FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Procházka

Capturing, Visualizing, and Analyzing Data from Drones

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science

Specialization: Programming and Software Systems

Prague 2016

I thank my supervisor prof. RNDr. Roman Barták, Ph.D. for his kind guidance, advice and time.

Title: Capturing, Visualizing, and Analyzing Data from Drones

Author: Tomáš Procházka

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: An application for visualization and analysis of the data recorded during the flight of a drone is proposed and implemented. The application displays plots of the recorded quantities, estimates the trajectory and shows the orientation of the drone in time. A file format based on comma separated values and XML description that allows reading of recordings with a dynamic structure is designed. The application is extensible, programmed in Java using the NetBeans Platform. It can be extended to support new types of drones, file types or visualization tools. The application is tested with the recordings of the drone Parrot AR.Drone 2.0. Data can be recorded by a separate application while controlling the drone by the keyboard.

Keywords: drones, data, grabbing, visualizing

Contents

1 Introduction

In the last few years, the advances in the field of drones have been drawing a lot of media attention. This is not only due to the fact that vehicles, which can fly and act autonomously, are an attractive and exciting topic, but mostly because of the possibilities that arouse in this area thanks to the progress in other branches of technological research, which brought, among other things, batteries with higher capacities and smaller yet more powerful computer chips.

This work concerns itself with drones of small dimensions, such as those used by hobbyists, rather than big, military drones. In practice, these drones are presently used mostly for taking pictures and capturing video at locations, which would otherwise be unreachable, and for remote observation of inaccessible locations and events. They are also used for entertainment and, which is the most important for us, in science, for example for research on artificial intelligence.

We can expect a further development both in the research that concerns drones directly (it tries to improve them) and indirectly (it uses them only as means of reaching other goals). Therefore, there will be a natural need to analyse data, which we can get from the flying machines in order to assess their performance and improve it. In this thesis, an application that serves this purpose is proposed and designed.

We want the application to be multiplatform and extensible. If we consider that new generations of drones are released regularly, the extensibility is necessary because otherwise the application would become quickly outdated. It has to be possible to adapt it for new conditions.

Our aim is to make a software that could be used with an arbitrary drone. However, for testing, the AR.Drone 2.0 is used, which is a quadcopter from the French company Parrot. Despite it was originally created as a toy for augmented reality games, it became popular in a range of other areas including research at universities.

Many authors have included a detailed introduction to quadcopters and the AR.Drone in their works. In the thesis, only the issues that are essential for understanding of this work are explained. For further information, please refer to, for example the article “AR-Drone as a Platform for Robotic Research and Education” [1].

Before the actual data visualisation and analysis, two questions have to be decided. The first question is how the data will be recorded and the second question is how they will be persisted. These issues will be discussed separately.

We collect two types of data. First, there are the values measured by the sensors, which are located on board of the drone such as gyroscope, magnetometer and altimeter. To the same category, we also include data, which were not measured directly, but were calculated by the on-board computer based on the measured values, for example by integrating those values or applying some filter. Together they form the current state of the drone. Second, there are the commands; the input data that control in which direction and at what speed should the drone move.

The data can be recorded either by the drone itself – the recording is saved into the drone’s internal memory – or it can be recorded by another device – for example mobile phone or computer – if the drone is able to send these data over the network. Such a device is usually connected to the drone, so that the drone can be controlled remotely, by a human pilot or by a piece of software which could not run on the drone itself for performance reasons.

Considering that many papers have been published on topics such as orientation of a drone in an unknown environment, automatic landing of a drone on a marked location or autonomous flight of a group of drones in a fixed formation, we can expect that some tools for the data analysis do exist. We will do the exploration and the comparison of the currently available software in the second chapter.

In the third chapter, the conceptual decisions that had to be made will be explained.

In the fourth and fifth chapter, we will study the concepts of the data analysis, namely how to interpret the orientation of the drone and how to calculate the approximate trajectory of the drone.

The sixth chapter introduces the resulting application from the user's perspective and the actual implementation is discussed in the seventh chapter.

Finally, in the eighth chapter, we discuss the results – whether the developed software complies with all the requirements we had and what are the possibilities of improvement and extensions.

Furthermore, the work also contains several attachments with user documentations for the supplied programs and tutorials for both end users and programmers who would like to extend the application for their needs.

2 Related work

An exploration of the currently available software that relates to controlling drones with focus on the software that in some way supports the AR.Drone has been performed. There are multiple reasons why quite a lot of time has been dedicated to this research.

The first and obvious reason is that we would not like to create something that already exists. If there was, however, a project that would coincide with our requirements, our application could be built on top of it.

Another reason relates to the choice of the suitable file format for saving the recordings. Despite it is always possible to create a new file format, if there exists any standard file format for these data, a preferred approach would be to use it.

In the following subchapters, individual projects and their data recording and data analysis features are introduced.

2.1 yaDrone Control Center

Many libraries and programs, that enable remote control of the AR.Drone from a computer, have been implemented in Java. However, most of them are outdated and inactive. The most interesting is a library yaDrone [2]. It is used to control the AR.Drone 2.0 and it is based on older Java projects (ArDroneForP5 [3], JavaDrone [4], lab-drone [5]). Apart from the library, yaDrone contains also application called ControlCenter, which provides a graphical user interface for controlling the drone using keyboard. The state of the drone is displayed, the received navigation data are shown in a text form and the video stream can be displayed as well, from either the front or the bottom camera of the AR.Drone. It is even possible to activate very simple plugins with plots of the recent attitude and altitude. There is no support for recording of the flight data, though.

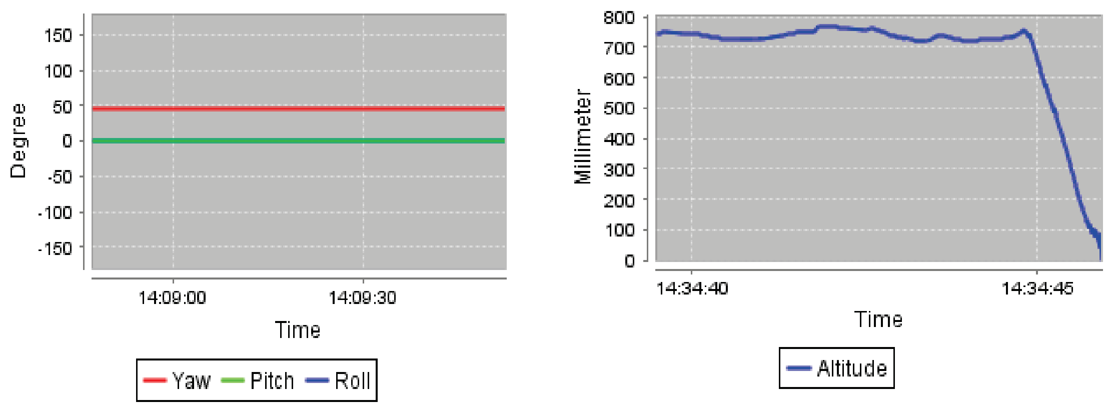


Figure 1 *Plots showing recent attitude (z, y and x rotation) and altitude available as plugins in the yaDrone Control Center*

The API and code of the yaDrone framework seems to be well designed and extensible and the ControlCenter can be extended by plugins.

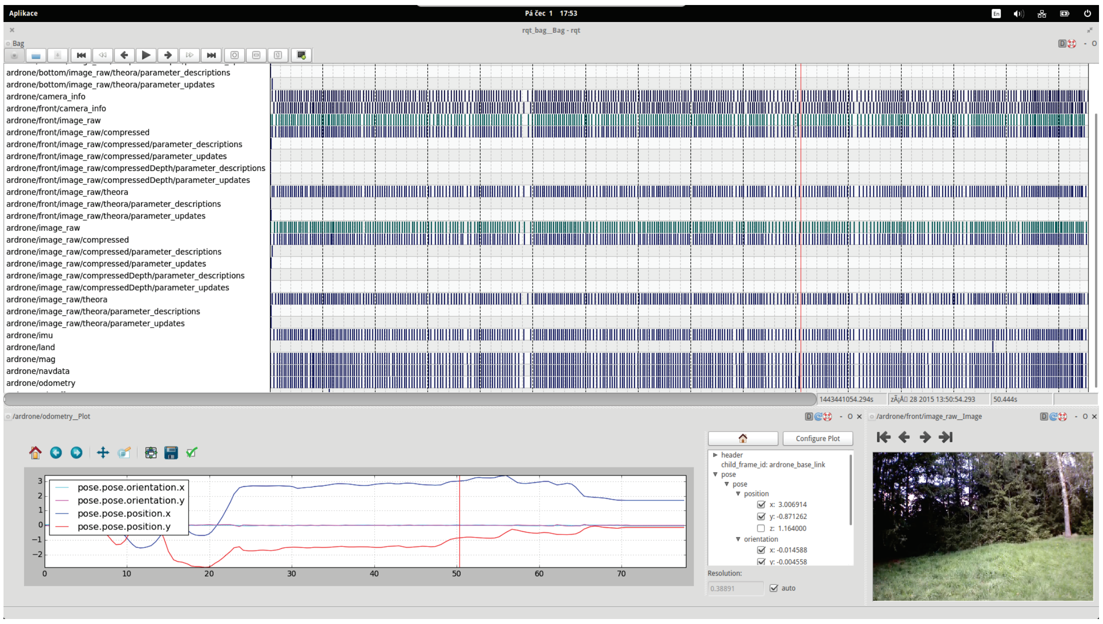
2.2 Ardrone_autonomy

Ardrone_autonomy [6] is an add-on to ROS [7]. ROS (The Robot Operating System) is a framework for building of robot applications. It is composed of a set of libraries and tools. Ardrone_autonomy supplies it with a driver for communication with the AR.Drone 1.0 and the AR.Drone 2.0.

ROS can run on Linux only; installation instructions are available for Ubuntu and Debian. (Although there is an experimental partial support for some other operation systems.) Writing of own programs is possible in C++ and Python. Most of the ROS tools are controlled from the command line.

ROS program consists of ROS Nodes[1] that exchange information in a transparent and public manner. Any node can receive any information that some other node publishes. ROS contains a tool called rosbag that can listen to published messages and save them to a file. Using rosbag it is possible to intercept and record the navigation data that are sent from the drone as well as the commands that are sent from the driver to the drone.

The file format in which the data are saved is called ROS Bag and it is binary. The ROS user can replay the data, which means that the recorded messages are republished in the same order and with the same delays as they have been recorded. The data can be visualised with a couple of generic tools. The simplest one is rqt_plot that opens a window with a plot showing the values of a given quantity in time. One can inspect the content of the recording using the utility rqt_bag. Other tools worth mentioning are rviz and ros_gui.



```
ardrone/bottom/image_raw/theora/parameter_descriptions
ardrone/bottom/image_raw/theora/parameter_updates
ardrone/camera_info
ardrone/front/camera_info
ardrone/front/image_raw
ardrone/front/image_raw/compressed
ardrone/front/image_raw/compressed/parameter_descriptions
ardrone/front/image_raw/compressed/parameter_updates
ardrone/front/image_raw/compressedDepth/parameter_descriptions
ardrone/front/image_raw/compressedDepth/parameter_updates
ardrone/front/image_raw/theora
ardrone/front/image_raw/theora/parameter_descriptions
ardrone/front/image_raw/theora/parameter_updates
ardrone/image_raw
ardrone/image_raw/compressed
ardrone/image_raw/compressed/parameter_descriptions
ardrone/image_raw/compressed/parameter_updates
ardrone/image_raw/compressedDepth/parameter_descriptions
ardrone/image_raw/compressedDepth/parameter_updates
ardrone/image_raw/theora
ardrone/image_raw/theora/parameter_descriptions
ardrone/image_raw/theora/parameter_updates
ardrone/imu
ardrone/land
ardrone/mag
ardrone/navdata
ardrone/odometry

1443441054.294s    zÃ¡Å 28 2015 13:50:54.293    50.444s

▶ header
  child_frame_id: ardrone_base_link
▼ pose
  ▼ pose
    ▼ position
        ☑ x: 3.006914
        ☑ y: -0.871262
        ☐ z: 1.164000
    ▼ orientation
        ☑ x: -0.014588
        ☑ y: -0.004558
```

***Figure 2** ARDrone 2.0 recording opened in rqt_bag*

2.3 LabVIEW

An extension exists for the graphical programming language and studio LabVIEW [8] that enables it to communicate with the AR.Drone. For instance, the ability to control the drone with devices LeapMotion and Kinect has been implemented [9]. LabVIEW contains components for analysing data and therefore it would probably be possible to use them even with drones. However, LabVIEW is a commercial program and thus it is mentioned here only for the sake of completeness.

[1] ROS Node is a single process of the program.

2.4 Node-Ar-Drone

Node-Ar-Drone [10] is an interface for controlling the AR.Drone using JavaScript. The JavaScript code runs in Node.js[1].

This environment is used by the participants of we sthe NodeCopter.js events. NodeCopter.js is, according to the official description, an event where 15 to 60 developers meet to create software for drones in groups of three [11]. As volunteers from America and Europe have organized almost 30 of these full day events since 2012, there is a large number of projects available on the internet.

One of the projects on this platform is an application Ar.Drone WebFlight for controlling drones from a web browser. The application itself can be extended by plugins. A plugin that records the flight data and saves them into file exists together with a plugin that replays the recording. This is useful for developers who can test their code with a simulated flight without actually flying with a drone. The complete states sent from the airplane are saved in the json[2] format and selected important values are also saved as CSV[3].

2.5 QGroundControl, Paparazzi UAV

QGroundControl and PaparazziUAV are programs that enable autonomous flights of drones during which the plane does not have to maintain the Wi-Fi (or any other) connection with the computer. QGroundControl is multiplatform, Paparazzi is limited to Linux. Both these programs provide tools for analysis of recorded data.

To be able to navigate autonomously, the drone needs to support GPS. AR.Drone does not comply with this requirement unless a special device called Flight Recorder is attached to it. QGroundControl communicates with the AR.Drone using MAVLink[4] protocol whereas Paparazzi replaces the on-board software by its own.

2.6 Other Drone Models

The available software for other common drone models is interesting for us, too.

The situation for the successor of AR.Drone, Parrot Bebop, is almost the same. There is a ROS driver bebop_autonomy from the same author as ardrone_autonomy. Even a communication interface for JavaScript, node-bebop, exists. Both QGroundControl and PaparazziUAV support this model.

Popular drones produced by another leading company in this field, DJI, seem to be rather incompatible with the software introduced above, though. However, these drones at least automatically record flight data into their internal memory. It is possible to take the logs out and a number of online services claim to be capable of analysing the log files and converting them to CSV. These services are often commercial and provide only a limited set of features free.

In fact, the AR.Drone also records the flight data into its internal memory. However, the data are saved in a binary and undocumented format. An official application AR.Drone Academy is able to read it and display statistics. The target group for that application are, of course, hobbyists, not researchers. Even Parrot Bebop saves the flight data into its memory and a Python script that converts it to comma-separated values is available on the internet [12].

[1] JavaScript runtime used mainly for running server-side applications

[2] JavaScript Object Notation – a human readable textual data-interchange format that, basically, serializes an object as a set of key – value pairs

[3] Comma-separated values

[4] Micro Air Vehicle Communication

2.7 Summary

The most of the existing applications are tied to a specific drone or they have been created to serve a different purpose and provide the flight analysis merely as an addition.

For example, it would be easily possible to extend the yaDrone Control Center application with more advanced tools for data analysis thanks to its plugin system that loads all available extensions dynamically at runtime. Unfortunately, as the program has been created directly for the AR.Drone 2.0 it would not be feasible to add a support for another drone.

An exception is ROS. The ROS tools provide similar functionality to what we would like to achieve. In spite of that, instead of extending ROS with additional features a new application will be programmed. The reasons are as follows. ROS violates the requirement on portability and it is a complex system. The user has to understand its philosophy in order to be able to work with it. Our application should be accessible and easy to use.

On the other hand, as ROS is widely used among the developers in robotics, it would be convenient to maintain at least some compatibility with it. This could be achieved by supporting the ROS Bag file format in our application.

