Wind estimation for fuel management

A Thesis

Presented to

The Division of Information Technologies

Tallinn University of Technology

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

Karl Uibo

May 2018

Approved for the Division
(Thomas Johann Seebeck Department of Electronics)

Olev Märtens

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:

Abstract

This thesis is written in and is pages long, including chapters, figures and tables.

Annotatsioon

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti xx leheküljel, xx peatükki, xx joonist, xx 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

Table of Contents

List of Figures

List of Tables

Introduction

The following thesis looks at the flight logs of Eli Ltd. to assess the fuel economy and the information given to the user. To be written when rest of the paper is finished.

Chapter 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 powerlines, surveilance, 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 nessecary to gather data with the payload. A pilot needs to have certain skills and training to be able to safely operate an airial 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 expencive payload. For the high-end movie industry this is fine. Where this overhead becomes prohibitive is in the industrial applications. To inspect powerlines atleast 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 aswell as the camera and know enough of the powerlines 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 powerlines 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 expencive. This problem is smaller when it comes to tele-communications companies

as they have single cell towers which allways need to be inspected one at a time. For the energy sector a multirotor is prohibitively expencive 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 expencive camera and camera operator to fly over the powerlines at the right angles to later analyse the photos or video taken. This has a higher chanche of missing faults but compared to multirotors is cheaper alternative.

