32}\hat{x_2} + a_{33}\hat{x_3} \tag{3.3}$$

Since the cosine function is used to express the direction of the vectors, this relationship is called directional cosine. We can also write Eq. 3.1, Eq. 3.2, Eq. 3.3 into a compact way as:

$$C = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{3.4}$$

Eq. 3.4 is called the direction cosine matrix DCM with regard to the object attitude in figure 3.1.

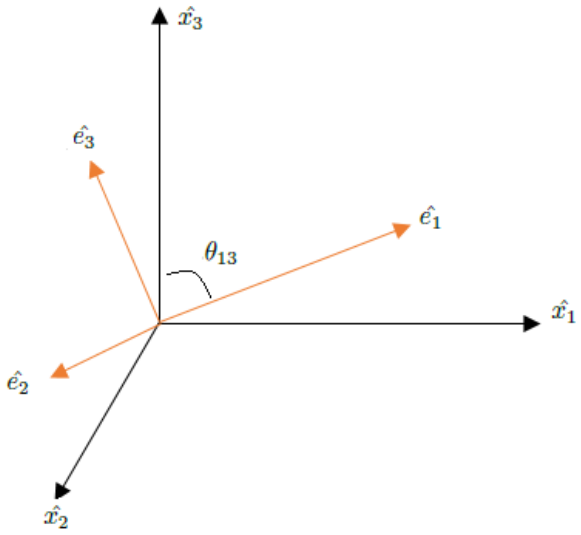


Figure 3.1: Reference frame and body frame

Now we assume a vector presented in the body frame as

$$\vec{r} = e_1\hat{e_1} + e_2\hat{e_2} + e_3\hat{e_3} \tag{3.5}$$

we can calculate it's corresponding presentation $\vec{x}$ in the reference frame by substituting $\hat{e_i}$ using equation Eq. 3.1, Eq. 3.2 and Eq. 3.3 which leads to the following results:

$$x_1 = a_{11}e_1 + a_{21}e_2 + a_{31}e_3 \tag{3.6}$$

$$x_2 = a_{12}e_1 + a_{22}e_2 + a_{32}e_3 \tag{3.7}$$

$$x_3 = a_{13}e_1 + a_{23}e_2 + a_{33}e_3 \tag{3.8}$$

or in a more compact way:

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix} C \tag{3.9}$$

It can be proved that C is a rotation matrix[1], this means its transpose is its inverse: $CC^T = CC^{-1} = I$. So we can also write equation Eq. 3.9 in another form:

$$\vec{r} = C\vec{x} \tag{3.10}$$

Here in Eq. 3.10, $\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$ and $\vec{r} = \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix}^T$ which follows a more common expression for the vector.

By now we have a convenient method to represent a rotation: each rotation of the body frame has a corresponding direction cosine matrix (DCM), and we can use the DCM to transform any arbitrary vector presented by the coordinate

in one frame to the other. So our problem now is: given the angular rate on all three body frame axis (which can be measured by a 3-axis gyroscope), how to decide the corresponding DCM? A differential equation can be employed to answer this[1]:

$$\dot{C} = C[\omega\times] \tag{3.11}$$

where $[\omega\times]$ is a skew symmetric matrix as this:

$$[\omega\times] = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{3.12}$$

the ω in Eq. 3.12 indicates angular velocity of a right hand rotation around the axis presented by the subscript. Eq. 3.12 is actually a set of 9 differential equations and by solving it, we can get the DCM which relates to the rotation caused by the angular rates in Eq. 3.11.

Now we have a fully defined method DCM to represent a rotation, together with the kinetic differential function to calculate it. In subsection 3.2.1 we will introduce another method called quaternion, and discuss the relationship between quaternion and DCM.

Quaternion

Basic definitions of quaternion

Quaternion is a $\mathbb{R}^4$ vector in the form:

$$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix} \tag{3.13}$$

or in an extended complex number form as:

$$\mathbf{q} = q_1 + q_2 i + q_3 j + q_4 k \tag{3.14}$$

The definition of i, j, k follows the normal complex number as:

$$i \times i = j \times j = k \times k = -1 \tag{3.15}$$

and the relation ship between the imagine vectors is:

$$i \times j = k;\ j \times k = i;\ k \times i = j; \tag{3.16}$$

However the quaternion has some special properties differ from a normal vector or complex number. The most important one is the quaternion is not commutative. So the inverted relationship of Eq. 3.16 is:

$$j \times i = -k;\ k \times j = -i;\ i \times k = -j; \tag{3.17}$$

The sum and product of the quaternions is similar as two vectors, suppose we have $\mathbf{q} = (q_1, q_2, q_3, q_4)^T$ and $\mathbf{p} = (p_1, p_2, p_3, p_4)^T$, then:

$$\mathbf{q} + \mathbf{p} = \begin{pmatrix} q_1 + p_1 \\ q_2 + p_2 \\ q_3 + p_3 \\ q_4 + p_4 \end{pmatrix} \tag{3.18}$$

and

$$\mathbf{q} \times \mathbf{p} = \begin{pmatrix} q_1p_1 - q_2p_2 - q_3p_3 - q_4p_4 \\ q_2p_1 + q_1p_2 - q_4p_3 + q_3p_4 \\ q_3p_1 + q_4p_2 + q_1p_3 - q_2p_4 \\ q_4p_1 - q_3p_2 + q_2p_3 + q_1p_4 \end{pmatrix} \tag{3.19}$$

The conjugate $\mathbf{q}^*$ of $\mathbf{q}$ is:

$$\mathbf{q}^* = \begin{pmatrix} q_1 \\ -q_2 \\ -q_3 \\ -q_4 \end{pmatrix} \tag{3.20}$$

We should also define the length of a quaternion as:

$$|\mathbf{q}| = \sqrt{\mathbf{q} \times \mathbf{q}^*} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} \tag{3.21}$$

and the inverse:

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{|\mathbf{q}|^2} \tag{3.22}$$

The unit quaternion means the length of the quaternion is unity, which has the form:

$$\mathbf{q} = \begin{pmatrix} \cos\theta/2 \\ \mathbf{u}\ \sin(\theta/2) \end{pmatrix} \tag{3.23}$$

where

- θ is a rotation angle.
- $\mathbf{u}$ is a 3-dimensional unit vector stands for the image part of a quaternion.

Expression rotation using quaternion
In order to express a rotation in quaternion, the first thing is to convert a regular vector into a "pure" quaternion. Suppose we are going to express a rotation of a vector $\vec{n} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$, the corresponding pure quaternion with regards to the $\vec{n}$ is:

$$\vec{n}_q = \begin{bmatrix} 0 & x_1 i & x_2 j & x_3 k \end{bmatrix}^T \tag{3.24}$$

Note the $\vec{n}_q$ indicates its the quaternion version expression of the original vector $\vec{n}$. Basically to expression a vector in a quaternion way, simply writes the coordinates of the vector in the imaginary part of the quaternion, and the real part of this quaternion is zero.

Now assuming a rotation applies to $\vec{n}_q$, which can be expressed by a unit quaternion $\mathbf{q}$:

$$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix} \tag{3.25}$$

Suppose we rotate the vector $\vec{n}_q$ in a fixed reference coordinate, which means the vector is rotating and the coordinate remains stationary. The vector after the rotation can then be represented as:

$$\vec{n}'_q = q\vec{n}_q q^* \tag{3.26}$$

It can be proved that $\vec{n}'_q$ is also a pure quaternion as the $\vec{n}_q$[18]. Then the imaginary part in $\vec{n}'_q$ is the coordinates we want for the rotated vector.

A convenient feature for the quaternion is, if we know a quaternion presenting a rotation, then it is very straight forward to find out the rotating axis and angle for this rotation. Remember the quaternion which express a rotation is a unit quaternion, back to the definition of the unit quaternion Eq. 3.23, the rotation axis is simply presented by the vector $\mathbf{u}$ in the imaginary. Meanwhile, the θ stands for how much angle the object rotates.

By now we have know how to express a rotation in a quaternion way. However we also need to know the kinetic differential equation which link the quaternion with the angular rate on all three body frame axis, which is:

$$\dot{q} = F_q(\boldsymbol{\omega})q \tag{3.27}$$

where

$$F_q(\boldsymbol{\omega}) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix} \tag{3.28}$$

Eq. 3.27 is actually a four differential equations set.

Relationship between DCM and quaternion

Mathematically speaking, the DCM and quaternion expression for a rotation is equivalent. So if we know a rotation expressed in quaternion $\mathbf{q}$, we can directly write out the corresponding DCM. Here we should consider two situations of rotation. First one is, we have a fixed reference coordinate, and there is a vector $\vec{n}$ inside it. By applying a rotation $\mathbf{q} = \begin{pmatrix} q_1 & q_2 & q_3 & q_4 \end{pmatrix}^T$, the vector $\vec{n}'_q$ after the rotation, which is expressed in the reference coordinate can be represented as:

$$\vec{n}' = C\vec{n} \tag{3.29}$$

where

$$C = \begin{bmatrix} (q_1^2 + q_2^2 - q_3^2 - q_4^2) & 2(q_2q_3 - q_1q_4) & 2(q_2q_4 + q_1q_3) \\ 2(q_2q_3 + q_1q_4) & (q_1^2 - q_2^2 + q_3^2 - q_4^2) & 2(q_3q_4 - q_1q_2) \\ 2(q_2q_4 - q_1q_3) & 2(q_3q_4 + q_1q_2) & (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix} \tag{3.30}$$

This rotation can be expressed in figure 3.2a.

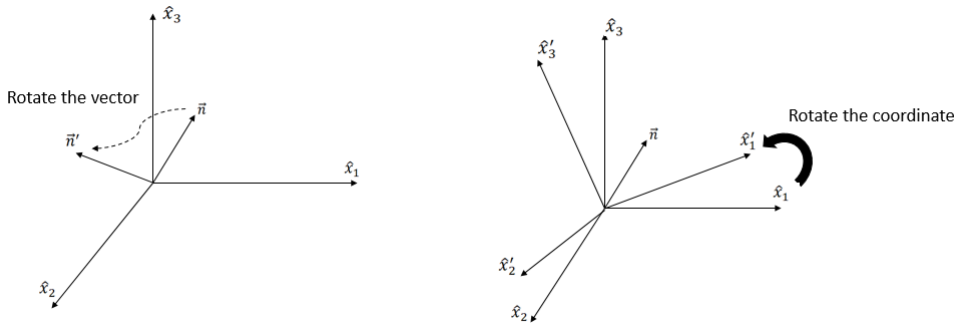
There is another kind of rotation, that is, suppose we have a fixed vector $\vec{n}$ which is expressed in a rotation body coordinate. At the beginning, the rotation body coordinate is coincide with the fixed reference coordinate. Now a rotation $\mathbf{q}$ applied to the body coordinate but vector $\vec{n}$ and the reference coordinate remain stationary. We want to know the expression of $\vec{n}$ in the rotated body coordinate. This can be done by:

$$\vec{n}' = C'\vec{n} \tag{3.31}$$

where

$$C' = \begin{bmatrix} (q_1^2 + q_2^2 - q_3^2 - q_4^2) & 2(q_2q_3 + q_1q_4) & 2(q_2q_4 - q_1q_3) \\ 2(q_2q_3 - q_1q_4) & (q_1^2 - q_2^2 + q_3^2 - q_4^2) & 2(q_3q_4 + q_1q_2) \\ 2(q_2q_4 + q_1q_3) & 2(q_3q_4 - q_1q_2) & (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix} \tag{3.32}$$

This rotation can be expressed in figure 3.2b



(a) Rotate a vector in a fixed coordinate

(b) Rotate the coordinate when the vector remains fixed

Figure 3.2: Comparison between two rotations

As a conclusion, we have discussed two method for representing the rotations, they are DCM and quaternion. The relationship between them are also discussed. Notice that there are two types of rotation introduced in this section, the rotation of a vector in a fixed coordinate and the rotation of the coordinate when the vector remains the same.

Comparing the DCM and quaternion, we found that, DCM is more suitable to interact with a vector, since to rotate a vector, we simply multiply the DCM with it. On the other hand, quaternion uses less parameters to express a rotation (four parameters comparing to nine in DCM). Moreover, given a quaternion, it is intuitive to point out the rotation angle and rotation axis. When talking about the kinetics for the two methods, to get a DCM we need to solve a nine differential equations set, while to get a quaternion we only need to solve a

four equations set. So generally speaking, quaternion is more easy to compute and DCM is more suitable to use with a vector. In this thesis, we employs the quaternion method to estimate the attitude of the UAV, then convert the quaternion into DCM, and use the DCM method to apply the attitude to the data from Px4flow.

3.2.2 Attitude estimate model

In this thesis, we are going to estimate the attitude using the quaternion method mentioned in subsection 3.2.1. The input of the algorithm is the data from IMU sensors, e.g. the data from gyroscope, accelerometer and compass. The output will be the quaternion which indicates the rotation. Notice here the attitude output is not an "absolute" value to indicate the attitude, such as a specific set of Euler angle. Choosing the quaternion as the attitude output makes it very adaptable when combine the estimated attitude information with other data. We will shortly see this later in this subsection.

The gyroscope will be used as the primary estimator and the accelerometer and the compass will be used as two observers. Here for convenience, we write again the kinetic differential equation for the quaternion as:

$$\dot{\mathbf{q}} = F_q(\boldsymbol{\omega})\mathbf{q} \tag{3.33}$$

where

$$F_q(\boldsymbol{\omega}) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix} \tag{3.34}$$

So the data from gyroscope will be put into Eq. 3.34. Then Eq. 3.33 is used to estimate the quaternion with regards to the attitude.

On the other hand, for the observers, at the very beginning time t_0, we assume the data output from accelerometer and the compass are $\vec{n}_a = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}^T$ and $\vec{n}_c = \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}^T$. Suppose the system rotates to a new attitude at time t_1, and we calculate the quaternion $q_g = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^T$ using the data from gyroscope. Using the relationship between the quaternion and DCM from subsection 3.2.1, the according DCM can be expressed as:

$$C = \begin{bmatrix} (q_1^2 + q_2^2 - q_3^2 - q_4^2) & 2(q_2q_3 + q_1q_4) & 2(q_2q_4 - q_1q_3) \\ 2(q_2q_3 - q_1q_4) & (q_1^2 - q_2^2 + q_3^2 - q_4^2) & 2(q_3q_4 + q_1q_2) \\ 2(q_2q_4 + q_1q_3) & 2(q_3q_4 - q_1q_2) & (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix} \tag{3.35}$$

Notice here we are using the DCM which stands for the coordinate rotation.

Then the initial output from accelerometer $\vec{n}_a$ and the compass $\vec{n}_c$ should be rotated into the following vectors:

$$\begin{aligned} \vec{n}'_A &= C\vec{n}_A \\ \vec{n}'_C &= C\vec{n}_C \end{aligned} \tag{3.36}$$

We can also write this in a more compact way:

$$V' = \begin{bmatrix} a'_1 & a'_2 & a'_3 & c'_1 & c'_2 & c'_3 \end{bmatrix}^T = [C\ C]V \tag{3.37}$$

where

$$V = \begin{bmatrix} a_1 & a_2 & a_3 & c_1 & c_2 & c_3 \end{bmatrix}^T \tag{3.38}$$

Compare the calculated vector V' to the output from the accelerometer and the compass at time t_1, we will get a difference between the estimator (gyroscope) and the observers (accelerometer and compass). This difference can be processed by a Kalman filter, which will produce a overall better estimation for the attitude. The according Kalman filter process will be illustrated in detail in section 3.3.

Suppose we have an estimation for the attitude, we need to combine this attitude information with the output from Px4flow. Now we use $V_p = \begin{pmatrix} V_{px} & V_{py} & V_{pz} \end{pmatrix}$ stands for the output velocity data from Px4flow, $V_{true} = \begin{pmatrix} V_{truex} & V_{truey} & V_{truez} \end{pmatrix}$ stands for the "true" velocity after combining the attitude information, and using figure 3.3 as a reference. The combination can be done by the following routine.

Step 1 At the very beginning when the whole system is booting up. Set a "pointer" vector $P = \begin{pmatrix} P_x & P_y & P_z \end{pmatrix}$, where $P_x = 1$, $P_y = 0$, $P_z = 0$. This pointer vector P is used as a reference vector to indicate the initial attitude of the object. Since a quaternion stands for a rotation with regards to the initial attitude, we can rotate the pointer vector P using the quaternion so the current attitude can be expressed.

Step 2 Each time there is an available attitude data, rotates the pointer vector P by the according DCM C as $P' = \begin{pmatrix} P'_x & P'_y & P'_z \end{pmatrix} = CP$. P' stands for the current attitude of the system. Notice that the original P should maintain unchanged for further usage.

Step 3 V_{true} can be computed as:

$$\begin{aligned} V_{truex} &= V_{px} \times P'_x + V_{py} \times P'_y \\ V_{truey} &= -V_{px} \times P'_y + V_{py} \times P'_x \end{aligned} \tag{3.39}$$

3.2.3 Angular rate compensate model

In this subsection, we will use $\hat{x}$, $\hat{y}$, $\hat{z}$ represent the axis for reference frame and use $\hat{x_B}$, $\hat{y_B}$, $\hat{z_B}$ for the body frame. The $\hat{z_B}$ is set as the direction that the camera of Px4flow facing to (which means the camera "looks" toward the $\hat{z_B}$ direction). We also put a test vector $\vec{r}$ locates in reference frame along the $\hat{z}$ axis, and another test vector $\vec{r_B}$ in the body frame along the $\hat{z_B}$ axis. So the $\vec{r}_B$ will rotate together with the sensor while the$\vec{r}$ remains fixed in the reference frame.

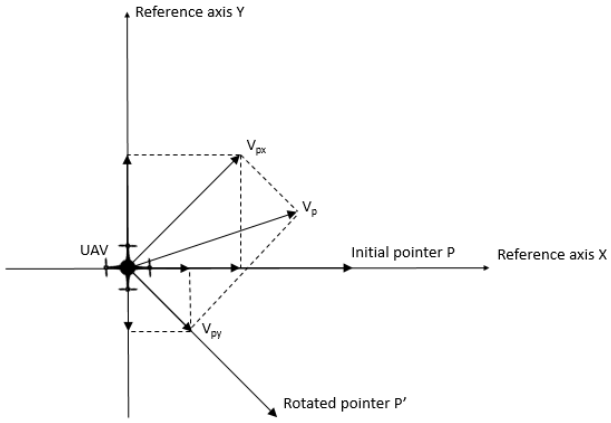


Figure 3.3: Combining output from Px4flow with attitude

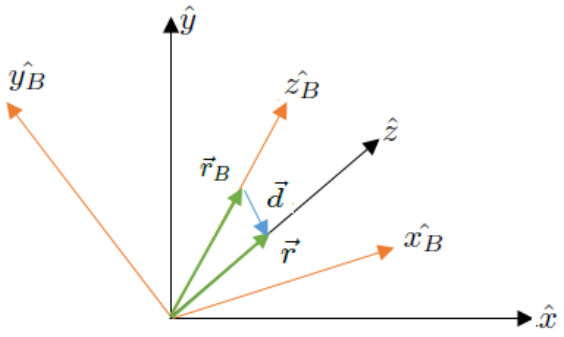


Figure 3.4: Body frame of the sensor rotates to another attitude at t_1

At the initial state t_0, we regard that the body frame and the reference frame is coincide. Now suppose the sensor rotates to a new state at time t_1. Figure 3.4 shows this set up.

