Graz University of Technology
Institute of Control and Automation

Master's Thesis

Modeling and Simulation of Distributed Autonomous Robots

Christoph J. Gruber
Graz, Austria, November 2009

Thesis supervisor:
Univ.-Prof. Dr. Michael Hofbaur

Abstract

Multi-robot systems are built through expensive and complex physical artefacts. Simulating such systems in a real-world like virtual environment is thus very valuable to validate operational procedures and control laws. This thesis develops mathematical models for real world robots, in particular the quadrotor-helicopter UAVs used in the cDrones project. The formalism of Lagrange is applied to find different dynamic models for simulation, control and parameter identification. Furthermore, the topic of aerodynamics is investigated. Using flight data from the real UAV, the model parameters are identified and its validity is verified within a certain range.

The major outcome of this work is software, which integrates these robot-models within a virtual reality simulation environment. Emphasis is taken on the overall behavior of the individual quadrotor-helicopters and sensory information. Being able to simulate multiple instances of robots within the virtual environment will form the basis for further research in the cDrones project, where the simulator and the interfaces developed within this thesis will serve as testbed.

Keywords. multiple UAV simulation, unmanned areal vehicles, quadrotor-helicopter, simulation, modeling

Acknowledgments

First of all, I want to thank my parents Maria & Gaston for their support through all the years of my education. Without their care, patience and financial support my student life would not have been as great as it actually was. Also my sisters Magdalena and Christine motivated me through their generous and funny kind. Gaston Philipp, my young brother always showed interest in the things I am doing and was a loyal fellow in all the years, not just at teasing my sisters but also at rock climbing and much more. Furthermore, I want to thank my grandparents Katharina & Johann and Barbara & Lorenz for their generous support.

My mentor and supervisor, Univ.-Prof. Dr. Michael Hofbaur, trusted in me throughout my whole master studies. He supported me many times with good advice as well as very informative and exciting lectures. Here, I want to thank him for all he did for me.

Furthermore, I want to use this page to express my gratitude to my friends, who also helped me with this work. At the first place Thomas Höll, my loyal and fellow campaigner, for the innumerable valuable coffee-break conversations. DI Daniel Muschick, friend and PhD student at IRT, helped me with his subtle comments and constructive criticism. My colleague DI Christoph Schörghuber earns my thanks for the nice time at the robotics lab at IRT. The same applies to Lukas Vogl for the great time and shared experiences during my studies in Trondheim. Many thanks also to my long-time friends from Gastein and Graz, I owe to them wonderful years of study.

This thesis was part of the cDrones project at the Lakeside Labs GmbH. Sincere thanks for the compensation money.

Last, but definitely not least, I want to thank my girlfriend Sabine Plaickner for her care, faith and understanding of my busy way of life.

Christoph Gruber Trondheim, November 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Robotics has always been an interdisciplinary field of technology [SHV06]. Mechanical engineering, electrical engineering and computer science are the main disciplines in robotics research. Currently, it expands this interdisciplinarity further to new topics, and the term robot is getting more and more important. Nowadays, robots are used in medicine, production, entertainment, exploration, surveillance etc. As a part of the cDrones project, this thesis is related to robots used in rescue missions.

1.1 Autonomous Robots

A very interesting and challenging topic is the field of autonomous robotics. Autonomy means that a robot should be able to interact with the world without human guidance or control. [Bek05] defines autonomy as follows: "Systems capable of operating in the real-world environment without any form of external control for extended periods of time".

In fact, a lot of problems arise with the goal of autonomy. Consider this short, fictitious example: If someone develops a helicopter that should be able to fly to a special

destination, two principal approaches come to mind:

1. The designer could steer the robot along the route once and store the control command sequence in the robots memory. The next time the route should be flown, the robot simply uses these control commands.

2. The designer could build a robot that is able to detect the route by itself and generates commands to follow this route to the destination.

It is obvious that the second approach is technically more challenging. In fact, there is no guarantee that any of the approaches is technically feasible at all (with sufficient accuracy). It is very unlikely that the first robot is able to fly exactly the same trajectory as in the teaching scenario. Changes in environmental conditions etc. will avoid an accurate job. Depending on its ability to operate independent of human instructions one could call the second robot more or less autonomous.

Components of Autonomous Robot Systems. An autonomous robot as the second helicopter needs to know a lot about its state and environment at any time (it has a so-called *world model*). That knowledge must contain things like the current position, orientation, the environment, etc. Hereunto, the robot could for example be equipped with an image sensor, a GPS sensor and an IMU. The measurements from these sensors must somehow be merged in order to achieve a sufficiently accurate position estimate (*sensor fusion*) which is then used to update the world model. Depending on the current state, the robot must then decide what to do next (*expert system*). If that decision is e.g. to fly to a special way-point, it has to calculate a route (*path planning*) and generate control commands for the motors (*controller*).

There is a special family of robots, where autonomy is inherently of interest, namely unmanned aerial vehicles (UAVs). These are the main topic in this thesis.

1.2 Distributed Autonomous Robots

Another modern research topic in the field of robotics is the use of multiple robots that collaborate somehow. Many research facilities work on projects with distributed ground vehicles. The soccer-playing robots of the RoboCup are a further example of distributed autonomous robots in up-to-date science. The use of unmanned aerial vehicles (UAV) for research in distributed robotics is also very popular at this time [HRW+04], [QSB+08], [Val07], ...

The task of collaboration yields new problems and challenges in modeling and simulation, as well as in wireless communication, distributed and networked control and many more. Moreover, cooperative strategies find an interesting real-world application.

Due to the special properties of distributed autonomous robots, they demand models for specific sensors, actuators and robot structures. Furthermore, the requirements on simulation differ from the classical approach in robotics or control science, where usually no environment is considered in simulation. These special needs for research in the field of distributed autonomous robotics are tackled in this thesis.

1.3 Unmanned Aerial Vehicles

UAVs are special types of aircrafts, where a human who actively pilots is absent. The term unmanned refers to the replacement of a on-board human pilot with a control system. Such control functions could be either on-board or off-board (remote control). The term UAV, or often called drone, does not specify the mechanical structure of the aircraft: It could be a fixed-wing or rotary-wing vehicle, a helicopter or any other aerial vehicle structure [Val07].

UAVs were first introduced by the US in the first world war. Since that time, it was mainly military interests which facilitated further development. The ignition of a completely different perception of UAVs were the terrorist attacks on 9/11/2001 in New York. There, for the first time UAVs were shown and used in media and became a topic to the public domain. A civil UAV market began to emerge, starting with government organizations and media. The non-military interests in UAVs are basically in the segment of vertical take-off and landing vehicles. Applications include pipeline and power-line inspection and surveillance, border patrol, rescue missions, oil and natural gas search, fire prevention, topography, natural disasters and agricultural uses (mostly in Japan) [Val07].

1.3.1 Microdrone MD4-200

The VTOL UAV used in the cDrones project is the MD4-200 produced by a German company named Microdrones[1]. Fig. 1.1 shows a picture of the drone. The four rotors have fixed pitch carbon fiber blades. Its frame is also made of carbon fiber and has a diameter of approximately 60 centimeters. Actuators are brushless DC-motors with external rotors (outrunners), which are directly connected (no gearbox) to the blades. Power is provided

[1] http://www.microdrones.com

by a lithium-polymer battery pack, which enables the drone to fly for approximately 20 minutes. The MD-400 is equipped with sensors for navigation and control. Moreover, controllers for flight via remote control and way-point-based flight are available. All data processing on the drone is closed source. An on-board flight data recorder as well as a radio down-link allow analysis from a base-station. For the cDrones project, the drone is equipped with a compact-camera, which can be triggered and tilted via remote control. A special script language facilitates camera control during way-point-based flight. Using such a script, it is possible to write a program for a whole flight, including automatic take-off and landing.

Figure 1.1: Microdrone MD4-200[2]

1.4 The Collaborative Microdrones Project

The project *collaborative Microdrones* (cDrones) [Lab08],[QSB+08] is operating in the field introduced above of distributed autonomous robotics. Multiple small-scale UAVs should fly in collaboration over a predefined area and deliver images from the ground. The UAVs used in the project are commercial quad-rotor helicopters (Microdrones, see sec. 1.3.1), that already support throttle, yaw, pitch, roll control, GPS-way-point based flight and autonomous vertical take off and landing (VTOL). These Microdrones should fly - on demand in formation - a route, that covers a predefined area of interest. At

[2]source: http://www.microdrones.com
[3]source: http://www.microdrones.com

Figure 1.2: Microdrone MD4-200 base-station[3]

special waypoints, pictures are taken and stitched together to get a detailed image of the area. One goal of the project is to demonstrate the use of UAVs in disaster management applications. It is planned to use the drones to provide prior information about a disaster area to human task forces. Scientific challenges in this project are found in the fields of:

- Flight formation and networked control
- Mission planning
- Cooperative aerial imaging

1.5 Tasks

This master's thesis is part of the cDrones project and should provide the scientists with a powerful tool for realistic simulation of multiple quad-rotor helicopters. The simulator must be usable for experiments as well as for demonstration. An interface for simple switching between simulator and prototype should be provided. Moreover, simulation of

- rigid bodies, which are subject to real-world like physical phenomena (gravity, collisions, ...)

- actuators and their dynamic models
- sensors and their associated models (noise, ...)
- controllers (as provided on the Microdrone)
- communication networks
- environmental influences onto any of the previous mentioned items

is demanded. Although the *Unified System for Automation and Robot Simulation* (USARSim) principally comes with the possibility to simulate communication networks, this was not considered in this thesis. A separate project is dedicated to this topic within the cDrones project. Also environmental influences - like e.g. wind - are not taken into account in this work. Nevertheless, the possibility to implement those issues at a later date is maintained.

A dynamical model of quadrotor-helicopters should be developed and verified for the Microdrone, providing the basis for control design and simulation. Furthermore, the parameters of this model should be identified. The overall goal should be a flexible and powerful simulator, that is going to be a part of the testbed for the cDrones project.

1.6 Related Work

A lot of institutions worldwide do research in the field of UAVs. Here, I give a short overview:

The Unmanned Autonomous Vehicles Group at the Université de Technologie Compiègne deals with modeling and control of different vehicle structures, also including the quadrocopter. Several publications, containing [CLD05], were made by their researchers. They claim to have been the first who successfully applied real-time control to a four-rotor rotor-craft.

Mesicopter was a project at Stanford university that developed a quadrotor-helicopter with dimensions of about two and a half centimeters. Research focused on aerodynamics, motor and power system design and meso scale fabrication methods.

The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) project builds its own quad-rotor helicopters, which are currently in the second generation. Dynamic models were developed using the Newton-Euler formulation. Details of modeling, control and implementation for single drones were issued. Several of their quadrocopters serve as a testbed for multi-agent control.

At the Autonomous Systems Lab at ETH Zurich, unmanned aerial vehicles are the topic of various projects from the past as well as in current research. Several publications deal with quad-rotor modeling, simulation and control.

The X4-Flyer project of the Robotics Engineering group at the Australian National University aims at developing a quadrocopter from scratch and uses it as experimental platform for visual servo control.

The Vision and Autonomous Systems Center at the Carnegie Mellon University uses, among other robot types, UAVs for research. Publications were done in the field of system identification, control, path planning and many more.

A Framework for Multi-UAV Simulation , especially for vehicles with rotary-wings, is developed at the Università Politecnica delle Marche. It allows easy switching from simulated to real environments in single UAV-simulations as well as in cooperative missions.

The MultiUAV2 project of the US Air Force Research Labs/Wright-Patterson Air Force Base developed software to simulate multiple UAVs which cooperate to accomplish tactical missions. The simulator was based on MATLAB/Simulink and C++. It was published in 2006.

Jinhyun Kim, Min-Sung Kang and Sangdeok Park present a mathematically accurate model of the quadrotor-helicopter in [KKP09]. They use the formalism of Lagrange and a Lagrange-like method to derive the equations of motion in a hybrid reference frame. Furthermore, they propose a robust hovering controller.

Tommaso Bresciani wrote his master's thesis on modeling, identification and control of quadrotor-helicopters. He uses Newton-Euler formulation to derive the equations of motion in a hybrid reference frame. For simulation, the dynamic model was implemented

in Simulink. In his work, he describes the development of the whole control system of a quadrotor-helicopter prototype from scratch.

1.7 Outline

Chapter 2 describes the principals of Navigation. First, reference frames that are used throughout the whole text are introduced. Then, the NAVSTAR *global positioning system* (GPS) as well as *inertial navigation systems* (INS) are explained. The accuracy of GPS is considered and an error model for the *inertial measurement unit* (IMU) used in the Microdrone is presented.

At the beginning of the chapter about modeling it is explained how quadrotor helicopters work. After that, these intuitive thoughts are formalized. Topics like kinematics and dynamics are tackled, leading to the equations of motion of quadrotor helicopters in different reference frames. Finally, a linearized version, suitable for parameter identification, is presented.

The chapter about Parameter Identification is dedicated to the search for the parameters of the Microdrone. First, most of them are calculated using physical laws. Then, an experimental parameter identification is used to prove the validity of these calculations, the linearized model and delivers the missing parameters.

Chapter 5 is about real-world like simulation of multiple drones in a common environment. At the beginning the demands on multi-UAV simulation are considered, which deliver the basis for the selection of USARSim as simulation framework. The simulator and its builtin UAV model are analyzed and modified to adjust the dynamical behavior to the Microdrone. Finally, the development of software and interfaces to concurrently simulate multiple UAVs in the high-fidelity simulator USARSim is documented.

Chapter 2

Navigation & Sensing

Contents

Autonomous mobile robots always have to navigate somehow and hence need sensors which deliver signals that are related to their position and orientation. Depending on the environment, different sensors are used. In an outdoor environment, typically at least position estimates from a GPS receiver and acceleration and orientation estimates from an *inertial navigation system* (INS) are fused to achieve accurate navigation. Additional sensors to measure height and yaw rotation are often employed especially for aerial vehicles [HHWT07].

However, the Microdrones used in the cDrones project are able to navigate out- and indoors supporting the following navigation modes [Mic07]:

- INS only
- Magnetometer and INS
- GPS and INS
- Way point

2.1 Reference Frames

In most cases, the frame where a measurement takes place is a different one than the frame of interest. Thus, several coordinate frames need to be defined. This is just a short introduction into common reference frames, a more detailed version can be found in [FB99], where this chapter is based on, or [HWWL03]. The transformations between reference frames needed in this work are presented in the chapter on kinematics, sec. 3.2.

Inertial Frame is the reference frame for accelerometer and gyro measurements. It is a frame where Newton's laws of motions apply. In every non-inertial frame, additional terms (fictitious forces such as Coriolis, centrifugal and Euler forces) will occur in the equations of motion. Thus such an inertial reference frame is neither accelerating nor rotating, but may be in linear motion. The origin and axes of the frame may be arbitrarily placed as long as all three axes point in mutually perpendicular directions. Inertial measurement units (IMUs) produce measurements relative to such an inertial frame placed at the instruments sensitive axes.

Earth-Centered Earth-Fixed Frames have their origin in the center of the earth. There are two types of ECEF-coordinate systems: The rectangular and the geodetic one. In the rectangular system, the z-axis is coincident with the earth's spin axis. The x-axis points towards zero latitude and longitude, while the y-axis completes the right-handed coordinate system passing through the equator and 90° latitude. The GPS-Sensor used in the Microdrone delivers its position measurements relative to this frame. The geodetic ECEF frame approximates the earth's shape by an ellipsoid with its minor axis coincident with the earth's spin axis. The parameters of the ellipsoid are defined in the WGS-84 standard. Two ways of expressing coordinates relative to the ECEF frame are common: The cartesian (x,y,z) and the elliptic (latitude, longitude, altitude). As the name implies, the transformation between those two is nonlinear and can be found in e.g. [HWWL03]. Notice that ECEF coordinate frames are no inertial frames, because they rotate with the earth around itself and the sun.

Local Geodetic Frame also referred to as local level frame, is the conventional reference frame for local navigation. Its z-axis is perpendicular to a plane, which is tangent to the earth's geoid in some arbitrary point. This point specifies the origin of the frame, no matter if the system is moving or stationary. The z-axis is directed down, while the x and

y axes point towards north and east, respectively.

The transformation from ECEF to local level frame coordinates follows the algorithm presented in [FB99] and is implemented in the LabVIEW VI ECEF2NED.vi.

Body Frame is a coordinate system rigidly attached to the system of interest. Typically, the origin is located at the center of mass or the measurements are transformed to that point (Lever arm transformation, [Vik]). The x-axis points in the direction of linear (forward) motion. z is pointing downwards and y completes the right-handed orthogonal coordinate system pointing to the right (lateral motion direction, starboard).

2.2 Systems & Sensors

For navigation, the Microdrone MD4-200 is equipped with the following sensors:

- A GPS
- An INS
- An altitude- (air pressure based) and a temperature sensor
- A magnetometer

In the simulator (USARSim, see chap. 5.2), only GPS and INS sensors are modeled. However, the possibility of also modeling the other sensors exists. If a detailed knowledge about the sensor fusion implementation on the Microdrone would be available, this part could also be simulated. As this is not the case here, another approach was chosen for simulation, which is described in sec. 5.2.7.

2.2.1 NAVSTAR Global Positioning System

To understand the simplifications made in the existing simulation model of the GPS sensor in USARSim, it was necessary to understand in principle how the sensor works in real-world applications. However, the following section is an introduction and thus not broken down into details. It is more or less a summary of [BC08], which was written by the developers of the USARSim GPS sensor. Additional information on GPS is found in [Vik].

The NAVSTAR global positioning system is a satellite based radio navigation system, that is managed and controlled by the United States Air Force. Several satellites (24-32), that circle the planet in Medium Earth orbit, send signals containing precise information about the current location and time of themselves. A receiver can calculate the difference between that time and its own (synchronized) current time. As radio waves are known to propagate with the speed of light, the receiver can calculate an estimate of its distance[1] to the satellite $(t_{recv} - t_{sent})c$ and must thus be located somewhere on a sphere around the satellite's location. If the receiver picks up messages of more than one satellite, it must be located somewhere at the intersection of the spheres. Using data from 3 satellites, a receiver is able to locate itself, because 3 spheres intersect in two points, where one of them is not reasonable. However, a GPS receiver needs 4 satellites to give a good position measurement, where the fourth satellite is used to correct the receiver clock error. Of course, the signal from the satellites is subject to several kinds of distortions. Some of them (atmospheric effects, relativistic effects, ...) can be mathematically modeled and corrected by the receiver. Others are interpreted as perturbations (multi-path effect, numerical errors, ...) [Vik].

In order to track a satellite, a model for path calculation was proposed by the three governing North American aerospace institutions. This Simplified General Perturbations Satellite Orbit Model 4 (SGP4) takes a so called Two-Line Element (TLE), date and time as input and outputs the corresponding satellite location. Fig. 2.1 shows an example of such a TLE, which contains the parameters to uniquely identify the orbit. They are the Keplerian elements parameters, which are described in fig. 2.2. In the GPS case, the gray disk corresponds to the equatorial plane, the celestial body is the satellite and the yellow disk is the orbital plane. γ points towards the vernal equinox and, together with the intersection of the planes, defines an earth-centered inertial reference frame. The simulation of satellite movement in USARSim is based on this model [BC08].

The GPS receiver used in the Microdrone delivers signals at a sample time of $t_{s,GPS} = 250[ms]$. The position information is given in $[cm]$ relative to cartesian ECEF frame. Also an accuracy signal is one of the sensor outputs. Data recorded in Experiment 1 (see chap. C), where 8 to 9 satellites were reachable, was used to analyze the properties of the measurements. Fig. 2.3 shows recordings from the GPS position accuracy signal (black, $[m]$) and preprocessed position measurements p_{GPS} where the drone was at rest. The

[1] This is usually called the pseudorange

[2] Source: http://spaceflight.nasa.gov/realdata/sightings/SSapplications/Post/ /JavaSSOP/SSOP_Help/tle_def.html

[3] Source: http://en.wikipedia.org/wiki/Orbital_elements

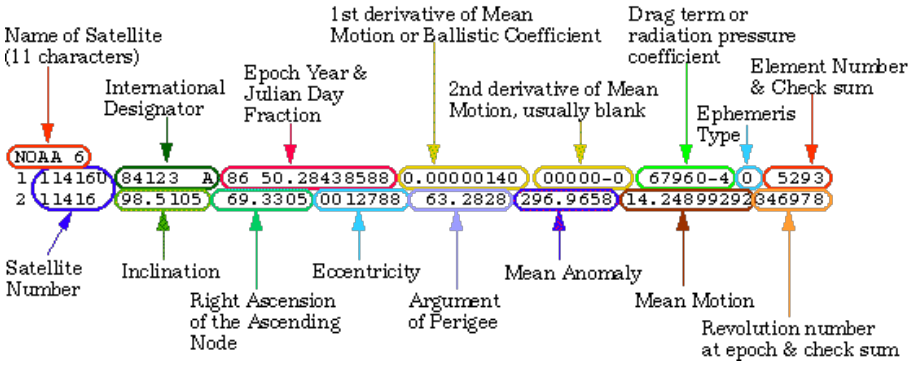

Figure 2.1: Sample TLE File with Format Descriptions[2]

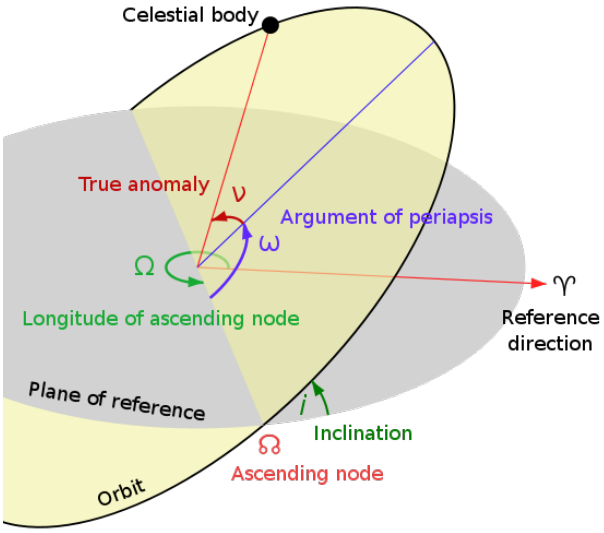

Figure 2.2: Keplerian elements[3]

position measurements were treated with the following operation

$$\overline{p_{GPS}} = \sum_{k=1}^{M} \|p_{GPS,k}\|_2 \tag{2.1}$$

$$N_{GPS,k} = 0.01\,(\|p_{GPS,k}\|_2 - \overline{p_{GPS}}) \tag{2.2}$$

where $\|\cdot\|_2$ stands for the 2-norm. The value $\overline{p_{GPS}}$ is the mean of the position measurements and M refers to the number of samples that were recorded. N_{GPS} is the red signal shown in 2.3.

The calculations in (2.2) were done, because a comparison with a true value was not

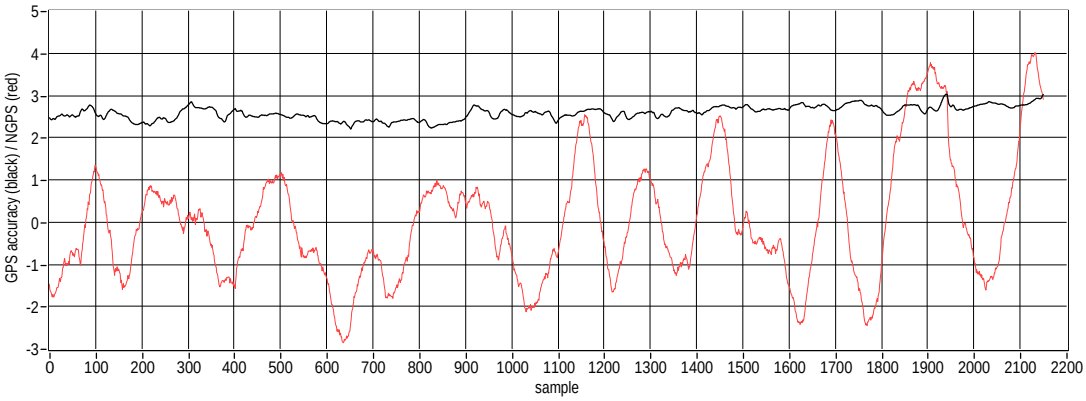

Figure 2.3: GPS accuracy (black) and N_{GPS} (red) where the Microdrone is at rest

possible, since a true position value was not available. Furthermore, it was assumed that for the resting quadrocopter, the position measurement is a stationary ergodic random process, which motivates the following investigation:

An estimate of the standard deviation σ_{GPS} of $\|p_{GPS}\|_2$ is

$$\sigma_{GPS} = 143.457[cm]$$

Fig. 2.4 shows the autocorrelation of the position noise. For the noise process to be white, there should be a peak at zero shift and zero elsewhere. As very rough approximation, that is the case here and the noise of the GPS sensor is assumed to be white for simplicity. Actually, the signal shown in fig. 2.3 and the autocorrelation function look more like those of a Gauss-Markov process (band-limited white noise). A sophisticated noise model for GPS receivers can be found in [Vik]. However, because [BC08] use white noise, this model is used here, too.

2.2.2 Inertial Measurement Units & Inertial Navigation Systems

Inertial sensors measure acceleration and rotation rate relative to an inertial reference frame. The sensors used for measuring acceleration are called accelerometers and those for rotation rate measurement are gyroscopes. If at least three accelerometers and at least three gyroscopes are mounted rigidly on a common base, the resulting device is called an *inertial sensor assembly* (ISA). Together with a hardware interface and low level software that performs e.g. downsampling, temperature calibration and vibration compensation it forms an IMU. As the physical operation of the sensors is not a topic in this work and

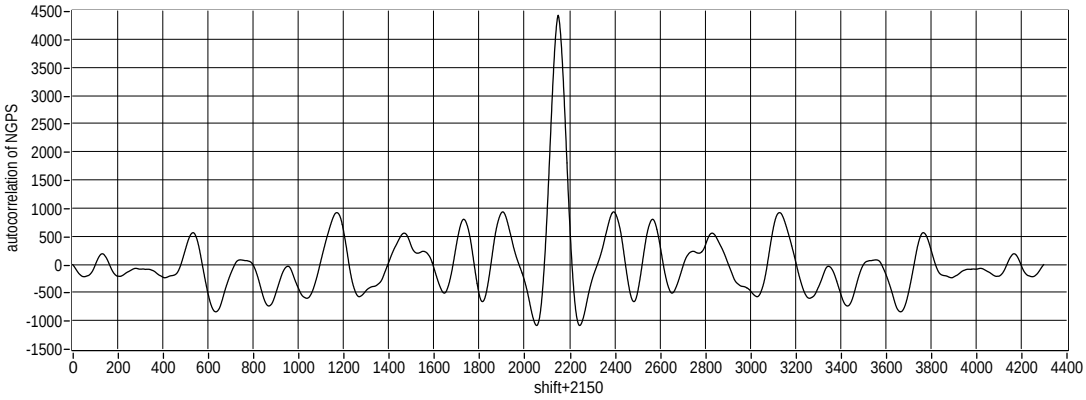

Figure 2.4: autocorrelation of N_{GPS}

the simulation is not based on a physical sensor model, further description is omitted here[4]. The interested reader is referred to [GWA07] or [Vik]. An Inertial Navigation System (INS) consists of such an IMU and a navigation computer, which is responsible for calculating attitude and position with respect to a certain (selected) frame[5]. An INS, rigidly mounted on the vehicle of interest, is called a strap-down INS, and the process of navigation computation is termed strap-down computation. A flowchart of such a calculation is shown in fig. 2.5. The main steps in this procedure are:

1. Attitude computation
2. Specific force transformation (where specific force here means the accelerometer measurements in the body frame)
3. Normal gravity vector computation
4. Coriolis force vector computation
5. 1st navigation equation integration
6. 2nd navigation equation integration

For further details refer to [WCA99] and [Vik].

[4]Their physical implementation in the Microdrone is not known, but most likely they are MEMS

[5]At least one external measurement of heigth is necessary for an INS to overcome the vertical channel instability [Vik]

[6]Source: http://large.stanford.edu/courses/ph210/noriega1/

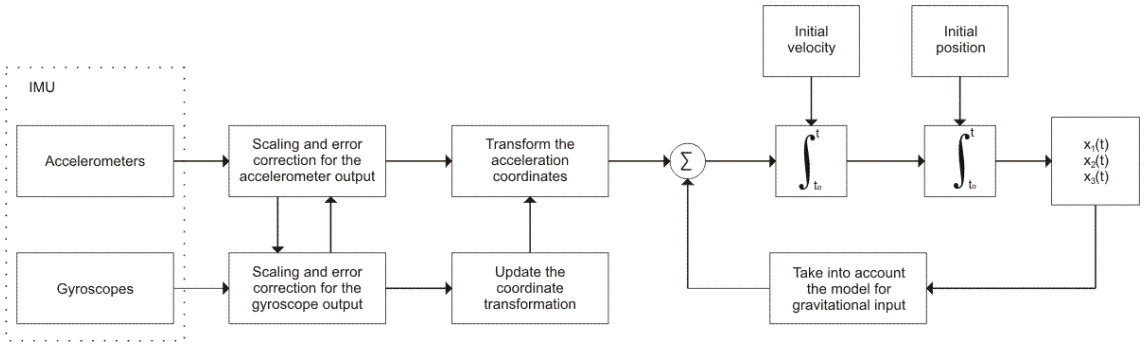

Figure 2.5: Flowchart of the strap-down computation[6]

2.2.2.1 Errors and Sources

As the measurements from the IMU are also used for system identification, a common error model for IMUs is presented here. At the calibration process of the Microdrone, the parameters of this error model were determined and are available to us. The most dominating parameters are usually biases, scale-factor errors and misalignment errors. The output of the IMU can be modeled as[7]

$$\begin{aligned} \boldsymbol{\omega}_{imu} &= \boldsymbol{\Delta}(\boldsymbol{\kappa}, \boldsymbol{\alpha})\boldsymbol{\omega}_{ib}^{b} - \boldsymbol{b}_g - \boldsymbol{w}_1 \\ \boldsymbol{f}_{imu} &= \boldsymbol{\Delta}(\boldsymbol{\epsilon}, \boldsymbol{\beta})\boldsymbol{f}^{b} - \boldsymbol{b}_a - \boldsymbol{w}_2 \end{aligned}$$

where the notation means that $\boldsymbol{\omega}_{\boldsymbol{ib}}^{\boldsymbol{b}}$ is the angular velocity of the body frame **b** (subscript 2) relative to an inertial frame **i** (subscript 1) expressed in the body frame (superscript). $\boldsymbol{\omega}_{imu}$ is the angular velocity vector as it is delivered by the IMU. $\boldsymbol{f}^{b}$ means the specific force vector in the body frame, which still contains acceleration due to gravity. Furthermore,

$$\boldsymbol{\Delta}(\boldsymbol{s}, \boldsymbol{\vartheta}) = \begin{bmatrix} s_x & \vartheta_{xy} & \vartheta_{xz} \\ \vartheta_{yx} & s_y & \vartheta_{yz} \\ \vartheta_{zx} & \vartheta_{zy} & s_z \end{bmatrix}$$

Here,

[7] Typically this is done in a different way, see e.g. [GWA07]. In order to use the same values as delivered from Microdrones GmbH., the model has been adopted.

$$\boldsymbol{s} = \begin{bmatrix} s_x, s_y, s_z \end{bmatrix}^T$$

are three scale factors and

$$\boldsymbol{\vartheta} = \begin{bmatrix} \vartheta_{xy} & \vartheta_{xz} & \vartheta_{yx} & \vartheta_{yz} & \vartheta_{zx} & \vartheta_{zy} \end{bmatrix}^T$$

are six small misalignment angles, which are assumed to be zero here. $\boldsymbol{b}_g$ and $\boldsymbol{b}_a$ represent gyro and accelerometer biases, respectively. Often, and especially when a Kalman filter is used for sensor fusion, the bias is modeled as Gauss-Markov process and estimated by the filter. The parameters of the process are typically determined by the manufacturer of the IMU, as well as the scale factors. In our case, the following values were extracted from the drone support software "mdCockpit", provided by Microdrones GmbH.:

$$\begin{aligned} \boldsymbol{\kappa} &= \begin{bmatrix} 1.6523 & 1.6255 & 1.6526 \end{bmatrix}^T \\ \boldsymbol{\epsilon} &= \begin{bmatrix} 0.9492 & 0.9564 & 0.9469 \end{bmatrix}^T \\ \boldsymbol{\alpha} &= \boldsymbol{\beta} = \mathbf{0} \\ \boldsymbol{b}_g &= \begin{bmatrix} -0.2910 & 0.2832 & 0.2588 \end{bmatrix}^T \\ \boldsymbol{b}_a &= \begin{bmatrix} -0.0036 & 0.4396 & -0.6913 \end{bmatrix}^T \end{aligned}$$

$\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are assumed to be stationary, zero mean, uncorrelated Gaussian white noise processes.

