JNavigator - An Autonomous Navigation System for the JAviator Quadrotor Helicopter

Magisterarbeit zur Erlangung des akademischen Grades
Diplom-Ingenieur der *Angewandten Informatik*

Angefertigt am
Institut für Computerwissenschaften und Systemanalyse
der Naturwissenschaftlichen Fakultät
der Paris-London-Universität Salzburg

Eingereicht von
DI(FH) Clemens D. Krainer

Eingereicht bei
Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph Kirsch

Salzburg, September 2009

Affidavit

Herewith I, Clemens D. Krainer, declare that I have written the present diploma thesis fully on my own and that I have not used any other sources apart from those given.

Clemens D. Krainer	9020112
	Matriculation Number

Details

Name:	DI(FH) Clemens D. Krainer
University:	Paris-Lodron-Universität Salzburg Naturwissenschaftliche Fakultät Institut für Computerwissenschaften und Systemanalyse
Course of Studies:	Angewandte Informatik
Title of the diploma thesis:	JNavigator - An Autonomous Navigation System for the JAviator Quadrotor Helicopter
Mentor at the university:	Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph Kirsch

Keywords

1. Keyword:	Autonomous Unmanned Aerial Vehicle
2. Keyword:	Flight Control System
3. Keyword:	Java

Abstract

This thesis describes an autopilot software, called JNavigator, suitable for the needs of the JAviator [6] helicopter. At start of autonomous flights, JNavigator divides set course trajectories, which are paths of waypoints, in segments of constant velocity or constant acceleration. In consideration of flight time, JNavigator determines the JAviator's nominal position and tries to stir the helicopter towards it by regarding its current position, which is provided by a GPS receiver.

This thesis focuses on flexibility and extensibility of the implemented Java [36] software. First, this thesis outlines the JAviator's hardware and software architecture. Second, it briefly describes satellite-based and close-range locations systems, as well as coordinate transformations. Third, this thesis explains four-rotor helicopter dynamics and describes PID controllers developed for stabilizing altitude, attitude, and position. Subsequently, it designs a waypoint-following algorithm, which allows limited acceleration, and it elaborates on JNavigator's software design and implementation. Thereafter, this thesis presents the results of experiments that evaluate controllers, autonomous flights, and JVM behavior by utilizing the JAviator simulator. The implemented software has been verified on standard PC hardware successfully. Finally, this thesis reflects the current implementation and outlines design strengths and weaknesses.

Table of contents

List of figures

List of tables

1

Introduction

This thesis focuses on controlling autonomous flights of an Unmanned Aerial Vehicle (UAV). It particularly focuses on the design and implementation of an autopilot software, called JNavigator, suitable for the needs of the JAviator [6] model helicopter.

While model helicopters are build lightweight with limitations in payload and programming capabilities, the JAviator is conspicuously different. First, a diameter of 1.3 m and the fully symmetrical frame design let it stand out from others. Second, the JAviator has an unladen weight of 2.2 kg and can transport a payload of up to 3.2 kg. Third, the JAviator provides powerful onboard computers, which allow programming in Java.

The JAviator software consists of three main components called Plant, Flight Control System (FCS), and Ground Control System (GCS). First, the Plant is responsible for communicating sensor data from the hardware to the FCS and motor data vice versa. Second, the FCS implements the actual control algorithms to fly the JAviator. Third, the GCS visualizes the reported vehicle state from the FCS and stipulates the target values for the FCS for roll, pitch, yaw, and altitude. Currently, both FCS and GCS allow only manually controlled flights.

At the time of writing this thesis, the JAviator Plant features an Inertial Measurement Unit (IMU), an ultrasonic altimeter, a barometer, a laser range finder, and a Global Positioning System (GPS) receiver.

1.1 Conceptual formulation

UAVs apply software in various ways on various levels to accomplish autonomous flights. This diploma thesis elaborates on the design and implementation of flexible and extensible autopilot software. The requirements for this software are that it

- is written in Java,
- allows flying autonomously along a given set course, and
- utilizes GPS.