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

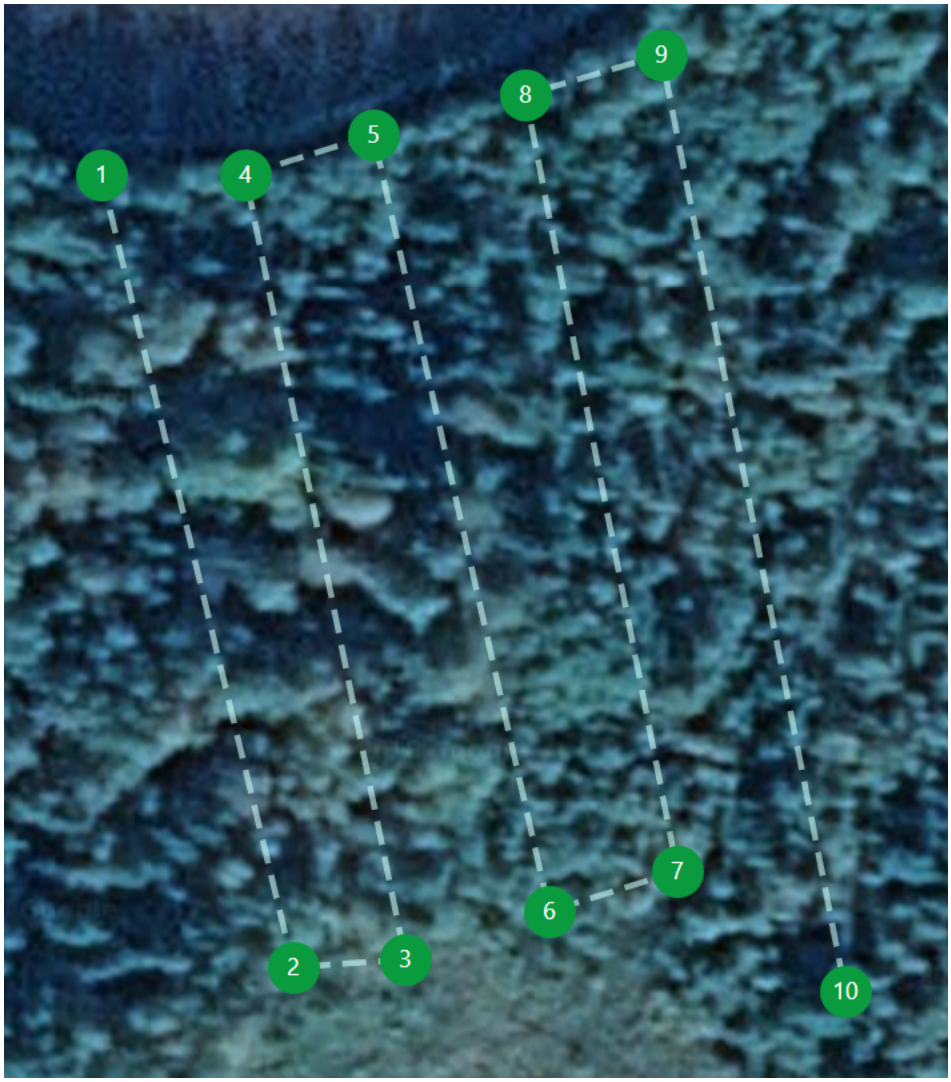

Figure 1.1: Search and rescue mission.[1]

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 alot better about their product. A skilled pilot however would be able to fly and potentially save someones life in that situation. A better aproach is needed.

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

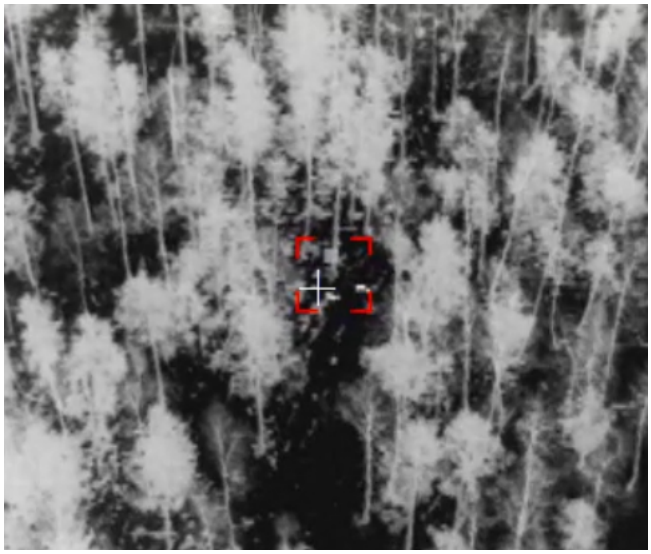

Figure 1.2: Heat signals of two deer in the woods.[2]

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

Area surveilance 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

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

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.

Chapter 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 succesful 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.

[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.

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 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 inrcease.

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.

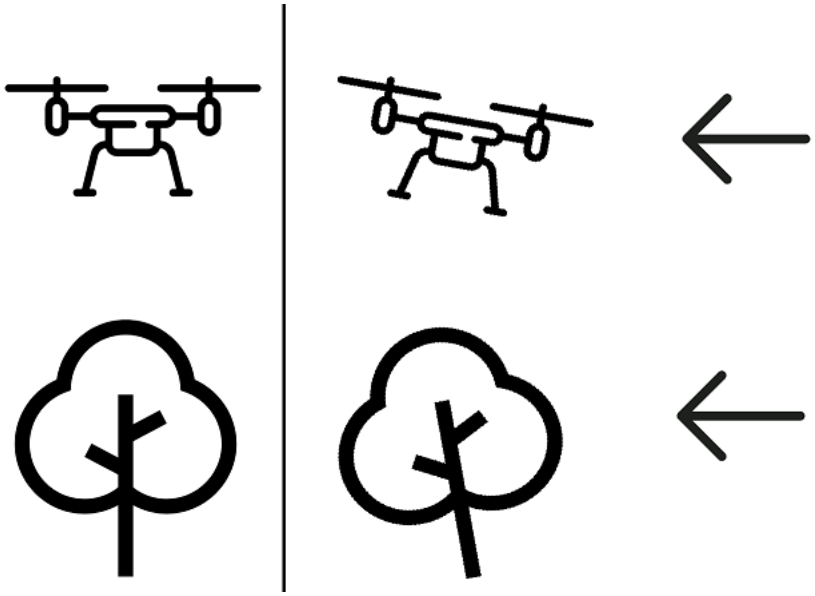

Figure 2.1: Stationary multirotor without and with wind.[3]

Secondly, when flying against the wind, the current draw increases to match the increased power required to stay airborne.

[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 at the first two effects more closely and the third in relation to maximum flight speed.