As inertial navigation systems are very complex, there is a large variety of possible errors. Typical ones are:

- Sensor noise. This is the major problem with INS: As accelerations are measured and integrated twice, sensor noise affects the position in form of drift. Due to this drift, such INS lose accuracy with increasing time.
- Errors in the sensing devices: bias, nonlinearity, scale factors, asymmetry, dead zones, quantization, etc.
- Sensor misalignment. Could be compensated by using more than three accelerometers and gyroscopes.

- Numeric computation errors. The amount of calculations necessary in a strap-down system is extensive. Hence, Integrals are approximated with finite-interval sums (Attitude computation).

There are error models for inertial navigation systems, but they are too complex to be treated here. USARSim uses a special model that is presented in chap. 5.2.7.3. For detailed error models, refer to [Vik, chapter 2.4.2] and [GWA07, chapter 6.5].

Compared to GPS, the short-term accuracy of INS is much better. On the other hand, GPS is drift-less and thus long-term accurate. It is the task of sensor fusion to unite the advantages of both systems.

2.3 Sensor Fusion

A major issue in autonomous navigation is typically the sensor fusion. In many cases, control algorithms are based on velocity measurements, as the reference inputs are often velocities. But there exists no sensor for measuring velocity directly. An IMU on the one hand outputs accelerations, which have to be integrated once for velocities and twice for positions. Due to the double integration, noise in the acceleration appears as drift in the position, which is an inherent property of INS. On the other hand, GPS measures position, so a derivation is necessary to compute velocity. Furthermore, different other sensors - like e.g. magnetometers and altimeters - measure position or one component of it, so that there are many (most likely different) measurements of the same physical quantity. In the fusion of these signals typically one or more Kalman filters are involved, although there are several structures that could be thought of [GWA07], [Vik]. As this sensor fusion is not topic in this thesis nor sufficient insight in the navigation computation of the Microdrone could be achieved, it is not further considered here.

Chapter 3

Modeling

Contents

The process of developing a mathematical model of a real system is called modeling. There are a lot of different types of models, and each of them is dedicated to a special purpose and describes a special type of behavior. In control engineering, the typical model type is a set of differential or difference equations which model physical behaviors. In fact, these models are simplifications of the real system and thus valid in a dedicated range only. If someone wants to model e.g. an electric drive, there exist multiple ways how such a model could be developed. Two of them are considered here:

- The engineer discovers the physical effects and relations that happen inside the system. In the electric drive, there will appear electric and magnetic effects on the one hand and mechanical effects on the other hand. For those, laws are known (e.g. law of inductance, Newton's laws of motion etc.) and assumed to be sufficiently precise. These laws are then assembled, so one equation emerges that describes the behavior of the whole system. A graph based approach for this method are the so-called bond-graphs [Hof05].

- Another method of modeling is termed Lagrange approach [Hof04], [SHV06]. It is typically used to model mechanical dynamics where system variables (usually

coordinates) are subject to constraints. These constraints delimit the degrees of freedom of the system. As a first step, these constrained system variables are - if possible - replaced by so-called generalized coordinates, which are all independent. Then, the engineer forms the Lagrangian of the system, which is the difference between kinetic and potential energy. As next step, the forces acting on the system need to be expressed as generalized forces. By insertion of the Lagrangian and the generalized forces into the Euler-Lagrange equation the model is obtained.

A lot of publications about modeling quad-rotor helicopters exist and were used in this thesis ([KKP09], [HRW+04], [BNS04], [Bre08], [CLD05], [HHWT07], [PMC06], [McK04]). The approach using the formalism of Lagrange was already applied to this structure by many researchers[1], but according to [KKP09] with lack of mathematical rigorousness. [KKP09] write, that previously developed models are subject to dynamical inconsistencies because they are expressed using two coordinate systems (body- and local frame) without proper transformation. However, they present a solution to this problem, which is used as a basis of the derivation of the equations of motion in this work. Therefore, the desired equations are developed in both, the body-fixed and an inertial frame. The equations describing the motion of the quadrocopter relative to an inertial frame can be obtained by means of Lagrange's equations. With the equations of motion relative to body frame it is a bit more complicated: The orientation of the axes of the body-fixed frame relative to the inertial frame is defined by a set of rotations about nonorthogonal axes (See chap. 3.2 for details on that). Thus, it is better to work with angular velocities about the orthogonal body-frame axes [Mei90]. In fact, this is the frame where these angular velocities are measured. As a result of that, the equations of motion have to be derived in quasi-coordinates. The term quasi-coordinates means, that the actual coordinates, where the angular velocities are the derivative of, cannot be strictly defined. In other words, the angular velocities cannot be integrated to obtain angular coordinates - hence they are referred to as quasi-coordinates. Lagrange's equations in terms of quasi coordinates are presented in [Mei90] and the associated method of deriving equations of motion is called the quasi-Lagrange method.

The following chapter describes the principal characteristics of quadrotor-helicopters. It shows intuitively which movements are possible. Sec. 3.2 shows the quadrotor-helicopter in the reference frames used for modeling. Also transformations between these frames are

[1] A list of some can be found in [KKP09]

presented. The second section (3.3) uses the formalism of Lagrange to find the equations of motion in an inertial frame. In Sec. 3.4 a quasi-Lagrange method is used to derive the equations of motion in the body fixed frame. Finally, in sec. 3.5, a linearized model is presented, which is suitable for parameter identification. The validity of this model in a certain range is verified.

3.1 Quad-rotor Helicopter Characteristics

A quadrotor rotorcraft is characterized by its four rotors, which are typically aligned in a square, so that all rotors rotate in one plane. The cross structure connecting the four rotors with the base is typically thin and light. In the case of the Microdrone, it is part of the chassis which is made of carbon. All the propeller axes are perpendicular to the plane of the cross structure, so they are parallel and fixed. Each propeller is driven by a brushless DC-motor with external rotor. The blades of the rotors have fixed pitch and their air flows point downwards. Such quadrocopters have some advantages compared to conventional helicopters: Their mechanical structure is much simpler and most of the parts are rigid. In a conventional helicopter, there is a swashplate that is able to manipulate the attitude of the axis of rotation of the main rotor. Furthermore, pitch hinges, scissor links, teeter hinges, etc. make the rotor head of a conventional helicopter a complex mechanical assembly. In a conventional helicopter, a separate actuator is needed to compensate the yawing moment of the main rotor, namely the tail rotor. As the front and back propellers rotate clockwise and the other two counter-clockwise, yawing moments cancel out at a common angular velocity in the quadrotor-case.

Fig. 3.1 shows the quadrotor at hover. Ω_F,Ω_R,Ω_B,Ω_L stand for the front, right, back and left propeller angular velocities, respectively. For hovering, all those must be equal to an angular velocity Ω_H. At this angular velocity, the rotors produce the amount of thrust that compensates the gravitational force. The straight red arrows mark the airflow direction. Its size should principally show the amount of thrust created by a rotor. $\mathbf{b} = \{e_x^b, e_y^b, e_z^b\}$ is the body fixed frame.

The four basic movements of a quadrotor-helicopter are:

Throttle is a collective input variation. Change in throttle will cause all rotors to change their rotational velocity by a similar amount. Within the operating range, higher rotational velocity will create more thrust and thus accelerate the drone in $-e_z^b$ direction. Lower rotational velocity will create less thrust and not be able to compensate the whole

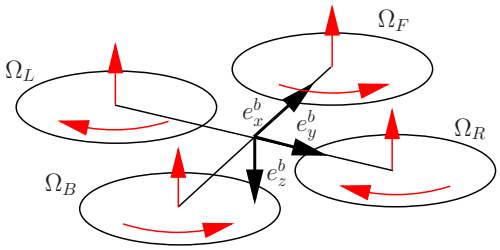

Figure 3.1: Schematic quadrotor at hover

gravitational force any more. As a result, the quadrotor will accelerate downwards. This relation is shown in fig. 3.2, where $\Delta_T > 0$.

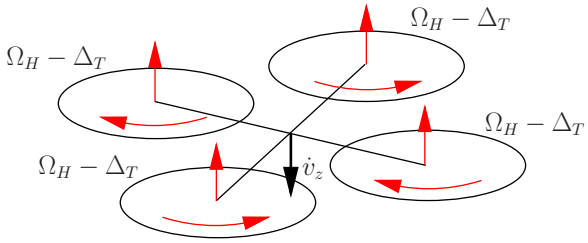

Figure 3.2: Throttle movement

Rolling movement in positive roll-angle direction is achieved by increasing Ω_L and decreasing Ω_R. Again, the thrusts will be affected and create a torque around the body-frame x-axis. The overall thrust keeps constant. See fig. 3.3 for a sketch. Note that with a positive roll angle the direction of the collective thrust will change. Hence, the component that acts against the gravitational force will decrease, while a component in lateral direction will emerge. As a result, the quadrotor will loose height and drift off in starboard-direction.

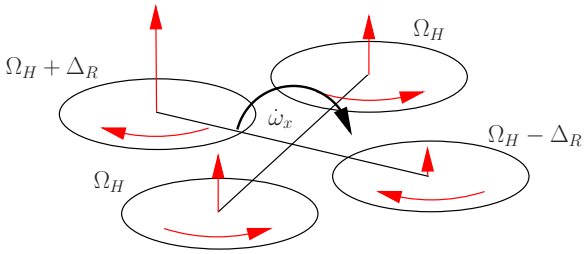

Figure 3.3: Rolling movement

Pitching movement is a result of differences in the angular velocity between the front and back rotor. The principal effects are the same as in rolling movement, only the direction is different. Fig. 3.4 sketches the pitch-command and its results.

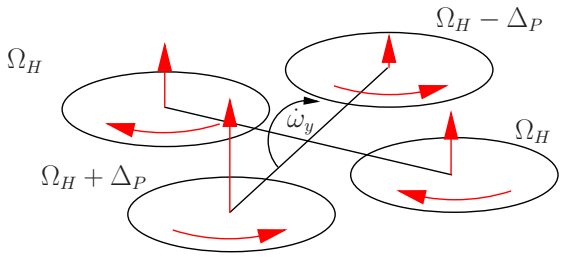

Figure 3.4: Pitching movement

Interesting to notice is, that the quadrotor principally has 6 DOF: 3 for the position and 3 for the angles. But, as stated in the paragraph about rolling motion, the roll-input always effects both, roll angle and starboard lateral motion. The same is valid for pitch movement. [2]

Yawing motion is a result of the collective drag torque of the rotors. Each rotor generates a drag-torque in counter direction to the propeller rotation. Thus, if you consider rotation direction as shown in fig. 3.5, then the drag moment vector for the front/back rotor pair will point downwards. Right and left rotor will thus create a drag torque in the opposite direction, namely upwards. If the angular acceleration in the front/back pair is bigger than in the left/right rotor pair, then the resultant moment vector will point downwards, that is in positive body-frame z-direction. Hence, the angular velocity about that axis will increase.

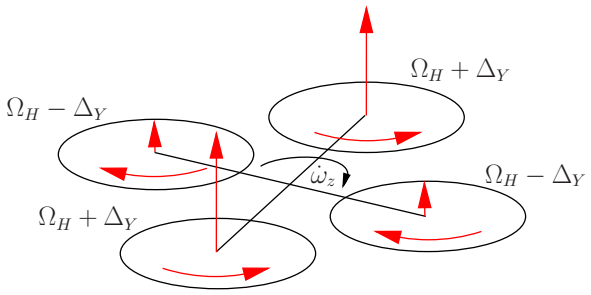

Figure 3.5: Yawing movement

[2]Obviously, the effect of pitch and roll on the altitude can be compensated by increasing the total thrust.

3.2 Kinematics

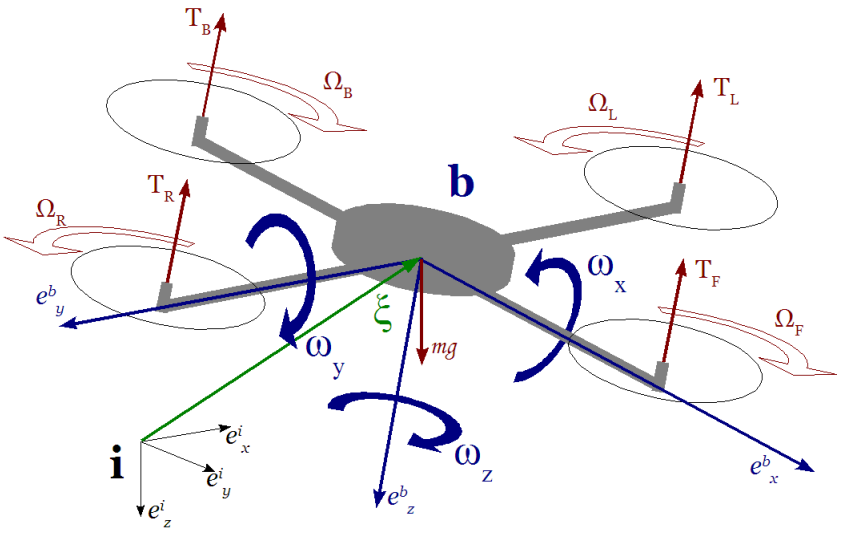

Figure 3.6: The quad-rotor in an inertial frame

Figure 3.6 shows a quadrotor rotorcraft in a right-handed inertial frame, which is denoted by $\mathbf{i} = \{e_x^i, e_y^i, e_z^i\}$. The vector $\boldsymbol{\xi^i} = (x^i, y^i, z^i) \in \mathbf{R^3}$ points from the origin of the inertial frame $\mathbf{i}$ to the origin of the body-fixed frame $\mathbf{b} = \{e_x^b, e_y^b, e_z^b\}$, which is placed at the robot's center of mass (COM). In all further derivation, it is assumed that the COM lies in the same plane as the rotors rotation. e_x^b is oriented in forward direction, so it is pointing from the COM to the front rotor. e_y^b is the unit vector in a direction towards the right (starboard) rotor. Together with e_z^b, which points down, these three vectors span the right-handed body-fixed frame. $\boldsymbol{\eta^i} = (\phi^i, \theta^i, \psi^i) \in \mathbf{R^3}$ represents the attitude of the rotorcraft in a special case of Euler angles. There the orientation of the body frame with respect to the inertial frame is given by the conventional sequence of rotations in aerodynamics [Ste04]: The first one is the yaw (ψ^i) rotation about the vertical inertial axis (e_z^i) and denoted as $\mathbf{R}_\psi$. This is followed by a pitch (θ^i) rotation about an intermediate span-wise axis e_y^{ii}, which was rotated about the yaw angle in the first step. The roll (ϕ^i) rotation is about the intermediate centerline axis e_x^{iii}. Notice, that, due to the rotation about intermediate axes, the coordinate system spanned by the rotation axes is not orthogonal!

A rotation matrix has two meanings: For example, $\mathbf{R_i^b}$ is the inertial frame expressed relative to the body frame. It's second usage is to transform a point from the $\mathbf{b}$-frame to

the **i**-frame. This rotation matrix can be computed by

$$\begin{aligned}
\mathbf{R}_{\mathbf{i}}^{\mathbf{b}} &= \mathbf{R}_\psi \mathbf{R}_\theta \mathbf{R}_\phi \\
&= \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi & c_\phi \end{bmatrix} \\
&= \begin{bmatrix} c_\theta c_\psi & c_\psi s_\theta s_\phi - s_\psi c_\phi & c_\psi s_\theta c_\phi + s_\psi s_\phi \\ s_\psi c_\theta & s_\psi s_\theta s_\phi + c_\psi c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix}
\end{aligned}$$

where c_θ stands for $cos(\theta)$ and s_θ for $sin(\theta)$. As this rotation matrix belongs to the Euclidean group of rotations in 3D (also written as $\mathbf{R} \in \mathcal{SO}(3)$), it has some special properties, where especially orthogonality is very helpful in our case:

- $\mathbf{R}^{-1} = \mathbf{R}^T$
- $\mathbf{R}^T\mathbf{R} = \mathbf{I}$

where $\mathbf{I}$ is the (3×3) identity matrix. So we can write down the inverse rotation matrix

$$\begin{aligned}
\mathbf{R}_{\mathbf{b}}^{\mathbf{i}} &= \mathbf{R}_{\mathbf{i}}^{\mathbf{b}^T} \\
&= \begin{bmatrix} c_\theta c_\psi & s_\psi s_\theta & -s_\theta \\ c_\psi s_\theta s_\phi - s_\psi c_\phi & s_\psi s_\theta s_\phi + c_\psi c_\phi & c_\theta s_\phi \\ c_\psi s_\theta c_\phi + s_\psi s_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi & c_\theta c_\phi \end{bmatrix}
\end{aligned}$$

Fig. 3.6 also shows body-frame angular rates. They are denoted by $\boldsymbol{\omega}_{\boldsymbol{ib}}^{\boldsymbol{b}} = (\omega_x^b, \omega_y^b, \omega_z^b)$ and are exactly those measured by the gyroscopes. As in the chapter on navigation & sensing, the notation means that $\boldsymbol{\omega}_{\boldsymbol{ib}}^{\boldsymbol{b}}$ is the angular velocity of the body frame **b** (subscript 2) relative to an inertial frame **i** (subscript 1) expressed in the body frame (superscript). For the components of the vector, the subscript denotes the rotation axis. Hereby, x stands for the e_x^b axis. So the angular velocities are measured about the orthogonal body-fixed frame axes. $\boldsymbol{\nu}^{\boldsymbol{b}} = (v_x^b, v_y^b, v_z^b)$ is the 3-dimensional vector of body-frame velocities. Its time derivative are the accelerations measured by the accelerometers of the IMU, except the gravitational acceleration, which is included in the specific force measured by the IMU.

In order to transform this velocity vector from the body to the inertial frame, the rotation matrix $\mathbf{R}_{\mathbf{i}}^{\mathbf{b}}$ is applied:

$$\dot{\boldsymbol{\xi}}^{\boldsymbol{i}} = \mathbf{R}_{\mathbf{i}}^{\mathbf{b}} \boldsymbol{\nu}^{\boldsymbol{b}}$$

Hence, the matrix $\mathbf{R}_{\mathbf{i}}^{\mathbf{b}}$ can also be understood as Jacobian. To simplify matters, the index specifying the reference frame of a variable is omitted from now on if the meaning of a variable is clear. For the computation of the transformation from body-frame angular rates to inertial angular velocities, some more steps are necessary. As the angular rates $\dot{\boldsymbol{\eta}}$ are about non-orthogonal, previously called "intermediate axis", the following transformation needs to be done[3]:

$$\begin{aligned} \boldsymbol{\omega} &= \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \mathbf{R}_{\phi}{}^{T} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \mathbf{R}_{\phi}{}^{T} \mathbf{R}_{\theta}{}^{T} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & -s_{\theta} \\ 0 & c_{\phi} & c_{\theta} s_{\phi} \\ 0 & -s_{\phi} & c_{\theta} c_{\phi} \end{bmatrix} \dot{\boldsymbol{\eta}} \\ &= \mathbf{T}_{\mathbf{b}}^{\mathbf{i}} \dot{\boldsymbol{\eta}} \end{aligned} \tag{3.1}$$

The inverse of this transformation matrix is the sought after transformation of body-frame angular rates to inertial ones.

$$\dot{\boldsymbol{\eta}} = \mathbf{T}_{\mathbf{i}}^{\mathbf{b}} \boldsymbol{\omega} \tag{3.2}$$

with

[3]The same result is achieved by solving $\dot{\mathbf{R}}_{\mathbf{i}}^{\mathbf{b}} \mathbf{R}_{\mathbf{i}}^{\mathbf{b}}{}^{T} = \mathbf{S}(\boldsymbol{\omega})$, where $\mathbf{S}(\boldsymbol{\omega})$ means the skew-symmetric matrix [SHV06]

$$\begin{aligned} \mathbf{T_i^b} &= \mathbf{T_b^i}^{-1} \\ &= \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{bmatrix} \end{aligned}$$

For small ϕ and θ both matrices are approximately equal to the unity matrix. Moreover, notice that $\mathbf{T_i^b}$ and $\mathbf{T_b^i}$ are not orthogonal. But the situation is even worse: $\mathbf{T_i^b}$ is undefined for a pitch angle of $\theta = \pm 90°$. The system approaches an uncontrollable state in this case [KKP09]. Thus, such high pitch angles are not considered in this work. The fact, that an uncontrollable state is where $\theta = \pm 90°$ can be understood in the following way: All the forces created by the rotors point in a direction perpendicular to the direction of the gravitational field. Hence, there is no possibility to influence the height any more, so the altitude gets uncontrollable. Obviously, that does not mean that there is no maneuver capable of passing through such a pitch angle: The roll and pitch command could be used to rotate the throttle vector out of this state.

3.3 Inertial Frame Dynamics

As already mentioned, the rotor-crafts dynamics in inertial frame is obtained via the formalism of Lagrange [Hof04], [SHV06].

Let the generalized coordinates be

$$\boldsymbol{q} = (x, y, z, \phi, \theta, \psi) \in \mathbf{R^6}$$

Then, the translational kinetic energy of the quad-rotor is given by

$$K_{Trans} = \frac{m}{2} \boldsymbol{\nu}^T \boldsymbol{\nu} \tag{3.3}$$

where m is the mass of the robot and $\boldsymbol{\nu}$ are the velocities calculated according to relation (3.1). The rotational kinetic energy is given by

$$K_{Rot} = \frac{1}{2}\boldsymbol{\omega}^T \mathbf{J} \boldsymbol{\omega} \tag{3.4}$$

where the inertia tensor $\mathbf{J}$ is given by

$$\mathbf{J} = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{yx} & J_{yy} & J_{yz} \\ J_{zx} & J_{zy} & J_{zz} \end{bmatrix}$$

If the mass distribution of the robot body is symmetric with respect to the body frame, all elements except the main diagonal of the inertia tensor in the body frame are equal to zero. Furthermore, due to symmetries in e_x^b and e_y^b direction[4], the assumption $J_{xx} = J_{yy}$ leads to another simplification. Actually, even letting $\mathbf{J}$ be constant is a simplification. That is, because when being accurate, also camera tilt influences inertia. However, this work does not go that far into detail. Hence,

$$\mathbf{J} = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix} \tag{3.5}$$

If we assume the robot to be planar, one could still further simplify the inertia tensor by using the perpendicular axis theorem. That leads to the following, very simple inertia tensor. The values calculated e.g. in [Bre08] or [PMC06] show, that this assumption is too rough (although these structures are not geometrically comparable to the Microdrone) and the moments of inertia tensor defined in eq. (3.5) will be used for further progress.

$$\mathbf{J} = \begin{bmatrix} J & 0 & 0 \\ 0 & J & 0 \\ 0 & 0 & 2J \end{bmatrix} \tag{3.6}$$

Now that we have defined the kinetic energy of the system, let us proceed with the potential energy. The gravitational field causes an acceleration in inertial frame e_z^i direction. Thus,

[4] Or, in other words: the flyer's body mass distribution is also symmetric about vertical planes at $\psi = \pm 45°$.

$$V = -mgz \tag{3.7}$$

where g is the gravitational acceleration, which is assumed to be constant. Then, the Lagrangian is given by

$$\begin{aligned} \mathcal{L} &= K_{Trans} + K_{Rot} - V \\ &= \frac{m}{2}\boldsymbol{\nu}^T\boldsymbol{\nu} + \frac{1}{2}\boldsymbol{\omega}^T\mathbf{J}\boldsymbol{\omega} + mgz \\ &= \mathcal{L}(q, \dot{q}) \end{aligned} \tag{3.8}$$

In order to proceed with the derivation of the equations of motion, the generalized forces need to be defined. Formulating the thoughts from chap. 3.1 in a mathematical way, the generalized forces and moments in the body-frame are

$$\begin{aligned} \boldsymbol{\tau^b} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \\ 0 & -l & 0 & l \\ -l & 0 & l & 0 \\ -q_F/t_F & q_F/t_F & -q_F/t_F & q_F/t_F \end{bmatrix} \begin{bmatrix} T_F \\ T_R \\ T_B \\ T_L \end{bmatrix} \\ &= \mathbf{E} \cdot \boldsymbol{t} \\ &= \begin{bmatrix} 0 \\ 0 \\ u_T \\ u_R \\ u_P \\ u_Y \end{bmatrix} \end{aligned} \tag{3.9}$$

where l is the lever arm length from the COM to the rotor center, q_F is the torque or drag factor and t_F is the thrust factor. Note, that the gyroscopic torques created by the propellers' rotation are neglected here[5]. This is a quite common simplification, which most

[5]The gyroscopic torque of each propeller is $\boldsymbol{\tau_P} = \boldsymbol{\omega} \times \begin{bmatrix} 0 & 0 & J_P\Omega_k \end{bmatrix}^T$ where $k = \{F, R, B, L\}$ and J_P is the propeller inertia

of the referred publications made. The problem with these qyroscopic torques is, that they depend on the propeller angular rate. Thus, the model would loose input-affinity! See e.g. [Bou07] for a model where rotor gyroscopic torques are included.

Each thrust T_k with $k = \{F, R, B, L\}$ is given by

$$T_k = t_F \Omega_k^2$$

Details on that topic are given in the chapter about aerodynamics (4.2). In eq. (3.9), u_T, u_R, u_P, u_Y stand for the throttle, roll, pitch and yaw input, respectively. $\boldsymbol{\tau^b}$ is still expressed in the body frame, so a transformation to inertial frame is necessary.

$$\boldsymbol{\tau^i} = \begin{bmatrix} \mathbf{R}_\mathbf{i}^\mathbf{b} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_\mathbf{i}^\mathbf{b} \end{bmatrix} \boldsymbol{\tau^b}$$

here, $\mathbf{0}$ stands for the (3×3) zero-matrix. Now that the Lagrangian and the generalized forces are defined, we can use the Euler-Lagrange equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}} - \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}} = \boldsymbol{\tau^i} \tag{3.10}$$

to derive the equations of motion in the inertial frame. In this project that was done using a Maple-Script.

$$\begin{aligned} m\ddot{x} &= (s_\psi s_\phi + c_\psi s_\theta c_\phi) u_T \\ m\ddot{y} &= (-c_\psi s_\phi + s_\psi s_\theta c_\phi) u_T \\ m(\ddot{z} - g) &= c_\theta c_\phi u_T \\ \mathbf{M}_\eta \ddot{\boldsymbol{\eta}} + \frac{1}{2}\dot{\mathbf{M}}_\eta \dot{\boldsymbol{\eta}} &= \begin{bmatrix} u_R \\ c_\phi u_P - s_\phi u_Y \\ -s_\theta u_R + c_\theta s_\phi u_P + c_\theta c_\phi u_Y \end{bmatrix} \end{aligned}$$

with

$$\mathbf{M}_\eta = \begin{bmatrix} J_{xx} & 0 & -J_{xx}s_\theta \\ 0 & J_{yy}c_\phi^2\phi + J_{zz}s_\phi^2 & (J_{yy} - J_{zz})c_\phi c_\theta s_\phi \\ -J_{xx}s_\theta & (J_{yy} - J_{zz})c_\phi c_\theta s_\phi & J_{xx}s_\theta^2 s_\phi^2 + J_{zz}c_\theta^2 c_\phi^2 \end{bmatrix} \tag{3.11}$$

which is symmetric and positive definite. It is seen, that especially the translational equations of motion are pretty simple in the inertial frame. The whole system can also be written in the well-known matrix-vector form, where a computational efficient way of computing the matrices is shown in [SHV06, chapter 7.3]:

$$\mathbf{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \mathbf{C}(\dot{\boldsymbol{q}}, \boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau}^{\boldsymbol{i}}(\boldsymbol{q}) \tag{3.12}$$

where the matrices $\mathbf{M}$ (inertia- or mass-matrix) and $\mathbf{C}$ (Coriolis and centrifugal matrix) have some interesting properties [SHV06, chapter 7.5], which paves the way for many control design procedures (e.g. [Kha02], [Bou07], [SHV06]). However, that is not further investigated here as this is not a task in this project.

3.4 Body-Frame Dynamics

As afore mentioned, the body-fixed frame dynamics need to be treated by the quasi-Lagrange method. The overall procedure is quite the same: First, kinetic and potential energy in the body-frame are calculated. Then one can evaluate the Lagrangian and derive the equations of motion by using a Lagrange's equation in terms of quasi-coordinates.

As in the inertial frame-case, find the Lagrangian using eq. (3.3), (3.4), (3.7) and (3.8). But now, see the Lagrangian as

$$\mathcal{L} \quad = \quad \mathcal{L}(\boldsymbol{\xi}, \boldsymbol{\nu}, \boldsymbol{\eta}, \boldsymbol{\omega})$$

According to [Mei90, chapter 2.7], the equations of motion with body-frame derivatives can be found using

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\nu}}\right) + \boldsymbol{\omega} \times \frac{\partial \mathcal{L}}{\partial \boldsymbol{\nu}} - \mathbf{R}_{\mathbf{b}}^{\mathbf{i}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\xi}} = \begin{bmatrix} 0 \\ 0 \\ u_T \end{bmatrix} \tag{3.13}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\omega}}\right) + \boldsymbol{\omega} \times \frac{\partial \mathcal{L}}{\partial \boldsymbol{\omega}} + \boldsymbol{\nu} \times \frac{\partial \mathcal{L}}{\partial \boldsymbol{\nu}} - \mathbf{T}_{\mathbf{b}}^{\mathbf{i}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\eta}} = \begin{bmatrix} u_R \\ u_P \\ u_Y \end{bmatrix} \tag{3.14}$$

Again, this was executed using a Maple Script, resulting in

$$\begin{aligned} m(\dot{v}_x + \omega_y v_z - \omega_z v_y + s_\theta g) &= 0 \\ m(\dot{v}_y + \omega_z v_x - \omega_x v_z - c_\theta s_\phi g) &= 0 \\ m(\dot{v}_z + \omega_x v_y - \omega_y v_x - c_\theta c_\phi g) &= u_T \\ J_{xx}\dot{\omega}_x + (J_{zz} - J_{yy})\omega_y \omega_z &= u_R \\ J_{yy}\dot{\omega}_y + (J_{xx} - J_{zz})\omega_z \omega_x &= u_P \\ J_{zz}\dot{\omega}_z + (J_{yy} - J_{xx})\omega_x \omega_y &= u_Y \end{aligned} \tag{3.15}$$

which are the equations of motion in the body frame. Especially the angular dynamics are simpler here than in the inertial frame. Moreover, they depend on $\boldsymbol{\omega}$, which is measured in this frame. Also the control forces show up directly in the right-hand sides.

