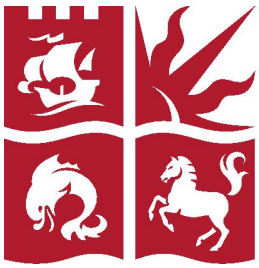

University of BRISTOL

Sensor Fusion

Characterisation and the Kalman Filter

Robert Bragg

May 2017

Final year project thesis submitted in support of the degree of

Bachelor of Engineering in Computer Science & Electronics

Department of Electrical & Electronic Engineering

University of Bristol

Abstract

Sensor fusion is the act of combining data from multiple sources to provide a more accurate estimate of the true value. At its most basic this could comprise of a simple average between identical sensors to try and minimise the effect of noise. This project looks at the more advanced Kalman filter which uses extrapolation and dynamic weighted averaging to produce optimal estimates. This filter was applied to the case of an autonomous rotary-wing aerial vehicle or drone to aid in object tracking and following within uncontrolled, volatile environments. Sensory apparatus was chosen to be cheap, power efficient and light due to budget restrictions and the constraints imposed by using a drone. Code was originally written and tested in MATLAB using simulated input data. Simulated test results showed a marked improvement in estimation accuracy for all state aspects when compared to naive sensor averaging, especially for those reliant on relatively high noise sensors. The code was then ported to golang so as to facilitate later integration with the drone software and to enable concurrent calculation work. Further testing needs to be done once integrated to ensure the filter is sufficient for real world use.

Dedication and acknowledgements

I would like to thank my supervisor Dr Naim Dahnoun for his advice and support throughout the duration of this thesis project.

I would also like to thank my peers who worked along side me in the MVB 5.01 computer science and electronics lab.

And lastly I would like to thank the MVB school office staff who ensure that everything runs smoothly.

AUTHOR'S DECLARATION

Unless otherwise acknowledged, the content of this thesis is the original work of the author. None of the work in this thesis has been submitted by the author in support of an application for another degree or qualification at this or any other university or institute of learning.

The views in this document are those of the author and do not in any way represent those of the University.

The author confirms that the printed copy and electronic version of this thesis are identical.

SIGNED: ..... DATE: .....

TABLE OF CONTENTS

List of Figures

Figure **Page**

List of Tables

CHAPTER 1

INTRODUCTION

Key to autonomy is the concept of self, and awareness of how ones self fits into the surrounding environment. For an autonomous robot this is largely encompassed by position and orientation as well as movement relative to the surroundings, collectively referred to as the state of the robot. This information is vital for enabling the robot to operate unhindered and without incident. Drones are a versatile platform as their access to a third dimension while maintaining the ability to hold stationary (unlike a fixed wing aircraft) means that they can be deployed in many environments with a wide variety of uses. This does come at the cost of needing to have even greater accuracy with regards to pathing and collision avoidance. Use cases for drones include aerial mapping and photography, surveillance, package delivery and scientific research in hostile environments.

The highly accurate sensory apparatus often required for this kind of navigation can be very expensive, hence a smarter method of sensory integration allowing use of cheaper apparatus could cut back on a lot of the cost. This project will compare a variety of cheap sensors to develop an appropriate suite capable of precise data production. Following that a form of sensor fusion will be applied to the sensory suite as well as any other gleanable information to provide accurate state estimates.

The drone itself is based on a HobbyKing^{TM} S550 hexcopter frame [1] with parts:

- Turnigy 4000mAh 4S 40C Lipo Pack
- AfroFlight Naze32 Rev6 Controller
- 2-6s 30 Amp BL Heli ESCs
- EMAX MT2216-810KV brushless motors

- 10x4.5 CW/CCW propellers

and was assembled as part of the project.

Previous work by Santana *et al.* looks to accomplish similar results to this project albeit at a smaller scale with much reduced sensory apparatus [2]. Another paper by Yang *et al.* shows use of the extended Kalman filter (a non-linear Kalman filter variant) to successfully land a multi-sensor rotary-wing unmanned aerial vehicle on a moving ship deck, detailing the use of low cost sensors with limited computational power [3].

The following paper presents the use of the Kalman filter on a sensory array that is larger than most examples seen in relevant papers but which is also cost efficient. It strives to achieve high quality results from lower quality components to facilitate the goals outlined above. Discussed also is the choice of sensory apparatus, the workings of the Kalman filter, the non-linear filter variants including why they're unsuitable, and testing procedure.

Paper copies of this thesis include an attached CD containing the program code described in the appendices.

CHAPTER 2

HOLISTIC PROJECT MAP

The project is a collaboration of separate parts to form a fully functional whole. While this paper focuses on the sensory apparatus and data interpretation, it is worth noting how it links to the other work taking place providing further functionality to the drone. Figure 2.1 describes the relationship between these component pieces.

The filter used in this project takes in data from many of these other sections as well as the sensory apparatus described in this paper. Distance to objects is provided by the disparity mapping, and positional/velocity information is relayed from the GPS. Fast and efficient

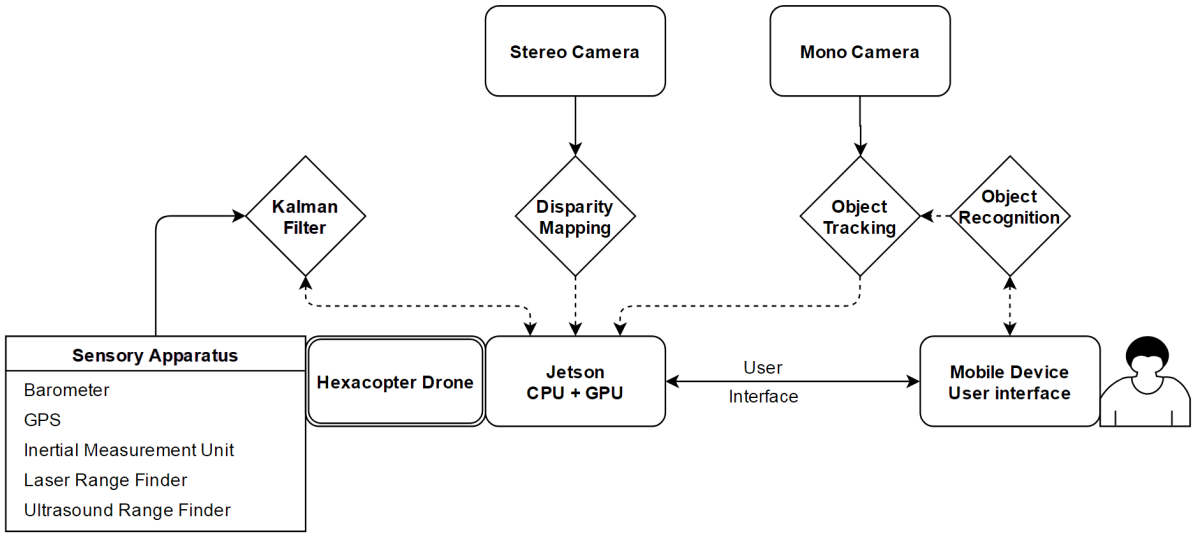

FIGURE 2.1. Project Diagram

communication between these components is vital and requires coordination between projects.

CHAPTER 3

SENSORY APPARATUS

Some data will be obtained from sources external to this project such as the GPS and the stereo camera disparity map. Additional sensors must be chosen to augment these incoming data sources and to cover as many blind spots as possible in the drones perception of the environment. The data required is largely based around range finding and drone movement to aid with collision avoidance.

3.1 Sensor Basics

Sensors come in two basic forms, active and passive. Active sensors require a signal to be generated by the apparatus that is later detected by a receiver, a good example of this would be a RADAR system which emits radio waves then detects the reflections. Passive sensors receive signals solely generated by the environment, such as heat detected by a thermal camera. The two forms have contrasting pros and cons. Active sensors are generally more accurate and less sensitive to noise, however this comes at the cost of significantly higher energy consumption. Passive sensors are correspondingly very low power, however they rely on the environment to provide usable signal at all times. They do allow "silent running", meaning that their use cannot be detected by a third party.

While it would be ideal for the drone to make full use of low power passive sensors, the range finding data required is difficult to obtain through passive methods, relying for instance on expensive stereo cameras. Though the drone will have one stereo camera aboard, it is not financially viable to have multiple units facing in all directions. For this reason the stereo camera will be mounted forwards facing, and much cheaper sensors will be utilised elsewhere.

3.2 Sensor Experimentation

There are a few sensors that can obtain the ranging data required, each with strengths and weaknesses.

- Ultrasonic time-of-flight, active: These sensors have a conical detection pattern that allows detection of a fairly wide area but at the cost of spatial resolution, figure 3.1 shows the basic function. Range and depth accuracy is variable with the quality of the sensor, but is typically around 5 cm to 4 m with 1 cm resolution. Polling rate for this sensor is limited by the speed of sound; for a sensor rated to 2 m the theoretical minimum time between readings is given by the travel time to and back from maximum distance, in this case $(2\,m \times 2) \div 343\,ms^{-1} = 12\,milliseconds$ and hence polling rate of $1/12\,milliseconds = 86\,Hz$. There are also certain conditions that can affect the operation of the sensor. Temperature will affect air density and hence the speed of sound meaning that calibration must be performed to ensure accurate readings. The reflective surface must be perpendicular to the sensor for the sound waves to bounce back to the receiver, deviation from this angle will drastically reduce signal return. The degree to which the surface absorbs sound can cause objects at the range extremities to not be detected, and makes it infeasible to use received power as a distance estimator outside of a calibrated, controlled environment, leaving time of flight as the only realistic range finding method. Advanced receivers can however determine whether a detected object is moving away from or towards the receiver based on Doppler shift [4].

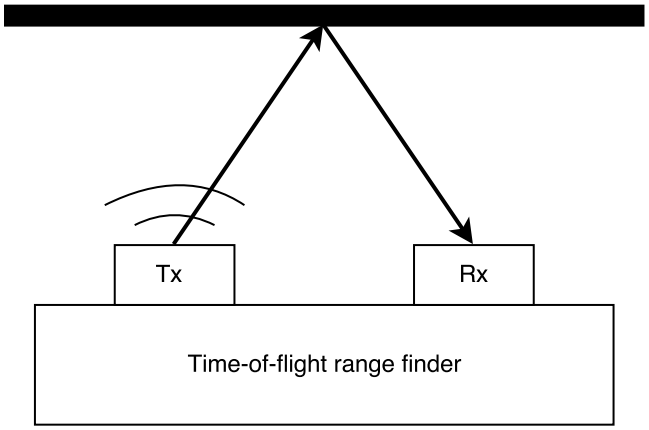

FIGURE 3.1. Diagram showing the basic mechanism behind a time-of-flight range finder

- Laser time-of-flight, active: While being fairly similar to the ultrasonic time-of-flight sensors, the laser has a much tighter conical detection pattern, a maximum of about 5^o, and a much faster signal travel speed resulting in higher possible polling rates. However it generally has a lower detection range due to the higher passive environmental noise and the tighter

beam has a higher tolerance on reflection angle. Their power level is also fairly limited due to needing to make them eye-safe.

- LiDAR, active: An extension of laser time-of-flight, LiDAR (Light Detection and Ranging) usually utilises a single laser to produce a two or three dimensional point cloud by rapidly sweeping the laser and taking measurements at various angles [5][6]. This apparatus is often used for topographical mapping from aircraft [7]. The technique greatly increases the complexity of the receiving equipment however as rotation of the measure point must be accounted for and inter-point interference should be minimised. A similar but much less applicable technology SoDAR uses the same principles but with ultrasonics, mostly utilised to measure atmospheric turbulence.

- Pressure altimeter, passive: Pressure altimeters are cheap but reliable sensors with very low weight and power consumption. They are affected by varying atmospheric pressure caused by moving weather systems however are fairly immune to value drifting otherwise.

LiDAR would clearly be the most ideal for this project due to its ability to rapidly scan a full 360^{o} using the arguably superior laser time-of-flight sensor. However, even a relatively cheap two dimensional LiDAR unit weighs 340g, a not insignificant amount of weight for a drone to carry, and more crucially costs over £400 which is not fiscally viable for this project [8]. The best alternative seems to be using a mixture of ultrasonic and laser time-of-flight sensors with one or two pressure altimeters for height. The ultrasonic sensors are very cheaply available [9] with the laser sensors a bit more expensive but still feasible. These readings can then be mixed to provide a better value estimate than would be achieved from each independently.