Here we want to know the relationship between these two frames. Following the method introduce in subsection 3.2.2, we collect the gyro data from the IMU. Here we make the assumption that the angular rate of the sensor is constant in this period, so we get the information about the angular rate around the body axis: ω_{xB}, ω_{yB}, ω_{zB}. After this, recall the kinetic differential equation of the quaternion:

$$\dot{q} = F_q(\boldsymbol{\omega})q \tag{3.40}$$

the attitude of the sensor can be computed. Then convert the quaternion into the corresponding DCM as:

$$C = \begin{bmatrix} (q_1^2 + q_2^2 - q_3^2 - q_4^2) & 2(q_2q_3 + q_1q_4) & 2(q_2q_4 - q_1q_3) \\ 2(q_2q_3 - q_1q_4) & (q_1^2 - q_2^2 + q_3^2 - q_4^2) & 2(q_3q_4 + q_1q_2) \\ 2(q_2q_4 + q_1q_3) & 2(q_3q_4 - q_1q_2) & (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix} \tag{3.41}$$

Notice the DCM here presents the rotation of the coordinate for a fixed vector, as discussed in subsection 3.2.1. Now We can represent the test vector $\vec{r}_B$ from body frame into the reference frame:

$$\vec{r}_R = C\vec{r}_B \tag{3.42}$$

Here $\vec{r}_R$ stands for the same vector as $\vec{r}_B$ but presented in the reference frame. Now by subtract $\vec{r}$ with $\vec{r}_R$, we get the difference vector $\vec{d}$.

$$\vec{d} = \vec{r} - \vec{r}_R \tag{3.43}$$

The difference vector $\vec{d}$ is the extra flow caused by the rotation of the sensor, which is desired to remove from the sensor outputs. The partial component of the $\vec{d}$ in $\hat{x}$ axis ($\vec{d_x}$) introduces an error into the optical sensor in $\hat{x}$ direction, and similar for $\vec{d_y}$. By dividing the $\vec{d_x}$ with the magnitude of the test vector $|\vec{r}|$, the according extra angle caused by the rotation of the sensor is obtained. So the actual length of the test vector $\vec{r}$ is not essential to this algorithm. Notice the $\vec{d_x}$ introduce the angle error around the $\hat{y}$ axis so it needs to subtract to the ω_y, vice versa for $\vec{d_y}$.

As a conclusion, we summarize the compensation procedure in three steps:

Step 1 Collect angular rate data from the sensor, calculate the direction cosine matrix C using Eq. 3.40 and Eq. 3.41.

Step 2 Compute the difference vector $\vec{d}$ by Eq. 3.42 and Eq. 3.43.

Step 3 Subtract $\vec{d_x}$ and $\vec{d_y}$ part of the $\vec{d}$ to the output of the optical sensor.

3.2.4 Position estimate model

The position estimate model can be divided into two parts: the estimator model for the Px4flow, and the observer model for the Marvelmind. We will discuss them sequentially.

Px4flow estimator model

To model this Px4flow estimator, we use x and y indicate the position of the UAV in X and Y axis. Then we have:

$$\begin{aligned} \dot{x} &= (\omega_{fy} - \omega_{compensate\_y}) \cdot z + u_{fy} \\ \dot{y} &= (\omega_{fx} - \omega_{compensate\_x}) \cdot z + u_{fx} \end{aligned} \tag{3.44}$$

where

- ω_{fy} is the angular rate measured by Px4flow, unit: rad/s
- $\omega_{compensate\_y}$ is the compensate part, unit: rad/s
- z is the distance from Px4flow to the ground, measured by Marvelmind, unit: m
- u_{fy} is the measurement error of the Px4flow, unit: m/s

ω_{fy} indicates the angular rate of a point moving in the FOV of Px4flow. As we have discussed in 2.3, multiplying ω_{fy} by the distance from the system to the ground z gives the ground speed.

The power spectral density (PSD) of measurement error u_{fy} is shown in figure 3.5. It can be observed that the power of the error is quite uniform along the frequency domain, so it will be regarded as a white noise in this thesis.

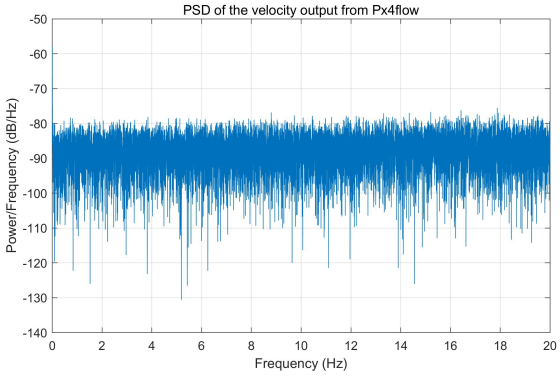


Figure 3.5: PSD of the measurement error from px4flow

$\omega_{compensate\_y}$ is the angular rate caused by the rotation of the system which is calculated by the method in subsection 3.2.3. We assume the Px4flow sensor is facing into the Z axis direction. We take a test vector $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$ which is the unit vector along the Z axis and let this test vector rotates around together with the system's body axis when the system is rotating. Now we suppose the Px4flow has an arbitrary rotation and the corresponding DCM is indicated by C. So the difference between the test vector before and after this rotation can

be expressed as:

$$\vec{d} = C \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.45}$$

In 3.2.3, we employs the quaternion method to calculate the DCM matrix, however, here we would like to directly use the kinetic equation for the DCM to express the relationship between the DCM and the gyroscope data:

$$\dot{C} = C[\omega\times] \tag{3.46}$$

where $[\omega\times]$ is a skew symmetric matrix as this:

$$[\omega\times] = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{3.47}$$

This is a set of 9 differential equations where ω_x, ω_y, ω_z are the angular rates around the body axis of the system (measured by the gyroscope). In the following parts we will use C_{ij} indicates elements of C on the i-th row and the j-th column. A good news is, since we only need to know the components of $\vec{d}$ on X and Y axis, according to Eq. 3.45, only the C_{13} and C_{23} are needed. All the equations in Eq. 3.47 associate with C_{13} and C_{23} are listed below:

$$\begin{aligned} C_{11} &= C_{12}\omega_z - C_{13}\omega_y + u_z - u_y \\ C_{12} &= C_{13}\omega_x - C_{11}\omega_z + u_x - u_z \\ C_{13} &= C_{11}\omega_y - C_{12}\omega_x + u_y - u_x \\ C_{21} &= C_{22}\omega_z - C_{23}\omega_y + u_z - u_y \\ C_{22} &= C_{23}\omega_x - C_{21}\omega_z + u_x - u_z \\ C_{23} &= C_{21}\omega_y - C_{22}\omega_x + u_y - u_x \end{aligned} \tag{3.48}$$

The u_x, u_y, u_z in the equations above are the measurement error of the gyroscope and they are also regarded as white noises (with unit of rad/s). The figure 3.6 shows the PSD for the gyroscope. By solving Eq. 3.48, the C_{13} and C_{23} are computed and according to Eq. 3.45, they are the compensate parts $\omega_{compensate\_y}$ and $\omega_{compensate\_x}$ in Eq. 3.44. So we can rewrite these two equations as the following:

$$\begin{aligned} \dot{x} &= (\omega_{fy} - C_{13}) \cdot z + u_{fy} \\ \dot{y} &= (\omega_{fx} - C_{23}) \cdot z + u_{fx} \end{aligned} \tag{3.49}$$

Equations Eq. 3.49 and Eq. 3.48 are all the equations related to the Px4flow, which describes our estimator. They can be written in a more compact way:

$$\dot{X} = F \cdot X + G \cdot W \tag{3.50}$$

where

$$X = \begin{bmatrix} x & y & \omega_{fx} & \omega_{fy} & C_{11} & C_{12} & C_{13} & C_{21} & C_{22} & C_{23} \end{bmatrix}^T \tag{3.51}$$

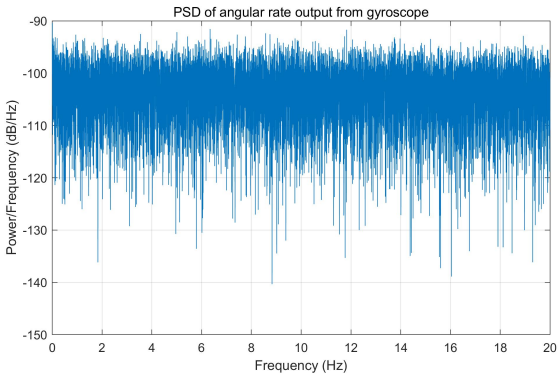


Figure 3.6: PSD for the data from gyroscope

$$G = \begin{bmatrix} z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.52}$$

and

$$W = \begin{bmatrix} u_{fy} & u_{fx} & 0 & 0 & u_z - u_y & u_x - u_z & u_y - u_x & u_z - u_y & u_x - u_z & u_y - u_x \end{bmatrix}^T \tag{3.53}$$

We call F as our system matrix and it is shown as:

$$F = \begin{bmatrix} 0 & 0 & 0 & z & 0 & 0 & -z & 0 & 0 & 0 \\ 0 & 0 & z & 0 & 0 & 0 & 0 & 0 & 0 & -z \\ 0 & 0 & 0 & 0 & 0 & \omega_z & -\omega_y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\omega_z & 0 & \omega_x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_y & -\omega_x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega_z & -\omega_y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\omega_z & 0 & \omega_x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{3.54}$$

The G matrix in Eq. 3.50 is used to modify the noise vector W. It introduces a relationship between the noise of the Px4flow u_{fx}, u_{fy} with the altitude z. So the velocity estimated by the Px4flow will become more noisy when the altitude of the vehicle increases.

Before ending this paragraph, we want to discuss the reason we choose the DCM method in the system model. As we have seen in subsection 3.2.3,

the angular rate compensation algorithm is based on the quaternion method to estimate the attitude. The system model we built in this paragraph is used by the Kalman filter and the Kalman filter expects a linear system model. However, from Eq. 3.27, the rotation relationship between the quaternion and the vector is non-linear. This can surely be solved by using a linearised approximation which we will shortly see in next paragraph. But there is a more simple approach. Recalling the DCM and quaternion is mathematically equivalent, employs the DCM method to replace the quaternion will not actually change the system model. But from Eq. 3.10, the DCM method provides a linear relationship for rotating the vector. This means we can void the non-linear problem using the DCM method, while still keeps the noise characteristic of our system unchanged.

Marvelmind observer model

Now we want to model the observer model for Marvelmind. The measurement from Marvelmind is straight forward: it outputs the distance of the hedgehog to the three fixed beacons. The coordinates of the fixed beacons are indicated as $\begin{bmatrix} a_1 & b_1 & c_1 \end{bmatrix}^T$, $\begin{bmatrix} a_2 & b_2 & c_2 \end{bmatrix}^T$ and $\begin{bmatrix} a_3 & b_3 & c_3 \end{bmatrix}^T$. So the distance to the three beacons D_1, D_2, D_3 are:

$$\begin{aligned} D_1 &= \sqrt{(x-a_1)^2 + (y-b_1)^2 + (z-c_1)^2} + u_{b1} \\ D_2 &= \sqrt{(x-a_2)^2 + (y-b_2)^2 + (z-c_2)^2} + u_{b2} \\ D_3 &= \sqrt{(x-a_3)^2 + (y-b_3)^2 + (z-c_3)^2} + u_{b3} \end{aligned} \tag{3.55}$$

u_{b1} to u_{b3} are the measurement errors of the Marvelmind and, again, are regarded as white noises (unit:m). Their PSD can be found in figure 3.7. Eq. 3.55

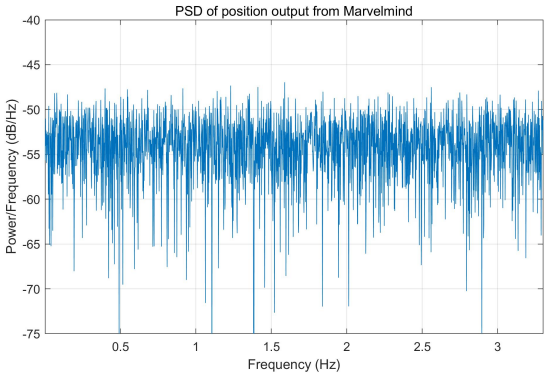


Figure 3.7: PSD of measurement error from Marvelmind

is obviously non-linear. In order to apply the Kalman filter, a linearised form of these equations are required. We employ the Taylor series expansion and the first order expansions of Eq. 3.55 are:

$$\Delta D = J \cdot \Delta X + V \tag{3.56}$$

where Δ indicates it is the "increment" part of the original element. The J in Eq. 3.56 is the Jacobian matrix:

$$J = \begin{bmatrix} \frac{\partial D_1}{\partial x} & \frac{\partial D_1}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial D_2}{\partial x} & \frac{\partial D_2}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial D_3}{\partial x} & \frac{\partial D_3}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.57}$$

And the V in Eq. 3.56 is the measurement error matrix for Marvelmind:

$$V = \begin{bmatrix} u_{b1} & u_{b2} & u_{b3} \end{bmatrix} \tag{3.58}$$

Now we have the whole mathematical model for our combined position estimate system. Finally we can begin building the Kalman filter for our system.

3.3 Extended Kalman filter algorithm

In this combined position estimate system, there exists two Kalman filter. One is used to estimate the attitude using the data from IMU, i.e. the accelerometer, compass and gyroscope; The other one is used to fusion the position estimate from Px4flow and Marvelmind. In this section, we will first introduce the mathematical background of the Kalman filter. After that, the Kalman filter for the attitude estimation will be illustrated, since the attitude information is required by the position estimate. At last, we will discuss the filter for the position estimate.

3.3.1 Background of the Kalman filter

Kalman filter is named after Rudolf E. Kalman, who published the article about this algorithm on prediction problems in 1960. However a similar algorithm, developed by Thorvald N. Thiele and Peter Swerling, was introduced earlier in 1958. [11] The Kalman filter follows the step of Wiener filter, which uses the minimum mean-square error methods to find a best estimate of a noisy signal. However Kalman filter employs the state space model to analysis the system and solves the problem in discrete time domain. These features make the Kalman filter very adaptable to the computer technology. One of the very first applications of the Kalman filter is the Apollo program [15] in 1960s. After that, the Kalman filter is widely used in guidance and navigation systems. Further more, many algorithms, e.g. extended Kalman filter, particle filter, unscented Kalman filter, based on the Kalman filter are developed to expand its capability in varies fields.

The Kalman filter is a recursive algorithm. We first consider a system process can be modelled into the following discrete state space form:

$$x_{k+1} = \Phi_k x_k + w_k \tag{3.59}$$

where

- x_k is a $n \times 1$ vector stands for the process state vector at time t_k.
- Φ_k is a $n \times n$ matrix stands for the state transition matrix that evolutes the x_k to x_{k+1}
- w_k is a $n \times 1$ vector stands for a white noise sequence whose covariance is known.

We also suppose there is an observation of the system measuring the system states at discrete time points, which can be expressed as:

$$z_k = H_k x_k + v_k \tag{3.60}$$

where

- z_k is a $m \times 1$ vector stands for the observation of the system at time t_k
- H_k is a $m \times n$ matrix relates the system state x_k to the observation z_k at time t_k
- v_k is a $m \times 1$ vector stands for the observation error whose covariance is known and has zero cross-correlation with the w_k

The covariance matrix for w_k and v_k are expressed in Q_k and R_k as:

$$E[w_k w_i^T] = \begin{cases} Q_k, & i = k \\ 0, & i \neq k \end{cases} \tag{3.61}$$

$$E[v_k v_i^T] = \begin{cases} R_k, & i = k \\ 0, & i \neq k \end{cases} \tag{3.62}$$

$$E[w_k v_i^T] = 0, \text{ for all } k \text{ and } i \tag{3.63}$$

Now if we use $\hat{x}_k^-$ stands for an estimation of system state and define the estimation error as:

$$e_k^- = x_k - \hat{x}_k^- \tag{3.64}$$

where the hat symbol '^' in $\hat{x}$ indicates that this is an estimate value of x. The '$-$' superscript means this estimation is based on all the knowledge before time t_k, and we will shortly see how this estimation can combine with the information presented at time t_k. With the estimation error defined, we can write the error covariance matrix as:

$$P_k^- = E[e_k^- e_k^{-T}] = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \tag{3.65}$$

For now, we have an estimation $\hat{x}_k^-$ for the system state at time t_k, and this estimate is based on the information before t_k, so it is called a prior estimate. We are seeking a method to use the observation z_k to correct $\hat{x}_k^-$. This can be done by:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-) \tag{3.66}$$

where

- $\hat{x}_k$ is a $n \times 1$ vector stands for the corrected estimation of the system states.
- K_k is a weight factor needs to be determined

Suppose we have found a weight factor K_k to correct the system estimation, then the error covariance matrix after the correction is:

$$P_k = E[e_k e_k^T] = E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T] \tag{3.67}$$

Notice in P_k there is no '$-$' superscript any more, since the estimation $\hat{x}_k$ also contains the information from observation z_k at time t_k. Since we are trying to find the MMSE solution of $\hat{x}_k$, this is equally to say that at time t_k, find an weight factor K_k that minimize the P_k. As it is shown in [2], the optimized weight factor for the MMSE solution can be expressed as:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{3.68}$$

This optimized weight factor is called the *Kalman gain*. Substitute Eq. (3.66) and Eq. (3.68) into Eq. (3.67), with some rearrangement, the updated error covariance matrix can be expressed in a compact form as [2]:

$$P_k = (I - K_k H_k) P_k^- \tag{3.69}$$

This updated covariance matrix can be used as a prior error matrix in the following time step t_{k+1}. Now we can summarize the Kalman filter routine into the following four steps:

Step 1 Compute the Kalman gain for the current time step:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{3.70}$$

Step 2 Update estimation for the current time step:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{X_k}^-) \tag{3.71}$$

Step 3 Update error covariance for the current time step:

$$P_k = (I - K_k H_k) P_k^- \tag{3.72}$$

Step 4 Project ahead estimations and error matrix for the next time step:

$$\hat{x}_{k+1} = \Phi_k \hat{x}_k \tag{3.73}$$

$$P_{k+1}^- = \Phi_k P_k \Phi_k^T + Q_k \tag{3.74}$$

This routine is also shown in figure 3.8. However in practical, the state space model and observation model of a system are not always linear as Eq. (3.59) and Eq. (3.60). Some non-linear method should be used combining with the Kalman filter routine. A straight forward method is called an Extended Kalman filter (EKF) which employs the Jacobian matrix and Taylor series expansion to linearise the system near a specific point. This EKF algorithm is used in this thesis and the detail will be discussed in the following.

