TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karl Uibo 153620IVEM

ANALYSIS OF FLIGHT DATA FOR WEATHER DEPENDANT MULTIPLE EXECUTION AUTONOMOUS MISSIONS

Master's Thesis

Supervisor: Olev Märtens
Professor

Tallinn 2018

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Karl Uibo

07.05.2018

Abstract

This thesis explores how to achieve consecutive autonomous flights without a pilots input. Previously the focus of research has been in optimizing the flight characteristics of a multirotor but we take a higher level approach to find the requirements needed to be able to achieve fully autonomous flight with use cases in the industry.

We created *python* code to import data from individual log files into a database. An algorithm for filtering the database to find necessary data to design a model of the flight behavior of an individual multirotor is created. An analysis of voltage graphs of the available flights were analyzed and a requirement for a smart controller in each battery to monitor the state of charge, state of health and to identify each battery is found.

This thesis is written in English and is 35 pages long, including 4 chapters, 16 figures and 8 tables.

Annotatsioon

Selles töös otsitakse lahendust probleemile, kuidas saavutada autonoomset multirootori lennu missiooni nii, et lendude vahel ei peaks ükski inimene sekkuma samal ajal maksimeerides lennuaega ja lennu pikust. Senini on põhiliselt uuritud autopiloodi juht-algoritmide efektiivsust ja on loodud ilmastiku ja sensori müra kindlamaid algoritme. Selles töös vaadeldakse laiemat pilti, kus selleks, et multirootorid saaksid olla tõeliselt autonoomsed ja võiksid pakuda senini liialt kulukateks osutunud funktsionaalsusi. Selleks analüüsitakse erinevaid võimalusi binaarsetest lennulogidest ühtse andmestiku loomiseks. Vaadeldakse erinevaid faili tüüpe ning lõpuks valitakse *SQLite* andmebaas. Luuakse andmete mudel hoidmaks infot logidest ja kirjutatakse kood *python* keeles, mis loob nimetatud andmebaasi. Andmebaasis olevate andmete analüüsi tulemusel leitakse parameetrid, mille alusel on võimalik kogu andmebaasist välja filtreerida andmed, mille abil on võimalik luua multirootori käitumise mudel erinevatel lennu kiirustel õhu suhtes. Leitakse ka vajadus, lendude pinge graafikuid analüüsides, et saavutada autonoomsus, mis arvestab ilma tingitud muutustega autonoomsete missioonide vahel, on tarvilik inteligentne kontroller, mis teaks autopiloodile anda infot aku täituvusest. Kontroller peaks mõõtma ja arvestama aku temperatuuriga ja jälgima aku degradatsiooni ajas ning kasutus kordadel. Lisaks peaks kontroller väljastama unikaalse identifikaatori, et andmeanalüüsi teostades oleks võimalik jälgida muutusi aku parameetrites.

See töö on kirjutatud Inglise keeles, on 35 lehekülge pikk, sisaldab 4 peatükki, 16 joonist ja 8 tabelit.

List of abbreviations and terms

Abbreviation	*Meaning*
UAV	Unmanned Aerial Vehicle
GPS	Global Positioning System
GLONASS	Global Navigation Satellite System
IMU	Inertial Measurement Unit
GB	GigaByte - 1000^3 bytes
kB	kiloByte - 1000^1 bytes
MB	MegaByte - 1000^2 bytes
GCS	Ground Control Station
CSV	Comma-Sepparated Values
JSON	JavaScript Object Notation
SQL	Structured Query Language
DBMS	Database Management Systems
PTZ camera	Point-Tilt-Zoom camera
RTL	Return To Launch

Table of Contents

List of Figures

List of Tables

Introduction

Currently the autonomy of multirotor aircraft is limited to single flights. A mission can be made that traverses a given path and points its camera at points of interest. The current technology lets the pilot assign the speed of flight during the traversal of the mission. Eli Ltd has developed a technology that they call *nests*. With this technology the batteries of a multirotor can be recharged and swapped for charged ones. This creates the possibility to create missions that are automatic and require no input from a pilot. Due to the dynamic nature of weather the flight speeds of the mission need to be changed. By changing the speeds safety and higher operational ability is achieved. Current technology does not dynamically change the mission parameters to fit the weather conditions.

This thesis does not concern itself with use cases where high situational awareness is needed such as detecting objects in video, tracking objects in video, dynamically generating new missions given external stimulus. Instead use cases that use missions that follow a preplanned path are discussed. Such use cases are area surveillance missions for securing a perimeter of a secure site or autonomous patrol of a segment of a countries border.

More concretely in this thesis the library of binary flight logs of Eli Ltd is taken and used to build a database for further analysis. In the first chapter we survey the state of the art. The second chapter discusses the intricacies of piloting an unmanned aerial multirotor vehicle. In the third chapter various file formats are analyzed and chosen from to be used as storage for the database. Furthermore the analysis prerequisites are discussed and the database schema is designed accordingly. The schema is then used in the *python* code created by the thesis author to translate the binary flight logs into a *SQLite* database. In the following chapter initial analysis is conducted on the data. The goal of which is to find the relevant data to model the multirotor behavior in different conditions and requirements for the model to be used in practice.

1 State of the art

A multirotor UAV may carry different payloads for different use cases but predominantly the payload is a video or photo camera. The video and photos can be used for 3D mapping, search and rescue operations, inspecting power lines, surveillance, inspecting wind turbine structural integrity, counting deer in woods or simply capturing cinematic footage to name a few. However the act of piloting itself is secondary to the goal. The act of piloting is necessary to gather data with the payload. A pilot needs to have certain skills and training to be able to safely operate an aerial vehicle. The cost of hiring a pilot can be much higher than the operational costs of the UAV and payload combined. Thus companies across the globe look to simplify the piloting.

In movie industry camera drones are used with two people. One person to pilot the multirotor and one person to manipulate the camera. This has the overhead of hiring two people and the multirotor and the expensive payload. For the high-end movie industry this is fine. Where this overhead becomes prohibitive is in the industrial applications. To inspect power lines at least one person is needed to do the piloting and directing of the camera. The video analysis can be done during the flight or later. To keep the needs of man power to a minimum the operator needs to be able to operate the multirotor as well as the camera and know enough of the power lines to analyse the video feed. Another option is to have two people - one to pilot and film and a second one to do the analysis. The operator however might not film the correct parts for the analyst to do his job. Thus introducing the risk that the collected data is unusable. In the case of power lines not only a single pylon needs to be inspected, rather several hundred kilometers worth need to be assessed at a time. Having a single person drive to each location to fly a multirotor would take a very long time for a single person and as the person needs to be highly trained the process would become prohibitively expensive. This problem is smaller when it comes to tele-communications companies as they have single cell towers which allays need to be inspected one at a time. For the energy sector a multirotor is prohibitively expensive and not because of the device costs but rather the manpower costs.

The energy sector could use multirotors, but they have a more cost effective way of detecting faults. The method involves renting a full size helicopter with a pilot and fitting it with a expensive camera and camera operator to fly over the power lines at the right angles to later analyse the photos or video taken. This has a higher chance of missing faults but compared to multirotors is cheaper alternative.

Automating the missions to inspect power lines or cell towers requires high intelligence on the part of the autopilot and will not be assessed in this thesis. Instead search and rescue missions and surveillance missions are assessed. These missions have the potential to be simplified to the point where additional training on the part of the pilot is not needed.

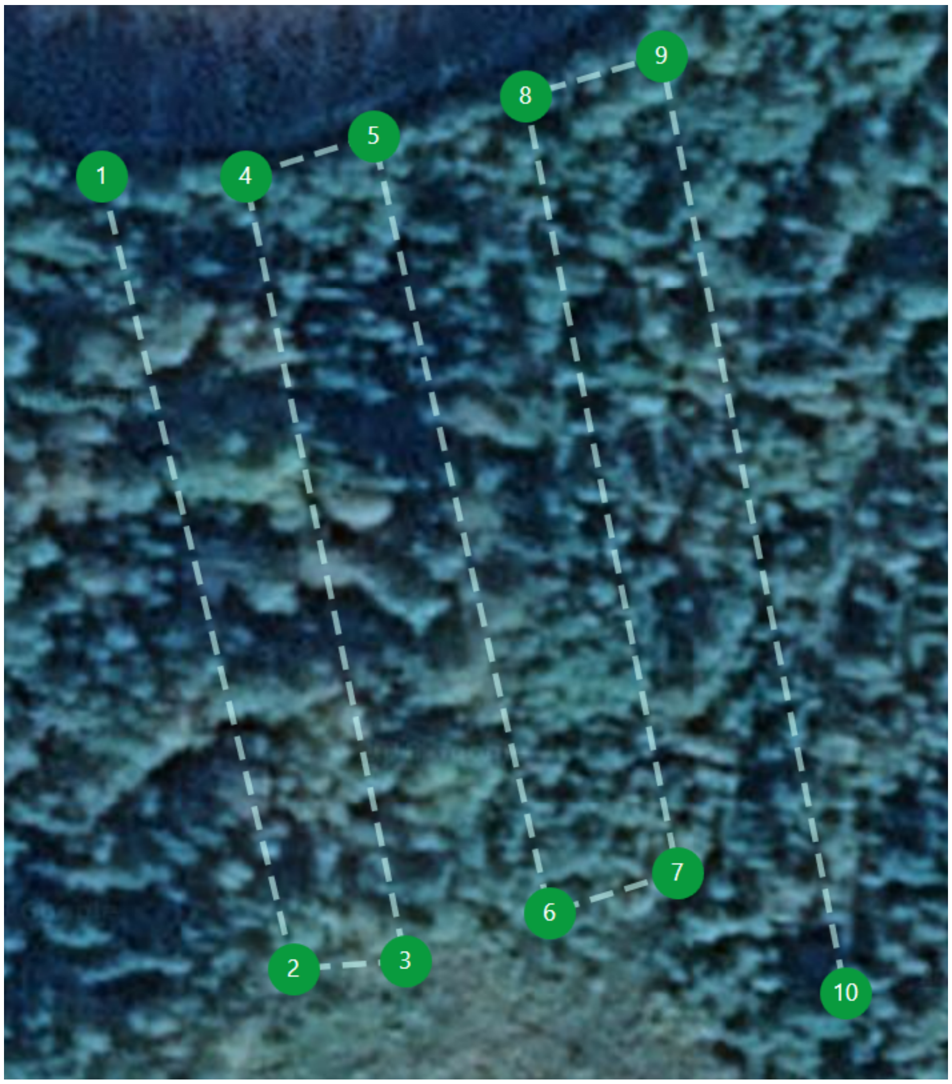

Figure 1.1: Search and rescue mission.[1]

[1]Image captured from Elix software made by Eli Ltd.

To find a person lost in woods a mission such as seen on figure 1.1 is used. To execute such a mission the pilot needs to take into account several variables such as wind speed and battery condition. A naive simplification to piloting can be made by limiting the maximum disturbance (wind) allowed by forcibly landing when such disturbance is detected. This sets a hard limit to the conditions an unskilled pilot can fly in. By enforcing such a limit the risk of failure can be minimized and the producer of the multirotor would feel a lot better about their product. A skilled pilot however would be able to fly and potentially save someones life in that situation. A better approach is needed.