3.3 Implemented Sensory Apparatus

The final sensory array for the drone was not fully assembled however the general layout was proposed and can be seen in figure 3.2.

3.3.1 Proposed Layout

- One ultrasonic time-of-flight range finder facing up, down, and at each cardinal direction.
- One laser time-of-flight range finder facing up, down, and at each cardinal direction.
- One disparity mapped stereo camera facing forwards.
- One barometer.
- One global positioning system unit.
- Two inertial measurement units.

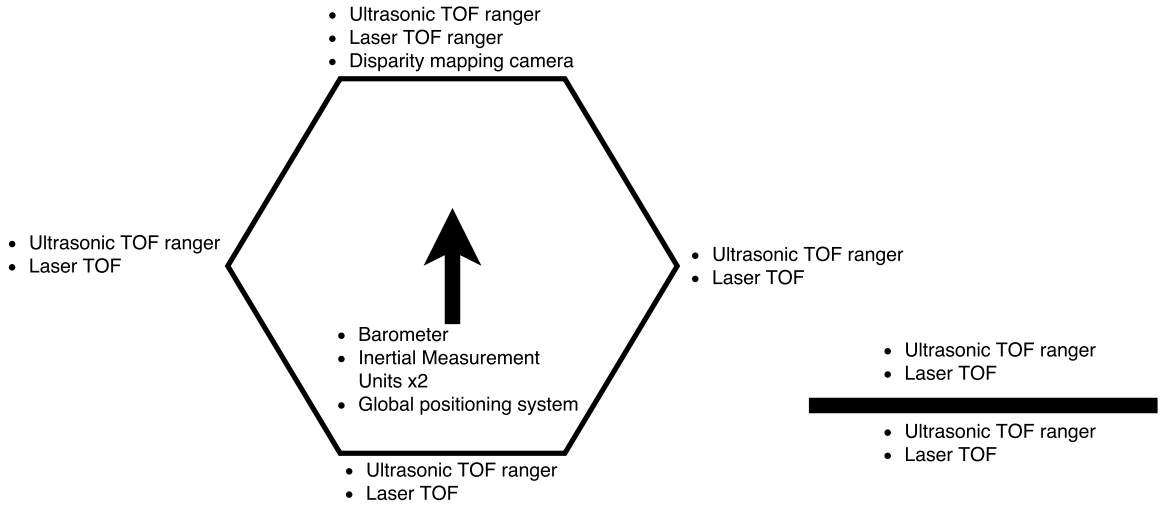

FIGURE 3.2. Diagram showing top-down (left) and front (right) views of the drone indicating where sensory apparatus has been placed.

This will provide ample data to work on as well as redundancy in the case of failure of one of the sensors.

3.3.2 Sensor Characterisation

Sensor characterisation was achieved through data sheet information except for the disparity mapping which was done manually by the project focusing on it. Table 3.1 gives sensor characteristics.

Sensor	**Units**	**Min. Range**	**Max Range**	**Sensor Variance**
Ultrasonic time-of-flight [10]	*cm*	2	400	4
Laser time-of-flight [11]	*cm*	0	120	1
Stereo camera disparity Map	*cm*	10	5500	10
Barometer [12]	*mbar*	10	1200	3
Global positioning system [13]	*m*	N/A	N/A	2
Inertial measurement unit (1) [14]	*o*	N/A	N/A	8.75
Inertial measurement unit (2) [15]	*o*	N/A	N/A	5

Table 3.1: Sensor characteristics

The camera used was the Tara - USB 3.0 Stereo Vision Camera made by e-con Systems [16]. Though versatile it is too expensive to have more than one.

CHAPTER 4

INTRODUCING THE KALMAN FILTER

When looking at combining data from various sources to provide estimates better than could have been achieved from each piece of data individually, there are currently four main algorithms to choose from; the central limit theorem, Bayesian networks, Dempster-Shafer theory or the Kalman filter.

4.1 Filter Options

The central limit theorem is derived from probability theory and posits that for a large number of independent uniformly distributed events with well defined mean and finite variance, their mean will tend towards a normal distribution [17]. This theory is not applicable to the project as the number of data points per iteration will be fairly small, and both the mean and variance will change as the drone moves.

Bayesian networks, otherwise known as Bayes networks or belief networks, are probabilistic directed acyclic graphs [18]. Although most often seen using binary true/false examples, Bayesian networks can be equally used for variables with either discreet or non-discreet values. One of the main uses for Bayesian networks is to predict the unknown values of nodes in the network using known nodes that the unknown node is dependent on such as the example shown in figure 4.1. In this project however each measurement will be assumed to be independent and so the effectiveness of a Bayesian network would be very minimal.

The Dempster-Shafer theory considers itself a generalisation of the Bayesian network algorithm where independent questions (nodes) also have a plausibility based on the probability of a related question. This "plausibility" is fundamentally different to a probability distribution characterised by belief, describing not the likelihood of a statement being true but instead describing to what degree similar question's results would influence the initial question's certainty [19].

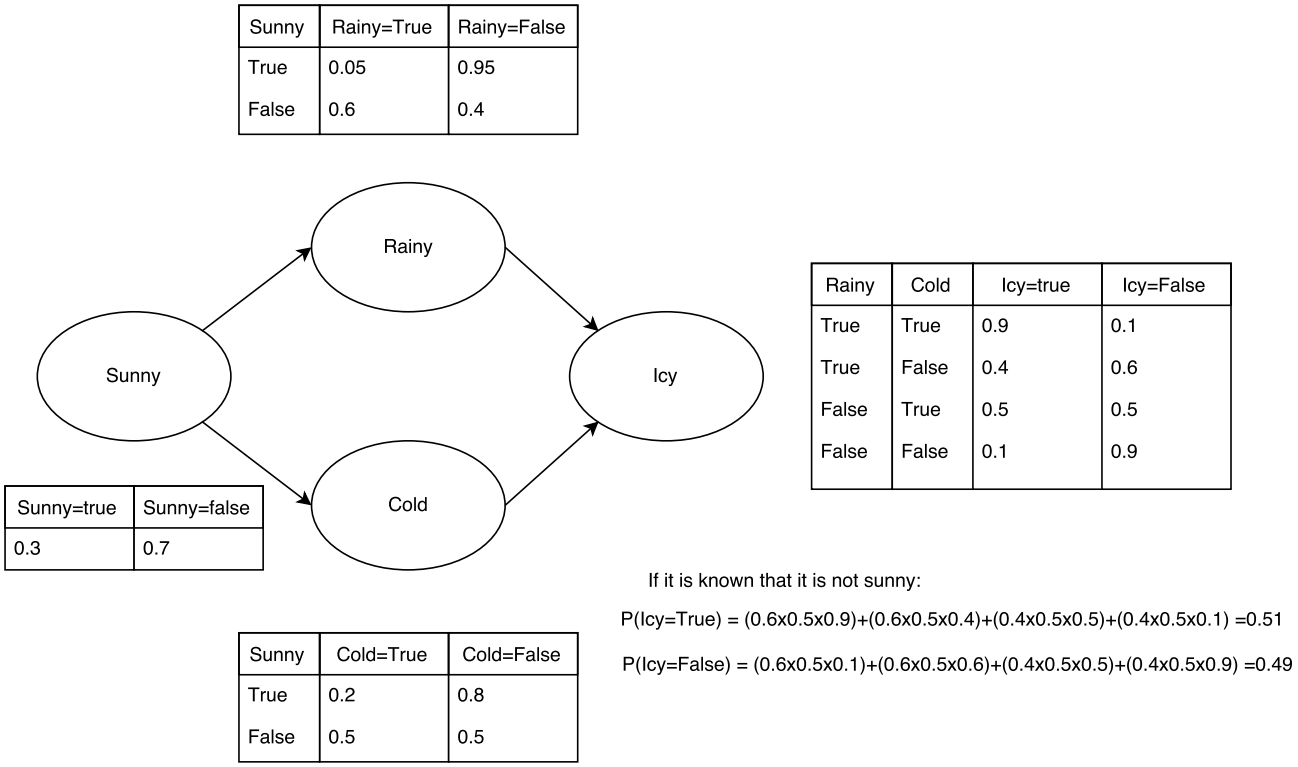

FIGURE 4.1. Diagram and calculations for a simple Bayesian network.

This theory could yield a suitable sensor fusion implementation however it is overly complex for this situation. DST is designed to deal with a degree of ignorance caused by incomplete data or sensors that are not well modelled, a scenario that should be mitigated in this situation through proper sensor characterisation.

The Kalman filter takes independent measurements of values, along with their known variance, and combines them to provide a new estimate with a lower variance [20]. This algorithm most closely matches the problem scenario and will be the one utilised going forward. Several non-linear variants of the Kalman filter exist which do not require state transition or observation models that are linear functions of the state. They are however substantially more complex, generally don't return an optimal estimate and are susceptible to divergence [21]. For the short-range sensors used, their response can generally be assumed to be linear. The GPS and IMU units do have linear error so will also not require a non-linear filter.

4.2 Explaining the Kalman Filter

Developed by Rudolf E. Kálmán and published in 1960, the filter uses Bayesian inference and joint probability distribution to estimate new data in discreet time. The algorithm uses two

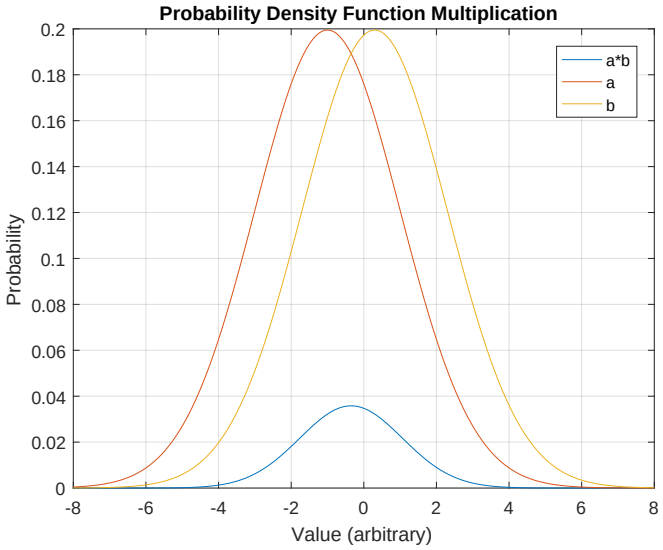

FIGURE 4.2. Graph showing the result of multiplying two probability distributions together. The result would then be renormalised to have a total probability of one.

principal steps; prediction of the new state from the old state along with any system inputs, followed by incorporation of all measurements taken. At all times the variance, or uncertainty, of the system is stored. Although this variance is not assumed to necessarily be Gaussian, it does yield an exact conditional probability estimate for that case [20].

In essence the process can be thought of as a form of weighted averaging where extrapolations or measurements with more certainty are given higher weighting.

The Kalman filter sees the world as a collection of variables with Gaussian distributed values and relies upon combination of these distributions to function; figure 4.2 shows graphically how two probability distributions are multiplied together to obtain a tighter distribution. Therefore key to understanding and implementing the filter is proper characterisation of the state and measurements. The state is the filter's representation of the variables to be estimated with a mean μ representing the best estimate. Each state item also has a variance σ^2 modelling the uncertainty in the measurement. Since the variance for one data item might influence the estimated value of another, the variances are combined into a symmetric covariance matrix which accounts for this [22][23][24]. For instance the estimated distance is correlated to the measured velocity and vice versa, this allows additional information to be retrieved from each sensor reading.

4.3 Step by Step Analysis of the Filter

Starting with the simple case where the one-dimensional vertical position and velocity of the drone needs to be estimated. The state here will be a vector containing the estimation means as shown:

$$\chi = \begin{vmatrix} pos \\ vel \end{vmatrix}$$

with an associated covariance matrix:

$$P = \begin{vmatrix} \sigma_{pp} & \sigma_{pv} \\ \sigma_{vp} & \sigma_{vv} \end{vmatrix}$$

Assuming there are two sensors, one measuring the height of the drone in meters (m), and one measuring the vertical velocity of the drone in meters per second (m/s). Their values will be stored in the vector:

$$z = \begin{vmatrix} posMeas \\ velMeas \end{vmatrix}$$

also with an associated covariance matrix:

$$R = \begin{vmatrix} \sigma_{pMpM} & \sigma_{pMvM} \\ \sigma_{vMpM} & \sigma_{vMvM} \end{vmatrix}$$

In the case of independent sensors, as is often the case, the covariance matrix will be diagonal. It is worth noting that the accuracy of the filter relies heavily on the accuracy of the covariance matrices and hence on proper characterisation of the state and the measurement apparatus.