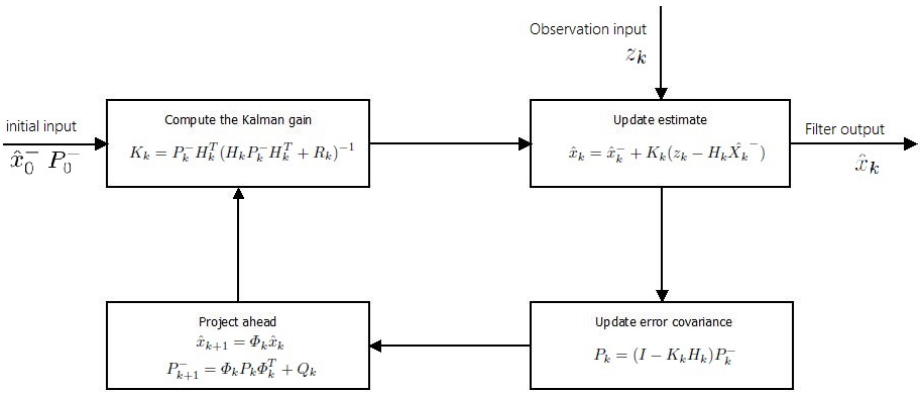


Figure 3.8: Kalman filter routine

3.3.2 Kalman filter for attitude estimate

As we have already discussed in section 3.3.1, the Kalman filter is based on the discrete space state model. From section 3.2.2, we have already build a continuous space state model to estimate the attitude. Recall the attitude estimator model is:

$$\dot{\mathbf{q}} = F_q(\boldsymbol{\omega})\mathbf{q} \tag{3.75}$$

where

$$F_q(\boldsymbol{\omega}) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix} \tag{3.76}$$

We need to build a discrete model according to this. As describes in [19], this can be done by the Van Loan method as:

Step 1 construct the matrix A:

$$A = \begin{bmatrix} -F_q & P \\ 0 & F_q^T \end{bmatrix} \cdot \Delta t \tag{3.77}$$

where P is the power spectral density matrix associate with W and Δt is the time interval between two discrete samples.

Step 2 construct the matrix B from A:

$$B = expm(A) = \begin{bmatrix} \cdots & \Phi^{-1}Q \\ 0 & \Phi^T \end{bmatrix} \tag{3.78}$$

where $expm(A)$ is the matrix exponential function of A.

Step 3 From Eq. 3.78, Φ is the transpose of the lower right part of B. After we gather the Φ, multiplying it's inverse to the upper right part gives the Q. The $\cdots$ in Eq. 3.78 just means the upper left part of the matrix is not used in our method so we do not care its value.

So the discrete space model according to Eq. 3.75 is:

$$\mathbf{q}_{k+1} = \Phi_k \cdot \mathbf{q}_k + W_k \tag{3.79}$$

where

- $\mathbf{q}_k$ is a 4×1 vector stands for the sampled quaternion at time point t_k.
- W_k is a 4×1 vector stands for the gyroscope error at time point t_k, whose covariance matrix is defined as:

$$Q_k = E[W_k \cdot W_k^T] \tag{3.80}$$

Besides the estimator, we also need a discrete version model for the observer. Recall the space model for the observer in section 3.2.2 is:

$$V = [C\ C]V' \tag{3.81}$$

where

$$C = \begin{bmatrix} (q_1^2 + q_2^2 - q_3^2 - q_4^2) & 2(q_2q_3 + q_1q_4) & 2(q_2q_4 - q_1q_3) \\ 2(q_2q_3 - q_1q_4) & (q_1^2 - q_2^2 + q_3^2 - q_4^2) & 2(q_3q_4 + q_1q_2) \\ 2(q_2q_4 + q_1q_3) & 2(q_3q_4 - q_1q_2) & (q_1^2 - q_2^2 - q_3^2 + q_4^2) \end{bmatrix} \tag{3.82}$$

and

$$V = \begin{bmatrix} a_1 & a_2 & a_3 & c_1 & c_2 & c_3 \end{bmatrix}^T \tag{3.83}$$

However, in our Kalman filter, we regard the quaternion q as the state vector. According to Eq.3.81, the observer model is a non-linear model with regards to q. So we will linearise Eq. 3.81 using the first order Taylor series expansion. The linearised observer takes the form as:

$$V_k = H_k\mathbf{q}_k + N_k \tag{3.84}$$

where

- V_k is a 6×1 vector, it is the observe vector stands for the data from accelerometer and compass at time point t_k
- $\mathbf{q}_k$ is a 4×1 vector, it is the system vector stands for the quaternion estimated by gyroscope at time point t_k
- N_k is a 6×1 vector stands for the observer error for the accelerometer and compass, the covariance matrix for N_k is:

$$R_k = E[N_k \cdot N_k^T] \tag{3.85}$$

- H_k is a 6×4 matrix, used to relate the system vector $\mathbf{q}_k$ with the observe vector V_k. It is the Jacobian matrix for $V = CV'$ with regards to $\mathbf{q}$:

$$H_k = \frac{\partial V'}{\partial \mathbf{q}} = \begin{pmatrix} \frac{\partial V_1}{\partial q_1} & \frac{\partial V_1}{\partial q_2} & \frac{\partial V_1}{\partial q_3} & \frac{\partial V_1}{\partial q_4} \\ \frac{\partial V_2}{\partial q_1} & \cdots & \cdots & \frac{\partial V_2}{\partial q_4} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial V_6}{\partial q_1} & \frac{\partial V_6}{\partial q_2} & \frac{\partial V_6}{\partial q_3} & \frac{\partial V_6}{\partial q_4} \end{pmatrix} \tag{3.86}$$

  V_i in Eq. 3.86 stands for the i-th row in vector V

Eq. 3.79 and Eq. 3.84 form the discrete model for the attitude estimate system. And now we can begin the Kalman filter routine on this discrete model:

Step 1 At time t_k, read data from gyroscope, calculate the quaternion $\hat{\mathbf{q}}_k$ using Eq. 3.75. Read data from accelerometer and compass, build the observer vector V_k.

Step 2 Substrate $\hat{\mathbf{q}}_k$ with the quaternion estimate from last time point $\hat{\mathbf{q}}_{k-1}$ to get the increment of the attitude:

$$\Delta\hat{\mathbf{q}}_k = \hat{\mathbf{q}}_k - \hat{\mathbf{q}}_{k-1} \tag{3.87}$$

. Substrate V_k with the observer vector from last time point V_{k-1} to get the increment of the observation:

$$\Delta V_k = V_k - V_{k-1} \tag{3.88}$$

Step 3 Compute the Kalman gain:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{3.89}$$

Step 4 Update estimation for the increment of the quaternion:

$$\Delta\hat{\mathbf{q}}_k = \Delta\hat{\mathbf{q}}_k^- + K_k(\Delta V_k - H_k \Delta\hat{\mathbf{q}_k}^-) \tag{3.90}$$

Step 5 Add the updated estimation $\Delta\hat{\mathbf{q}}_k$ back to $\mathbf{q}_{k-1}$ to get the total estimation for the quaternion $\hat{\mathbf{q}}_k = \Delta\hat{\mathbf{q}}_k + \mathbf{q}_{k-1}$

Step 6 Update error covariance for the current time step:

$$P_k = (I - K_k H_k) P_k^- \tag{3.91}$$

Step 7 Project ahead estimations and error matrix for the next time step:

$$\hat{\mathbf{q}}_{k+1} = \Phi_k \hat{\mathbf{q}}_k \tag{3.92}$$

$$P_{k+1}^- = \Phi_k P_k \Phi_k^T + Q_k \tag{3.93}$$

In step 1,The quaternion $\hat{\mathbf{q}}_k$ estimated from gyroscope is the estimator input of the Kalman filter. The vector V_k built by the accelerometer and compass data is the observer input. The total estimation for the quaternion $\hat{\mathbf{q}}_k$ in step 5 is the desired filtered attitude estimation at t_k, which is our system output. Note that the matrix Φ_k should be evaluated at each time point t_k using the Van Loan method introduced above.

3.3.3 Kalman filter for position estimate

As same as we have discussed in section 3.3.2, a discrete model with regards to the position estimate model shall be built before the Kalman filter routine. For convenience, we rewrite the position estimate model in section 3.2.4:

$$\dot{X} = F \cdot X + W \tag{3.94}$$

To build the discrete model, we employs the Van Loan method in section 3.3.2 again. And the corresponding discrete model for the position estimation is:

$$X_{k+1} = \Phi_k \cdot X_k + W_k \tag{3.95}$$

where

- X_k is a 10×1 vector stands for the system vector we get from Px4flow and gyroscope in Eq. 3.51 at time t_k.
- Φ_k is a 10×10 matrix stands for the discrete system model with regards to matrix F.
- W_k is a 10×1 vector stands for the Px4flow error at time t_k, whose covariance matrix is defined as:

$$Q_k = E[W_k \cdot W_k^T] \tag{3.96}$$

The discrete observer model for the Marvelmind is also required. According to Eq. 3.56, the model is:

$$D_k = J_k \cdot X_k + Z_k \tag{3.97}$$

where

- D_k is a 3×1 vector stands for the distance from hedgehog to each beacon measured by Marvelmind at time t_k
- Z_k is a 3×1 vector stands for the Marvelmind error and the covariances matrix of it is:

$$R_k = E[Z_k \cdot Z_k^T] \tag{3.98}$$

- J_k is a 3×10 matrix used to relate the system vector X_k with the observe vector D_k. It is a Jacobian matrix as:

$$J = \begin{bmatrix} \frac{\partial D_1}{\partial x} & \frac{\partial D_1}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial D_2}{\partial x} & \frac{\partial D_2}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial D_3}{\partial x} & \frac{\partial D_3}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.99}$$

Eq. 3.95 and Eq. 3.97 form the discrete model for the position estimation system. Now we can begin the Kalman filter routine:

Step 1 At time t_k, read data from Px4flow, calculate the system vector $\hat{X}_k$ using Eq. 3.48 and Eq. 3.49. Read data from accelerometer and compass, build the observer vector D_k.

Step 2 Substrate $\hat{X}_k$ with the position estimate from last time point $\hat{X}_{k-1}$ to get the increment of the attitude:

$$\Delta\hat{X}_k = \hat{X}_k - \hat{X}_{k-1} \tag{3.100}$$

. Substrate D_k with the observer vector from last time point D_{k-1} to get the increment of the observation:

$$\Delta D_k = D_k - D_{k-1} \tag{3.101}$$

Step 3 Compute the Kalman gain:

$$K_k = P_k^- J_k^T (J_k P_k^- J_k^T + R_k)^{-1} \tag{3.102}$$

Step 4 Update estimation for the increment of the position:

$$\Delta\hat{X}_k = \Delta\hat{X_k^-} + K_k(\Delta D_k - J_k\Delta\hat{X_k^-}) \tag{3.103}$$

Step 5 Add the updated estimation $\Delta\hat{X}_k$ back to X_{k-1} to get the total estimation for the position $\hat{X}_k = \Delta\hat{X}_k + X_{k-1}$

Step 6 Update error covariance for the current time step:

$$P_k = (I - K_k J_k)P_k^- \tag{3.104}$$

Step 7 Project ahead estimations and error matrix for the next time step:

$$\Delta\hat{X_{k+1}} = \Phi_k\Delta\hat{X}_k \tag{3.105}$$

$$P_{k+1}^- = \Phi_k P_k \Phi_k^T + Q_k \tag{3.106}$$

In step 1, the system vector X_k is gathered by the data from Px4flow, and it is the estimator input for the Kalman filter. The D_k is gathered by the data from Marvelmind, which is the observer input. Again, the matrix Φ_k should be evaluated at each time point t_k using the Van Loan method introduced above.

3.4 Simulation of the Kalman filter for position estimation

In this section, a simulation of the Kalman filter for the position estimation will be discussed. This simulation is performed using Simulink under the Matlab environment. All the data used in the simulation is generated by software, rather than the real data gather from the sensors. We will first introduce the structure of the Kalman filter and then proceed to the simulation results.

3.4.1 Kalman filter structure

One thing that we need to notice is the Kalman filter is an algorithm designed on the time domain. Its design procedure is quite different as the traditional filter (e.g. a low-pass filter) which analyses the system on the frequency domain, produces a transfer function and converts it back to the time domain. We design the Kalman filter by analysing the noise characteristic of its input. So in order to implement the Kalman filter, it is essential to figure out the filter structure and make it clear what are the input and output of the filter.

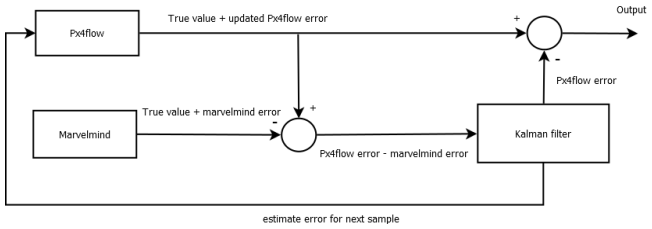


Figure 3.9: Data flows of the Kalman filter

The structure of our filter is shown in figure 3.9. It is a close loop feedback structure. The outputs from Px4flow and Marvelmind both contains the true information of the position, but also contains the measurement errors. The two outputs first subtract with each other so now we cancel out the true information contains in the two outputs. This means our Kalman filter will only deal with the errors from Px4flow and Marvelmind. After the filtering, the error from Px4flow will be extracted out and the subtract back to the original output of the Px4flow. The feedback is the estimated error for next sample, which is subtract to the Px4flow's output on the next sample time. In this way the output from Px4flow is corrected a little before comparing with the Marvelmind. The reason we employ this structure is, in the measurement model we made in section 3.3, only the first order Taylor series expansion is used to linearise the measurement model. If the difference between the Px4flow and Marvelmind is too large, then the approximation of the first order Taylor expansion can be bad. The feedback structure can reduce the difference between the two sensors and somehow improves the accuracy of the linearisation.

Another question of our filter structure maybe: why we take the combination of two errors as input of the Kalman filter, instead of directly sending the outputs from Px4flow and Marvelmind into the filter. This is also because we are using the Taylor expansion to linearise the model, and the Taylor expansion takes the increment part of the data as input. By subtracting the two sensor's outputs, the output from Px4flow can be regard as a base trajectory and the system model approximately works linearly in a very small increment interval around this base trajectory.

3.4.2 Simulation results

In order to verify the Kalman filter we build for the position estimate, a simulation is established in Simulink. In this simulation, all the data are generated in the software follows the system model we discussed in section 3.2. Figure 3.10 shows the noise model used in the simulation. In the noise model of Px4flow, the optical flow data first adds a white noise which stands for the measurement error of Px4flow. Then the attitude and angular rate compensate algorithm apple to the optical flow data. The noise of IMU is considered in this operation. After that, multiply the distance to ground and integral over time, the simulated position estimate of Px4flow is produced.

On the other hand, the simulated Marvelmind data is more easy to produce. We first begin with a position of the hedgehog and compute its distance to each of the beacons. Then a white noise is added as the measurement error of Marvelmind, thus provides the simulated distance data measured by the Marvelmind.

The power of the noise for both sensor (Px4flow and Marvelmind) are set to be equal and the position of hedgehog is set to be [0.127,-0.95]. Figure 3.11 shows the overall Simulink model of the simulation. In this simulation we assume the UAV remains fixed without any rotation or movement. So it simulates the stationary situation of the Kalman filter. The simulation time is 600 seconds and figure 3.12 shows the position output on X axis with regard to time. The situation on Y axis is similar so it is not shown here to avoid redundant. From figure 3.12 it can be observed that the output from Kalman filter gives an overall better estimate than the other sensors. The drift error from Px4flow has be eliminated and the result is much more smooth than the output from Marvelmind.

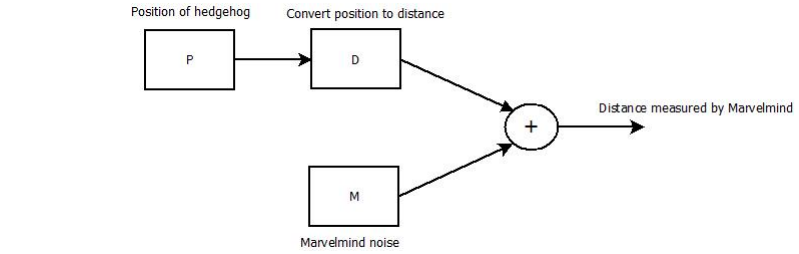
Figure 3.13 gives a position output from the three sources in a Cartesian coordinate. In this figure, the drift phenomena of the Px4flow is even more obvious. Further more, the output range from the Kalman filter and Marvelmind is similar (within a range of $1cm^2$), but the output from Kalman filter is more smooth, which means the outputs are more close to each other so they can build up a continuous line. Table 3.1 shows the standard deviation from the

Data source	Px4flow	Marvelmind	Kalman filter
Standard deviation (unit: meter)	0.0099	0.0013	0.0010

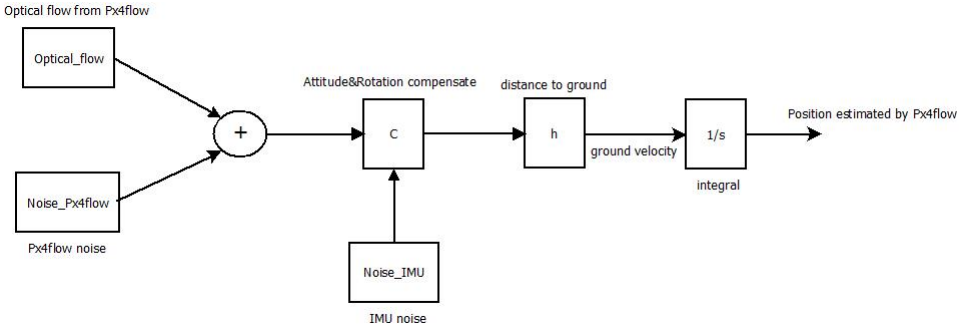
Table 3.1: Simulated standard deviation of different sources

two sensors and the filter, notice that the standard deviation of the Px4flow will keep growing with time and the value in the table is just the standard deviation when the test length is 10 minutes.