The exploration did not result in finding any standardized file format for saving messages with time information. Applications usually use their own conventions. The ROS file type has been designed specifically for saving messages interchanged by robots and software that controls them and therefore it is well suited even for our application.

3 Conceptual Decisions

Java has been chosen as a programming language in which the application is written because it is the easiest way of reaching portability as well as extensibility.

The tools for visualisation and analysis should be accompanied with an easy method to record some flight data so that a user could try it out. As usually each drone model has its own communication interface, the recorder has to be implemented for each model separately. As a part of this thesis, it will be done for the test model AR.Drone 2.0 only. One option would be to have a recorder component as a part of the main application. An advantage of this approach is that we could read and analyse the data in real time during the flight. Despite that, an alternative approach has been chosen. We might have multiple recorders for different models of drones. However, the tools for analysis can stay the same for all of them. Because of that different nature of these parts, they will be put into completely separate applications.

3.1 File Format

It has already been mentioned that it would be useful to support the ROS Bag files. In accordance with that, the final application contains an option to load ROS Bag files recorded from within ROS and the ardrone_autonomy project. However, a new file format has been also developed to which the recordings in our recorder are saved.

The problem with the ROS Bag files is that the ROS Bag 2.0 specification [13] describes how single messages inside the file are saved but it does not guarantee that the description of their content is also present in the file. Furthermore, even if we know the data types of the entries in the message, we still do not know their semantics. In our code, we could read the messages on topics[1] that are standardized, but it would be useful to be able to read the content of any file without having to change the code. Rather than extending the specification of the ROS Bag by some additional metadata, it was easier to devise a new file format.

The new file format consists of one or more CSV files and one file with an XML description of the content of the other files. Together, all files are compressed into a ZIP archive. One line in a CSV file corresponds to one message such as a pack of data sent from the drone to the control device or vice versa. Each line has to contain a time information. Only one message type is saved to a single CSV file, all lines in the file therefore have the same format. The recording can contain multiple CSV files because we can have multiple message types. Usually, at least two are available – the telemetry and state data sent from the drone and the commands sent in the opposite direction.

ROS Bag files are binary and contain a lot of metadata in order to be ready for situation that the file might be too large and it will be necessary to work with it in the external memory. In comparison to that, we will assume that our recordings will be small enough to fit into the memory. Consequently, it is not such a big issue that the data are saved in a text form rather than in a binary form, because we load them only once and then work with them inside of the memory. Conversely, it is an advantage that we can obtain the data in a human readable form by mere unpacking the recording.

Another benefit of this method of persisting data is that if one can get flight data in a form of CSV from some other source, it suffices to write an XML description and compress it to

[1] The term ‘topic’ has a special meaning in ROS. It is a channel for interchanging of messages of a certain type. However, it is sufficient for us to understand it intuitively.

make the recording openable in our application. The format of the XML description is discussed in the chapter about implementation and its complete specification can be found in the Attachment 4.

3.2 Data Recording

In this section, a separate application is proposed, the main function of which is to collect the data during a flight of AR.Drone 2.0.

AR.Drone 2.0 communicates with the device that remotely controls it, in our case a computer, using a Wi-Fi connection.

It should theoretically be possible to develop an application that would intercept the network communication, being practically independent even on the application that controls the drone. This would be the best conceivable solution, as it would be possible to use the recorder with any application mentioned in the chapter Related Work. We might also be able to record the data on one device and control the drone from another, for example a mobile phone. Even though that Java is a high-level programming language, it appears that libraries that provide access to low-level network communication do exist. These libraries, such as jNetPcap [14] and jPCap [15] even declare that they are multiplatform. Nevertheless, they depend on other native tools that have to be installed on the system first (libpcap or winpcap) and their portability is not always complete. The result might as well be dependent on the actual hardware of the computer. This approach would be complex and introduce many pitfalls and therefore it was avoided, considering also that it is not the main objective of this work to provide a comprehensive solution for AR.Drone.

As a result, it will not be possible to use the recorder with an arbitrary application. It would be desirable if developers could at least use the recorder as a library from within their own applications. In order not to have to implement whole communication protocol and interface for controlling the drone from scratch, the application will be based on the yaDrone project.

The yaDrone library will be extended to provide an interface for data recording and saving. Then a plugin for the graphical user interface ControlCenter will be created. The plugin enables the user to start and stop the recoding. The yaDrone extension will be called yaDrone-Recorder.

3.3 Visualisation and Analysis

The task of our main application is to load data from a file, analyse them and display the results to the user. The application will be called Drone Flight Inspector.

The graphical interface will contain multiple components, each component visualising some type of data. Multiple components might also visualise the same data, but from a different perspective. One component could display for example a plot of the altitude of the drone in time or trajectory of the flight.

Furthermore, a group of components can show the state of the drone in a certain moment such as its current attitude or a table of all available information. For this to work, we also need to have a component that will let the user to select time on a timeline.

We can see that, in most cases, these components may work independently on each other, which means that we should be able to separate them not only in the user interface but also in code. We want to have the application extensible and adding a new component is a natural way of extending it. Thus, it should be possible to add a new component effortlessly.

We could place the components on screen in a fixed layout, but it would cause problems in case of adding new components and it would be inconvenient for the user. To make the environment user-friendly, the components need to be resizable and it should be possible to rearrange them. We need a window management framework. It is unnecessary to design our own as such frameworks are available. First, Java Swing GUI toolkit has some support for multiple-document interface via a JDesktopPane container. However, it is very simple, outdated and insufficient for our use. Authors of the Java IDEs NetBeans and Eclipse provide solutions that are more sophisticated - NetBeans Platform [16] and Eclipse RCP (Rich Client Platform) [17]. Both these platforms offer much more than only a window management framework. Use of one of these platforms could also help us to fulfil the requirement for extensibility.

In terms of features, the platforms are comparable. The NetBeans Platform has been chosen only due to a personal preference.

4 Orientation in 3D Space

In this chapter, we study how orientation of objects in three-dimensional space is stored and how to work with the orientation data we receive from the AR.Drone.

4.1 Representing Orientation

We usually store the rotation that determines how the object has to be rotated from some reference placement in space in order to get its current orientation. The common ways of storing the rotation are introduced in the paragraphs below.

4.1.1 Rotation Matrices

Rotation matrices in 3D are three by three matrices of real numbers that define a linear mapping in the vector space of three-component real vectors. Rotation of a vector is performed by multiplying the vector by the rotation matrix from the left. An advantage of rotation matrices is that the composition of rotations is done easily using the matrix multiplication. This property is exploited in computer graphics, in which the rotation matrices are primarily used. For example, when drawing the pose of the drone in Drone Flight Inspector, the rotation of the drone is converted into the rotation matrix by which all the vertices of the AR.Drone 3D model are multiplied. Nevertheless, the conversion occurs underneath in the JavaFX framework and is not part of our code.

4.1.2 Quaternions

Quaternions are numbers that consist of four components, three imaginary parts and a real part. They are generalisation of complex numbers in four-dimensional space. The rotations are represented by unit quaternions[1]. The vector of the imaginary parts corresponds to the axis of rotation and the real part corresponds to the cosine of the rotation angle. An arbitrary rotation in 3D can be represented as a rotation along a single axis by a given angle. The quaternions are used, again, in computer graphics, because they allow a smooth interpolation between two orientations. However, they are also the most preferred way of representing rotations in ROS, according to the ROS document Standard Units of Measure and Coordinate Conventions [18].

4.1.3 Euler angles

Another option is to describe the orientation of an object by three consecutive rotations. As the axes of rotations, we take either the axes of the global coordinate system in which the object is located or the axes of the local coordinate system of the rotated object. The order of the rotations is given and applying them in a different order usually results in a wrong orientation of the object. The three angles that specify the rotations are called the Euler angles.

If we use the global coordinate system, the axes along which we rotate the object are fixed. However, if we use the local frame of reference, the axes of rotation rotate together with the object. After each rotation, we have to apply the transformation even on the remaining axes.

One could expect that we use the three distinct axes of the coordinate system for the three rotations, but it does not have to be the case. It is possible to take the same axis twice. Even if we choose the first and last axis the same, we can still represent an arbitrary orientation.

The problem of Euler angles is that there are too many alternative approaches to the selection of the rotation axes. Although some conventions exist, there is no unified

[1] Unit quaternion is a quaternion with norm equal to one. The norm is calculated in the same way as the Euclidean norm of a four-dimensional vector.

convention that would be used in all fields. This is the reason why Euler angles take the last place in the list of recommended rotation representations in the ROS standard mentioned above.

4.2 Attitude of the AR.Drone

4.2.1 Yaw, Pitch and Roll

In aeronautics, roll, pitch and yaw are traditionally used to represent the position of an aircraft in the space and we also receive these three values from the AR.Drone 2.0. The roll of the plane specifies how much it is tilted to the left or to the right. The forward or backward tilt is determined by the pitch value. Finally, the value of yaw indicates how much the drone is rotated along the vertical axis. The rotation axes with directions of the rotation are indicated in the *Figure 3* below. This figure has been included despite the fact that many similar images can be found on the internet as well as in the published works concerning drones, in order to emphasize the coordinate system that has been adopted in Drone Flight Inspector. The coordinate system is right-handed, x-axis points forward, y-axis to the right and the values on the z-axis increase as we move down. The direction of the rotation is determined by the right-hand rule.

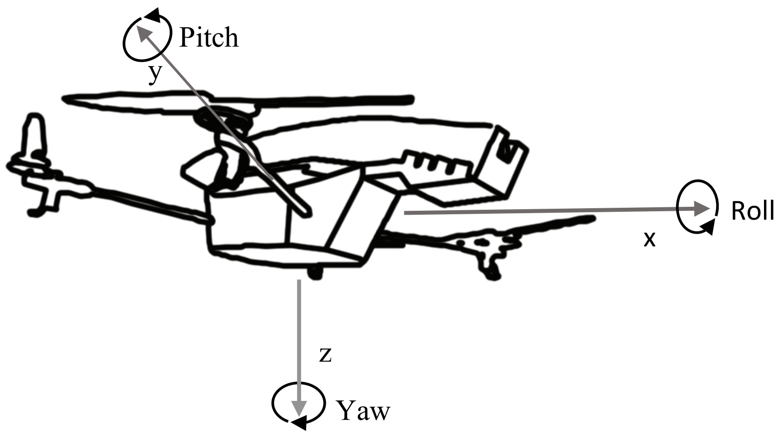


Figure 3 *The coordinate system used both by the AR.Drone 2.0 and by the Drone Flight Inspector application*

4.2.2 The Ordering of the Angles

Yaw, pitch and roll are Euler angles and so if we want to reconstruct the attitude of the drone we need to know the order in which to apply the rotations. However, the AR.Drone documentation for developers does not state the used convention.

To determine the correct order of the rotations, an experiment had to be done. Using yaDrone-Recorder, four sequences of two rotations have been recorded. The rotations have been selected such that if we apply them in the opposite order, the resulting attitude is apparently different. Then the orientation of the drone has been displayed in the Attitude 3D Window in Drone Flight Inspector and the order of the rotations has been modified to match the output of the window with the expected results. The correct order is: first yaw, then pitch and finally roll. This convention does not match with the order in which the angles are enumerated in the official documentation and the C++ library. However, it is reasonable as we first apply the rotation that can change the attitude of the drone most. Naturally, the frame of reference used by the drone is bound to its body. The test recordings with description can be found among the files attached to this thesis.

5 Trajectory Calculation

Determining the path that the drone followed during the flight is a vital part of the flight analysis. In this chapter, we study the basic methods used in this field and then we apply them in order to estimate the position of the AR.Drone in time. At the end, the algorithm that has been used in the Drone Flight Inspector application is evaluated.

5.1 Trajectory Calculation Techniques

5.1.1 Used Terminology

We start by specifying our target more precisely. We will work with two coordinate systems. The local coordinate system has its centre in the centre of gravity of the object and not only the position, but even the orientation of the object in relation to this coordinate system stays the same throughout the time. The global coordinate system is fixed to the environment in which we do the measurement. We declare the starting location of the object to be the centre of the global coordinate system, so that initially, the two coordinate systems coincide. As the object begins to move, its position towards the global coordinate system starts to change. By the position of the object in a certain time, we will understand the coordinates of the object in the global coordinate system. By the orientation of the object, we will understand the orientation of the local coordinate system with respect to the global coordinate system.

5.1.2 Sensor-based Position Estimation

In this subsection, we discuss how orientation and position of a device in three-dimensional space is estimated in general by using the minimal required number of sensors.

If we consider the position of the object as a function of time, by differentiating it, we obtain a function of velocity and its second derivative gives us the acceleration. In general case, there are no sensors that measure directly the position or velocity, but the acceleration can be measured by accelerometer.

Even though the accelerometer gives us a discretized stream of values instead of a continuous function and the measurements are not completely precise, we could perform estimated integration twice to obtain the desired position. The problem is that the accelerometer is located on the device and therefore the acceleration vectors are measured in coordinates of the local coordinate system, but we need the global coordinates.

If we knew the orientation of the device, we would be able to transform the acceleration vectors into the global coordinates. To compute the orientation, we need gyroscope measurements. The output of gyroscope is a stream of vectors of rotation rates. We calculate the actual rotations for short time intervals and we accumulate the rotation in a rotation matrix. The rotation matrix can then be used as a transformation matrix between the local and the global coordinate system.

Because of the discretization and a noise in the sensor readings, the computed position will be imprecise. To improve it, we use the data from other sensors, such as magnetometer, which produces vectors of intensities of the magnetic field. The issues that arise when determining the location based only on the sensors mentioned above are described and solved in the bachelor thesis by Filip Matzner “Tracking of 3D Movement” [19].

5.1.3 Model-based Position Estimation

As we already know, the mere integration of the sensory data does not lead to the precise position. Therefore, we would like to use all the available information about the specific situation to increase the accuracy of the computed position. We can create a model of the movement of the object and modify the results of our calculations in order to suppress the values that are suspicious with respect to the usual behaviour of the object and occurred most likely due to an error.

We can observe that there are many constraints on the flight behaviour of the drones that could be used. For example, drones are unable to change velocity or direction rapidly and the quadcopter cannot tilt more than to a certain given angle.

An often-used algorithm for model based position estimation is the Kalman filter. It estimates a state of a system. The state is a vector of values that characterise current conditions in the environment. In our case, it would be the position. The Kalman filter processes the data successively and is therefore able to compute in real-time as the data arrive. According to the current state and the model of the system, the Kalman filter predicts the next state. When new data become available the next state is determined based on the measured values, however it is not immediately accepted as a new current state. Instead, the result is compared with the expected value gained from the model and a new state is calculated based on the fact how much we believe to our model and how much we believe to the measured data.

The Kalman filter works under the assumptions that the changes of the state variables are linear, the noise that affects the measurements is white and it has Gaussian distribution. The fact that the noise is white means we are unable to predict its future values even when we know its past development. In practice, the biggest problem is with the linearity of the system. It is solved by the extended Kalman filter algorithm, which enables the use of the nonlinear transition function between states, at the expense of losing the mathematical property of being the optimal (best possible) estimate. The Kalman filter is further explained in the book “Stochastic Models, Estimation and Control” by Peter S. Maybeck [20].

5.2 Trajectory of the AR.Drone 2.0

The AR.Drone 2.0 offers us a wider range of possibilities and better prospects than the general case. The reason is that the AR.Drone is equipped with many sensors that help to improve the quality of the determined values.

There is an ultrasound sensor and a pressure sensor that both contribute to the computation of altitude. Moreover, the drone features an on-board computer that performs the sensor fusion and sends us, among other data, the resulting altitude. In spite of being a combination from two sources, the altitude computation sometimes fails in such a way, that it for example, unexpectedly starts to report negative numbers. There is therefore an opportunity to improve it by filtering the values. However, most of the time, the altitude measurement is accurate and we will consider it to be the z coordinate of the position of the drone. This simplifies our problem on determining the position in two dimensions.