From here the filter proceeds sequentially with step *n* being derived from step *n-1* combined with the control inputs and measurement readings.

The Kalman filter only ever uses the last time step to calculate the next, along with any information gained through control inputs or sensors. An initial state is required to begin the algorithm; while an accurate starting point is certainly very helpful, the filter will converge towards the true state fairly quickly.

4.3.1 Step One: Extrapolation with Control Input

4.3.1.1 Extrapolation

From the state values in the previous time step we can extrapolate to where we expect the state values to be now. Intuitively, the previous position and velocity will have an effect on the new position while only the previous velocity will effect the new velocity. Formally written from simple kinematic principals:

$$pos_n = pos_{n-1} + \Delta t vel_{n-1}$$

$$vel_n = vel_{n-1}$$

In matrix form:

$$\begin{vmatrix} pos_n \\ vel_n \end{vmatrix} = \begin{vmatrix} 1 & \Delta t \\ 0 & 1 \end{vmatrix} \begin{vmatrix} pos_{n-1} \\ vel_{n-1} \end{vmatrix}$$

This transformational matrix is known as the state transition matrix, denoted F, and details how each variable in the state matrix effects every other when we extrapolate to the next time step.

Having updated the prediction the covariance now also needs updating. Using the identity:

$$Cov(\chi) = \sigma$$

$$Cov(A\chi) = A\sigma A^T$$

It can be shown that since:

$$\hat{\chi}_{n|n-1} = F\chi_{n-1|n-1}$$

then:

$$P_{n|n-1} = FP_{n-1|n-1}F^T$$

4.3.1.2 Control input

At this stage any control levers can be introduced to the model. In this example, we will look at the idea of the drone having control over its acceleration. The acceleration can be introduced into the system using a similar style of matrix to that used in the extrapolation earlier where each element represents the effect of a control lever on every part of the state. This matrix is referred to as the input or control matrix, denoted B, and will be described in this example as:

$$B = \begin{vmatrix} dt^2/2 \\ dt \end{vmatrix}$$

from the kinematic equations:

$$pos_n = pos_{n-1} + \Delta t\, vel_{n-1} + \frac{1}{2} u\, \Delta t^2$$

$$vel_n = vel_{n-1} + u\, \Delta t$$

where u = acceleration.

The control process has its own associated variance caused by imperfect application in a real world environment; in this example for instance variance could be introduced through turbulence. Here the variance Q will simply be:

$$Q = \text{input variance} \times BB^T$$

4.3.1.3 Overall

The combined overall equations for this step are:

$$\hat{\chi}_{n|n-1} = F\hat{\chi}_{n-1|n-1} + Bu$$

where u is the input acceleration, and:

$$P_{n|n-1} = FP_{n-1|n-1}F^T + Q$$

describes the covariance update.

4.3.2 Step Two: Measurements Update

Now that there is a first estimate for where the drone should be based on the information available from the last step, the measurement information acquired this step can be utilised. Once again there will be a mean estimate value for each sensor (i.e. the reading) and an associated covariance matrix for the sensory apparatus as a whole.

4.3.3 Mapping the Readings to the State

Firstly there needs to exist an association matrix which maps the measurements vector onto the state space. In this example where we have the measurement vector:

$$z = \begin{vmatrix} posMeas \\ velMeas \end{vmatrix}$$

the association matrix is simply:

$$H = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

showing that the measured position has a one-to-one relationship with the state position, no relationship to the state velocity and vice versa for the measured velocity.

4.3.3.1 The Innovation Vector

From the measurement vector containing the sensor readings the innovation vector is derived. This describes the measurement residual or, more intuitively, the difference between each reading and the state estimate from step one and is given by:

$$Inn = z - H\hat{\chi}_{n|n-1}$$

The covariance of this innovation vector is derived from the estimation covariance and the measurement covariance as shown:

$$\sigma_{Inn} = HP_{n|n-1}H^T + R$$

4.3.3.2 Combination Through the Kalman Gain

The innovation now needs to be combined with the step one estimate to give an overall prediction of the state. The Kalman gain is used to weight the two previous steps, and can be thought of as the proportional confidence of the measurements relative to our extrapolated state prediction from step one. It should therefore be intuitive that the Kalman gain is derived from the covariances which describe the probable error. The Kalman gain is given in the equation below:

$$K = \frac{P_{n|n-1}H^T}{\sigma_{Inn}}$$

The state estimate can now be updated simply with the equation:

$$\hat{\chi}_{n|n} = \hat{\chi}_{n|n-1} + KInn$$

The final step then is to calculate the new covariance of the state estimate $P_{n|n}$ as to be used in the next algorithm iteration using:

$$P_{n|n} = P_{n|n-1} - KHP_{n|n-1}$$

4.3.4 Summary

The transition (F), control/input (B) and measurement (H) matrices stay the same throughout the entire filter run time as they are defined by equations for physical laws and are not dependent on any of the run time variables. The covariance matrix describing sensor noise is also constant, however it may be a function of other factors such as the current measurement e.g. the noise may increase as a measured distance increases.

CHAPTER 5

UTILISING THE KALMAN FILTER

The Kalman filter described in chapter 4 is clearly a very basic model dealing with only one dimension. In the reality of this project though, the drone will have to work in relation to a three dimensional world. This means that the filter will need to be expanded to reflect these additional dimensions, as well as take in and provide any additional data deemed necessary for operation of the drone's systems.

5.1 Expanding to Three Dimensional Movement

With one dimension the simplistic state contained only two items, position and velocity. Three dimensional position and velocity will require more items for each, however there are a number of different ways to represent them [25].

5.1.1 Three Dimensional Position

There are three main positional coordinate systems; Cartesian, cylindrical and spherical.

- Cartesian coordinates use the classic orthogonal axes, for three dimensions this requires three axes usually denoted x, y and z.
- Cylindrical coordinates use a single axis (usually denoted z), and then specify a location in space by giving a radial angle (Θ) and a distance from a point along the axis (r) as exampled in figure 5.1.
- Spherical coordinates require a polar axis and a perpendicular plane. The spatial position is then given by an angle around the plane (azimuth), an angle perpendicular to the plane (elevation) and a distance from the origin.

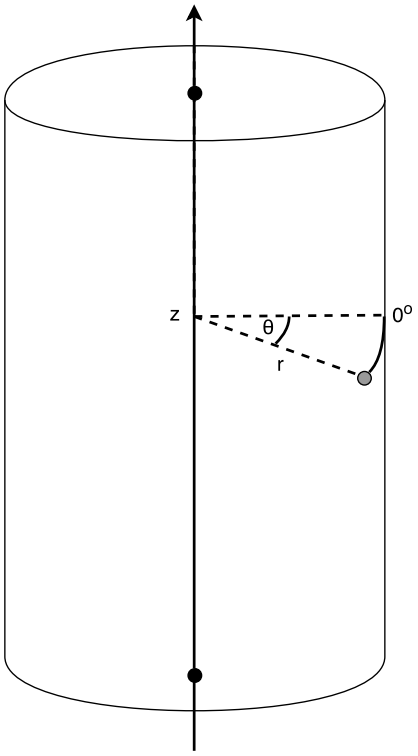

FIGURE 5.1. Diagram showing the workings of cylindrical coordinates.

As can be seen, all three systems require three state items to represent a spatial location. Since Cartesian coordinates are those most widely understood and with no obvious drawbacks when compared with the other systems, it seems to make sense that they should be used.

5.1.2 Three Dimensional Velocity

Velocity description is very similar to positional, although the cylindrical system is more complicated to visualise in this instance.

- The Cartesian system uses the three axes to describe movement relative to each orthogonal direction. These three combined give a single velocity vector.

- In the cylindrical system movement along the single axis is easy to visualise, being identical to one of the axes in the Cartesian system. The equivalent of the other two axes however is a function of both the change in radial angle and distance from the central axis.

- The spherical coordinate system in contrast is very intuitive. The origin is thought of as the moving object, the azimuth and elevation describe the movement direction, and the distance from the origin is instead the absolute velocity.

Although the spherical coordinate system is an elegant and intuitive system, the Cartesian system was chosen to match the positional one so as to avoid confusion and to make kinematic equations more simple to apply.

5.1.3 Three Dimensional Orientation

Several of the other systems on the drone require accurate orientation data to function. To this end including orientation was seen to be a natural step due to the importance of the information and the multiple inertial measurement units that will be available to provide the relevant data. Again there are a number of coordinate systems available, though these are more or less identical to the velocity systems, albeit with the absolute velocity component being unimportant. This has bigger ramifications than might be expected. The spherical and cylindrical systems used above are known as the quaternion system when related to orientation and have a couple of large benefits [26][27].

- In the previous systems there is a component relating to distance or absolute velocity. In the quaternion representation this is wholly unnecessary and can be removed. This reduces the number of state items required to two when compared to the three the other systems use.
- The Cartesian system experiences a phenomenon known as gimbal lock when two of the axes occupy the same plane, effectively leading to a loss of orthogonality. This causes a loss of one of the three degrees of freedom. The quaternion system does not suffer from this effect.

Once again, the Cartesian system was chosen since it fits mathematically into the existing framework more easily. Additionally, drones do not pitch at close to 90^o under normal operating conditions so the issue of gimbal lock should not occur.

CHAPTER 6

SIMULATION TESTING

Functional and performance testing of the Kalman filter was done in MATLAB using the code described in appendix B. This choice was made due to the intuitive graphical outputs produced as well as easy to use line breaks which allow careful tracking of the mathematics step by step. Simulations run for a fixed time length in discrete steps with data being stored for analysis afterwards.

6.1 Data Generation

Data generation comes in two stages, simulation of the drone's true movement and generation of sensory data.

6.1.1 Movement Simulation

The main lever for control of the drone's movement is through the control inputs. These can either be hard coded with constants, follow an equation over time, or be a constrained random variable. A mix of these were used over the course of general testing, however randomised inputs were considered best for final testing as they removed any question of bias. It could be argued that this is not representative of realistic drone behaviour however ability to work under worst case scenario is a good general starting point for testing of the filter. Control inputs were produced using a random number generator scaled to appropriate values. No official operational specifications exist for the drone used in this project, so maximum speed and acceleration were estimated based on the performance of similar drones in video [28]. This shows a maximum speed around 14 m/s with an approximate acceleration of 3 m/s^2. Simulated movement will therefore be limited to 15 m/s with a maximum acceleration of 5 m/s^2. The process noise (the noise associated

with drone adherence to control inputs) is more difficult to estimate and is dependent on factors such as weather conditions. Without real testing of the drone under specific conditions available to obtain noise readings, noise of equal proportion to the control inputs was used as a medium error estimate.

6.1.2 Sensor Simulation

Sensor data was provided by adding random noise to the true values thus producing noisy readings. Noise was scaled based on the sensors variance.

All random numbers were produced using MATLAB's *randn* function that produces random numbers with a mean value of zero and variance of one.

6.2 Results Analysis

There are a number of metrics that can be used to measure the effectiveness of the filter. For this analysis we'll use the term *average error*, this will be defined as the mean error in state estimate compared to the true value over every timestep of the simulation. These will generally be given separately for position, velocity and orientation.

The code was run 500 times to produce the results detailed below and shown in figures 6.1 to 6.3. At 20 seconds per run with 0.1 second resolution the total time step sample size is 10,000 which was considered large enough to be representative. The raw values used to produce the aforementioned figures are available in appendix A.

A general trend over the data is that average extrapolated error and average filter produced error are very close. This is to be expected as the extrapolation step is generally good at short term estimation i.e. from one time step to the next. However if we took away the measurement data we would see that the filter would drift away from the true values due to accumulated error originating from the external noise. In this way drift is compensated for by the sensors, and even noisy data like that produced by the GPS can provide usable long term trends. This phenomenon is also used inside the inertial measurement units where the gyroscope is accurate but tends to drift over time, while the accelerometer is not so accurate but has no tendency to drift [29].