As a conclusion, the Kalman filter gives an overall better result than Px4flow



(a) Noise model of Marvelmind



(b) Noise model of Px4flow

Figure 3.10: Noise models used in simulation

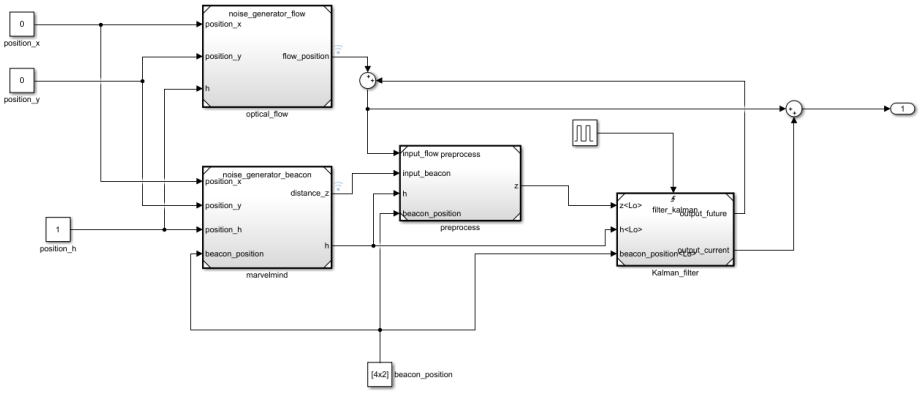


Figure 3.11: Simulink model

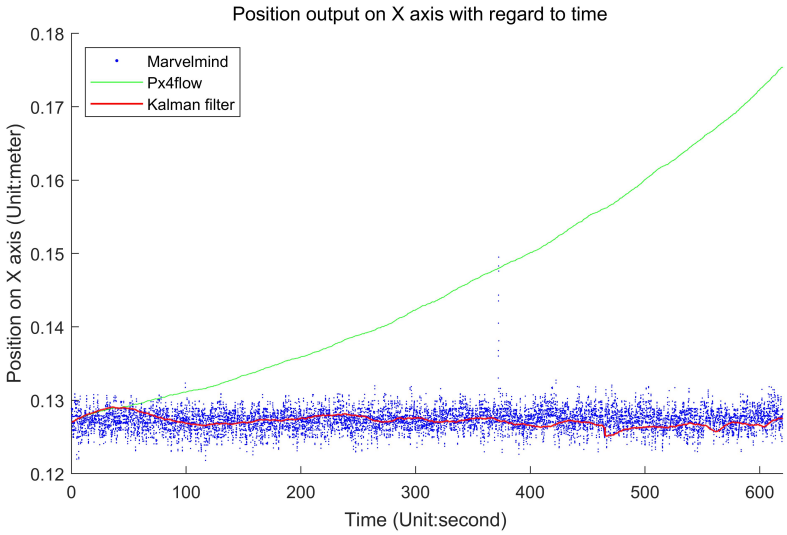


Figure 3.12: Simulated position output on X axis

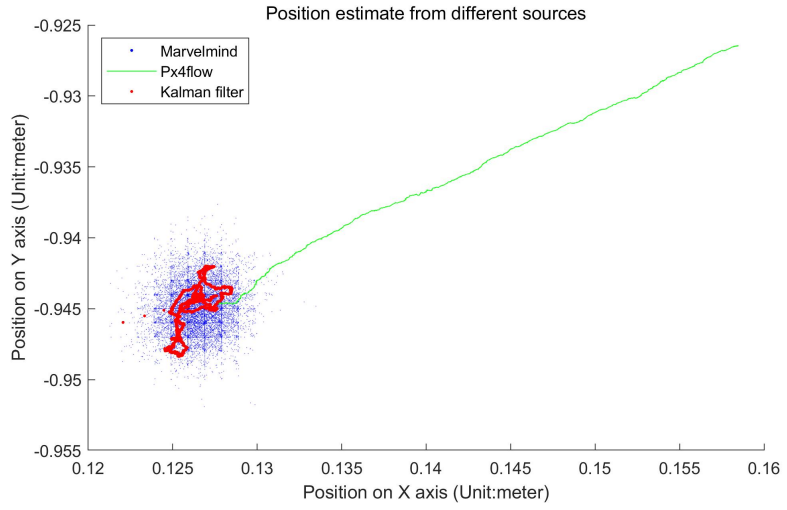


Figure 3.13: Simulated position output from different sources

and Marvelmind in this simulation. The deviation improvement with regard to Px4flow is notable while it is not that obvious compare to the Marvelmind. However if we also consider the output on the time domain, then the Kalman filter gives a more smooth result than Marvelmind. However, the noise power from Px4flow and Marvelmind are set to be equal in this simulation while practical this is not true. In section 4.3 we will test the Kalman filter on-board and compare the result with the simulation one shown in this section.

Chapter 4

Implementation

4.1 Introduction

In the previous chapters, we have introduce the hardware used in the combined position estimate system. We also build the system models for our system, together with the Kalman filter based on the modelling. Now it is time to implement all the algorithms on the central controller in our system, which is the Raspberry Pi single board computer. The program structure will firstly be introduced in section 4.2, then some on-board tests for all the algorithms, including the angular compensate, the attitude compensate and the position estimate, are performed and discussed in section 4.3. All the codes, including the C language and Python, will be included in the appendix.

4.2 Realization of data fusion algorithm on Raspberry Pi

4.2.1 Code generation

We have discussed in section 2.4 that, for computing efficiency purpose, the C language will be employed to implement the mathematical parts for our algorithm. However we will first implement the algorithm as a function in Matlab since the Matlab performs an advanced environment for testing and debug. After the code is tested in the Matlab, we employs an automatic code generation method to build the C functions. This automatic method is an application called "Matlab Coder" provided by MathWorks.

The Matlab Coder can generate the C source code, the static library and the dynamic-link library for variety types of platforms, such as Intel, ARM, Atmel, AMD, etc. However when generates the static library or dynamic library, the Matlab Coder can only generate the library files for the operation system the Matlab is currently running on. This means if we are using Matlab in Windows system at a X86 platform, then the Matlab Coder can only generate the .lib or

.dll files, which cannot be used in Linux on Raspberry Pi. So in this thesis, we decide to generate the C source code using Matlab Coder and compile it directly in the Linux system on Raspberry Pi. Figure 4.1 briefly shows the idea of how the C code is generated.

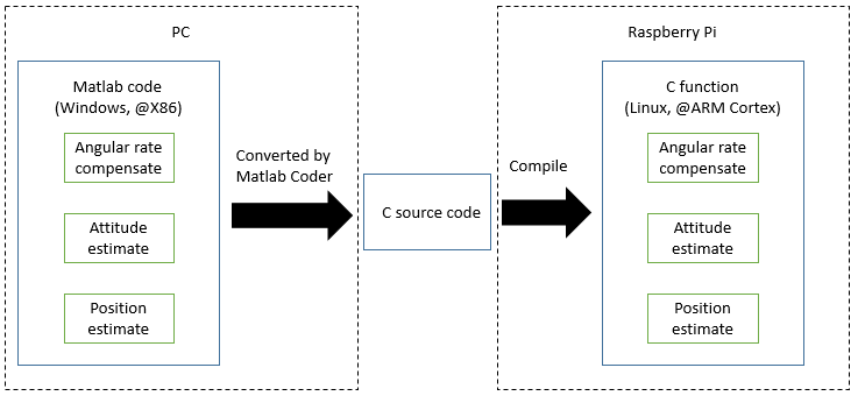


Figure 4.1: Generate C function from Matlab code

4.2.2 Overall program structure

To build the main logic part of the combined measurement system, which means the program that deals with the Input/Output transmission, file Reading/Writing, converting the unit of the data, etc, we decide to use the Python. This is because, comparing to the C, the Python is an Object-oriented program language, which is more user-friendly and easy to read. Remember the angular compensation and Kalman filter algorithm is written in C, so we need to find a way to use them in Python. Luckily the Python provide a C API which can handle this. By using the C API we just need to write an entry function for Python so we can use these two C program in Python as a normal function call.

The C API for Python is actually a package of C functions, which can be used to build a entry function. This entry function is in charge of converting the Python objects into the C data types. When a Python program calls the C function, it will first push its arguments into a stack, and pass the stack pointer to the entry function. The entry function pops the stack sequentially and assert each object to a specific C data types, then calls the actual C function using these converted arguments. The C function will process these converted data and then send the data back to the entry function again. Then entry function converts the data back to the desired Python object and store them in the stack. Finally the Python program reads the processed return value by popping the stack. When building the entry function, the developer should arrange the data types for each object carefully or a memory overflow may occur.

To sum up, figure 4.2 shows the structure of the programs. The dashed box in the figure stands for the program and the language used is indicated in the parentheses, while the solid box stands for a hardware. The output of the system can be either write into a file, to the console screen, or to other devices through any protocol supported by the Raspberry Pi.

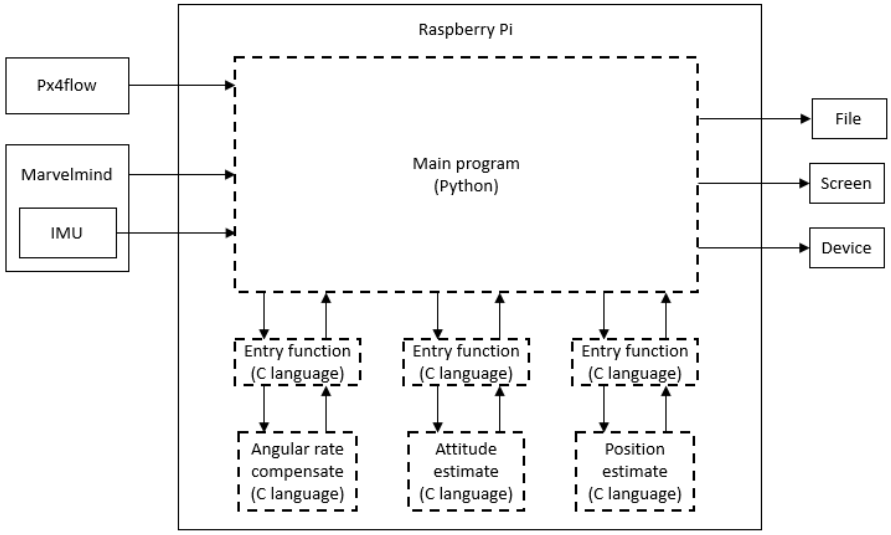


Figure 4.2: Program structure

Figure 4.3 shows a flow chart for the main Python program. After the system boots up, the program will first initialise the system, including configure the USB and I2C protocol, setting initial values used by the algorithms, creating files which will be used to store the filtered results, etc. Then the program goes into the following loop:

Step 1 The main program reads a packet from Marvelmind, examines whether its an IMU packet or a position packet. If it is an IMU packet, the program goes into step 2, otherwise it goes into step 6.

Step 2 The main program calls the attitude Kalman filter function to calculate the attitude of the UAV, based on the IMU data, then goes to step 3.

Step 3 The main program reads data from Px4flow, then goes to step 4.

Step 4 The main program calls the angular rate compensate function to counter the measurement error caused by the rotation of the UAV, then goes to step 5.

Step 5 The main program estimate the position from the processed Px4flow data, save the data into a global variable, then goes back to step 1.

Step 6 The main program reads the processed position estimation from the Px4flow, then goes to step 7.

Step 7 The main program calls the position Kalman filter function to fusion the data from Px4flow and Marvelmind, then goes to step 8.

Step 8 The main program post-process the fusioned position estimation (changing the units, arrange the data type, etc.), then output the result to file/screen/other devices based on the requirement. After this, the program goes back to step 1.

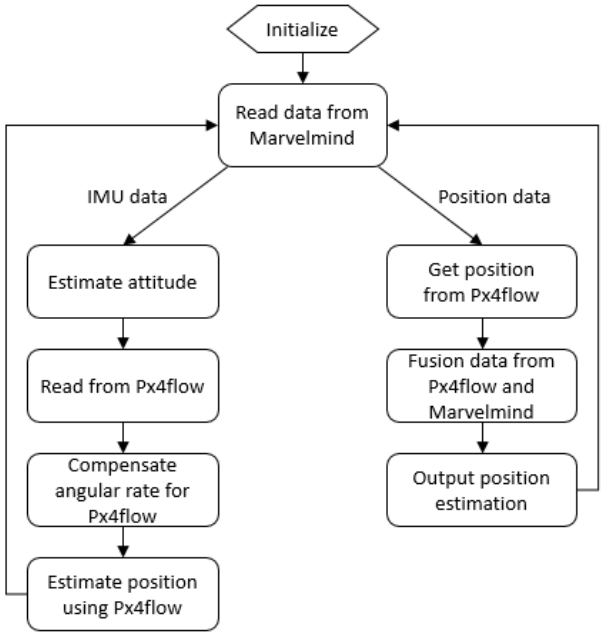


Figure 4.3: Main program flow chart

As a brief conclusion, in this section, the combined position estimate system have been implemented onto the Raspberry Pi. Some on-board tests have been made for this system and we will discuss their result in the following section.

4.3 Test result and analysis

In this section, some tests of the algorithms we discussed in chapter 3 will be provided. All the tests are on-board tests which means the algorithms are performed directly on the Raspberry Pi and then record the results. The results then are post-processed in Matlab on the PC to produce a readable image or compute the numerical features such as variance.

4.3.1 Angular rate compensate test

The algorithm tested in this subsection is the angular rate compensate test. Firstly a stationary test is performed for the algorithm which means we will test the performance when the Px4flow stays at a fixed location and rotates around its diagonal. Figure 4.4 shows the rotation axis of the test. During the test, Px4flow rotates around the diagonal from $-15°$ to $+15°$ at a frequency of about 1 Hz.

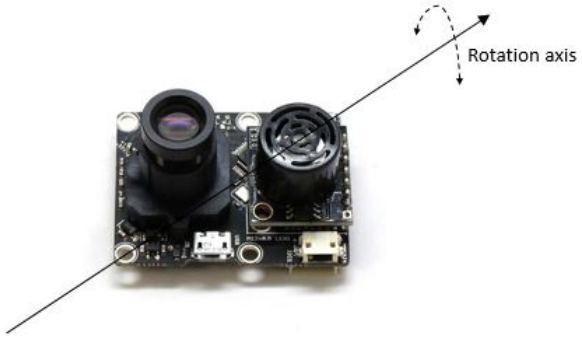


Figure 4.4: Px4flow rotates around its diagonal

Figure 4.5 shows the result for the test discussed above. To avoid redundant, the figure only shows the output on X axis of Px4flow since the the situations on both axis are similar. The red data in figure 4.5 shows the angular rate calculated from the IMU of Marvelmind while the blue data is the output of the Px4flow. Yellow data is the output of the compensate algorithm. Notice the data is sampled at 50 Hz so the maximum frequency in the power spectrum image is 25 Hz. From the power spectrum image it can be observed that the rotation of the sensor has been attenuated for about $17dB$. However from the image of time domain, we can find there still some periodic output remains after the compensation, which relates to the rotation. This phenomenon will be discussed later in this subsection.

We have also tested the performance of the algorithm while the Px4flow is moving, thus, the Px4flow does not only perform the same rotation as the stationary test, but also move along its diagonal as in figure 4.4. The test is made by moving the sensor along a roughly straight path and the rotation is performed manually. Figure 4.6 shows the test results. Notice that in this test, the altitude measured by the Marvelmind is used so the output is the position estimate from Px4flow. From the results, the improvement of the compensate algorithm can be observed easily. Further more, we can still find some remaining rotation phenomenon as we saw in the stationary test.

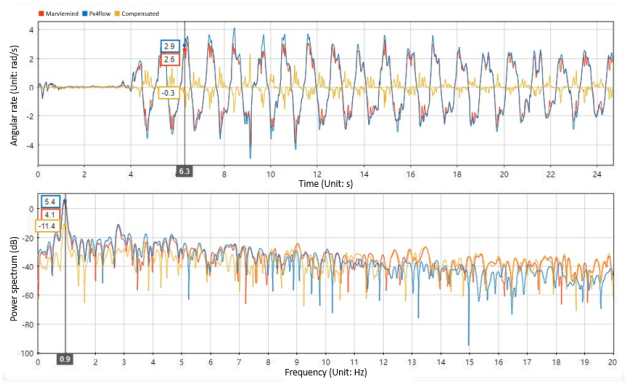
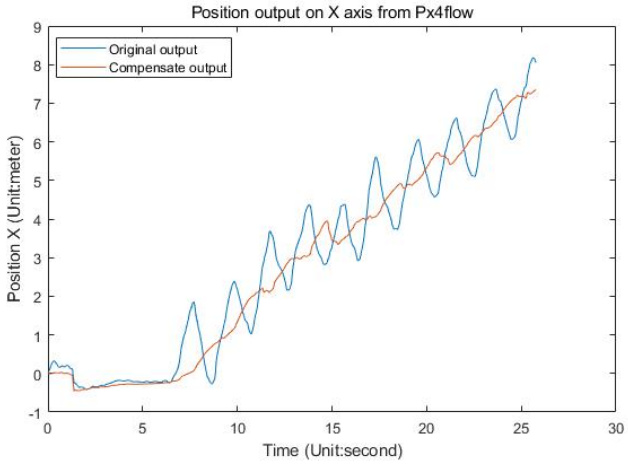
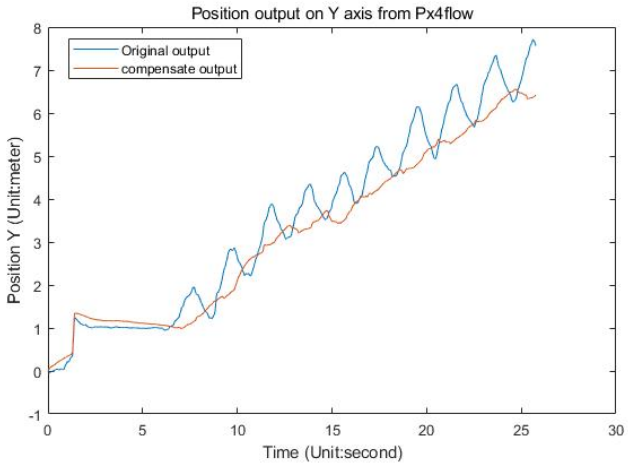


Figure 4.5: Stationary test of angular rate compensate



(a)



(b)

Figure 4.6: Angular rate compensate test while Px4flow is moving

As a conclusion, the angular rate compensate algorithm does provide a decent attenuation on the sensor rotations. However it can not perfectly remove all the influences introduced by the rotation. This is partially caused by the sensor noises from both Px4flow and IMU. On the other hand, since rotation of the system is guaranteed to centre at the Px4flow, the Px4flow does have a small movement with regards to the rotation centre of the system, which is shown in figure 4.7. In the figure, suppose the system rotate around the

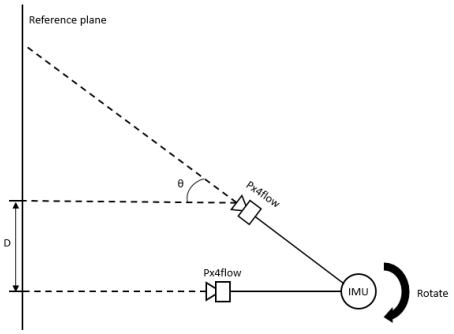


Figure 4.7: Distance between Px4flow and rotation centre causes extra error

position of the IMU (Marvelmind), angle θ will be compensated by the angular rate compensate algorithm we discussed above. However the displacement D in the figure is caused by the distance between the two sensors so it can not be compensate by our method. Furthermore, this displacement will increase along with the distance between the Px4flow and the centre. So in practical, it is suggested that the Px4flow should be installed as close as possible to the rotation centre of the UAV.

4.3.2 Attitude estimate test

In this subsection we will show some tests for the Kalman filter of attitude estimate introduced in section 3.3.2. In order to reduce influence from external electromagnetic (EM) field, we connect the Marvelmind with Raspberry Pi with a long USB cable (1 metre). Furthermore, since the Px4flow does not provide any data in this algorithm, we did not connect the Px4flow to the Raspberry Pi in the tests. All the unnecessary metal objects (knives, keys, rulers, etc.) have been kept away from Marvelmind.

The first test is a stationary test which the Marvelmind is located at a fixed position without any movement or rotation. The Euler angle of a test vector is computed separately by the data from compass, from gyroscope and from the Kalman filter. Figure 4.8 shows the orientation of the test vector and the result is shown in figure 4.9.