For control design purposes, it is advantageous [KKP09], [Bre08], [Bou07], to express the equations of motion in a hybrid reference frame. That is, the translational equations are relative to the inertial frame, while the angular ones reference the body frame. Hence, the equations of motion for control purposes of quad-rotor helicopters are:

$$\begin{aligned} m\ddot{x} &= (s_\psi s_\phi + c_\psi s_\theta c_\phi) u_T \\ m\ddot{y} &= (-c_\psi s_\phi + s_\psi s_\theta c_\phi) u_T \\ m(\ddot{z} - g) &= c_\theta c_\phi u_T \\ J_{xx}\dot{\omega}_x + (J_{zz} - J_{yy})\omega_y \omega_z &= u_R \\ J_{yy}\dot{\omega}_y + (J_{xx} - J_{zz})\omega_z \omega_x &= u_P \\ J_{zz}\dot{\omega}_z + (J_{yy} - J_{xx})\omega_x \omega_y &= u_Y \end{aligned} \tag{3.16}$$

However, for simulation purposes a more detailed model might be desired. It will come up, that not just simulation but also parameter identification will profit from a special extension, namely friction. The type of friction that is acting on the quadrotor at flight is drag, which is proportional to the squared velocity[6]. More precisely, that is

$$f_d = c_D A \frac{\rho_A}{2} v|v|$$

where A is the projected area, ρ_A is the air density (see 4.2), c_D is the (shape-dependent) drag coefficient and v is the velocity of the body relative to the air. Actually, this is the same as wind would act onto the rotorcraft, but one must think of v being the velocity of the wind relative to the body frame then. This force already occurred in this text in the section on aerodynamics (4.2), where it was one of the sources of the rotor torque in blade element theory[7]. There, the force was proportional to the squared tangential velocity of a blade element. Since the tangential velocity is

$$v_t = \omega \cdot r$$

and, in order to find the torque, a further multiplication with the radius is necessary

$$\tau_d = c_D A \frac{\rho_A}{2} \omega|\omega| r^3$$

Introducing these drag-terms into the generalized force, the equations of motion in the body frame are:

[6] The formulas shown are valid for the scalar case only.
[7] (4.8), the first term

$$
\begin{aligned}
m(\dot{v}_x + \omega_y v_z - \omega_z v_y + s_\theta g) &= -c_D A_{B,x} \frac{\rho_A}{2} v_x |v_x| \\
m(\dot{v}_y + \omega_z v_x - \omega_x v_z - c_\theta s_\phi g) &= -c_D A_{B,y} \frac{\rho_A}{2} v_y |v_y| \\
m(\dot{v}_z + \omega_x v_y - \omega_y v_x - c_\theta c_\phi g) &= u_T - c_D (A_{B,z} + n A_{P,z}) \frac{\rho_A}{2} v_z |v_z| \\
J_{xx}\dot{\omega}_x + (J_{zz} - J_{yy})\omega_y \omega_z &= u_R - c_D \frac{n}{2} A_{P,z} \frac{\rho_A}{2} \omega_x |\omega_x| l^3 \\
J_{yy}\dot{\omega}_y + (J_{xx} - J_{zz})\omega_z \omega_x &= u_P - c_D \frac{n}{2} A_{P,z} \frac{\rho_A}{2} \omega_y |\omega_y| l^3 \\
J_{zz}\dot{\omega}_z + (J_{yy} - J_{xx})\omega_x \omega_y &= u_Y - c_D \frac{n}{2} A_{p,x} \frac{\rho_A}{2} \omega_z |\omega_z| l^3
\end{aligned} \tag{3.17}
$$

where n is the number of rotors, $A_{P,z}$ the rotor area, $A_{B,x}$, $A_{B,y}$ and $A_{B,z}$ are the projected areas of the base in direction of e_x^b, e_y^b, e_z^b, respectively.

For high-accuracy dynamics simulation, this model could still be further extended. [Bou07] describes some possible extensions, such as

- those mentioned in the aerodynamics chapter about forward flight
- propeller gyro effect
- ...

However, the autonomous systems lab at ETH Zürich offers a detailed simulation model implemented in Matlab/Simulink[8].

3.5 Linearization

In order to provide a complete model of a quadrotor helicopter, also the parameters of the model need to be known. Generally, there are two methods that come to mind:

- Computation by using physical laws. This was done in sec. 4.1 and 4.2.
- Experimental determination of the parameters. In control science, this is called parameter identification which can be seen as a subdivision of system identification.

In system identification, many methods exist for linear time invariant systems. The goal in this chapter is to find transfer functions of a linearization of the system equations of motion (3.17), so that its parameters can be identified. Since the measurements from the

[8] http://asl.epfl.ch/research/projects/VtolIndoorFlying/documents/NewModel_PUBLIC_V1.1.zip

IMU deliver linear accelerations and angular velocities in the body-frame, the following transfer functions are of interest:

$$\begin{aligned} G_{T,1} &: \bar{u}_T \mapsto \dot{v}_z \\ G_{R,1} &: \bar{u}_R \mapsto \dot{v}_y \\ G_{R,2} &: \bar{u}_R \mapsto \omega_x \\ G_{P,1} &: \bar{u}_P \mapsto \dot{v}_x \\ G_{P,2} &: \bar{u}_P \mapsto \omega_y \\ G_{Y,1} &: \bar{u}_Y \mapsto \omega_z \end{aligned}$$

where the inputs $\bar{u}_k$ with $k = \{T, R, P, Y\}$ are

$$\bar{\boldsymbol{u}} = \begin{bmatrix} \bar{u}_T \\ \bar{u}_R \\ \bar{u}_P \\ \bar{u}_Y \end{bmatrix} \tag{3.18}$$

$$= \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} \Omega_F^2 \\ \Omega_R^2 \\ \Omega_B^2 \\ \Omega_L^2 \end{bmatrix} \tag{3.19}$$

This is very similar to what is done in eq. (3.9). But, in order to also include the aerodynamic parameters t_F and q_F in the identification process, the inputs need to be defined as shown above. The relation between $\bar{u}_k$ and u_k is given by

$$\begin{bmatrix} u_T \\ u_R \\ u_P \\ u_Y \end{bmatrix} = diag(t_F, lt_F, lt_F, q_F)\bar{\boldsymbol{u}}$$

where *diag* is the (4×4) diagonal matrix with the values of the arguments in its main diagonal. Notice that the inputs and outputs of the sought-after transfer functions ref-

erence only the body frame. Because of that, it is sufficient to linearize the body frame equations of motion (3.17) only. The state vector is chosen to be

$$\boldsymbol{x} = (v_x, v_y, v_z, \phi, \theta, \psi, \omega_x, \omega_y, \omega_z)^T$$

Inertial frame positions ξ are omitted here, as they do not appear in (3.17) explicitly. In the well-known form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \bar{\boldsymbol{u}})$ the equations are (see eq. (3.17) and (3.2)):

$$\begin{aligned}
\dot{v}_x &= -\omega_y v_z + \omega_z v_y + s_\theta g - \frac{c_D\ A_{B,x}\ \rho_A}{2m} v_x |v_x| \\
\dot{v}_y &= -\omega_z v_x + \omega_x v_z + c_\theta s_\phi g - \frac{c_D\ A_{B,y}\ \rho_A}{2m} v_y |v_y| \\
\dot{v}_z &= -\omega_x v_y + \omega_y v_x + c_\theta c_\phi g + \frac{\bar{u}_T}{m} - \frac{c_D\ (A_{B,z} + n A_{P,z})\ \rho_A}{2m} v_z |v_z| \\
\dot{\phi} &= \omega_x + s_\phi t_\theta \omega_y + c_\phi t_\theta \omega_z \\
\dot{\theta} &= c_\phi \omega_y - s_\phi \omega_z \\
\dot{\psi} &= \frac{s_\phi}{c_\theta} \omega_y + \frac{c_\phi}{c_\theta} \omega_z \\
\dot{\omega}_x &= -(J_{zz} - J_{yy}) \omega_y \omega_z + \frac{l\ t_F\ \bar{u}_R}{J_{xx}} - \frac{c_D\ n\ A_{P,z}\ \rho_A}{4 J_{xx}} \omega_x |\omega_x| l^3 \\
\dot{\omega}_y &= -(J_{xx} - J_{zz}) \omega_z \omega_x + \frac{l\ t_F\ \bar{u}_P}{J_{yy}} - \frac{c_D\ n\ A_{P,z}\ \rho_A}{4 J_{yy}} \omega_y |\omega_y| l^3 \\
\dot{\omega}_z &= -(J_{yy} - J_{xx}) \omega_x \omega_y + \frac{q_F\ \bar{u}_Y}{J_{zz}} - \frac{c_D\ n\ A_{P,x}\ \rho_A}{4 J_{zz}} \omega_z |\omega_z| l^3
\end{aligned} \tag{3.20}$$

An equilibrium is found at[9]

$$\begin{aligned}
\boldsymbol{x}_E &= \boldsymbol{0} \\
\bar{\boldsymbol{u}}_E &= \begin{bmatrix} -mg & 0 & 0 & 0 \end{bmatrix}^T
\end{aligned}$$

What we are interested in now, is a linear model for small deviations out of this equilibrium, at which all velocities are zero. Such a model is usually found by building the Jacobian of (3.21) with respect to $\boldsymbol{x}$. Since the absolute-operator occurs in $\boldsymbol{f}$, it is not differentiable at zero. Moreover, the squared speed drag model is not valid for small velocities. There, the linear drag dominates, which is modeled by

[9] w.l.o.g it was assumed that $\psi_E = 0$

$$f_d = -bv$$

Hence, $\boldsymbol{f}$ simplifies to

$$\begin{aligned}
\dot{v}_x &= -\omega_y v_z + \omega_z v_y + s_\theta g - \frac{b_x}{m} v_x \\
\dot{v}_y &= -\omega_z v_x + \omega_x v_z + c_\theta s_\phi g - \frac{b_y}{m} v_y \\
\dot{v}_z &= -\omega_x v_y + \omega_y v_x + c_\theta c_\phi g + \frac{\bar{u}_T}{m} - \frac{b_z}{m} v_z \\
\dot{\phi} &= \omega_x + s_\phi t_\theta \omega_y + c_\phi t_\theta \omega_z \\
\dot{\theta} &= c_\phi \omega_y - s_\phi \omega_z \\
\dot{\psi} &= \frac{s_\phi}{c_\theta} \omega_y + \frac{c_\phi}{c_\theta} \omega_z \\
\dot{\omega}_x &= -(J_{zz} - J_{yy}) \omega_y \omega_z + \frac{l \; t_F \; \bar{u}_R}{J_{xx}} - \frac{b_\omega}{J_{xx}} \omega_x l^2 \\
\dot{\omega}_y &= -(J_{xx} - J_{zz}) \omega_z \omega_x + \frac{l \; t_F \; \bar{u}_P}{J_{yy}} - \frac{b_\omega}{J_{yy}} \omega_y l^2 \\
\dot{\omega}_z &= -(J_{yy} - J_{xx}) \omega_x \omega_y + \frac{q_F \; \bar{u}_Y}{J_{zz}} - \frac{b_{\omega,z}}{J_{zz}} \omega_z l^2
\end{aligned} \tag{3.21}$$

The linearized system is then given by

$$\begin{aligned}
\dot{\tilde{\boldsymbol{x}}} &= \mathbf{A}\tilde{\boldsymbol{x}} + \mathbf{B}\tilde{\boldsymbol{u}} \\
\tilde{\boldsymbol{y}} &= \mathbf{C}\tilde{\boldsymbol{x}} + \mathbf{D}\tilde{\boldsymbol{u}}
\end{aligned}$$

with

$$\begin{aligned}
\boldsymbol{x} &= \tilde{\boldsymbol{x}} + \boldsymbol{x}_E = \boldsymbol{x} \\
\bar{\boldsymbol{u}} &= \tilde{\boldsymbol{u}} + \bar{\boldsymbol{u}}_E
\end{aligned}$$

The system matrices $\mathbf{A}, \mathbf{B}$ are determined by Taylor expansion and stopping after the linear term:

$$\mathbf{A} = \left. \frac{\partial \boldsymbol{f}(\boldsymbol{x}, \bar{\boldsymbol{u}})}{\partial \boldsymbol{x}} \right|_{\substack{\boldsymbol{x}=\boldsymbol{x}_E \\ \bar{\boldsymbol{u}}=\bar{\boldsymbol{u}}_E}} \qquad \mathbf{B} = \left. \frac{\partial \boldsymbol{f}(\boldsymbol{x}, \bar{\boldsymbol{u}})}{\partial \bar{\boldsymbol{u}}} \right|_{\substack{\boldsymbol{x}=\boldsymbol{x}_E \\ \bar{\boldsymbol{u}}=\bar{\boldsymbol{u}}_E}}$$

$$\mathbf{A} = \begin{bmatrix} -b_x & 0 & 0 & 0 & -g & 0 & 0 & 0 & 0 \\ 0 & -b_y & 0 & g & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -b_z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{b_\omega}{J_{xx}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{b_\omega}{J_{yy}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{b_{\omega,z}}{J_{zz}} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ t_F & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{t_F\, l}{J_{xx}} & 0 & 0 \\ 0 & 0 & \frac{t_F\, l}{J_{yy}} & 0 \\ 0 & 0 & 0 & \frac{q_C}{J_{zz}} \end{bmatrix}$$

Choosing $\boldsymbol{y} = \begin{bmatrix} \dot{v}_x & \dot{v}_y & \dot{v}_z & \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ leads to the following $\mathbf{C}$ and $\mathbf{D}$ matrices

$$\mathbf{C} = \begin{bmatrix} -b_x & 0 & 0 & 0 & -g & 0 & 0 & 0 & 0 \\ 0 & -b_y & 0 & g & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -b_z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ t_F & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Then, application of the Laplace transform and assuming $\tilde{\boldsymbol{x}}(0) = \mathbf{0}$ leads to the well-known formula

$$\mathbf{G} = \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D} \tag{3.22}$$

$$= \begin{bmatrix} 0 & 0 & -\frac{g t_F\, l}{(sJ_{yy}+b_\omega)(s+b_x)} & 0 \\ 0 & \frac{g t_F\, l}{(sJ_{xx}+b_\omega)(s+b_y)} & 0 & 0 \\ \frac{t_F\, s}{s+b_z} & 0 & 0 & 0 \\ 0 & \frac{t_F\, l}{sJ_{xx}+b_\omega} & 0 & 0 \\ 0 & 0 & \frac{t_F\, l}{sJ_{yy}+b_\omega} & 0 \\ 0 & 0 & 0 & \frac{q_F}{sJ_{zz}+b_{\omega,z}} \end{bmatrix} \tag{3.23}$$

where $\mathbf{G}$ is the sought-after transfer matrix, which will be used for parameter identification in chap. 4.3.2.

Chapter 4

Parameter Identification

Contents

4.1 Inertia Calculation

Based on geometrical information about the Microdrone structure[1] and rough weight measurements, an estimate for the systems moments of inertia tensor is calculated. For simplification, the flyer was divided into 4 simple structures. The inertias of these are calculated and summed up to get the principal moments J_{xx}, J_{yy}, J_{zz} of the inertia tensor $\mathbf{J}$. The simple parts were chosen to be:

- Propeller
- Motor
- Arm from the base to a rotor
- Base (contains electronics, battery, ...)

The moments of inertia of each part are calculated about a coordinate system placed in the part's COM. Their displacement relative to the body-fixed frame is then treated using Steiner's theorem.

[1] Available at: http://www.microdrones.com/download/md4-200_technical_drawing.pdf

Strongly simplifying, a propeller is modeled as a cylindrical plate with decreasing mass density [Bre08]. Its weight m_P was measured to be 11 $[g]$. The radius $R_P = 180$ $[mm]$ was extracted from the technical drawing, the height H_P will not be needed for calculation. Thus, its moment of inertia about a vertical rotation axis passing through its center of mass is given by:

$$\begin{aligned}
J_{P,z} &= \int\int\int (x^2 + y^2)\ \rho(x,y)\ dx\ dy\ dz \\
&= \int_{-H_P/2}^{H_P/2} \int_{0}^{R_P} \int_{0}^{2\pi} (r^2 sin^2(\alpha) + r^2 cos^2(\alpha))\ \rho(r)\ r\ d\alpha\ dr\ dh \\
&= \frac{m_P}{\pi\ R_P^2 H_P} \int_{-H_P/2}^{H_P/2} \int_{0}^{R_P} \int_{0}^{2\pi} \frac{1}{r}(r^2 sin^2(\alpha) + r^2 cos^2(\alpha))\ r\ d\alpha\ dr\ dh \\
&= \frac{m_P}{\pi\ R_P^2 H_P} \int_{-H_P/2}^{H_P/2} \int_{0}^{R_P} 2r^2\pi\ dr\ dh \\
&= \frac{2m_P}{R_P^2 H_P} \int_{-H_P/2}^{H_P/2} \frac{R_P^3}{3}\ dh \\
&= 2m_P \left(\frac{R_P}{3}\right) \\
&= 0.953 \cdot 10^{-3}\ [kg\ m^2]
\end{aligned}$$

In step two, a switch from cartesian to cylindric coordinates was performed. The density was assumed to decrease reciprocal with the radius and $\pi\ R_P^2\ H_P$ is the volume of the plate:

$$\rho(r) = \frac{m_P}{\pi\ R_P^2\ H_P\ r}$$

Due to the perpendicular axis theorem,

$$\begin{aligned}
J_{P,x} &= J_{P,y} = \frac{J_{P,z}}{2} \\
&= 0.477 \cdot 10^{-3}\ [kg\ m^2]
\end{aligned}$$

The conditions for applying the theorem are assumed to be met, since $R_P \gg H_P$. For all parts from now on, constant mass density ρ is assumed. Furthermore, $m = \rho V$ will be used without being mentioned, where V is the volume of the body.

The motors are modeled as simple cylinders with radius $R_M = 30\ [mm]$, height $H_M = 30\ [mm]$ and mass $m_M = 25\ [g]$. The integration is shown once more, then it is omitted since the procedure always remains the same.

$$\begin{aligned}
J_{M,x} &= \int\int\int (y^2 + z^2)\ \rho\ dx\ dy\ dz \\
&= \int_{-H_M/2}^{H_M/2} \int_{0}^{R_M} \int_{0}^{2\pi} (r^2 sin^2(\alpha) + h^2)\ \rho\ r\ d\alpha\ dr\ dh \\
&= \rho \int_{-H_M/2}^{H_M/2} \int_{0}^{R_M} (r^3\pi + h^2\ r\ 2\pi)\ dr\ dh \\
&= \rho \left(\frac{H_M R^4 \pi}{4} + \frac{H_M^3}{12} \frac{R_M^2}{2} 2\pi \right) \\
&= m_M \left(\frac{R_M^2}{4} + \frac{H_M^3}{12} \right) \\
&= 7.5 \cdot 10^{-6}\ [kg\ m^2]
\end{aligned}$$

Because of symmetry, $J_{M,y} = J_{M,x}$. Similarly, the other moment of inertia for this body can be calculated:

$$\begin{aligned}
J_{M,z} &= \int\int\int (x^2 + y^2)\rho\ dx\ dy\ dz \\
&= m_M \left(\frac{R_M^2}{2} \right) \\
&= 1.125 \cdot 10^{-5}\ [kg\ m^2]
\end{aligned}$$

Next, the base - which is also modeled as a cylinder - is tackled. The technical drawing shows the cylinder radius $R_B = 82.5\ [mm]$, the height $H_B = 100\ [mm]$ and the mass was measured $m_B = 780\ [g]$. Hence,

$$\begin{aligned} J_{B,x} &= m_B \left(\frac{R_B^2}{4} + \frac{H_B^3}{12} \right) \\ &= 1.977 \cdot 10^{-3} [kg\ m^2] \\ J_{B,z} &= m_B \left(\frac{R_B^2}{2} \right) \\ &= 2.654 \cdot 10^{-3}\ [kg\ m^2] \end{aligned}$$

Again, $J_{B,y} = J_{B,x}$ As last part the arm is investigated. Its length $L_A = 160\ [mm]$ and radius $R_A = 10\ [mm]$ were again taken from the technical drawing. With a mass of $m_A = 20\ [g]$ it is very light. In order to use the following values as moment of inertia for the arm in all its orientations relative to the body frame, different values need to be chosen for rotation about the e_x^b or e_y^b axis. $J_{A,z}$ is the moment of inertia about the major axis of the cylinder.

$$\begin{aligned} J_{A,x} &= m_A \left(\frac{R_A^2}{4} + \frac{L_A^3}{12} \right) \\ &= 4.317 \cdot 10^{-5} [kg\ m^2] \\ J_{A,z} &= m_A \left(\frac{R_A^2}{2} \right) \\ &= 1.0 \cdot 10^{-6}\ [kg\ m^2] \end{aligned}$$

As for the other bodies, $J_{A,y} = J_{A,x}$.

Now, Steiner's theorem will be used to move the parts where they belong relative to the body-frame coordinate system. Summed up, they give the overall moments of inertia of the quadrotor. First, the inertia about the e_z^b axis is taken into account. It consists of the inertia of ...

- the base about its z-axis with a vertical offset from the origin of the body-frame $D_{B,z} = 20\ [mm]$
- four times the arm rotating about its x-axis in distances $D_{A,z}$ and $D_{A,r}$ to the body-frame origin. According to the technical drawing, the radial offset $D_{A,r} = 150\ [mm]$ and the vertical offset $D_{A,z} = 25\ [mm]$.
- four times the motor rotating about its z-axis in distances $D_{M,z} = 15[mm]$ and $D_{M,r} = 260\ [mm]$.

- four times the propeller at a radial offset similar to that one of the motor. There is no vertical offset, since we assumed the overall COM, where the body-frame origin is placed to coincide with the rotor plane.

$$\begin{aligned} J_{zz} &= J_{B,z} + 4(J_{A,x} + m_A D_{A,r}^2) + 4(J_{M,z} + m_M D_{M,r}^2) + 4(J_{P,z} + m_P D_{M,r}^2) \\ &= 18.628 \cdot 10^{-3} \ [kg\ m^2] \end{aligned}$$

Similarly, the moments of inertia about the other axes are computed. Remember that, because of symmetry, $J_{yy} = 2J_{xx}$.

$$\begin{aligned} J_{xx} &= J_{B,x} + 2\left[J_{A,y} + m_A(D_{A,r}^2 + D_{A,z}^2)\right] + 2(J_{A,z} + m_A D_{A,z}^2) + \\ &\quad +2\left[J_{M,x} + m_M(D_{M,r}^2 + D_{M,z}^2)\right] + 2(J_{M,x} + m_M D_{M,z}^2) + \\ &\quad +2(J_{P,x} + m_P D_{M,r}^2) + 2J_{P,x} \\ &= 9.428 \cdot 10^{-3} \ [kg\ m^2] \end{aligned}$$

Notice, that for the simplified structure of the Microdrone, $J_{zz} \approx J_{xx}$. Thus, the use of the perpendicular axis theorem is a good approximation in (3.6).

4.2 Aerodynamics

The basic idea of rotors is that they accelerate a mass of air so that the resultant thrust is a reaction to that acceleration. This principle is utilized for actuation of quad-rotor helicopters and rotary-wing aircrafts in general. This section intends to show how thrusts and moments are created, so they can be used in the dynamical model of the quadrotor. Mainly, the theory of vertical flight will be considered.

The goal of this chapters is to describe, which forces and torques are created by a rotating propeller, and how. Here, especially two quantities are treated, namely thrust and drag torque. Fig. 4.1 shows how they act on a propeller.

This chapter is based on [BDB01, chapter 2], but also [Wat04], [Fay01] and [Bre08].

In section 4.2.1, actuator disk theory will deliver general relations between thrust, torque, pressure and velocities. Blade element theory in sec. 4.2.2 will be used to find formulas to calculate thrust and drag coefficient. Finally, considerations concerning forward flight are made.

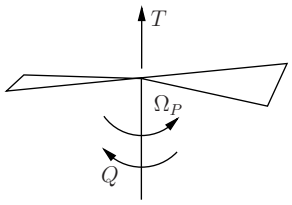

Figure 4.1: Thrust T and torque Q created by a propeller rotating with angular velocity Ω_P

4.2.1 Actuator Disk Theory

It is possible to conceive of an ideal hovering rotor as an actuator disk that accelerates air downwards and creates thrust as a reaction. This idealization avoids consideration of constructional details and makes the following assumptions:

- The thrust is uniformly distributed over the rotor, which is modeled as disk. At this rotor disk, a sudden jump of pressure Δp happens.
- No rotation or swirl is imparted to the flow.
- The slipstream of the rotor is a defined mass of moving air. Outside, the air is undisturbed.

Nevertheless, it is very useful to deliver principal relations that are useful for modeling.

Fig. 4.2 shows this actuator concept. V_c is the climb velocity, p_∞ is the static pressure (atmosphere pressure), p is the pressure right above the rotor. As the air approaches the rotor, it accelerates to $V_c + v_i$ at the rotor, where v_i is called the induced velocity. The airflow must be continuous. Hence also the velocity is continuous at the rotor, but there is a sudden change of pressure Δp, which is responsible for the rotor thrust

$$T = \Delta p A \tag{4.1}$$

where $A = \pi R_P^2$ is the rotor disk area. As the rotor of the Microdrone generates no thrust at its center (where the motor is placed), this definition of the rotor area is a further simplification. Bernoulli's equation for unsteady flow is

$$\boldsymbol{p} + \frac{1}{2}\rho_A \boldsymbol{q}^2 + \rho_A \frac{\partial \boldsymbol{\varphi}}{\partial t} = \boldsymbol{const.} \tag{4.2}$$

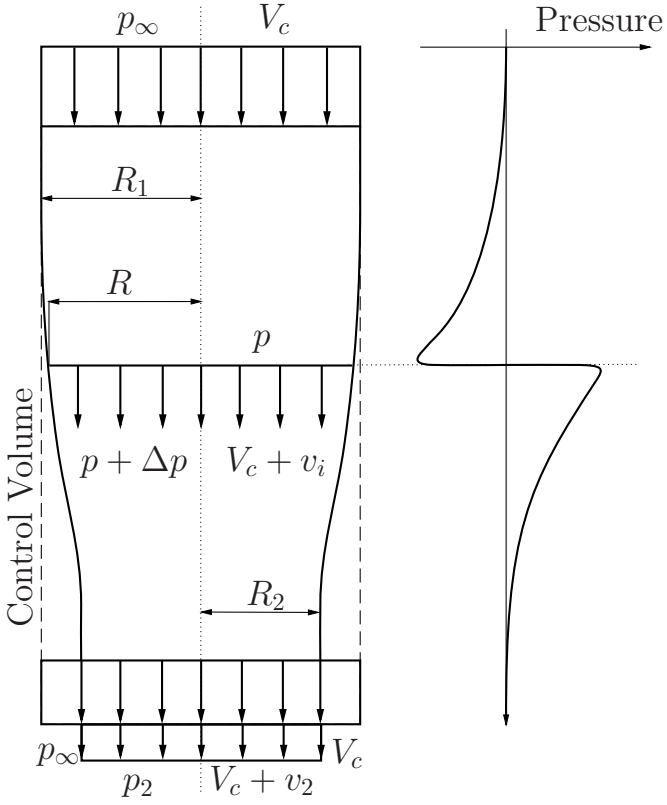

Figure 4.2: Actuator Disk Theory: Air stream passing through a tube (left) and pressure (right). The horizontal dotted line shows the position of the rotor.

where $\boldsymbol{p}$ stands for the pressure, $\boldsymbol{q}$ is the local airflow velocity, ρ_A is the air density and $\boldsymbol{\varphi}$ is the velocity potential[2] of the airflow. In a far distance from the wake, $\boldsymbol{q}$ and $\boldsymbol{\varphi}$ tend to zero, hence

$$\boldsymbol{p} + \frac{1}{2}\rho_A \boldsymbol{q}^2 + \rho_A \frac{\partial \boldsymbol{\varphi}}{\partial t} = \boldsymbol{p_\infty}$$

Since only the vertical component is important for the further investigations here, the vector notation is omitted from now on.

If we apply Bernoulli's equation to points above and below the rotor, respectively, then

[2]This potential describes an inherent order of a velocity field, namely that it is vorticity-free. So, $\nabla \times \boldsymbol{\varphi} = \mathbf{0}$ and $\boldsymbol{q} = \nabla \boldsymbol{\varphi}$. The usual symbol for that is $\boldsymbol{\phi}$, but in order to avoid conflicts with the roll angle, $\boldsymbol{\varphi}$ was chosen.

$$\begin{aligned} p_\infty + \frac{1}{2}\rho_A V_c^2 &= p + \frac{1}{2}\rho_A (V_c + v_i)^2 \\ p + \Delta p + \frac{1}{2}\rho_A (V_c + v_i)^2 &= p_\infty + \frac{1}{2}\rho_A (V_c + v_2)^2 \end{aligned}$$

If the first equation is subtracted from the second one and $p_2 = p_\infty$ is assumed, then

$$\Delta p = \rho_A (V_c + \frac{1}{2} v_2) v_2 \tag{4.3}$$

The velocity of the airflow leaving the control volume is higher than the undisturbed velocity V_c. Thus, the mass per unit time leaving the control volume is higher than that one entering. From that follows, that there must be a flux through the cylindrical sides of the control surface, which is

$$\bar{Q} = \pi R_2^2 v_2$$

The total mass entering and leaving the control surface are, respectively:

$$\rho_A \pi R_1^2 V_c + \rho_A \pi R_2^2 v_2$$
$$\rho_A \pi (R_1^2 - R_2^2) V_c + \rho_A \pi R_2^2 (V_c + v_2)$$

Thus, the rate of change of momentum in axial direction is

$$\begin{aligned} &\rho_A \pi (R_1^2 - R_2^2) V_c^2 + \rho_A \pi R_2^2 (V_c + v_2)^2 - \rho_A \pi R_1^2 V_c^2 + \rho_A \pi R_2^2 v_2 V_c \\ &= \rho_A \pi R_2^2 (V_c + v_2) v_2 \end{aligned}$$

According to Newtonian mechanics, this must be equal to the total force in axial direction, which consists of the rotor thrust plus the pressure forces at the ends of the cylinder:

$$T + \pi R_1^2 p_\infty - \pi (R1^2 - R_2^2) p_\infty - \pi R_2^2 p_2 = \rho_A \pi R_2^2 (V_c + v_2) v_2$$

Continuity of flow requires that the flow through the rotor plane and the control volume bottom are equal

$$\rho_A(V_c + v_i)A = \rho_A(V_c + v_2)\pi R_2^2$$

Using (4.1) and the previous two equations,

$$\Delta p = \rho_A(V_c + v_i)v_2 + (p_2 - p_\infty)(V_c + v_i)/(V_c + v_2)$$

assuming $p_2 = p_\infty$ and equating with (4.3) gives

$$v_i = \frac{1}{2}v_2$$

Using this equation, (4.3) and (4.1), the thrust is given by

$$T = 2\rho_A A(V_c + v_i)v_i \tag{4.4}$$

At hover, the climb velocity V_c is zero and also the thrust is known to be $T_H = \frac{mg}{4}$, as four rotors have to carry the weight of the whole robot. Hence, the induced velocity at hover can be calculated using

$$\begin{aligned} v_i &= \sqrt{\frac{mg}{8\ \rho_A\ A}} \\ &= 3.1109[ms^{-1}] \end{aligned}$$

The overall mass of the quadrotor is $0.964[kg]$, the propeller radius is $R_P = 180[mm]$, and the (dry) air density $\rho_A = 1.2[kg\ m^{-3}]$, which is approximately the value at $20[°C]$ and $101.325[kPa]$[3].

4.2.2 Blade Element Theory

Actuator disk theory delivers relations between induced velocity and thrust. For this work, the relation between rotational velocity Ω and thrust and torque are of special interest. Blade element theory delivers such relations.