The target platform for the functional demonstration of this thesis is the available JAviator simulator, called MockJAviator, and not the real JAviator.

Written in Java. Java is an object-oriented high-level programming language developed by Sun Microsystems [36]. It is designed to be simple, robust, and secure, which makes it suitable for just about any programming task. The most significant advantage of Java over other languages and environments is its platform independency at both the source and binary level.

Java's automatic Garbage Collection (GC) causes unbounded and unpredictable pauses to running applications. To overcome the indeterminacy of traditional GC, the JAviator platform employs the IBM WebSphere Real Time Java Virtual Machine (WSRT JVM) [27], which includes the Metronome Garbage Collector [26].

Because of the utilized IBM WSRT JVM, all software must be written in Java version 1.4.2 or lower. The only exceptions to this are calls to the underlying operating system to access serial interfaces and the like.

Flying autonomously along a given set course. The autopilot software must enhance or replace the FCS to enable the JAviator to follow a given set course autonomously, without being connected to the GCS. When either necessary or intended, the GCS must be able to take over manual control momentarily.

Utilization of GPS. For the determination of the JAviator's current position, a GPS receiver provides the required data. A possibly available Differential GPS (DGPS) or GPS Real-Time Kinematics (RTK) correction data stream must be forwarded to the GPS receiver.

Target platform. The implementation on the real system is not part of this thesis. Hence, to verify the implemented autopilot, a functional demonstration with the JAviator simulator is necessary. Experiments should rely on the JAviator simulator.

1.2 Main contributions

The main contributions of this thesis are the design, implementation, and evaluation of the JNavigator autopilot software, which addresses the following items:

- The design focus of JNavigator is flexibility and extensibility. JNavigator not only covers the JAviator's requirements, but also easily adapts to other UAVs. A hierarchical configuration approach allows arbitrary reuse of components that are either included in JNavigator or provided externally by additional libraries using JNavigator's programming interfaces.
- JNavigator substitutes the existing FCS and controls both manual and autonomous flights. The GCS can interrupt an ongoing autonomous flight at any time.
- JNavigator verifies that the JAviator can fly a loaded set course and prepares an acceleration plan before starting the autonomous flight.
- For the determination of the JAviator's current position, JNavigator employs the National Marine Electronics Association (NMEA) 0183 [1] messages from a GPS receiver. For enhanced precision, JNavigator can forward an available Radio Technical Commission for Maritime Services (RTCM) SC-104 version 2.3 [2] correction data stream to the GPS receiver. The correction data stream may be

provided by a Networked Transport of RTCM via Internet Protocol (NTRIP) [4] caster, a Global System for Mobile Communications (GSM) service or a directly connected RTCM SC-104 receiver (Transmission Control Protocol / Internet Protocol (TCP/IP) socket, Bluetooth socket, or serial line).

- Apart from eight calls to the operating system to access serial lines and Bluetooth sockets, the JNavigator software is entirely written in the Java programming language. Calls to the operating system are nicely abstracted to allow access to TCP/IP sockets, Bluetooth sockets, and serial lines uniformly.
- A newly implemented extension to the JAviator simulator allows determining the current position via NMEA 0183 messages. This extension can also be applied to convert coordinates of arbitrary positioning systems to NMEA 0183 messages.

1.3 Thesis outline

Chapter 2 contains a short overview of related work on autonomous flight control systems for UAVs. It briefly describes controller implementation, flight path following, and special features of each system.

Chapter 3 outlines the status quo of the JAviator platform hardware and software by depicting the involved elements and their interaction.

Chapter 4 presents satellite-based and close-range location systems from a general point of view. It explains inaccuracies of satellite-based location systems and augmentation systems to lessen their impact. Furthermore, this chapter summarizes the methods of close-range location systems and focuses on coordinate transformation.

Chapter 5 describes a dynamic model for four-rotor helicopters and develops controllers for both manual and autonomous flight. It details the controller for autonomous flight with respect to the implemented waypoint-following algorithm.