The actual test last for 1800 seconds and only the first 100 seconds is shown here for a brief view. From the result it can be found that the output from the

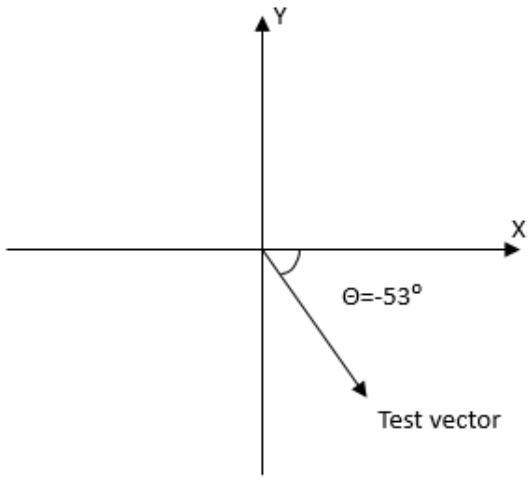


Figure 4.8: Orientation of the test vector

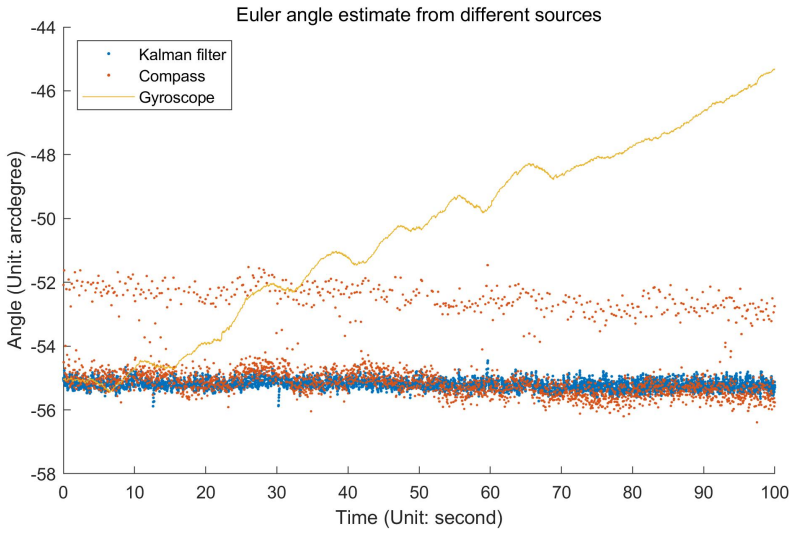


Figure 4.9: Stationary test for the attitude estimation

compass keeps jumping between $-52°$ to $-55°$. This may be caused by the electromagnetic field inside the Marvelmind. It can be also observed that the output from Kalman filter is much more steady than the compass. The output from gyroscope, however shows a significant drift error.

The second test is done by put the Marvelmind at a horizontal plan and rotate the system manually around the vertical axis by $90°$ in about 5 seconds. The rotation angle in the X-Y plan is estimated by the data from different sources. The result is shown in figure 4.10. It can be observed that even in a relatively short time interval (5 seconds), the drift error from the gyroscope can not be ignored. The output from Kalman filter shows an overall better estimation in both stationary and rotation tests.

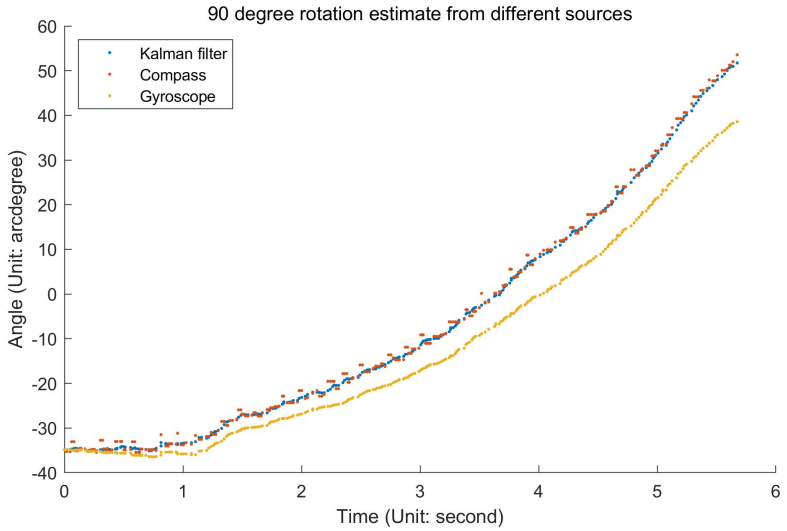


Figure 4.10: Euler angle estimate when the system rotates $90°$

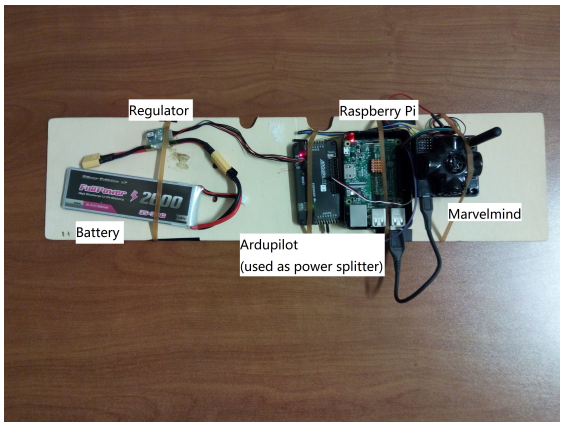
The standard deviation of the different data sources is shown in table 4.1. The output from the gyroscope is not counted in the table because the drift error of the gyroscope will keeps increasing as the time goes by without boundary, so it is meaningless to measure the standard deviation of the unbounded error. The performance of the compass and accelerometer is similar. In contrast, the Kalman filter provides a improvement of the accuracy for around 55%. As a conclusion, the Kalman filter designed for the attitude estimation in this thesis is efficient.

	Compass	Accelerometer	Kalman filter
Standard deviation	0.8532°	0.7503°	0.3522°

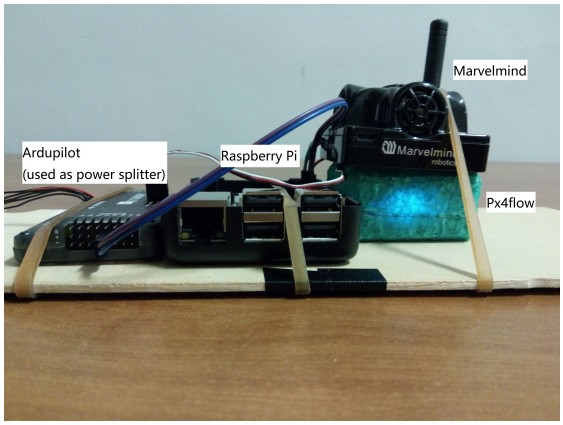
Table 4.1: Standard deviation for the estimated Euler angle from different sources

4.3.3 Position estimate test

In order to test the position estimate algorithm, all the components are settled at a bench. The Marvelmind and Px4flow coincide vertically to reduce the rotation compensate error caused by the distance between two sensors, as we discussed in section 4.3.1. At the location of the Px4flow, there is a hole on the bench so the camera can record images through it. Figure 4.11 shows the test system. Notice that there is a component called Ardupilot in the test system. Originally Ardupilot is an open-sourced control module for UAV. However in this implementation, it is only used as a power splitter. In our experiment, the Raspberry Pi is not capable to provide a stable power to the Marvelmind, so we use the Ardupilot to split the power from the regulator and power up the Raspberry Pi and Marvelmind separately. On the other hand, the Px4flow can work without any problem by powering from Raspberry Pi so it does not require a separate power line.



(a)



(b)

Figure 4.11: Position estimate test system

The first test will be shown is a qualitative analysis, which means this test is not by the meaning of "accurate". The test is done by manually moving the test system around a roughly square path. Figure 4.12 shows the position estimate from different sources in a Cartesian coordinate. The arrows at the corners indicate the test system rotate 90° clockwise. It can be observed that the drift error of the Px4flow is dramatic. At some position the estimation from Marvelmind will "jump" out of the path. This is because the ultrasonic beam is blocked by the body of human who is holding the system. From the output of the Kalman filter we can tell that the drift error has been eliminated well. When the estimate from Marvelmind jumps out of the path, the Kalman filter will firstly be influenced by the Marvelmind and then converge back to the path as soon as the Marvelmind provides a normal estimation.

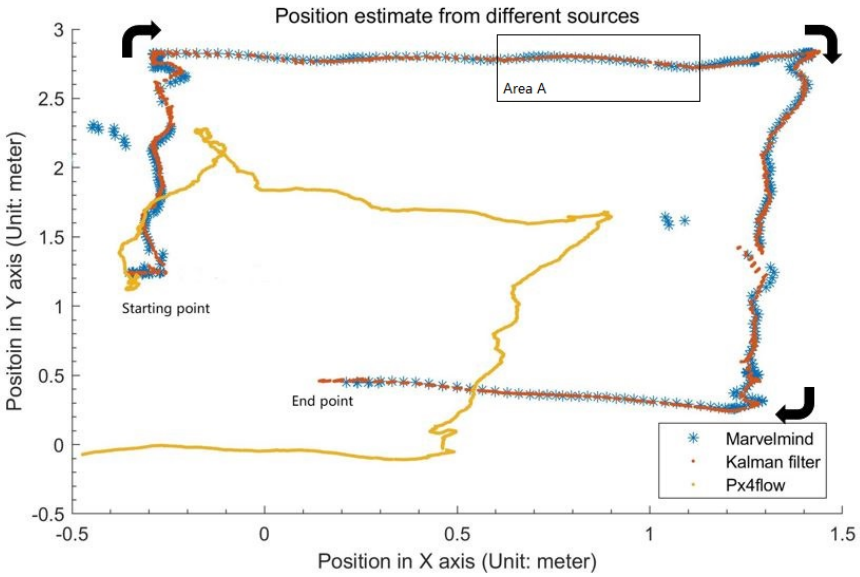


Figure 4.12: Manually position estimate test

If we take a close view of the "Area A" in figure 4.12, which is shown in figure 4.13, it can be found that the Kalman filter provides a much higher update frequency than the Marvelmind. This is because the Kalman filter will estimate the position from both the Px4flow and Marvelmind. The Px4flow provides a position estimation at the update frequency of the IMU, which is 100 Hz. Comparing to the Marvelmind, which update its estimation at 8 Hz, the Kalman filter has a 8 times update rate.

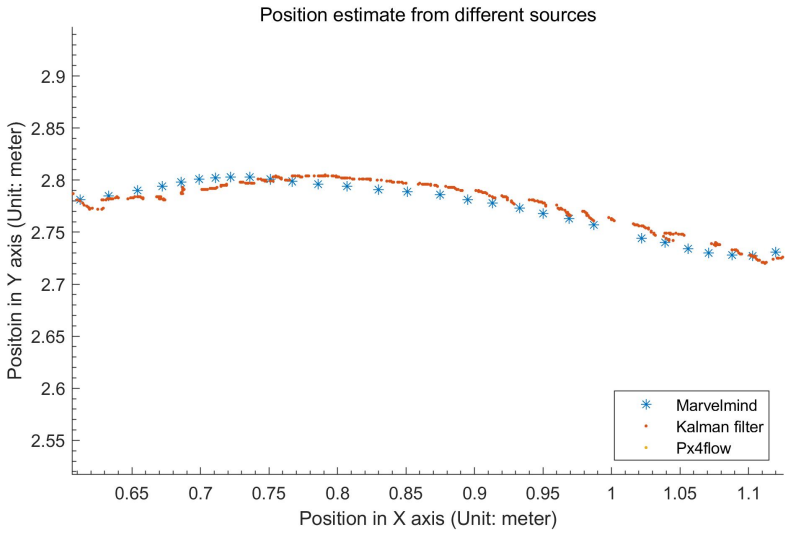
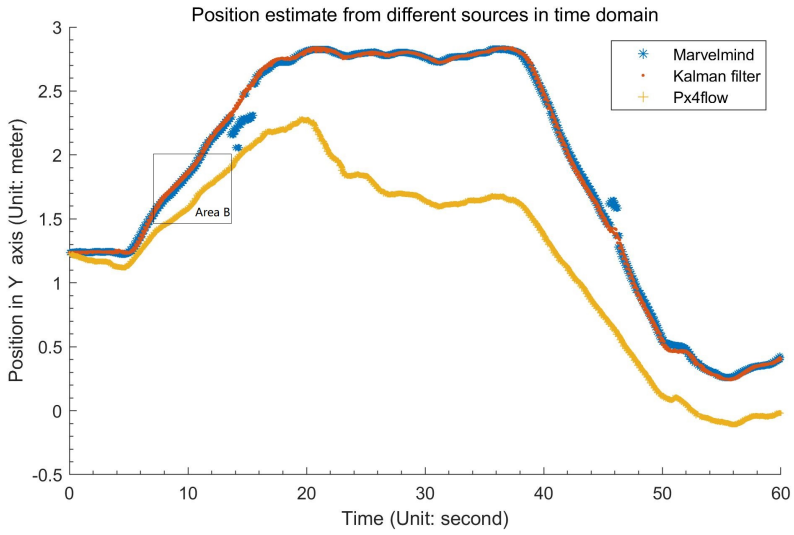
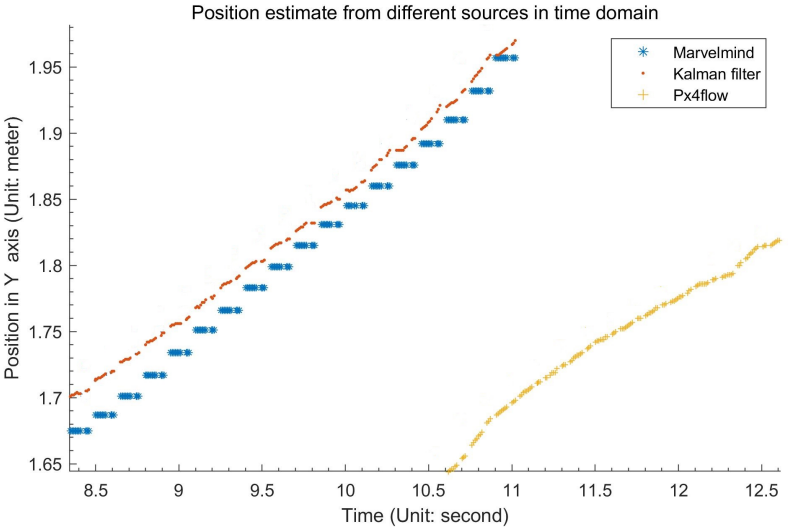


Figure 4.13: A close look of the manually test

The test result is also plotted in the time domain as shown in figure 4.14, together with a close look between 8-12 seconds. Again, the drift phenomenon for the Px4flow is significant. In the close look figure, the data from Marvelmind acts as a "stair". As we have discussed above, this is because the Marvelmind only as an 8 Hz update rate so it will remains the same during two samples. Thanks to the high frequency of the Px4flow, the Kalman filter can provide a more continuous position estimation.



(a)



(b) a close look of area B in (a)

Figure 4.14: Manually position estimate test in time domain

(a) (b)

Figure 4.15: An automatic car is used to test the system

In order to accurately test the system, the test bench is located at a automatic car, which is shown in figure 4.15 and the car will follows a path on the ground. However in this system set-up, the Px4flow will be too close to the ground. Since the Px4flow estimate the velocity by comparing the images, the texture on the ground will be too uniform to compare. One solution is to facing the camera of the Px4flow towards the ceiling and a net will be hung over the test ground as shown in figure 4.16. So the net will be used as the reference plane for the Px4flow.

Figure 4.16: A net is used as the reference plane for Px4flow

Another problem with this system set-up is, as shown in figure 4.17, the Marvelmind and Px4flow do not coincide vertically any more. Suppose the system rotates around the Marvelmind, the estimate from Px4flow will be a circle centred at Marvelmind with a radius equal to the distance between the two sensors, while the position estimated from Marvelmind will be a single point. The

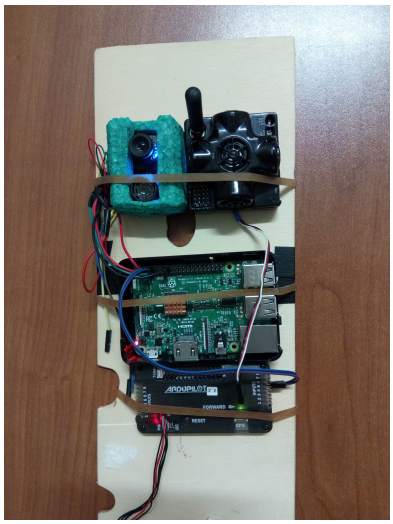


Figure 4.17: Two sensors are not coincide when Px4flow facing up

mechanism of this problem is very similar as the angular rate compensate error which we discussed in section 4.3.1. This kind of conflict between the estimation from two sensors is not considered when we design the Kalman filter. As a solution, in the following test with the Px4flow facing towards the ceiling, we simply not rotate the system and only test the performance while the autonomic car moves along a straight line. The speed of the car has been tuned to $0.1m/s$.

Figure 4.18 shows the test result. Notice that for a better view of the data, we removes the estimation between two Marvelmind samples, so in this figure the update rates from different sources are all 8 Hz. The green solid line in the figure indicates the true path of the test. The drift error of the Px4flow is still obvious even though the test length is only 30 seconds. On the other hand, the estimations from Kalman filter and the Marvelmind are similar. A numerical results of this test is shown in table 4.2. Notice that the variance of the Px4flow will keep growing as the time goes by. The value shows in the table is only the situation for 30 seconds. By comparing the performance, we can tell that the drift error of the Px4flow has been removed well by the Kalman filter, and the improvement with regards to the Marvelmind is limited: the improvement of the standard deviation comparing to the Marvelmind is about 15%.

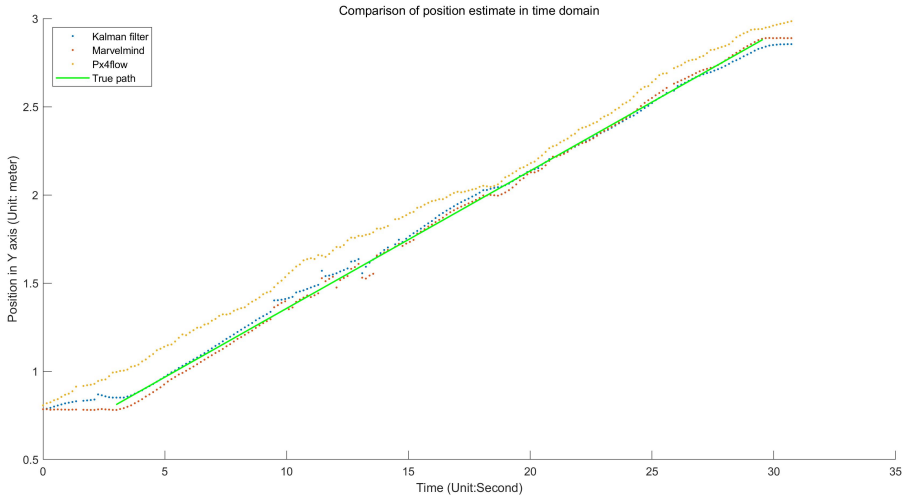


Figure 4.18: System test performs on the autonomic car

Data source	Px4flow	Marvelmind	Kalman filter
Standard deviation (Unit: meter)	0.0420	0.0319	0.0275
Maximum error (Unit: meter)	0.1406	0.0703	0.0815

Table 4.2: Position estimate performance from different sources

As a comparison with the simulation result in section 3.3.3, a stationary on-board test is also performed. This test is done by put the system at a fixed position without any movement or rotation for 600 seconds. The result is shown in figure 4.19.