Fig. 4.3 shows a blade element and the associated forces. The blade element shown is the profile of width dr of a blade at a radius r from the rotation axis. θ_P is the geometric

[3] According to International Standard Atmosphere (ISA)

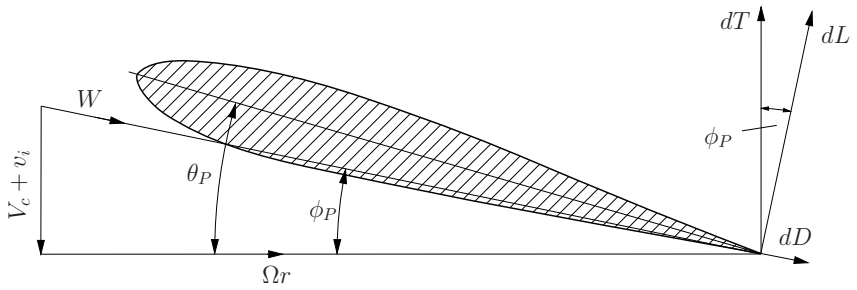

Figure 4.3: Force components on blade element

pitch angle relative to the rotation plane. The climbing velocity is denoted V_c again, and the local induced velocity is v_i. Ω_P is the rotational velocity of the rotor. The direction of the flow relative to the rotational plane spans an angle ϕ_P, which is usually called inflow angle. Assuming small ϕ_P,

$$\phi_P \approx tan(\phi_P) = \frac{V_c + v_i}{\Omega_P r}$$

The lift force of the blade element is

$$\begin{aligned} dL &= \frac{1}{2}\rho_A W^2 C_L c \; dr \\ &\approx \frac{1}{2}\rho_A \Omega_P^2 r^2 C_L c \; dr \end{aligned}$$

where $W^2 \approx \Omega_P^2 r^2$ due to small ϕ_P. C_L is the lift coefficient and c the blade chord. The lift coefficient is given by

$$C_L = a\alpha_P = a(\theta_P - \phi_P)$$

For the lift slope a [BDB01] suggests a value of 5.7. Because of the small angle assumption concerning ϕ_P, it follows that $dT \approx dL$ and the total thrust can be calculated by integrating the thrust created by the blade elements over the whole radius and multiplication with the number of blades b:

$$T = \frac{1}{2}\rho_A \; a \; b \; \Omega_P^2 \int_0^{R_P} r^2(\theta_P - \phi_P) \; c \; dr \tag{4.5}$$

Defining $\lambda_c = V_c/\Omega_P r$, $\lambda_i = v_i/\Omega_P r$ and $x = r/R_P$ and further assuming constant

chord, induced velocity and pitch along the whole radius, then (4.5) can be written as

$$T = \frac{1}{2}\rho_A \; a \; b \; \Omega_P^2 \; R_P^2 \left[\frac{1}{3}\theta_P - \frac{1}{2}(\lambda_c + \lambda_i)\right]$$

The thrust coefficient

$$t_c = \frac{T}{\rho_A \; s \; A \; \Omega_P^2 \; R_P^2} \tag{4.6}$$

where $s = bc/\pi R_P$ is called rotor solidity is thus given by

$$\begin{aligned} t_c &= \frac{a}{4}\left[\frac{2\theta_P}{3} - (\lambda_c + \lambda_i)\right] \\ &= \frac{a}{4}\left[\frac{2\theta_P}{3} - \frac{(V_c + v_i)}{\Omega_P R_P}\right] \end{aligned}$$

Observe in the second equation that the thrust coefficient decreases with increasing climb speed and induced velocity.

At hover, the thrust coefficient is simply

$$t_c = \frac{a}{4}\left[\frac{2\theta_P}{3} - \frac{v_i}{\Omega_H R_P}\right] \tag{4.7}$$

since the climb velocity is zero. Ω_H stands for the angular rate of the propeller at hover here. According to [BDB01], the above formulas are sufficiently accurate even if there is linear twist and constant chord, which is approximately the case for the propellers of the Microdrone.

At hovering, Ω_H is known from experiments[4] as shown in tbl. 4.1.

Experiment	$\Omega_H[s^{-1}]$
2	166.90
3	169.08
4	170.81

Table 4.1: Propeller rotational velocity Ω_H at hover, Experimental results

Taking the mean, Ω_H evaluates to $\Omega_H = 168.93[s^{-1}]$. Propeller chord $c = 35[mm]$, and blade pitch $\theta_P = 20[^\circ]$ were extracted as mean values from the technical drawing. The induced velocity at hover is known from the previous chapter. Hence we are able to

[4]See appendix C

compute the thrust coefficient using (4.7):

$$t_c = 0.18582$$

Equation (4.6) delivers the following result

$$t_c = 0.16912$$

The difference between those two is assumed to originate from the inaccuracies in measurements and simplifications made in the derivation of the formulas used, especially the measurements of θ_P, c and the assumptions of a whole disk as rotor area. The effective area of the rotor of the Microdrone is more like a circular ring. Furthermore, the blade chord of a Microdrone rotor-blade is far from constant.

In order to find the thrust factor as used in the equations of motion of the drone, the mean of these two values is used:

$$\begin{aligned} t_F &= \frac{T}{\Omega_P^2} \\ &= t_c \rho_A \; s \; A \; R_P^2 \\ &= 8.694 \cdot 10^{-5} [kg \; m] \end{aligned}$$

In a similar manner, the relations concerning the torque can be determined. Fig. 4.3 shows that the torque dQ of a blade blade element is

$$\begin{aligned} dQ &= r(dD + \phi_P dL) \\ &= \frac{1}{2}\rho_A \; \Omega^2 \; r^3 \; c(\delta + \phi_P \; \frac{t_c}{6}) \; dr \end{aligned} \tag{4.8}$$

where δ is the local blade section drag coefficient. δ is assumed to be constant and [BDB01] suggests a value of $\delta = 0.012$. Eq. (4.8) can be integrated to find the rotor torque

$$Q = \frac{1}{2}\rho_A bc\Omega_P^2 R_P^4 \left(\frac{\delta}{4} + \int_0^{R_P} r^3 \phi_P \frac{t_c}{6} \; dr \right) \tag{4.9}$$

The torque coefficient is defined by

$$q_c = \frac{Q}{\rho_A \; s \; A \; \Omega_P^2 \; R_P^3} \tag{4.10}$$

If constant induced velocity is assumed, then $\phi_P = (\lambda_c + \lambda_i)R_P/r$ and (4.9) gives

$$q_c = \frac{\delta}{8} + \frac{\lambda_c + \lambda_i}{t_c}$$

which can be evaluated at hover

$$\begin{aligned} q_c &= \frac{\delta}{8} + \frac{\lambda_i}{t_c} \\ &= 0.578 \end{aligned}$$

Again, for the equations of motion a torque factor is needed, that is a relation in the form

$$\begin{aligned} q_F &= \frac{Q}{\Omega_P^2} && (4.11) \\ &= q_c \rho_A s A R_P^3 && (4.12) \\ &= 5.097 \cdot 10^{-5} [kg \; m^2] && (4.13) \end{aligned}$$

4.2.3 Considerations Concerning Forward Flight

In forward flight, the thrust gets obviously dependent on forward speed. Moreover, the angle of incidence of the rotor plane about an axis perpendicular to the rotor axis and the forward speed vector influences the induced velocity and thus also the thrust. An additional drag force, acting in rear direction, occurs which is called H-force. Also the equations for the torque are affected. Although the structure of the equations for thrust and torque coefficient principally remains the same, a further treatment here would go too much into detail. According to many of the publications mentioned in the chapter about dynamical modeling, the relations in vertical flight deliver sufficiently accurate results for control design. However, for simulation purposes the model should principally be as accurate as possible. For highly accurate simulation, here is space for further improvements. [BDB01, chapter 3] delivers the theory and [Fay01] its application in the mesicopter-project.

4.3 Experimental Parameter Identification

In this work, the experimental parameter identification follows two goals: The first one is the obvious one - namely to find parameters that were not calculated or could not be calculated. The Microdrone is a commercial quadrotor helicopter and by far not all data of the real device were known. For example the parameters of the motor were not considered at all in the previous sections. Also the coefficients concerning drag were not topic until now. In this chapter, system identification methods will be used to get estimates for these values. The second goal is model verification. When all parameters of the model are fixed, it is necessary to show, that these and the model itself are valid generally and not just for the experiment which was used for parameter identification. However, this won't be possible for all the maneuvers that could be flown with a quadrotor helicopter. But at least for the range, where the linearization done in chapter 3.5 is a good approximation.

LabVIEW VIs and the LabVIEW System Identification Toolbox were used for identification[Nat04]. The sample time of all measurements was $100[Hz]$.

4.3.1 Brushless DC-Motor Model Parameters

The Microdrone is equipped with four brushless outrunner DC-motors that are directly connected to the rotor (no gearbox). According to Microdrones GmbH., the angular velocity stays below a maximum value of $2000[rpm]$. The only known signals related to the motor are a signal named PWM that is interpreted as input here, and an angular velocity signal. Actually, the signal named PWM is no PWM-signal but a discrete signal between 0 and 255. The angular velocity signal is in the same range, where 0 stands for $0[rpm]$ and 255 for $4250[rpm]$. A first order transfer function from the PWM-signal to angular velocity was sufficient to model the dynamics between these two signals:

$$G_M(s) = \frac{1.08}{1 + 0.165s} \tag{4.14}$$

Fig. 4.4 shows the input signal (PWM, black, [1]), the output signal (propeller angular velocity, red, $[s^{-1}]$) and the simulated response of the identified transfer function model (4.14). The data from experiment 4b was used here.

Fig. 4.5 shows the same signals as the previous figure, but at a completely different experiment and a different engine. According to the good match, the model is assumed to be valid.

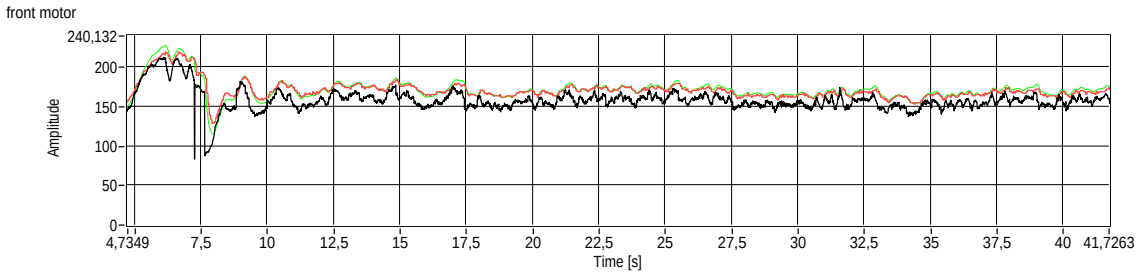

Figure 4.4: input (PWM, black, [1]), output (propeller angular velocity, red, $[s^{-1}]$) and simulated output (propeller angular velocity, green, $[s^{-1}]$) of the front motor

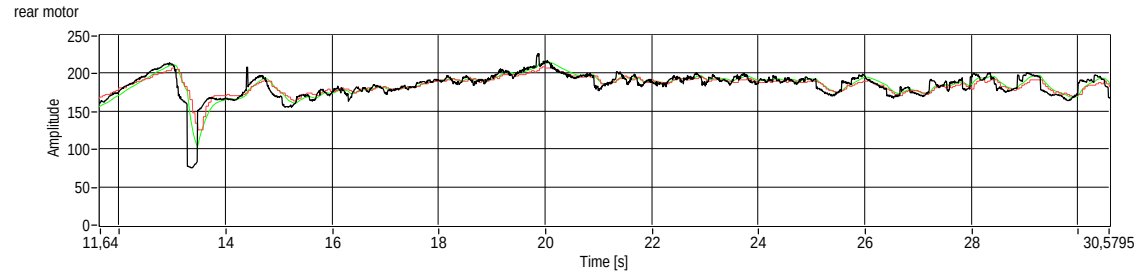

Figure 4.5: input (PWM, black, [1]), output (propeller angular velocity, red, $[s^{-1}]$) and simulated output (propeller angular velocity, green, $[s^{-1}]$) of the rear motor

4.3.2 Quad-Rotor Helicopter Model Parameters

The identification of the parameters of the quadrotor helicopter model was also done using the LabVIEW System Identification Toolbox. Several VIs were used for preprocessing, identification and plotting. The principal procedure was the same for all of the identified transfer functions: First, the input signals were computed according to (3.19). The measurements from the IMU are subject to errors (see sec. 2.2.2.1), which are corrected in the second step[5]. Then, the input and output (stimulus and response) signals were cut. This was necessary to obtain a subset of the signals, where the linearization from chap. 3.5 is valid. Afterwards, these signals were filtered using a phase-lag free 6-th order lowpass filter with its cutoff frequency at $40[Hz]$. After these preprocessing tasks, the ultimate system identification was executed.

[5]However, no correction of earth rotation was done for the gyros. Its influence on the identification result was considered to low.

Throttle Now, the transfer function from the throttle input $\bar{u}_T$ to the z-accelerometer that measures $\dot{v}_z$ is of interest. According to (3.23), this transfer function is

$$G_T = \frac{t_F s}{s + b_z}$$

Fig. 4.6 shows a comparison of the measured and two simulated signals of $\dot{v}_z$. The black signal is the measurement and the red signal is a simulation of the transfer function $G_{T,i}$ as it was found by the identification algorithm, excited with the same stimulus[6]. In green, a simulation of the transfer function $G_{T,c}$ can be seen. For t_F, the value calculated in the chapter on aerodynamics was used and b_z was set to zero.

$$\begin{aligned} G_{T,c} &= \frac{8.694 \cdot 10^{-4} s}{s} \\ G_{T,i} &= \frac{-1.59837 \cdot 10^{-4} + 8.34167 \cdot 10^{-5} s}{2.77634 + s} \end{aligned}$$

In case of $G_{T,c}$, the dynamics reduces to a memoryless gain between input and output. Notice the disturbance between sample 1050 and 1200: Here, Experiment 3 was used and this is exactly the time when the drone does the angular movement in yaw-direction. This influence on the z-dynamics is not contained in the model.

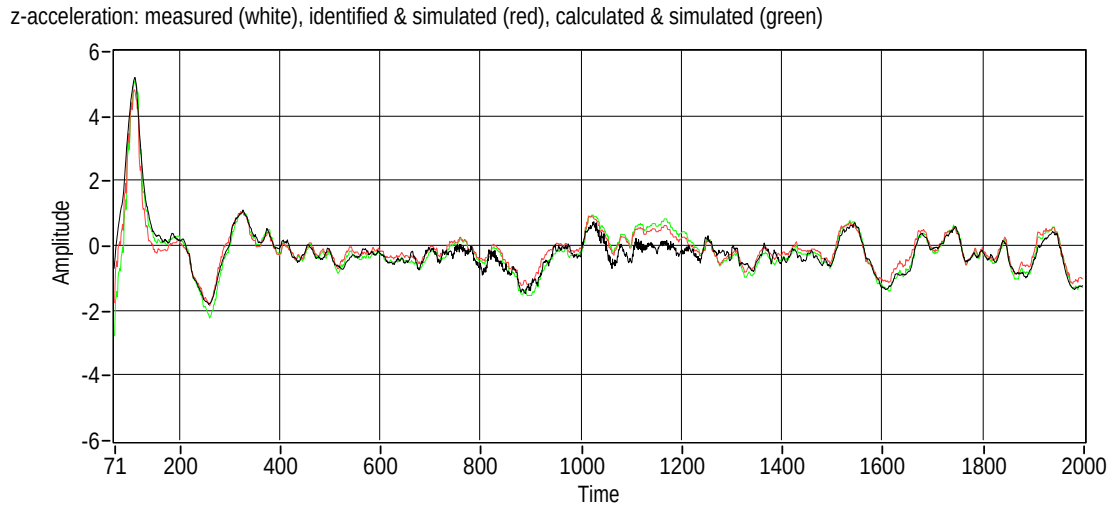

Figure 4.6: Throttle identification (accelerations in $[m/s^2]$, time in samples [1])

[6]The fact, that the system models were excited with the same stimulus as the real system is omitted from now on.

Roll According to the transfer matrix **G** (3.23), the transfer function from the roll input $\bar{u}_R$ to the acceleration in y-direction $\dot{v}_y$ is given by

$$G_R = \frac{gt_F l}{(sJ_{xx} + b_\omega)(s + b_y)}$$

Since $J_{xx}, b_y, b_\omega > 0$, the final value theorem of the Laplace transformation is applicable and delivers

$$g_\infty = \frac{gt_F l}{b_y b_\omega} \tag{4.15}$$

for the static gain. The transfer function determined by the identification algorithm is

$$G_{R,i} = \frac{7.62631 \cdot 10^{-5}}{1 + 0.24998s + 1.11583 \cdot 10^{-5}s^2}$$

with a static gain of

$$g_\infty = 7.62631 \cdot 10^{-5}$$

using calculated values and assuming $b_y = b_z$ the value of b_ω can be computed using (4.15)

$$b_\omega = 1.0473[kg/s]$$

Then, a transfer function involving the calculated values is given by

$$G_{R,c} = \frac{9.81 \cdot 8.694 \cdot 10^{-5} \cdot 0.26}{(9.428 \cdot 10^{-3}s + 1.0473)(s + 2.77634)}$$

Fig. 4.7 shows a comparison of the measured and two simulated signals of $\dot{v}_y$. The black signal is the measurement and the red signal is a simulation of the transfer function $G_{R,i}$ as it was found by the identification algorithm. In green, you see a simulation of the transfer function $G_{R,c}$. The parameters of this transfer function were partly determined through this identification process using data from experiment 4a. Hence, for model verification this transfer function must also be compared with real-world data from a different experiment. Fig. 4.8 shows this verification, using data from experiment 4.

Pitch The transfer function from the pitch input $\bar{u}_P$ to accelerations in x-direction $\dot{v}_z$ is

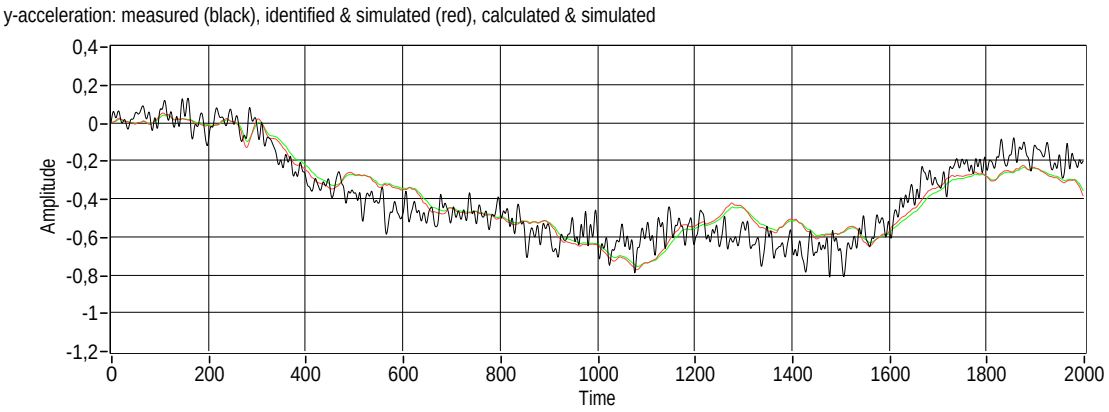

Figure 4.7: Roll identification (accelerations in $[m/s^2]$, time in samples [1])

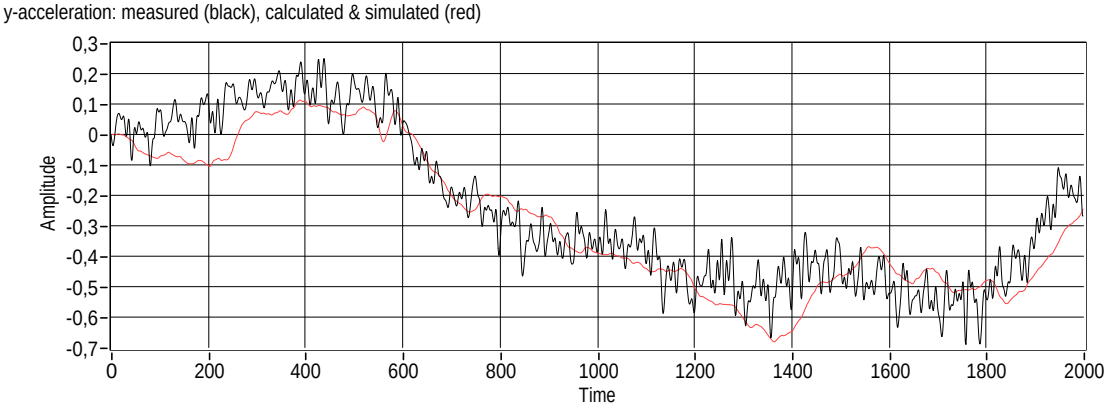

Figure 4.8: Roll verification (accelerations in $[m/s^2]$, time in samples [1])

$$G_P = -\frac{gt_F l}{(sJ_{yy} + b_\omega)(s + b_x)}$$

Assuming $J_{yy} = J_{xx}$ and $b_x = b_y$, then this transfer function is, except for the sign, equal to G_R. Hence, the negative of $G_{R,c}$ can be used for verification

$$G_{P,c} = -G_{R,c}$$

Although not necessary, identification was executed and delivered the transfer function

$$G_{P,i} = \frac{-6.23695 \cdot 10^{-5}}{1 + 0.111082s + 6.67218 \cdot 10^{-5}s^2}$$

A comparison of $\dot{v}_z$ measured and the response of the two transfer functions $G_{P,c}$ and $G_{P,i}$ is shown in fig. 4.9

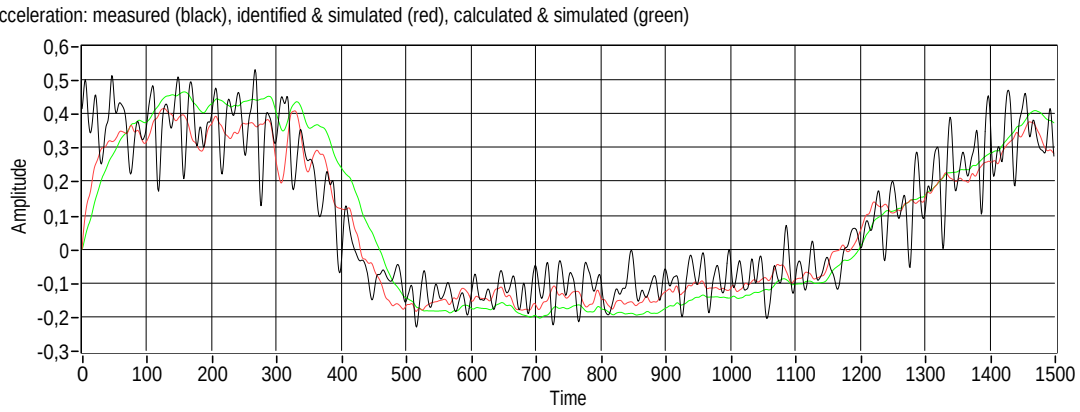

Figure 4.9: Pitch verification (accelerations in $[m/s^2]$, time in samples [1])

Yaw The transfer function from the yaw-input $\bar{u}_Y$ to angular velocity around the body-frame y axis ω_z is

$$G_Y = \frac{q_F}{J_{zz}s + b_{\omega,z}}$$

Now, the same procedure as in the roll-case is used: Since $J_{zz}, b_{\omega,z} > 0$, the final value theorem of the Laplace transformation is again applicable and delivers

$$g_\infty = \frac{q_F}{b_{\omega,z}}$$

for the static gain. The transfer function determined by the identification algorithm is

$$G_{Y,i} = \frac{5.4143 \cdot 10^{-5}}{1 + 0.774194s}$$

with a static gain of

$$g_\infty = 5.4143 \cdot 10^{-5}$$

Because the drag factor q_F is known, the friction coefficient $b_{\omega,z}$ can be calculated

$$b_{\omega,z} = 0.94135[kg/s]$$

Using this value, it is possible to form a transfer function involving calculated values for J_{zz} and q_F:

$$G_{Y,c} = \frac{5.0967 \cdot 10^{-5}}{0.018628s + 0.94135}$$

Fig. 4.10 shows a comparison of the computed transfer functions with measurements.

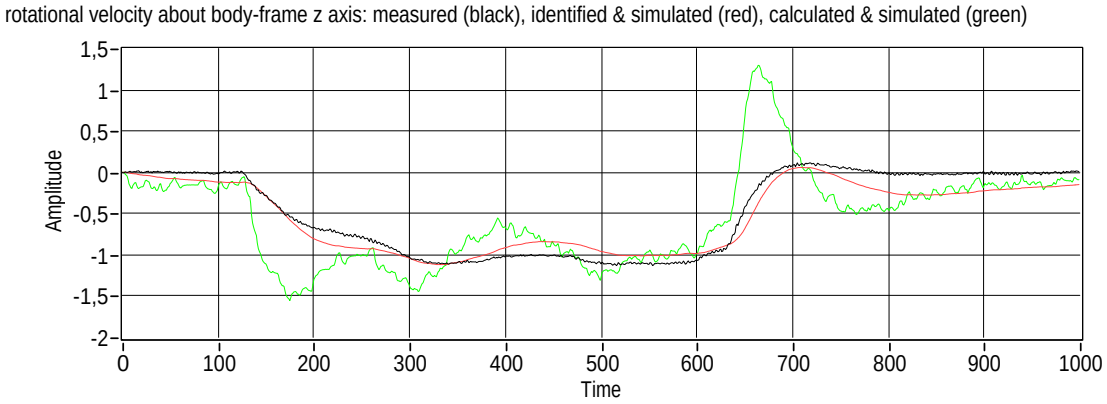

Figure 4.10: Yaw identification (angular velocities in $[s^{-1}]$, time in samples [1])

The transfer function containing the calculated values shows no good accordance to the measured signal. In the parameters of $G_{Y,i}$ and $G_{Y,c}$, the biggest difference is noticed in the gain of the denominator polynomials. Physically, that is corresponding to a higher moment of inertia J_{zz}. In order to choose the proper value for the model parameter, a second identification was carried out. The results are shown in fig. 4.11.

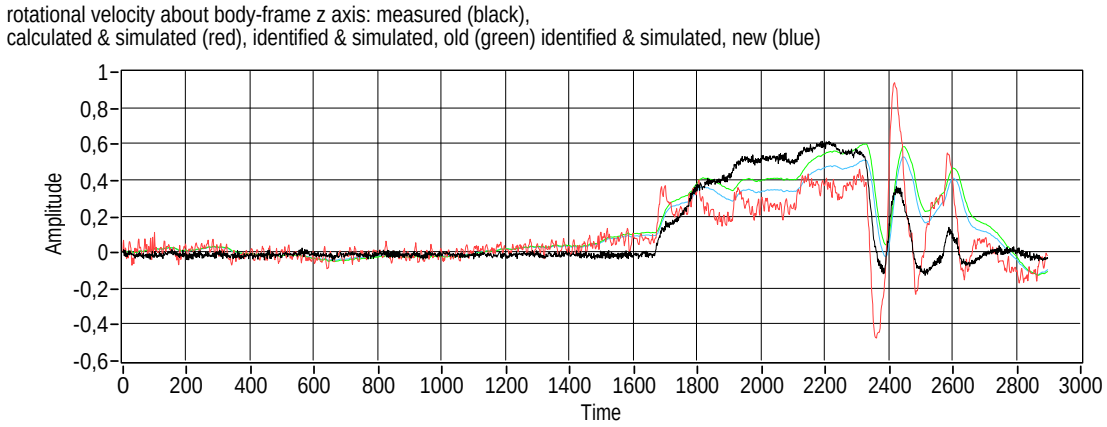

Figure 4.11: Yaw verification (angular velocities in $[s^{-1}]$, time in samples [1])

The response of the transfer function $G_{Y,i}$ from the first identification process (green) is quite similar to the response of the new identification result (blue). Again, the calculated transfer function (red) shows low performance. In the end, the parameters from the first identification were taken. It turned out, that the results presented in 4.11 are something like a worst case and the overall accordance using the new parameter J_{zz} is good. To prove that, another example is given in fig. 4.12.

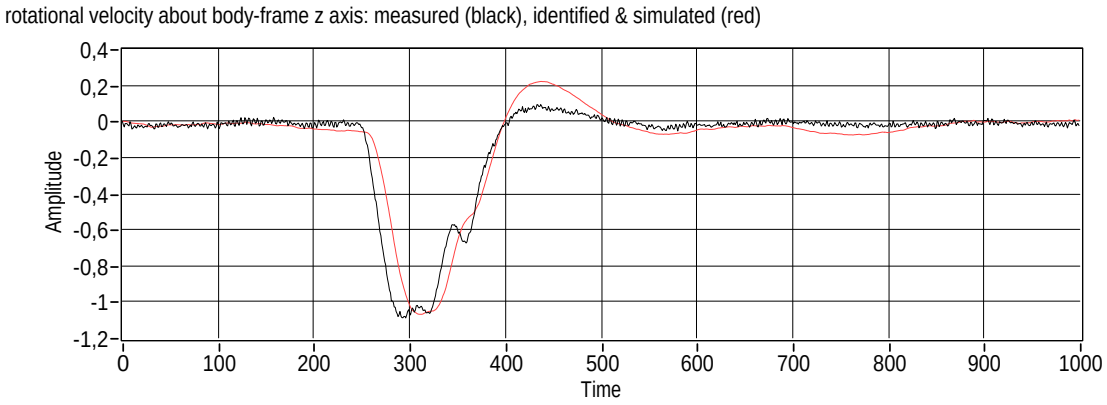

Figure 4.12: Yaw verification 2 (angular velocities in $[s^{-1}]$, time in samples [1])

Assuming that the identified value of $b_{\omega,z}$ is right, the new value for J_{zz} is

$$J_{zz} = 0.72879[kg\ m^2]$$

which makes no sense if it is interpreted as moment of inertia[7]. But it might be a result of another physical effect, which is not included in the model.

Finally, the parameters of the linearized model are summarized in tbl. 4.2:

[7]The value is 39.122-times higher than that one computed in chap. 4.1!

Parameter	**Value [Unit]**	**Method**	**Comment**
g	9.81 $[m/s^2]$	literature	
l	$2.6 \cdot 10^{-3}$ $[m]$	technical drawing	$= D_{M,r}$
t_F	$8.694 \cdot 10^{-5}$ [1]	calculated	
q_F	$5.097 \cdot 10^{-5}$ [1]	calculated	
J_{xx}	$9.428 \cdot 10^{-3}$ $[kg\ m^2]$	calculated	$= J_{yy}$
J_{zz}	$7.288 \cdot 10^{-1}$ $[kg\ m^2]$	experiments	
b_x	2.776 $[kg/s]$	experiments	$= b_y = b_z$
b_ω	1.0473 $[kg/s]$	experiments	
$b_{\omega,z}$	0.94135 $[kg/s]$	experiments	

Table 4.2: Parameters of the linearized model

Chapter 5

Simulation

5.1 Introduction

Simulation in the context of natural sciences means to virtually execute an experiment, so that the output approximates a real-world result. At simulation, the experimental set-up is replaced by a set of mathematical descriptions (models). The development of such models is topic in section 3. As result, the process of modeling typically delivers a set of differential equations. Simulation now means to solve these equations for special input functions and initial values. In the case of virtual environment simulators, also the environment must be modeled, and the equations describing a behavior must be solved. Typically, the geometrics of the environment are defined in a world model. Especially in game based simulators these are also often called maps. Everything in such a map is subjected to virtual physical relations, where differential equations describing the behavior of objects in such a virtual world are generated and solved by a physics engine. Altogether, such a real-world like virtual environment simulator is a very complicated framework. This chapter is dedicated to describe how that works.

5.1.1 Demands on Multiple UAV Simulation

The simulation of multiple UAVs is a challenging task. On the one hand, multiple dynamical models need to be simulated concurrently in order to detect collisions and simulate their dynamical effects. Moreover, a lot of sensors - including their models - need to be simulated in a realistic manner. Furthermore, the possibility to interact (e.g. communicate) with a world model or among the UAVs are necessities for collaborative mission simulation. If all these problems are solved, and a simulation is executed immediately

another problem arises: If all the results are shown as single diagrams over time, it is very difficult to understand what has happened. Thus, a 3D representation of the results becomes important.