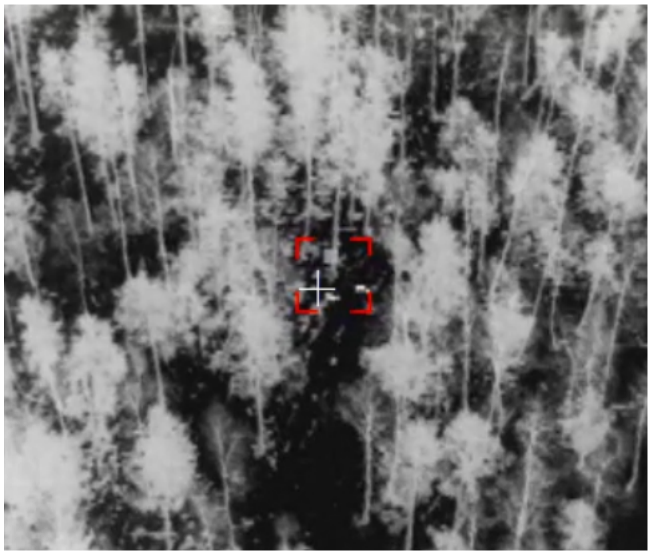
Figure 1.2: Heat signals of two deer in the woods.[2]

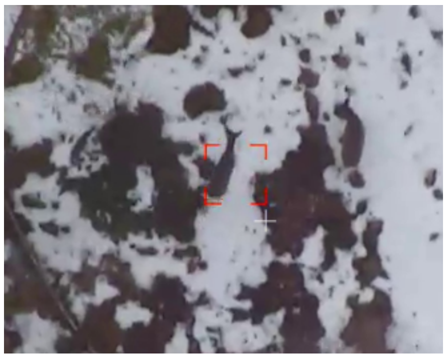
Figure 1.3: A deer in the woods.[3]

[2]Source: Eli Ltd photo repository.
[3]Source: Eli Ltd photo repository.

Area surveillance requires an automated mission that allows for specification of the camera direction and angle. A mission to survey the fences of a military compound would have to include waypoints that specify the locations the multirotor flies to and the locations or directions the multirotor camera is turned to while flying. While building such a mission takes some knowledge of the software the actual mission takes no input from a pilot (unless desired) and can be used the same way a regular camera video is used. When the viewer of the video feed spots that something is wrong, the multirotor could be set to manual control mode to be used as flying camera in a similar fashion to PTZ cameras. Such systems could be used instead of stationary cameras in cases where the areas are to big to have stationary camera infrastructure or change rapidly. Given the naive solution to dealing with different weather conditions outlined above would leave the area vulnerable every time the weather conditions exceed the piloting abilities of an untrained pilot. To remedy this the autopilot needs to be able to adjust to the weather conditions in the same way an experienced pilot would.

2 Piloting a multirotor UAV

During the flight of a multirotor UAV several things happen at once. Firstly the UAV fights against gravity to keep itself at the desired height. Secondly it has to follow the orders of the pilot - fly as desired. Thirdly the UAV also fights against the wind. This combination of three forces means that the system is highly dynamic. While the gravity is constant, the desires of the pilot change as well as the wind.

The pilot can be in control largely by two ways:

1. Manual control via analog controllers
2. Automatic flight mode via commands

During manual mode the UAV is directed by analog controllers that give it some sort of continuous input signal to adjust the throttle of the four motors. Changing the throttle results in either tilting of the aircraft or changing of its altitude or both. Different vendors offer differing levels of control. In *arducopter*[1] firmware there is *stabilize* mode that allows the pilot to fly manually, but the platform self-levels the roll and pitch axis. If the pilot releases the controls the UAV falls to the ground. In *DJI* Phantom 4[2] there are modes such as *Position Mode* which uses GPS and GLONASS satellite positioning where releasing the controls results in the platform remaining stationary in air even in windy conditions. A similar mode exists in arducopter called *loiter*. In this some simplification of piloting can be seen. The autopilot is there to remove a component of skills needed for successful piloting. In these modes the pilot is in control of the speed of flight and is in effect compensating for the effects of wind when flying in some direction. While staying still the GPS location is used to stay still.

Both *arducopter* and *DJI* have automatic flight modes where the multirotor flies according to a pre-programmed mission. The missions are user specified by global x/y coordinates, height values and the speed of flight between points. The multirotor

[1]A full-featured, open-source multicopter UAV controller firmware.

[2]A Chinese producer of multirotor UAV's. Phantom 4 is a successful commercially available multirotor platform.

autonomously attempts to fly through given mission points at the given height and speed. During such a flight the multirotor automatically compensates for wind. In the case of *DJI* a wind warning is given at 6 m/s winds and high wind warning at 9 m/s. *Arducopter* does not report the wind and leaves everything to the pilot.

The complications of piloting due to wind are numerous and we will look at a few of them.

1. Angles of the multirotor when stationary (*loiter* mode) reflect wind direction.
2. Current draw from the battery increases.
3. Vibrations aboard the multirotor increase.

When the aircraft is stationary above ground the wind causes it to tilt into the wind. In the reference frame of air, the multirotor is flying at the speed of the wind and in opposite direction, to, in the reference frame of the ground, remain stationary. Looking at the angles there is no difference between standing still in 5 m/s wind and flying in some direction with the speed of 5 m/s in windless weather.

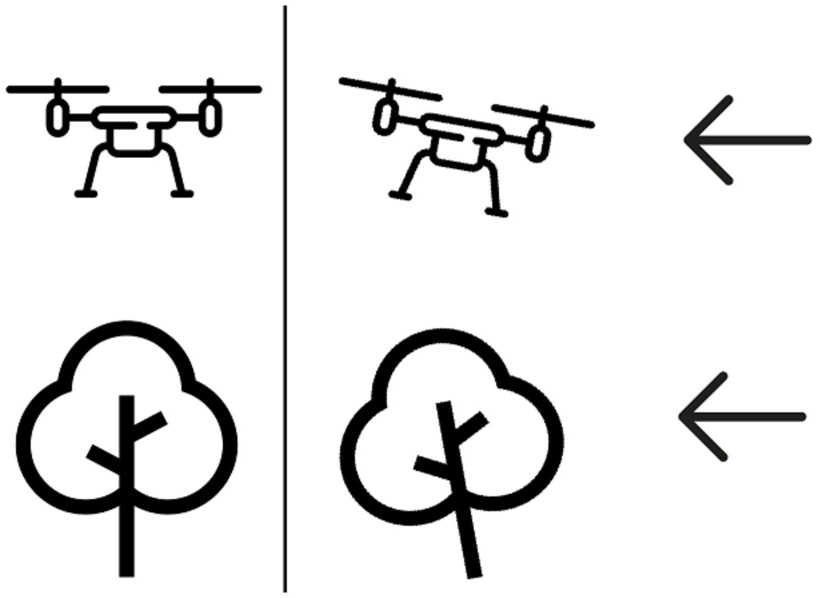

Figure 2.1: Stationary multirotor without and with wind.[3]

Secondly, when flying against the wind, the current draw increases to match the increased power required to stay airborne. From the power consumption point of view there is no difference in flying 5 m/s in some direction to standing still in 5 m/s wind.

[3]Arrow icon made by `https://www.flaticon.com/authors/lyolya`, multirotor and tree icon made by `https://www.flaticon.com/authors/freepik` from www.flaticon.com

Thirdly the vibrations aboard the aircraft increase. This can cause trouble for the flight controller as the output of the IMU's becomes noisy. Since the flight controller calculates the outputs of the motors using various sensors and among them IMU's, high vibration may cause the flight controller to be unable to determine its attitude and as a result cause the aircraft to loose control and crash. In this thesis we will look mainly at the first two effects.

To operate a multirotor manually the pilot needs to take those effects into account and keep their eyes on the readings of those parameters. The simplified flight modes such as *loiter* take most of the skill needed to pilot by implementing features on the autopilot. Several control algorithms have been developed, such as the widely used *proportional integral derivative* algorithm, *cascaded linear proportional integral derivative*[4] algorithm or the newer *incremental nonlinear dynamic inversion*[5] algorithm. These algorithms optimize for the stability of the flight reducing the need for the pilot to do it by hand but higher level algorithms are still at the level of naive implementations for features such as optimizing for flight time or distance or speed, optimizing for safe return to start. These features are of critical importance for achieving full autonomy in some scenarios.

For safety both *arducopter* and *DJI* use fail-safe modes to guarantee that the multirotor has enough power left to reach the take-off location. A naive approach that *arducopter* employs is to look at the battery voltage and at a fixed point start flying back. This safety behavior can be triggered in both manual control mode and automatic mission mode. This approach has to take into account the worst case scenario of flying against the wind a long distance to make sure the multirotor makes it back. This leave a portion of the available power in the battery as a buffer reserve. A smarter fail-safe would also take into account the distance from the starting location to scale the value where the fail-safe is triggered. This helps reduce the buffer energy requirement. To further reduce the energy left in the buffer the fail-safe would also need to adjust for the weather conditions. If the starting location is down wind then much less energy is needed than when the return home trip is taken against the wind. To achieve this type of smart fail-safe functionality the autopilot needs to be aware of two things:

1. The wind parameters - strength and direction.

2. The battery state - how much energy is left in the battery.

[4]Yu, Yang, Wang, Li, & Li (2015)
[5]Smeur, Croon, & Chu (2018)

To find the wind parameters a fusion of an on-board wind sensor and magnetometer could be used. An on board wind sensor has been used to measure winds in wind farms to model airflow[6]. The multirotors that Eli Ltd produces do not have on-board wind sensors. As there is no difference between a multirotor flying at a fixed speed in windless conditions or staying stationary when the wind is equal to that of the previous example. Therefore a model of behavior could be created given flight data where wind is minimal or zero. In the next chapter creating a database to use in modelling is discussed and the code to do so is created.

Given a database of flight logs the behavior of batteries can be analyzed. This is done later in this thesis.

[6]Palomaki, Rose, Bossche, Sherman, & Wekker (2017)

3 Database generation

3.1 Inital data

Eli Ltd collects the binary flight logs that are saved by arducopter firmware aboard the multirotor platform. The binary files contain *MAVLink* protocol packets. *MAVLink* stands for *Micro Air Vehicle Message Marshalling Library.*

> *MAVLink is a very lightweight, header-only message library for communication between drones and/or ground control stations. It consists primarily of message-set specifications for different systems ("dialects") defined in XML files, and Python tools that convert these into appropriate source code for supported languages.*[1]

Not all possible messages that the autopilot generates are sent to the GCS. It is configurable which packets are saved, which are sent to the GCS and which are dropped. In general the GCS receives some subset of all messages that is important for operating the multirotor. In the logs a bigger subset is recorded. For the logs present most important messages and communications are saved but extremely high resolution data is not saved. As such over a thousand logs have been collected. The total number of binary logs available is 1363 and amount to 57.2 GB of data. This would average 41966 KB per log. Among them are logs that are not actual flights but rather tests on the bench. These logs inflate the number of logs available. Some logs do contain the highest amounts of data and can be several hundred MB large and thus inflate the average. A normal flight can be expected to be between 4 to 100 MB's. An average 40+ minute flight is expected to be over 40 MB.

[1]ArduCopter team (n.d.-b)

3.2 Choosing the output data type

In this thesis *R*[2] programming language is used for data analysis. *R* is a language and environment for statistical computing and graphics. As it is open source no licences are required to set it up and use. This brings statistical computing and data science to the hands of everyone interested in the topic and avoids expensive mathematics suites.