Note that in figure 4.19 the estimation from Px4flow is not shown completely. Since the drift error of Px4flow is dramatic, its estimation of position goes far beyond the scale of the figure and shows the position up to 0.6 meter to keep the figure clear. The result of the on-board test is very similar as the simulation result we have seen in section 3.3.3. One difference is the output of Kalman filter is not as smooth as in the simulation. The reason is in practical, the noise of Px4flow is much higher than we assume in the simulation. As a result, the Kalman gain calculated by the algorithm will be more close to one, which means the Kalman filter "trusts" the Marvelmind more than the Px4flow. This also explains why the improvement of variance from the Kalman filter comparing to the Marvelmind is limited.

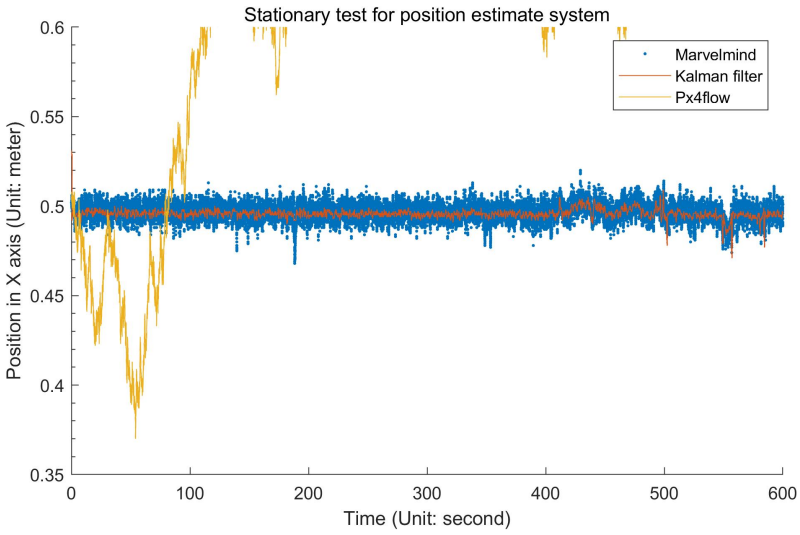


Figure 4.19: Stationary on-board test for the position estimate system

As a conclusion, the combined position estimate system we designed in this thesis is functional. It suppresses the drift error of the Px4flow. Comparing to the Marvelmind, it only provides a limited improvement by the means of variance, due to the high measurement noise from Px4flow. However, keep in mind that by employing the Px4flow, the update rate of the combined position estimate system is much higher than the Marvelmind.

Chapter 5

Conclusion and future works

5.1 Conclusion on combined position estimate system

A combined position estimate system has been designed and tested in this thesis. By combining the Px4flow optical flow sensor and the Marvelmind ultrasonic sensor, the system provides a synthesized position estimation. The system contains three main algorithm: the angular rate compensate for Px4flow, the attitude estimation from IMU and finally the position estimate algorithm. We will discuss their performance one by one in this section.

Conclusion on angular rate compensate algorithm
The angular rate compensate algorithm introduced in this thesis is used to counter the extra optical flow caused by the rotation of the UAV. According to the test in section 4.3.1, it provide a good suppression on the rotations: the optical flow caused by the rotation is reduced by about 17 dB. However this algorithm assumes the rotation is centred around the Px4flow. The error of this compensate algorithm will increase with the distance between Px4flow and the rotation centre.

Conclusion on attitude estimate algorithm
The attitude estimate algorithm is a Kalman filter which is used to fusion the data from IMU for providing an accurate and stable attitude measurement. The test shown in section 4.3.2 shows that this algorithm is highly efficient. Comparing to the attitude measured by the raw data from IMU sensors alone, this Kalman filter provides a more accurate estimation. If we consider the standard deviation as an accuracy factor, the improvement of the Kalman filter is about 55% comparing to the non-filtered data. In the meanwhile, this algorithm has

an update rate of 100 Hz, which we will shortly see plays an essential part in the position estimate algorithm.

Conclusion on position estimate algorithm
The position estimate algorithm employs the Kalman filter to fusion the position measurement from Px4flow and Marvelmind. The drift error in the estimation of Px4flow has been fully suppressed by the Kalman filter. Comparing to the Marvelmind, the Kalman filter only improves the standard deviation for about 15%, which is not a huge improvement comparing to the Kalman filter in the attitude estimation algorithm (55%). However, the Kalman filter also provides a much higher update frequency (100 Hz) while the Marvelmind measures the position at 8 Hz. The whole system can be regarded as a redundant measurement system: the Px4flow is used as the main estimator at an update rate of 100 Hz and the Marvelmind is used as an observer provides a correction at 8 Hz. So the whole system benefits from both sensors: it provides estimation as accurate as the Marvelmind and as fast as the Px4flow.

5.2 Future works

At the end of this report, some recommendations for future works are proposed here.

Firstly, the system built in this thesis assumes two sensors are installed as close as possible. Moreover they should be near to the rotation centre of the UAV. In practical this will not always be possible. An algorithm, which is capable of compensating the error caused by the distance between two sensors, as well as the distance between the sensor and the rotation centre, should be built in the future. This will extend the flexibility of the system.

Secondly, in the thesis the attitude of the system employs the compass as one of the data sources. For an indoor UAV, the working environment is usually complex. The metals, other electronic devices, electromagnetic field will introduce unexpected error for the compass. One solution for this is using the "paired" Marvelmind hedgehogs: two hedgehogs are mounted to the UAV simultaneously and the distance between them are known. By processing the position of the two hedgehogs, the attitude in the yaw axis can be measured, which can be used to replace the compass. However, recall that one main feature of our position estimate system is its high update rate thanks to the high frequency output of IMU, if we employs the paired Marvelmind hedgehogs to measure the attitude of the system, the overall update rate of the system will reduce to the same level of the Marvelmind. Some data fusion method could be employed here to merge the attitudes measured from the IMU and from the paired Marvelmind.

Thirdly, in the system introduced in this thesis, the Marvelmind is the only data source for the altitude measurement. So actually our system is a 2-dimensional position estimate system. The position in the Z axis is fully relies on the Marvelmind. In the future, some other sensors, such as a Lidar

sensor or even a 3D optical sensor can be used to measure the altitude of the UAV. And again, data fusion method could be employed to merge the altitude measurement from different sources.

Last but not the least, the position estimate system introduced in this thesis is a stand alone system. In an indoor UAV, there will be lots of other components such as the PID controller, aircraft flight control system, wireless communication unit, etc. The consistency of the data from different sources is essential in such a complex system. Thus, the data synchronization problem should be carefully considered before mounting the combined position estimate system.

Chapter 6

Acknowledgement

This is a master degree thesis and the work described in this report is carried out at the UAV laboratory from Dipartimento di Ingegneria Meccanica e Aerospaziale (DIMEAS) of Politecnico di Torino in the second semester of academic year 2017-2018.

A combined position estimate system is designed in this thesis, which will be used to measure the position of an indoor UAV. Two sensors, Px4flow and Marvelmind are used in this system. The extended Kalman filter (EKF) is employed as the main data fusion method. All the algorithms are implemented in the single board computer Raspberry Pi and several on-board tests are performed and discussed in the report.

I would like to bring my sincere tributes to my supervisor Rizzo Alessandro, Dabbene Fabrizio, and Guglieri Giorgio, who lead me to the research field of indoor UAV and provide me the opportunity to work in the UAV laboratory. I also want to deliver my special gratitude to my colleague Matteo Scanavino and Yuntian Li, who provide me valuable advices during my entire work.

The four months experience in the UAV laboratory is far more fast than I have ever imagine and it is so precious that I would use my whole life to forget.

Tianfang Sun
August, 2018

Appendix A

Hardware communication protocol

A.1 Communication protocol of Marvelmind

The Marvelmind supports USB, UART and SPI protocol for data transmission. In this thesis, the USB protocol is employed. The hedgehog will register itself as a "virtual serial" device under USB protocol. That is, when the USB is used to connect the hedgehog to an upper controller (a PC or a Raspberry Pi), we can read data from the USB port just as from a common serial port. The only difference is we do not need to take care the normal parameters of the serial port (baud, number of bits, parity, etc.), because the USB protocol will handle the transmitting.

Since we gather the data just as from a serial port, it also need to know the actual meanings of each byte we get. A "soft" protocol which designed by the developers of Marvelmind is used to describe the data structure. In this protocol, the data is transmitted in "packet". Each packet begins with a start byte $0xFF$, and ends with two CRC-16 check bytes. Table A.1 shows the general structure of all the packets in Marvelmind. Note the type of packet 0x47 indicates the packet is transmitting the sensor data from Marvelmind.

The payload field in table A.1 is determined by which kind of information is carried in that packet. In our case, the packet of the raw distance between the hedgehog to each beacons is used. The data structure of the payload field for the raw distance packet is shown in table A.2.

Offset	Size (bytes)	Type	Description	Value
0	1	uint8	Destination address	0xFF
1	1	uint8	Type of packet	0x47
2	2	uint16	Code of data in packet	See detail
4	1	uint8	Number of bytes of data transmitting	N
5	N	N bytes	Payload data according to code of data field	
5+N	2	uint16	CRC-16	

Table A.1: General structure of the packet in Marvelmind

Offset	Size (bytes)	Type	Description
0	1	uint8	Address of hedgehog
1	6		Distance item for beacon 1
7	6		Distance item for beacon 2
13	6		Distance item for beacon 3
19	6		Distance item for beacon 4
25	7	uint8	reserved

Table A.2: Payload field of the raw distance packet

And the distance item field has the structure as shown in table A.3.

Offset	Size (bytes)	Type	Description
0	1	uint8	Address of beacon (0 if item not filled)
1	4	uint32	Distance to the beacon, mm
5	1	uint8	Reserved (0)

Table A.3: Payload field of the raw distance packet

A.2 Communication protocol of Px4flow

Since the IMU is embedded inside the Marvelmind, it also follows the same communication protocol as the Marvelmind. The difference is the IMU using a different payload field from the Marvelmind. The according payload field of the packet of raw IMU sensors data can be found in table A.4. Note that for the compass, the precision in the Z axis is different with the X and Y axis.

Offset	Size (bytes)	Type	Description
0	2	int16	Accelerometer, X axis, 1 mg/LSB
2	2	int16	Accelerometer, Y axis, 1 mg/LSB
4	2	int16	Accelerometer, Z axis, 1 mg/LSB
6	2	int16	Gyroscope, X axis, 0.0175 dps/LSB
8	2	int16	Gyroscope, Y axis, 0.0175 dps/LSB
10	2	int16	Gyroscope, Z axis, 0.0175 dps/LSB
12	2	int16	Compass, X axis, 1100 LSB/Gauss
14	2	int16	Compass, Y axis, 1100 LSB/Gauss
16	2	int16	Compass, Z axis, 980 LSB/Gauss
18	6		Reserved
24	4	int32	Timestamp, ms
28	4	int8	reserved

Table A.4: Payload field of the raw distance packet

A.3 Communication protocol of Px4flow

Even though we will only use the I2C protocol of Px4flow in this thesis, the UART protocol will still be illustrated to keep integrity of the introduction. We will first demonstrate the UART protocol, then follows the I2C.

MAVLink protocol

MAVLink is an open-source protocol for communications between unmanned vehicle and the ground control station. It is very light-weighted and designed as a header-only message marshalling library. MAVLink arrange the data into packet with length of 8 to 263 bytes. The first 6 bytes and last 2 bytes of a packet is demanded and has reserved usages for each packet while all the rest

bytes will depend on what kind of message it is actually carrying. Figure A.1 shows an anatomy of the packet structure and table A.5 shows a detailed explanation for each byte in the packet. Notice the start sign is different for varies versions of MAVLink protocol and in Px4flow the 1.0 version is used.

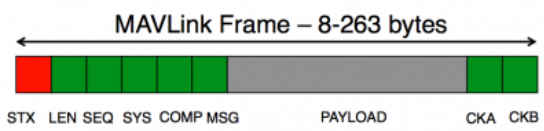


Figure A.1: Standard MAVLink packet

As we can see from table A.5, the payload in a packet is defined by the message ID. In Px4flow, the sensor sends optical flow data in OPTICAL_FLOW_RAD message (message ID: 106). The payload structure in the massage is shown in table A.6.

One important thing that is worth to mention is the order of the fields in these tables is not the actually data sequence in the MAVLink packet. The MAVLink will reorder these fields according to their data size. In the actual packet, the reordering follows the following rules:

- Fields are sorted according to their native data size, first (u)int64_t and double, then (u)int32_t, float, (u)int16_t, (u)int8_t
- If two fields have the same length, their order is preserved as it was present before the data field size ordering
- Arrays are handled based on the data type they use, not based on the total array size
- The CRC field is calculated after the reordering

I2C protocol

The default 7-bit I2C address of the PX4FLOW is 0x42. However it can be selected from 0x42 to 0x49 by soldering the jumpers on the sensor board. The order of the I2C frame is shown in table A.7. Some of the fields is ellipsis because they are following the same structure as MAVLink protocol in table A.6 so it will not be repeated here.

Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss.
3	System ID	1 - 255	ID of the sending system. Allows to differentiate different components on the same network.
4	Component ID	0 - 255	ID of the sending component. Allows to differentiate different components of the same system.
5	Message ID	0 - 255	ID of the message - the id defines what the payload means and how it should be correctly decoded.
6 to (n+6)	Data	0 - 255	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum	0 - 255	ITU X.25/SAE AS-4 hash, excluding packet start sign,

Table A.5: Data structure in a MAVLink packet

Field Name	Type	Description
time_usec	uint64	Timestamp (microseconds, synced to UNIX time or since system boot) (Units: us)
sensor_id	uint8	Sensor ID
integration_time_us	uint32	Integration time in microseconds. Divide integrated_x and integrated_y by the integration time to obtain average flow. The integration time also indicates the. (Units: us)
integrated_x	float	Flow in radians around X axis (Sensor right-hand rotation about the X axis induces a positive flow. Sensor linear motion along the positive Y axis induces a negative flow.) (Units: rad)
integrated_y	float	Flow in radians around Y axis (Sensor right-hand rotation about the Y axis induces a positive flow. Sensor linear motion along the positive X axis induces a positive flow.) (Units: rad)
integrated_xgyro	float	RH rotation around X axis (rad) (Units: rad)
integrated_ygyro	float	RH rotation around Y axis (rad) (Units: rad)
integrated_zgyro	float	RH rotation around Z axis (rad) (Units: rad)
temperature	int16	Temperature * 100 in centi-degrees Celsius (Units: cdegC)
quality	int8	Optical flow quality. 0: no valid flow, 255: maximum quality
time_delta_distance_us	int32	Time in microseconds since the distance was sampled. (Units: us)
distance	float	Distance to the centre of the flow field in meters. Positive value (including zero): distance known. Negative value: Unknown distance. (Units: m)

Table A.6: Payload structure in OPTICAL_FLOW_RAD message

Field Name	Description
0x00	Framecounter lower byte
0x01	Framecounter upper byte
0x02	latest Flow*10 in x direction lower byte
0x03	latest Flow*10 in x direction upper byte
...	...
0x13	Sonar Timestamp
0x14	Ground distance lower byte
0x15	Ground distance upper byte
0x16	Framecounter since last I2C readout lower byte
0x17	Framecounter since last I2C readout upper byte
0x18	Accumulated flow in radians*10000 around x axis since last I2C readout lower byte
0x19	Accumulated flow in radians*10000 around x axis since last I2C readout upper byte
...	...
0x22	Accumulation timespan in microseconds since last I2C readout byte 0
0x23	Accumulation timespan in microseconds since last I2C readout byte 1
0x24	Accumulation timespan in microseconds since last I2C readout byte 2
0x25	Accumulation timespan in microseconds since last I2C readout byte 3
...	...
0x2A	Ground distance in meters*1000 lower byte
0x2B	Ground distance in meters*1000 upper byte
0x2C	Temperature in Degree Celsius*100 lower byte
0x2D	Temperature in Degree Celsius*100 upper byte
0x2E	Averaged quality of accumulated flow values

Table A.7: I2C frame in Px4flow

Appendix B

Program code

B.1 Main program

The main program is listed here.