From the previous subchapter, we know that the drone sends us the orientation of the drone as the yaw, pitch and roll values. Thanks to that, we do not have to do the integration of the raw gyroscope measurements. Furthermore, we also receive the velocities in x-axis and y-axis. These values are more accurate than those we would get by integration of the accelerometer readings, because the on-board computer analyses the image from the bottom camera and improves the velocity estimates based on the movement of the key points recognized in the image.

It is possible to create a model that is based on the parameters sent by the drone and use it to enhance the position calculation. The extended Kalman filter is utilised in the master thesis of Jakob Julian Engel “Autonomous Camera-Based Navigation of a Quadrocopter” [21]. An advantage of this approach is that the commands for the drone can be included as the inputs and reflected in the model, so that it predicts the expected behaviour of the drone. On the other hand, as the received data have already been filtered by the on-board computer, the white Gaussian noise requirement is violated. Therefore, compensations have to be made.

5.2.1 Trajectory in Drone Flight Inspector

In Drone Flight Inspector, no advanced filters are applied and the trajectory is calculated by estimating the definite integral of the velocity function.

At first, we define our global coordinate system. It will have its centre at the point in which the drone is located at the start of the recording. However, the direction of the axes will not be bound to the initial orientation of the drone. As the drone reports its yaw (rotation around the vertical axis) with relation to the north, the coordinate system can be aligned with the cardinal directions. The positive x-axis points to the north. It follows that the positive y-axis points to the west.

Next, we need to convert the coordinates of the two-component velocity vector to the global coordinate system. Because the pitch and roll angles are always small and the velocity values are computed with respect to the movement of the image of the terrain, we will assume that the velocity vector is parallel to the terrain, disregarding the tilt of the drone. Therefore, as we consider only the yaw rotation, transforming the velocity into the global coordinate system becomes an easy trigonometry task in 2D plane, as illustrated in a figure below for the x component of the velocity vector.

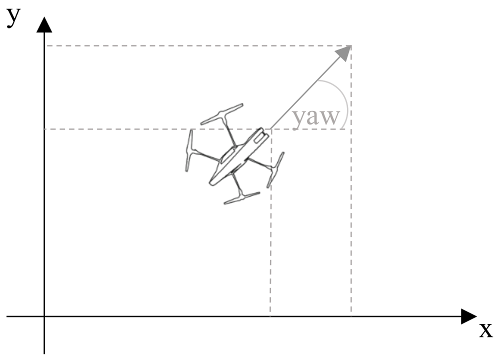


Figure 4 *The velocity of the drone in the x-axis conversion to the global coordinate system*

Finally, we estimate the translation of the drone in the time intervals between each two subsequent velocity records and we get the positions by summing the translations. We approximate the velocity change in each interval by a linear function and handle the average of the velocities at the endpoints of the interval as the average velocity.

5.2.2 Evaluating Outputs

In order to evaluate the performance of the algorithm, a series of flights has been executed in which the drone has been navigated along a rectangular path and it had to land in the same place as it took off. Then, the distance of the landing point from the initial location has been measured in the calculated trajectory. The sides of the rectangle were 6 and 3 metres long; therefore, the total length of the path was 18 meters. The path is depicted in the *Figure 5* together with a sample result of the algorithm.

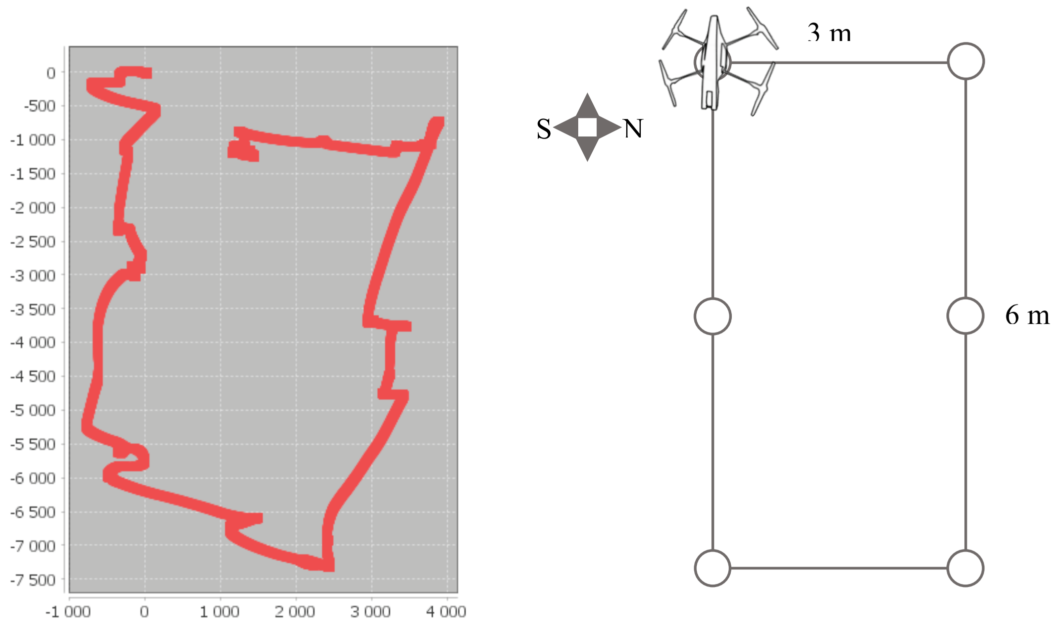


***Figure 5** The ideal path on the right and the calculated trajectory on the left (values in millimetres), which reflects the inaccuracy of both the algorithm (landing and take-off not on the same spot) and driving (the curled curve shape)*

Several test flight recordings were affected by a gap in the navigation data stream. Due to a network error or because the CPU of the drone was overloaded, no velocity data have been sent for up to a few seconds. In such case, the interval between the velocity values is too long to be sufficiently approximated by a linear function. As a result, a drift occurs in the calculated trajectory. To fix this issue, the algorithm has been modified to allow only the intervals shorter than twenty milliseconds. In case of a longer time lag, the algorithm leaves the drone in place instead of trying to estimate its movement. This modification improved the affected recordings significantly but, of course, it is impossible to compensate for the missing data.

Two variations of the test have been performed. In the first of them, the drone had to fly without any yaw rotation. This test has been repeated eight times. Alternatively, the drone had to fly only forward, rotating ninety degrees on each corner including the last one, so that it lands in the same location as it started. This version has been repeated eleven times. Furthermore, in five additional tests, the drone had to follow the rectangular path twice. Two example two-round tests are shown in the *Figure 6*.

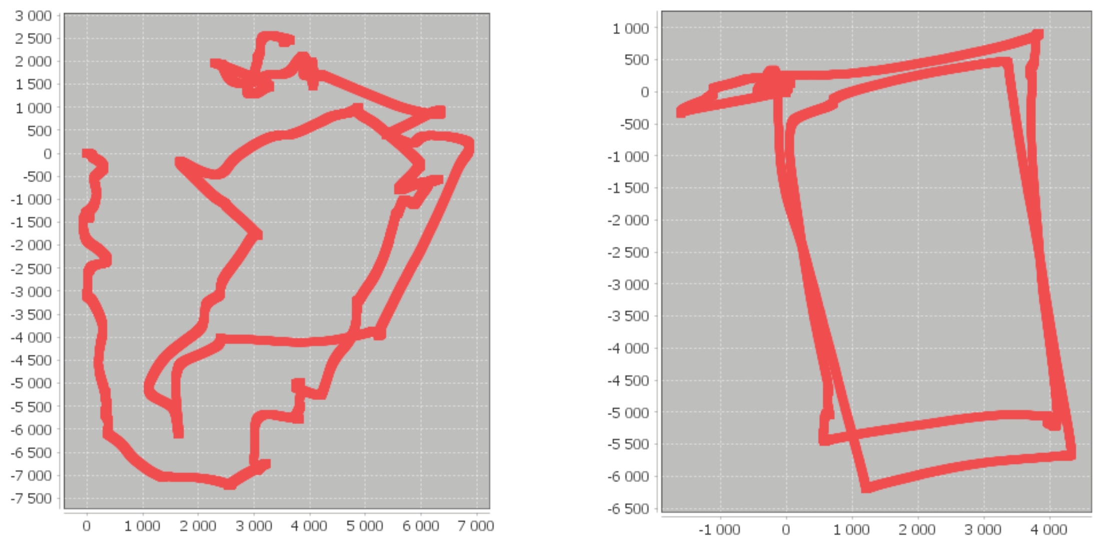


Figure 6 *Two results of the two-round tests. The trajectory on the right (recorded without rotation) is more accurate due to the flight being straighter. The trajectory on the left has been recorded with rotating the drone.*

If we include all recordings, the average distance between the starting and landing point in the path calculated by the algorithm is approximately 143 centimetres. Excluding the two-round flights, the average distance amounts to 128 centimetres, the average for the no-rotation recordings only is 123 centimetres and for the recordings with yaw rotation, it is equal to 131 centimetres. The results table is located in the Attachment 5. All recordings are available in the electronic attachment of this thesis.

Even though we might expect that the results of the flights without rotation will be better than the results of the flights with rotation, the real difference is negligent. The average distance of the flights with rotation is slightly higher, but we have to consider that the flights with rotation have been disadvantaged. The drone sometimes had a tendency to drift during rotations; therefore the piloting of the flights with rotation was more demanding and the flights usually longer with less straight trajectory.

Many external factors influenced the flights, such as the current condition of the drone, the errors of the pilot, the state of the battery or the strength and direction of the wind. Therefore, no statistical conclusions can be drawn from the results of the experiment about the algorithm; we would have to collect much more data. However, we can see that the output is good enough to provide us with the overview of the flight, but, as expected, we would have to use a more advanced technique in order to achieve results suitable for situation in which the precise position is critical, such as the autonomous navigation of a drone.

Some of the recordings have been affected by momentary gaps in the velocity stream, as already discussed. On the other hand, two circumstances that improved the quality of the recordings should also be mentioned.

Firstly, the drone has been flying above a rough terrain (mostly grass), which is a precondition for the proper functioning of the algorithm that analyses the bottom camera image. As we rely on the precision of the velocity values, the results would be worse in the environment with a terrain that does not contain any orientation points discernible by the on-board computer.

Secondly, the flights have been recorded in an area in which there were no Wi-Fi networks or electrical appliances that would influence the magnetic field intensities measured by the drone sensors. The yaw values could be less accurate in a location in which the magnetic field interference is stronger. To illustrate this phenomenon, two recordings of a motionless drone are included in the attached files. In the recording acquired in a place with many Wi-Fi networks, the stated yaw value changes by more than one hundred degrees in a course of five minutes. In the recording from an environment with no external Wi-Fi network, the yaw measurement fluctuates around a single value during the whole time.

6 Introduction to Drone Flight Inspector

This short chapter shows how the application Drone Flight Inspector can be used in practice, before the implementation details are explained in the next chapter. For a complete description of all features of Drone Flight Inspector, refer to the Drone Flight Inspector User Manual in the Attachment 1.

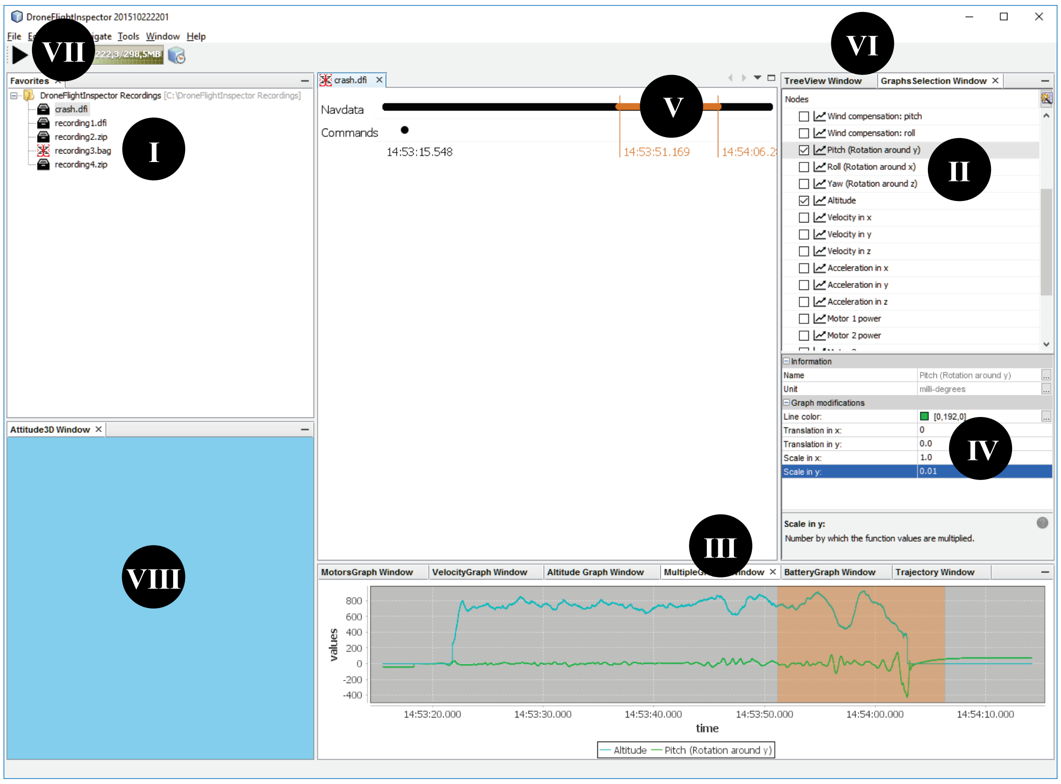


Figure 7 *Drone Flight Inspector user interface*

We analyse a recording in which the crash of a drone is recorded. The steps are marked in the screenshot of the user interface above. The file with the recorded crash can be found in the electronic attachment.

I. First, we open the recording in the Favourites panel.
II. Then, we check the quantities that we want to plot on the Graphs Selection Window. We choose pitch and altitude.
III. We display the Multiple Graphs Window.
IV. As we can see, the pitch values are big in comparison with the altitude, because the pitch is in millidegrees. We select the pitch in the Graphs Selection Window and set its scale in the y-axis to 0.01.
V. We select the critical part of the recording by dragging with the mouse over the timeline.
VI. We switch the right panel to display the Tree View Window.
VII. We replay the highlighted segment by pressing the play button.
VIII. We observe the orientation of the drone in the Attitude 3D Window.

7 Implementation

In this chapter, we discuss the design of our applications and address the issues that arouse during the implementation. All subsections are devoted to the Drone Flight Inspector application apart from the last one, which describes the development of yaDrone-Recorder. Because Drone Flight Inspector is based on the NetBeans Platform, basic concepts of this framework are explained first. Then, the division of the application into modules is presented. The next two chapters clarify the topic essential for each developer: the representation and sharing of the data. In the rest chapters, the implementation of the individual functionalities is described. Each chapter corresponds to one to three modules of the application.

7.1 The NetBeans Platform

The NetBeans Platform is a framework that is used for development of the NetBeans IDE. Because it is modular, unneeded modules can be removed and an arbitrary desktop application can be built on top of it. The graphics is drawn mainly by Swing, but when designing new user interface elements, more modern technologies such as JavaFX and even HTML are available. We do not lose portability, Windows, Mac OS X, Linux and Solaris are supported.

The main benefit of using the NetBeans Platform is that the developer saves time programming features that are common for most desktop applications such as the window system, toolbar buttons or mechanism for sharing data between components. Furthermore, the framework is time-tested and helps to keep the application maintainable.

The main drawback is that the architecture is quite complex and the user has to adopt its philosophy. In our case, this could be an obstacle for a developer who would like to extend our application. To prevent that, the techniques that could potentially be needed are explained in this chapter.

7.1.1 Modules

A module is a primary building block of a NetBeans Platform application. Each application has at least one module. A module consists of one or more Java packages and metadata files. Some packages can be public, which means that they are visible for other modules. If a module needs to call code from another module, it has to declare the dependency explicitly. All code could theoretically be in one module only, but the purpose of modules is to keep distinct features in different modules for clarity and maintainability of the code. We discuss the division of our application into modules in a separate section.