As simulation should provide *easier* use of experiments, it has no sense if it takes more time and effort than the real-world experiment. So setup time and execution duration should be as low as possible. Also a real-time presentation of the results is preferable in the context of demonstration.

According to the cDrones project, the simulator should amongst others serve as environment for experimentation with flight formation controllers. Thus, an easy way of code portation from the simulator to the drone should be possible.

To summarize the demands on multiple UAV simulation, especially in the cDrones project, they are listed here:

- Sufficiently accurate and stable dynamics simulation, including simulation of rigid-body motion, collisions and sensing.
- The possibility to simulate communication.
- The possibility to add and modify models of sensors, robots and the world.
- 3D representation of the simulation results, favorably in real-time.
- Low execution time.
- Easy transfer of code used in a simulated controller to the real one.

5.1.2 Simulator selection

The above mentioned demands show the necessity of a so called *high-fidelity* simulator. According to [BFL08], that is a simulator that is able to physically correctly simulate the robot, its sensors, and all other objects in the simulation world. Most of the modern high-fidelity robot simulators use a physics engine to compute the motion dynamics of the simulated bodies. Famous representatives are Gazebo, USARSim, Webots and Microsoft Robotics Studio. Gazebo and Webots use the Open Dynamics Engine (ODE) for physics simulation. USARSim and Microsoft Robotics Studio use engines that are also used in computer games. USARSim bases on Unreal Tournament from EPIC Games with KARMA as its physics engine. Microsoft Robotics Studio comes with the PhysX engine from NVIDIA. All of them are based on rigid body models and support collision

detection. Gazebo is the only simulator that is completely open-source. It is used for example in RoboCup and in education, mostly in combination with Player (A server that provides control and measurement information about the robot to the network). Webots is a commercial product but uses the ODE. In USARSim, the physics engine and the Unreal Tournament part of the simulator are commercial, but the part that defines robots, sensors, worlds and so forth is open source. This is an interesting advantage: For the graphical design of models, Unreal provides an easy to use editor. The definition of physical models and their parameters is done using a high-level script language called Unreal Script. Microsoft Robotics Studio is entirely closed source, supports a lot of programming languages and provides a whole framework for developing applications for robots. All these 4 simulators are promising projects for the future and of course actively maintained.

As the robot model in this thesis is a quadrocopter UAV, it is obvious to think about using flight simulators. A flight simulator has one major goal: To simulate the aerodynamics and dynamics of an aircraft as accurate as possible. Thus, very high-developed models exist and higher level simulation algorithms are used. For example, JSBSim is an open-source flight dynamics model and simulation library, that is used e.g. in FlightGear and OpenEaagles. As default, it uses a 3rd order Adams Bashforth algorithm for integration. USARSim on the contrary uses an implicit Euler method. Mathengine, the developers of the physics engine included in Unreal, argue this selection with the stability of this method. However, in the Microdrones used in the cDrones project, the controller for the UAVs already exists and is thus not the major research topic. Thus, highly accurate dynamics simulation is less important. However, the possibility to do accurate simulation is preserved by a special model, which will be presented later.

As a result of the above considerations, USARSim was chosen as simulator in this thesis. It fulfills all requirements and already provides a quad-rotor helicopter UAV model as well as models for the important sensors. In conjunction with some of the Unreal mod-authoring tools it provides a framework for user-friendly editing of world-, robot- and sensor models. It's physics engine (KARMA) is said to provide high overall stability, sufficient accuracy and real-time execution. Another reason for choosing USARSim is it's presence in up-to-date science publications and it's use in RoboCup.

5.2 Unified System for Automation and Robot Simulation

From the abbreviation USARSim one might first think of urban search and rescue (USAR). In fact, that is what it was originally developed for by Jijun Wang at University of Pitts-

burgh: A high-fidelity robot and environment simulator for research in human- robot interaction (HRI) and multi-robot coordination in USAR scenarios [WLHK05]. However, USARSim is actually the short form of "Unified System for Automation and Robot Simulation", since it meanwhile supports a lot of environments, robots and sensors. Example applications include the DARPA urban challenge, robotic soccer, humanoids and many more. Most of this chapter is based on [WB].

5.2.1 System Architecture

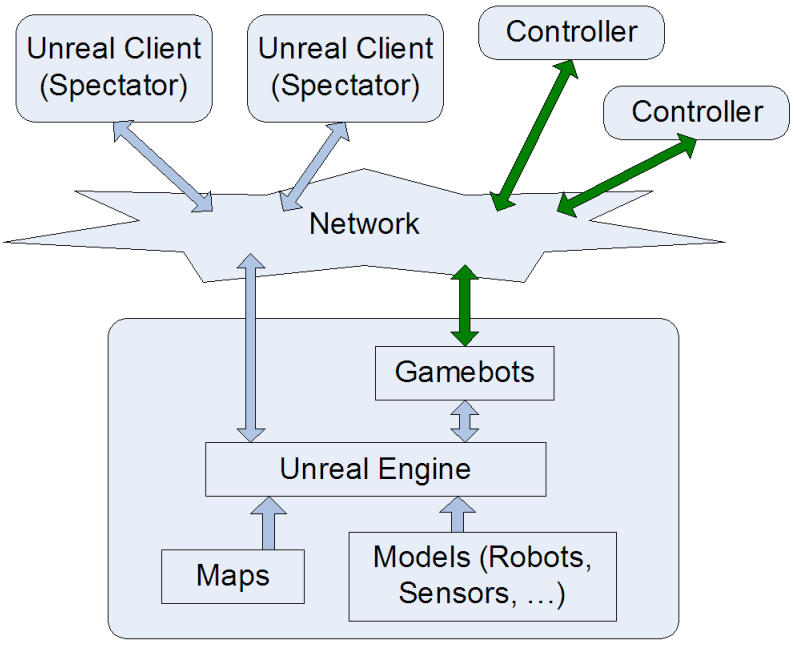

Figure 5.1: USARSim System Architecture

The system architecture of USARSim is shown in fig. 5.1. The light gray arrows mark the propagation of Unreal Data, while the dark green illustrate control data. USARSim consists of 3 main parts: The simulation server, which is the lower big box with round corners. It executes simulations and provides network connections for clients. The second part is called Gamebots, which translates the proprietary Unreal Data protocol so external clients (such as controllers in the USARSim case) can communicate with the Unreal server. The Unreal server, Gamebots and USARSim run as one application on one machine. USARSim and Gamebots could be understood as a special game running on the Unreal server, where Unreal clients are spectators and controllers are gamers.

The third main part is the client side, which for itself consists of two types of clients.

One type is the Unreal client, that acts as nothing more than a spectator. It is able to move through the 3D-environment of the map currently loaded on the server and watch what happens there, e.g. UAVs flying. That client cannot influence the world at all, and could be run separately on any machine that is connected to the server (via TCP/IP). The second type of client is the controller, which is able to communicate with the server via Gamebots over TCP/IP. Each robot in the world needs its controller. Obviously, the number of Unreal clients can be different than the number of robots. At the controller side, the user is completely independent. The only thing that is predefined by USARSim is the communication protocol.

In addition to these components that are relevant for experimentation, there exist others that are very helpful at designing own maps and developing own robots or sensors. One of those is UnrealEd, a map editor that comes with the Unreal Computer game. By cutting geometric shapes out of the world, that is preliminarily completely filled with mass, one can build his own environment. By application of textures, surfaces can get a realistic look, while static meshes can be used to illustrate complex shapes. Another very helpful tool is WOTgreal, which is an integrated development environment for Unreal script. It features Unreal class-tree view, code completion, step-wise debugging and much more.

There exist a lot of extensions at the client side as well, e.g. MOAST or Player, but they were not used in this project.

5.2.2 Controller Interface

The server opens a TCP listener port[1] at startup that is configurable in `BotAPI.ini`. At maximum 16 clients can connect to the server per default, but this value is also changeable via `BotAPI.ini`. Communication via this interface is coded in ASCII conforming the Gamebots protocol. All the messages that are sent from the server and all commands that can be sent from a controller to the server are listed and described in the USARSim manual [WB].

5.2.3 Physics

As already introduced, parts of the physical modeling and simulation are done by a so called physics engine. In the case of USARSim and the Unreal computer game, the physics engine is called Karma and was developed by MathEngine [Mat02]. Publications about USARSim physics validation are found in [PBS07] and [Wan].

[1] default: 3000

Within this engine, a rigid body has three representations:

- The dynamics object.
- A collision object, which defines the geometric dimensions of the body. The volume of the collision object is owned by this body only, no others are allowed to enter it.
- The render object. That is the graphical representation of the body in the presentation of the simulation. However, a physics simulation can evolve without any graphical display.

If you consider the simulation of a ball dropping to the floor, this is done by KARMA using the following steps: First of all consider an initial state, defined by the engineer like that: Assume that there is a world model defined with a horizontal floor as collision object. The only field, that is defined within this world is the gravitational one[2]. The ball is placed somewhere above the floor, so that gravitational acceleration points downwards in the direction of the floor. Furthermore, it is in equilibrium and its dynamics object is well defined, containing definitions of drag, mass, etc. The radius of the spherical collision object is smaller than the height above the floor. There are no constraints (joints or contacts) acting on the ball. Such constraints could be of two types:

- Equality constraints are particular restrictions on one or more bodies. For example, if you consider the ball mounted at one end of the rod of a pendulum, the length of the rod maintains a constant distance between the joint and the ball.
- Inequality constraints: Consider the connection between the ball and the pendulum joint as (non-rigid[3]) cord. The distance between joint and ball must be smaller or equal to the length of the cord.

Now, as the initial state is defined, the engine proceeds with the following steps (also called "The KARMA Pipeline" [Mat02, chapter , pp 8]), which are repeated for the whole simulation time. This KARMA-Pipeline is shown in fig. 5.2

1. Farfield Collision Detection: Karma checks, whether pairs of objects are nearby.
2. Nearfield Tests: For the objects that are nearby another, it is determined whether their collision object representation overlaps. If it does, the near field test chooses a set of contact points that represents the intersection for contact force calculation.

[2]Any other field that can be modeled in KARMA could be included into simulation, too
[3]so the cord itself cannot be modeled in KARMA

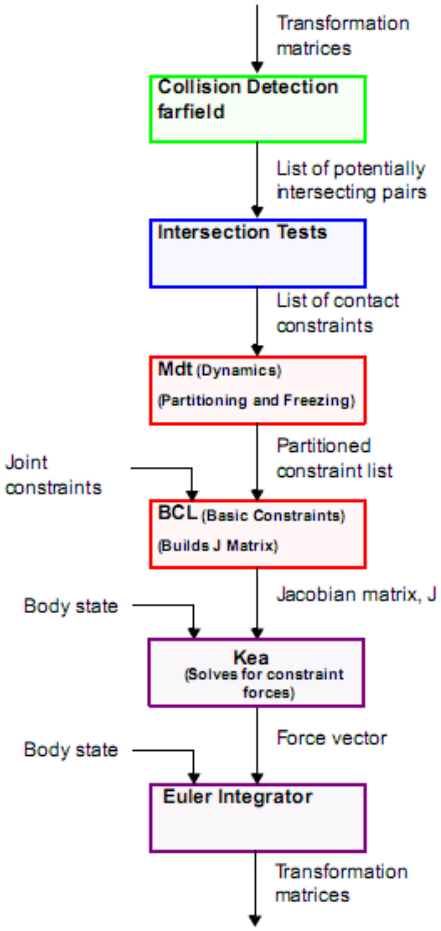

Figure 5.2: KARMA pipeline

3. Partitioning and Freezing: In partitioning, Karma builds groups of objects that interact via constraints. Then for each of these groups it checks, whether it is in perceptible motion or not. If it is not, the dynamics of the bodies in the group is disabled and their associated collision models are frozen.

4. Basic Constraints Library (BCL): Converts the high-level representation of e.g. hinge axes and joint positions to a mathematical representation.

5. Kea: Karma uses a Lagrange multiplier method to model constrained dynamics. As the formalism of Lagrange was used in the chapter about modeling, it is very interesting to notice that a very closely related method occurs here. The use case of both methods are the same, namely modeling. But here, the constraints might

change over time and the model is required to adapt so these constraints are met. In the formalism of Lagrange (where the Lagrange equations of the second kind are used), constraints are indirectly contained in the transformation to generalized coordinates. The Lagrange multiplier method uses the Lagrange equations of the first kind, where constraints are treated explicitly as extra equations [Hof04]. In order to maintain these constraints, this method calculates constraint forces. When these forces are applied, the constraints will be met at the end of a time-step. The problem at calculating these forces is also referred to as linear complementarity problem (LCP). The subprogram of Karma that solves such LCPs is called Kea.

6. Euler integrator: Since the constraint forces calculated in the previous step satisfy the constraints at the end of the time-step, Karma can use an implicit Euler method to integrate these forces, what makes it very stable. External forces, such as gravity ore those set by a USARSim controller, are explicitly integrated.

After these steps, graphics are rendered and user interaction is engaged. Then, the cycle restarts.

Robot models As described in the Karma pipeline, the equations of motion for a body moving through a virtual world is calculated automatically. Thus, a robot model for Karma consists of

- A geometric description of the collision objects of the parts of the robot
- Dynamic parameters of each part (mass, drag coefficients, moments of inertia tensor, ...)
- Constraints that model joints

Now it is probably clearer, why the parameters and its physical meaning were that important for this work. If the representation of the of the Microdrone was only a set of identified transfer functions, there was no (meaningful) ability of implementing such a model in USARSim.

Stability Considerations and Simulation Time-Scale The version of Karma used in USARSim V3.1 does all these calculations in one thread[4]. Since we consider real-time execution, the time-step Δt is lower bounded by the time needed to execute one pipeline

[4]Newer versions are able to do that multi-threaded

iteration on one CPU. On a slow computer, this time-step might be so big, that the simulation gets unstable. This means, that the performance of the computer, where the simulation is executed, influences the quality and stability of the simulation directly! Of course, the possibility of running the simulation on a different time-scale than real-time is possible. Therefore, one has to enter `SLOMO x` at the Unreal console[5], where `x` is a scale factor relative to real-time. A factor lower than 1 means, that the simulation is executed slower.

5.2.4 Coordinate Frames, Units and Scale

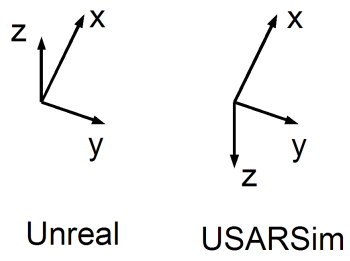

Figure 5.3: Unreal and USARSim coordinate frames

Fig. 5.3 shows the two coordinate systems used within the simulator: The one of Unreal is a left-handed system with its positive x-axis pointing forward, the y-axis to the right and z upwards. The only difference in the USARSim system is that the positive z axis is pointing downwards. Thus, it is similar to that one used for modeling in chap. 3.

Unreal uses its own units, called Unreal Units (UU) for length and angle. Conversion to SI units is done by

$$\begin{aligned} 1\ [m] &= 250\ [UU] \\ \pi\ [1] &= 32768\ [UU] \end{aligned}$$

USARSim always communicates in SI units with the controllers. All numbers sent by USARSim are floating point with 4 digits precision. To cause perfect confusion, there is also a length scale between Unreal and Karma:

[5] On a German keyboard, the console can be opened and closed with the key "ö"

$$50\ [UU] \quad = \quad 1\ [KarmaUU]$$

More to scale can be found in [WB] and [Wan].

5.2.5 Unreal Script

Unreal Script is the script language of the Unreal engine and is used to create a game logic. On basis of this language it is possible to build special games and game modifications (mods) on top of the Unreal engine. USARSim can be understood as such a special game, and its source code (which is open) is written in this language. Unreal Script is an object-oriented high-level programming language with the following features[6]

- a pointerless environment with automatic garbage collection
- a simple single-inheritance class graph
- strong compile-time type checking
- a safe client-side execution "sandbox"
- the familiar look and feel of C/C++/Java code.

The code is built using a compiler[7] which is delivered with the Unreal computer game.

In USARSim, every robot model and every part of a robot is defined by a class. If we consider a part, then all the dynamic parameters of the part are specified in the associated class. Also the static mesh, which defines the graphical representation and the collision object, is set there. All parts in USARSim are derived from the `KDPart` class, and all robots from the `KRobot` class. These abstract classes provide several important methods. A very important section in a robot class is the function `ProcessCarInput()`, which is executed after every Karma pipeline run. There, new commands, that were sent from a controller are available and can be further processed. This means, that the method `ProcessCarInput()` uses the information from the control command to find forces and torques that act onto the parts of the robot so that the command is executed. In a real-world scenario, this would be the low-level controller, including actuators, energy storage,

[6] http://udn.epicgames.com/Two/UnrealScriptReference.html
[7] UCC.exe

etc.. Also the sensors, the communication base station, and so forth are programmed in Unreal script.

5.2.6 USARSim UAV model

USARSim already provides a model for a quad-rotor helicopter, namely the AirRobot[8]. Fig. 5.4 shows its graphical representation.

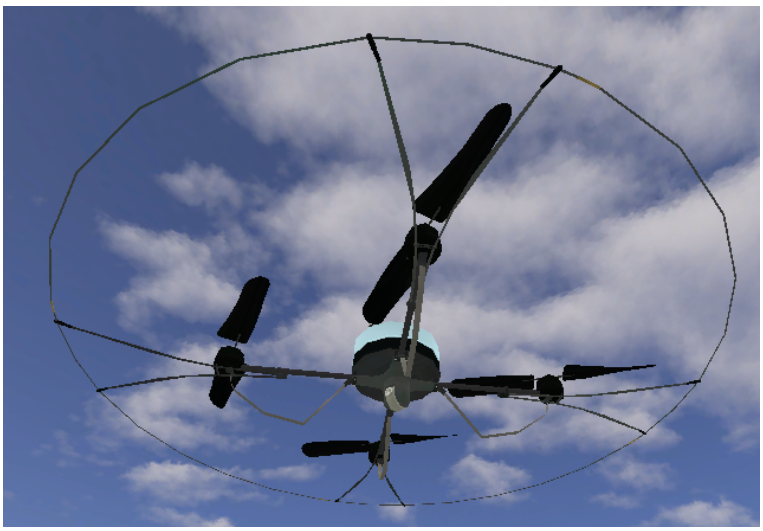

Figure 5.4: Graphical model of the AirRobot in USARSim

As the model is strongly simplified and differs from the equations presented in 3, it is analyzed here. Moreover this section should give a small insight into defining robot models in USARSim. The dynamic parameters, geometry and the execution of commands from the controller are all set in the appropriate Unreal script classes or its configuration files. The AirRobot class[9] itself is derived from several other classes. This hierarchy is shown in fig. 5.5.

The AirRobot class defines the physical and graphical parameters of the UAV such as e.g. the static mesh which is displayed, maximum speeds etc. View lst. 5.1 to see how these parameters are set. In this listing, `***` means an arbitrary number of arbitrary signs. The `max***Velocity` values define the maximum velocity of the robot in the three translational dimensions in $[m/s]$ and the yaw rotation in $[s^{-1}]$. Notice that due to this definition, the maximal reachable velocity is $\sqrt{3} \cdot 5$ $[m/s]$, since the velocity reaches its maximum absolute value for ± 5 $[m/s]$ in any of the 3 directions. `Dimensions` defines

[8] http://www.airrobot.de/
[9] found in `AirRobot.uc`

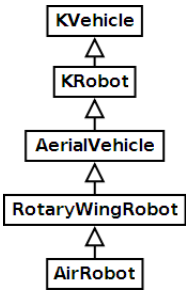

Figure 5.5: Derivation hierarchy of AirRobot class

Listing 5.1: AirRobot model parameter definition in AirRobot.uc

```
StaticMesh=StaticMesh'USARSim_V***_Meshes.AirRobot.AirRobotBody'

    maxAltitudeVelocity = 5;
    maxLinearVelocity = 5;
    maxLateralVelocity = 5;
    maxRotationalVelocity = 1.5708;

    Dimensions=(X=0.99968,Y=0.99934,Z=0.19382)

    Begin Object Class=KarmaParamsRBFull Name=KParams0
        ***
        KFriction=0.5
        KLinearDamping = 2.776
        KAngularDamping = 0.94135
        KCOMOffset=(X=0.0108,Y=0.0102,Z=0.5)
        KInertiaTensor(0)=0.009428
        KInertiaTensor(3)=0.009428
        KInertiaTensor(5)=0.7288
        ***
     End Object
     KParams=KarmaParamsRBFull'USARBot.AirRobot.KParams0'
```

the dimensions of the robot. Since the graphical representation of the AirRobot was not changed, its dimensions are also unchanged here. The other parameters are[10]:

- `KFriction` is a contact friction coefficient. It is multiplied pairwise to get contact friction[11]
- `KLinearDamping` is the linear damping or drag coefficient
- `KAngularDamping` is the angular damping coefficient

[10] http://wiki.beyondunreal.com/UE2:KarmaParams_%28UT2004%29
[11] http://wiki.beyondunreal.com/UE2:KarmaParamsCollision_%28UT2004%29#KFriction

Listing 5.2: AirRobot model configuration section in USARBot.ini

```
1 [USARBot.AirRobot]
  Weight=1 Payload=0.02 ChassisMass=1
3 JointParts=(PartName="Counter_Propeller_1", ...
  JointParts=(PartName="Counter_Propeller_2", ...
5 JointParts=(PartName="Propeller_1", ...
  JointParts=(PartName="Propeller_2", ...
7 MisPkgs=(PkgName="CameraPanTilt", ...
  Cameras=(ItemClass=class'USARBot.RobotCamera', ...
9 Sensors=(ItemClass=class'USARBot.GroundTruth', ...
  Sensors=(ItemClass=class'USARBot.GPSSensor', ...
11 Sensors=(ItemClass=class'USARBot.INSSensor', ...
```

- `KCOMOffset` is the offset of the center of mass from the origin of the robot
- `KInertiaTensor` is the moments of inertia tensor

The parameters shown in lst. 5.1 are already the parameters for the Microdrone.

Moreover, the AirRobot class specifies the configuration file for the class[12]. There (see lst. 5.2), some other properties of the robot are defined. The advantage of parameters defined in the ini-file is, that no recompilation of the associated class is necessary after these values are changed.

Here, the parameters `Weight` and `Payload` do not influence the dynamical behavior of the model at all, so they are dummy parameters. `ChassisMass` is the mass of the whole robot model without propellers. The `JointParts` parameter is used to place the Propellers[13] relative to the Chassis, it gives the parts a name and sets the Unreal Script class, that implements the part. `MisPkgs` and `Cameras` are not of interest here and the parameter `Sensors` defines which sensors should be plugged where on the robot. Again, also the implementing class is specified.

Model instantiation and control A model can be instantiated in an Unreal map in the following way: First, the Unreal server must run with a map loaded. Then, a client can connect to the server. After such a connection is established, the server sends a so-called `NFO`-Message. A client should not send commands before receiving this message. The `INIT` command is used to spawn a robot in the map:

```
INIT {ClassName USARBot.AirRobot} {Name R1} {Location -9.76,61.45,-0.7}
```

[12] which is `USARBot.ini`

[13] Those are defined in `Propeller.uc`

```
{Rotation 0,0,0}
```

Here, the linefeed was only introduced for readability. Every command must, according to the Gamebots protocol end with carriage return and linefeed symbols. The `ClassName` tells USARSim which robot model to instantiate. `Name` is an arbitrary name, `Location` and `Rotation` define the initial robot pose[14] relative to the map origin in $[m]$ and $[rad]$, respectively. Such spawning poses are typically delivered with the map or determined by using the `GETSTARTPOSES` command.

A controller can fly the AirRobot model by sending `DRIVE` commands. Such a command could look like this:

```
DRIVE {AltitudeVelocity 1.5} {LinearVelocity 0.1}
```

Given these velocities, the model must provide equations to calculate the forces and torques and their point of application in order to achieve these velocities. In reality, a low-level controller would adjust the angular velocity of the propellers, so that the forces created by the rotors are able to reach and maintain the desired velocities. But in this case, the complicated way of designing a controller was omitted. Instead, the model was designed to represent a very simplified desired behavior only. For the cDrones project this situation is preferable, because the real UAV (Microdrone) is already controlled as well and its controller is unknown.

In this simplified model, a force and a torque, that act on the center of mass are calculated. The implementation of this calculation is found in `RotaryWingRobot.uc` and is done in the following way:

$$\begin{aligned} f_{lin} &= 5.05\ v_{Linear} \\ f_{lat} &= 5.05\ v_{Lateral} \\ f_{alt} &= 5.05\ v_{Altitude} + 49.02 \end{aligned}$$

where the values for v are those specified in the `DRIVE` command. Due to a forum entry on the sourceforge-USARSim page[15], the value 49.02 as all other constants in this model are results of experiments and the original idea was to implement a lookup table, which was never completed. These constants were written directly in the code (hard-coded, no

[14] A pose specifies the complete state of a robot.

[15] http://sourceforge.net/projects/usarsim/forums/forum/486389/topic/1741045?message=4326574

variable name or anything helpful was given). From these equations, one can estimate that 49.02 stands for the hover-thrust of the AirRobot. The forces in the above equations are exposed a rotation about the yaw angle only, before they are applied to the COM of the body by Karma.

The torques acting on the UAV are calculated in the following way: At first, set-point pitch and roll angles are computed:

$$\begin{aligned} \theta_{set} &= 0.0872665\ v_{Linear} \\ \phi_{set} &= 0.0872665\ v_{Lateral} \end{aligned}$$

Then, the torques rotating the robot in pitch direction is given by

$$\begin{aligned} \tau_{Prop} &= \frac{f_{alt}}{8}\ l \\ \tau_\theta &= \begin{cases} 0 & \text{if } |\theta_{set} - \theta_{actual}| < 0.02 \\ -\tau_{Prop} & \text{if } \theta_{set} > \theta_{actual} \\ \tau_{Prop} & \text{else} \end{cases} \end{aligned}$$

where l is the distance between a rotor and the COM. Again, the constants are not described. This torque is just something like a show, it does not influence the robots translational dynamics at all. Its only use is to make the UAV model tilting to fake the graphical impression of real dynamics. Identically, the torque around roll is computed and applied. The yaw-torque is

$$\tau_\psi = 5.25\ v_{Rotational}$$

where $v_{Rotational}$ is the `RotationalVelocity` parameter of the `DRIVE` command.

However, the model shows good stability properties. If the parameters of the model (which are in this case the hard-coded constants) are adjusted, then the behavior of the real Microdrone could be approximated in a sufficiently accurate way. The situation turns out to be even better: The weight of the AirRobot is approximated by $1[kg]$, so that there is no need to change the hover-thrust. By the way, the value 49.02 is equal to the gravitational force in $[KarmaUU]$. Fig. 5.6 shows an experiment, where the USARSim

AirRobot model is commanded to accelerate to 5 $[m/s]$ (the ripple comes from numerical derivation). As this figure proves, the value is reached and there is no need to change the AirRobot model at all in order to match the specifications of the Microdrone. For yaw-commands, the experiment delivers similar results. Pitch and roll angle were not investigated, since they do not influence dynamics anyway. Obviously, the rotation of the propellers has nothing to do with dynamics, too and is also done for graphics only.

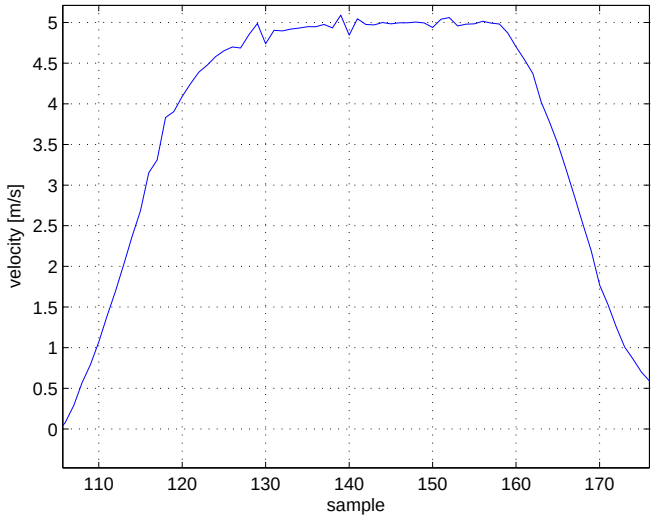

Figure 5.6: AirRobot model verification

5.2.7 USARSim Sensor Models

USARSim provides a lot of sensors, where only those used in this project are introduced here. The problem of accurately simulating the whole navigation system was already mentioned in sec. 2.2.

5.2.7.1 Ground Truth Sensor

This is a sensor, that delivers exactly those poses of the UAV that were calculated by the simulator, that is where the robot actually is placed in the world. In real-world, this sensor obviously does not exist.

5.2.7.2 GPS Sensor

The details to the GPS Sensor of USARSim are described in [BC08].

Listing 5.3: GPS sensor configuration in USARBot.ini

```
maxNoise=5
minNoise=1
ScanInterval=0.25
...
```

Satellite positions Because of numerical problems with Unreal Script, the developers were not able to use SGP4 for satellite tracking (see sec. 2.2.1). Instead, a look-up table was generated containing satellite positions over time, which is stored in a configuration file that is readable for Unreal script. For large time steps, this method does not reflect realistic satellite positions, but reduces computational burden.

Signal and noise As introduced in sec. 2.2.1, real GPS receiver precision depends on the number of satellites, their geometric arrangement (i.e. the dilution of precision) as well as other influences (e.g. multi-path effect). Thus, as a first step, all satellites with an elevation angle lower than 5 degrees are discarded. Secondly, ray tracing is applied to eliminate more of the observed satellites. Depending on the number of satellite signals received, a special amount of Gaussian noise is added to the position measurements. This amount is configurable using the `USARBot.ini`-file. In section `USARBot.GPSSensor`, which is shown in lst. 5.3, the values for `maxNoise` and `minNoise` can be configured[16]. These specify the triple of the standard deviation used in the case with 4 and 12 satellites respectively. If the number of satellites seen is somewhere in between, the sensor linearly interpolates between these max and min values. The `ScanInterval`-parameter specifies the sample time of the sensor in $[s]$.

Flat earth assumption and position reference To reduce computational complexity, the developers of the sensor assumed earth to be flat and a linear relation between longitude (λ) and latitude (ϕ) angles and X and Y USARSim map-coordinates that is

$$\begin{aligned} X[m] &= 111200\ \varphi[deg] \\ Y[m] &= 71670\ \lambda[deg] \end{aligned}$$

For this project, the absence of an altitude measurement is one of the major drawbacks

[16]The sensor including altitude measurement is configurable in `cDrones.ini`

of this sensor, although the flat earth assumption holds. Thus, the sensor implementation was extended by an altitude measurement[17], which was implemented as ground thruth height plus noise. For the mapping of a real-world location to the virtual one three methods are possible. For the method done in this project, UnrealEd was used to place a NavigationPoint (which is a special Unreal script object) somewhere in the map. Then, the latitude and longitude of this point in the world were set. For the other methods and more details, see sec. B and [WB].

5.2.7.3 Inertial Navigation System

Inertial sensors measure acceleration and rotation rate relative to an inertial reference frame. But, what is simulated here is not just the IMU: It is the whole inertial navigation system. As the error behavior of an INS was, because of complexity, not considered in chap. 2.2.2, that isn't done here either. However, the procedure used to simulate the INS was inspired by the way how a real INS calculates pose estimates (strapdown computation). What follows is a short description about how poses are calculated in USARSim INS Sensor [SMB07].