PositionEstimate.py

```
1  from threading import Thread
2  import smbus
3  import serial
4  from Pix4flowDriver import Pix4flowDriver
5  from MarvelmindDriver import MarvelmindDriver
6  from Kalman import kalman_filter
7  from Angular_compen import ag_cmp
8  from Attitude_calculation import attitude_calculation
9  from Marvelmindprotocol import Packagefinder
10 from math import pi
11
12
13 class KalmanPositionEstimate(Thread):
14
15     def __init__(self, write_file=False, print_result=False,
           result_output=False, i2cbus=1, serialbus='COM5'):
16         Thread.__init__(self)
17         self.m = [1.0,0.0,0.0]
18         self.wf = write_file
19         self.pt = print_result
20         self.serialbus = serialbus
21         self.i2cbus = i2cbus
22         self.ro = result_output
23         self.mavX = 0.0
24         self.mavY = 0.0
25         self.mavZ = 0.0
26         self.initial()
27
28
29     def initial(self):
30         self.x_last = [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
               1.0, 0.0]
```

```
31          self.p_last = [0.0] * 100
32          self.terminate = False
33          self.px4_handler = None
34          self.marvel_handler = None
35          self.position_out = []
36          self.marvel_x = []
37          self.marvel_y = []
38          self.marvel_z = []
39          self.timestamp = []
40          self.b_position = [0.0, 0.0, 1.850, 0.0, 3.562, 1.850,
                1.950, 2.132, 1.850]
41          self.px_posi = []
42          self.result_out = []
43
44          while self.px4_handler is None:
45              try:
46                  self.px4_handler = smbus.SMBus(self.i2cbus)
47                  self.px4 = Pix4flowDriver(self.px4_handler, 0x42
                        )
48              except Exception:
49                  pass
50          while self.marvel_handler is None:
51              try:
52                  self.marvel_handler = serial.Serial(self.
                        serialbus, timeout=3)
53                  self.marvel = MarvelmindDriver(self.
                        marvel_handler, )
54                  self.finder = Packagefinder(self.marvel_handler)
55              except Exception:
56                  pass
57          self.marvel_handler.reset_input_buffer()
58          self.marvel.refresh_cor()
59          self.marvel.refresh_dist()
60          self.px4.refresh_opticalflowrad()
61          self.position_in = [self.marvel.return_x() / 1000.0,
                self.marvel.return_y() / 1000.0]
62          self.marvel_handler.reset_input_buffer()
63          self.px4.refresh_opticalflowrad()
64          self.marvel.refresh_cor()
65          self.marvel.refresh_dist()
66          self.marvel_tstamp = self.marvel.return_timestamp() /
                1000.0
67          self.x_last[0] = self.position_in[0]
68          self.x_last[1] = self.position_in[1]
69          self.px_po = [self.x_last[0], self.x_last[1]]
70          self.mav_data = [self.marvel.return_dist_beacon_1() /
                1000.0, self.marvel.return_dist_beacon_2() / 1000.0,
71                           self.marvel.return_dist_beacon_3() /
                                1000.0, self.marvel.return_z() /
                                1000.0]
72          self.px4_delT = 0.0
73          self.px4_data = 0.0
74          self.px4_cmp = 0.0
75          self.mav_data_new = 0.0
76          self.mav_delT = 0.0
77          self.marvel.refresh_rawimu()
78          self.imu_tstamp = self.marvel.return_rimu_timestamp() /
```

```
            1000.0
79          self.q = [1.0, 0.0, 0.0, 0.0]
80          self.quat_out = self.q
81          self.imu_p = [0.0] * 16
82          self.gyro_biasx = -70.0
83          self.gyro_biasy = 12.0
84          self.gyro_biasz = 130.0
85          self.acc_biasx = 22.3461
86          self.acc_biasy = -7.2230
87          self.acc_biasz = 19.5191
88          self.compass_biasx = -100.7691
89          self.compass_biasy = -435.6860
90          self.compass_biasz = -111.2812
91          self.imu_x = [(self.marvel.return_accx() + self.
                acc_biasx),
92                        (self.marvel.return_accy() + self.
                            acc_biasy),
93                        (self.marvel.return_accz() + self.
                            acc_biasz),
94                        (self.marvel.return_compassx() + self.
                            compass_biasx),
95                        (self.marvel.return_compassy() + self.
                            compass_biasy),
96                        (self.marvel.return_compassz() + self.
                            compass_biasz) * (1100 / 980)]
97          self.imu_x_ini = self.imu_x
98
99      def run(self):
100         while self.terminate is False:
101             try:
102                 while self.px4_handler is None:
103                     try:
104                         pass
105                         self.px4_handler = smbus.SMBus(self.
                                i2cbus)
106                         # self.px4_handler = serial.Serial('COM4
                                ', 9600, timeout=3)
107                         self.px4 = Pix4flowDriver(self.
                                px4_handler, 0x42)
108                     except Exception:
109                         pass
110                 while self.marvel_handler is None:
111                     try:
112                         self.marvel_handler = serial.Serial(self
                                .serialbus, timeout=3)
113                         self.marvel = MarvelmindDriver(self.
                                marvel_handler, )
114                         self.finder = Packagefinder(self.
                                marvel_handler)
115                     except Exception:
116                         pass
117
118                 msgid = self.finder.find()
119                 if msgid == '\x11':
120                     self.PositionFilter()
121                     self.marvel_handler.reset_input_buffer()
122                 elif msgid == '\x03':
```

```
123                     self.AttitudeFilter()
124                     self.Px4Estimate()
125             except OSError:
126                 pass
127         self.marvel_handler.close()
128         self.marvel_handler = None
129         self.px4_handler = None
130 
131     def WriteFile(self, filename="kalman_output.txt"):
132         if self.wf is False:
133             return None
134         with open(filename, 'a') as f:
135             for i in range(0, min(len(self.position_out), len(
                    self.marvel_x), len(self.timestamp))):
136                 f.write("%+15.3f\t" % self.timestamp[i] + "
                        %+15.3f\t" % self.position_out[i][0] +
137                         "%+15.3f\t" % self.position_out[i][1] +
                            "%+15.3f\t" % self.marvel_x[i] +
138                         "%+15.3f\t" % self.marvel_y[i] + "%+15.3
                            f\t" % self.marvel_z[i] +
139                         "%+15.3f\t" % self.px_posi[i][0] + "
                            %+15.3f\t" % self.px_posi[i][1] + "\
                            n")
140         del self.timestamp[:]
141         del self.position_out[:]
142         del self.marvel_x[:]
143         del self.marvel_y[:]
144         del self.marvel_z[:]
145         del self.px_posi[:]
146 
147     def GetResult(self):
148         temp = self.result_out[:]
149         del self.result_out[:]
150         return temp
151 
152     def AttitudeFilter(self):
153         if self.marvel.readpackage_rawimu() == 'package_error':
154             return None
155         delT = self.marvel.return_rimu_timestamp() / 1000.0 -
                self.imu_tstamp
156         self.imu_tstamp = self.marvel.return_rimu_timestamp() /
                1000.0
157         gyro = [(self.marvel.return_gyrox() + self.gyro_biasx) *
                 (0.0175 * pi / 180),
158                 (self.marvel.return_gyroy() + self.gyro_biasy) *
                     (0.0175 * pi / 180),
159                 (self.marvel.return_gyroz() + self.gyro_biasz) *
                     (0.0175 * pi / 180)]
160         acc = [(self.marvel.return_accx() + self.acc_biasx),
161                (self.marvel.return_accy() + self.acc_biasy),
162                (self.marvel.return_accz() + self.acc_biasz)]
163         compass = [(self.marvel.return_compassx() + self.
                compass_biasx),
164                    (self.marvel.return_compassy() + self.
                        compass_biasy),
165                    (self.marvel.return_compassz() + self.
                        compass_biasz) * (1100 / 980)]
```

```
166         [self.quat_out, self.imu_x, self.q, self.imu_p] =
                attitude_calculation(gyro, acc, compass, self.
                quat_out, self.imu_x, self.q, self.imu_p, delT, self
                .imu_x_ini)
167         self.quat_out = list(self.quat_out)
168         self.imu_x = list(self.imu_x)
169         self.q = list(self.q)
170         self.imu_p = list(self.imu_p)
171         return None
172
173     def Px4Estimate(self):
174         self.px4.refresh_opticalflowrad()
175         self.px4_delT = self.px4.return_integration_time_us() /
                1000000.0
176         self.px4_data = [-self.px4.return_integrated_x() /
                10000.0, -self.px4.return_integrated_y() / 10000.0,
177                          -self.px4.return_integrated_xgyro() /
                                 10000.0, -self.px4.
                                 return_integrated_ygyro() /
                                 10000.0,
178                           self.px4.return_integrated_zgyro() /
                                 10000.0]
179         self.px4_cmp = ag_cmp([self.px4_data, self.px4_delT])
180         self.px4_data[0:2] = self.px4_cmp[0:2]
181         n = self.rotate(self.quat_out, self.m)
182         true_x = (self.px4_data[1] * self.mav_data[3]) * n[0] +
                (self.px4_data[0] * self.mav_data[3]) * n[1]
183         true_y = (self.px4_data[0] * self.mav_data[3]) * n[0] -
                (self.px4_data[1] * self.mav_data[3]) * n[1]
184         self.position_in = [self.position_in[0] + true_x,
185                             self.position_in[1] + true_y]
186         timenow = self.imu_tstamp
187         self.px_po = [self.px_po[0] + true_x,
188                       self.px_po[1] + true_y]
189         if self.wf is True:
190             self.position_out.append(self.position_in)
191             self.timestamp.append(timenow)
192             self.marvel_x.append(self.mavX)
193             self.marvel_y.append(self.mavY)
194             self.marvel_z.append(self.mavZ)
195             self.px_posi.append(self.px_po)
196
197     def PositionFilter(self):
198         if self.marvel.readpackage_cor() == 'package_error':
199             return None
200         if self.marvel.refresh_dist() == 'package_error':
201             return None
202         self.px4.refresh_opticalflowrad()
203         self.px4_delT = self.px4.return_integration_time_us() /
                1000000.0
204         self.px4_data = [-self.px4.return_integrated_x() /
                10000.0, -self.px4.return_integrated_y() / 10000.0,
205                          -self.px4.return_integrated_xgyro() /
                                 10000.0, -self.px4.
                                 return_integrated_ygyro() /
                                 10000.0,
206                           self.px4.return_integrated_zgyro() /
```

```
                                10000.0]
207             self.px4_cmp = ag_cmp([self.px4_data, self.px4_delT])
208             self.px4_data[0:2] = self.px4_cmp[0:2]
209             self.mav_data_new = [self.marvel.return_dist_beacon_1()
                    / 1000.0, self.marvel.return_dist_beacon_2() /
                    1000.0,
210                                  self.marvel.return_dist_beacon_3()
                                      / 1000.0, self.marvel.return_z
                                      () / 1000.0]
211             if abs(self.mav_data_new[0] - self.mav_data[0]) > 1 or
                    abs(self.mav_data_new[1] - self.mav_data[1]) > 1 or
                    \
212                     abs(self.mav_data_new[2] - self.mav_data[2]) > 1
                            or abs(self.mav_data_new[3] - self.mav_data
                            [3]) > 1:
213                 self.mav_data = self.mav_data
214             else:
215                 self.mav_data = self.mav_data_new
216             n = self.rotate(self.quat_out, self.m)
217             true_x = (self.px4_data[1] * self.mav_data[3]) * n[0] +
                    (self.px4_data[0] * self.mav_data[3]) * n[1]
218             true_y = (self.px4_data[0] * self.mav_data[3]) * n[0] -
                    (self.px4_data[1] * self.mav_data[3]) * n[1]
219             self.position_in = [self.position_in[0] + true_x,
220                                 self.position_in[1] + true_y]
221             self.mav_delT = self.marvel.return_timestamp() / 1000.0
                    - self.marvel_tstamp
222             self.marvel_tstamp = self.marvel.return_timestamp() /
                    1000.0
223             [self.position_in, self.x_last, self.p_last] =
                    kalman_filter(self.mav_data, self.px4_data, self.
                    position_in, self.x_last, self.p_last, self.mav_delT
                    , self.b_position)
224             self.position_in = list(self.position_in)
225             self.x_last = list(self.x_last)
226             self.p_last = list(self.p_last)
227             timenow = self.marvel.return_timestamp()
228             self.px_po = [self.px_po[0] + true_x,
229                           self.px_po[1] + true_y]
230             self.mavX = self.marvel.return_x() / 1000.0
231             self.mavY = self.marvel.return_y() / 1000.0
232             self.mavZ = self.marvel.return_z() / 1000.0
233             if self.wf is True:
234                 self.position_out.append(self.position_in)
235                 self.marvel_x.append(self.mavX)
236                 self.marvel_y.append(self.mavY)
237                 self.marvel_z.append(self.mavZ)
238                 self.timestamp.append(timenow / 1000.0)
239                 self.px_posi.append(self.px_po)
240             if self.pt is True:
241                 print("\n")
242                 print("%+15s\t" % "Time" + "%+15s\t" % "Position X"
                        + "%+15s\t" % "Position Y" + "\n")
243                 print("%+15.3f\t" % timenow + "%+15.3f\t" % self.
                        position_in[0] + "%+15.3f\t" % self.position_in
                        [1] + "\n")
244                 print("\n")
```

```
245             print("=" * 100)
246         if self.ro is True:
247             temp = self.position_in[:]
248             temp.insert(0, timenow)
249             self.result_out.append(temp)
250         return None
251
252     def stop(self):
253         self.terminate = True
254         print "Programme terminated"
255
256     def rotate(self,quat,r):
257         q1 = quat[0]
258         q2 = quat[1]
259         q3 = quat[2]
260         q4 = quat[3]
261         L = (q1 ** 2.0 + q2 ** 2.0 + q3 ** 2.0 + q4 ** 2.0) **
                0.5
262         q1 = q1 / L
263         q2 = q2 / L
264         q3 = q3 / L
265         q4 = q4 / L
266         C = [(q1 ** 2 + q2 ** 2 - q3 ** 2 - q4 ** 2), 2 * (q2 *
                q3 + q1 * q4), 2 * (q2 * q4 - q1 * q3),
267              2 * (q2 * q3 - q1 * q4), (q1 ** 2 - q2 ** 2 + q3 **
                    2 - q4 ** 2), 2 * (q3 * q4 + q1 * q2),
268              2 * (q2 * q4 + q1 * q3), 2 * (q3 * q4 - q1 * q2), (
                    q1 ** 2 - q2 ** 2 - q3 ** 2 + q4 ** 2)]
269         n1 = C[0] * r[0] + C[1] * r[1] + C[2] * r[2]
270         n2 = C[3] * r[0] + C[4] * r[1] + C[5] * r[2]
271         n3 = C[6] * r[0] + C[7] * r[1] + C[8] * r[2]
272         return [n1,n2,n3]
```

B.2 Description of the main program

The `PositionEstimate` package contains one class called `KalmanPositionEstimate` and four methods in it: `start`, `WriteFile`, `GetResult` and `stop`. A brief introduction for them is listed below:

`KalmanPositionEstimate` This is the only class in the package. It inherits from the `Thread` class from the `threading` package. It has five arguments:

- `write_file` A bool type argument used to choose either write the output of the Kalman filter to a file or not. If this argument is `True`, the user can call the `WriteFile` method to write the filtered result to a file, the file name can be designated in `WriteFile`. If this argument is `False` then the `WriteFile` will just return a `None`. For more details about method `WriteFile`, see the corresponding description below.
- `print_result` A bool type argument used to choose either print the output of the Kalman filter to the console screen or not. If this argument is `True`, the position estimate in X and Y axis will be printed on screen.
- `result_output` A bool type argument used to choose either return the output of the Kalman filter or not. If this argument is `True`, the user can call the `GetResult` method to return the filter output. For more details about method `GetResult`, see the corresponding description below.
- `i2cbus` A integer argument used to indicate the which I2C bus the Px4flow is connected with. In raspberry there are two I2C buses: I2C 0 and I2C 1. Set '0' or '1' in this argument corresponding to the I2C bus the Px4flow is in. If not assigned, the I2C 1 will be used as default.
- `serialbus` A string argument used to indicate which serial bus the Marvelmind is connected with. If not assigned, the `'/dev/ttyACM0'` will be used as default.

`start()` This method is inherit from the `Thread` class. It is used to start the Kalman filter process. For example, if we generate an object of class `KalmanPositionEstimate` called `test`, then call the method `test.start()` will start the filtering. The filtered result can be gathered using other methods as `WriteFile` or `GetResult`. It will also continuously print the result on the screen if the `print_result` is set to `True`.

`WriteFile(filename)` This method is used to write the result from Kalman filter to a file. The file name can be specific in the argument `filename`, which is a string argument. If not assigned, the `"kalman_output.txt"` will be used as default. During the processing of the Kalman filter, all the results will be temporarily stored in a cache if the argument `write_file` is set to `True`. When the `WriteFile(filename)` method is called, the data in the cache will be write to the file and then the cache will be cleaned

up. This means the user can periodically call this method and it will write the data output from Kalman filter between the two method calls to the specific file. In this way the cache will only store limited amount of data to avoid memory overflow.

`GetResult()` This method is used to combine the system with other applications. Similar to the `WriteFile` method, if the argument `result_output` is set to `True`, all the results from Kalman filter will be stored in a cache. When the `GetResult()` is called, it will return a $N \times 3$ list. The N is the amount of the data group. For each data group, it contains three float values. They are: the timestamp for the data (Unit: second), the position estimation in X axis (Unit: meter) and the position estimation in Y axis (Unit: meter).

`stop()` This method is used to stop the system. After is method is called, the I2C bus and serial bus will be closed, all the caches will be cleaned up and the process of the Kalman filter will be terminated.

B.3 Marvelmind communication protocol

The program used to read data using the Marvelmind protocol is listed here.

Marvelmindprotocol.py

```
import struct
from crcmod import predefined

INIT = 0
TOP_DTCT = 1
COD_DTCT = 2
REV_DTCT = 3
NOB_DTCT = 4
MSG_DTCT = 5
CRC_DTCT = 6
END = 7

class Packagefinder:
    def __init__(self, handler):
        self.handler = handler

    def find(self):
        if self.handler.is_open is False:
            self.handler.open()
        state = INIT
        msgid = 'NO_PACKAGE'
        while state != END:
            byte_in = self.handler.read()
            if state == INIT:
                if byte_in == '\xFF':
                    state = TOP_DTCT
            elif state == TOP_DTCT:
                if byte_in == '\x47':
                    state = COD_DTCT
                else:
                    state = INIT
            elif state == COD_DTCT:
                msgid = byte_in
                state = END
        return msgid


class Marvelmindprotocol:
    def __init__(self, handler=None, msgid='\x04'):
        self.handler = handler
        self.msgid = msgid
        self.msg = []
        self.msglen = 0
        self.crc16 = predefined.mkCrcFun('modbus')

    def readpackage(self):
        if self.handler.is_open is False:
            self.handler.open()
        state = REV_DTCT
        length = 0
```

```
51          self.msg = []
52          self.msglen = 0
53          while state != END:
54              byte_in = self.handler.read()
55              if state == REV_DTCT:
56                  state = NOB_DTCT
57              elif state == NOB_DTCT:
58                  length = byte_in
59                  self.msglen = struct.unpack('B', length)[0]
60                  state = MSG_DTCT
61              elif state == MSG_DTCT:
62                  if len(self.msg) < self.msglen:
63                      self.msg.append(byte_in)
64                  else:
65                      crc_lo = byte_in
66                      state = CRC_DTCT
67              elif state == CRC_DTCT:
68                  msg = b''.join(self.msg)
69                  if self.crc16('\xFF\x47' + self.msgid + '\x00' +
                         length + msg + crc_lo + byte_in) != 0:
70                      self.msg = []
71                      self.msglen = 0
72                      return 'package_error'
73                  else:
74                      state = END
75          return None
76
77      def refresh(self):
78          if self.handler.is_open is False:
79              self.handler.open()
80          state = INIT
81          length = 0
82          self.msg = []
83          self.msglen = 0
84          while state != END:
85              byte_in = self.handler.read()
86              if state == INIT:
87                  if byte_in == '\xFF':
88                      state = TOP_DTCT
89              elif state == TOP_DTCT:
90                  if byte_in == '\x47':
91                      state = COD_DTCT
92                  else:
93                      state = INIT
94              elif state == COD_DTCT:
95                  if byte_in == self.msgid:
96                      state = REV_DTCT
97                  else:
98                      state = INIT
99              elif state == REV_DTCT:
100                 state = NOB_DTCT
101             elif state == NOB_DTCT:
102                 length = byte_in
103                 self.msglen = struct.unpack('B', length)[0]
104                 state = MSG_DTCT
105             elif state == MSG_DTCT:
106                 if len(self.msg) < self.msglen:
```

```
107                     self.msg.append(byte_in)
108                 else:
109                     crc_lo = byte_in
110                     state = CRC_DTCT
111             elif state == CRC_DTCT:
112                 msg = b''.join(self.msg)
113                 # noinspection PyTypeChecker
114                 check = '\xFF\x47' + self.msgid + '\x00' +
                        length + msg + crc_lo + byte_in
115                 if self.crc16(check) != 0:
116                     self.msg = []
117                     self.msglen = 0
118                     return 'package_error'
119                 else:
120                     state = END
121         return None
122
123     def getvalue(self, fmt='', lo=0, hi=0):
124         msg = b''.join(self.msg)
125         return struct.unpack(fmt, msg[lo:hi])
```

B.4 Marvelmind data convert

This program is used to convert the binary stream reads from Marvelmind into the specific data type required by other programs.