So far we have talked about

Chapter 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] Mavlink github page: `https://github.com/mavlink/mavlink`

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[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,DAccY
1521817093.27547407,688646373,-489.633148193,86.0853424072,-237.143630981...
1521817093.37760210,688748501,-489.633148193,86.0853424072,-235.895751953...
1521817093.47829199,688849191,-489.633148193,86.0853424072,-234.620544434...
...
```

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.

[2] R-project homepage: `https://WWW.r-project.org/`

[3] Python implementation of MAVLink protocol. Web page: `https://github.com/ArduPilot/pymavlink`

[4] Comma separated values file is a text file where a comma is used to separate values.

[5] Github repository: `https://github.com/ArduPilot/pymavlink/blob/master/tools/mavlogdump.py`

[6] Web page: `https://WWW.json.org`

- An ordered list of values.

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 *mavlogdump.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* by Jeroen Ooms, Duncan Temple Lang and Lloyd Hilaiel[7] and *rjson* by Alex Couture-Beil[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],

[7] https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf
[8] https://cran.r-project.org/web/packages/rjson/rjson.pdf
[9] https://WWW.sqlite.org/about.html
[10] https://cran.r-project.org/web/packages/RODBC/index.html

rsqlserver[11], *RJDBC*[12], *bigrquery*[13] and many others among which is *RSQLite*[14]. *RSQLite* embeds the *SQLite* database engine in *R*, providing a *DBI*[15]-compliant interface. *DBI* is an *R* package that defines a common interface between *R* and database management systems (*DBMS*). 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.

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

[11] `https://github.com/agstudy/rsqlserver`
[12] `https://cran.r-project.org/web/packages/RJDBC/index.html`
[13] `https://github.com/r-dbi/bigrquery`
[14] `https://github.com/r-dbi/RSQLite`
[15] `https://cran.r-project.org/web/packages/DBI/README.html`

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 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.[16]

3.4 Database normalization

ArduPilot[17] development team hosts an organization of projects on *GitHub*[18] that contains several useful projects such as the *ardupilot*[19] autopilot firmware codebase for multirotors, planes, rovers and much more. Among the project is *pymavlink*[20] 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[21] defined *normal forms* to permit querying and manipulation by a universal data language[22]. The third normal form (3NF)[23] 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