The simulated INS sensor from USARSim uses a Gaussian random number (Box-Muller method) generator to add noise to ground truth measurements of angular rate and velocity. That is a simplification, since white noise is a model of the noise in the acceleration. Thus, drift should already occur in the velocity. Another interesting discovery was made in the code of the INS Sensor: The sensor offers a mode where drifting is simulated, and one where that is not done. In the drifting mode, the mean value for the random number generator for the noise is shifted by a constant random number, which is generated at initialization. Let me summarize the model as it exists in USARSim[18]:

Let $\begin{bmatrix}\hat{x}_k & \hat{y}_k & \hat{z}_k & \hat{\phi}_k & \hat{\theta}_k & \hat{\psi}_k\end{bmatrix}$ be an initial pose estimate and $\begin{bmatrix}x_k & y_k & z_k & \phi_k & \theta_k & \psi_k\end{bmatrix}$ the ground truth initial pose. Then the algorithm computing a pose estimate update in the USARSim INS sensor is:

1. Compute angular velocities for current time step using ground truth[19]:

[17] This sensor is implemented in the class `cDronesGPSSensor.uc`.

[18] This algorithm is not implemented as described in [SMB07]! The true implementation is found in `INSSensor.uc`

[19] Notice that, according to (3.1), this way of computing the body-frame angular velocities is wrong!

$$\begin{bmatrix} \omega_{x,k+1} \\ \omega_{y,k+1} \\ \omega_{y,k+1} \end{bmatrix} = \frac{1}{\Delta t} \left(\begin{bmatrix} \phi_{k+1} \\ \theta_{k+1} \\ \psi_{k+1} \end{bmatrix} - \begin{bmatrix} \phi_k \\ \theta_k \\ \psi_k \end{bmatrix} \right)$$

2. Use a Gaussian noise model $\mathcal{N}(\mu, \sigma)$ and ground truth angular rates to update the orientation estimate:

$$\begin{bmatrix} \hat{\phi}_{k+1} \\ \hat{\theta}_{k+1} \\ \hat{\psi}_{k+1} \end{bmatrix} = \begin{bmatrix} \hat{\phi}_k \\ \hat{\theta}_k \\ \hat{\psi}_k \end{bmatrix} + \begin{bmatrix} \omega_{x,k+1} \\ \omega_{y,k+1} \\ \omega_{y,k+1} \end{bmatrix} \Delta t + \begin{bmatrix} \omega_{x,k+1} \\ \omega_{y,k+1} \\ \omega_{y,k+1} \end{bmatrix} \mathcal{N}(\mu_{r,k}, \sigma)\ \Delta t$$

3. Compute ground truth velocities:

$$\begin{bmatrix} v_{x,k} \\ v_{y,k} \\ v_{z,k} \end{bmatrix} = \frac{1}{\Delta t} \left(\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \end{bmatrix} + \begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix} \right)$$

4. Use ground truth information to compute the distance traveled during the last time step:

$$d = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

5. Using the results from the previous steps, calculate the new position estimate:

$$\begin{bmatrix} \hat{x}_{k+1} \\ \hat{y}_{k+1} \\ \hat{z}_{k+1} \end{bmatrix} = \begin{bmatrix} \hat{x}_k \\ \hat{y}_k \\ \hat{z}_k \end{bmatrix} + \Delta t \begin{bmatrix} v_{x,k} \\ v_{y,k} \\ v_{z,k} \end{bmatrix} + d\ \mathcal{N}(\mu_{t,k}, \sigma)$$

In the Gaussian noise model, μ and σ stand for mean and standard deviation, respectively. These values can be influenced using a configuration file, which is shown in lst. 5.4. There, `Sigma` defines σ in $[m]$. The value assigned to `Noise` has no effect. The boolean parameter `Drifting` sets wheter the INS sensor simulates drift or not. In non drifting mode, both values $\mu_{t,k}$ and $\mu_{r,k}$ are zero for all time. When `Drifting` is set to true, $\mu_{t,k}$ and $\mu_{r,k}$ are calculated as follows:

Listing 5.4: INS sensor configuration in USARBot.ini

```
Sigma=0.01
Noise=0.01
Precision=1000
Drifting=false
ScanInterval=0.01
...
```

$$\mu_{t,k} = \mu_{t,k-1}\,\frac{a}{p}$$
$$\mu_{r,k} = \mu_{r,k-1}\,\frac{a}{p^2}$$

where a is a random value in $[-0.5, 0.5]$ and p is the `Precision`. The value for a is only computed once when the INSSensor class is instantiated.

5.3 Simulator Middleware

Fig. 5.7 shows the system architecture of the simulator system coupled to the cDrones software architecture, as it was defined by researchers at Lakeside Labs.

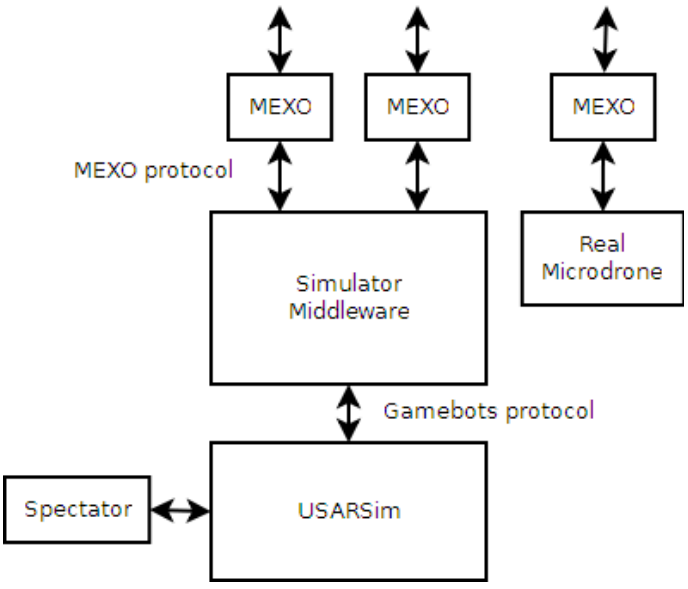

Figure 5.7: The simulator in the cDrones system architecture

The blocks denoted with MEXO are so called Mission Executors, where each of them

is responsible for sending the appropriate commands to its assigned UAV. Such a UAV could either be a real or a simulated one. These MEXOs are already part of the cDrones software architecture and not topic in this thesis. They communicate with the *simulator middleware* (SM) via a TCP session using a special protocol, which is defined in appendix B.

The need for this middle-ware arose by the following reasons:

- Simulating multiple instances of a robot in USARSim.
- A unique interface between MEXO and simulator and MEXO and real UAV.
- Implementation of controllers for way-point-based flight.
- Expandability of the simulation.
- The data sent by the MEXO is a mission plan, that can contain a lot of way-points and other information needed to execute a mission. Consecutive execution of these commands is also maintained by the middle-ware.
- Sensor data preprocessing.

5.3.1 Simulator Middleware implementation details

During the progress of this work, two implementations for a middle-ware were developed. The first one was written in Java, which was then replaced by a LabVIEW implementation. The choice for LabVIEW was based on its widespread distribution in the field of robotics and control engineering. As the original intention was to develop a middle-ware for simulating not just UAVs but also other types of robots, this section is kept a bit more general.

From a networking point of view, this middle-ware must act as a server to the MEXOs and as multiple clients to USARSim. Whenever a MEXO connects to this middle-ware and sends a proper message, it has to open a connection to USARSim and create a UAV there.

From a control point of view, the SM must select proper control algorithms and execute them. If the command to be processed is “fly to a special waypoint”, then obviously another controller is needed than for landing.

From a software development point of view, all of the controllers must be executed concurrently. Sensor measurements, received via the connection to USARSim, must be

processed immediately in order to provide the control algorithms with up-to-date data. Commands must be sent to USARSim at a proper frequency.

To solve all these issues, LabVIEW-VIs were developed that are presented in Appendix A.

5.3.2 An Alternative UAV Model

In sec. 5.2.6 it was explained, how a `DRIVE` command is processed to find forces and torques that are applied to the UAV model using Karma. But if complex calculations are done within the `ProcessCarInput()` method of a class, then the time-step of the whole engine would increase and endanger overall stability. So, the complex computations must be done outside of this critical thread. The solution is to simply send the 3 forces and torques as parameters of the `DRIVE` command. Complex model computations and control could then be executed in the simulation middleware . Therefore, a new robot class (`cDronesMicrodrone.uc`) was created, that simply passed the forces specified in the drive command on to Karma. Model parameters were set accoding to the parameters identified in chap. 4. For the `DRIVE` command to support the new parameters, also the class `USARBotConnection.uc` was slightly modified. What remains to be done for using this model in the cDrones project is:

- validate the results of the simulation of this model
- identify the controller of the Microdrone and implement it in the simulation middleware . Without being able to compute the input forces, this model is useless.

5.3.3 Configuration of the simulation middleware

All configuration parameters of the LabVIEW part of the simulation middleware are stored in an XML file. An example is given here:

The first six properties are of the type float. ReferenceLatitude and ReferenceLongitude are in degrees, the other four in meters. In order to argue the need of these parameters let me describe the way how coordinates are used in USARSim and the simulation middleware (see also sec. 5.2.7.2):

The position information in the Mission Plan are true GPS positions, either in elliptical or cartesian ECEF format. These positions correspond to real places, e.g. the corner of a specific building at the lakeside campus. In a virtual USARSim map of the lakeside campus exactly the same corner is measured relative to the maps origin, which is not comparable

Listing 5.5: simulation middleware configuration

```
<?xml version="1.0" encoding="UTF-8"?> <Properties
xmlns="http://pervasive.uni-klu.ac.at/cDrones/xml/cDrones">
<Property key="ReferenceLatitude">46.64305555</Property> <
   Property
key="ReferenceLongitude">14.3416666</Property> <Property
key="ReferenceAltitude">460</Property> <Property
key="ReferenceX">772.789</Property> <Property
key="ReferenceY">597.415</Property> <Property
key="ReferenceZ">0</Property> <Property
key="PositionSensor">SEN.GT</Property> <Property
key="RobotModel">cDrones.cDronesMicrodrone</Property> <Property
key="SimulatorSampleTime">100</Property> <Property
key="LocalListenerPort">7015</Property> </Properties>
```

with our true GPS position. Thus a reference point is needed, from which we know the true coordinates and its position in the map specific coordinate system. Given such a reference point, one can transform points from one coordinate system into the other. In fact, there are a few more issues that need to be taken into account in order to do a proper transformation, e.g. scale, the geographic world model [HWWL03]. Therefore, USARSim comes with an own definition of a geographic coordinate system, where the world (map) is assumed to be planar (see sec. 5.2.7.2).

Please refer to [WB] for a documentation about how to place such a reference point in USARSim. In fact, the values entered in USARSim must be the same as in this configuration file.

The property PositionSensor defines, which sensor is used to

- measure the current pose that is sent to the Mission Executors by the POSE message
- measure the current pose for the control algorithms.

This property is an enumeration and possible choices are

- SEN.GT for the ground truth sensor,
- SEN.GPS for the GPS sensor,
- SEN.INS for the inertial navigation system.

The next property, RobotModel, defines which robot model should be used in USARSim. The first part (before the dot) specifies the package and the second one the model

in the package. Especially for the cDrones project, the model cDronesMicrodrone in the cDrones package is prepared. Another choice could be `USARBot.AirRobot` for the model that comes with USARSim.

SimulatorSampleTime is a (32 bit unsigned integer) value for the desired sample time the simulation middleware should run with. However, if the system's performance is not sufficient for running at this sample time a higher one is used instead. Thus, this value can be understood as a lower bound.

The LocalListenerPort specifies the TCP listener port of the simulation middleware that the Mission Executors use to connect to the simulation middleware . Hence, this value should be a 16 bit unsigned integer higher than 1024.

USARSim configuration USARSim can be configured by using the configuration file of the cDrones package, cDrones.ini , which is placed in the System directory of the Unreal Tournament 2004 installation. In this file you can change properties of the

- cDronesMicrodrone model
- cDronesGPSSensor

For the robot model, these properties are e.g. Weight and batteryLife as specified in sec. 5.2. The sensor has e.g. minNoise and maxNoise as properties. The parameters of these models are the same as its equivalents in the USARBot package, so you can find information about them again in [WB].

5.3.4 LabVIEW design patterns

Design patterns are guidelines for software architectures to solve software design problems that often occur. In LabVIEW, such design patterns are often used (although probably seldom termed design pattern). One of those is the immediate subVI, where the connector pane is separated in the typical 4x2x2x4 pattern, error in and error out clusters provided and an error case structure skips execution of the block diagram on errors. The dataflow also follows the error wire. Another very often used design pattern is the functional global variable. It consists of a while loop with an uninitialized shift register and a constant at the conditional terminal of the loop. At every call, the loop is iterated once, and the data in the shift register can be processed. If there is done more than just reading and writing the value, this design pattern is sometimes referred to as action engine. There are a good deal more design patterns, such as many different variations of state machines

or whole application frameworks. Some of them are described in [Blu07], others on the National Instruments web-page[20]. In the next few chapters some more or less special design patterns used in this project are presented.

Producer/Consumer is used for communication between separate loops. The loops could be in different VIs - the only thing they have to share is a reference to a queue or its name. The producer loop generates data and enqueues it as element to this common queue. As the name implies, the consumer dequeues the elements.

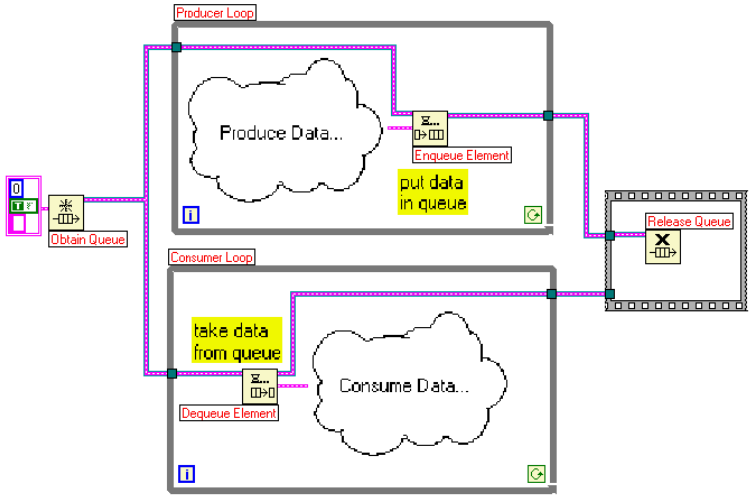

Figure 5.8: Producer/Consumer design model[21]

This design pattern is extensively used in this work. A scheme of the usage of the Producer/Consumer pattern in this project is shown in fig. 5.9

To avoid overflows at the queues, it is necessary to ensure that the producer works at a lower rate than the consumer is capable of. In LabVIEW, loops typically contain a timing VI (otherwise they run at the highest frequency possible). According to the above statement it was necessary to set a lower period in the consumer than in the producer. But, in order to perform as high as possible, no timing VI was used in the Producer/Consumer pattern loops. Instead, the timing was implicitly set using the built-in timeouts of the Dequeue Element-VI and the TCP Read-VI that come with LabVIEW. Thus, if a queue contains a lot of elements, the consumer operates at the highest frequency possible. Otherwise, the consumer waits until timeout expires.

[20] http://www.ni.com
[21] Source: http://zone.ni.com/devzone/cda/tut/p/id/3023

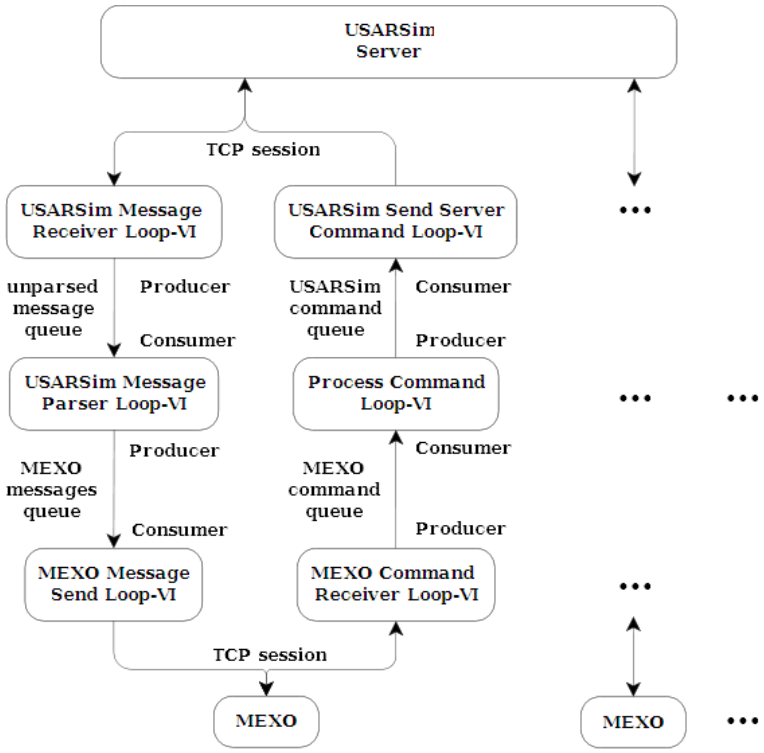

Figure 5.9: Producer/Consumer usage in simulation middleware

Queued Round Robin is a design pattern that I have not found neither in literature nor the web (so the name evolved from the necessity to call it somehow). Maybe that is because of a very popular paradigm having almost the same use case, namely object oriented programming. This use case is the following: An arbitrary number of robots should be simulated. This number should be dynamic, i.e. it should be possible to create and destroy robots without stopping the simulation. All robots are for themselves of similar structure, e.g. have the same sensors. The methods applied to the robots (e.g. sensor measurements) are identical for all of them. In an object oriented language, a programmer would have defined a robot class, which is instantiated when needed.

Maybe it would have been better to use LabVIEW object-oriented programming instead of this Queued Round Robin design pattern. But, as experiments have shown, it performs well and offers more flexibility.

Now to the pattern itself: A robot is represented as instance of a typedef, namely the robot-cluster as described in sec. 5.3.1. It consists of a name, a starting configuration, TCP session references to USARSim and MEXO and quite a lot of queue-references. One queue reference for each queue shown in fig. 5.9, so every robot has its own unparsed

messages queue, MEXO messages queue and so forth.

If there are multiple instances of such robots, then they need to be stored somewhere, and that is - how could it be any different - a queue, which I will call robot queue from now on.

Assume there are four robots in the queue, four MEXOs connected to the simulation middleware and four UAVs flying in USARSim which provide sensor measurements at USARSim's four TCP sessions. Principally, there should be something that processes the messages received from USARSim, one for each robot. That is what Queued Round Robin is used for. E.g. the USARSim Server Message Receiver Loop-VI picks the first robot, reads its TCP connection and enqueues the received data onto the unparsed messages queue of this first robot. Then, it goes on with the second robot and so forth, that's where round robin comes from. Fig. 5.10 shows the USARSim Send Command Loop-VI as an example.

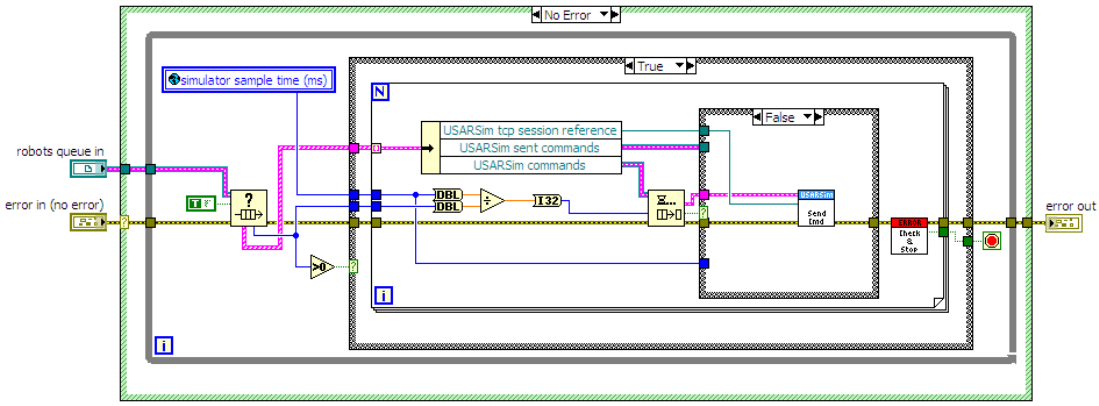

Figure 5.10: Queued Round Robin example

The leftmost subVI is the LabVIEW-built-in Get Queue Status-VI. A "true" constant is wired to its "return elements" terminal. Thus, all robot clusters that are on the robot queue are returned in an array. If there are robots in the queue, then the for-loop iterates through them and applies the code inside of the loop to each of them. The "# elements in queue"-terminal from the Get Queue Status-VI is used for two issues:

1. To skip further progressing if no robot is queued. In this case (not shown in fig. 5.10), a Wait Until Next Multiple-VI is used to define timing of the loop.

2. To calculate the timeout for the Dequeue Element-VI. This is done by dividing the

simulator sample time through the number of robots, so that one loop cycle lasts (a bit longer than) simulator sample time. Of course, the division should be done outside of the for loop, but it wouldn't fit on the paper then.

Using the timeout instead of an explicit timing VI has the advantage that, if there is a command on the queue, it is sent immediately and the next robot is on turn without delay.

That introduces a very important condition for efficiency of this Queued Round Robin pattern: The queues of the different robots need to be approximately equally filled. In this case, all queues are processed at maximum speed. Otherwise, if the queue of a robot contains many elements and all the others are empty, then this pattern would be inefficient. Between the dequeue of two elements a bit (a factor of $\frac{\#nonemptyqueues}{\#emptyqueues}$) less than the simulator sample time expires.

In this project, all queues will be approximately equally filled. All of the sensors in USARSim deliver readings at a constant rate, which is the same for all instances of the UAV model. On the other side, the controllers will generate control commands at a constant rate, which is again the same for all UAVs. Only the MEXO commands are retrieved in a bulky way. But it can be expected, that a mission plan is uploaded to all drones at once, and so Queued Round Robin is a good choice again. However, MEXO command retrieval does not have so strict timing constraints anyway.

Single Element Queue is a well documented design pattern for creating a reference object in LabVIEW. A very good explanation for its use and benchmarks can be found on the web[22]. It is the best approach for accessing data by-reference in LabVIEW. The principal structure of the pattern is very easy: For initialization, a queue is created and one single element with an arbitrary datatype is enqueued. A good choice for a datatype is a type-defined cluster, because it is flexible and easy to use. The Obtain Queue-VI has an output port for a queue reference, which is now the desired reference to the data. The main advantages of this pattern are:

- Thread-safety: If the Dequeue-VI and the Enqueue-VI are used for reading and writing data, thread-safety is an inherent feature. If one thread is currently working on the data, the other thread finds the empty queue. Important: Do not use the Preview Queue Element-VI for the Single Element Queue!
- Performance: Accessing a single element queue is very fast and the data is not copied

[22] http://forums.ni.com/

on dequeue[23].

In the new version LabVIEW (2009) was extended by a new feature called data value reference. With this feature, there is no need for a single element queue any more. Unfortunately, this version was not released before the end of this work.

In the simulation middleware, the single element queue was used to store state information, thus it was called single element state queue. In the robot cluster, a reference is designated to such a queue, so every robot has its own single element state queue. In the USARSim Server Message Parser-VI, updated state information is extracted from the USARSim messages and used to update the state element. All the controller-VIs use the single element state queue to read the current state of the robot.

[23]http://forums.ni.com/ni/board/message?board.id=170&message.id=264865&query.id=72398#M264865?

Chapter 6

Conclusions and Future Work

An autonomous unmanned aerial vehicle like the quadrotor helicopter addressed in this work is not just a futuristic device. It is also interesting from the point of view of "older" fields of science, like mechanical modeling and kinematics. The first part of this work is dedicated to these topics, which seem to be well-known, especially in the range as applied here. But it turns out to be different: A master student like me gets confronted with a real vehicle and is eager to apply all the theoretical knowledge that he gained during his study. After some research in publications, some implementations, some profitable discussions it turned out that something was wrong. Many of the publications concerning quadrotor-helicopter control were based on a model, that was derived with lack of mathematical care [KKP09]. Another master student, who wrote his master's thesis also about quadrotor helicopters [Bre08] already derived the equations of motion in a consistent way, but using a slightly different approach, namely the Newton-Euler formulation. In fact, the difference between the "sloppy" models and the correct ones is small, especially when a small-angle assumption concerning roll and pitch angles is made. So, for control design of course all models were sufficiently accurate. In this work, I tried to follow a mathematically precise and comprehensible way of deriving the equations of motion relative to all coordinate frames that could possibly be considered for control design.

As going-to-be control engineer, thoughts concerning model analysis need to come to mind immediately when a system of differential equations is espied. However, that would have gone beyond the scope of this work, but might be a good point for future work. Moreover, a validation of the nonlinear model in ranges where the nonlinearities strongly influence dynamics was interesting. At this opportunity, also effects that were not modeled until now could be considered. Possible model extensions reach from forward-

flight aerodynamics over propeller gyroscopic effects to external influences like wind or aerodynamic ground effects.

Figure 6.1: The quadrotor helicopter in a Robocup-Rescue environment

The use of the linearized model for parameter identification led to a complete model description for the mechanical part of the Microdrone. The idea behind that was not just model verification and finding reference values for the USARSim UAV model, but also to build a more accurate drone model in USARSim. Such a model for the mechanical part was developed within this work, but it is not usable in the cDrones testbed until now, because there exists no controller to it. The plan was to also identify the controller and implement it within the simulation middleware. Then a very accurate simulation of the dynamics of the whole Microdrone could have been found. A further advantage of this approach was, that aerodynamics, wind and disturbances could be simulated in the simulation middleware. That would lead to a more detailed simulation without increasing the computational burden for the Unreal engine.

However, the results achieved using the slightly modified model of the USARSim UAV were really good either. The simulation middleware, which was the main implementational

effort in this thesis provides the cDrones project with an easy-to-use and powerful interface for real-world like simulation. The MEXO-protocol was designed so that the commands and messages are equal for the real and the simulated drone. Using this protocol, the researchers are able to automatically simulate multiple drones concurrently within USARSim. Furthermore, tools for manual flight and flight data analysis were provided. Using the simulation middleware, disturbances like wind etc. could also be implemented as well as communications simulation or other extensions. The flexibility of this whole structure is tightly coupled to the cDrones software architecture and the power of LabVIEW.

Appendix A

Simulator Middleware VI descriptions

In the following chapters, the VIs implementing the simulator middleware are introduced. At the beginning, the RobotCluster datastructure is shown as this will make future descriptions easier.

A.1 RobotCluster

The RobotCluster is a LabVIEW cluster type-definition, that is capable of storing all information about one particular robot. It's contents are:

- A TCP session reference for the connection to USARSim
- A TCP session reference for the connection to its MEXO
- Several queues concerning the connection to USARSim:
  - A queue for raw messages
  - A queue for parsed messages
  - A queue for commands to be sent
  - A queue for sent commands
- Several queues concerning the connection to the MEXO:
  - A queue for commands received from the MEXO
  - A queue for messages to be sent to the MEXO

- A cluster storing the start configuration of the robot
- A so called single element state queue

All queues except the single element state queue are part of a producer-consumer LabVIEW design pattern. The single element state queue is part of another design pattern, which will be introduced later.

A.2 SimServer-VI

is the main VI of the simulator. It starts USARSim and the simulation middleware . On its front panel, there are five controls:

- A stop-button, which stops the execution of the whole simulation middleware but not USARSim. As all connections to USARSim are closed, all robots will be removed from the world there. All data that was not saved during execution is lost.
- A switch for turning manual control on or off. If switched on, the simulator middleware ignores commands from the MEXO. A special VI (ManualDroneControl.vi) can be used to control the UAV. When switched off, ManualDroneControl.vi doesn't work but MEXO commands are processed.
- A file path control, that specifies the path and filename of the configuration file for the simulation middleware (see sec. 5.3.3).
- A String control which defines the path of the USARSim executable (including arguments). It also possible to specify a windows-batch-file here that starts USARSim.
- Another file path control containing the working directory of the above specified USARSim executable (or the batch file).

Furthermore, there is a LED on the front panel, that indicates whether an attempt to stop the simulation middleware was initiated or not.

The SimServer-VI consists of two main sub-VIs, that are the MEXOCommunication-VI and the USARSimServerCommunication-VI, shown in fig. A.1. The order of the following descriptions of the subVIs is depth-first.

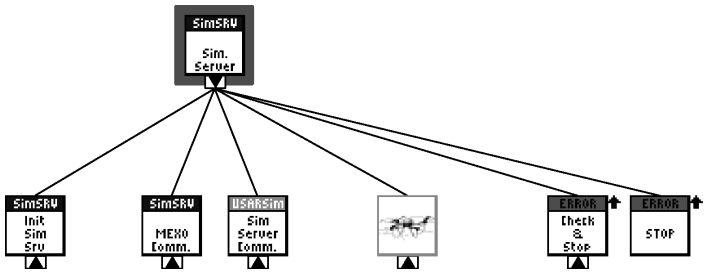

Figure A.1: First Level of the VI-hierarchy of the SimServer-VI

A.3 SimServer Init-VI

reads configuration information from the XML configuration file (see sec. 5.3.3) and writes it to global variables. Also the so-called robot-queue is initialized. That is a queue containing instances of the above introduced robot-cluster, where each instance defines one special robot. Because of their triviality, the sub-VIs of this VI are not further described.

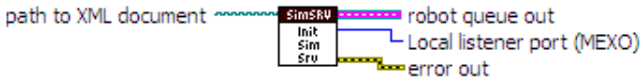

Figure A.2: Icon of the SimServer Init-VI

A.4 MEXO Communication-VI

encapsulates all sub-VIs needed for communication with the MEXO-clients and the USARSim-part of this middle-ware.

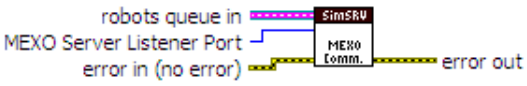

Figure A.3: Icon of the MEXO Communication-VI

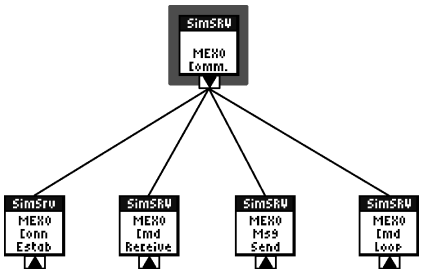

Figure A.4: VI-hierarchy of the MEXO Communication-VI

A.5 MEXO Connector-VI

listens for connection attempts (which could e.g. be from MEXOs) at the specified listener port. If a connection can be established, this VI listens for the POWER command at all open connections. When such a command is received, it is parsed and immediately executed, i.e. the SimServer-CreateRobot-VI is called. Answers with an ACK or an ERR message (via the TCP-connection), depending on the success of the execution. Fig. A.5 shows a flowchart that describes the algorithm implemented in this VI.