The binary data in the log files is not readily accessible from *R*. As a result it is necessary to convert the data to some other format. The *pymavlink* project - a *python* implementation of MAVLink protocol[3] - has a few tools that do just that. First of the options is to convert the log into a CSV[4] file. *Mavlogdump.py*[5] is the tool available for conversion. With CSV files comes a caveat - you may only export one message type at a time. This is to be expected since the the CSV files separate table columns with commas and the log files have message types with different lengths of columns. Example lines of CSV file taken from a flight log containing NTUN packets.

```
timestamp,TimeUS,DPosX,DPosY,PosX,PosY,DVelX,DVelY,VelX,VelY,DAccX,...
1521817093.27547407,688646373,-489.633148193,86.0853424072...
1521817093.37760210,688748501,-489.633148193,86.0853424072...
1521817093.47829199,688849191,-489.633148193,86.0853424072...
```

In *R* it is very simple to import CSV files and this at first seemed a viable candidate file type for storage. However since there are many different types of messages, each log would in turn be converted to as many CSV files. Potentially leaving us with tens of thousands of files. This means *R* would have to import the relevant files one by one and processing them. With thousands of files to import the analysis time would be negatively impacted as the data import poses an overhead.

Another option is to use JSON[6]. JSON is a data-interchange format designed to be easily read and written by humans as well as to be easily parsed and generated by machines. There are two structures in JSON:

- A collection of name/value pairs.
- An ordered list of values.

[2]R Development Core Team (2004)
[3]ArduPilot team (n.d.-b)
[4]Comma separated values file is a text file where a comma is used to separate values.
[5]ArduPilot team (n.d.-a)
[6]Crockford (n.d.)

```
},
```

Both are familiar to programmers using the *C*-family languages. Both can be seen in the following example:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Two consecutive entries from the binary log converted to json format using *mavlog-dump.py* look like this:

```
{"meta": {
    "timestamp": 1521817579.039407,
    "type": "PIDP"
  },
  "data": {
    "D": -0.0005177092389203608,
    "TimeUS": 1174410306,
    "I": 0.0324220173060894,
    "AFF": 0.0,
    "Des": -0.012198338285088539,
    "P": -0.005622388795018196,
    "FF": -0.0
  }
}
{"meta": {
    "timestamp": 1521817579.039419,
    "type": "PIDY"
  },
  "data": {
    "D": -0.0,
    "TimeUS": 1174410318,
    "I": -0.08129391074180603,
    "AFF": 0.0,
    "Des": 0.006200382951647043,
    "P": 0.008107485249638557,
    "FF": 0.0
  }
}
```

R has libraries to deal with JSON data. Such as *jsonlite*[7] and *rjson*[8]. As with CSV, JSON files need to be imported one by one, analyzed and unloaded. This has considerable overhead. Further more an output file from a 47.6 MB log file becomes 649.8 MB. That is an increase of size of thirteen times! The library of logs that is 57.2 GB becomes 743.6 GB of JSON data as a result. At least there are an order of magnitude lower amount of files.

The final option to consider is *SQLite*[9]. *SQLite* is a *SQL* database engine that is self-contained and serverless. It is designed to be *SQL* database, but without concerning itself with a server process and user management and other common *SQL* attributes. As such its designed to be a competitor to both JSON and CSV. The benefit of *SQLite* is that it can be used by various third party programs and programming languages. *Python* and *Tcl* have *SQLite* built in. Raw data can be imported from CSV files and the data can be compressed to similar sizes to *Zip* files. Since the database is a single file it can easily be written to a USB memory stick or with smaller databases emailed to a colleague directly. Given some understanding of *SQL* *SQLite* offers an easy to use database.

R language has adapters to connect to *SQL* databases such as *RODBC*[10], *RJDBC*[11], *bigrquery*[12] and many others among which is *RSQLite*[13]. *RSQLite* embeds the *SQLite* database engine in *R*, providing a *DBI*[14]-compliant interface. *DBI* is an *R* package that defines a common interface between *R* and database management systems (*DBMS*).

[7] Ooms (2014)
[8] Couture-Beil (2014)
[9] (“About sqlite,” n.d.)
[10] Ripley & Lapsley (2017)
[11] Urbanek (2018)
[12] Wickham (2018)
[13] Müller, Wickham, James, & Falcon (2017)
[14] R Special Interest Group on Databases (R-SIG-DB), Wickham, & Müller (2018)

With *RSQLite* package it is possible to directly manipulate data in *SQLite* database.

3.3 Tidy data

Tidy data is a set of principles that help organize data in data sets. This helps make data cleaning easier and faster by not having to start from scratch every time. Another benefit of tidy data is that it is easier to design data analysis tools which can assume that the data input is always tidy. Both benefits help the data analyser to focus on the underlying problem rather than managing data half the time[15].

Table 3.1: Tidy data.

	mpg	cyl	disp	hp	drat	wt	qsec	vs
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0

	am	gear	carb
Mazda RX4	1	4	4
Mazda RX4 Wag	1	4	4
Datsun 710	1	4	1
Hornet 4 Drive	0	3	1
Hornet Sportabout	0	3	2

Here we are looking at the *mtcars* data set that is inbuilt in *R*. In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

This data set is already tidy. Each row is an *observation* and each column represents a *variable*. Every element in the table is a *value*. Any other arrangement of data is

[15]Wickham (2014)

considered *messy*. By having tidy data it is easy to manipulate data such as group by column info and tie break on another column. Tidy data is particularly well suited for vectored programming languages like R where each observation of each variable is always paired.

The five most common problems with messy data sets are:

1. Column headers are values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables.

Table 3.3: Column headers as variables.

religion	`<$10k`	`$10-20k`	`$20-30k`	`$30-40k`	`$40-50k`	`$50-75k`
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don‘t know	15	14	15	11	10	35
Evangelical	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically	228	244	236	238	197	223
Jehovah's	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Here we can see a table of data where the column header - the variable itself is a value. This can help make very dense and informative tables but for working with the data it is not tidy. If we wish to separate the data into segments of 5000 dollars then we can say that this table also has multiple variables in a single column. Rest of the problems we will not look at.

3.4 Database normalization

ArduPilot[16] development team hosts an organization of projects on *GitHub*[17] that contains several useful projects such as the *ardupilot* autopilot firmware codebase for multirotors, planes, rovers and much more. Among the project is *pymavlink* project which provides a *python MAVLink* interface and utilities. Among those utilities are tools mentioned in chapter Choosing the output data type. With these tools we determined that none of them fit our needs exactly but those tools can be used as a source of information for building our own tools.

The data storage format was decided to be *SQLite* database. To increase the efficiency of working with the data the data set should be tidy. By taking that into account when designing the *SQLite* database, time can be saved by not having to separately tidy up the data set before analysis. Further more Edgar F. Codd[18] defined *normal forms* to permit querying and manipulation by a universal data language[19]. The third normal form (3NF)[20] is considered as being tidy data. Simply by designing the database using the normal forms, especially the third normal form allows us to reach tidy data.

By instructing *mavlogdump.py* to dump the logs from binary form to a readable text file we can see the general shape of the data in the binary logs.

```
mavlogdump.py log.BIN > dump.txt
```

Table 3.4: Excerpt from *dump.txt*.

Time	Message	Data
2018-03-23 17:26:57.12:	ATT	Inner Table
2018-03-23 17:26:57.12:	RATE	Inner Table
2018-03-23 17:26:57.12:	PIDR	Inner Table

Table 3.5: Message ATT.

Message	Data
TimeUS	2412496394

[16] ArduCopter team (n.d.-a)
[17] An online platform for version controlling software development
[18] ("Edgar f. codd," 2018)
[19] One such language is *SQL*
[20] ("Third normal form," 2018)

Message	Data
DesRoll	-2.41
Roll	-2.26
DesPitch	2.64
Pitch	2.8
DesYaw	96.07
Yaw	96.28
ErrRP	0.01
ErrYaw	0.02

Table 3.6: Message RATE.

Message	Data
TimeUS	2412496408
RDes	-0.977470874786
R	0.30203345418
ROut	-0.0680121853948
PDes	-1.36304438114
P	-1.30311119556
POut	0.0843362286687
YDes	-1.62430250645
Y	-0.70737850666
YOut	-0.1534512043
ADes	1.88155674934
A	-2.99711227417
AOut	0.368027120829

Table 3.7: Message PIDR.

Message	Data
TimeUS	2412496431
Des	-0.0153276510537
P	-0.00370784359984
I	-0.0374843552709

Message	Data
D	-0.0268199834973
FF	-0.0
AFF	0.0

First normal form is a property of a relation where each attribute of the domain contains only indivisible values and the value of each attribute contains only a single value from that domain[21]. In table 3.4 we see that there are inner tables in the message. This is in violation of the requirement of atomic values. The inner tables need to be removed from the message and separated into another table.

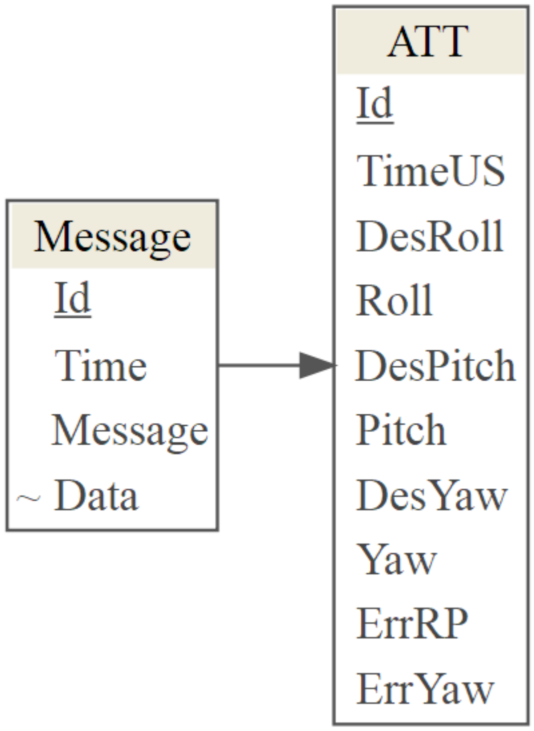

Figure 3.1: Message and ATT relationship.

Figure 3.1 has two tables where *Message* table represents all the different messages and table *ATT* represents any single *data* table of a message - here the *ATT* message. From tables 3.5, 3.6 and 3.7 we can see that the inner tables are of different shape. This means that each message requires a separate table. The current *Message* structure does not permit more than one *data* table to be used.

[21](“First normal form,” 2018)

Another option would be to unwrap the inner *data* table to be a part of the outer *Message* table. Here we run into a similar problem. Since the inner *data* tables are of different shape, the single *Message* table would have to contain each possible variable in the logs and be empty most of the time. This table is realizable in a database.

Table 3.8: *Message* table.

Id	Time	Message	TimeUS	DesRoll	Roll	..	RDes	R	..	Des	P
123	time	ATT	value	value	value	..	NA	NA	..	NA	NA
124	time	RATE	value	NA	NA	..	value	value	..	NA	NA
125	time	PIDR	value	NA	NA	..	NA	NA	..	value	value

This table contains mostly *NA* values as for every message only a subset of variables is relevant. When working with this table, the *NA* values need to be removed. We can solve this problem by converting this table from wide form to long form. Since we are storing several flights into the same database we need to add a flight identificator.

Table 3.9: Long table.