7.1.2 Top Components

Top Component is a window inside of the application graphical window system. The Top Component is based on the Swing JPanel. The position of the TopComponent depends on the group in which it is registered. The groups are called modes. Modes are named after the NetBeans IDE windows, for instance, the mode 'output' lies at the same location as the output window in the NetBeans IDE. New modes can be defined and users can, of course, move windows wherever they want. NetBeans IDE provides a wizard for creating of a new Top Component. In Drone Flight Inspector, every tool for data visualisation will be associated with one or two Top Components.

7.1.3 Files and File Editors

When a user opens a file in the NetBeans Platform application, a new editor is usually opened. Suitable editor is selected according to the extension of the file. From the developer's point of view, an editor is a Top Component. A tutorial on how to add support for a new file extension is included in the Attachment 3.

In Drone Flight Inspector, in place of the editor window we open a Top Component with a timeline that displays the distribution of the messages in the file.

7.1.4 Layers and Annotations

Each module can have one XML file called layer. The structure of XML tree mimics a filesystem with nested folders and files. At runtime, all layer files are merged into a single XML file, called system filesystem. Layer files contain a configuration of the application. Luckily, most entries are generated automatically from annotations, for example when registering open action for a Top Component or a new file type extension.

7.1.5 Concept of a Lookup

In the NetBeans Platform, lookup is a storage for objects. The owner of the lookup can fill it by objects and anyone who has access to it can search for the objects. It is possible to search only by a class or interface name. The lookup returns a list of objects of the given type. The strength of the lookup lies in the fact that it is able to report changes in the result for the given query.

An interesting subclass of the class *Lookup* is *AbstractLookup*, which we use if we want to be able to set the stored content directly. Other *Lookup* subclass we will use is *ProxyLookup* that redirects queries to some other lookup. The advantage is that the underlying lookup can be changed in the background. Another ability of the *ProxyLookup* is that it can merge multiple lookups into one.

Apart from the option to create our own lookup, there are two significant lookups directly in the framework. The first is the global lookup, which can be obtained by calling *Lookup.getDefault().* The global lookup contains instances of services that can be used throughout the application. The second is the global context lookup, accessible by calling a static method *Utilities.actionsGlobalContext().* This lookup provides us with objects that are associated with the currently selected element. The content of this lookup changes automatically whenever the user selects some other element, such as a Top Component, by clicking on it.

7.1.6 Services

In the previous section, availability of services from the global lookup was mentioned. We can understand the word service in this context simply as a class that can do something for us. If we want a new service to appear in the global lookup, all we have to do is to mark the class we want to place there with an annotation *ServiceProvider*. An instance of the class is created automatically at runtime and it is inserted into the global lookup.

Services are used on many places in Drone Flight Inspector because they enable us to use the Dependency Injection design pattern that brings us extensibility. Dependency injection means that some code can use a dependency (a service), without knowing where it came from. Someone else is responsible for injecting the dependency, in our case, putting the service into the global lookup.

As an example, if we have an interface *FileImporter*, we can get all file importer services by calling *Lookup.getDefault.lookupAll(FileImporter.class)*. Then we can choose an instance most suitable for importing of some file. Any developer can extend our application by creating a class that implements the *FileImporter* interface and marking it by the *ServiceProvider* annotation.

7.1.7 Including Java Libraries

Drone Flight Inspector uses several external libraries. In the NetBeans Platform, external libraries have to be wrapped in a module wrapper. A module wrapper contains only a JAR archive with the library. If we want to include the library from the code in another module, all we have to do is to set dependency on the wrapper module.

The other important NetBeans Platform concepts such as Actions and Nodes API will be explained in the following chapters when we will actually need them.

7.2 Division into Modules

The Drone Flight Inspector is split into 16 modules. A dependency graph with all modules is depicted in the *Figure 8*.

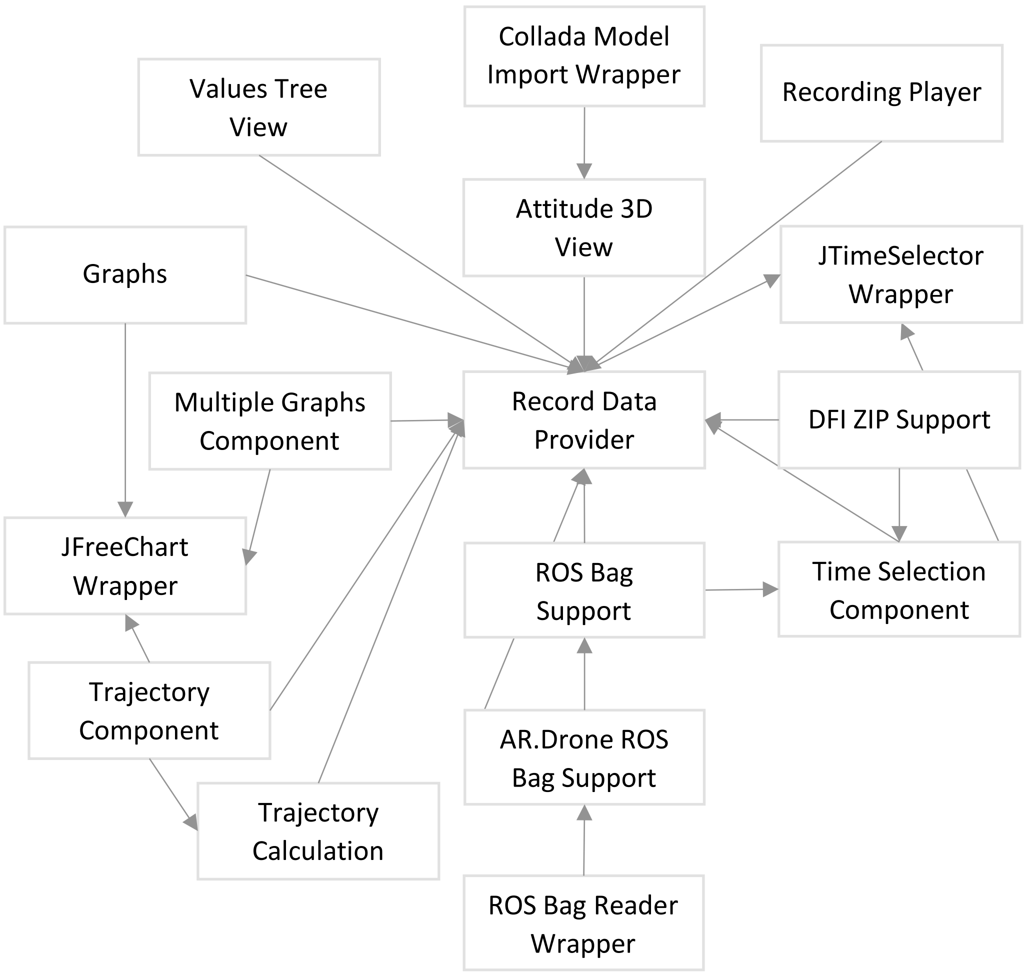


Figure 8 *Diagram of Drone Flight Inspector modules – arrows leading from the module X point towards modules on which the module X is dependent*

The Record Data Provider module forms the core of the application. It contains all the classes with data structures in which the loaded data are stored. The services that provide access to the loaded file data are also part of this module.

The other modules, excluding the library wrapper modules, can be divided into two categories, the modules that are responsible for loading of the recordings and modules that use the loaded data. It is worth noting that the modules that need the data are not directly dependent on the modules that load them. All the information is exchanged through the intermediary Record Data Provider module.

7.3 Data Sharing

In this subchapter, we study how data are delivered to components that need them. When introducing the NetBeans Platform, the lookup was described as a storage for objects. It seems a suitable solution to have a lookup that serves as a central storage for all data. The code that reads the file can fill the lookup with the file content and the visualisation components can ask the lookup for the data they need.

7.3.1 Issues with Multiple Opened Files

The NetBeans supports the multiple-document interface, which means that the data that should be visualised may change as the user opens or selects another file. As we need to change the content of the lookup according to selection, it sounds reasonable to use the global action lookup, which always contains the data that are associated with the current selection.

Each Top Component has associated lookup and we could put the file data into the lookup of the Top Component that represents the opened file. The content is propagated to the global context lookup whenever the component is selected. This works unless the user selects some Top Component that does not represent any file. Because selection of any visualisation tool would lead to removal of the file data from the global context lookup, this solution is unsuitable. Therefore, a new lookup called *FileLookup* has been created.

7.3.2 File Lookup

File lookup is available through a *FileLookupProvider* service in the global lookup, but can be obtained in an easier way via a static method *FileLookup.getDefault().* In the background, each opened file has its own lookup. The file lookup is a proxy lookup that always provides the content of the currently selected recording.

The file lookup may serve not only as a source of the recording data but also as a medium for communication between components. When we want to update the content of the File Lookup, we can get it as a return value from the method *FileLookupProvider.getSelectedFileLookupContent()*.

7.3.3 Avoiding Problems With Multithreading

When working with the file lookup, there is a risk that its content might suddenly change due to switching of the selected file performed on another thread. If we want to keep access to the content of the currently selected file, we can get directly the lookup that is associated with the selected file from file information class *FileLookupProvider.getSelectedFile()*.

7.4 Data Model

The choice of suitable data structures for storing the recording in the memory, which will be referred to as ‘the data model’ of the application, is essential for reaching both acceptable performance and maintainable code.

First, we examine the structure of the recorded data by example. From board of the drone, we receive information such as the current acceleration in the direction of the x-axis. We get this information regularly, so in the recording we have a series of acceleration measures. Each value can be associated with time in which it was measured, sent or received. For a vector of measured values, we therefore have one or more vectors of time values. Naturally, the acceleration in the x-axis will not be sent alone, but it will be accompanied with acceleration in y-axis and z-axis. Together, it forms a group of three acceleration series. The acceleration might be sent separately, but it might as well be combined with other measured values.

Next, let us generalise. The recording consists of groups. Each element of the group is either another group or a series of values. A vector of time values has to be attached to each group and series. A nested group or series shares the time stamps with its parent group.

This model of a series tree is reflected in the code.

7.4.1 Series Tree

In the implementation, a group of series is considered only a special type of series. The *SeriesWrapper* interface enables us to traverse a series tree by asking each series on its children or parent. The *SeriesGroupWrapper* class is an implementation of the *SeriesWrapper* interface that represents a group of series. A single series is wrapped in a class that implements the *SingleSeriesWrapper* interface. Many subclasses exist to provide additional features with respect to the series actual content type. For example, *DoubleQuantity* class encapsulates a series of values of type *double*.

When a file is opened, series trees are constructed and all series are put into the lookup associated with the new file, regardless of whether they are elements of a bigger group or not. Thanks to that, it is possible to find any series of a given type effortlessly by querying the File Lookup. For example, to get all series that can be plotted (which means their values can be converted to doubles) it suffices to write: *FileLookup.getDefaut().lookupAll(GraphableQuantity.class)*[1].

In order to provide an easier access to commonly used groups, special subclasses of *SeriesGroupWrapper* are used, such as *AccelerationWrapper*, which is a group that, by convention, contains three double quantities for acceleration in x-axis, y-axis and z-axis. Similarly, there are *MagnetometerWrapper*, *RotationWrapper* and *MotorsWrapper* that contains one *DoubleQuantity* with motor powers for each motor the drone has.

A final note about implementation relates to the Java data structures. Arrays are used instead of lists to save memory. Most of the data in the recording are primitive value types that cannot be stored in generic *List* without being boxed[2] and the boxed values take more than twice as much space as the original data.

7.4.2 Values Tree

The built-in global action lookup provides access to the current selection, which enables other components to react, for example by showing details about the selected element. Our own lookup works in the same way. When the user selects time, the Time Selection Component inserts the selected values into the file lookup.

Each series receives the selected time value and returns a *ValueWrapper* that wraps the value in the series with a timestamp nearest to the given time. The *ValueWrapper* is again only an interface and specific series return specific value wrappers. If a series is, in fact, a group of series, the resulting value wrapper wraps a group of *ValueWrappers*. As a result, we can construct a values tree that corresponds to the original series tree.

[1] In practice, we usually need to listen for changes so we use *FileLookup.getDefault().lookupResult(GraphableQuantity.class).addLookupListener(…)*, instead.

[2]Boxing is a process of wrapping a primitive value type into object so that it can be used in contexts in which a reference type is needed.

7.4.3 Time Stamps

It is not sufficient to attach an array of time values to the series, because we also have to know the semantics of the numbers. In Drone Flight Inspector, the values are encapsulated in the *TimeStampArray* class. *TimeStampArray* contains also the information whether the message was incoming or outgoing (from the perspective of the device that records the flight) and where the time information was recorded. There are two possible sources. The time stamp added on board the drone is considered more precise. However, it does not have to be always available, especially for the outgoing data such as commands that are sent to the drone. For that reason, the application works also with the timestamps included by the recorder device the moment the message was received. On-board time stamps are used when the precision is important, such as when calculating the drone's trajectory. Recorder time stamps are used otherwise.

7.5 Time Selection

Time Selection Component lets the user to select time, publishes the selected values and enables other components to set selected time from the outside.

The graphical component *JTimeSelector* that draws the timeline is implemented as a separate Swing panel and is included to DroneFlightInspector as an external library. Its interface consists of methods for adding and removing layers, adding and removing change listeners and setting the selected time. A layer is given by a name and a set of time values and it corresponds to one row in the component. Each time value is marked by a point on the timeline. Time values are internally stored as long integers. In the constructor, *JTimeSelector* accepts a function that converts the values into a human-readable string format.

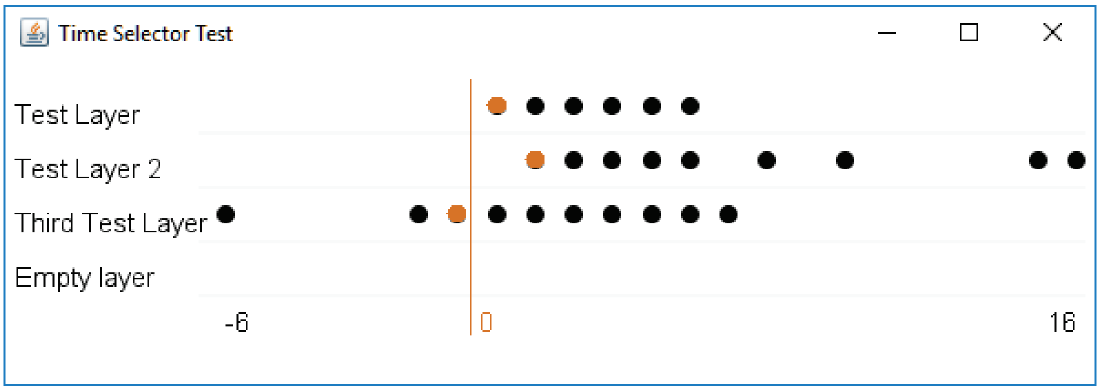


Figure 9 *JTimeSelector with four layers*

JTimeSelector reacts to mouse clicks (selection of time value), mouse dragging (selection of time interval), mouse wheel actions (zoom in or zoom out) and mouse dragging with pressed Control key (scrolling over the timeline, works if the timeline is zoomed in). JTimeSelector with selected time value is displayed in the Figure 9.

Time Selection Component listens for selection changes that are caused by a user interaction and on each change, it rebuilds the values tree. Old values tree is removed from the lookup and a new one is inserted.

AbstractLookup does not have the ability to add or remove multiple elements at once, unless we are willing to replace all its content. Tree nodes (*ValueWrappers*) are therefore removed and inserted one by one. To enable optimisations, the Time Selection Component owns an instance of *ValuesTreeConsistent* class and it keeps it in the lookup of the file throughout the time it does not make changes to the values tree. As a result, components that use multiple values from the values tree can delay their update until an instance of *ValuesTreeConsistent* is present in the lookup, saving potentially dozens of needless updates that would be performed if the component responded on every single change of the values tree.

Apart from *ValueWrappers,* the Time Selection Component inserts an instance of *TimeValueSelection* or *TimeIntervalSelection* class into the lookup so that anyone can get the information about currently selected time or time interval. However, even if some other part of the application wants to change the selected time, it must not remove or replace these instances.