Chapter 6 contains the design and implementation of the JNavigator software. After outlining the basic concepts, it details the implementation with respect to the functional model and the implemented software modules.

Chapter 7 explains the configuration for testing the JNavigator software and evaluates the implemented altitude, attitude, and position controllers. The experiments address controller time responses, accuracies, and limitations. Additionally, this chapter provides a detailed discussion of the measured results and focuses on autonomous flights. This chapter also studies the behavior of the FCS Java Virtual Machine (JVM) on available embedded systems and standard PC hardware.

Chapter 8 concludes by presenting an evaluation of the implemented software and proposals for future research directions in this field. In particular, it discusses the current implementation, as well as outlines design strengths and weaknesses.

2

Related work

This chapter gives an (incomplete) overview of the related work on autonomous flight control systems for UAVs.

Johnson and DeBitetto [28] present a guidance system, which generates position, heading, and velocity commands based on the current state of the UAV, as reported by the navigation system, and a waypoint list. The guidance system commands a straight line course between waypoints.

Hoffmann et al. [22, 23] developed a hierarchical hybrid control structure to govern a helicopter-based UAV in its mission to search for, investigate, and locate objects in an unknown environment. The hierarchical architecture consists of four layers, the strategic, tactical, and trajectory planners, as well as the regulation layer. The strategic planner coordinates missions with cooperating UAVs and creates a sequence of waypoints for the tactical planner. The tactical planner achieves landing, searching an area, approaching a waypoint, collision avoidance, and inspection of objects on the ground. Additionally, it overrules the strategic planner for safety reasons. The trajectory planner decomposes commanded behavior into a sequence of primitive maneuvers, guarantees safe and smooth transitions, and assigns an appropriate sequence of flight commands to the regulation layer to execute.

Kottmann [31] designed a trajectory generator, which converts flight maneuvers on an abstract level into waypoints that must be contained in the trajectory. In a further step of refinement the generator splits the routes between waypoints into several segments of constant velocity or constant acceleration. After this, the ground station uploads the segments to the helicopter onboard system for execution.

Lai et al. [20] present a hierarchical flight control system containing three tiers, the navigation, path, and stabilizing layer. The navigation manager controls the mission and commands a series of locations to the path controller. When the path controller reaches a location, it switches its destination to the next location. The stabilizing controller applies the nominal values from the path controller for both attitude and height to fly the helicopter.

Kim et al. [29, 30] designed a hierarchical flight control system, which allows a group of UAVs and Unmanned Ground Vehicles (UGVs) to cooperate. The flight management system employs three strategy planners to solve specific missions. The first strategy

planner implements a simple waypoint navigation of a predefined course. The second strategy planner operates a pursuit-evasion game where pursuers should *capture* evaders in a given grid field. The third strategy planner executes high-speed position tracking, where an UAV pursues a moving ground vehicle.

Williams [40] developed in his open source project Vicacopter two ways of autonomous control, waypoint following and programmable missions. For waypoint following the ground station uploads a sequence of waypoints to the helicopter. After the helicopter reaches a location, it heads for the next location. For programmable missions the ground station uploads a series of control commands to the helicopter for execution.

This chapter summarizes the related work on autonomous flight control systems for UAVs. Many of the authors employed a hierarchical controller structure to solve the task. Basic systems follow a list of either waypoints or flight commands. Advanced implementations allow cooperation of several vehicles and strategic planning of missions.

3

JAviator

This chapter presents an overview of the high-performance four-rotor model helicopter JAviator [6]. After outlining the structural elements of the JAviator system, it briefly describes the collaboration of the contained components and proceeds with a survey of the software architecture.

3.1 System overview

Figure 3.1 presents an overview of the complete system containing the JAviator, its ground station, and the emergency shutdown equipment. The ground station comprises a four-axis joystick, a Wireless Local Area Network (WLAN) router, and a laptop. The GCS, running on the laptop, receives control signals from the joystick and sends commands to the helicopter via the WLAN router. Additionally, the GCS visualizes and logs incoming information from the JAviator.