Id	Flight	Message	Timestamp	Parameter	Value
125	4	ATT	2018-03-23 17:26:57.12	TimeUS	2412496394
126	4	ATT	2018-03-23 17:26:57.12	DesRoll	-2.41
127	4	ATT	2018-03-23 17:26:57.12	Roll	-2.26
...	...	...	...	...	...
134	4	RATE	2018-03-23 17:26:57.12	TimeUS	2412496408
135	4	RATE	2018-03-23 17:26:57.12	RDes	-0.977470874786
136	4	RATE	2018-03-23 17:26:57.12	R	0.30203345418
...	...	...	...	...	...
145	4	PIDR	2018-03-23 17:26:57.12	TimeUS	2412496431
146	4	PIDR	2018-03-23 17:26:57.12	Des	-0.0153276510537
147	4	PIDR	2018-03-23 17:26:57.12	P	-0.00370784359984

In table 3.9 we added flight identificator but we would like to have more information on the flights. To reduce metadata repetition another table is needed. There is also quite a lot of repetition in other values in the table. The message type and timestamp repeat for each parameter in the inner table. The parameters also repeat each time the message is repeated. Each of the repeating elements can be moved to a dedicated table.

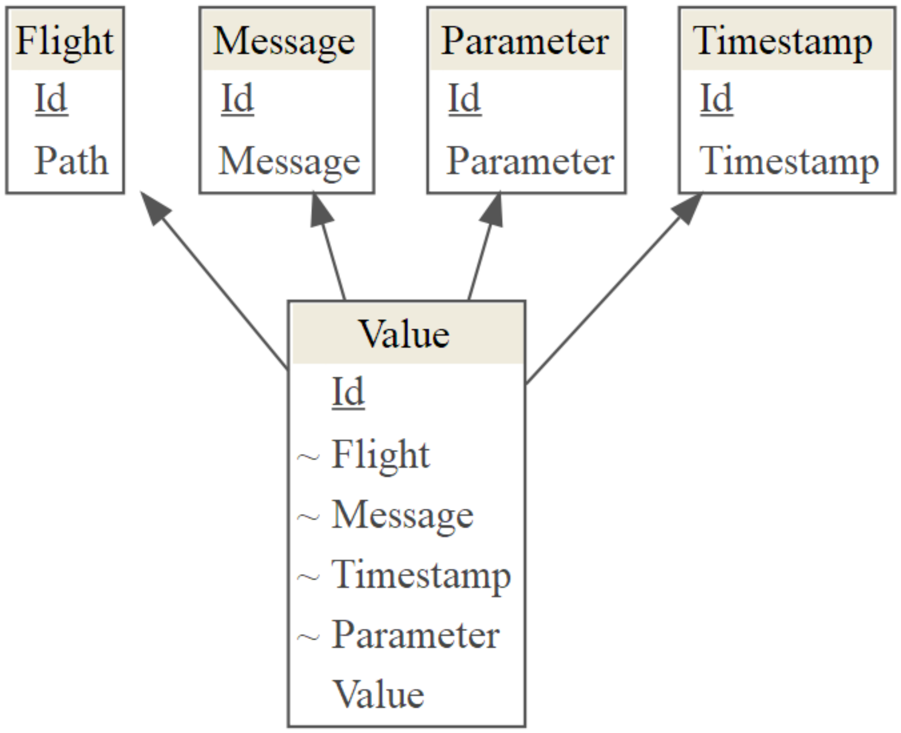

Figure 3.2: Final database schema.

3.5 The code

The code starts off with taking as an input the folder in which the binary files is stored in. The folder is searched recursively so that sub folders containing log files are also searched through. The files are counted and the sizes saved. A filtering is done by file size. Files larger than 100 MB and smaller than 4 MB are skipped. After the first log file is processed the time taken is used to calculate an estimation as to how long the whole process will take. Further more the total time elapsed is also displayed.

Mavutil.py from *pymavlink* project is used for translating the binary data to in memory representation. From there the data is buffered and written to the *SQLite* database. Each new message type adds its parameter types to *Parameter* table as seen in figure 3.2 and each new message type is added to the messages table. This allows for the specification for messages to change and several versions of the specification to be used in different logs. The message name and the parameter names are decoupled from each other. An assumption that the message specification does not change in a single log is made.

After processing a log a new entry to the *Flight* table is made. The metadata that is added is the path where the file was stored. As the logs are separated into folders by which multirotor it was obtained from or from which period in time or which version of the multirotor was used then the path gives some sort of information that could be used for filtering in the analysis stage.

Timestamps are allowed to repeat as searching through the whole table means more processing needed while creating the database. This leaves a possibility to create a secondary script that would look through the *Timestamp* table and removing duplicates and substituting the *Id* in the *Value* table.

There is an overhead to writing into the *SQLite* database. A single write entails opening the database connection, writing to the database and closing the connection. The opening and closing operation adds an overhead that over hundreds of thousands of writes becomes significant. To remedy this a bulk insert operation is available. Every 10 000 lines of the binary log a bulk write is done. That number is experimental in nature as in testing making the number bigger did not seem to save any processing time. Making it smaller however slows the processing down. There is a possibility that further optimization can be done.

In case where the script crashes or some error happens or new logs are to be added to the database the code first checks the existence of the database. If the database exists the supporting tables are read into memory to be tested against so that new messages and parameters get added to the database.

While *SQLite* is a good option for an on disk storage of the database, other databases are available and have some useful features. One of which is the ability to use window functions on the database. This means that it is possible to use filter kernels on the data while still in database. Such functions using *SQLite* require loading the relevant segment of data (such as a single flight) into *R*. An attempt was made to port the working *python* code to instead of *SQLite* to use *PostgreSQL*[22]. The code does work but is orders of magnitude slower. Heavy optimization is needed to streamline the process and as *SQLite* interface is simpler in nature this idea was left as something to be done later given more time. Moving to a full featured relational database management system would allow easier porting of the analysis to a web based service later, but that is not relevant in the context of this thesis.

The code for creating the database is written in *python* and is added to the appendix.

[22]source (n.d.)

4 Initial analysis of the data

As mentioned in the second chapter for the multirotor there is no difference whether it is flying at a given direction at a set speed and staying stationary in winds opposite to the flying direction of the previous example and with the same speed as the multirotor was flying. The following equation can be written:

$$\vec{w} = \vec{v} + \vec{u}$$

where $\vec{w}$ is the ground speed of the multirotor, $\vec{v}$ is the wind speed of the multirotor and $\vec{u}$ is the wind speed. If we increase the wind speed $\vec{u}$ the multirotor autopilot finds the angles and motor power that allow it to fulfill the desired ground speed $\vec{w}$. When $\vec{w}$ stays constant and $\vec{u}$ increases $\vec{v}$ changes to account for the change in wind speed. The change in $\vec{v}$ is dependent on the flight direction and wind direction. When flying against the wind $\vec{w} = \vec{v} - \vec{u}$ as the multirotor has to compensate for the increased wind speed. When flying down wind the speeds add up - the multirotor has to do less work to fly at the desired speed $\vec{w}$. To find the wind speed a model of the multirotor behavior is needed. Another way of looking at it is to instead look for wind speed of the multirotor. This itself is the model of the multirotor flight. The user gives the desired ground speed $\vec{w}$, the wind is $\vec{u}$ and what we are after is the resultant behavior $\vec{v}$. To find the flight model $\vec{v}$ we can take the aforementioned equation and substitute the wind speed $\vec{u}$ with 0.

$$\vec{w} = \vec{v}$$

This way the desired speed equals the wind speed. With this in mind it is necessary to extract from the database the data of the flights where wind speed is close to zero to create the model. The chapter Extracting windless flight data does just that.

Another important factor to consider is the state of the battery. To accurately model the multirotor behavior in diverse flight conditions the information about how much energy is left is tantamount. In the chapter Analysis of battery performance the data available is analyzed.

4.1 Extracting windless flight data

To find the flights where the wind is minimal we could look at the levels of vibration aboard the multirotor but that is not the easiest way nor is it very accurate due to its non-linear nature. Further more the vibration levels stay relatively low for normal flight conditions. As such it is more useful for estimating the upper limits of air speed the multirotor is able to achieve.

A better option is to note that when wind speed is zero the multirotor moves at the desired ground speed. If the desired ground speed is also zero then the behavior of the multirotor should be stable as well. Without wind the angles of the multirotor should fluctuate around zero degrees and by looking at the average angles when the multirotor is commanded to hold its position in *loiter* or *position hold* mode we can detect the logs that have nearly no wind. There will be some fluctuation in the angles due to GPS inaccuracy and barometric drift and drift of the IMU's and other sensors. This should amount to white noise and average out given time. By taking the average angles over the time of remaining stationary we will get the average angle and direction of the wind as the multirotor will effectively be flying to counteract the wind. Any sufficiently large angle constitutes wind. For training the model only the windless flights are needed, but since some wind is expected the level of cutoff where a flight is considered to have been in windless condition. Various levels of "windless" data could be used as different cutoff values result in different amounts of training data.

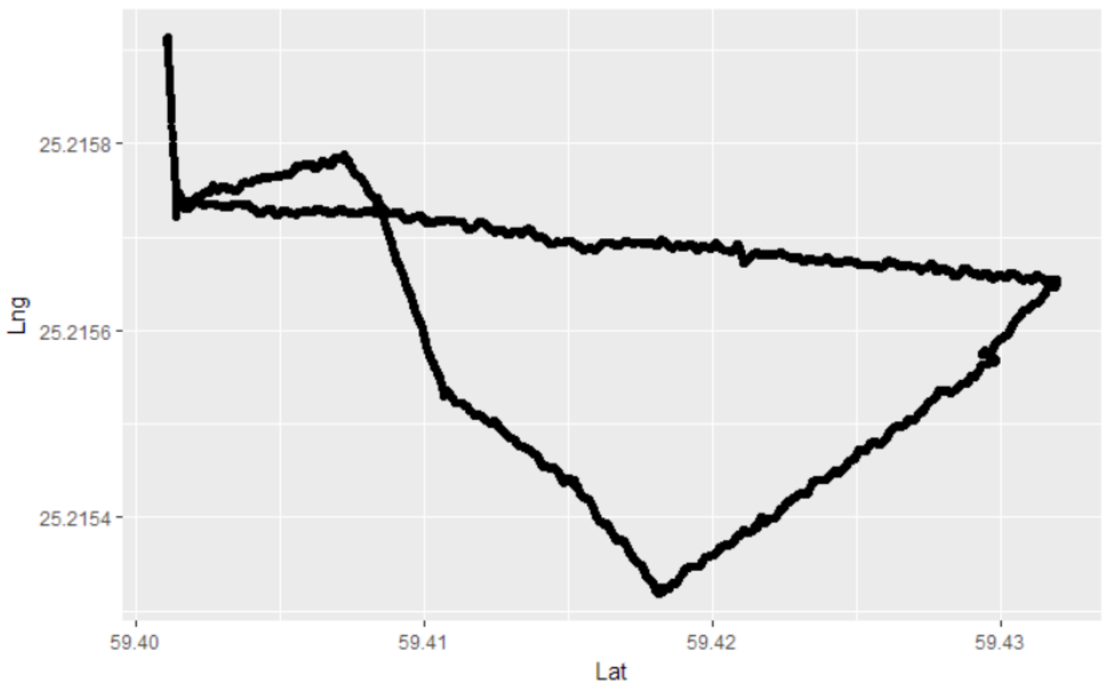

Figure 4.1: Latitude and longitude data points of a flight.