In theory, The Time Selection Component could respond to the change of *TimeValueSelection* or *TimeIntervalSelection* by updating the Swing component and the selected values, but the problem occurs when multiple threads try to change time value concurrently. Such situation could lead to an erroneous state of having multiple *TimeSelection* instances in the lookup.

A correct way to change the selected time from the outside is to issue a *SetTimeRequest,* which means to put an instance of the *SetTimeRequest* class into the lookup. The Time Selection Component processes and removes the requests from the lookup sequentially.

7.6 Data Loading

Two new file types are registered and the action that is called whenever a new file is opened creates a new Time Selection Component and passes it a proper instance of *FileImporter* interface. The process of registration of a new file type is described in the tutorial in the Section 0.

The Time Selection Component runs the *FileImporter* during its initialization phase. The data loading is executed asynchronously in order not to block the user interface thread. In the Netbeans Platform, Request Processors are used to perform asynchronous tasks in a thread pool. We have a separate Request Processor that organizes the loading of recordings and limits the number of files that can be loaded at once. Opening of many files simultaneously would cause the process to be slower instead of faster, because sequential disk file read is more efficient than random access.

When the application is opened, it automatically reloads all the files that were opened the last time. This works effortlessly thanks to the capability of Top Components to save data when it is closed and load them when it is reopened. This feature is applied using the *TopComponent.Description* annotation and methods *writeProperties* and *readProperties*. The Time Selection component saves the path of the file and the name of the *FileImporter* that was used to load it. After the restart, it finds the matching *FileImporter* among the services registered in the global lookup and uses it to read the file.

7.6.1 Import from ROS

As there have not been any Java library for parsing ROS Bag files available[1], such library had to be designed. The resulting library, named RosbagReader, is simple and supports only the features required to load the flight data. The data in the Bag file must not be compressed. The interface of the library follows the visitor design pattern: the library reads the binary file until it reaches a message, then it invokes the user code to process the message. The user code can decide how to handle the message according to its topic. RosbagReader provides an interface for getting fields of different data types such as integral and floating-point types, strings or arrays from the content of the message without having to consider low-level details such as that the endianness used by ROS Bag is the opposite to that of Java.

[1] In the meantime, a library for reading ROS Bag files in Java has been released by P. J. Reed from Southwest Research Institute [25].

The actual implementation of the *FileImporter* interface that parses the recording of the AR.Drone flight is straightforward. There are two relevant topics in the recording: 'ardrone/navdata' that contains the state information sent from the drone and 'cmd_vel' with the commands sent to the drone. Their structure is described in the ardrone_autonomy documentation [22].

7.6.2 Import from DFI File Type

Drone Flight Recorder file is an archive that consists of an XML description file 'description.xml' and several text files with values separated by commas or tabs. We can imagine these files as tables, in which each row corresponds to one record and the labels of the table columns are located in the description file.

The description file contains one XML element for each file. Inside of the file element, there is an XML element for each column in the file. The column elements have different names according to the data type of the values in the column and attributes that describe the quantity in the column. In addition to that, adjacent column elements can be joined in a group. A group is an XML element characterised by one attribute only, its name. Naturally, groups can be nested. Each data file has to have at least one column with time values.

When converting the structure to the data model of the application, each file is represented as a group. The columns in the files are translated into series; the column with time values is used as their timestamps. The arrangement of the series in subgroups corresponds to the structure of the groups in the XML description.

The next step is to find an efficient way of loading the data from comma or tab separated files without knowing their structure in advance. The most efficient would be to generate the bytecode for reading the file according to the XML description and dynamically load and execute it. This option has been rejected because of its excessive complexity compared to the importance of the problem. Another possibility is to use some existing library for reading CSV / TSV files. However, the available libraries do not seem to bring any benefit (except a bit less work for the programmer), more likely a worse performance because of being designed for situations that are more complicated.

The implemented solution is based on virtual method calls and it appears to be sufficiently fast. We first read the description of the file structure and create an array of instances implementing *ValueReader* interface with a single *ValueReader* for each column. *ValueReader* is an interface that provides a method for reading of a value from a file. Each data type has its own implementation of the *ValueReader* interface. The *ValueReader* keeps the read values in its own data structure. When the file is opened, the readers are called in a cycle, while there are data available. A one line is read in a single iteration.

Classes that implement the *SeriesConstructor* interface take the values read by the *ValueReaders* and create appropriate *SeriesWrappers*. A suitable *SeriesConstructor* is selected according to the name of the XML element that describes the column.

Since each series has different properties such as name and unit, a new *SeriesConstructor* has to be created for each series. The instances of *SeriesConstructors* are supplied by classes that implement the *ConstructorProvider* interface.

To summarize the process, the XML parser reads an element and if it is not a group, it finds a *ConstructorProvider* that is associated with the element name. The *ConstructorProvider* creates an instance of *SeriesConstructor*, which, in turn, provides a *ValuesReader* that can read the values in a column of the data file. Once the data file is read, the *SeriesConstructor* obtains the data from *ValuesReader* and constructs a *SeriesWrapper*.

The importance of the algorithm lies in the fact that *SeriesConstructor* instances are services loaded from the global lookup. It is therefore possible to add a support for a new content type without having to touch the algorithm code, only by creating a new class that extends *SeriesConstructor* and registering it as a service.

The XML elements distinguish the series only by their data type, for example, we have elements named *intQuantity* and *doubleQuantity* for integral and floating-point values. However, there are certain quantities that have a specific semantics and are often used, such as the altitude of the drone. In the data model, these should be wrapped into the special series wrappers that subclass the general series type. In order to use the correct wrappers, the series constructor first takes the name attribute of the quantity and tries to match it with one of the special quantity providers. If it succeeds, it leaves the special quantity provider to generate the *SeriesWrapper*; otherwise, it generates a generic *SeriesWrapper*. The same procedure is done even for *SeriesGroupWrappers*.

The special providers are also registered as services and thus it is possible to add a new special series easily without having to touch the DFI ZIP Support module in which the code is located. Moreover, this, in contrast to adding a different data type, can be done without having to extend the XML syntax by additional elements. The available special provider interfaces that can be registered as services are named *SpecialDoubleQuantProvider* and *SpecialGroupProvider*. The currently supported names with special meanings are 'Acceleration', 'Altitude', 'Battery level', 'Magnetometer readings', 'Motor powers', 'Rotation' and 'Velocity'.

7.7 Tree View

When the user select a time, we would like to display all the values with time stamps close to the selected time. Our Time Selection Component provides us the values as a values tree. Therefore, our task is to visualise the values tree.

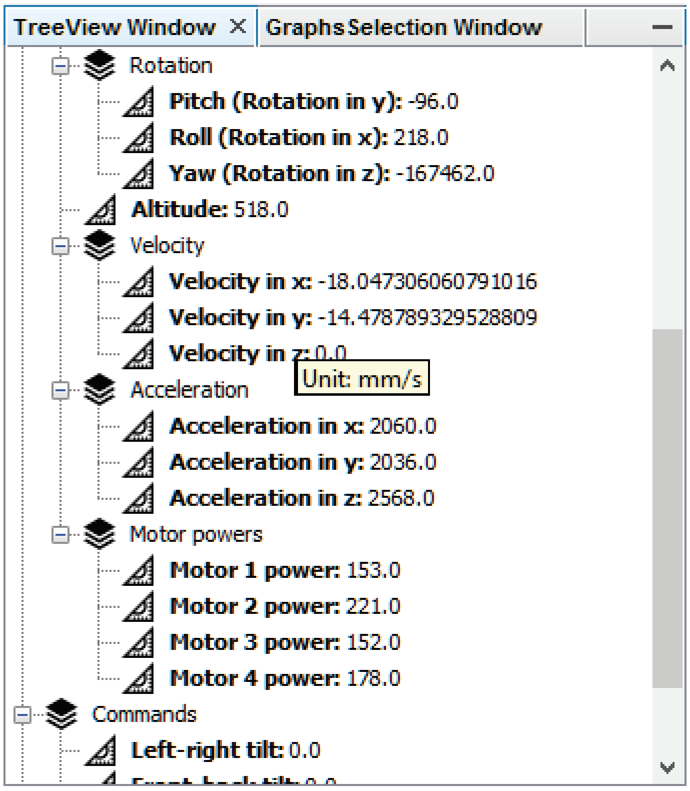
The NetBeans Platform contains components that we could use. Nevertheless, they all use the Nodes API.

7.7.1 Nodes API

The creators of NetBeans have designed Nodes API in order to have a unified data representation usable by different graphical visualisation components. It is a model of the data the NetBeans user interface components can work with. It belongs between our data model and the visualisation component, to the so-called presentation layer of our application. The nodes contain the data in a form that is closer to that what is displayed on screen. The elements (the nodes) have associated icons, description that is shown as a tooltip when the user hovers with the mouse over the element, as well as actions that can be performed with the element.

Each node can have child nodes so the hierarchy of nodes forms a tree similarly to our hierarchy of values. We assign each node which is not a leaf node a child factory that is used by the NetBeans Platform framework to generate the children when they are needed. There are mechanisms which enable children to be generated asynchronously or not all at once if it is a time-consuming operation. However, we will not use these features.

We create an invisible virtual root node with a child factory that generates nodes for all root *ValueWrappers*. We get all value wrappers from the file lookup and we skip those, which have parent. Each created node is then declared a leaf node if the *ValueWrapper* it represents has no children. Otherwise, it gets a child factory that generates nodes for its children. We assign the name of the *ValueWrapper* into the display name attribute of the generated nodes. If the node is a leaf node and the *ValueWrapper* contains a value convertible to string, we also append the wrapped value.



```
TreeView Window ×   GraphsSelection Window
    Rotation
        Pitch (Rotation in y): -96.0
        Roll (Rotation in x): 218.0
        Yaw (Rotation in z): -167462.0
    Altitude: 518.0
    Velocity
        Velocity in x: -18.047306060791016
        Velocity in y: -14.478789329528809
        Velocity in z: 0.0
                      Unit: mm/s
    Acceleration
        Acceleration in x: 2060.0
        Acceleration in y: 2036.0
        Acceleration in z: 2568.0
    Motor powers
        Motor 1 power: 153.0
        Motor 2 power: 221.0
        Motor 3 power: 152.0
        Motor 4 power: 178.0
Commands
    Left-right tilt: 0.0
```

Figure 10 *The top component with Bean Tree View*

Finally, we have a model of data that we can display. NetBeans contain many components called views that we could use to actually show the data to the user. As we want to maintain the tree structure of groups and subgroups, we use a tree view, specifically the Bean Tree View, shown in the Figure 10.

We create a top component and place the Bean Tree View Swing component on its panel. It remains to interconnect the data with the view. The *ExplorerManager* class is used for this. The parent component of the view, in our case the top component, implements *ExplorerManager.Provider* interface through which it provides its explorer manager. The root node is passed to the explorer manager. The Bean Tree View automatically finds the *ExplorerManager* in the hierarchy of its parents and displays the tree.

The root node listens for changes in the file lookup and if the values are replaced due to either change of time selection or selection of a different file, it updates its children. Originally, a new nodes tree had always been constructed. However, it caused slowness during the replay mode in which the values are updated about twenty times per second. The reason is probably the tree view is designed to respond on user interaction only and is not prepared for such frequent updates. To avoid this issue, before recreating the nodes hierarchy we first check if the structure of the tree does not remain the same. If it does, we only update the display names of the old nodes. We may notice that the structure of the tree within a single file stays always the same, so the tree has to be reconstructed completely only if the user switches to another file. This suffices to make the updates smooth.

The tree view also utilises the optimisation introduced in the chapter about Time Selection Component. Instead of reacting on changes of *ValueWrappers*, it tracks the presence of *ValuesTreeConsistent* object and the update is performed only if the values tree is consistent.

7.8 Plotting Graphs

Drone Flight Inspector contains a set of simple plots that display the common quantities, such as altitude or battery level and then there is the Multiple Graphs Window with a single canvas, which can be used to plot an arbitrary quantity or even multiple quantities at the same time. The Multiple Graphs Window also offers features to scale and translate the plotted values.

7.8.1 jFreeChart Library

In order to display the charts in a convenient user interface with less effort, an external graph-drawing library has been employed. The library jFreeChart [23] has been chosen because of its popularity and therefore availability of large number of sources. It brings many capabilities and customization options at the expense of performance and more problematic reaching of specific needs.

An important concept of jFreeChart is the *Dataset*. It is a bridge between our data and the graphing component. The charts use it to obtain information on how many series there are to be drawn and to get the actual x and y coordinates. We create our own implementation by extending *AbstractXYDataset* for all the plots in Drone Flight Inspector.

7.8.2 Simple Graphs

The windows that display static graphs are based on a single top component with a Swing jFreeChart panel inside it. The top components track their respective quantities in the lookup and update their dataset if a change occurs. The graphs display quantities that have origin on board the drone (altitude, velocity, motor powers and battery percentage) and thus the on-board time information is likely available. The on-board time is plotted on the x-axis.

7.8.3 Customizable Graphs Panel

Apart from the Multiple Graphs Window that displays the graphs, there needs to be a second window in which the user selects the graphs to plot. These windows needs to communicate. The selection window has to pass the graphs and their customization options to the plotting window. We use a tested communication mechanism – the file lookup. The Graphs Selection Window shares a *GraphedQuantity* object for each quantity to plot. The *GraphedQuantity* object contains the quantity to plot, the desired translation and scale in x-axis and y-axis as well as the colour of the line. We set the colour explicitly in order to avoid confusing the user by rearranging the colours of lines whenever some line is removed from the plot.

The Multiple Graphs Window listens for changes of *GraphedQuantities* in the lookup and updates its dataset accordingly. Furthermore, it is also notified by individual *GraphedQuantities* about changes in their properties, because the user may change only a single property of the quantity such as its translation or line colour. This is done in a standard way using the observer pattern: each *GraphedQuantity* maintains a set of listeners it informs about changes. The dataset associated with the jFreeChart panel in the Multiple Graphs Window keeps the current list of *GraphedQuantities* and applies the translation and scale on each point before passing it to the graphing tool.

As opposed to the static graphs windows, an arbitrary quantity can be plotted in the Multiple Graphs Window and so we can make no assumptions about availability of the on-board timestamps. Due to that, the recorder time values are plotted on the x-axis.

Next, we need to design the Graphs Selection Window. It has to display a list of quantities that can be graphed. We get the list from the file lookup and use some NetBeans component to display it. As we need to be able to select or deselect quantities, the *OutlineView* has been chosen, as it is capable of adding checkboxes to elements.

To get the data for the component to display, we employ the Nodes API again. This time, our hierarchy is flat. We create a single artificial root node and let it generate a list of all quantities as its children.

The NetBeans platform simplifies us also the creation of the user interface for editing the properties of the graphs with its *PropertySheetView* component. Our nodes override the method *createSheet* which returns the set of properties that can be changed. When we place *PropertySheetView* and *OutlineView* into the same Top Component they share the common *ExplorerManager.* As a result, the *ProperySheetView* starts to display the properties of the node selected in the *OutlineView* automatically and lets the user to edit them. When the user confirms a change of a property, the *PropertySheetView* propagates the new value into the original data structure by invoking the setter method using reflection.

7.8.4 Graph Markers

The jFreeChart library enables us to add markers in the graphs. We use this in the Multiple Graphs Component to highlight the part of the graph that corresponds to the selection in the Time Selection Component. If a single time value is selected, a vertical line is drawn in the plot using the *ValueMarker* class. If an interval is selected, the same interval is highlighted in the graph using the *IntervalMarker* class.

7.9 Attitude Component

The Attitude Component displays a 3D view with a model of AR.Drone. The pose of the drone is set according to the real attitude of the drone in the time selected on the timeline.

A decision had to be made, which technology for displaying 3D graphics in Java to use. There is a surprisingly large number of competing projects. It would be better to avoid using too low-level libraries that only wrap OpenGL calls (LWJGL[1], JOGL[2]) but also extensive game frameworks like jMonkeyEngine that have many features we do not need. Another requirement was that the technology should support loading of a textured model from a file. Furthermore, it had to be possible to incorporate it into the NetBeans Swing window system. In the end, the 3D capabilities of JavaFX has been selected, which seemed to be adequately complex. An advantage of JavaFX is that it is a part of Java and so it is not necessary to add a new dependency.