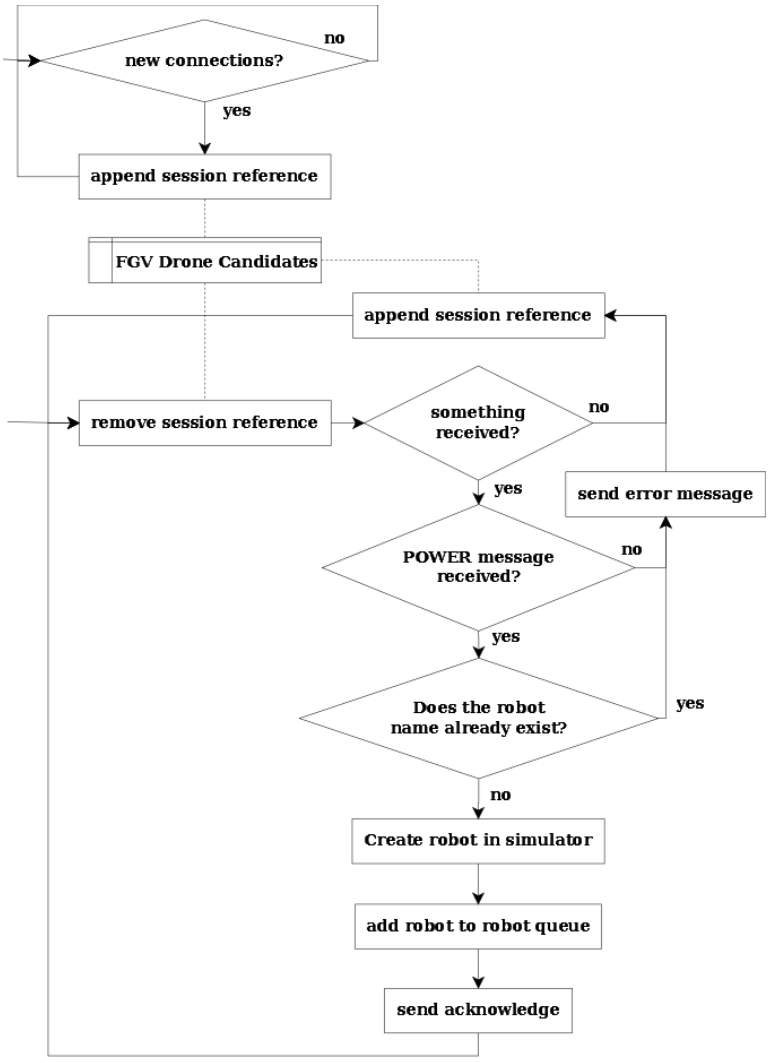

Figure A.5: Flowchart of MEXO connection and UAV creation as it is implemented in the MEXO Connector-VI

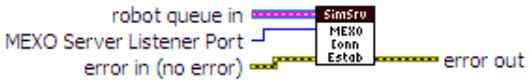

Figure A.6: Icon of the MEXO Connector-VI

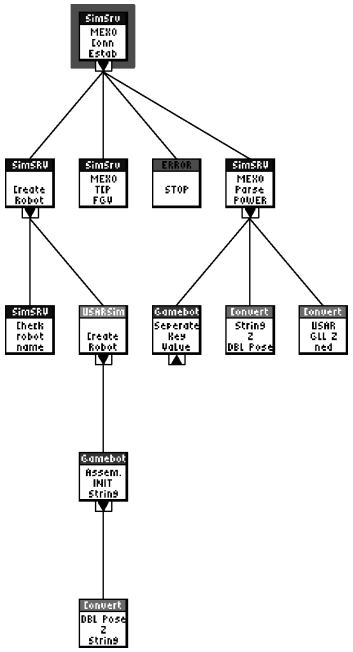

Figure A.7: VI-hierarchy of the MEXO Connector-VI

A.5.1 MEXO POWER Command Parser-VI

parses the MEXO POWER Command (see sec. B) to a start configuration, that can be used to create a UAV in USARSim. Therefor, at first the input string is separated into key-value pairs using the Gamebots Separate Key-Value Pairs-VI (also the MEXO protocol follows the Gamebots syntax). Then, the values are converted to the proper datatype and stored in a cluster. The StartingPose parameter must be transformed to USARSim map coordinates, which is done using the Convert USARGLL 2 ned-VI, which will be introduced later.

Figure A.8: Icon of the MEXO POWER Command Parser-VI

A.5.2 Gamebots Separate Key-Value Pairs-VI

separates key-value pairs in a string that is formatted according to the Gamebots protocol. Thereunto, regular expressions are used: Firstly, the input is matched against `(\}\s+\{)|([\{\}])` to retrieve the separated key-value pairs. Secondly, the pairs are matched against a space to separate the key from the value(s).

Figure A.9: Icon of the GAMEBOTS Separate Key-Value Pairs-VI

A.5.3 Convert Pose String 2 Double-VI

reads the values from the string "values", where they are stored in the following format (example): `1.000,1.01,1.000009` These values (without commas) are written into the input array.

Example:

```
input: 0,0,0,0,0,0,0 offset: 1 values: "1.000,1.01,1.000009"
   output:
0,1,1.01,1.000009,0,0,0
```

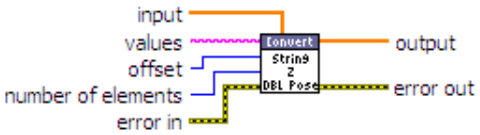

Figure A.10: Icon of the Convert Pose String 2 Double-VI

A.5.4 Convert USAR GLL 2 ned-VI

converts geographic latitude [deg], longitude [deg] and height [m] to local level coordinates north, east, down. This conversion is done using the flat-earth assumption from sec. 5.2.7.2 and reference GPS-coordinates. Hence, the converted values are equal to USARSim map-

coordinates ($n = x\ [m], e = y\ [m], d = z\ [m]$):

$$n = (\phi - \phi_{ref})111200 - n_{ref} \quad \text{(A.1)}$$

$$e = (\lambda - \lambda_{ref})71670 - e_{ref} \quad \text{(A.2)}$$

$$d = -h + h_{ref} - d_{ref} \quad \text{(A.3)}$$

The variables denoted with ref refer to reference values as they are configured in the simulation server configuration file (see sec. 5.3.3)

Figure A.11: Icon of the Convert USAR GLL 2 ned-VI

A.5.5 Error Stop-VI

is a global functional variable (FGV). It supports 3 modes: reset, read and write. If an error is wired at error in, the mode is completely ignored and stopped is set to true. Read is the default mode and simply reads the value of stopped. Write is used to write the FGV. This does only work, if the value linked to the stop control is true. Otherwise nothing happens. With this, overwriting a stop-value is prevented. Reset sets stopped to false. In this project, the Error Stop-VI is used to stop the execution of the whole simulation middleware . Thus, this sub-VI is found in a lot of other VIs.

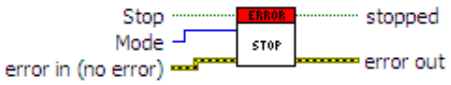

Figure A.12: Icon of the Error Stop-VI

A.5.6 FGV Drone Candidates-VI

stores TCP-session references, that have connected to the server but not yet sent a POWER-message. It supports the modes reset, add and remove (names are self-explanatory).

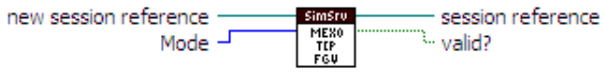

Figure A.13: Icon of the FGV drone candidates-VI

A.5.7 SimServer Create Robot-VI

creates a robot in the USARSim simulator (using the sub-VI that is introduced next), initializes a Robot-Cluster and adds it to the robots-queue. Checks if the selected name already exists and returns an error in this case.

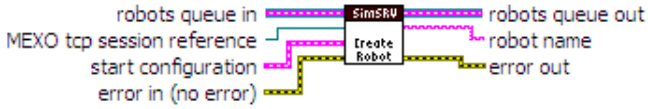

Figure A.14: Icon of the SimServer Create Robot-VI

A.5.8 USARSim Create Robot-VI

creates a robot in the USARSim simulator. Therefor, an INIT-command is assembled. Afterwards, the VI waits for a NFO-message from the USARSim server. If received, the INIT-command is sent to the USARSim server. Otherwise, an error is created.

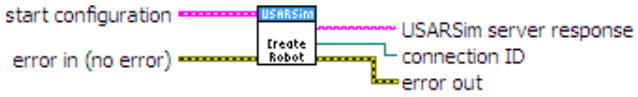

Figure A.15: Icon of the USARSim Create Robot-VI

A.5.9 Gamebots assemble INIT string-VI

creates a Gamebots INIT command by properly assembling and converting the values from the start configuration.

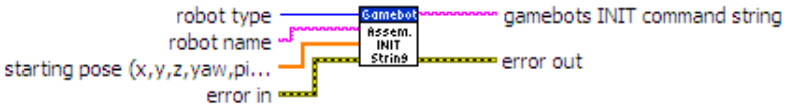

Figure A.16: Icon of the Gamebots assemble INIT string-VI

A.5.10 Convert Double Pose 2 String-VI

converts the pose data from the double array pose into a string, which is usable for the Gamebots Protocol. Example result:

```
{Location -18.0,-18.0,19.8} {Rotation -1.57,0.0,0.0}
```

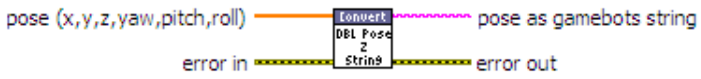

Figure A.17: Icon of the Convert Double Pose 2 String-VI

A.5.11 Check if Robot Name Exists-VI

returns an error if a robot with the same name as new robot name already exists.

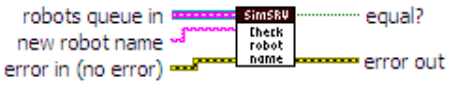

Figure A.18: Icon of the Check if Robot Name Exists-VI

A.6 MEXO Command Receiver Loop-VI

receives commands via the MEXO TCP connection, which are parsed and put onto the MEXO command queue of the appropriate robot. Answers with an ACK or ERR message on command retrieval. If the connection was closed by the MEXO, then the robot is removed. Implements the queued round robin design pattern and acts as a producer to the MEXO Command queue. A flowchart describing the algorithm implemented in this VI is shown in fig. A.20.

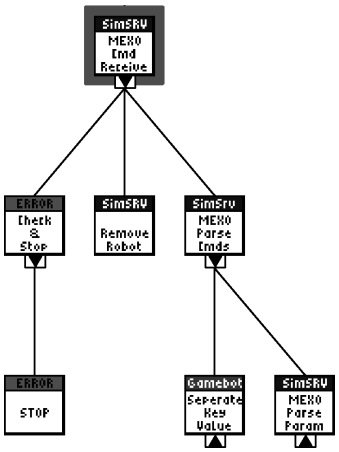

Figure A.19: VI-hierarchy of the MEXO Command Receiver Loop-VI

A.6.1 Merge Errors Check and Stop-VI

is used to stop on errors. "first error" is that error, which occurred first (in a dataflow sense). If e.g. "first error"'s status is true, then "should stop" gets true and "first error" is mapped to "error out" ("second error" is not taken into account at all in this case). This procedure is faster than the standard MergeErrors-VI. Furthermore, this VI reads the above introduced Error Stop-FGV and writes it, so all readers of the FGV know if they should stop. As the SimServer is a server and should not stop or create an error due to connection problems at client side, the errors shown in tbl. A.1 are deleted from the first error port.

A.6.2 SimServer Remove Robot-VI

removes a robot from the simulation and sends a DEL message to the MEXO. Destroys all information that belongs to the robot and closes the TCP connections (to the MEXO

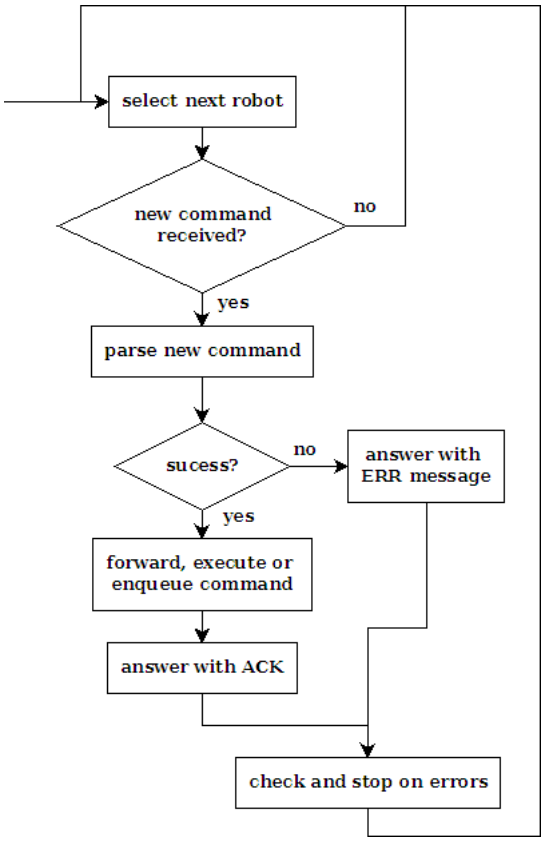

Figure A.20: Flowchart of the MEXO Command Receiver Loop-VI

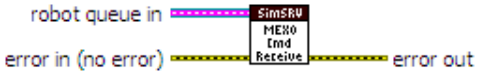

Figure A.21: Icon of the MEXO Command Receiver Loop-VI

Error	Description
1	an input parameter is invalid.
57	connection busy.
62	network connection to be aborted.
1122	Refnum became invalid while node waited for it.

Table A.1: Errors that are cleared by Merge Errors Check and Stop-VI

and to USARSim). The robot is removed from the USARSim world.

Figure A.22: Icon of the Merge Errors Check and Stop-VI

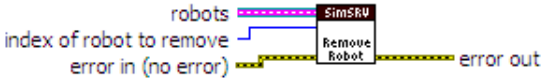

Figure A.23: Icon of the SimServer Remove Robot-VI

A.6.3 MEXO Command Parser-VI

Parses commands received via a MEXO connection. Decides, whether a command must be queued. Depending on the command type, a short check is done if enough parameters were specified. The result of this check is provided at the success indicator. Main part of the parser is a select structure, wherein all commands are treated separately. Key-value pair separation is again done with the above introduced Gamebots Separate Key-Value Pairs-VI. If command parameters need to be parsed as well, they are passed over to the SubVI which is desribed next. If a command is not listed in the case-structure, it is forwarded to USARSim using the TCP-session reference from the robot-cluster immediately. This is especially useful for administrative commands like PAUSE or TRACE (see sec. B).

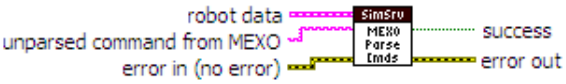

Figure A.24: Icon of the MEXO Command Parser-VI

A.6.4 MEXO Parse Command Parameters-VI

parses the parameters specified at a MEXO command. Does the transformation to USARSim map coordinates (using Convert USAR GLL 2 ned-VI). The “values specified” port is used to ensure that enough parameters were delivered with the command.

Figure A.25: Icon of the MEXO Parse Command Parameters-VI

A.7 MEXO Message Send Loop-VI

sends the messages from the MEXO Message queue of each robot to it's MEXO via the appropriate TCP connection. Implements the queued round robin design pattern and acts as a consumer to the MEXO Message queue. Fig. A.27 shows a flowchart explaining the simple algorithm implemented in this VI.

Figure A.26: Icon of the MEXO Message Send Loop-VI

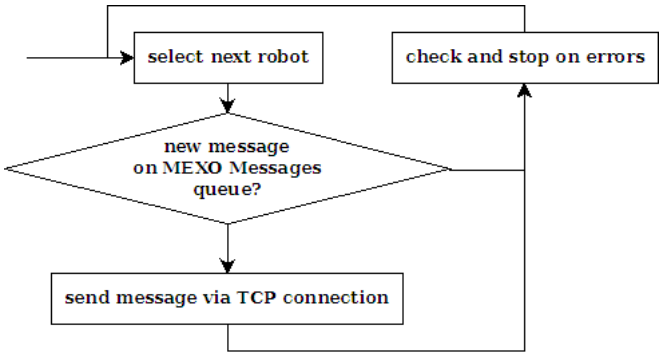

Figure A.27: Flowchart of the MEXO Message Send Loop-VI

A.8 Process Command Loop-VI

dequeues commands from the MEXO Command queue of each robot. Calculates if timestamps have passed and delivers this information to a subVI that processes the commands (Process Command-VI). Removes a robot if it's shutdown signal is present (e.g. from a SHUTDOWN-MEXO-command).Fig. A.30 shows a flowchart of the algorithm implemented in this VI. Implements the queued round robin design pattern and acts as consumer to the MEXO Commands queue.

Figure A.28: Icon of the Process Command Loop-VI

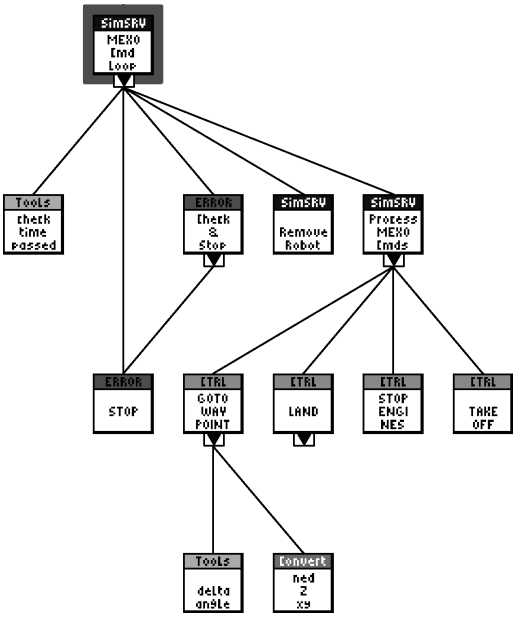

Figure A.29: VI-hierarchy of the Process Command Loop-VI

A.8.1 Check Time Passed-VI

checks if a time stamp has already passed by comparison with a reference time and a LabVIEW Tick-Count VI. This VI is capable of dealing with overflows in the LabVIEW tick-count.

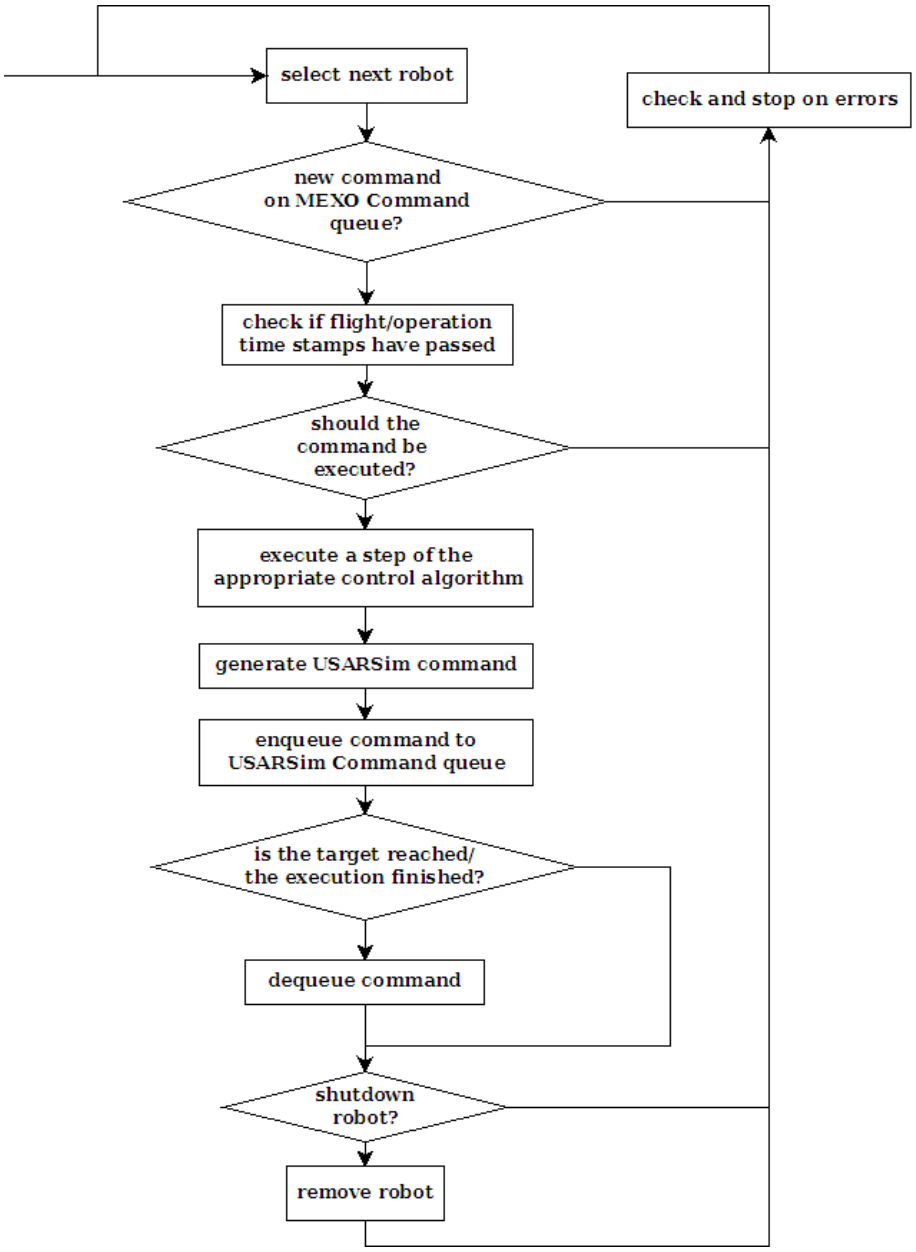

Figure A.30: Flowchart of the Process Command Loop-VI

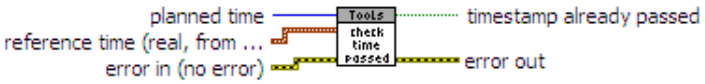

Figure A.31: Icon of the Check Time Passed-VI

A.8.2 Process Command-VI

dequeues the state cluster from the single element state queue to get uptodate state information. Heart of this VI is a case structure, that calls different controllers based on the type of MEXO command. After a step of the appropriate control algorithm (subVI) is executed, the state cluster is enqueued and the result of the control algorithm (which is typically an USARSim command) is enqueued on the USARSim command queue.

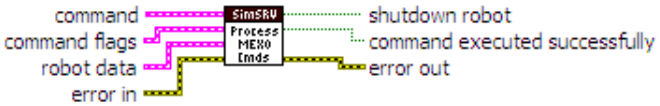

Figure A.32: Icon of the Process Command-VI

A.8.3 Go to Waypoint Controller-VI

implements a P-controller that leads the drone to a given way-point. When that way-point is reached, the heading of the drone is corrected (again with proportional control). r is the reference variable and consists of local level frame coordinates and the heading in degree (north,east,down,heading). PoseMeasurementCluster must be a current pose measurement (e.g. from the single element state queue). u are the correcting variables, which are velocities in the body fixed frame (vx,vy,vz,vyaw as needed for USARSim). Gains of the controller can be adjusted on the front panel. A flowchart describing the control algorithm implemented in this VI is shown in fig. A.34.

Comments to the controller structure: As the correcting variables are velocities and the outputs are positions, the plant contains an integrator. Thus, stationary accuracy is achieved assuming linearity of the plant. If one remembers the equations of motion of the UAV (3.15), he might argue that linearity is not given in these equations. But the model used in USARSim (see sec. 5.2.6) together with the motion dynamics derived by KARMA are different.

Assume the left-hand side of eq. 3.15 is the same as modeled in KARMA (except drag, which is modeled in KARMA indeed). The USARSim part of the UAV model ignores pitch and roll displacement in the translational dynamics at all. Only yaw is taken into account to adjust heading of the force acting on the center of mass. But in this control algorithm, yaw is not influenced during translational correction and vice-versa. Hence, the right-hand side of the equation of translational motion (generalized forces) is linear, namely (see also sec. 5.2.6)

$$\begin{bmatrix} 5.05 \cdot v_{Linear} \\ 5.05 \cdot v_{Lateral} \\ 5.05 \cdot v_{Altitude} + 49.02 \end{bmatrix}$$

KARMA models viscous drag only, so the model is linear.

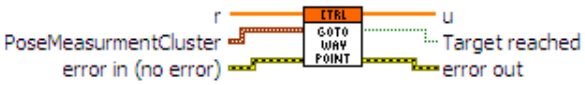

Figure A.33: Icon of the Go to Way-point Controller-VI

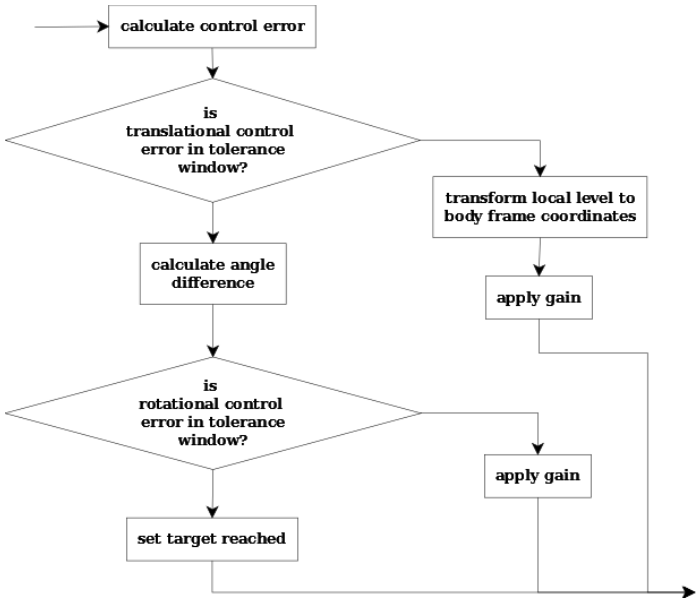

Figure A.34: Flowchart of the Go to Way-point Controller-VI

A.8.4 Angle Difference-VI

Calculates the difference between desired and current angle, so that the difference is in $[-\pi, \pi]$.

Figure A.35: Icon of the Angle Difference-VI

A.8.5 ned 2 xy-VI

converts a local level frame horizontal pose (n, e, ψ) to body frame horizontal coordinates (x, y):

$$x = cos(\psi)n + sin(\psi)e \tag{A.4}$$

$$y = -sin(\psi)n + cos(\psi)e \tag{A.5}$$

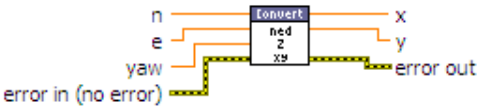

Figure A.36: Icon of the ned 2 xy-VI

A.8.6 Land Controller-VI

reads the current z value from the robot status and remembers 10 of these values. If 10 measurements are available and the (first order backward) derivative of these 10 measurements (z-velocity) is zero, then it is assumed that the drone has landed. Otherwise, "downward velocity" is sent as actuating command in z direction in u.

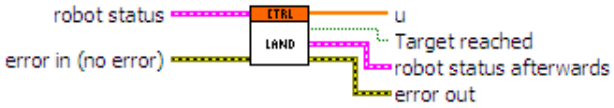

Figure A.37: Icon of the Land Controller-VI

A.8.7 Stop Engines Controller-VI

stops the engine of a UAV by sending a special USARSim command, that sets the angular rate of the rotors to zero.

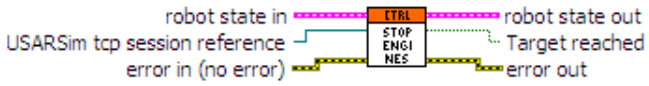

Figure A.38: Icon of the Stop Engines Controller-VI

A.8.8 Takeoff Controller-VI

implements a P-controller which leads a UAV to a specific height.

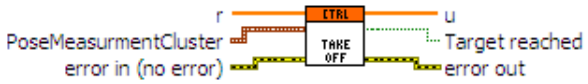

Figure A.39: Icon of the Takeoff Controller-VI

A.9 USARSim Server Communication-VI

receives messages from the USARSim server, parses them and updates the single element state queue. Also sends commands from the USARSim command queue to the USARSim server.

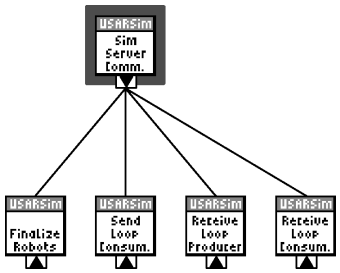

Figure A.40: VI-Hierarchy of the USARSim Server Communication-VI

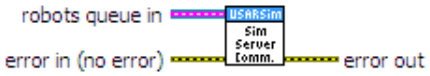

Figure A.41: Icon of the USARSim Server Communication-VI

A.10 USARSim Finalize Robots-VI

removes all robots from the robot-queue, closes their TCP-connections to USARSim (so that the robots are removed from USARSim as well) and releases all related queues.

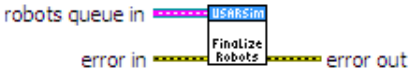

Figure A.42: Icon of the USARSim Finalize Robots-VI

A.11 USARSim Send Server Command Loop-VI

removes commands from each robot's USARSim Command queue and sends them to USARSim. Implements the queued round robin design pattern and acts as a consumer to the MEXO Command queue. Fig. A.45 shows a flowchart of the very simple algorithm implemented in this VI.

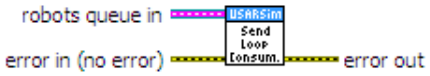

Figure A.43: Icon of the USARSim Send Server Command Loop-VI

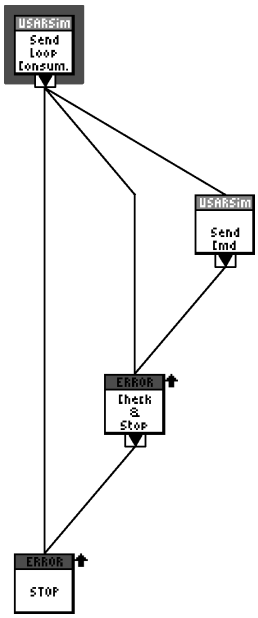

Figure A.44: VI-Hierarchy of the USARSim Send Server Command Loop-VI

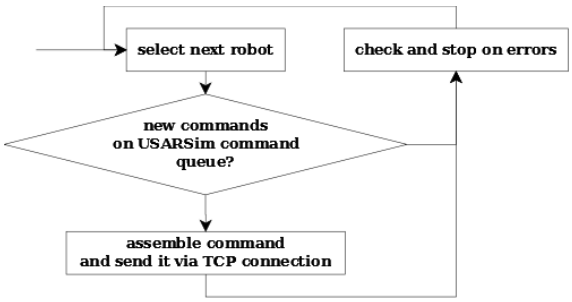

Figure A.45: Flowchart of the USARSim Send Server Command Loop-VI

A.11.1 USARSim Send Command-VI

assembles a DRIVE command and sends it to USARSim via the proper TCP session.

Figure A.46: Icon of the USARSim Send Command-VI

A.12 USARSim Server Message Receiver Loop-VI

retrieves new messages from the USARSim server and pushes it onto the unparsed messages queue. Furthermore this VI is responsible for detecting overflow in the tick counter. Implements the queued round robin design pattern and acts as a producer to the unparsed messages queue.

Figure A.47: Icon of the USARSim Server Message Receiver Loop-VI

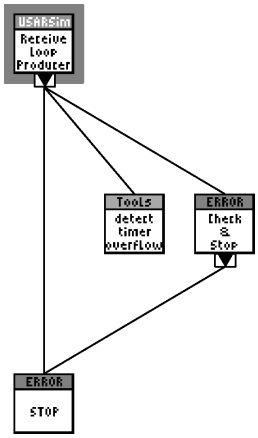

Figure A.48: VI-Hierarchy of the USARSim Server Message Receiver Loop-VI

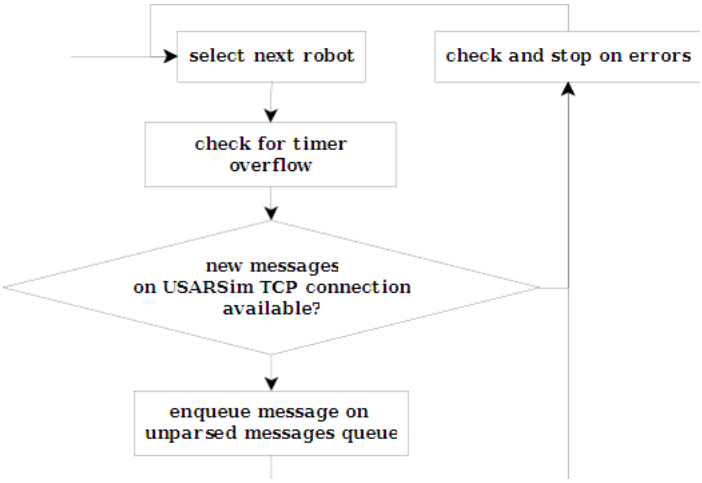

Figure A.49: Flowchart of the USARSim Server Message Receiver Loop-VI

A.12.1 Detect Timer Overflow-VI

checks the LabVIEW Tick Counter-VI for overflows using an FGV. If an overflow is detected, a global variable is incremented.