In figure 4.1 we can see a flight that consists of segments of noisy data. This flight was mostly flown in manual *loiter* mode and as a result is this noisy.

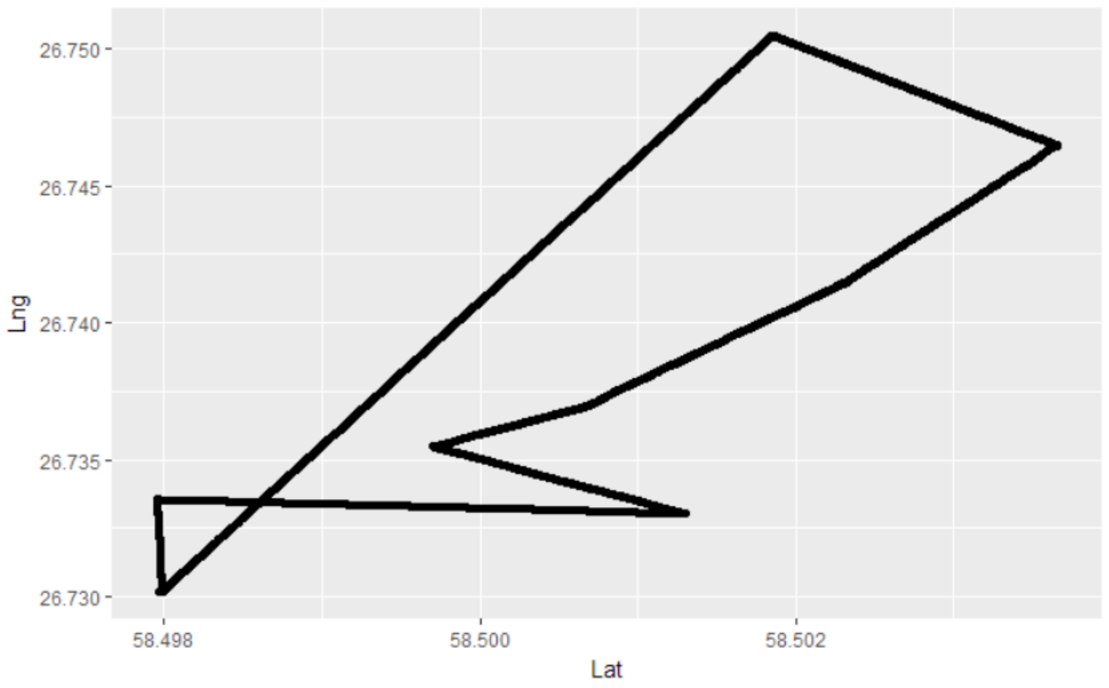

Figure 4.2: Latitude and longitude data points of a flight.

In figure 4.2 a flight using *auto* mode is used. In this mode the autopilot does its best to directly fly to the points specified in the mission. While this information is useful for getting an idea of the flight trajectory it does not help us with finding windless flights as the information we are looking for is missing. Namely the sections that the multirotor is stationary. Since the graph does not contain any data about time, only individual points, we are unable to see where the multirotor stands still. A guess would be that at the turning points of the straight segments the multirotor could potentially stand still.

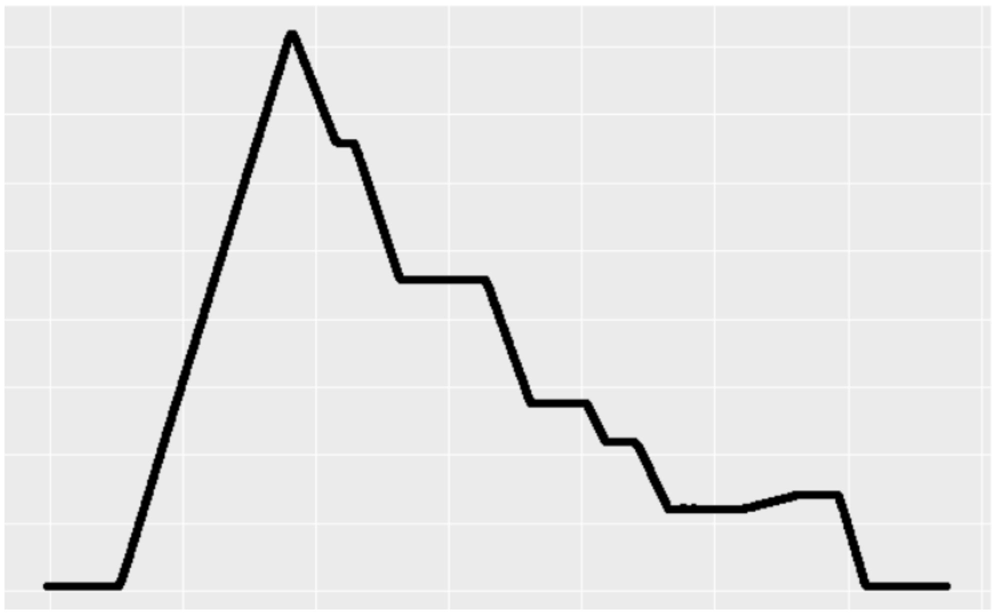

Figure 4.3: Longitude data points in timeseries of a flight.

Looking at the longitude time-series graph of the same flight on figure 4.3 we can clearly see where the multirotor stands still in the longitude axis. However this is not sufficient to tell where the multirotor is truly stationary as motion could be had on the latitude axis.

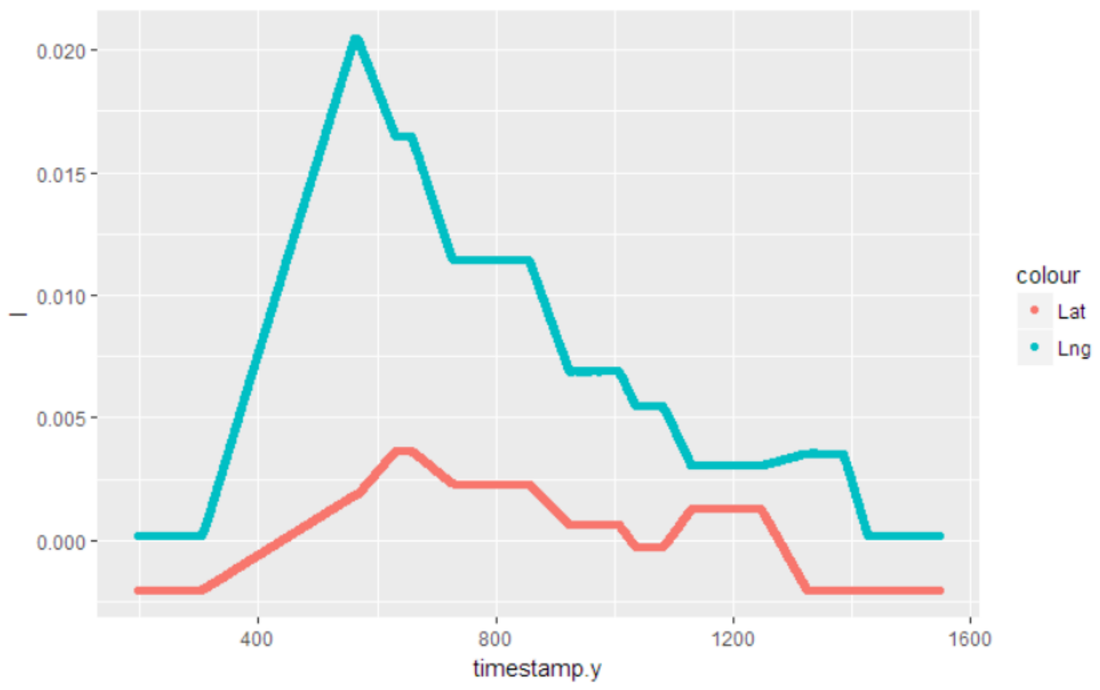

Figure 4.4: Longitude and latitude on the same timeseries graph of a flight.

Figure 4.4 has time on the x axis and latitude/longitude point value changes on the y axis. The figure is made by taking both latitude and longitude axes and merging them into a single one. Then the data is shifted to very nearly zero so that both graphs can be seen. This helps us see where the multirotor is actually stationary. Where the value of both latitude and longitude stays unchanged for a period of time the multirotor is stationary. Graphically we can see where this is so but as a function of data we still need an additional function to find the segments where the multirotor stays still. To find the segments various approaches could be attempted. One of them would be to construct lines through points in the time-series and looking for where the slope of the line changes compared to the last one. Filtering would be needed to account for the noise present. After that line segments where the graph moves perpendicular to either latitude/longitude axis are taken into consideration. From there latitude and longitude segments have to be compared to find the segments where both are perpendicular to the axis. This is to remove segments where motion is recorded as perpendicular to either axis. This can be seen on the end of the latitude longitude graphs on figure 4.4. There we can see motion on the longitude graph but not on the latitude graph. Where both graphs are parallel to time the multirotor is truly stationary. However the aforementioned approach is not the best on as it ignores corner cases such as when the multirotor has turned to face the direction of future motion and taken some angle to start moving but from inertia has not yet started moving. Luckily we have more parameters than latitude and longitude coordinates in the logs. We also have information about what mode the multirotor was in.

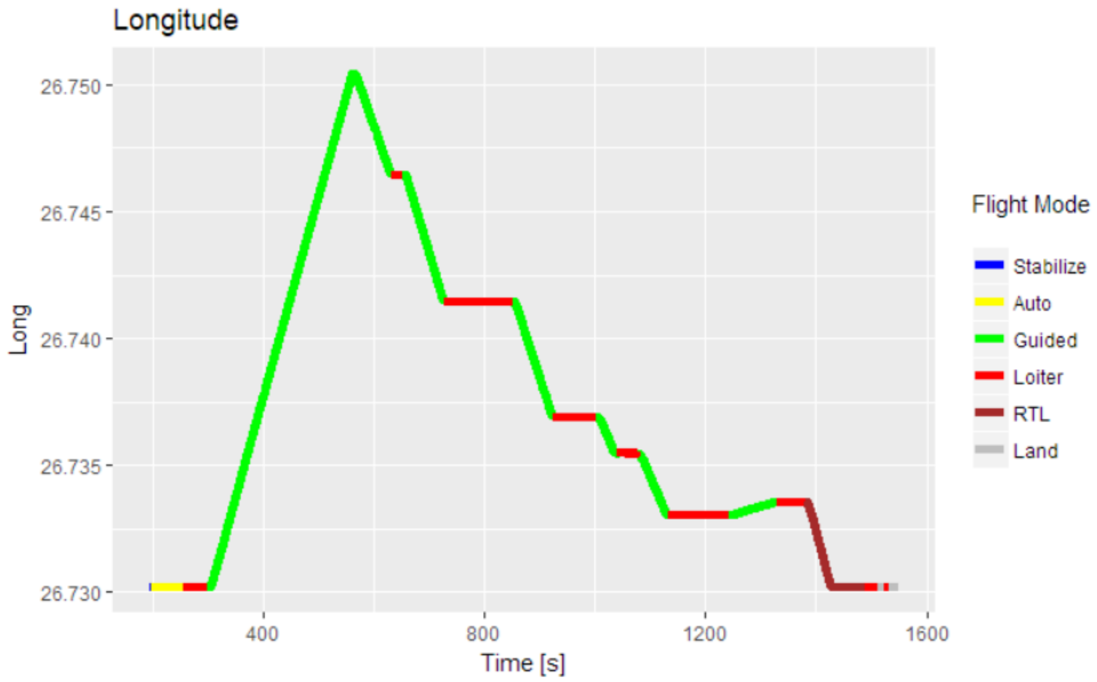

Figure 4.5: Longitude data points in timeseries of a flight augmented with mode information.