Tables A.2 and A.3 in appendix A show the effect that removing sensor data has on the filter estimates; larger average error that increases with simulation run time. Position estimate error can be seen to be especially increased, this is due to its derivation from velocity meaning that error is introduced two-fold through both the previous position and velocity estimate.

6.2.1 Average Positional Error

The average measured error for position varies substantially between axes as evidenced in figure 6.1. This can be attributed to the differences in sensory apparatus available for each. The y

position has lowest average measured error due to using only relatively low variance ultrasonic and laser sensors. The x position has increased average measured error due to inclusion of the stereo camera disparity map in its available data. Addition of the very noisy GPS measurement to the z position sensory apparatus correspondingly raises the average measured error for that axis. As can be noted however, the difference in measured error does not adversely affect the filtered average error. Indeed the z position has a lower average error than the x position. This demonstrates that the Kalman filter is making efficient use of even very noisy data through appropriate weighting.

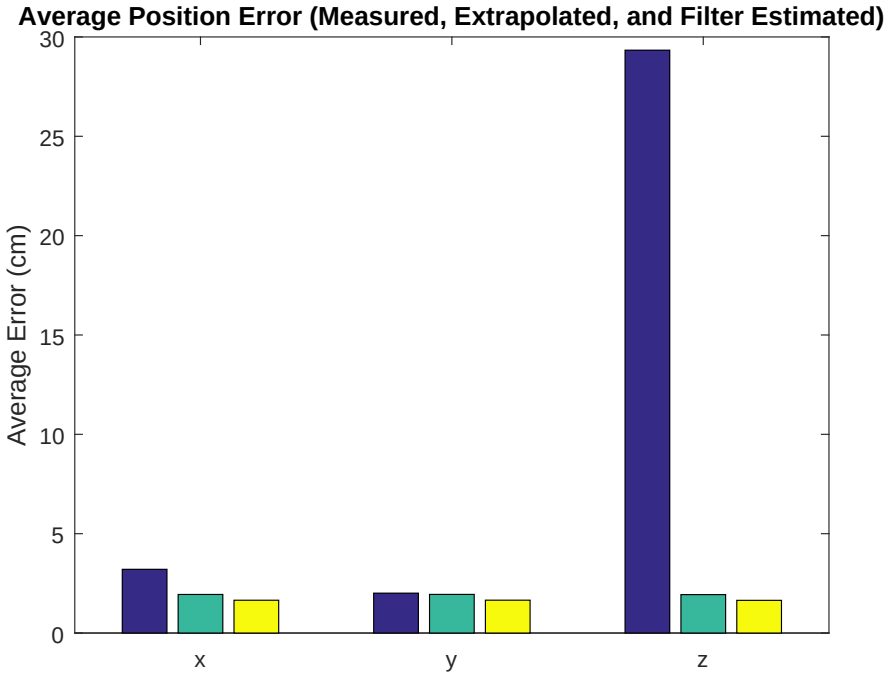

FIGURE 6.1. Bar chart showing average error from true results for measured, extrapolated and filter obtained position in the x, y and z axis.

6.2.2 Average Velocity Error

Average velocity error displayed in figure 6.2 shows a similar but even more exaggerated case of that seen in the z position data, very clearly showing that the extrapolation step contributes heavily to the much improved filter estimate.

6.2.3 Average Orientation Error

The orientation sensory apparatus does not incorporate any relatively high noise components, therefore it can be seen in figure 6.3 that the extrapolation and filter estimates are not so distant

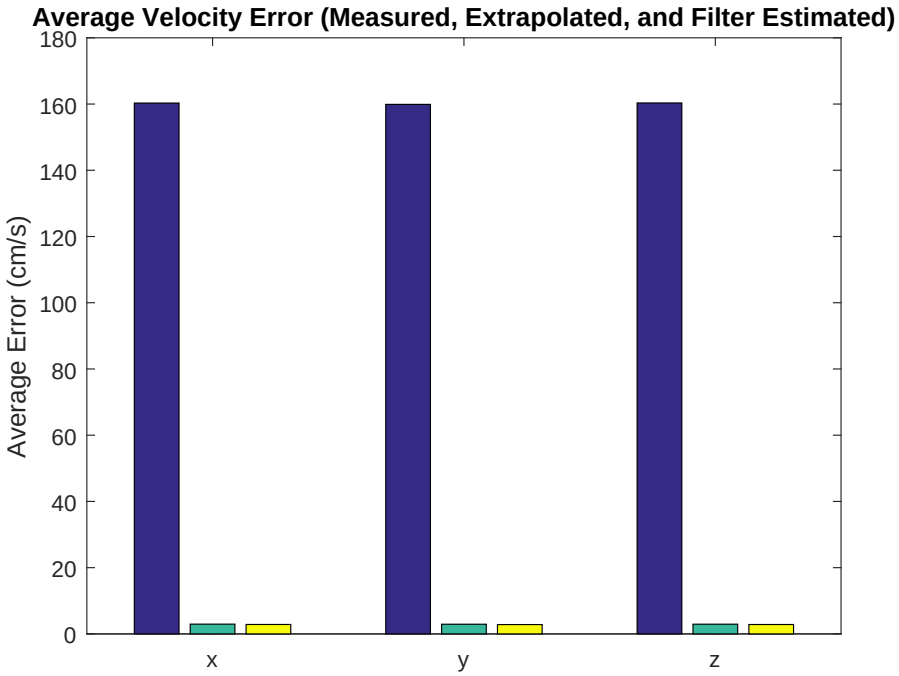

FIGURE 6.2. Bar chart showing average error from true results for measured, extrapolated and filter obtained velocity in the x, y and z axis.

from the measured as was the case in the previous examples. A marked improvement can however still be observed.

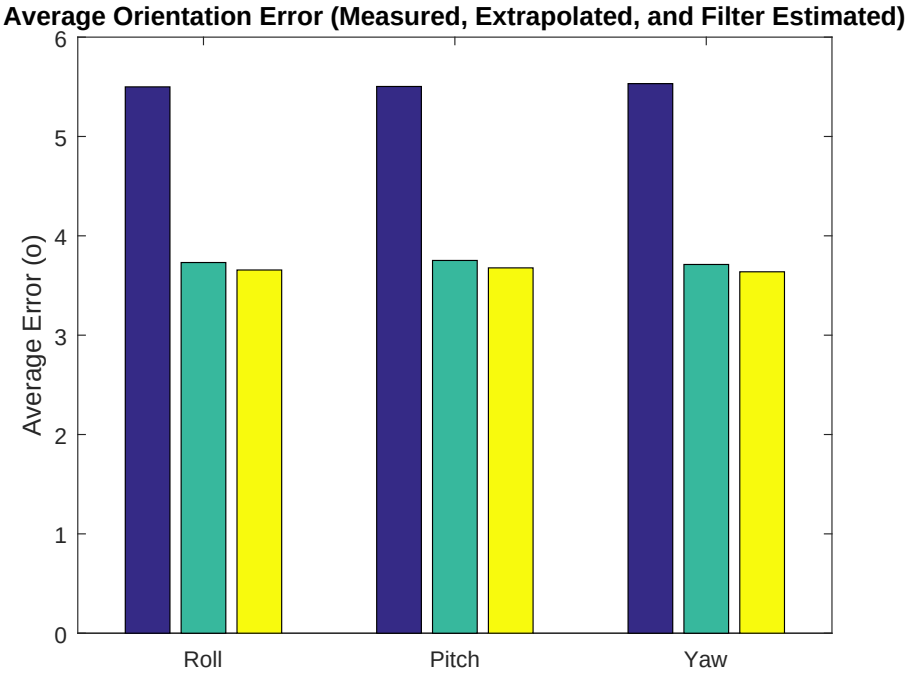

FIGURE 6.3. Bar chart showing average error from true results for measured, extrapolated and filter obtained orientation in pitch, roll and yaw.

CHAPTER 7

FINAL IMPLEMENTATION

Golang was the language of choice for the final implementation of the algorithm for a number of reasons; the easy concurrency that Golang offers, ease of integration with other parts of the drone software and also just for the opportunity of learning a new language. On the downside, Golang's matrix libraries are not as mature as those found in older languages such as Python and so some of the ported code was not as idiomatic as would have been otherwise possible.

The ported code can be seen described in appendix C. It is implemented using the gonum matrix package "mat64" and provides identical functionality to that of the original MATLAB code. The program was tested with basic examples to assure functional correctness using reference results from the MATLAB code.

Following this the mathematical operations were split into chunks that could be calculated concurrently to improve timing efficiency. Concurrency is the act of running different portions of a program at the same time using more than one thread to reduce the program's run time. The code described in appendix D shows how these concurrent parts were split into go functions that communicate with the main function via channels. Channels are a typed message pipeline that acts as a mediator for inter-function communication; the sending function places a message into the channel that is later collected by the receiving function. Channel calls are blocking, meaning that the functions will not continue past that line until valid information has been sent and received through the channel [30].

CHAPTER 8

CONCLUSION AND FURTHER WORK

The Kalman filter seems to be an effective method for data fusion with the sensory apparatus currently available. Its application to the drone outlined in this project is both advantageous and relevant when looking to achieve the goals set out in the introduction of this paper.

8.1 Conclusion

With reference to the results produced in chapter 6, the performance increases are shown in table 8.1.

Aspect	Multiplicative Performance Increase
Position	6.96
Velocity	55.8
Orientation	1.50

Table 8.1: Multiplicative increase in performance of the Kalman filter relative to naive sensor averaging.

The greatest performance increase was seen in velocity due to the inaccurate sensor equipment available for that state aspect, and lowest for orientation where relatively accurate sensor data is available. There is still room for improvement as will be discussed in section 8.2 however it is believed that sufficient base functionality has been established. This can be confirmed with real world testing to ensure the safety of the platform is never compromised.

8.2 Further Work

8.2.1 Integration and Real World testing

The filter code needs to be integrated with the code handling sensory apparatus so as to receive data input, and also to the flight controller so as to provide output data. As the target code is written largely in the same language this should be a fairly simple process. Following that, further testing of the filter needs to be done once the drone itself is fully functional to finish characterisation of all the noise parameters.

8.2.2 Improved Sensory Apparatus

Discussed in chapter 3 was the promising but infeasible LiDAR system. There could be area here to develop a more cost efficient version of this system using cheap off the shelf components which would greatly enhance the information input available.

8.2.3 Kalman Filter Variants

In chapter 4 the idea of Kalman filter variants was looked at and dismissed as unnecessary. In the event of a future change in instrumentation or use conditions, these variants may become useful. The extended Kalman filter is the standard used in many commercially available products. It does however have some aforementioned drawbacks in terms of being suboptimal, not being completely stable under some starting conditions, and generally being difficult to implement [31]. The much lauded alternative to the EKF is the Sigma-point (unscented) Kalman filter which may well be a better alternative to the EKF for further developments of this project [32].

8.2.4 Multiple Filters

Pushing the data through multiple filters then combining the estimates could be an interesting experiment to see if more accurate estimates can be obtained. This paper would suggest a filter based on Dempster-Schafer filter to be a good starting consideration.

8.2.5 Dynamic Noise Updates

As has been stressed previously, the Kalman filter relies heavily on accurate characterisation of the sensor behaviour. This also needs to be done prior to run time and the values cannot be altered later on if the sensor behaviour changes. Reasons for characteristic changes could include but are not limited to alteration in environmental conditions or sensor malfunction. Ideally the sensor noise would be able to be be dynamically updated during filter runtime to reflect estimated accuracy. Eom *et al.* looked at using a neural network to estimate measurement noise covariance [33]. This could either be investigated further, or a more naive approach could be looked at

which has slightly lower performance but with the benefit of greatly reduced complexity and computational load.

APPENDIX A

Simulated Average Error Raw Values

Tables A.1 to A.3 give the simulation output values for average error with and without inclusion of the sensory apparatus in the filter. Average error is described as the mean distance from the true simulated value each filter time step over a large number of simulation runs. Table A.3 shows the affect that doubling simulation run time has on state estimates when no sensory apparatus input is available.