[16] Tidy data paper: `http://vita.had.co.nz/papers/tidy-data.pdf`, more relaxed essay: `https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html`

[17] `http://ardupilot.org`, `https://github.com/ArduPilot`

[18] An online platform for version controlling software development

[19] `https://github.com/ArduPilot/ardupilot`

[20] `https://github.com/ArduPilot/pymavlink`

[21] Computer scientist who invented relational database management systems `https://en.wikipedia.org/wiki/Edgar_F._Codd`

[22] One such language is *SQL*

[23] `https://en.wikipedia.org/wiki/Third_normal_form`

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

Table 3.5: Message ATT.

Message	Data
TimeUS	2412496394
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
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[24]. 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.

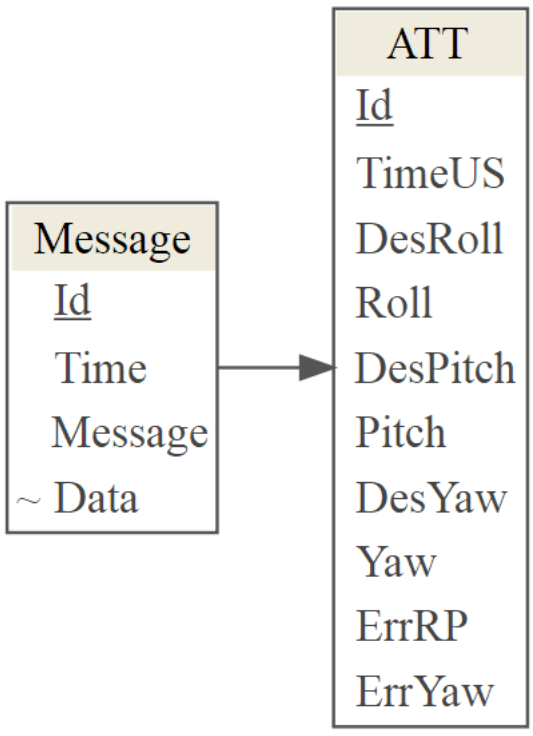

Figure 3.1: Message and ATT relationship.

[24] https://en.wikipedia.org/wiki/First_normal_form

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.

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 alot 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.

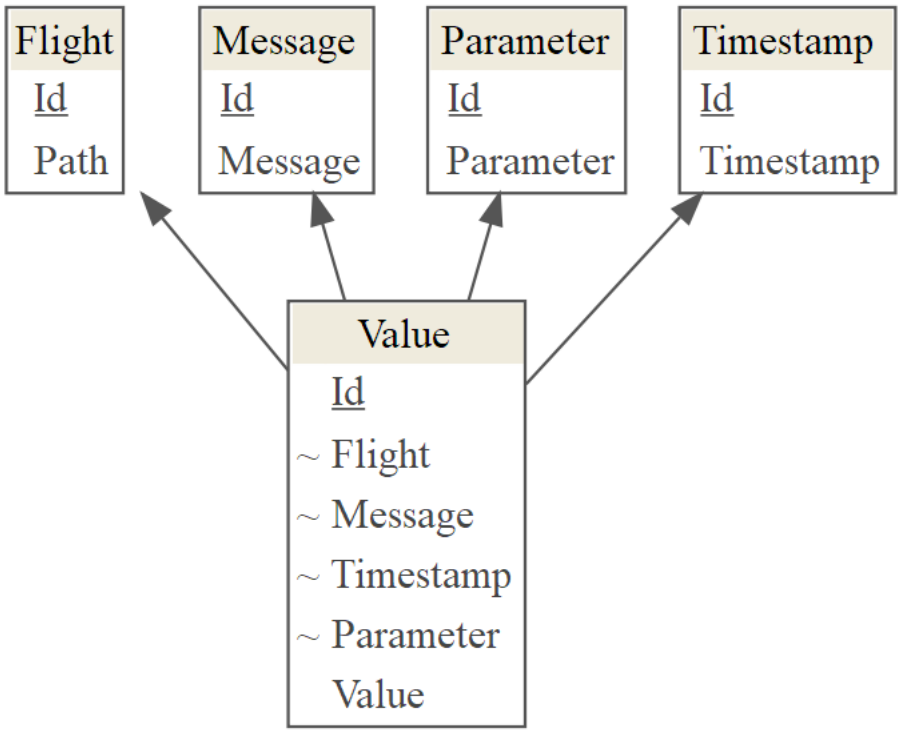

Figure 3.2: Final database schema.

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*[25]. 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

[25]Open source relational database. `https://WWW.postgresql.org`

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.

Chapter 4

Modelling stuff

After building the database

Chapter 5

The results of it all

The stuff i wanted to do and did.

Conclusion

Welp, thats it.

Appendix 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)
            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)
```

A.1 Demo appendix

This first appendix includes all of the R chunks of code that were hidden throughout the document (using the `include = FALSE` chunk tag) to help with readibility and/or setup.

In the main Rmd file

```
# This chunk ensures that the thesisdown package is
# installed and loaded. This thesisdown package includes
# the template files for the thesis.
if(!require(devtools))
  install.packages("devtools", repos = "http://cran.rstudio.com")
# Probably should install this from my own repo!!
if(!require(thesisdown))
  devtools::install_github("ismayc/thesisdown")
library(thesisdown)
```

In Chapter ??:

References

Angel, E. (2000). *Interactive computer graphics : A top-down approach with opengl*. Boston, MA: Addison Wesley Longman.

Angel, E. (2001a). *Batch-file computer graphics : A bottom-up approach with quicktime*. Boston, MA: Wesley Addison Longman.

Angel, E. (2001b). *Test second book by angel*. Boston, MA: Wesley Addison Longman.

Deussen, O., & Strothotte, T. (2000). Computer-generated pen-and-ink illustration of trees. *"Proceedings of" SIGGRAPH 2000*, 13–18.