7.9.1 JavaFX and NetBeans Platform

Despite NetBeans has been created a long time before JavaFX, currently it is possible to extend it with JavaFX content. The integration is done through the Swing component *JFXPanel* that is inserted into the Swing window. Inside of the *JFXPanel*, JavaFX is used.

7.9.2 3D in JavaFX

We set the content of the *JFXPanel* by passing it a *Scene* object. The Scene contains a scene graph, which is a tree data structure. Vertices of the tree are called nodes. The leaf nodes of the scene graph are the objects that form our scene. In our case it is the model of the drone and an ambient light. We put the objects into groups that are represented by *Group* nodes. The Group node without parent is passed to the Scene as the root node of the scene graph. The translation, rotation or shear transformations can be applied on the nodes. We use the rotation transformation to rotate the group with the drone model according to the rotation values found in the file lookup. Finally, to display the scene properly we set a camera object for the *Scene*. We use an instance of the *PerspectiveCamera* class.

[1] Lightweight Java Game Library
[2] Java Binding for the OpenGL API

7.9.3 Loading of the Model

JavaFX lacks the support for loading of meshes from files. Luckily, there exists a set of model importers from Interactive Mesh [24]. Although the libraries are not open source, they are free to use. Drone Flight Inspector includes the library for loading models in the COLLADA[1] format.

7.10 Trajectory component

We have two trajectory related modules, the one that computes the trajectory and the second that displays it. The principles of the trajectory calculation have been discussed in a separate chapter. In this chapter, its integration into the system of our application is described.

The top component that displays the trajectory is simple; it tracks the changes of the trajectory series in the lookup and displays the set of point in a jFreeChart scatter plot. The component is prepared for the situation that there could be multiple trajectories calculated by different algorithms. The *PointSeriesDataset* provides the graphing component with multiple series of values, if they are available.

7.10.1 Data Loaded Response

As we want to have components separated, we do not want the Time Selection Component to know anything about the trajectory calculation. Despite that, we need it to add the trajectory into the lookup after it loads a file. To maintain the separation and to add another extensibility point, the *DataLoadedCallback* interface has been introduced. After the Time Selection Component loads the file, it finds all services that implement *DataLoadedCallback* interface and invokes them. It is therefore possible to add a code that should do some transformation on every newly opened file to any module.

7.10.2 Lazy evaluation

It would be possible to calculate the trajectory immediately after a file is loaded. However, a bit more efficient would be to delay the calculation and do it on demand, when some component requires it. This behaviour is supported by lookups. Instead of the trajectory itself, we place into the lookup a proxy object and a converter. The first time someone requires the trajectory from the lookup, the lookup automatically uses the converter to compute the trajectory based on the data in the proxy object. The class *TrajectoryDelegate* that implements the *InstanceContent.Convertor* interface does the conversion from the *TrajectoryDelegateData* proxy object to an instance of *Trajectory* class.

7.11 Replaying the recording

When the user presses the Play button, a new thread is started and it periodically updates the currently selected time until the end of the recording or the end of the interval selection is reached. It changes the time by issuing the *SetTimeRequests*. Similarly, there is the Stop button that interrupts the playback thread by changing its running state *Boolean* variable and the Rewind button, which issues a *SetTimeRequest* to move the time selection to the beginning of the recording.

7.11.1 NetBeans Platform Actions and Cookies

We place the buttons to the NetBeans toolbar, which is done via actions. An action is a class that implements the *ActionListener* interface and it needs to be registered, which can be done using annotations. By adding annotation, we can also set the icon of the action and

[1] Collaborative Design Activity – a standardized 3D data file format originally created by Sony Interactive Entertainment. Apart from the 3D models it can contain also textures and other data.

display it in the toolbar or in the application menu. After clicking on the toolbar button, the method *actionPerformed* is invoked.

Our buttons should not be enabled under all conditions. The play and rewind button should be active only if we have an opened recording and we are not currently playing it. The stop button should be clickable only while the recording is being played. In the NetBeans Platform, the so-called cookies are used to create conditionally enabled actions. The cookie is a class that is bound to some action. If there is an instance of the cookie class in the global action lookup, then the action is enabled, otherwise it is disabled. When the button is clicked, the action class is instantiated and it receives the instance of the cookie. The idea is that the action itself does not have to know how to do the task but it uses the cookie to perform the desired action according to the context.

However, the problem is, that the objects associated with the currently opened recording are accessible through the file lookup but not through the global action lookup. In order to be able to put our cookies to the lookups of the files and the buttons worked the global action lookup had to be replaced to include the content of the file lookup. Fortunately, it was relatively easy to do. The global action lookup is in the background gained from the *ContextGlobalProvider* service. A different class implementing *ContextGlobalProvider* has been created, registered as a service and the priority of the service has been increased by changing the optional argument '*position'* of the *ServiceProvider* annotation. The implementation of *ContextGlobalProvider* mimics the original behaviour but merges its content with the file lookup content by using *a ProxyLookup.*

When a new file is opened, we insert cookies for the play and rewind actions (*RecordingPlayer* and *RewindCookie* classes) into the lookup of the file. If the file is selected, the cookies appear in the global action lookup and the play and rewind action are enabled. *RecordingPlayer* adds the *StopCookie* to the lookup of the file when it starts to play the recording and removes it afterwards.

7.12 yaDrone Recorder

The last section of the implementation chapter briefly introduces the changes that have been made in the yaDrone library to allow data recording. The work with the resulting program is described in the tutorial in the Attachment 2.

The yaDrone library handles the Wi-Fi communication with the AR.Drone 2.0 over four channels, navdata[1] and commands channels on UDP ports and video and configuration channels on TCP ports. The communication protocol: the format of the interchanged messages, the initialisation sequence and other rules of data transmission are implemented according to the AR.Drone Developers Guide and the sample C++ library ARDroneLib. Both these sources are available as a part of the Parrot AR.Drone SDK 2.0.

The classes *CommandsManager* and *NavDataManager* have been altered so that they have the ability to register listeners that want to be notified of all incoming navdata messages and outgoing command messages. Then the Recorder class that intercept the messages and output them to a data stream in the CSV format has been created.

The next task was to provide an option to record the data from the graphical user interface of the application yaDrone ControlCenter. The Control Center uses the Reflections library to load plugins dynamically. On runtime, all instances of *ICCPlugin* are found and instantiated

[1] Navdata are, in the terminology of the AR.Drone, the sensor data and the state information which the drone sends to the controller device

by reflection. A new *ICCPlugin* with a Swing component has therefore been created that lets the user to start recoding and save the recording to a file. The XML description file is added to the recording and the files are optionally packed into a zip archive so that the recording can be directly opened in Drone Flight Inspector.

8 Conclusion

8.1 Results Discussion

The foundations of an application for flight data analysis have been laid, several visualisation tools have been developed, and we should answer the question, whether all our requirements have been fulfilled.

One of the main requirements was the extensibility of the application. This goal has been reached by means of the NetBeans Platform framework, which is inherently extensible, and the new DFI file type that is not bound to a certain drone model.

Particularly, the ability to add a support for a new drone model has been mentioned. The easiest solution is to utilise the ability of the DFI file type to load dynamic data. It suffices to save the data as comma-separated values and create an XML description. If we already have the data recorded in a different file format, it is likely that a way of converting the data to CSV exists, because CSV is an unofficial standard used by many applications. If it does not exist or we want to open the file in Drone Flight Inspector directly, we can add a new file type support programmatically, which is described in the Attachment 3.

Furthermore, it should be possible to extend the application by new visualisation tools. This can be done by creating new modules. The module can access all data via the *FileLookup*; it only needs to add a dependency on the Record Data Provider module. If a new window is created in the module, it automatically appears in Drone Flight Inspector after the module is compiled and launched. Also, there is the option to register quantities with special meaning, as described in the Subchapter 7.6.2. For example, if a programmer decides to add a window that displays a map with the GPS position of the drone, the latitude and longitude could be registered as special quantities, which would make them easily accessible from the *FileLookup*.

The criterion of the quality of every application is its performance and responsiveness of the user interface. Drone Flight Inspector is capable of managing tens of simultaneously opened recordings without problems. However, for the large recordings, the redrawing of the plots becomes slow and as it is performed on the user interface thread, it affects the user experience. If there was a need to use the application with very big amounts of data, the graphing library would probably have to be replaced.

Finally, there is the question whether the application fulfils its purpose of visualisation and analysis of the data sufficiently. The answer depends on the demands of a particular user. The application currently covers the set of the basic data analysis tools, but it would naturally be possible to extend their capabilities further.

8.2 Design Decisions Evaluation

The NetBeans Platform framework contributes to the fulfilment of multiple requirements we had. On the other hand, its utilisation has slowed the development process rapidly, because it enforces specific implementation techniques. However, apart from the extensibility benefits, we would not be able to develop such a configurable and comfortable user interface without the use of an external window framework. Therefore, the utilisation of the NetBeans Platform has been a reasonable choice. The potential programmers who would like to extend the application do not need to modify the advanced properties of the framework and the important patterns have been described in the implementation chapter.

While extending the yaDrone project, it has been necessary to make many modifications in order to remove bugs in various parts of the library and to improve the user experience, which has been time-consuming. Despite that, the use of the already existing framework has saved time and it could be beneficial even for the future users who will be able to use the additional features of the yaDrone project. The weakest part of the yaDrone-Recorder is the video decoding, which is not completely responsible. Nevertheless, yaDrone-Recorder provides a sufficient environment for recording the flight data.

8.3 Future Work

From the perspective of data visualisation, as we now have the orientation and trajectory visualised separately, the next step could be to join the information and perform a full 3D simulation of the flight. From the perspective of data analysis, the trajectory algorithm could be improved so that it utilises all available information, such as the data from the commands channel.

The current tools in Drone Flight Inspector concentrate on the processing of numerical data. However, drones are usually equipped with cameras and the video could also be analysed or at least displayed to the user. A thorough exploration would have to be done in order to find an efficient Java video decoding library. We need to replay the video file and seek in it as the user changes the time selection without affecting the overall performance of the application.

The user can examine the actions that the drone performs, such as whether it accelerates forward or rotates around the z-axis. Nevertheless, these actions are typically executed to reach more advanced goals, such as to fly from one point to another or to rotate by ninety degrees. It would be convenient if the application was able to detect these intentions and divide the recording into segments based on the tasks that the drone pursues. This functionality could be implemented using decision trees. The decision tree processes every message in the recording. The tree is traversed from the root node to a leaf node; leaf nodes represent the actions of the drone. On every node, the successor is selected according to the value of a single quantity. Drone Flight Inspector could include the tool that would interpret the trees and visualise the results.

9 Bibliography

[1] **Krajník, T., Vonásek, V. and Fišer, D.** *AR-Drone as a Platform for Robotic Research and Education.* Prague : Springer, 2011, pp. 172–186.

[2] **Universität Hamburg.** YADrone - Yet Another AR.Drone 2 Framework. *Verteilte Systeme und Informationssysteme (VSIS).* [Online] Universität Hamburg, 10 January 2014. [Cited: 30 June 2016.] https://vsis-www.informatik.uni-hamburg.de/oldServer/teaching/projects/yadrone/index.html.

[3] **Yoshida, Shigeo.** ARDroneForP5: Processing, You can control AR.Drone frome Processing -. *NaBi.* [Online] 21 June 2011. [Cited: 1 July 2016.] http://kougaku-navi.net/ARDroneForP5/index_en.html.

[4] **Codeminders.** GitHub - codeminders/javadrone: Java API and demo programs to control Parrot's AR.Drone. *GitHub.* [Online] 28 March 2015. [Cited: 1 July 2016.] https://github.com/codeminders/javadrone.

[5] **lab-drone.** lab-drone - Google Code Archive - Long-term storage for Google Code Project Hosting. *Google Code Archive.* [Online] 5 January 2013. [Cited: 1 July 1016.] https://code.google.com/archive/p/lab-drone/.

[6] **Mani Monajjemi and contributors.** AutonomyLab/ardrone_autonomy: ROS driver for Parrot AR-Drone 1.0 and 2.0 quadrocopters. *GitHub.* [Online] Autonomy Lab, Simon Fraser University, 2 April 2016. [Cited: 1 July 2016.] https://github.com/AutonomyLab/ardrone_autonomy.

[7] **Open Source Robotics Foundation.** *ROS.org | Powering the world's robots.* [Online] Open Source Robotics Foundation. [Cited: 1 July 2016.] http://www.ros.org/.

[8] **National Instruments.** LabVIEW System Design Software - National Instruments. *National Instruments: Test, Measurement, and Embedded Systems.* [Online] National Instruments. [Cited: 1 July 2016.] http://www.ni.com/labview/.

[9] **Brian, Lee.** AR Drone Control with Leap & Kinect using LabVIEW. *YouTube.* [Online] National Instruments, 15 October 2013. [Cited: 1 July 2016.] https://www.youtube.com/watch?v=qUKwS_tfG3s.

[10] **Geisendörfer, Felix.** felixge/node-ar-drone: A node.js client for controlling Parrot AR Drone 2.0 quad-copters. *GitHub.* [Online] 4 December 2014. [Cited: 2016 July 2.] https://github.com/felixge/node-ar-drone.

[11] **Nodecopter Core Team.** *The NodeCopter - Programming flying robots with node.js.* [Online] [Cited: 2 July 2016.]

[12] **Lloyd, Matthew.** - cucx/bebop: Parrot Bebop Drone Hacking. *GitHub.* [Online] 9 December 2014. [Cited: 18 July 2016.] https://github.com/cucx/bebop.

[13] **Open Source Robotics Foundation.** Bags/Format/2.0 - ROS Wiki. *ROS.org | Powering the world's robots.* [Online] Open Source Robotics Foundation, 6 September 2013. [Cited: 4 July 2016.] http://wiki.ros.org/Bags/Format/2.0.

[14] **Sly Technologies Inc.** jNetPcap OpenSource | Protocol Analysis SDK. [Online] Sly Technologies Inc., 7 August 2015. [Cited: 5 July 2016.] http://jnetpcap.com/.

[15] **Charles, Patrick.** Network Packet Capture Facility for Java download. *SourceForge.net.* [Online] Slashdot Media, 24 January 2013. [Cited: 5 July 2016.] https://sourceforge.net/projects/jpcap/.

[16] **Oracle Corporation.** NetBeans IDE - NetBeans Rich-Client Platform Development (RCP). *NetBeans IDE.* [Online] [Cited: 6 July 2016.] https://netbeans.org/features/platform/.

[17] **The Eclipse Foundation.** Platform - Eclipsepedia. *Eclipse.* [Online] The Eclipse Foundation, 24 March 2016. [Cited: 6 July 2016.] https://wiki.eclipse.org/Platform.

[18] **Tully Foote, Mike Purvis.** REP 103 -- Standard Units of Measure and Coordinate Conventions. *ROS.org | Powering the world's robots.* [Online] Open Source Robotics Foundation, 31 December 2014. [Cited: 21 July 2016.] http://www.ros.org/reps/rep-0103.html.

[19] **Matzner, Filip.** Tracking of 3D Movement. Prague : Charles University in Prague Faculty of Mathematics and Physics, 2014.

[20] **Maybeck, Peter S.** *Stochastic Models, Estimation and Control: Volume 1.* s.l. : Academic Press, 1979. 0124110428.

[21] **Engel, Jakob Julian.** Autonomous Camera-Based Navigation of a Quadrocopter. Munich : Technical University Munich, December 2011.

[22] **Monajjemi, Mani.** *ardrone_autonomy — indigo-devel documentation.* [Online] 25 April 2015. [Cited: 14 July 2016.] http://ardrone-autonomy.readthedocs.io.

[23] **Gilbert, David.** JFreeChart. *JFree.org.* [Online] Object Refinery Limited, 31 July 2014. [Cited: 18 July 2016.] http://www.jfree.org/jfreechart/.

[24] **Lammersdorf, August.** JavaFX 3D Model Importers. *InteractiveMesh.org.* [Online] InteractiveMesh e.K., 9 February 2014. [Cited: 2016 July 17.] http://www.interactivemesh.org/models/jfx3dimporter.html.