Aspect	Axis	Data Source	Units	Average Error
Position	x	Measured	cm	3.21
		Extrapolated		1.96
		Filtered		1.66
	y	Measured		2.00
		Extrapolated		1.95
		Filtered		1.65
	z	Measured		29.33
		Extrapolated		1.94
		Filtered		1.65
Velocity	x	Measured	cm/s	160.35
		Extrapolated		2.97
		Filtered		2.87
	y	Measured		160.32
		Extrapolated		2.99
		Filtered		2.89
	z	Measured		160.32
		Extrapolated		2.95
		Filtered		2.84
Orientation	roll	Measured	o	5.51
		Extrapolated		3.76
		Filtered		3.68
	pitch	Measured		5.51
		Extrapolated		3.73
		Filtered		3.66
	yaw	Measured		5.50
		Extrapolated		3.78
		Filtered		3.70

Table A.1: Average error observed in MATLAB code for each data source from the true simulated value with sensor data included in the filter over 10,000 time steps.

Aspect	Axis	Data Source	Units	Average Error
Position	x	Measured	cm	-
		Extrapolated		54.21
		Filtered		54.21
	y	Measured		-
		Extrapolated		53.74
		Filtered		53.74
	z	Measured		-
		Extrapolated		51.64
		Filtered		51.64
Velocity	x	Measured	cm/s	-
		Extrapolated		7.81
		Filtered		7.81
	y	Measured		-
		Extrapolated		7.71
		Filtered		7.71
	z	Measured		-
		Extrapolated		7.32
		Filtered		7.32
Orientation	roll	Measured	o	-
		Extrapolated		7.26
		Filtered		7.26
	pitch	Measured		-
		Extrapolated		7.68
		Filtered		7.68
	yaw	Measured		-
		Extrapolated		8.00
		Filtered		8.00

Table A.2: Average error observed in MATLAB code for each data source from the true simulated value without sensor data included in the filter over 10,000 time steps.

Aspect	Axis	Data Source	Units	Average Error
Position	x	Measured	cm	-
		Extrapolated		157.48
		Filtered		157.48
	y	Measured		-
		Extrapolated		148.85
		Filtered		148.85
	z	Measured		-
		Extrapolated		152.70
		Filtered		152.70
Velocity	x	Measured	cm/s	-
		Extrapolated		11.15
		Filtered		11.15
	y	Measured		-
		Extrapolated		10.79
		Filtered		10.79
	z	Measured		-
		Extrapolated		11.05
		Filtered		11.05
Orientation	roll	Measured	o	-
		Extrapolated		10.79
		Filtered		10.79
	pitch	Measured		-
		Extrapolated		10.61
		Filtered		10.61
	yaw	Measured		-
		Extrapolated		10.68
		Filtered		10.68

Table A.3: Average error observed in MATLAB code for each data source from the true simulated value without sensor data included in the filter over 20,000 time steps.