On figure 4.5 the longitude time-series graph is augmented with information about which mode the multirotor was in during the flight. The take off sequence starts with brief entry to *stabilize* mode which we will ignore. After that *auto* mode with the setting to stay still until a non-empty mission is uploaded. Soon after *loiter* mode is entered. From figure 4.4 we can see that the multirotor remained stationary for the duration of *loiter* mode. After entering *guided* mode, which is also an automatic mission mode, the multirotor starts moving. In this flight each time the multirotor is stationary the mode is *loiter*, excepting take off sequence and landing procedure. In the landing procedure the *RTL* mode returns the multirotor to the launch location and lands. During the landing the multirotor remains stationary. Before fully landing the *RTL* mode is interrupted by the pilot by issuing *loiter* command. We can assume that here we had a skilled pilot who wanted to see how long the battery would last until being truly empty and thus estimating the overall health of the battery. After that an automatic landing fail-safe is executed by the autopilot only to be interrupted again with the *loiter* command by the pilot. From this graph we can see that to find the windless flights we need to find all flight segments where:

- The mode is *loiter* and both latitude and longitude coordinates are stationary.
- The segment after launch where the initial mission is empty and both latitude and longitude coordinates are stationary.

We could also use mode *land* to check for wind conditions but it needs to be taken into account that the latitude is changing and *land* and *loiter* are not comparable. Also in *land* mode the multirotor can be manipulated manually so latitude and longitude coordinates need to be checked for changes. *RTL* mode should be split in two parts since it internally contains both *guided* segment and *land* segment. For the *land* part in *RTL* same considerations need to be made.

Once the segments have been filtered out the average angle of the multirotor needs to be calculated. Since the multirotor operates in three dimensional space there are three angle parameters. As the vehicle is capable of moving in any direction the orientation of the front is not important for our purposes. The parameter for that is yaw. Instead we average over the pitch and roll parameters.

The previous analysis is sufficient to filter help filter out relevant flights from the database. From there the relevant data may be separated into training and verification data to test the model. Further decimation of data is needed as the data is collected at a high frequency. The desired directions of flight need to be found from the data as well as the desired flight speed. The exact details for the model creation are left for the model creator.

After the model is created the direction of the wind needs to be calculated. The direction of wind helps us optimize our battery use in the case of automated missions. Flying against the wind takes more power than flying by the wind. As we mentioned $w = v + u$ which means that the angle difference between w and u have to be compensated by v. Since all elements in the equation are vectors where the magnitude is the speed of motion and the angle is the angle. By subtracting from the desired ground speed w the model of the air speed v we get the calculated wind speed u. From here we can extract the wind direction. The calculated wind speed can be compared to real measurements to assess the accuracy of the model. An experiment may be conducted by flying to multirotors with 10 to 100 meters apart from each other where one of the multirotors is the multirotor from which the model is built from. The other multirotor should carry an accurate wind measurement device such as the one mentioned in *Wind Estimation in the Lower Atmosphere Using Multirotor*

Aircraft[1]. Both multirotors should fly at the same height and at the same time to reduce variance of measurements in time. The missions should be identical except for the spatial displacement to avoid unwanted collisions.

4.2 Analysis of battery performance

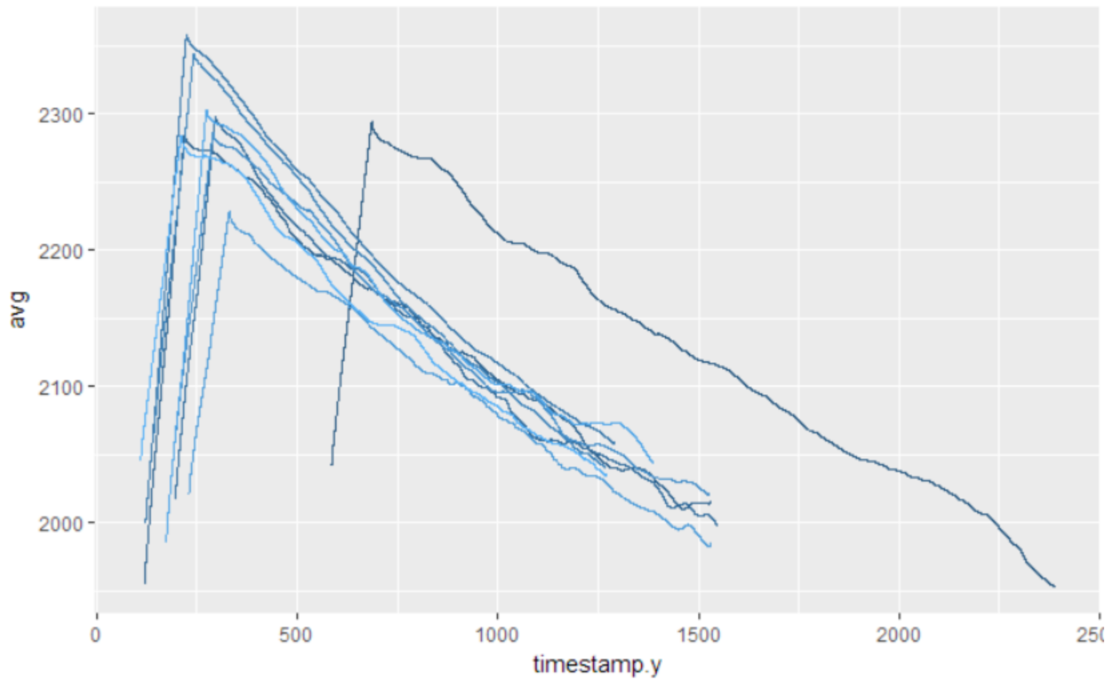

Figure 4.6: Low-pass filtered battery voltage graphs.

An figure 4.6 we have 9 heavily filtered graphs of voltage during 9 flights. The smoothing is done by applying a moving average filter with width of 1000 values to smooth the otherwise noisy graphs. Here we can se a couple of problems. Firstly the beginnings are distorted until the filter averages up to a more realistic value. These parts need to be removed. Second problem is that since the logs start before launch command the graphs can be shifted in time and thus not align. This can be counter acted to shifting all the graphs by the first voltage measurement value.

[1]Palomaki et al. (2017)

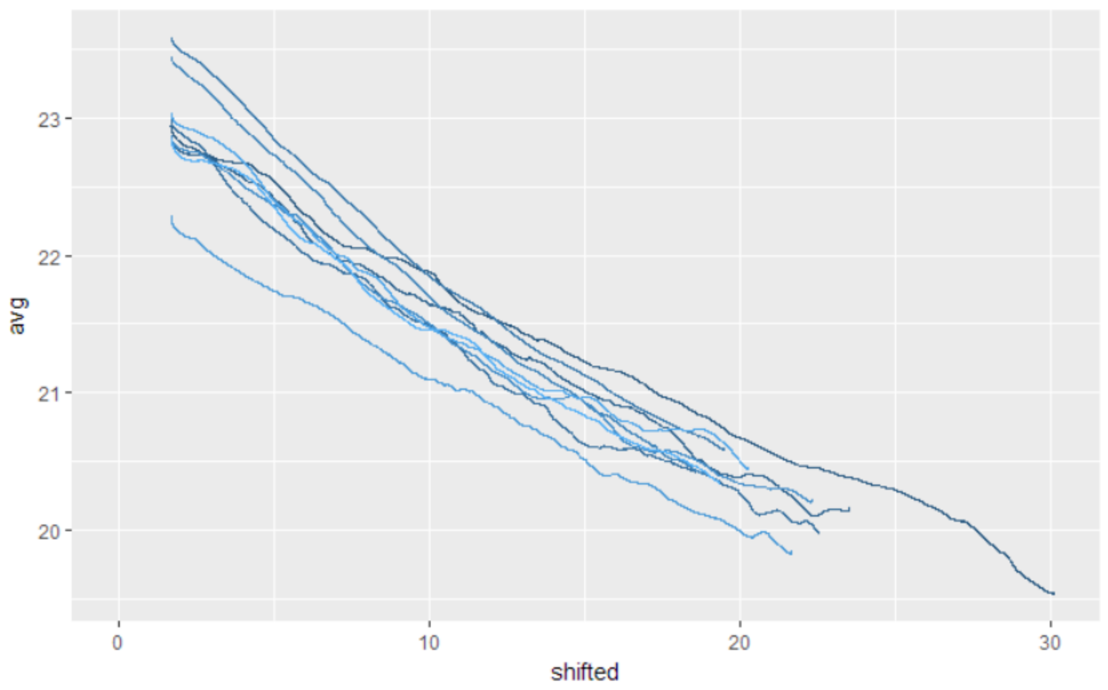

Figure 4.7: Improved low-pass filtered battery voltage graphs.

On figure 4.7 these problems have been rectified.

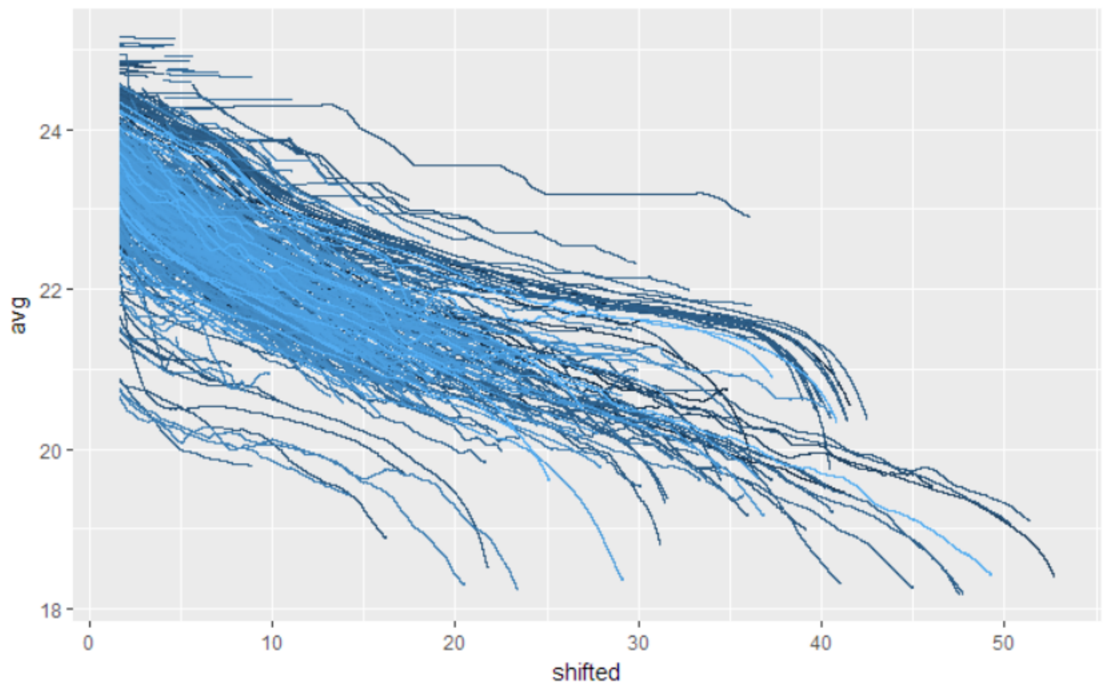

Figure 4.8: Improved low-pass filtered battery voltage graphs of the whole database.