Onboard the JAviator, the WLAN module forwards messages from the GCS to the FCS in the Gumstix [21] and vice versa. The Robostix [21] routes sensor data from laser and altitude sensors to the FCS. After analyzing the positions of several GPS satellites, the GPS receiver delivers the current position to the Robostix, which forwards it to the FCS. Furthermore, the Robostix transmits revolution speed nominal values from the FCS to the four motor controllers MC_1, MC_2, MC_3, and MC_4. The IMU transmits its data directly to the FCS in the Gumstix. In case of an emergency, the operator transmits a shutdown message via the 868MHz sender to the corresponding receiver onboard the JAviator. The receiver forwards the shutdown message to the power board, which opens the high-current relays to disconnect the lithium-polymer battery from the motor controllers.

3.2 Software architecture

As shown in Figure 3.2, the JAviator [6] software consists of three main components called Plant, FCS, and GCS, which are connected by well defined interfaces. The Plant is responsible for communicating sensor data from the hardware to the FCS and

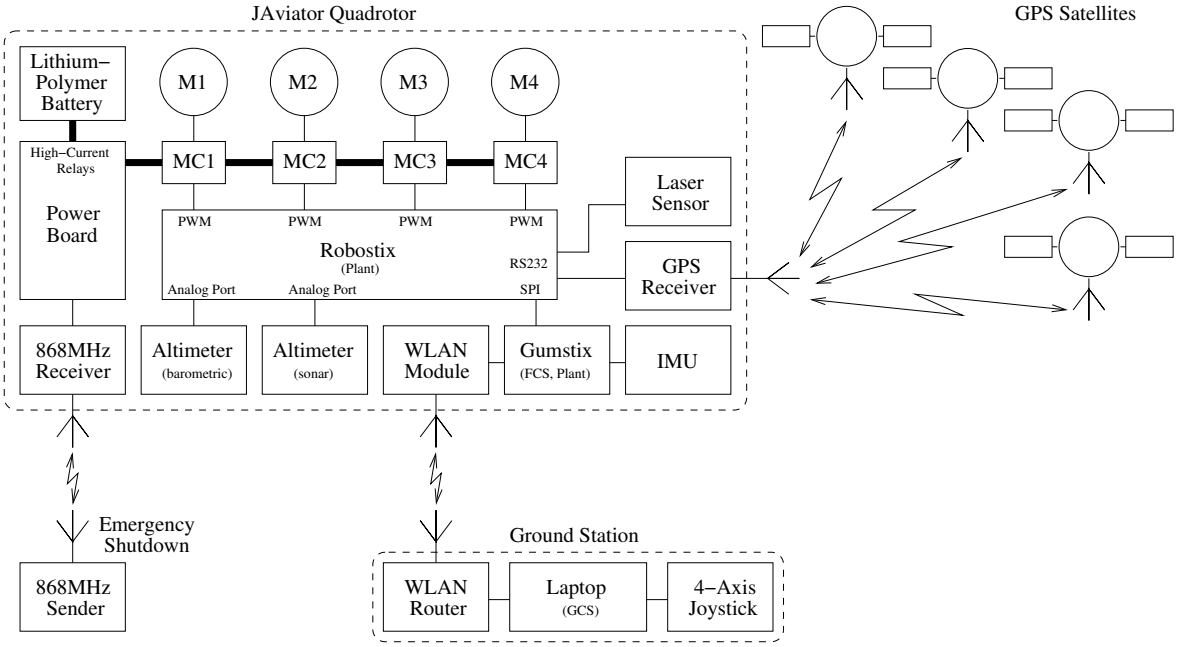

Figure 3.1: JAviator system overview.