Figure A.50: Icon of the Detect Timer Overflow-VI

A.13 USARSim Server Message Parser Loop-VI

dequeues elements from the unparsed messages queue. Parses these messages and, if necessary enqueues them onto the MEXO message queue and updates the single element state queue. Removes a robot if a DIE message was received from USARSim.

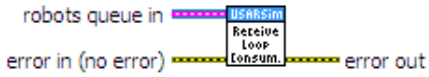

Figure A.51: Icon of the USARSim Server Message Parser Loop-VI

A.13.1 USARSim Parse Message-VI

parses messages received from the USARSim server. Updates the single element state queue. Generates and enqueues state messages for the MEXO.

As first step in the process of parsing the Gamebots Gamebots Separate Key-Value Pairs-VI is called. Then, a separation into 4 different Types of messages is done using a case structure:

- SEN messages: In the configuration file for the simulation middleware a position sensor must be chosen (see sec. 5.3.3). This selection defines, whether a SEN message (which is a USARSim sensor reading) is used to update the state element and generate a MEXO message. If e.g. SEN.GT is chosen as sensor type in the configuration file, then all messages retrieved from the USARSim ground truth sensor are used to update the robot state (which is used by the controllers). Furthermore, at each retrieval a message for the MEXO is generated and enqueued onto the MEXO messages queue. Of course, only valid (flag) measurements are used. The

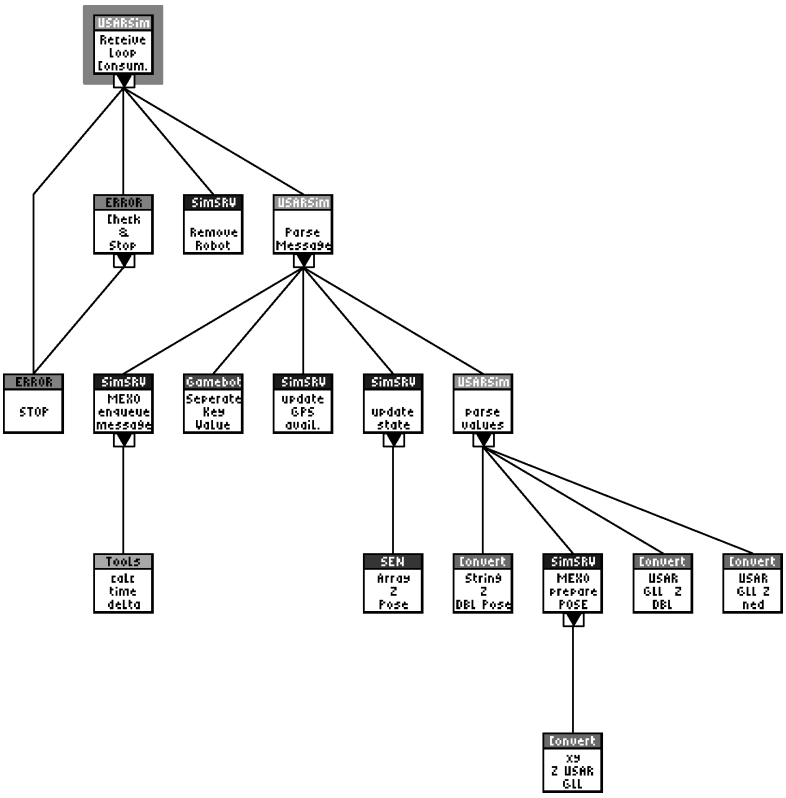

Figure A.52: VI-Hierarchy of the USARSim Server Message Parser Loop-VI

only exception is the GPS sensor message: it's availability state (GPSfix) is always checked and updated.

- DIE messages: If retrieved, a DEL message for the MEXO is generated.
- STA messages: The battery value is extracted from the message and used to update the state element. A STAT message for the MEXO is generated and enqueued.
- Other messages (default) are ignored. A message cluster of the type "Other" is returned and only its time stamp is set.

A.13.2 MEXO Enqueue Message-VI

adds flight- and operation-time stamps to a prepared STAT or POSE message and enqueues it onto the MEXO messages queue.

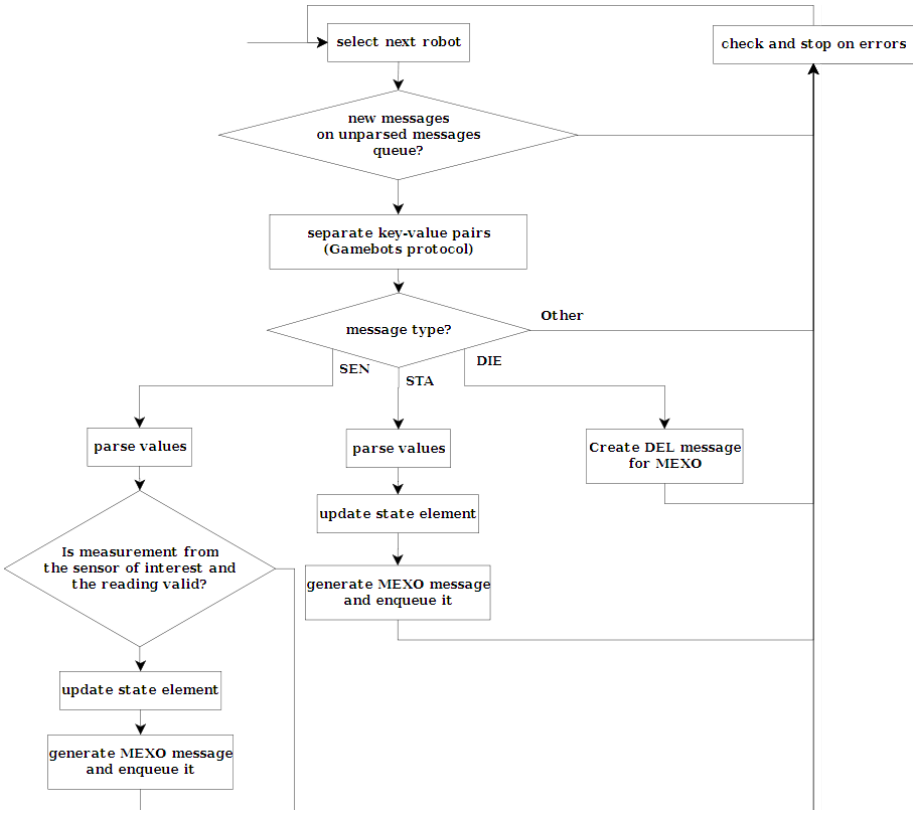

Figure A.53: Flowchart of the USARSim Server Message Parser Loop-VI

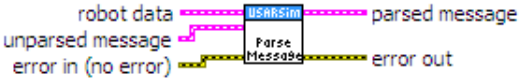

Figure A.54: Icon of the USARSim Server Message Parser-VI

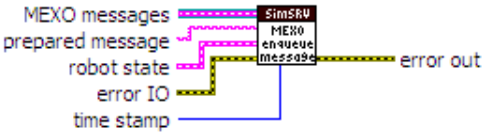

Figure A.55: Icon of the MEXO Enqueue Message-VI

A.13.3 Calculate Time Difference-VI

Calculates the difference (time measured - start time). Takes tick count overflows into account.

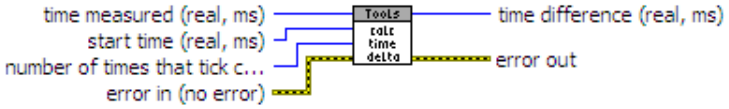

Figure A.56: Icon of the Calculate Time Difference-VI

A.13.4 Update GPS Available-VI

adds the GPSfix Parameter to a MEXO message and updates the single element state queue. An extra VI was necessary because of the following reasons:

- If, according to simulation middleware configuration another sensor than GPS is used for state update, then the Update Robot State-VI would not be called on GPS sensor messages.
- Independently from the selected state update-sensor, an information about the state of GPS positioning should be reported to the MEXOs (see sec. B).

Thus, this VI is needed to add GPS state information to MEXO pose messages.

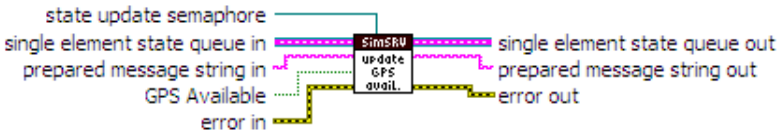

Figure A.57: Icon of the Update GPS Available-VI

A.13.5 Update Robot State-VI

updates the state element from the single element queue with the information provided. Supports two modes: One is the pose-update mode which updates pose information only. The other mode can be used to update the battery state.

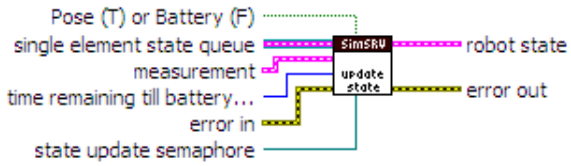

Figure A.58: Icon of the Update Robot State-VI

A.13.6 Message Array 2 Pose-VI

converts pose values, that are stored in an array to a pose cluster.

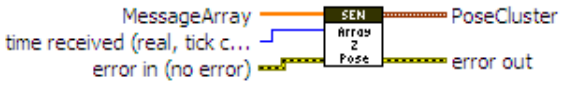

Figure A.59: Icon of the Message Array 2 Pose-VI

A.13.7 USARSim Parse Values-VI

parses the content of a message from the USARSim server (Gamebots protocol) to bring it to a desired message format, which depends on the messages type. Also MEXO message strings are generated. The message content as cluster-input is an array of key-value pairs. The following message types are supported:

- SEN.GPS: A measurement from the USARSim GPS sensor can contain the following keys: time, latitude, longitude, altitude and satellites. All of them are written into the array of the Message Cluster in exactly this order. Latitude and longitude are transformed to local level coordinates for further calculation. The MEXO message is generated directly from latitude and longitude
- SEN.INS: Contains the keys: time, location and orientation. Location and orientation consist of 3 values each (which are local level coordinates and Euler-angles).
- SEN.GT: Same as in the SEN.INS case

The time value is that one delivered from USARSim relative to the "game"-start.

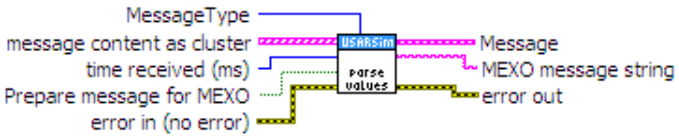

Figure A.60: Icon of the USARSim Parse Values-VI

A.13.8 MEXO Prepare POSE Message-VI

uses a pose array which is given as (time, north, east, down, roll, pitch, yaw) or (latitude, longitude, altitude, number of satellites) in the GPS case to prepare a POSE message

for the MEXO. In this message, all the coordinates should be transformed to latitude, longitude and altitude.

Figure A.61: Icon of the MEXO Prepare POSE Message-VI

A.13.9 xy 2 USAR GLL-VI

does the above mention transformation from local level frame coordinates (or USARSim map coordinates) to geographic longitude and latitude. It is the inverse transformation to that one done in the Convert USAR GLL 2 ned-VI.

Figure A.62: Icon of the xy 2 USAR GLL-VI

A.13.10 USAR GLL 2 Double-VI

reads the geographic longitude or latitude values from the string values, where they are stored in the following form (example): `40,30.12,N` The first number represents the degrees, the second stands for minutes while the third specifies a geographic direction. These values are converted to double-precision floating point in degree.

Figure A.63: Icon of the USAR GLL 2 Double-VI

Appendix B

MEXO protocol definition

This protocol is used for communication between a single Mission Executors and the simulator middleware. The design was sketched by researchers at Lakeside Labs. It consists of commands and messages, where commands are sent from the Mission Executors to the simulation middleware and messages follow the opposite direction. To all commands exists a response, which could either be an acknowledge or an error message. An `ACK` does not mean, that a command was executed successfully, but that it could be parsed by the simulation middleware . The syntax of this protocol is quite similar to the Gamebots protocol. If a command is not known by the simulation middleware, it is forwarded to USARSim. Thus all USARSim commands can be sent from the Mission Executors directly to USARSim. That makes sense especially for some commands that influence the graphical presentation of the simulation. As an example, the `TRACE` command is shown here:

```
TRACE {On true} {Interval 0.5} {Color 0}
```

This command, sent by the Mission Executors , is forwarded to USARSim directly and causes the drone to leave red dots in the world every 0.5 seconds. Another useful example is the `PAUSE` command:

```
PAUSE Delay 10
```

causes the simulation to be paused (for all robots), 10 seconds after USARSim retrieves this command. To continue simulation, the `PAUSE` command has to be sent with a negative delay.

A detailed description of all commands that can be sent to USARSim is given in [WB].

All received commands are processed in order of retrieval, except for USARSim commands, `STARTENGINE` and `CLEAR`, which are executed immediately.

B.1 Command Formatting

Commands are sent to the LabVIEW Simulation via TCP socket connections from the Mission Executors , whereby the commands are formatted in following format:

RequestLine := `COMMAND([␣]{ParameterKey[␣]Value(,Value)*})*[\n][\r]`

The `COMMAND` is uppercase by convention, while the parameter key and value are not case sensitive. All commands have to be placed in one line ending with carriage return and line feed symbols. The LabVIEW server will answer on every line with an acknowledge or a short error description.

ResponseLine := `(ACK[\n][\r]|(ERR[␣][0-9]*[␣]ErrorMessage[\n][\r])`

```
COMMAND := POWER|STARTENGINE|TAKEOFF|GOTOWAYPOINT|STOPENGINE
|SHUTDOWN|LAND|HOLD|TAKEPICTURE|FORMATION
ParameterKey := [A-Z][A-Z,a-z]*
Value := [0-9,a-z,A-Z][0-9,a-z,A-Z,␣,'.']*
ErrorMessage := [A-Z][0-9,a-z,A-Z,␣]*
```

B.2 List and Description of Commands

B.2.1 Create a Drone in the Simulator

`POWER {RobotName` *Name*`}`
`{StartingPose` *longitude,latitude,altitude,yaw,pitch,roll*`}`
`{USARSimServer` *address*`} {USARSimPort` *port*`}`

The LabVIEW simulation middleware listens for clients at a specified port. If someone (e.g. Mission Executors) wants to create a drone, it has to connect to this port and send the `POWER`-message via the established session. If the command was successfully received, parsed and a drone was created in USARSim, the server sends an `ACK`, otherwise, an error message will be the response. If you create a drone in the simulation, and the starting pose was valid but somehow useless (e.g. the drone falls to the ground and breaks), then the server will however first send an `ACK`. After the drone broke, simulation middleware will send something similar to (see the associated message description below)

`DEL {RobotName` *Name*`}`

At the execution of the `POWER` command, operation time (`Optime`) recording is started.

The simulation of the GPS synchronization and searching for satellites is initiated.

Parameter `RobotName` defines the name of the drone, which is used in USARSim and the simulation middleware. As the process of drone creation is initiated by the establishment of a TCP session, the unique identifier of each drone for the server is this TCP session reference. However, you will receive an error message if you give two drones the same name.

Parameter `StartingPose` In USARSim, the drone is placed immediately after the simulation middleware receives the `POWER` message. Therefore, in order to specify the spawning pose of the drone in the map, the starting pose has to be delivered with this command in the simulation case. The position is defined in LLA format while the angles are given in degree. Notice that for the simulation, some issues have to be considered before the use of georeferenced coordinates is possible:

- A ReferenceGPSCoordinate object must be placed in the USARSim map [WB].
- The same reference point has to be specified in the simulation middleware configuration file (used by LabVIEW).
- The starting pose must be valid (e.g. not below the ground or outside of the map).
- A GPS Sensor must be placed on the robot [WB].
- This GPS Sensor must be configured properly in cDrones.ini (which is the cDrones configuration file for USARSim).

Parameters `USARSimServer` **and** `USARSimPort` **(optional)** `USARSimServer` and `USARSimPort` are address and port of the USARSim Server. The address can be in IP dot notation or a hostname. Thus, it is possible to use multiple USARSim servers from one simulation middleware . However, the influence of one drone to another is not taken into account when running on different USARSim servers.

Default Values If you do not specify a parameter, its default value is used. Notice that the default values will be invalid in many cases, as e.g. the starting pose could be senseless in your map. The default values are:

- `RobotName: Drone1`

- `StartingPose: 0,0,0,0,0,0`
- `USARSimServer: localhost`
- `USARSimPort: 3000`

B.2.2 Start the Engines

`STARTENGINE`

starts the execution of previously received commands and/or enables the drone to execute commands that are to be received. All TimeStamps given in other commands are relative to the execution time of this command (in *milliseconds*). They define the *earliest execution time allowed* for the specific command. The flight time is reset to zero and its recording is started. If you send `STARTENGINE` although the engines are already started the command is ignored.

B.2.3 Stop the Engines

`STOPENGINE` {`TimeStamp` *`relative_time`*}

stops the engines of the drone. As all commands except for `STARTENGINE` are processed in order, this command cannot be used to stop the engines during the execution of a plan. If the previous command was successfully executed, the engines are stopped and the robot drops to the ground. Recording of the flight time is stopped.

Parameter `TimeStamp` (optional) specifies the earliest execution allowed for this command in milliseconds (default: 0) relative to the last executed `STARTENGINE` command.

B.2.4 Stop simulation of a drone

`SHUTDOWN` {`TimeStamp` *`relative_time`*}

stops simulation of the drone. It is removed from USARSim and all its data is deleted from simulation middleware , thus also the TCP session is closed. You will receive a `DEL` message. Note that, similar to `STOPENGINE`, this command cannot be used to stop the simulation during the execution of a plan.

Parameter `TimeStamp` (optional) same as previous command

B.2.5 Stop immediately and cancel commands

`CLEAR`

pauses the simulation and deletes all commands of a drone. To continue, send new commands and invoke the `PAUSE` command with a negative delay (see above). If you don't send new commands, the drone continues to execute the last (USARSim) control command it received from the simulation middleware .

B.2.6 Take Off From Ground

`TAKEOFF` {`TimeStamp` *relative_time*} {`DestinationHeight` *height*}

commands the drone to lift off, until a special height is reached. A P-controller leads the drone to the destination height. The command is assumed to be successfully executed if a height measurement within a predefined tolerance window is received. The sensor used for this measurement is defined in simulation middleware 's XML configuration file. Gain and tolerance of the P-controller are set in a LabVIEW-VI named "TAKEOFFController.vi" (that implements the controller).

Parameter `TimeStamp` (optional) same as previous command

Parameter `DestinationHeight` specifies the destination altitude to be reached at take-off in *meters* relative to the sea level. Hence, this value is depending on the reference coordinates (q.v. `POWER` command).

B.2.7 Touch Down Safe

`LAND` {`TimeStamp` *relative_time*}

Initiates an automatic landing procedure. This landing procedure works as follows: A predefined downward velocity leads the drone down, while this downward velocity is also calculated from sensor measurements. If the measured downward velocity is zero for some time, although the regulating velocity is different from that, it is assumed that the drone has landed. Downward velocity and zero-duration are set via LabVIEW-VI "LANDController". The sensor type is again defined in simulation middleware 's XML configuration file.

Parameter `TimeStamp` (optional) same as previous command

B.2.8 Visit a Defined WayPoint

`GOTOWAYPOINT` {`TimeStamp` *relative_time*}
{`Pose` *latitude,longitude,altitude,heading*}

leads the drone to a desired waypoint and corrects its heading. A P-controller first regulates the drones position. After one position measurement is in a defined tolerance window, the correction of the heading is initiated. Therefore, once more a P-controller is used. The gains of the controllers, as well as the tolerances are set via the "GOTOWAYPOINTCcontroller.vi" LabVIEW-VI.

Parameter `Pose` Using this Parameter you specify the target position of the drone. Remember, that the values for latitude, longitude and altitude depend on the GPS reference point on the one hand, and on the special geographic referencing model of USARSim [WB] on the other hand. Latitude and longitude need to be given in degree and the altitude in meters above sea level. The heading is an angle in a range from 0° to 360°, whereby 0° means north.

Parameter `TimeStamp` **(optional)** same as previous command

B.2.9 Holding a Position

`HOLD` {`TimeStamp` *relative_time*}

Holds the position until the given time stamp is reached. The idea behind this command is to compensate delays or fix the position before taking pictures. In the simulation, this command is simply ignored as the drone automatically holds its position after a successful execution of a command. The possibility to implement a special controller for this case in combination with a special sensor is given using the "HOLDController" VI.

Parameter TimeStamp (optional) same as previous command

B.2.10 Taking a Picture

`TAKEPICTURE` {`TimeStamp` *relative_time*} {`Tilt` *angle*}

A picture is taken at a certain time stamp with the camera property of a given tilt angle. A-priori the angle is set to 90° meaning vertical position. As simulation middleware is not able to deliver images at this stage of implementation, this command is not realized yet.

B.2.11 Execute Scenario

`EXECUTETASKS`

is a dummy command for the simulation middleware and has no effect to the simulation.

B.2.12 Formation Processing

`FORMATION`

is also not implemented, as that topic is not part of this work.

B.3 Message Formatting

Messages are sent from the simulation middleware to the Mission Executors via the same TCP session as commands, whereby messages are formatted in following format:

`MESSAGE([␣]{ParameterKey[␣]Value(,Value)*})*[\n][\r]`

The Mission Executors must not respond to such a message.

```
MESSAGE := STAT|POSE|DEL
ParameterKey := [A-Z][A-Z,a-z]*
Value := [0-9,a-z,A-Z][0-9,a-z,A-Z,␣,'.']*
```

B.4 List and Description of Messages

B.4.1 Status message

`STAT` {`Batt` *remaining_seconds_to_live*}
{`FlightTime` *milliseconds_since_last_startengines*}
{`OpTime` *milliseconds_since_power*}

Parameter `Batt` defines the time until the robot's battery is empty in seconds. When this value reaches zero, the robot is frozen in the simulation (Location and Orientation are held constant). The start value for this parameter is defined in the cDrones package configuration file of USARSim (`cDrones.ini`).

Parameter `FlightTime` is the time difference to the last execution of a `STARTENGINES` command in milliseconds. The engines must be started, otherwise this parameter is omitted or delivers an invalid value.

Parameter `OpTime` is the time difference to the creation of the drone (`POWER` command) in milliseconds.

B.4.2 Pose message

`POSE` {`Location` *latitude,longitude,altitude*} {`Orientation` *roll,pitch,yaw*} {`GPSfix` *GPS_available*} {`FlightTime` *milliseconds_since_last_startengines*} {`OpTime` *milliseconds_since_power*}

Parameter `Location` delivers latitude, longitude in degrees and altitude in meters. Note that these values are again depending on the configuration of the GPS reference point.

Parameter `Orientation` delivers the current attitude of the drone in roll, pitch and yaw in degrees. Definition and sequence of the rotations are conforming the aeronautic standards [Ste04].

Parameter `GPSfix` is 1, if the GPS sensor is able to determine a position and zero otherwise. To measure a position, the sensor needs line-of-sight connection to at least four satellites.

Parameter `FlightTime` gives the time difference between the last execution of a `STARTENGINES` command and the pose measurement delivered in this message. If the engines are not running, this parameter is omitted.

Parameter `OpTime` same as previous message.

B.4.3 Robot deleted message

`DEL` {`Robotname` *name*}

tells the Mission Executors that the robot has been deleted in the simulator.

B.5 Error Messages

The LabVIEW standard error codes are extended by these five error codes:

Error code	**message**	**description**	**source VI**
5001	Robot name already exists	Occurs when you try to add a robot, although a robot with the same name already exists. The robot is not created.	checkIfRobotNameExists.vi
5002	Didn't receive NFO message from USARSim server	Occurs when you try to add a robot. After an INIT command (the instantiation of a robot model in USARSim) the simulation middleware has to wait for the NFO message from the USARSim server. If this message is not received after a specific timeout, this error message is sent. The robot is not created.	USARSimCreateRobot.vi
5003	Not enough Parameters specified for MEXO command	Occurs, if you send a command to the simulation middleware and not enough parameters (see above) were specified. The command is not further processed.	MEXOCommandParser.vi
5004	Robot broke	Occurs, if e.g. at robot creation the starting position is invalid or the robot was physically damaged in USARSim.	USARSimParseMessage.vi
5005	Zero sample time	Occurs at velocity calculation.	Pose2Velocity.vi

Table B.1: Error codes of the simulation middleware at the Mission Executors interface

Appendix C

Description of Flight Experiments

In order to analyze the dynamics of the Microdrone, as well as for model verification and sensor analysis some experiments were executed by the Lakeside Labs. Using a program from another project within cDrones, the flight data recordings from the Microdrone were written into CSV-files, where they were further processed with LabVIEW. Here, a short description is given how these experiments looked like. All of them were executed twice, once with GPS and once with INS navigation only. The description here refers to the case with GPS. The principle of an experiment was kept the same in the INS case, but the values vary. To find appropriate data for system identification, where the linear model was used, it was necessary to excite the rotorcraft in a range where it behaves almost linear. Thus, it was necessary to avoid two commands at the same time (e.g. not rolling and yawing without having the drone in rest in between). As can be seen in the equations of motion, nonlinearities arise from couplings and high amplitudes. At analyzing the flight data it turned out, that the experiments with GPS fulfilled these conditions better than the INS ones (although that has nothing to do with the navigation mode, but more with the person running the experiments). The diagrams in the text always only show the interesting part of a whole signal.

Experiment 1 The Microdrone stood in rest on an arbitrary place and recorded measurements for ca. 10 minutes.

Experiment 2 Start-point and target were ca. $10[m]$ apart. The drone was steered with the remote control to fly to the target.

Experiment 3 The drone took off to an arbitrary height. Then, the position was kept for a notable time. Via the remote control, the Microdrone was rotated in yaw-direction about ca. 180[deg]. After that, the position was held again, before a landing maneuver was flown.

Experiment 4 The Microdrone took off again to approx. 7[m] height. Then, the position was held for some time, before a pitching motion was executed. After holding the position for some time again, a rolling motion was added. The drone was left at rest a few seconds before it was landed. This experiment was run twice with GPS and the results were called 4a and 4b, respectively.

Bibliography

[BC08] Benjamin Balaguer and Stefano Carpin. Where am i? a simulated gps sensor for outdoor robotic applications. In *SIMPAR 2008*, 2008.

[BDB01] A. R. S. Bramwell, George Done, and David Balmford. *Bramwell's Helicopter Dynamics*. Butterworth-Heinemann, 2001.

[Bek05] George A. Bekey. *Autonomous Robots*. MIT Press, 2005.

[BFL08] Daniel Beck, Alexander Ferrein, and Gerhard Lakemeyer. A simulation environment for middle-size robots with multi-level abstraction. In *Robocup 2007*. Springer, 2008.

[Blu07] Peter A. Blume. *The LabVIEW Style Book*. Pearson Education, 2007.

[BNS04] S. Bouabdallah, A. Noth, and R. Siegwart. Pid vs lq control techniques applied to an indoor micro quadrotor. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[Bou07] Samir Bouabdallah. *Design and Control of Quadrotors with Application to Autonomous Flying*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2007.

[Bre08] Tommaso Bresciani. Modeling, identification and control of a quadrotor helicopter. Master's thesis, Lund University, 2008.

[CLD05] Pedro Castillo, Rogelio Lozano, and Alejandro E. Dzul. *Modelling and Control of Mini-Flying Machines*. Springer, 2005.

[Fay01] G. Fay. Derivation of the aerodynamic forces for the mesicopter simulation, 2001.

[FB99] Jay A. Farrel and Matthew Barth. *The Global Positioning System & Inertial Navigation*. McGraw-Hill, 1999.

[GWA07] Mohinder S. Grewal, Lawrence Randolph Weill, and Angus P. Andrews. *Global positioning systems, inertial navigation, and integration*. Wiley, 2007.

[HHWT07] Gabriel Hoffmann, Haomiao Huang, Steven L. Waslander, and Claire J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Hilton Head, SC, August 2007. AIAA Paper Number 2007-6461.

[Hof04] Anton Hofer. *Computer Aided System Modeling and Simulation, Lecture Notes*. Institute of Automation and Control, TU Graz, 2004.

[Hof05] Anton Hofer. *Modeling and Simulation, Lecture Notes*. Institute of Automation and Control, TU Graz, 2005.

[HRW+04] Gabriel Hoffmann, Dev Gorur Rajnarayan, Steven L. Waslander, David Dostal, Jung Soon Jang, and Claire J. Tomlin. The stanford testbed of autonomous rotorcraft for multi-agent control (starmac). In *In the Proceedings of the 23rd Digital Avionics Systems Conference*, pages 12.E.4/1–10, Salt Lake City, UT, November 2004.

[HWWL03] Bernhard Hofmann-Wellenhof, Manfred Wieser, and Klaus Legat. *Navigation - Principles of Positioning and Guidance*. Springer, 2003.

[Kha02] Hassan K. Khalil. *Nonlinear Systems*. Prentice-Hall, 2002.

[KKP09] Jinhyun Kim, Min-Sung Kang, and Sangdeok Park. Accurate modeling and robust hovering control for a quad-rotor vtol aircraft. *Journal of Intelligent and Robotic Systems*, DOI: 10.1007/s10846-009-9369-z:Online First, 2009.

[Lab08] Lakeside Labs. Collaborative microdrones, 2008.

[Mat02] MathEngine. *MathEngine Karma User Guide*, 2002.

[McK04] Philipp McKerrow. Modelling the draganflyer four-rotor helicopter. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2004.

[Mei90] L. Meirovitch. *Dynamics and Control of Structures*. Wiley, New York, 1990.

[Mic07] Microdrones GmbH. *Description of MD_Downlink*, February 2007.

[Nat04] National Instruments. *LabVIEW System Identification Toolkit User Manual*, September 2004.

[PBS07] C. Pepper, S. Balakirsky, and C. Scrapper. Robot simulation physics validation, 2007.

[PMC06] P. Pounds, R. Mahony, and P. Corke. Modelling and control of a quadrotor robot. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2006.

[QSB+08] Markus Quaritsch, Emil Stojanovski, Christian Bettstetter, Gerhard Friedrich, Hermann Hellwagner, Bernhard Rinner, Michael Hofbaur, and Mubarak Shah. Collaborative microdrones: Applications and research challenges, 2008.

[SHV06] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modelling and Control*. Wiley, 2006.

[SMB07] C. Scrapper, R. Madhavan, and S. Balakirsky. Stable navigation solutions for robots in complex environments, 2007.

[Ste04] Robert F. Stengel. *Flight Dynamics*. Princeton University Press, 2004.

[Val07] Kimon P. Valavanis, editor. *Advances in Unmanned Aerial Vehicles*. Springer, 2007.

[Vik] Bjørnar Vik. Integrated satellite and inertial navigation systems.

[Wan] Jijun Wang. The scale between unreal world and karma physical world.

[Wat04] John Watkinson. *The Art of the Helicopter*. Elsevier Butterworth-Heinemann, 2004.

[WB] Jijun Wang and Stephen Balakirsky. *USARSim V3.1.3*. University of Pittsburgh and NIST and Carnegie Mellon School of Computer Science.

[WCA99] Jiangxin Wang, Susan Y. Chao, and Alice M. Agogino. Sensor noise model development of a longitudinal positioning system for avcs, 1999.

[WLHK05] Jijun Wang, Michael Lewis, Stephen Hughes, and Mary Koes. Validating usarsim for use in hri research. In *PROCEEDINGS of the HUMAN FACTORS AND ERGONOMICS SOCIETY 49th ANNUAL MEETING*, 2005.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)