Figure 4.8 displays all the voltage graphs of all the flights in the database. That is 1363 graphs. Here we can see that some shorter flights are not really flights at all and need to be filtered out from the database. We also see that there are at least two distinct battery types used. Ones that fly for 40 minutes maintaining higher voltage and ones that fly for longer while loosing voltage at a higher rate. On the bottom left we can see a few graphs that start off by dropping rapidly and then recovering a little. This is a sign of cold batteries. The multirotor uses 550 watts on average and thus heats the battery rapidly. Launching with a cold battery reduces the flight time considerably but warming due to consumption helps restore some of the capacity. This exemplifies the need for pre-heated batteries.

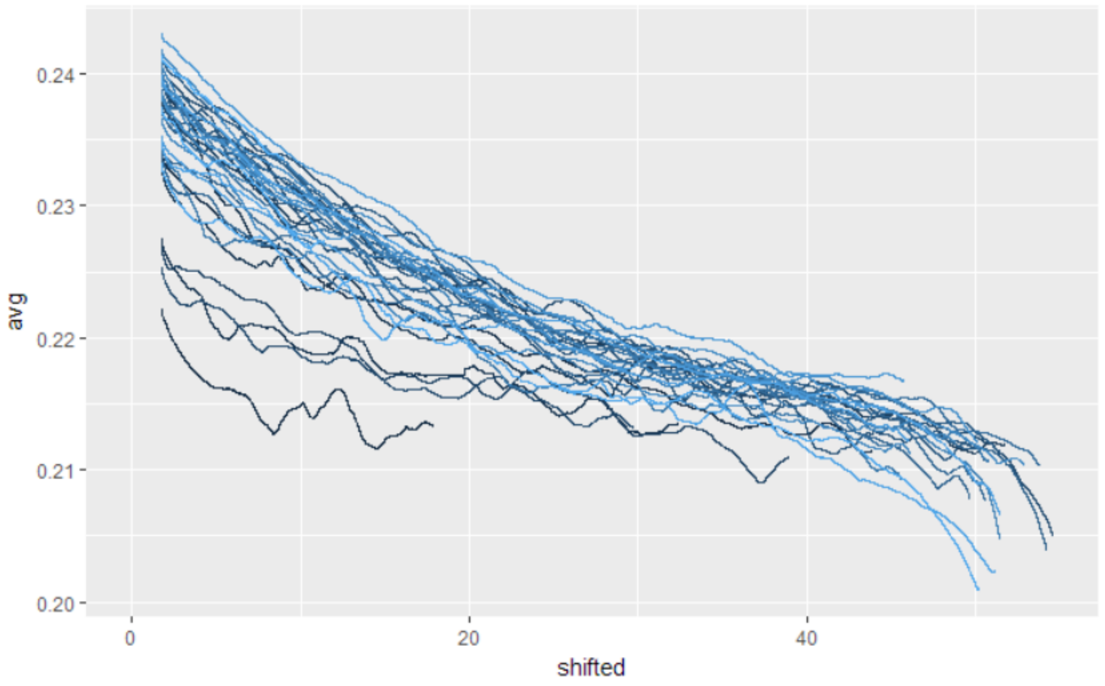

Figure 4.9: Improved low-pass filtered battery voltage graphs of selected flights.

Figure 4.9 displays the graphs of a single multirotor aircraft during a few days of time where counting wildlife in Estonian woods was carried out. The graphs show the improved battery that is capable of consistently flying for the guaranteed by Eli ltd 40 minutes in good weather but also going above that and reaching 50 minutes on most flights. On 4 graphs we can see the effects of not letting the battery cool off after use and before recharging and not fully charging the batteries.

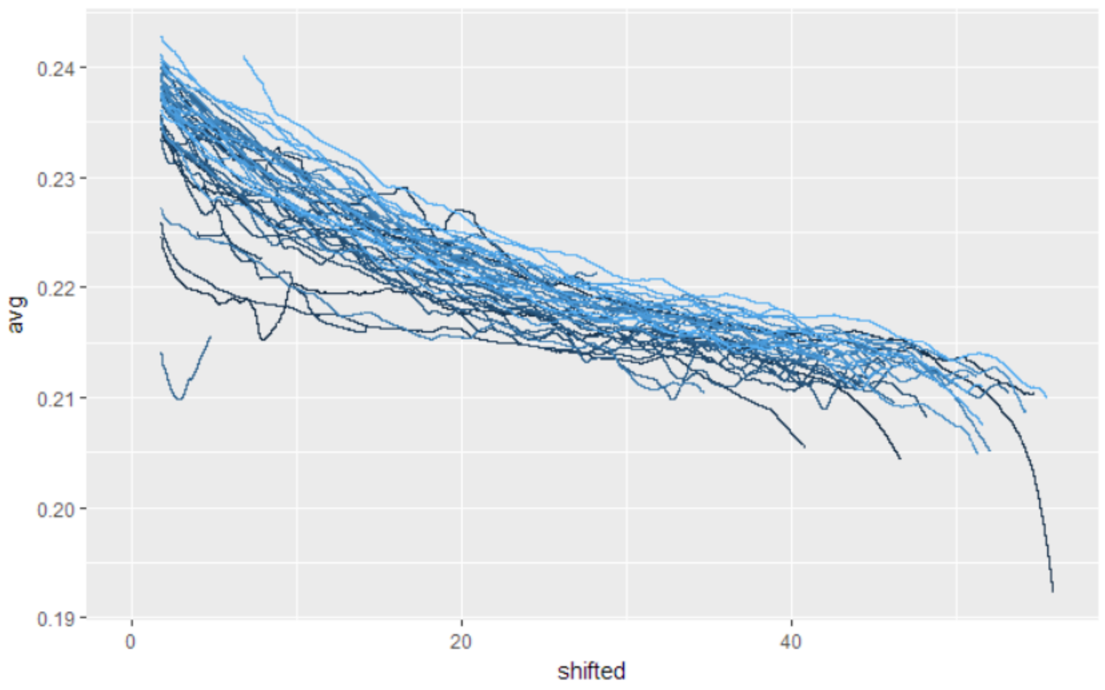

Figure 4.10: Improved low-pass filtered battery voltage graphs of selected flights of another multirotor.

Figure 4.10 displays the graphs of another multicopter that was also used in the aforementioned missions. Here we can see that during one flight the battery was allowed to empty more than usual. This risks breaking the batteries.

From the graphs we can determine that battery estimation is very difficult without knowing the type of the battery, the internal temperature and the state of charge. Another important parameter is the state of health of a battery as they degrade over time. Any smart algorithm needs to take into account the wear and tear of the battery. As long as there is no smart controller inside the battery to uniquely identify it, give its state of charge and state of health any smart algorithm will need to take into account the variability of the battery output. A simplification can be made by expecting fully charged batteries for every flight and placing that cognitive load on the pilot. Each piece of missing information about the battery state requires a bigger buffer for the safety mechanism reducing overall flight time and distance.

Conclusion

In order to create automatic missions that can be executed multiple times without external input regardless of weather a model of the multirotor behavior and a smart battery controller is needed. Until now the main focus of research has been on improving the autopilot flight controller algorithms.

Multiple data storage types were assessed. *SQLite* was chosen to be used as it is simple to use and acts like a real *SQL* database and it is easy to share as it is in a single file. Compared to other options it takes less disk space.

The data representation was analyzed and designed for ease of use in the analysis. The relevant code was created in *python* programming language converting the binary logs into *SQLite* database. This allows us to analyze several flights of data at the same time and make comparisons. Until now the tools allowed for analysis of a single log at a time. Further improvement suggestions were made.

Using the created database initial data analysis is done. As a result we present the way to filter out the relevant data to train the model of multirotor behavior on. Furthermore the need for a smart battery controller is shown. An experiment to test the behavior model accuracy in calculating wind speed and direction is proposed.