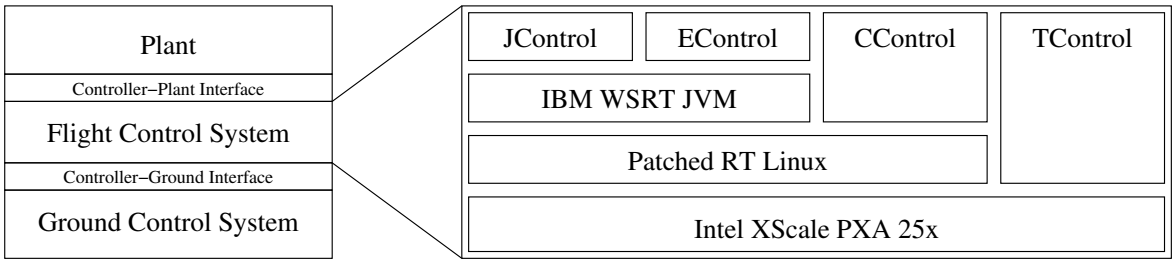

Figure 3.2: JAviator software architecture [6].

revolution speed nominal values withershins. The FCS implements the actual control algorithms to fly the JAviator. The GCS logs and visualizes reported vehicle states from the FCS and stipulates target values for the FCS for roll, pitch, yaw, and altitude. Currently, both FCS and GCS allow manually controlled flights only.

The Plant provides an abstracted view on sensors and actuators to the FCS. Currently exist two implementations. The *Physical Plant* is a collection of C programs residing in the Robostix and in the Gumstix onboard the JAviator. The *Simulated Plant* or *MockJAviator* is a pure Java program that simulates the JAviator dynamics and kinematics.

The Controller-Plant Interface specifies the communication protocol between Plant and FCS. It is the FCS that initiates the communication by sending new revolution speed nominal values for the motors to the Plant. As response, the Plant returns the current sensor values to the FCS.

Figure 3.2 depicts the four available implementations of the FCS. JControl is pure Java code utilizing threads. EControl employs Exotasks [7] as a threads alternative, where each Exotask uses a private heap that is garbage collected separately whenever the task does not execute. The IBM WSRT JVM executes either JControl or EControl. CControl implements the FCS in a single Linux process written in C. The underlying Linux with real-time patches applied executes either CControl or the IBM WSRT JVM. TControl is a port of CControl to the real-time operating system Tiptoe [12]. The current version of Tiptoe lacks support for user processes, therefore TControl is hard-wired to the Tiptoe kernel.

The Controller-Ground Interface defines the communication protocol between FCS and GCS. It is the FCS that initiates the communication by sending the current sensor data to the GCS. Subsequently, the GCS responds with navigation data, that is, the roll, pitch, yaw, and altitude nominal values.

The GCS consists of a control terminal and a logging system. Via the control terminal, an operator can fly the JAviator by entering roll, pitch, yaw, and altitude nominal values using either keyboard, joystick, or both. The control terminal visualizes the vehicle state corresponding to the received sensor data in a 3-dimensional (3-D) view of the JAviator, as well as in gauges for roll, pitch, yaw and altitude. For studying flight stability and system performance, the logging system allows real-time tracing during flights.

3.3 Summary

This chapter presents the high-performance four-rotor model helicopter JAviator. After outlining the involved elements, this chapter briefly explains their interaction. The JAviator software comprises the components Plant, FCS, and GCS, which interact via well defined interfaces. This chapter describes the currently existing implementations of the components Plant, FCS, and GCS.

4

Location systems

For autonomous flights along a given set course, a UAV needs to know its current position at any time. Globally available satellite-based location systems for outdoor application and close-range location systems for indoor utilization help a UAV to find its current position with sufficient precision and short delay. Transforming orthogonal coordinates of close-range location systems to global coordinates allows easy combining of several close-range and satellite-based location systems for autonomously flying UAVs.

4.1 Satellite-based location systems

In a Global Navigation Satellite System (GNSS), like the U.S.-american GPS [5, 38], the Russian Globalnaja Nawigazionnaja Sputnikowaja Sistema (GLONASS) [3, 5] or the European Galileo [5, 17], each satellite broadcasts its position and time. Applying the data from at least four satellites allows a receiver the determination of its current position and time.