APPENDIX B

KALMAN FILTER MATLAB CODE

The Kalman filter code written in MATLAB, heavily commented. This includes the lines required to generate graphs and provide average error values. This will function as is in the basic MATLAB program.

```
1   % function kalman(duration, dt)
2   %
3   % Kalman filter simulation for a drone's 3D position and velocity
        with roll, pitch and yaw.
4   %
5   % State [posx(cm); posy(cm); posz(cm); velx(cm/s); vely(cm/s); velz(
        cm/s); roll(o); pitch(o); yaw(o)]
6   %
7   % E = covariance
8   %
9   close all;
10
11  %%%%TESTING PURPOSES%%%%
12  % INITIALISE DATA GATHERING
13  TotalPosxErrorExt = 0;   % cumulative position extrapolation error
14  TotalPosyErrorExt = 0;   % cumulative position extrapolation error
15  TotalPoszErrorExt = 0;   % cumulative position extrapolation error
16  TotalPosxErrorMeas = 0;  % cumulative position measurement error
17  TotalPosyErrorMeas = 0;  % cumulative position measurement error
```

```
18      TotalPoszErrorMeas = 0;  % cumulative position measurement error
19      TotalPosxErrorFil = 0;   % cumulative position error
20      TotalPosyErrorFil = 0;   % cumulative position error
21      TotalPoszErrorFil = 0;   % cumulative position error
22      TotalVelxErrorExt = 0;   % cumulative velocity extrapolation error
23      TotalVelyErrorExt = 0;   % cumulative velocity extrapolation error
24      TotalVelzErrorExt = 0;   % cumulative velocity extrapolation error
25      TotalVelxErrorMeas = 0;  % cumulative velocity measurement error
26      TotalVelyErrorMeas = 0;  % cumulative velocity measurement error
27      TotalVelzErrorMeas = 0;  % cumulative velocity measurement error
28      TotalVelxErrorFil = 0;   % cumulative velocity error
29      TotalVelyErrorFil = 0;   % cumulative velocity error
30      TotalVelzErrorFil = 0;   % cumulative velocity error
31      TotalRollErrorExt = 0;   % cumulative position extrapolation error
32      TotalPitchErrorExt = 0;  % cumulative position extrapolation error
33      TotalYawErrorExt = 0;    % cumulative position extrapolation error
34      TotalRollErrorMeas = 0;  % cumulative position measurement error
35      TotalPitchErrorMeas = 0; % cumulative position measurement error
36      TotalYawErrorMeas = 0;   % cumulative position measurement error
37      TotalRollErrorFil = 0;   % cumulative position error
38      TotalPitchErrorFil = 0;  % cumulative position error
39      TotalYawErrorFil = 0;    % cumulative position error
40      TotalSensoravgError = 0;
41      %%%%TESTING PURPOSES%%%%
42
43      % INPUTS
44      duration = 20;    %length of simulation (seconds)
45      dt = 0.1;          %step size (seconds)
46
47      % Measurement noise due to imperfect sensors.
48      sensornoise = [4     % ultra x1 measurement noise (cm)
49          4     % ultra x2 measurement noise (cm)
50          4     % ultra y1 measurement noise (cm)
51          4     % ultra y2 measurement noise (cm)
52          4     % ultra z1 measurement noise (cm)
53          4     % ultra z2 measurement noise (cm)
54          1     % laser x1 measurement noise (cm)
55          1     % laser x2 measurement noise (cm)
```

```
56          1      % laser y1 measurement noise (cm)
57          1      % laser y2 measurement noise (cm)
58          1      % laser z1 measurement noise (cm)
59          1      % laser z1 measurement noise (cm)
60          10     % disparity x measurement noise (cm)
61          200    % GPS z measurement noise (cm)
62          200    % GPS velocity x measurement noise (cm/s)
63          200    % GPS velocity y measurement noise (cm/s)
64          200    % GPS velocity z measurement noise (cm/s)
65          10     % barometer z measurement noise (cm)
66          8.75   % IMU1 angular velocity roll measurement noise (o/s)
67          8.75   % IMU1 angular velocity pitch measurement noise (o/s)
68          8.75   % IMU1 angular velocity yaw measurement noise (o/s)
69          5      % IMU2 angular velocity roll measurement noise (o)
70          5      % IMU2 angular velocity pitch measurement noise (o)
71          5];    % IMU2 angular velocity yaw measurement noise (o)
72
73      % External noise due to imperfect replication of instruction wrt the
            environment.
74      externalnoise = [1      % x acceleration noise (cm/sec^2)
75                       1      % y acceleration noise (cm/sec^2)
76                       1      % z acceleration noise (cm/sec^2)
77                       1      % roll input noise (o/sec)
78                       1      % pitch input noise (o/sec)
79                       1];    % yaw input noise (o/sec)
80
81      % Computational matrices.
82      F = [1 0 0 dt 0 0 0 0 0
83           0 1 0 0 dt 0 0 0 0
84           0 0 1 0 0 dt 0 0 0
85           0 0 0 1 0 0 0 0 0
86           0 0 0 0 1 0 0 0 0
87           0 0 0 0 0 1 0 0 0
88           0 0 0 0 0 0 1 0 0
89           0 0 0 0 0 0 0 1 0
90           0 0 0 0 0 0 0 0 1 ]; % transition matrix, used to extrapolate
                the new state from the previous acceleration
91      B = [dt^2/2 0 0 0 0 0
```

```
92          0 dt^2/2 0 0 0 0
93          0 0 dt^2/2 0 0 0
94          dt 0 0 0 0 0
95          0 dt 0 0 0 0
96          0 0 dt 0 0 0
97          0 0 0 dt 0 0
98          0 0 0 0 dt 0
99          0 0 0 0 0 dt     ]; % input matrix, used to account for the
                effect of inputs on the new state when extrapolating from the
                 old
100     H = [1 0 0 0 0 0 0 0 0
101          1 0 0 0 0 0 0 0 0
102          0 1 0 0 0 0 0 0 0
103          0 1 0 0 0 0 0 0 0
104          0 0 1 0 0 0 0 0 0
105          0 0 1 0 0 0 0 0 0
106          1 0 0 0 0 0 0 0 0
107          1 0 0 0 0 0 0 0 0
108          0 1 0 0 0 0 0 0 0
109          0 1 0 0 0 0 0 0 0
110          0 0 1 0 0 0 0 0 0
111          0 0 1 0 0 0 0 0 0
112          1 0 0 0 0 0 0 0 0
113          0 0 1 0 0 0 0 0 0
114          0 0 0 1 0 0 0 0 0
115          0 0 0 0 1 0 0 0 0
116          0 0 0 0 0 1 0 0 0
117          0 0 1 0 0 0 0 0 0
118          0 0 0 0 0 0 1 0 0
119          0 0 0 0 0 0 0 1 0
120          0 0 0 0 0 0 0 0 1
121          0 0 0 0 0 0 1 0 0
122          0 0 0 0 0 0 0 1 0
123          0 0 0 0 0 0 0 0 1]; % measurement matrix, maps the sensor
                measurements onto the state space (has to be hand altered
                upon change of sensor input or state space)
124
125     for i = 1;
```

```
126
127     % Initial state.
128     x = [0; 0; 0; 0; 0; 0; 0; 0; 0]; % initial state vector, [position x/
            y/z; velocity x/y/z; orientation roll/pitch/yaw]
129     xhat = x; % initial state estimate (accurate)
130     %xhat = [1000; 2000; 1500; 20; 40; 30; 0; 0; 0];    % initial state
            estimate (custom, comment out if not needed)
131     R = diag(sensornoise).^2;  % measurement error E
132     Qu = repmat(cat(1,externalnoise(1),externalnoise(2),externalnoise(3),
            externalnoise(1),externalnoise(2),externalnoise(3),externalnoise
            (4),externalnoise(5),externalnoise(6)).^2,1,length(x)) .* (B * B')
            ; % process noise E (of xhat); describes the effect of process
            noise on estimated position and velocity
133     P = Qu; % initial estimation E (of xhat)
134
135     % Initialize arrays for later plotting.
136     pos = [];       % true position array
137     poshat = [];    % estimated position array
138     posmeas = [];   % measured position array
139     vel = [];       % true velocity array
140     velmeas = [];   % measures velocity array
141     velhat = [];    % estimated velocity array
142
143     for t = 0 : dt: duration,
144         % Use a constant commanded acceleration in cm/sec^2 and a
                constant commanded rotation in o/sec.
145         u = [randn; randn; randn; 1; 1; 1];
146         % Simulate the linear system.
147         ProcessNoise = cat(1,externalnoise(1),externalnoise(2),
                externalnoise(3),externalnoise(1),externalnoise(2),
                externalnoise(3),externalnoise(4),externalnoise(5),
                externalnoise(6)) .* ((randn(length(x),1))); % generates
                process noise for each state element
148         x = F * x + B * u + ProcessNoise;
149         % Simulate the noisy measurements
150         MeasNoise = sensornoise .* randn(length(sensornoise),1);    %
                randomly weighted measurement noise
```

```
151         z = H * x + MeasNoise;    % actual position with added noise to
                create simulated (mean) measurement readings
152         % Extrapolate the most recent state estimate to the present time;
153         % this is the new prediction based on extrapolation from the
                position
154         % and velocity of the previous state plus any compensation that
                needs
155         % to be made due to acceleration.
156         xhat = F * xhat + B * u;
157         %%%%TESTING PURPOSES%%%%
158         TotalPosxErrorExt = TotalPosxErrorExt + abs(x(1)-xhat(1));
159         TotalPosyErrorExt = TotalPosyErrorExt + abs(x(2)-xhat(2));
160         TotalPoszErrorExt = TotalPoszErrorExt + abs(x(3)-xhat(3));
161         TotalVelxErrorExt = TotalVelxErrorExt + abs(x(4)-xhat(4));
162         TotalVelyErrorExt = TotalVelyErrorExt + abs(x(5)-xhat(5));
163         TotalVelzErrorExt = TotalVelzErrorExt + abs(x(6)-xhat(6));
164         TotalRollErrorExt = TotalRollErrorExt + abs(x(7)-xhat(7));
165         TotalPitchErrorExt = TotalPitchErrorExt + abs(x(8)-xhat(8));
166         TotalYawErrorExt = TotalYawErrorExt + abs(x(9)-xhat(9));
167         %%%%TESTING PURPOSES%%%%
168         % Form the Innovation vector;
169         % measured position - extrapolated position, describes the
                measurement
170         % residual.
171         Inn = z - H * xhat;
172         % Compute the E of the Innovation;
173         % Epp + measurement error E.
174         InnE = H * P * H' + R;
175         % Form the Kalman Gain matrix;
176         % this represents the proportional confidence we have in our
177         % measurements vs our extrapolated prediction.
178         K = (((F * P * F') + Qu) * H') / InnE;
179         % Update the state estimate;
180         % the estimate plus the confidence weighted innovation.
181         xhat = xhat + K * Inn;
182         % Correct roll, pitch and yaw to be mod 360
183         xroll = mod(xhat(7),360);
184         xpitch = mod(xhat(8),360);
```

```
185         xyaw = mod(xhat(9),360);
186         % Compute the E of the estimation error for next iteration.
187         P = ((F * P * F') + Qu) - (K * H * (F * P * F' + Qu));
188         % Save some parameters for plotting later (must be hand altered
                upon change of sensor input).
189         pos = [pos; x(1) x(2) x(3)];
190         posmeas = [posmeas; z(1) z(2) z(3)];
191         poshat = [poshat; xhat(1) xhat(2) xhat(3)];
192         vel = [vel; x(4) x(5) x(6)];
193         velmeas = [velmeas; z(4) z(5) z(6)];
194         velhat = [velhat; xhat(4) xhat(5) xhat(6)];
195         %%%%TESTING PURPOSES%%%%
196         TotalPosxErrorMeas = TotalPosxErrorMeas + abs(x(1)-z(1)) + abs(x
                (1)-z(2)) + abs(x(1)-z(7)) + abs(x(1)-z(8)) + abs(x(1)-z(13));
197         TotalPosyErrorMeas = TotalPosyErrorMeas + abs(x(2)-z(3)) + abs(x
                (2)-z(4)) + abs(x(2)-z(9)) + abs(x(2)-z(10));
198         TotalPoszErrorMeas = TotalPoszErrorMeas + abs(x(3)-z(5)) + abs(x
                (3)-z(6)) + abs(x(3)-z(11)) + abs(x(3)-z(12)) + abs(x(3)-z(14)
                ) + abs(x(3)-z(18));
199         TotalVelxErrorMeas = TotalVelxErrorMeas + abs(x(4)-z(15));
200         TotalVelyErrorMeas = TotalVelyErrorMeas + abs(x(5)-z(16));
201         TotalVelzErrorMeas = TotalVelzErrorMeas + abs(x(6)-z(17));
202         TotalRollErrorMeas = TotalRollErrorMeas + abs(x(7)-z(19)) + abs(x
                (7)-z(22));
203         TotalPitchErrorMeas = TotalPitchErrorMeas + abs(x(8)-z(20)) + abs
                (x(8)-z(23));
204         TotalYawErrorMeas = TotalYawErrorMeas + abs(x(9)-z(21)) + abs(x
                (9)-z(24));
205         TotalPosxErrorFil = TotalPosxErrorFil + abs(x(1)-xhat(1));
206         TotalPosyErrorFil = TotalPosyErrorFil + abs(x(2)-xhat(2));
207         TotalPoszErrorFil = TotalPoszErrorFil + abs(x(3)-xhat(3));
208         TotalVelxErrorFil = TotalVelxErrorFil + abs(x(4)-xhat(4));
209         TotalVelyErrorFil = TotalVelyErrorFil + abs(x(5)-xhat(5));
210         TotalVelzErrorFil = TotalVelzErrorFil + abs(x(6)-xhat(6));
211         TotalRollErrorFil = TotalRollErrorFil + abs(x(7)-xhat(7));
212         TotalPitchErrorFil = TotalPitchErrorFil + abs(x(8)-xhat(8));
213         TotalYawErrorFil = TotalYawErrorFil + abs(x(9)-xhat(9));
```

```
214         for i2=1:numel(z);                                          %
                sensor errors added together weighted by sensor noise
215             for i3=1:numel(x);
216                 TotalSensoravgError = TotalSensoravgError + abs( (x(i3)-z
                        (i2) * 1/sensornoise(i2)) * (H(i2) * x(i3)) );
217             end
218         end
219         %%%%TESTING PURPOSES%%%%
220     end
221
222     end
223
224     %%%%TESTING PURPOSES$$$$
225     % Calculate average position and velocity errors
226     AvgPosxErrorExt = TotalPosxErrorExt / ((duration/dt)*i);
227     AvgPosyErrorExt = TotalPosyErrorExt / ((duration/dt)*i);
228     AvgPoszErrorExt = TotalPoszErrorExt / ((duration/dt)*i);
229     AvgVelxErrorExt = TotalVelxErrorExt / ((duration/dt)*i);
230     AvgVelyErrorExt = TotalVelyErrorExt / ((duration/dt)*i);
231     AvgVelzErrorExt = TotalVelzErrorExt / ((duration/dt)*i);
232     AvgRollErrorExt = TotalRollErrorExt / ((duration/dt)*i);
233     AvgPitchErrorExt = TotalPitchErrorExt / ((duration/dt)*i);
234     AvgYawErrorExt = TotalYawErrorExt / ((duration/dt)*i);
235     AvgPosxErrorMeas = (TotalPosxErrorMeas/5) / ((duration/dt)*i);
236     AvgPosyErrorMeas = (TotalPosyErrorMeas/4) / ((duration/dt)*i);
237     AvgPoszErrorMeas = (TotalPoszErrorMeas/6) / ((duration/dt)*i);
238     AvgVelxErrorMeas = (TotalVelxErrorMeas) / ((duration/dt)*i);
239     AvgVelyErrorMeas = (TotalVelyErrorMeas) / ((duration/dt)*i);
240     AvgVelzErrorMeas = (TotalVelzErrorMeas) / ((duration/dt)*i);
241     AvgRollErrorMeas = (TotalRollErrorMeas/2) / ((duration/dt)*i);
242     AvgPitchErrorMeas = (TotalPitchErrorMeas/2) / ((duration/dt)*i);
243     AvgYawErrorMeas = (TotalYawErrorMeas/2) / ((duration/dt)*i);
244     AvgPosxErrorFil = TotalPosxErrorFil / ((duration/dt)*i);
245     AvgPosyErrorFil = TotalPosyErrorFil / ((duration/dt)*i);
246     AvgPoszErrorFil = TotalPoszErrorFil / ((duration/dt)*i);
247     AvgVelxErrorFil = TotalVelxErrorFil / ((duration/dt)*i);
248     AvgVelyErrorFil = TotalVelyErrorFil / ((duration/dt)*i);
249     AvgVelzErrorFil = TotalVelzErrorFil / ((duration/dt)*i);
```

```
250     AvgRollErrorFil = TotalRollErrorFil / ((duration/dt)*i);
251     AvgPitchErrorFil = TotalPitchErrorFil / ((duration/dt)*i);
252     AvgYawErrorFil = TotalYawErrorFil / ((duration/dt)*i);
253     AvgSensoravgError = (TotalSensoravgError*(sum(1/sensornoise))) / ((
            duration/dt)*i*nnz(H));
254
255     figure('name','Position Comparison');
256     analysis = [AvgPosxErrorMeas AvgPosxErrorExt AvgPosxErrorFil;
            AvgPosyErrorMeas AvgPosyErrorExt AvgPosyErrorFil; AvgPoszErrorMeas
             AvgPoszErrorExt AvgPoszErrorFil];
257     bar(analysis);
258     ylabel('Average Error (cm)');
259     labels = {'x','y','z'};
260     set(gca,'xticklabel',labels)
261     title('Average Position Error (Measured, Extrapolated, and Filter
            Estimated)');
262
263     figure('name','Velocity Comparison');
264     analysis = [AvgVelxErrorMeas AvgVelxErrorExt AvgVelxErrorFil;
            AvgVelyErrorMeas AvgVelyErrorExt AvgVelyErrorFil; AvgVelzErrorMeas
             AvgVelzErrorExt AvgVelzErrorFil];
265     bar(analysis);
266     ylabel('Average Error (cm/s)');
267     labels = {'x','y','z'};
268     set(gca,'xticklabel',labels)
269     title('Average Velocity Error (Measured, Extrapolated, and Filter
            Estimated)');
270
271     figure('name','Orientation Comparison');
272     analysis = [AvgRollErrorMeas AvgRollErrorExt AvgRollErrorFil;
            AvgPitchErrorMeas AvgPitchErrorExt AvgPitchErrorFil;
            AvgYawErrorMeas AvgYawErrorExt AvgYawErrorFil];
273     bar(analysis);
274     ylabel('Average Error (o)');
275     labels = {'Roll','Pitch','Yaw'};
276     set(gca,'xticklabel',labels)
277     title('Average Orientation Error (Measured, Extrapolated, and Filter
            Estimated)');
```

```
278
279     %%%%TESTING PURPOSES%%%%
280
281     % RESULT PLOTTING
282     t = 0 : dt : duration;
283
284     % Position Comparison
285     figure('name', 'Position Comparison')
286
287     subplot(2,2,1);
288     plot(t,pos(:,1), t,poshat(:,1));
289     grid;
290     xlabel('Time (sec)');
291     ylabel('Position (cm)');
292     title('Figure 1 - Vehicle X Position (True and Estimated)')
293
294     subplot(2,2,2);
295     plot(t,pos(:,2), t,poshat(:,2));
296     grid;
297     xlabel('Time (sec)');
298     ylabel('Position (cm)');
299     title('Figure 2 - Vehicle Y Position (True and Estimated)')
300
301     subplot(2,2,3);
302     plot(t,pos(:,3), t,posmeas(), t,poshat(:,3));
303     grid;
304     xlabel('Time (sec)');
305     ylabel('Position (cm)');
306     title('Figure 3 - Vehicle Z Position (True, Measured, and Estimated)'
          )
307
308     % Position Error
309     figure('name', 'Position Error')
310
311     subplot(2,2,1);
312     plot(t,pos(:,1)-posmeas(:,1), t,pos(:,1)-poshat(:,1));
313     grid;
314     xlabel('Time (sec)');
```

```
315     ylabel('Position Error (cm)');
316     title('Figure 1 - X Position Measurement Error and Position
            Estimation Error');
317
318     subplot(2,2,2);
319     plot(t,pos(:,2)-posmeas(:,2), t,pos(:,2)-poshat(:,2));
320     grid;
321     xlabel('Time (sec)');
322     ylabel('Position Error (cm)');
323     title('Figure 2 - Y Position Measurement Error and Position
            Estimation Error');
324
325     subplot(2,2,3);
326     plot(t,pos(:,3)-posmeas(:,3), t,pos(:,3)-poshat(:,3));
327     grid;
328     xlabel('Time (sec)');
329     ylabel('Position Error (cm)');
330     title('Figure 3 - Z Position Measurement Error and Position
            Estimation Error');
331
332    % Velocity Comparison
333     figure('name','Velocity Comparison')
334
335     subplot(2,2,1);
336     plot(t,vel(:,1), t,velmeas(:,1), t,velhat(:,1));
337     grid;
338     xlabel('Time (sec)');
339     ylabel('Velocity (cm/sec)');
340     title('Figure 1 - X Velocity (True, Measured, and Estimated)');
341
342     subplot(2,2,2);
343     plot(t,vel(:,2), t,velmeas(:,2), t,velhat(:,2));
344     grid;
345     xlabel('Time (sec)');
346     ylabel('Velocity (cm/sec)');
347     title('Figure 2 - Y Velocity (True, Measured, and Estimated)');
348
349     subplot(2,2,3);
```

```
350     plot(t,vel(:,3), t,velmeas(:,3), t,velhat(:,3));
351     grid;
352     xlabel('Time (sec)');
353     ylabel('Velocity (cm/sec)');
354     title('Figure 3 - Z Velocity (True, Measured, and Estimated)');
355
356     % Velocity Error
357     figure('name', 'Velocity Error')
358
359     subplot(2,2,1);
360     plot(t,vel(:,1)-velmeas(:,1), t,vel(:,1)-velhat(:,1));
361     grid;
362     xlabel('Time (sec)');
363     ylabel('Velocity Error (cm/sec)');
364     title('Figure 1 - X Velocity Measurement Error and Velocity
            Estimation Error');
365
366     subplot(2,2,2);
367     plot(t,vel(:,2)-velmeas(:,2), t,vel(:,2)-velhat(:,2));
368     grid;
369     xlabel('Time (sec)');
370     ylabel('Velocity Error (cm/sec)');
371     title('Figure 2 - Y Velocity Measurement Error and Velocity
            Estimation Error');
372
373     subplot(2,2,3);
374     plot(t,vel(:,3)-velmeas(:,3), t,vel(:,3)-velhat(:,3));
375     grid;
376     xlabel('Time (sec)');
377     ylabel('Velocity Error (cm/sec)');
378     title('Figure 3 - Z Velocity Measurement Error and Velocity
            Estimation Error');
379
380     % 3D Position Over Time
381     figure('name', '3D Position Over Time')
382     scatter3(pos(:,1),pos(:,2),pos(:,3),10,t,'filled');
383     hold on;
384     scatter3(poshat(:,1),poshat(:,2),poshat(:,3),20,t);
```

```
385     ax = gca;
386     ax.XDir = 'reverse';
387     xlabel('x-position (cm)')
388     ylabel('y-position (cm)')
389     zlabel('z-position (cm)')
390     colormap parula;
391     cb = colorbar;
392     cb.Label.String = 'Time (s)';
393     rotate3d on;
394
395     % 3D Position w/ Velocity
396     figure('name', '3D Position with Velocity')
397     scatter3(pos(:,1),pos(:,2),pos(:,3),10,sqrt((vel(:,1).^2)+(vel(:,2)
            .^2)+(vel(:,3).^2)),'filled');
398     hold on;
399     scatter3(poshat(:,1),poshat(:,2),poshat(:,3),20,sqrt((velhat(:,1).^2)
            +(velhat(:,2).^2)+(velhat(:,3).^2)));
400     ax = gca;
401     ax.XDir = 'reverse';
402     xlabel('x-position (cm)')
403     ylabel('y-position (cm)')
404     zlabel('z-position (cm)')
405     colormap parula;
406     cb = colorbar;
407     cb.Label.String = 'Velocity (cm/s)';
408     rotate3d on;
```