[25] **Reed, P. J.** swri-robotics/bag-reader-java: A pure Java library that can read and deserialize ROS bag files. *GitHub.* [Online] Southwest Research Institute, 20 June 2016. [Cited: 14 July 2016.] https://github.com/swri-robotics/bag-reader-java.

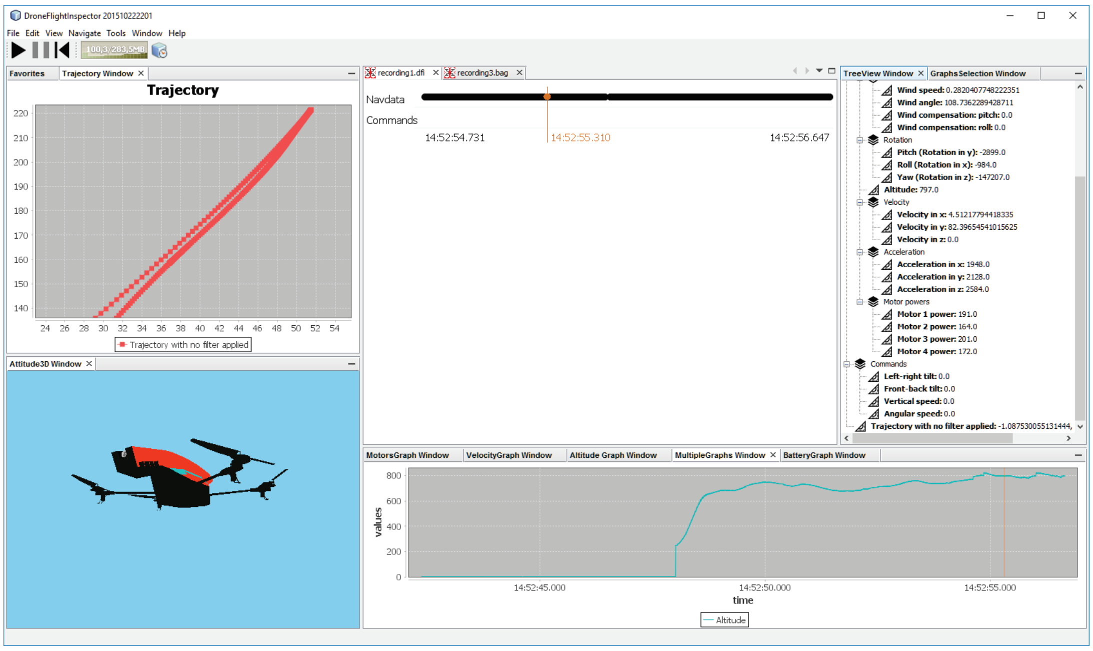
Attachments

Attachment 1

Drone Flight Inspector User Manual

Introduction

You can use Drone Flight Inspector to visualize and analyse data recorded during a drone flight.



```
Wind speed: 0.2820407748222351
Wind angle: 108.7362289428711
Wind compensation: pitch: 0.0
Wind compensation: roll: 0.0
Rotation
  Pitch (Rotation in y): -2899.0
  Roll (Rotation in x): -984.0
  Yaw (Rotation in z): -147207.0
Altitude: 797.0
Velocity
  Velocity in x: 4.51217794418335
  Velocity in y: 82.39654541015625
  Velocity in z: 0.0
Acceleration
  Acceleration in x: 1948.0
  Acceleration in y: 2128.0
  Acceleration in z: 2584.0
Motor powers
  Motor 1 power: 191.0
  Motor 2 power: 164.0
  Motor 3 power: 201.0
  Motor 4 power: 172.0
Commands
  Left-right tilt: 0.0
  Front-back tilt: 0.0
  Vertical speed: 0.0
  Angular speed: 0.0
Trajectory with no filter applied: -1.087530055131444,
```

How to Obtain Data

Drone Flight Inspector can open recordings that are saved in the Drone Flight Inspector file format with extensions '.dfi' or '.zip'. The main source of these files is the application yaDrone-Recorder. The application also supports the file format of AR.Drone flight recordings generated by ROS (the Robot Operating System).

DFI File Format

The application yaDrone-Recorder provides a graphical user interface in which AR.Drone 2.0 can be controlled by keyboard and the flight can be recorded. Please, follow the yaDrone-Recorder User Manual to learn how to record the data.

DFI file is a ZIP archive that contains a group of files with comma separated values (CSV files) and a description file. Advanced users can make virtually any CSV data readable by Drone Flight Inspector by providing them with an XML description and packing them into ZIP.

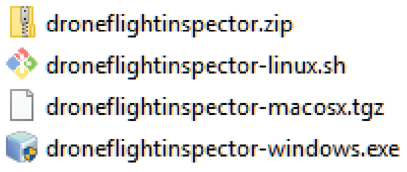
ROS Recordings

Users familiar with the Robot Operating System can use the ardrone_autonomy driver to control the AR.Drone and record flight data. Data are captured by the tool rosbag and produced files have the extension '.bag'. Rosbag categorizes messages based on their ROS topic. Two topics used by Drone Flight Inspector are ardrone/navdata and cmd_vel.

Installation

Drone Flight Inspector is a multiplatform application. The only requirement is a computer with Java SE Runtime Environment 8 or newer.

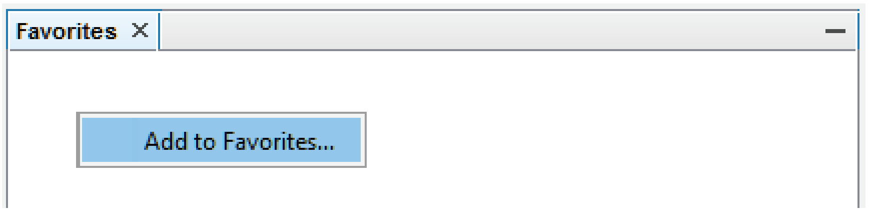
Select the installer suitable for your operating system. Alternatively, on Windows and Linux it is possible to manually extract the zip archive and launch the application from the subdirectory 'droneflightinspector/bin' without installation.


droneflightinspector.zip
droneflightinspector-linux.sh
droneflightinspector-macosx.tgz
droneflightinspector-windows.exe

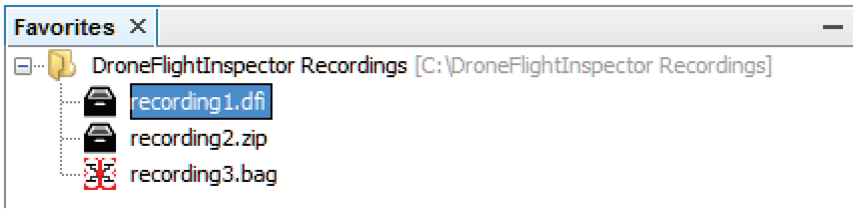
The installation wizard will guide you through the installation process.

Opening Recordings

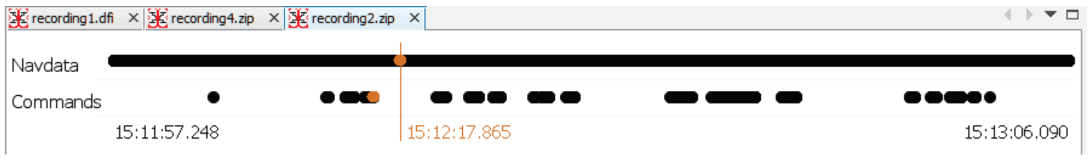
Run Drone Flight Inspector. Use Favourites window to open the recording. You should see the Favourites window on the left. Right-click into the window and choose 'Add to favourites'.



File chooser dialog opens. Select the folder that contains your recordings and confirm the selection.


Favorites
DroneFlightInspector Recordings [C:\DroneFlightInspector Recordings]
recording1.dfi
recording2.zip
recording3.bag

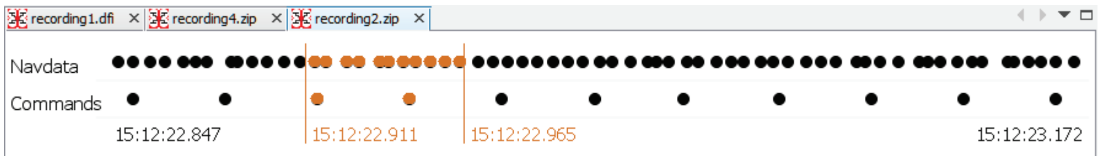
The selected folder will remain in the Favourites window even after a restart of the application. Open a recording by double-clicking on its name. Once the file is loaded, a timeline appears.



Controlling the Timeline

The timeline shows multiple data streams within separate layers. Every recorded message is displayed as a dot. You can select a time by clicking on the timeline. Nearest message is marked by an orange dot on each layer. When time is selected, other windows show details about the state of the drone in the selected time.

You can also select a time interval by dragging the mouse over the component. To zoom in or zoom out, scroll the mouse wheel. When zoomed in, move along the timeline by dragging the mouse with Control key pressed.

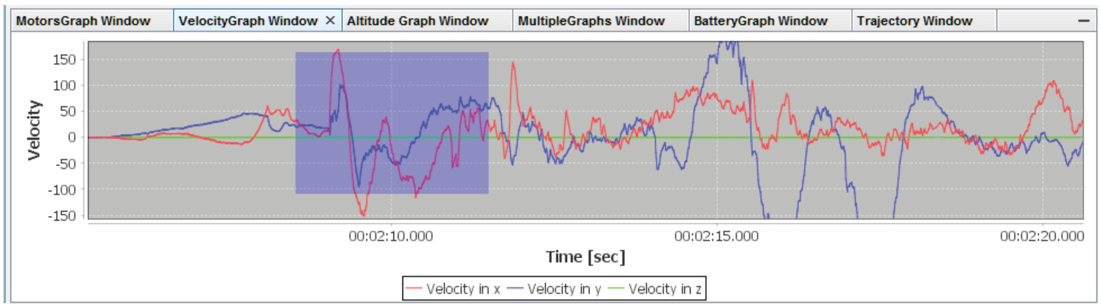


Window System

All windows can be repositioned by dragging their titles by the mouse. If a window is closed, it can be reopened from the drop-down menu 'Window' in the menu bar. From there it is also possible to reset all windows to their original positions. A window can be maximized by double-clicking on its title.

Plots and Trajectory View

Drone Flight Inspector contains several windows that display plots of measured quantities in time, namely Motors, Velocity, Altitude and Battery Graph Window. A Trajectory Window that displays the estimated trajectory of the drone is controlled in the same way. All these windows display the plots immediately providing that the required data are present in the recording.

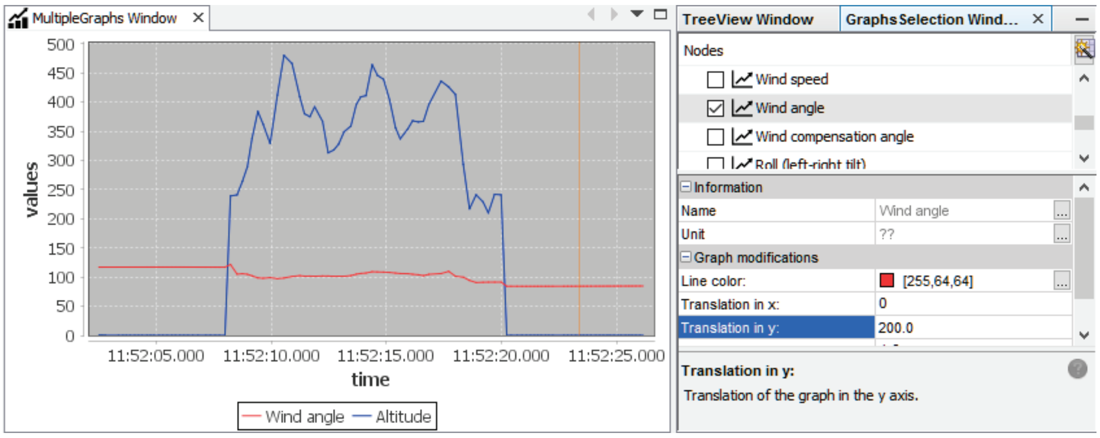


To zoom in, drag your mouse over the plot from right to left. A blue rectangle highlights the area to be zoomed in. To quickly return to the original view, drag your mouse from right to left. This action is similar to the swipe left gesture used on touch-screen devices. Moving along the plot can be achieved by dragging the mouse over it with Control key down. To display additional options, right-click on the panel.

Multiple Graphs Window

If you want to plot other recorded data than those which have their own panel or you want to compare multiple quantities, use Multiple Graphs Window.

Multiple Graphs Window is linked with the Graphs Selection Window. The Graphs Selection Window lists all quantities present in the recording that can be plotted. To actually plot the quantity, tick the check-box on the left of the quantity name. Because different quantities can have different scales, there is an option to transform the quantity by setting translation and scale in both axes.

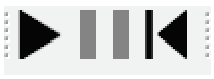


Information	
Name	Wind angle
Unit	??
Graph modifications	
Line color:	🟥 [255,64,64]
Translation in x:	0
Translation in y:	200.0

Translation in y:
Translation of the graph in the y axis.

Replaying the Recording

Use the toolbar buttons in the top left corner of the application to control the playback.



When play button is pressed, the recording starts to be played from the current time selection. If a time interval is selected, play button replays only the selected part of the recording. If not, the playing continues to the end of the recording. When the playback finishes, the original time selection is reselected.

Unsurprisingly, the middle button stops playing immediately and the rightmost button moves the selection cursor to the beginning of the recording.

Trajectory View

Trajectory is estimated based on the speed of the drone in x and y direction. The unit used on the trajectory plot is therefore dependent on the unit of the velocity values. To find out the unit of a quantity, either select it on the Graphs Selection Window, which displays details, or hover over the quantity name in the Tree View Window.

Attachment 2

yaDrone-Recorder Tutorial

This guide shows how flight data are recorded using yaDrone-Recorder.

Requirements: A computer with Wi-Fi and installed Java, AR.Drone 2.0 with charged battery

1) Prepare the drone

- Insert the battery into the drone.

2) Connect the computer to the Wi-Fi network of the drone
3) Start the application yaDrone-Recorder

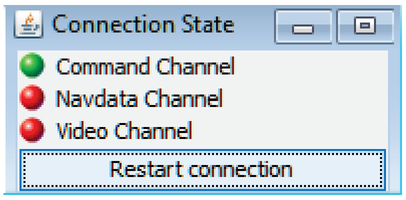
- YADrone Control Center opens. Inside, you should see the Plugin Manager window.

4) Open the important plugins in the Plugin Manager

- Open the Keyboard Control plugin which you will use to control the drone.
- Open the Battery plugin to see current battery level.
- Open the Connection State plugin to see the current state of the three communication channels. The green light is displayed next to each channel name if the connection is established. Note that the information is approximate and it might stay green even after the connection is lost.
- Turn on the Video plugin to see the image from one of the drone cameras.
- Finally, open the NavData Recorder plugin.

5) Ensure the drone is connected, likely it is not

- Lights on Connection State Plugin must all be green.
- Check that the Battery panel displays the current battery percentage and has a green background.
- The Video panel should display camera image.
- If one of the points above is not true, check that you are connected to the drone's Wi-Fi and then press the 'Restart connection' button on the Connection State plugin.



- If the video is still not displaying, try restarting the connection again or click in the middle of the Video panel (in the place where the image should be) which switches between the two cameras of the drone.