A Python database creation code

```
#!c:\python27\python.exe
# -*- coding: utf-8 -*-
'''
read a mavlink binary file into SQLite database
'''
from __future__ import print_function
from __future__ import division
from builtins   import range
from argparse   import ArgumentParser
from pymavlink  import mavutil
from timeit     import default_timer as timer
import os
import sys
import re
import sqlite3
import collections
import datetime
import math
extensions = collections.defaultdict(int)
def main(argv):
    # parser = ArgumentParser(description=__doc__)
    # parser.add_argument("database", help='Path to SQLite Database')
    # parser.add_argument("logs", metavar="LOG", nargs="+",
      # help='List of logs to process')
    # args = parser.parse_args(argv)
    # for filename in args.logs:
    #     process_tlog(filename)
    size = 0
    folder_path = argv[1]
    for path, dirs, files in os.walk(folder_path):
        for filename in files:
            if os.path.splitext(filename)[1].lower() == '.bin':
                filesize = os.path.getsize(path+os.sep+filename)
                if filesize < 100000000 and filesize > 4000000:
                    size += filesize
    print('Total filesize to process: ' + humanize_bytes(size))
    todo_size = size
    time_elapsed = 0
    for path, dirs, files in os.walk(folder_path):
        for filename in files:
            if os.path.splitext(filename)[1].lower() == '.bin':
                filesize = os.path.getsize(path+os.sep+filename)
                if filesize > 100000000 or filesize < 4000000:
                    continue
                print('Processing file\tFile size\tFolder')
                print('{}\t{}\t\t{}'.format(filename,
                  humanize_bytes(filesize), path))
                start = timer()
                process_tlog(path+os.sep+filename)
                end = timer()
                diff = end-start
                time_elapsed += diff
                time_spent = str(datetime.timedelta(
                  seconds=math.floor(time_elapsed)))
                processing_time = str(datetime.timedelta(
                  seconds=math.floor(diff)))
                todo_size -= filesize
                processed = ((float(size) - float(todo_size)) /
                  float(size)) * float(100)
                bit_time = float(diff) / float(filesize)
                time_left = str(datetime.timedelta(seconds=
                  math.floor(float(bit_time) * float(todo_size))))
                print("""File\t\tProcessing Time\tTime Elapsed\t
                  Time Left\tPercentage done""")
                print('{}\t{}\t\t{}\t\t{}\t\t{:2.2f}%'.format(filename,
                  processing_time, time_spent, time_left, processed))
def process_tlog(filename):
    '''convert a ardupilot BIN file to SQLite database'''
    mlog = mavutil.mavlink_connection(filename, dialect='ardupilotmega',
      zero_time_base=True)
    connection = create_connection('rmkRoheline.db')
    # conn = create_connection(args.database)
    database = {}
    try:
        database = load_database_to_ram(connection)
    except:
        database = create_database(connection)
    add_flight(database, filename, connection)
    counter = 0
    while True:
        line = mlog.recv_match()
        # Exit on file end
        if line is None:
            break
        message = line.get_type()
        # Remove bad packets
        if message == 'BAD_DATA':
            continue
        # FMT defines the format and PARM is params...
        # not sure if i need em or not
        if message in ['FMT', 'PARM']:
            continue
        process_header(line, database, connection)
        process_data(line, database)
        counter+=1
        if counter % 10000 == 0:
            bulk_write_values(database, connection)
```

```python
            bulk_write_timestamps(database, connection)
    bulk_write_values(database, connection)
    bulk_write_timestamps(database, connection)
    connection.commit()
    connection.close()
def process_header(line, database, connection):
    message = line.get_type()
    if message not in database['messages']:
        add_message(message, database, connection)
    fieldnames = line._fieldnames
    parameters = []
    for field in fieldnames:
        val = getattr(line, field)
        if not isinstance(val, str):
            if type(val) is not list:
                parameters.append(field)
            else:
                for i in range(0, len(val)):
                    parameters.append(field + '%s'% i+1)

    add_parameters(parameters, database, connection)
    if 'buffer' not in database:
        database['buffer'] = {}
    database['buffer'][message] = parameters
def process_data(line, database):
    # add message type with parameters to buffer to be able
    # to save to sqlite db later
    message = line.get_type()
    fieldnames = line._fieldnames
    data = []
    add_timestamp(line._timestamp, database)
    for field in fieldnames:
        val = getattr(line, field)
        if not isinstance(val, str):
            if type(val) is not list:
                data.append("%.20g"% val)
            else:
                for i in range(0, len(val)):
                    data.append("%.20g"% val[i])

    parameters_and_values = zip(database['buffer'][message], data)
    parameter = 0
    value = 1
    if 'values' not in database['buffer']:
        database['buffer']['values'] = []
    for parameter_pair in parameters_and_values:
        database['last value id'] += 1
        database['buffer']['values'].append((
            database['last value id'],
            database['last flight id'],
            database['last timestamp id'],
            database['messages'][message],
            database['parameters'][parameter_pair[parameter]],
            parameter_pair[value]
        ))
def load_database_to_ram(connection):
    """ load a database specified by connection into memory
    :param connection: connection to database
    :return: database
    """
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM parameter')
    parameters = dict(map(lambda (id, name): (name.encode('ascii'),
      id), cursor.fetchall()))
    cursor.execute('SELECT * FROM message')
    messages = dict(map(lambda (id, name): (name.encode('ascii'),
      id), cursor.fetchall()))
    cursor.execute('SELECT * FROM parameter ORDER BY id DESC LIMIT 1')
    last_parameter_id = (cursor.fetchall())[0][0]
    cursor.execute('SELECT * FROM message ORDER BY id DESC LIMIT 1')
    last_message_id = (cursor.fetchall())[0][0]
    cursor.execute('SELECT * FROM timestamp ORDER BY id DESC LIMIT 1')
    last_timestamp_id = (cursor.fetchall())[0][0]
    cursor.execute('SELECT * FROM value ORDER BY id DESC LIMIT 1')
    last_value_id = (cursor.fetchall())[0][0]
    cursor.execute('SELECT * FROM flight ORDER BY id DESC LIMIT 1')
    last_flight_id = (cursor.fetchall())[0][0]
    database = {
        'last value id': last_value_id,
        'last flight id': last_flight_id,
        'last message id': last_message_id,
        'last timestamp id': last_timestamp_id,
        'last parameter id': last_parameter_id,
        'messages': messages,
        'parameters': parameters
    }
    return database
def create_database(connection):
    """ create a database specified by connection and leave a copy
    into memory
    :param connection: connection to database
    :return: database
    """
    cursor = connection.cursor()
    cursor.execute("""
      CREATE TABLE flight (
        id INTEGER NOT NULL PRIMARY KEY,
        path TEXT
        )
      """)
    cursor.execute("""
      CREATE TABLE message (
        id INTEGER NOT NULL PRIMARY KEY,
        message TEXT
        )
```

```
  """)
cursor.execute("""
  CREATE TABLE timestamp (
    id INTEGER NOT NULL PRIMARY KEY,
    timestamp INTEGER
    )
  """)
cursor.execute("""
  CREATE TABLE parameter (
    id INTEGER NOT NULL PRIMARY KEY,
    parameter TEXT
    )
  """)
cursor.execute("""
CREATE TABLE value (
    id INTEGER NOT NULL PRIMARY KEY,
    flight INTEGER NOT NULL,
    message INTEGER NOT NULL,
    timestamp INTEGER NOT NULL,
    parameter INTEGER NOT NULL,
    value REAL,
    FOREIGN KEY(flight) REFERENCES flight(id)
    FOREIGN KEY(message) REFERENCES message(id)
    FOREIGN KEY(timestamp) REFERENCES timestamp(id)
    FOREIGN KEY(parameter) REFERENCES parameter(id)
)""")
database = {
    'last flight id': 0,
    'last message id': 0,
    'last timestamp id': 0,
    'last parameter id': 0,
    'last value id': 0,
    'messages': {},
    'parameters': {}
}
```

```
    return database
def bulk_write_values(database, connection):
    # write buffered values to database
    cursor = connection.cursor()
    sql = get_sql('value',['id', 'flight', 'timestamp', 'message',
      'parameter', 'value'])
    cursor.executemany(sql, database['buffer']['values'])
    database['buffer']['values'] = []
def bulk_write_timestamps(database, connection):
    cursor = connection.cursor()
    sql = get_sql('timestamp',['id', 'timestamp'])
    cursor.executemany(sql, database['buffer']['timestamps'])
    database['buffer']['timestamps'] = []
def add_flight(database, filename, connection):
    database['last flight id'] += 1
    cursor = connection.cursor()
    sql = 'INSERT INTO flight(id, path) VALUES(?,?)'
    cursor.execute(sql, [database['last flight id'], scrub(filename)])
def add_message(message, database, connection):
    database['last message id'] += 1
    database['messages'][message] = database['last message id']
    sql = get_sql('message', ['id', 'message'])
    data = (database['messages'][message], message)
    cursor = connection.cursor()
    cursor.execute(sql, data)
def add_timestamp(timestamp, database):
    if 'timestamps' not in database['buffer']:
        database['buffer']['timestamps'] = []
    database['last timestamp id'] += 1
    database['buffer']['timestamps'].append((
      database['last timestamp id'], timestamp))
def add_parameters(parameters, database, connection):
    buffer = []
    params = filter(lambda p: p not in
      database['parameters'], parameters)
    for parameter in params:
        database['last parameter id'] += 1
        database['parameters'][parameter] = database['last parameter id']
        buffer.append((database['last parameter id'], parameter))

    sql = get_sql('parameter', ['id', 'parameter'])
    cursor = connection.cursor()
    cursor.executemany(sql, buffer)
    connection.commit()
def get_sql(table_name, column_names):
    questionmarks = ','.join(map(lambda x: '?', column_names))
    column_names_string = ','.join(column_names)
    return 'INSERT INTO {}({}) VALUES({})'.format(scrub(table_name),
      column_names_string, questionmarks)
def create_connection(db_file):
    """ create a database connection to the SQLite database
        specified by db_file
    :param db_file: database file
    :return: Connection object or None
    """
    try:
        conn = sqlite3.connect(db_file)
        return conn
    except Exception as e:
        print(e)
    return None
def scrub(table_name):
    return ''.join( chr for chr in table_name if chr.isalnum())
def humanize_bytes(bytes, precision=1):
    """Return a humanized string representation of a number of bytes.
    Assumes `from __future__ import division`.
    >>> humanize_bytes(1)
    '1 byte'
    >>> humanize_bytes(1024)
    '1.0 kB'
    >>> humanize_bytes(1024*123)
    '123.0 kB'
    >>> humanize_bytes(1024*12342)
    '12.1 MB'
    >>> humanize_bytes(1024*12342,2)
    '12.05 MB'
    >>> humanize_bytes(1024*1234,2)
    '1.21 MB'
    >>> humanize_bytes(1024*1234*1111,2)
    '1.31 GB'
    >>> humanize_bytes(1024*1234*1111,1)
    '1.3 GB'
    """
    abbrevs = (
        (1<<50L, 'PB'),
        (1<<40L, 'TB'),
        (1<<30L, 'GB'),
        (1<<20L, 'MB'),
        (1<<10L, 'kB'),
        (1, 'bytes')
    )
    if bytes == 1:
        return '1 byte'
    for factor, suffix in abbrevs:
        if bytes >= factor:
            break
    return '%.*f %s' % (precision, bytes / factor, suffix)
if __name__ == "__main__":
     main(sys.argv)
```

References

About sqlite. (n.d.). *SQLite Home Page*. Retrieved from `https://www.sqlite.org/about.html`

Couture-Beil, A. (2014). *Rjson: JSON for r*. Retrieved from `https://CRAN.R-project.org/package=rjson`

Crockford, D. (n.d.). JavaScript object notation. *JSON*. Retrieved from `https://WWW.json.org/`

Edgar f. codd. (2018, May). *Wikipedia*. Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/Edgar_F._Codd`

First normal form. (2018, April). *Wikipedia*. Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/First_normal_form`

Müller, K., Wickham, H., James, D. A., & Falcon, S. (2017). *RSQLite: 'SQLite' interface for r*. Retrieved from `https://CRAN.R-project.org/package=RSQLite`

Ooms, J. (2014). The jsonlite package: A practical and consistent mapping between json data and r objects. *arXiv:1403.2805 [Stat.CO]*. Retrieved from `https://arxiv.org/abs/1403.2805`

Palomaki, R. T., Rose, N. T., Bossche, M. van den, Sherman, T. J., & Wekker, S. F. J. D. (2017). Wind estimation in the lower atmosphere using multirotor aircraft. *Journal of Atmospheric and Oceanic Technology*, *34*(5), 1183–1191. `http://doi.org/10.1175/JTECH-D-16-0177.1`

R Development Core Team. (2004). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from `http://www.R-project.org`

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., & Müller, K.

(2018). *DBI: R database interface*. Retrieved from https://CRAN.R-project.org/package=DBI

Ripley, B., & Lapsley, M. (2017). *RODBC: ODBC database access*. Retrieved from https://CRAN.R-project.org/package=RODBC

Smeur, E., Croon, G. de, & Chu, Q. (2018). Cascaded incremental nonlinear dynamic inversion for mav disturbance rejection. *Control Engineering Practice*, *73*, 79–90. http://doi.org/https://doi.org/10.1016/j.conengprac.2018.01.003

source, O. (n.d.). The world's most advanced open source relational database. *PostgreSQL*. Retrieved from https://www.postgresql.org/

team, A. (n.d.-a). Ardupilot. *Copter documentation*. Retrieved from http://ardupilot.org/

team, A. (n.d.-b). Marshalling / communication library for drones. *GitHub*. Retrieved from https://github.com/mavlink/mavlink

team, A. (n.d.-a). Pymavlink. *GitHub*. Retrieved from https://github.com/ArduPilot/pymavlink/blob/master/tools/mavlogdump.py

team, A. (n.d.-b). Python implementation of mavlink protocol. *GitHub*. Retrieved from https://github.com/ArduPilot/pymavlink

Third normal form. (2018, April). *Wikipedia*. Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Third_normal_form

Urbanek, S. (2018). *RJDBC: Provides access to databases through the jdbc interface*. Retrieved from https://CRAN.R-project.org/package=RJDBC

Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, *59*(10). Retrieved from http://www.jstatsoft.org/v59/i10/

Wickham, H. (2018). *Bigrquery: An interface to google's 'bigquery' 'api'*. Retrieved from https://CRAN.R-project.org/package=bigrquery

Yu, Y., Yang, S., Wang, M., Li, C., & Li, Z. (2015). High performance full attitude control of a quadrotor on so(3). In *2015 ieee international conference on robotics and automation (icra)* (pp. 1698–1703). http://doi.org/10.1109/ICRA.2015.7139416