APPENDIX C

KALMAN FILTER GOLANG CODE

The Kalman filter code written in Golang with basic commenting. This includes functions to generate a command line printout of the state vector and its P matrix each iteration. Golang compiler 1.8 was used to produce the executable file.

```
/* Port of Kalman Filter from MATLAB to Go */

package main

import("fmt"
            "math"
            "github.com/gonum/matrix/mat64")

func main(){
        //Declare
        var duration float64 = 0.1
        var dt = 0.1
        var t float64                   //time
        var Pt mat64.Dense              //p_{t|t-1} estimation covariance
        var K mat64.Dense               //Kalman gain matrix
        var Inn mat64.Dense             //Innovation vector
        var InnE mat64.Dense            //Innovation covariance
        var a mat64.Dense               //temp variable a
        var b mat64.Dense               //temp variable b
```

```
20          var c mat64.Dense                   //temp variable c
21          var d mat64.Dense                   //temp variable d
22          var e mat64.Dense                   //temp variable e
23          var f mat64.Dense                   //temp variable f
24          var g mat64.Dense                   //temp variable g
25          var h mat64.Dense                   //temp variable h
26          var i mat64.Dense                   //temp variable i
27          var j mat64.Dense                   //temp variable j
28
29
30          //Intialise
31          sensnoise := mat64.NewDense(6,1,[]float64{0,0,0,0,0,0}) //sensor
                noise
32          R := mat64.NewDense(6,6,[]float64{
33                  math.Pow(sensnoise.At(0,0),2),0,0,0,0,0,
34                  0,math.Pow(sensnoise.At(1,0),2),0,0,0,0,
35                  0,0,math.Pow(sensnoise.At(2,0),2),0,0,0,
36                  0,0,0,math.Pow(sensnoise.At(3,0),2),0,0,
37                  0,0,0,0,math.Pow(sensnoise.At(4,0),2),0,
38                  0,0,0,0,0,math.Pow(sensnoise.At(5,0),2),
39                  })      //measurement error covariance
40          extnoise := mat64.NewDense(6,1,[]float64{0,0,0,0,0,0})  //
                external noise
41          F := mat64.NewDense(9,9,[]float64{
42                  1,0,0,dt,0,0,0,0,0,
43                  0,1,0,0,dt,0,0,0,0,
44                  0,0,1,0,0,dt,0,0,0,
45                  0,0,0,1,0,0,0,0,0,
46                  0,0,0,0,1,0,0,0,0,
47                  0,0,0,0,0,1,0,0,0,
48                  0,0,0,0,0,0,1,0,0,
49                  0,0,0,0,0,0,0,1,0,
50                  0,0,0,0,0,0,0,0,1,
51                  })      //transition matrix
52          B := mat64.NewDense(9,6,[]float64{
53                  math.Pow(dt,2)/2,0,0,0,0,0,
54                  0,math.Pow(dt,2)/2,0,0,0,0,
55                  0,0,math.Pow(dt,2)/2,0,0,0,
```

```
56                  dt,0,0,0,0,0,
57                  0,dt,0,0,0,0,
58                  0,0,dt,0,0,0,
59                  0,0,0,dt,0,0,
60                  0,0,0,0,dt,0,
61                  0,0,0,0,0,dt,
62                  })      //control input matrix
63          H := mat64.NewDense(6,9,[]float64{
64                  1,0,0,0,0,0,0,0,0,
65                  0,1,0,0,0,0,0,0,0,
66                  0,0,1,0,0,0,0,0,0,
67                  0,0,0,1,0,0,0,0,0,
68                  0,0,0,0,1,0,0,0,0,
69                  0,0,0,0,0,1,0,0,0,
70                  })      //measurement matrix
71          X := mat64.NewDense(9,1,[]float64{0,
72                                            0,
73                                            0,
74                                            0,
75                                            0,
76                                            0,
77                                            0,
78                                            0,
79                                            0})       //initial state vector: x
                                                 /y/z pos, x/y/z vel, orientation
                                                  roll/pitch/yaw
80          Xhat := X //initial estimate
81          U := mat64.NewDense(6,1,[]float64{0,
82                                            0,
83                                            0,
84                                            0,
85                                            0,
86                                            0})       //control matrix
87
88          a.Mul(B,B.T()) //9,9
89          a0 := []float64{extnoise.At(0,0),extnoise.At(1,0),extnoise.At
                (2,0),extnoise.At(0,0),extnoise.At(1,0),extnoise.At(2,0),
                extnoise.At(3,0),extnoise.At(4,0),extnoise.At(5,0)}       //
```

```
            extnoise slice for Qu
90          a1 := a0
91          for n:=0;n<8;n++ {
92                  a1 = append(a1,a0...)
93          }       //slice for Qu
94          Qu := mat64.NewDense(9,9,a1) //proto Qu I
95          Qu.MulElem(Qu,Qu)       //proto Qu II
96          Qu.MulElem(Qu,&a)       //Qu
97          P := Qu //initial estimation covariance
98
99          for t=0; t<duration; t+=dt {
100
101                 //Readings
102                 Z := mat64.NewDense(6,1,[]float64{0,
103                                                   0,
104                                                   0,
105                                                   0,
106                                                   0,
107                                                   0})   //measurement vector: x/y
                                                              /z pos, x/y/z vel
108
109                 //Calculations
110                 //Xhat = F*Xhat + B*U
111                 b.Mul(F,Xhat)           //9,1
112                 c.Mul(B,U)              //9,1
113                 Xhat.Add(&b,&c)         //9,1
114                 //Inn = Z - H*Xhat
115                 d.Mul(H,Xhat)           //6,1
116                 Inn.Sub(Z,&d)           //6,1
117                 //InnE = H*P*H' + R
118                 e.Mul(H,P)              //6,9
119                 f.Mul(&e,H.T())         //6,6
120                 InnE.Add(&f,R)          //6,6
121                 //Pt = F*P*F'
122                 g.Mul(F,P)              //9,9
123                 g.Mul(&g, F.T())        //9,9
124                 Pt.Add(&g,Qu)           //9,9
125                 //K = (Pt + Qu)*H'*InnE^-1
```

```
126                 h.Mul(&Pt,H.T())            //9,6
127                 i.Inverse(&InnE)            //6,6
128                 K.Mul(&h,&i)                //9,6
129                 //Xhat = Xhat + K*Inn
130                 b.Mul(&K,&Inn)              //9,1
131                 Xhat.Add(Xhat,&b)           //9,1
132                 //P = (Pt + Qu) - (K*P*F' + Qu)
133                 g.Mul(&K,H)                 //9,9
134                 j.Mul(&g,&Pt)               //9,9
135                 g.Add(&Pt,Qu)               //9,9
136                 P.Sub(&g,&j)                //9,9
137         }
138
139         //Xhat printout: x/y/z position, x/y/z velocity, roll/pitch/yaw
                orientation
140         fmt.Printf("\n  Xhat = \n")
141         for na:=0;na<9;na++{
142                 if P.At(na,0)<0{
143                 fmt.Printf(" %f\n",Xhat.At(na,0))
144                 }else{
145                 fmt.Printf("  %f\n",Xhat.At(na,0))
146                 }
147         }
148
149         //P printout
150         fmt.Printf("\n  P = \n")
151         for ni:=0;ni<9;ni++{
152                 for nj:=0;nj<9;nj++{
153                         if P.At(nj,ni)<0{
154                                 fmt.Printf(" %f",P.At(nj,ni))
155                         }else{
156                                 fmt.Printf("  %f",P.At(nj,ni))
157                         }
158                 }
159                 fmt.Printf("\n")
160         }
161         fmt.Printf("\n")
162 }
```