6) Prepare the Keyboard Control window

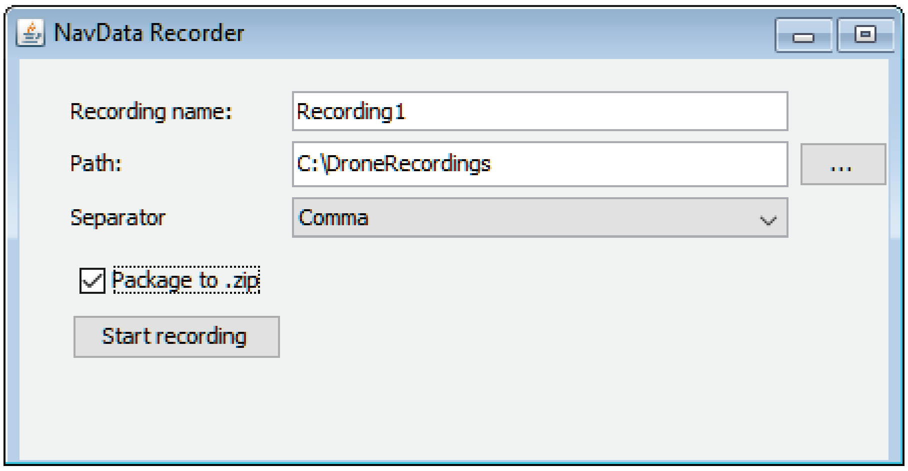
- Resize the window by dragging its corner with your mouse.
- Choose between the original and the alternative layout.

7) Ensure that all four motor LEDs light green

- The drone has to be on a flat surface.
- If the indicators light red, try resetting the drone by pressing the 'E' key on the Original Layout or the 'R' key on the Alternative Layout
- If even that does not help, restart the drone by taking the battery out and putting it back again.

8) Configure the NavData Recorder plugin

- Set a name and a location of the recording
- Leave the 'Package to .zip' option unchecked if you want to get pure comma/tab separated values. Check the option if you want to be able to directly open the recording in Drone Flight Inspector.



Recording name: Recording1

Path: C:\DroneRecordings

Separator: Comma

[✓] Package to .zip

9) Press the 'Start recording' button in the NavData Recorder panel

10) Fly with the drone

- Press enter to take off, press space to land
- Control the drone according to the selected Keyboard Layout

11) Press the 'Stop recording' button in the NavData Recorder panel

12) Open and analyse your new recording in Drone Flight Inspector

Attachment 3

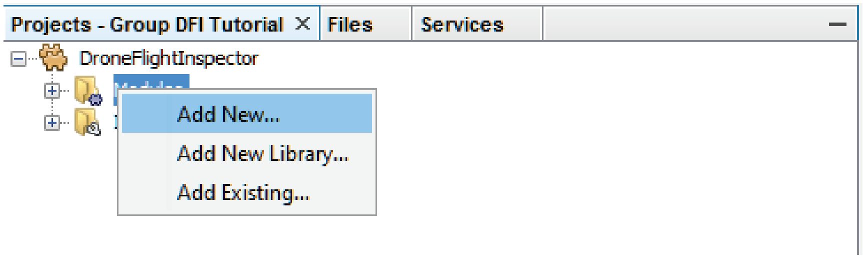
Adding a File Type Tutorial

If you have data recorded in a file format other than ROS Bag or CSV and want to be able to open them in Drone Flight Inspector, the simplest way might be to write your own import code. These instructions demonstrate how to register a new file type in Drone Flight Inspector.

Requirements: knowledge of Java, installed NetBeans IDE

1) Creating of a new module

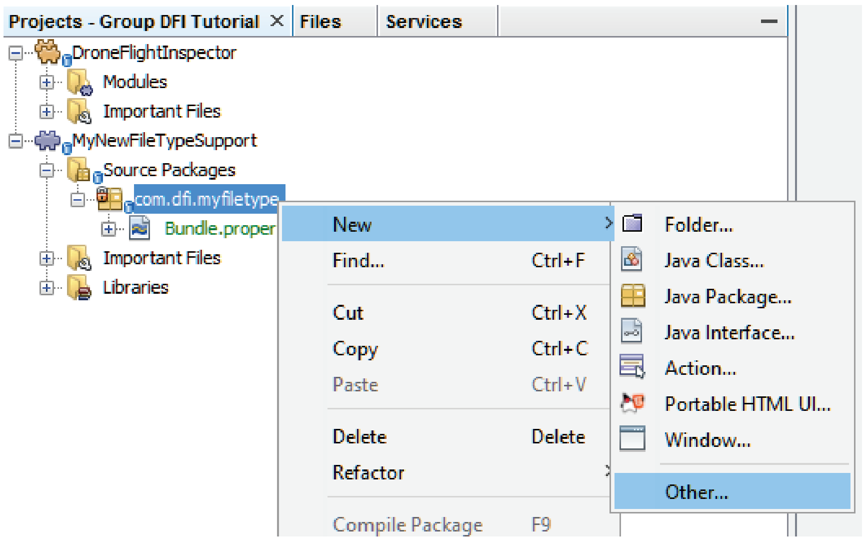
- Open the Drone Flight Inspector project in NetBeans IDE.
- Add a new module.



- Set a module name such as *MyNewFileTypeSupport* and press Next.
- Invent a package name, for example *com.dfi.myfiletype* and press Finish.

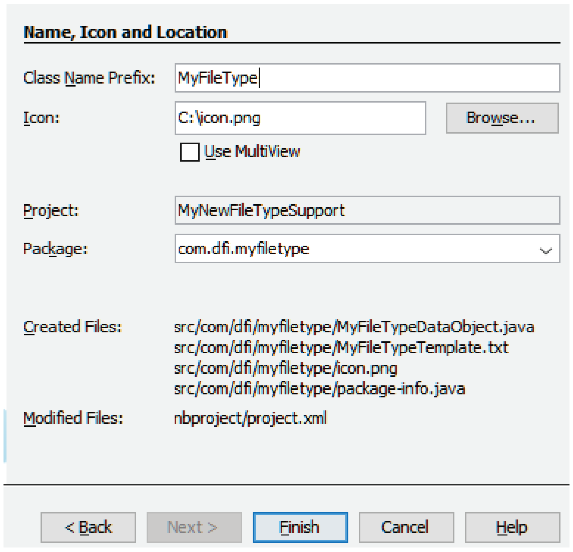
2) Add a file extension support

- Right-click the package name and choose New, Other...



- In the new file dialog, select ‘Module Development’ in ‘Categories’ and ‘File Type’ in ‘File Types’, then press the Next button.
- Fill in the MIME Type of your new file type, for example, ‘application/My-Format’, and associated file extension, e.g. ‘txt’, and click on Next.

- Set a Class Name Prefix, 16 x 16 icon and **uncheck Use MultiView**.



Name, Icon and Location

Class Name Prefix: MyFileType

Icon: C:\icon.png

Use MultiView: [ ]

Project: MyNewFileTypeSupport

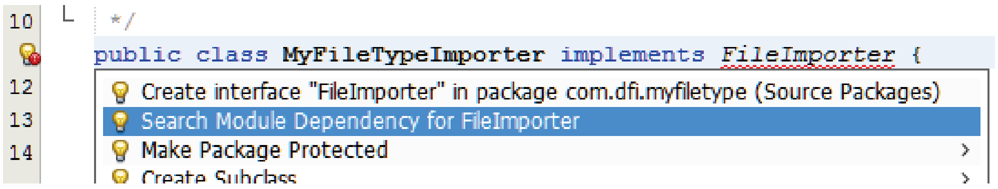
Package: com.dfi.myfiletype

Created Files:
src/com/dfi/myfiletype/MyFileTypeDataObject.java
src/com/dfi/myfiletype/MyFileTypeTemplate.txt
src/com/dfi/myfiletype/icon.png
src/com/dfi/myfiletype/package-info.java

Modified Files: nbproject/project.xml

3) Create an importer class

- Create a new class *MyFileTypeImporter*
- Change the new class so that it implements *FileImporter*.
- Click on the bulb and select 'Search Module Dependency for *FileImporter*'.



```
*/
public class MyFileTypeImporter implements FileImporter {
```

Create interface "FileImporter" in package com.dfi.myfiletype (Source Packages)
Search Module Dependency for FileImporter
Make Package Protected
Create Subclass

- In the 'Add Module Dependency' dialog, select 'Record Data Provider' and confirm the dialog.
- Click on the bulb again and choose 'Implement all abstract methods'.

4) Create a class that prepares the editor with Time Selection Component

- Create a new class named *MyFileTypeOpenSupport*.
- Edit the code according to the following snippet. Adjust the class names to match yours.

```
import org.openide.cookies.OpenCookie;
import org.openide.loaders.MultiDataObject;
import org.openide.loaders.OpenSupport;
import org.openide.windows.CloneableTopComponent;
public class MyFileTypeOpenSupport extends OpenSupport
implements OpenCookie{
    public MyFileTypeOpenSupport(MultiDataObject.Entry entry)
{
        super(entry);
    }
    @Override
    protected CloneableTopComponent
createCloneableTopComponent() {
        MultiDataObject obj = entry.getDataObject();
        MyFileTypeImporter importer = new
MyFileTypeImporter();
        return new TimeSelectionComponent(obj, importer);
    }
}
```

- Search module dependency for *TimeSelectionComponent* in the same way as in the previous step; confirm the new dependency.

5) Actually use the class *MyFileTypeOpenSupport* when a new file is opened

- Open the generated class *MyFileTypeDataObject*.
- In the constructor, replace the line:

```
registerEditor(„application/my_format‟, false);
```

with:

```
getCookieSet().add(new
MyFileTypeOpenSupport(getPrimaryEntry()));
```

6) Register the file importer as a service

- This step is necessary if we want our recording to be automatically reopened after a restart of Drone Flight Inspector.
- Switch to the *MyFileTypeImporter* class.
- Add a service provider annotation before the class definition:

```
@ServiceProvider(service =
cz.dfi.fileimporertinterface.FileImporter.class)
public class MyFileTypeImporter implements FileImporter {
```

- Add missing import for *org.openide.util.lookup.ServiceProvider* by clicking on a lightbulb.

7) Finally, import the data

- We replace the content of the method *loadRecords* in the *MyFileTypeImporter* class.
- The method should return false if we are not able to read the file, otherwise true.
- Create a File Input Stream to read the data, pass the method's first argument to its constructor:

```
FileInputStream is = new FileInputStream(data);
```

- Use *SeriesGroupWrapper.create* to create a group of series.
- Use *SeriesGroupWrapper.create_timelineLayer* to create a group of series that is displayed in the Time Selection Component as a separate layer.
- Create quantities, such as *DoubleQuantity*, by their constructor. Prefer the use of prepared wrappers from the package *cz.dfi.datamodel.common* in the Record Data Provider module. Available wrappers are *AccelerationWrapper*, *AltitudeWrapper*, *BatteryPercentWrapper*, *MagnetometerWrapper*, *MotorsWrapper*, *RotationWrapper* and *VelocityWrapper*.
- Use *IntEnumerationSeries* to create an enumeration of integer values associated with string labels.
- Publish the series by inserting them into the lookup by *content.add* (content is the second argument of the method) or alternatively, add group and all its children automatically using *ImportHelper.addTreeToLookup*
- See the example:

```
TimeStampArray stamps;
long[] rec = {100,200,300};
long[] board= {1100,1200,1300};
double[] values = new double[]{1,2,3};
stamps = new TimeStampArray(rec, board);
SeriesGroupWrapper group
```

```
        = SeriesGroupWrapper.create_timelineLayer("States",
stamps);
DoubleQuantity numbers
        = new  DoubleQuantity(values,"nums", "cm", stamps);
group.addChild(numbers);
double[] accX, accY,accZ;
accX=accY=accZ=values;
AccelerationWrapper acc
        = new AccelerationWrapper("mm/s", stamps, accX, accY,
accZ);
AltitudeWrapper alt = new AltitudeWrapper(values, "cm",
stamps);
group.addChildren(acc,alt);
ImportHelper.addTreeToLookup(group, content);
return true;
```

- Remember that time stamps have to be in nanoseconds. For further information, see the documentation of the mentioned classes and their constructors.

8) Launch Drone Flight Inspector

- In the Favourites window, the files with the new extension should have the icon you have selected in step 2) and it should be possible to open them.

Attachment 4

DFI File Type Specification

This attachment contains the specification of the Drone Flight Inspector file type that is used for storing a variable structure of data. The files that comply with this specification use the extension ‘dfi’.

Drone Flight Inspector data file is a ZIP archive. An XML file with the name ‘description.xml’ has to be present directly in the root of the archive. Moreover, the archive has to contain one or more files with plain text values separated by commas or tabs and line ending characters.

Values Files

Each line of the comma or tab separated values file has to contain exactly the same number of values, forming a rectangular table of values. It is not allowed to skip a value by insertion of two successive separators. A value may consist of any characters other than the current separator (comma or tab) and line ending characters.

XML Description Structure

The ‘description.xml’ describes the structure of the values files. The file has to contain the XML document element *recording*. The root element has one child element *toplevelgroup* for each values file in the ZIP archive. The *toplevelgroup* element represents the values file and has one descendant for each column in the file. These elements describe the data type and semantics of the values in the column, available elements are *intQuantity*, *doubleQuantity*, *intEnumeration* and *time*. However, they do not have to be direct children of the *toplevelgroup* element. One or more descendants of the *toplevelgroup* can be wrapped in an element named *group*.

The element *toplevelgroup* has three mandatory attributes, *name*, *file* and *separator.* The *name* attribute may contain an arbitrary string name that characterises the content of the file specified in the *file* attribute. The *file* attribute has to contain name of a values file or a relative path to the file if the file is not in the root of the archive. Two different *toplevelgroup* elements must not refer to the same file. The *separator* attribute may contain the value *comma* or *tab* and it must correspond to the separator used in the file.

The element *group* has a single required attribute *name*.

The elements *intQuantity* and *doubleQuantity* have a mandatory attribute *name* and an optional attribute *unit*. If the element is the nth descendant of a *toplevelgroup* element it describes the series of values in the nth column of the file specified in the *file* attribute of the *toplevelgroup*. If the element is *intQuantity*, the values in the column must be integers. Similarly, if the element is *doubleQuantity*, all the values in the column must have floating point number format.

The *intEnumeration* element has an only required attribute *name.* It represents a column in the comma or tab separated values file in the same way as *intQuantity* and *doubleQuantity*. The values in the corresponding column must be integers. The *intEnumeration* element provides a mapping of the integer values to string, human-readable state names. For each value that can appear in the column, the *intEnumeration* element has to contain a child *option* element, which must have attribute *value* and attribute *label*. The attribute *value* contains the integer value from the data file and the attribute *label* specifies its string description.

Each *toplevelgroup* must have at least one direct child with the name *time* that specifies a column with time values. All time values are integers, when parsed stored as longs (64 bits integers). The attributes mandatory for the *time* element are *unit* and *source*. The possible values of *unit* are *nanoseconds*, *microseconds*, *milliseconds* or *seconds*. *Source* can be either *recorder* or *board*.

Attachment 5

Detailed Experiment Results

The table contains results of 25 test flights recorded during the experiment described in the Chapter 5.2.2.

Flight Number	Rotation	Flight length	The distance of the landing point from the initial position [mm]	Rounds
1	no	02:00	1093	1
2	no	02:00	955	1
3	yes	02:01	2142	1
4	yes	02:50	1288	1
5	yes	02:43	4274	2
6	no	01:37	1813	1
7	yes	02:06	2845	1
8	no	01:10	362	1
9	yes	02:07	2547	1
10	yes	01:54	747	1
11	yes	02:00	1096	1
12	yes	01:30	648	1
13	no	01:52	2178	1
14	no	00:48	694	1
15	yes	01:02	554	1
16	yes	02:54	1080	1
17	yes	01:09	862	1
18	no	01:16	151	2
19	no	00:39	331	1
20	yes	01:28	628	1
21	no	00:46	2385	1
22	yes	01:50	1281	2
23	no	01:09	1538	2
24	yes	01:41	2686	2
25	yes	01:47	1512	2
Average		01:42	1427,6	

Attachment 6

List of Attached Files

The following files are attached to this thesis:

1. Drone Flight Inspector installation files for Windows, Linux and macOS
2. Drone Flight Inspector ZIP distribution for Windows and Linux
3. Drone Flight Inspector documentation
4. yaDrone-Recorder Java executable file
5. Drone Flight Inspector source code
6. yaDrone-Recorder source code
7. Recorded test flights including:
    a. Trajectory evaluation flights with description
    b. Orientation test flights
    c. Recording of a crash
    d. Yaw fluctuation test recordings