Marvelmindprotocol.py

```
import struct
from crcmod import predefined

INIT = 0
TOP_DTCT = 1
COD_DTCT = 2
REV_DTCT = 3
NOB_DTCT = 4
MSG_DTCT = 5
CRC_DTCT = 6
END = 7

class Packagefinder:
    def __init__(self, handler):
        self.handler = handler

    def find(self):
        if self.handler.is_open is False:
            self.handler.open()
        state = INIT
        msgid = 'NO_PACKAGE'
        while state != END:
            byte_in = self.handler.read()
            if state == INIT:
                if byte_in == '\xFF':
                    state = TOP_DTCT
            elif state == TOP_DTCT:
                if byte_in == '\x47':
                    state = COD_DTCT
                else:
                    state = INIT
            elif state == COD_DTCT:
                msgid = byte_in
                state = END
        return msgid


class Marvelmindprotocol:
    def __init__(self, handler=None, msgid='\x04'):
        self.handler = handler
        self.msgid = msgid
        self.msg = []
        self.msglen = 0
        self.crc16 = predefined.mkCrcFun('modbus')

    def readpackage(self):
        if self.handler.is_open is False:
            self.handler.open()
        state = REV_DTCT
        length = 0
```

```
51          self.msg = []
52          self.msglen = 0
53          while state != END:
54              byte_in = self.handler.read()
55              if state == REV_DTCT:
56                  state = NOB_DTCT
57              elif state == NOB_DTCT:
58                  length = byte_in
59                  self.msglen = struct.unpack('B', length)[0]
60                  state = MSG_DTCT
61              elif state == MSG_DTCT:
62                  if len(self.msg) < self.msglen:
63                      self.msg.append(byte_in)
64                  else:
65                      crc_lo = byte_in
66                      state = CRC_DTCT
67              elif state == CRC_DTCT:
68                  msg = b''.join(self.msg)
69                  if self.crc16('\xFF\x47' + self.msgid + '\x00' +
                         length + msg + crc_lo + byte_in) != 0:
70                      self.msg = []
71                      self.msglen = 0
72                      return 'package_error'
73                  else:
74                      state = END
75          return None
76
77      def refresh(self):
78          if self.handler.is_open is False:
79              self.handler.open()
80          state = INIT
81          length = 0
82          self.msg = []
83          self.msglen = 0
84          while state != END:
85              byte_in = self.handler.read()
86              if state == INIT:
87                  if byte_in == '\xFF':
88                      state = TOP_DTCT
89              elif state == TOP_DTCT:
90                  if byte_in == '\x47':
91                      state = COD_DTCT
92                  else:
93                      state = INIT
94              elif state == COD_DTCT:
95                  if byte_in == self.msgid:
96                      state = REV_DTCT
97                  else:
98                      state = INIT
99              elif state == REV_DTCT:
100                 state = NOB_DTCT
101             elif state == NOB_DTCT:
102                 length = byte_in
103                 self.msglen = struct.unpack('B', length)[0]
104                 state = MSG_DTCT
105             elif state == MSG_DTCT:
106                 if len(self.msg) < self.msglen:
```

```
107                     self.msg.append(byte_in)
108                 else:
109                     crc_lo = byte_in
110                     state = CRC_DTCT
111             elif state == CRC_DTCT:
112                 msg = b''.join(self.msg)
113                 # noinspection PyTypeChecker
114                 check = '\xFF\x47' + self.msgid + '\x00' +
                        length + msg + crc_lo + byte_in
115                 if self.crc16(check) != 0:
116                     self.msg = []
117                     self.msglen = 0
118                     return 'package_error'
119                 else:
120                     state = END
121         return None
122
123     def getvalue(self, fmt='', lo=0, hi=0):
124         msg = b''.join(self.msg)
125         return struct.unpack(fmt, msg[lo:hi])
```

B.5 Px4flow data convert

This program is used to convert the binary stream reads from Px4flow into the specific data type required by other programs.

Pix4flowDriver.py

```
import i2cprotocol

class Pix4flowDriver:
    def __init__(self, handler, addr, flowradid=0x16):
        self.handler = handler
        self.optical_flow_rad_handler = i2cprotocol.i2cprotocol(
            self.handler, addr, flowradid)

    def refresh_opticalflowrad(self):
        self.optical_flow_rad_handler.refresh()

    def return_integrated_x(self):
        return self.optical_flow_rad_handler.getvalue('h',2,4)
            [0]

    def return_integrated_y(self):
        return self.optical_flow_rad_handler.getvalue('h',4,6)
            [0]

    def return_integrated_xgyro(self):
        return self.optical_flow_rad_handler.getvalue('h',6,8)
            [0]

    def return_integrated_ygyro(self):
        return self.optical_flow_rad_handler.getvalue('h',8,10)
            [0]

    def return_integrated_zgyro(self):
        return self.optical_flow_rad_handler.getvalue('h',10,12)
            [0]

    def return_integration_time_us(self):
        return self.optical_flow_rad_handler.getvalue('I',12,16)
            [0]

    def return_time_usec_rad(self):
        return self.optical_flow_rad_handler.getvalue('I',16,20)
            [0]

    def return_distance(self):
        return self.optical_flow_rad_handler.getvalue('h',20,22)
            [0]

    def return_temperature(self):
        return self.optical_flow_rad_handler.getvalue('h',22,24)
            [0]

    def return_quality(self):
        return self.optical_flow_rad_handler.getvalue('B',24,25)
            [0]
```

B.6 Angular rate compensate algorithm

This program is the Matlab function used to perform the angular rate compensate algorithm.

angular_compen.m

```
function comp = angular_compen(rad,timestamp)
    ag = rad(3:5)./timestamp;
    g = [0,0];
    y_last = [1,0,0,0,1,0,0,0,1];
    C = derfcn(ag,timestamp,y_last);
    p = (C*[0;0;1]-[0;0;1]).';
    g(1) = -p(2)./timestamp;
    g(2) = p(1)./timestamp;
    comp = rad(1:2) - g.*timestamp;
end

function y_solve  = derfcn(ag,timestamp,y_last)
y0 = y_last;
agx = ag(1);
agy = ag(2);
agz = ag(3);
tspan = [0,timestamp];

[t,y] = ode45(@(t,y) odefcn(t,y,agx,agy,agz),tspan,y0.');
y_solve = [y(end,1),y(end,2),y(end,3);y(end,4),y(end,5),y(end,6)
    ;y(end,7),y(end,8),y(end,9)];
end
```

B.7 Kalman filter for attitude estimate

This program is the Matlab function used to perform the attitude estimate algorithm.

attitude_calculation.m

```
function [quat_out,x_next,q_next,p_next] = attitude_calculation(
    gyro,acc,compass,quat_in,x_last,q_last,p_last,delT,x_ini)

    %noise matrix
    W = [4.5270e-05,0,0,0;0,4.5270e-05,0,0;0,0,4.5270e
        -05,0;0,0,0,4.5270e-05];
    R = [3.3004e-05,0,0,0,0,0;
        0,3.3004e-05,0,0,0,0;
        0,0,3.3004e-05,0,0,0;
        0,0,0,1.0049e-04,0,0;
        0,0,0,0,1.0049e-04,0;
        0,0,0,0,0,1.0049e-04];
    Z = [acc,compass].'-x_last;
    %System matrix calculate
    F = [0,-gyro(1),-gyro(2),-gyro(3);gyro(1),0,gyro(3),-gyro
        (2);gyro(2),-gyro(3),0,gyro(1);gyro(3),gyro(2),-gyro(1)
        ,0].*0.5;
    G = eye(4);
    A = [-F,G*W*(G.');zeros(4,4),(F.')].*delT;
    B = expm(A);
    phi = B(5:8,5:8).';
    Q = phi * B(1:4,5:8);
    quat = quat_in;
    L = sqrt(quat(1)^2+quat(2)^2+quat(3)^2+quat(4)^2);
    quat = quat./L;
    %Measurement matrix calculate
    h11 = 2*quat(1)*x_ini(1)+2*quat(4)*x_ini(2)-2*quat(3)*x_ini
        (3);
    h12 = 2*quat(2)*x_ini(1)+2*quat(3)*x_ini(2)+2*quat(4)*x_ini
        (3);
    h13 = -2*quat(3)*x_ini(1)+2*quat(2)*x_ini(2)-2*quat(1)*
        x_ini(3);
    h14 = -2*quat(4)*x_ini(1)+2*quat(1)*x_ini(2)+2*quat(2)*
        x_ini(3);

    h21 = -2*quat(4)*x_ini(1)+2*quat(1)*x_ini(2)+2*quat(2)*
        x_ini(3);
    h22 = 2*quat(3)*x_ini(1)-2*quat(2)*x_ini(2)+2*quat(1)*x_ini
        (3);
    h23 = 2*quat(2)*x_ini(1)+2*quat(3)*x_ini(2)+2*quat(4)*x_ini
        (3);
    h24 = -2*quat(1)*x_ini(1)-2*quat(4)*x_ini(2)+2*quat(3)*
        x_ini(3);

    h31 = 2*quat(3)*x_ini(1)-2*quat(2)*x_ini(2)+2*quat(1)*x_ini
        (3);
    h32 = 2*quat(4)*x_ini(1)-2*quat(1)*x_ini(2)-2*quat(2)*x_ini
        (3);
    h33 = 2*quat(1)*x_ini(1)+2*quat(4)*x_ini(2)-2*quat(3)*x_ini
        (3);
```

```
36        h34 = 2*quat(2)*x_ini(1)+2*quat(3)*x_ini(2)+2*quat(4)*x_ini
              (3);
37
38        h41 = 2*quat(1)*x_ini(4)+2*quat(4)*x_ini(5)-2*quat(3)*x_ini
              (6);
39        h42 = 2*quat(2)*x_ini(4)+2*quat(3)*x_ini(5)+2*quat(4)*x_ini
              (6);
40        h43 = -2*quat(3)*x_ini(4)+2*quat(2)*x_ini(5)-2*quat(1)*
              x_ini(6);
41        h44 = -2*quat(4)*x_ini(4)+2*quat(1)*x_ini(5)+2*quat(2)*
              x_ini(6);
42
43        h51 = -2*quat(4)*x_ini(4)+2*quat(1)*x_ini(5)+2*quat(2)*
              x_ini(6);
44        h52 = 2*quat(3)*x_ini(4)-2*quat(2)*x_ini(5)+2*quat(1)*x_ini
              (6);
45        h53 = 2*quat(2)*x_ini(4)+2*quat(3)*x_ini(5)+2*quat(4)*x_ini
              (6);
46        h54 = -2*quat(1)*x_ini(4)-2*quat(4)*x_ini(5)+2*quat(3)*
              x_ini(6);
47
48        h61 = 2*quat(3)*x_ini(4)-2*quat(2)*x_ini(5)+2*quat(1)*x_ini
              (6);
49        h62 = 2*quat(4)*x_ini(4)-2*quat(1)*x_ini(5)-2*quat(2)*x_ini
              (6);
50        h63 = 2*quat(1)*x_ini(4)+2*quat(4)*x_ini(5)-2*quat(3)*x_ini
              (6);
51        h64 = 2*quat(2)*x_ini(4)+2*quat(3)*x_ini(5)+2*quat(4)*x_ini
              (6);
52
53        H = [h11,h12,h13,h14;
54             h21,h22,h23,h24;
55             h31,h32,h33,h34;
56             h41,h42,h43,h44;
57             h51,h52,h53,h54;
58             h61,h62,h63,h64];
59
60        %Kalman gain calculate
61        K = p_last*(H.')/(H*p_last*(H.') + R);
62        %Update estimate
63        delq = q_last - quat;
64        q_new = q_last + K*(Z-H*delq);
65        %Update error
66        p_new = (eye(4)-K*H)*p_last;
67        %Predict state
68        q_next = phi*q_new;
69        %predict error
70        p_next = phi*p_new*(phi.')+Q;
71
72        quat_out = q_new.';
73        x_next = [quatro(q_new.',x_ini(1:3).'),quatro(q_new.',x_ini
              (4:6).')].';
74
75  end
```

B.8 Kalman filter for position estimate

This program is the Matlab function used to perform the position estimate algorithm.

kalman_filter.m

```
function [position_out,x_next,p_next] = kalman_filter(mav,px4,
    delT,position_in,x_last,p_last,b_position)
    X_last = x_last;
        P_last = p_last;
    %beacon position
    b1 = b_position(:,1);
    b2 = b_position(:,2);
    b3 = b_position(:,3);
    %sample time
    t = delT;
    %height of the beacon
    %l = b1(3);
    %flow velocity from px4
    vx = px4(2);
    vy = px4(1);
    %angular rate from px4
    omegax = px4(3);
    omegay = px4(4);
    omegaz = px4(5);
    position_x = position_in(1);
    position_y = position_in(2);
    %height update
    position_z = mav(4);
    %noise matrix
    W = [7.4756e-05,zeros(1,9);0,7.7235E-05,zeros(1,8);zeros
        (1,10);zeros(1,10);zeros(1,4),1.3965e-07,zeros(1,5);
        zeros(1,5),8.7105e-08,zeros(1,4);
        zeros(1,6),1.3279e-07,zeros(1,3);zeros(1,7),1.3965e-07,
            zeros(1,2);zeros(1,8),8.7105e-08,0;zeros(1,9)
            ,1.3279e-07];
    %R = [2.9524e-05,0,0;0,2.6953e-05,0;0,0,1.1116e-05];
    R = [3.5458e-04,0,0;0,3.0458e-04,0;0,0,4.1225e-04];
    hx = [sqrt((position_x-b1(1))^2+(position_y-b1(2))^2+(
        position_z-b1(3))^2);
        sqrt((position_x-b2(1))^2+(position_y-b2(2))^2+(
            position_z-b2(3))^2);
        sqrt((position_x-b3(1))^2+(position_y-b3(2))^2+(
            position_z-b3(3))^2)];
    Z = [mav(1)-hx(1);mav(2)-hx(2);mav(3)-hx(3)];
    %System matrix calculate
    omegaM = [0,-omegaz,omegay;omegaz,0,-omegax;-omegay,omegax
        ,0];
    F = [0,0,0,position_z,0,0,-position_z,0,0,0;0,0,position_z
        ,0,0,0,0,0,0,-position_z;zeros(1,10);zeros(1,10);zeros
        (3,4),-omegaM,zeros(3,3);zeros(3,7),-omegaM];
    G = eye(10);
    A = [-F,G*W*(G.');zeros(10,10),(F.')].*t;
    B = expm(A);
    phi = B(11:20,11:20).';
    Q = phi * B(1:10,11:20);
```

```
40      %Measurement matrix calculate
41      h1 = (position_x - b1(1))/sqrt((position_x-b1(1))^2+(
            position_y-b1(2))^2+(position_z-b1(3))^2);
42      h2 = (position_y - b1(2))/sqrt((position_x-b1(1))^2+(
            position_y-b1(2))^2+(position_z-b1(3))^2);
43      h3 = (position_x - b2(1))/sqrt((position_x-b2(1))^2+(
            position_y-b2(2))^2+(position_z-b2(3))^2);
44      h4 = (position_y - b2(2))/sqrt((position_x-b2(1))^2+(
            position_y-b2(2))^2+(position_z-b2(3))^2);
45      h5 = (position_x - b3(1))/sqrt((position_x-b3(1))^2+(
            position_y-b3(2))^2+(position_z-b3(3))^2);
46      h6 = (position_y - b3(2))/sqrt((position_x-b3(1))^2+(
            position_y-b3(2))^2+(position_z-b3(3))^2);
47      H = [h1,h2,zeros(1,8);h3,h4,zeros(1,8);h5,h6,zeros(1,8)];
48      %Kalman gain calculate
49      K = P_last*(H.')/(H*P_last*(H.') + R);
50      %Update estimate
51      delX = X_last - [position_x;position_y;zeros(8,1)];
52      x_new_temp = X_last + K*(Z-H*delX);
53      %Update error
54      P_new = (eye(10)-K*H)*P_last;
55      %Predict state
56      x_next_temp = phi*x_new_temp;
57      %x_next_temp(1) = x_next_temp(1) + vy * position_z;
58      %x_next_temp(2) = x_next_temp(2) + vx * position_z;
59      x_next_temp(3) = vx;
60      x_next_temp(4) = vy;
61      %predict error
62      p_next_temp = phi*P_new*(phi.')+Q;
63      position_out = [x_new_temp(1);x_new_temp(2)];
64      p_next = p_next_temp;
65      x_next = x_next_temp;
66  end
```

Bibliography

[1] John E Bortz. A new mathematical formulation for strapdown inertial navigation. *IEEE transactions on aerospace and electronic systems*, (1):61–66, 1971.

[2] Robert Grover Brown, Patrick YC Hwang, et al. *Introduction to random signals and applied Kalman filtering*, volume 3. Wiley New York, 1992.

[3] Federico Castanedo. A review of data fusion techniques. *The Scientific World Journal*, 2013, 2013.

[4] P. Daponte, L. De Vito, G. Mazzilli, F. Picariello, S. Rapuano, and M. Riccio. Metrology for drone and drone for metrology: Measurement systems on small civilian drones. In *2015 IEEE Metrology for Aerospace (MetroAeroSpace)*, pages 306–311, June 2015.

[5] Hugh F Durrant-Whyte. Sensor models and multisensor integration. *The international journal of robotics research*, 7(6):97–113, 1988.

[6] S. Gupte, Paul Infant Teenu Mohandas, and J. M. Conrad. A survey of quadrotor unmanned aerial vehicles. In *2012 Proceedings of IEEE Southeastcon*, pages 1–6, March 2012.

[7] D. L. Hall and J. Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6–23, Jan 1997.

[8] David Hoag. Apollo guidance and navigation: Considerations of apollo imu gimbal lock. *Canbridge: MIT Instrumentation Laboratory*, pages 1–64, 1963.

[9] D. Honegger, L. Meier, P. Tanskanen, and M. Pollefeys. An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications. In *2013 IEEE International Conference on Robotics and Automation*, pages 1736–1741, May 2013.

[10] Farid Kendoul, Isabelle Fantoni, and Kenzo Nonami. Optic flow-based vision system for autonomous 3d localization and control of small aerial vehicles. *Robotics and Autonomous Systems*, 57(6):591 – 602, 2009.

[11] Steffen L Lauritzen, Steffen L Lauritzen, Thorvald Nicolai Thiele, Johannes Thiele, and Anders Hald. *Thiele: pioneer in statistics*. Clarendon Press, 2002.

[12] Yuntian Li, Matteo Scanavino, Elisa Capello, Fabrizio Dabbene, and Giorgio Guglieri. A novel distributed architecture for uav indoor navigation. In *International Conference on Air Transport – INAIR 2018*, 2018.

[13] L. Mainetti, L. Patrono, and I. Sergi. A survey on indoor positioning systems. In *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 111–120, Sept 2014.

[14] Mautz and Rainer. Overview of current indoor positioning systems. *Geodezija ir kartografija*, 35.1:18–22, 2009.

[15] Howard Musoff and Paul Zarchan. *Fundamentals of Kalman filtering: a practical approach*. American Institute of Aeronautics and Astronautics, 2009.

[16] D. Shatat and T. A. Tutunji. Uav quadrotor implementation: A case study. In *2014 IEEE 11th International Multi-Conference on Systems, Signals Devices (SSD14)*, pages 1–6, Feb 2014.

[17] A. J. P. Taylor. *Jane's Book of Remotely Piloted Vehicles*.

[18] Zachary Treisman. A young person's guide to the hopf fibration. *arXiv preprint arXiv:0908.1205*, 2009.

[19] Charles Van Loan. Computing integrals involving the matrix exponential. *IEEE transactions on automatic control*, 23(3):395–404, 1978.