APPENDIX D

CONCURRENT CALCULATION KALMAN FILTER GOLANG CODE

The Kalman filter code written in Golang, basic commenting. The mathematics is made concurrent where possible through use of goroutines and buffered communication channels. Golang compiler 1.8 was used to produce the executable file.

```
/* Port of Kalman Filter from MATLAB to Go */

package main

import("fmt"
       "math"
       "github.com/gonum/matrix/mat64")

func main(){
    //Declare
    var duration float64 = 0.1
    var dt = 0.1
    var t float64           //time
    var Pt mat64.Dense      //p_{t|t-1} estimation covariance
    var K mat64.Dense       //Kalman gain matrix
    var Inn mat64.Dense     //Innovation vector
    var InnE mat64.Dense    //Innovation covariance
    var a mat64.Dense       //temp variable a
    var b mat64.Dense       //temp variable b
```

```
20      var c mat64.Dense         //temp variable c
21      var d mat64.Dense         //temp variable d
22      var e mat64.Dense         //temp variable e
23      var f mat64.Dense         //temp variable f
24      var g mat64.Dense         //temp variable g
25      var h mat64.Dense         //temp variable h
26      var i mat64.Dense         //temp variable i
27      var j mat64.Dense         //temp variable j
28
29
30      //Intialise
31      sensnoise := mat64.NewDense(6,1,[]float64{0,0,0,0,0,0}) //sensor
            noise
32      R := mat64.NewDense(6,6,[]float64{
33           math.Pow(sensnoise.At(0,0),2),0,0,0,0,0,
34           0,math.Pow(sensnoise.At(1,0),2),0,0,0,0,
35           0,0,math.Pow(sensnoise.At(2,0),2),0,0,0,
36           0,0,0,math.Pow(sensnoise.At(3,0),2),0,0,
37           0,0,0,0,math.Pow(sensnoise.At(4,0),2),0,
38           0,0,0,0,0,math.Pow(sensnoise.At(5,0),2),
39           }) //measurement error covariance
40      extnoise := mat64.NewDense(6,1,[]float64{0,0,0,0,0,0}) //external
            noise
41      F := mat64.NewDense(9,9,[]float64{
42           1,0,0,dt,0,0,0,0,0,
43           0,1,0,0,dt,0,0,0,0,
44           0,0,1,0,0,dt,0,0,0,
45           0,0,0,1,0,0,0,0,0,
46           0,0,0,0,1,0,0,0,0,
47           0,0,0,0,0,1,0,0,0,
48           0,0,0,0,0,0,1,0,0,
49           0,0,0,0,0,0,0,1,0,
50           0,0,0,0,0,0,0,0,1,
51           }) //transition matrix
52      B := mat64.NewDense(9,6,[]float64{
53           math.Pow(dt,2)/2,0,0,0,0,0,
54           0,math.Pow(dt,2)/2,0,0,0,0,
55           0,0,math.Pow(dt,2)/2,0,0,0,
```

```
56          dt,0,0,0,0,0,
57          0,dt,0,0,0,0,
58          0,0,dt,0,0,0,
59          0,0,0,dt,0,0,
60          0,0,0,0,dt,0,
61          0,0,0,0,0,dt,
62          }) //control input matrix
63      H := mat64.NewDense(6,9,[]float64{
64          1,0,0,0,0,0,0,0,0,
65          0,1,0,0,0,0,0,0,0,
66          0,0,1,0,0,0,0,0,0,
67          0,0,0,1,0,0,0,0,0,
68          0,0,0,0,1,0,0,0,0,
69          0,0,0,0,0,1,0,0,0,
70          }) //measurement matrix
71      X := mat64.NewDense(9,1,[]float64{0,
72                                        0,
73                                        0,
74                                        0,
75                                        0,
76                                        0,
77                                        0,
78                                        0,
79                                        0}) //initial state vector: x/y/z
                                               pos, x/y/z vel, orientation
                                              roll/pitch/yaw
80      Xhat := X //initial estimate
81      U := mat64.NewDense(6,1,[]float64{0,
82                                        0,
83                                        0,
84                                        0,
85                                        0,
86                                        0}) //control matrix
87
88      a.Mul(B,B.T()) //9,9
89      a0 := []float64{extnoise.At(0,0),extnoise.At(1,0),extnoise.At(2,0),
            extnoise.At(0,0),extnoise.At(1,0),extnoise.At(2,0),extnoise.At
            (3,0),extnoise.At(4,0),extnoise.At(5,0)}    //extnoise slice for Qu
```

```
90      a1 := a0
91      for n:=0;n<8;n++ {
92          a1 = append(a1,a0...)
93      }   //slice for Qu
94      Qu := mat64.NewDense(9,9,a1) //proto Qu I
95      Qu.MulElem(Qu,Qu)    //proto Qu II
96      Qu.MulElem(Qu,&a)    //Qu
97      P := Qu //initial estimation covariance
98
99      for t=0; t<duration; t+=dt {
100
101         //Readings
102         Z := mat64.NewDense(6,1,[]float64{0,
103                                           0,
104                                           0,
105                                           0,
106                                           0,
107                                           0})   //measurement vector: x/y
                                                      /z pos, x/y/z vel
108
109         //FILTER MATHS
110         channelInn  := make(chan mat64.Dense, 1)
111         channelXhat := make(chan mat64.Dense, 1)
112         channeli    := make(chan mat64.Dense, 1)
113         channelInnE := make(chan mat64.Dense, 1)
114         channelh    := make(chan mat64.Dense, 1)
115         channelPt   := make(chan mat64.Dense, 1)
116
117         //Concurrent Calculations
118         go func(){
119             b.Mul(F,Xhat)          //9,1   1a
120             c.Mul(B,U)             //9,1   1a
121             Xhat.Add(&b,&c)        //9,1   2a
122             d.Mul(H,Xhat)          //6,1   3a
123             Inn.Sub(Z,&d)          //6,1   4a
124             channelInn <- Inn
125             channelXhat <- *Xhat
126         }()
```

```
127
128         go func(){
129             e.Mul(H,P)              //6,9    1b
130             f.Mul(&e,(H).T())       //6,6    2b
131             InnE.Add(&f,R)          //6,6    3b
132             i.Inverse(&InnE)        //6,6    4b
133             channeli     <- i
134             channelInnE <- InnE
135         }()
136
137         go func(){
138             g.Mul(F,P)              //9,9    1c
139             g.Mul(&g,(F).T())       //9,9    2c
140             Pt.Add(&g,Qu)           //9,9    3c
141             h.Mul(&Pt,(H).T())      //9,6    4c
142             g.Add(&Pt,Qu)           //9,9    4c
143             channelh     <- h
144             channelPt    <- Pt
145         }()
146
147         varInn   := <- channelInn
148         Inn      = varInn
149         varXhat  := <- channelXhat
150         *Xhat    = varXhat
151         vari     := <- channeli
152         i        = vari
153         varInnE  := <- channelInnE
154         InnE     = varInnE
155         varh     := <- channelh
156         h        = varh
157         varPt    := <- channelPt
158         Pt       = varPt
159
160         //Sequential Calculations
161         K.Mul(&h,&i)            //9,6    5c/4b
162         b.Mul(&K,&Inn)          //9,1    6c/5b
163         g.Mul(&K,H)             //9,9    6c/5b
164         j.Mul(&g,&Pt)           //9,9    7c/6b
```

```
165             P.Sub(&g,&j)            //9,9    8c/7b
166             Xhat.Add(Xhat,&b)       //9,1    7c/6b/3a
167         }
168
169             //Xhat printout: x/y/z position, x/y/z velocity, roll/pitch/yaw
                    orientation
170         fmt.Printf("\n  Xhat = \n")
171         for na:=0;na<9;na++{
172             if P.At(na,0)<0{
173             fmt.Printf(" %f\n",Xhat.At(na,0))
174             }else{
175             fmt.Printf("  %f\n",Xhat.At(na,0))
176             }
177         }
178
179         //P printout
180         fmt.Printf("\n  P = \n")
181         for ni:=0;ni<9;ni++{
182             for nj:=0;nj<9;nj++{
183                 if P.At(nj,ni)<0{
184                     fmt.Printf(" %f",P.At(nj,ni))
185                 }else{
186                     fmt.Printf("  %f",P.At(nj,ni))
187                 }
188             }
189             fmt.Printf("\n")
190         }
191         fmt.Printf("\n")
192 }
```

APPENDIX E

SOFTWARE LISTING

The table below lists all software used throughout this project.

Software	**Version**	**Publisher**
MATLAB	R2016a	MathWorks
Golang compiler	1.8 linux/amd64	Google Inc.
Microsoft Office Suite	2013	Microsoft
Adobe Acrobat Reader DC	2017.009.20044	Adobe
Overleaf	-	Writelatex Limited
Latex Template	-	Victor Brena

Table E.1: Software used in the duration of this project

BIBLIOGRAPHY

[1] $HobbyKing^{TM}$.
S550 hexcopter frame kit with integrated pcb 550mm (black).
https://hobbyking.com/en_us/s550-hexcopter-frame-kit-with-integrated-pcb-550mm-black.html.
(Date last accessed 21-May-2017).

[2] Lucas Vago Santana, Alexandre Santos Brandao, Mario Sarcinelli-Filho, and Ricardo Carelli.
A trajectory tracking and 3d positioning controller for the ar. drone quadrotor.
In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 756–767. IEEE, 2014.

[3] Xilin Yang, Luis Mejias, and Matt Garratt.
Multi-sensor data fusion for uav navigation during landing operations.
In *Proceedings of the 2011 Australian Conference on Robotics and Automation*, pages 1–10. Australian Robotics and Automation Association Inc., Monash University, 2011.

[4] Florian Ion Petrescu.
A New Doppler Effect.
BoD–Books on Demand, 2012.

[5] LiDAR-UK.
How does lidar work?
http://www.lidar-uk.com/how-lidar-works/, 2017.
(Date last accessed 26/04/2017).

[6] Arthur P Cracknell.
Introduction to Remote Sensing.
CRC press, 2007.

[7] Amar Nayegandhi.
Green, waveform lidarin topo-bathy mapping ‚Äìprinciples and applications.
https://www.ngs.noaa.gov/corbin/class_description/Nayegandhi_green_lidar.pdf, 2006.
(Date last accessed 25/04/2017).

[8] Slamtec.
Rplidar a2 the thinest lidar.
https://www.slamtec.com/en/Lidar, 2016.
(Date last accessed 26/04/2017).

[9] Daorier 10pcs ultrasonic hc-sr04 distance measuring transducer and obstacle detection module for arduino.
https://www.amazon.co.uk/DaoRier-Ultrasonic-Measuring-Transducer-Detection/dp/B06XK6VTQ1/ref=sr_1_3?ie=UTF8&qid=1493202561&sr=8-3&keywords=ultrasonic+sensor+10pcs, 2017.
(Date last accessed 26/04/2017).

[10] SRF04.
Ultra-Sonic Ranger.
Robot Electronics.
Same as HC-SR04.

[11] VL53L0X.
World smallest Time-of-Flight ranging and gesture detection sensor.
STMicroelectronics, May 2016.
Rev. 1.

[12] MS5611-01BA03.
Barometric Pressure Sensor, with stainless steel cap.
measurement specialities, October 2012.

[13] NEO-6M.
u-blox 6 GPS Modules.
ublox, December 2011.
Rev. E.

[14] L3GD20.
MEMS motion sensor: three-axis digital output gyroscope.
STMicroelectronics, August 2011.
Rev. 1.

[15] MPU-9250.
Product Specification.
InvenSense, January 2014.
Rev. 1.

[16] e-con Systems.

Tara - usb 3.0 stereo vision camera.
https://www.e-consystems.com/3D-USB-stereo-camera.asp.
(Date last accessed 21-May-2017).

[17] John Rice.
Mathematical statistics and data analysis.
Nelson Education, 2006.

[18] Eugene Charniak.
Bayesian networks without tears.
AI magazine, 12(4):50, 1991.

[19] Glenn Shafer.
Dempster–Shafer theory.
http://glennshafer.com/assets/downloads/articles/article48.pdf, 2002.
Date last accessed 27/04/2017.

[20] Rudolph Emil Kalman et al.
A new approach to linear filtering and prediction problems.
Transactions of the ASME – Journal of Basic Engineering, (82 (Series D)):35–45, 1960.

[21] Tine Lefebvre*, Herman Bruyninckx, and Joris De Schutter.
Kalman filters for non-linear systems: a comparison of performance.
International journal of Control, 77(7):639–653, 2004.

[22] Tim Babb.
How a kalman filter works in pictures.
http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/, 2015.
(Date last accessed 03-March-2017).

[23] R Faragher.
Understanding the basis of the kalman filter.
IEEE Signal Processing Magazine, pages 128–132, September 2012.

[24] D Simon.
Kalman filtering.
http://www.embedded.com/design/configurable-systems/4023342/
Kalman-Filtering, 2001.
(Date last accessed 28-February-2017).

[25] S Widnall and J Peraire.
Lecture L5 - other coordinate systems.
University Lecture, 2008.

[26] K Großekatthöfer and Z Yoon.
Introduction into quaternions for spacecraft attitude representation.
TU Berlin, 16, 2012.

[27] F Sebastian Grassia.
Practical parameterization of rotations using the exponential map.
Journal of graphics tools, 3(3):29–48, 1998.

[28] Hummingbird.UAV.
Speed run with a dji f550 copter.
`https://www.youtube.com/watch?v=PPPUZrrHswU`, May 2015.

[29] Kenneth Gade.
Introduction to inertial navigation and kalman filtering, October 2009.
Tutorial for IAIN World Congress, Stockholm, Sweden.

[30] A tour of go.
`https://tour.golang.org/`.
(Date last accessed 18-May-2017).

[31] Simon J Julier and Jeffrey K Uhlmann.
Unscented filtering and nonlinear estimation.
Proceedings of the IEEE, 92(3):401–422, 2004.

[32] Rudolph Van Der Merwe and Eric A Wan.
Sigma-point kalman filters for integrated navigation.
In *Proceedings of the 60th Annual Meeting of the Institute of Navigation (ION)*, pages 641–654, 2004.

[33] Ki Hwan Eom, Seung Joon Lee, Yeo Sun Kyung, Chang Won Lee, Min Chul Kim, and Kyung Kwon Jung.
Improved kalman filter method for measurement noise reduction in multi sensor rfid systems.
Sensors, 11(11):10266–10282, 2011.