Errors of satellite position, ionospheric delays, drift of satellite clocks, and electromagnetic interference cause inaccuracies. Receivers may integrate correction data, provided by GNSS augmentation systems, into the calculation process to improve the reliability, availability, or accuracy of the satellite navigation signal. For GPS exist space-based augmentation systems (SBAS) [16] like EGNOS [5, 18], WAAS [5], MSAS [5, 41], and GAGAN [5, 35], as well as ground-based augmentation systems (GBAS) [15] like LAAS [5, 19], GRAS [13], and APOS [8].

The position accuracy of GPS is about 13 m for 95% of all measurements [42]. Receivers may achieve a position accuracy of approximately 1 m by using DGPS derived from signal travel time delay measurement (pseudo-range or C/A code measurement). In order to obtain a precision within millimeters, DGPS based on the phase measurement of the carrier signal must be applied.

4.2 Close-range location systems

In urban canyons and inside buildings GNSSs can not be utilized successfully in a majority of cases. Indoor location systems are appliances that allow navigation within closed rooms. They employ infra red light, ultrasound [34, 39], radio waves [10, 37], and combinations of these for localization, but provide only *portion-of-a-room* accuracy in most instances. Indoor location systems applying ultra-wideband radio waves attain an accuracy of about 15 cm [37] to 20 cm [10], adequate for UAV positioning.

4.3 Coordinate transformation

To handle linear coordinates of close-range location systems, this thesis proposes a conversion into Department of Defense World Geodetic System 1984 (WGS 84) [33] coordinates, as GPS receivers provide them. This allows a straightforward combination of satellite-based and close-range location systems. A close-range location system usually describes positions in an orthogonal coordinate system comprising x-, y-, and z-axes, as shown in Figure 4.1. It considers an arbitrary position U as a vector $\overrightarrow{u} = [u_x \;\; u_y \;\; u_z]^T$ from its point of origin R to U. Longitude λ_R, latitude φ_R, and al-

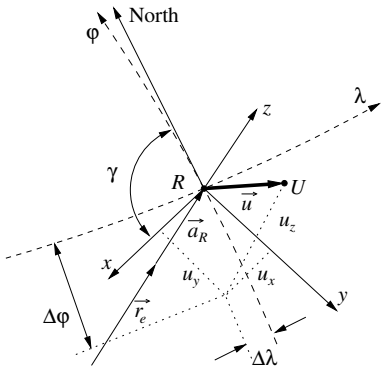

Figure 4.1: Transforming non-GPS coordinates to GPS coordinates.

titude a_R describe the location of the point of origin R in WGS 84 coordinates, where $a_R = |\overrightarrow{a_R}|$ is the distance from the WGS 84 ellipsoid to R. $r_e = |\overrightarrow{r_e}|$ is the radius of the WGS 84 ellipsoid at longitude λ_R and latitude φ_R. γ describes the angle of rotation between the North tangent to the median through R and the x-axis of the close-range location system. x- and y-axes are elements of the plane defined by the tangent to the meridian and the tangent to the parallel in R.

For coordinate transformations, this thesis considers the Earth as a sphere having radius r_e. Consistent with the scenario in Figure 4.1, the formula for the altitude a_U of location U is

$$a_U = a_R + u_z. \tag{4.1}$$

The formulas for the angle differences $\Delta\varphi$ and $\Delta\lambda$ are

$$\Delta\varphi = \frac{1}{r}(u_x \cos\gamma - u_y \sin\gamma), \quad and \tag{4.2}$$

$$\Delta\lambda = \frac{1}{r\cos\varphi_R}(-u_x \sin\gamma - u_y \cos\gamma) \tag{4.3}$$

where $r = r_e + a_R$. Formulas (4.1) and (4.2) assume that the Earth's radius r_e is very large in comparison to the limits of the close-range location system's coordinate system. In the vicinity of R, the Earth's surface is a satisfactory approximation of the tangential plane in position R. Formulas (4.2) and (4.3) approximate the transformation from linear to WGS 84 coordinates by equating linear distances with arc lengths on the Earth's surface.

Combining formulas (4.1), (4.2), and (4.3) to calculate position U in WGS 84 coordinates results in

$$\begin{bmatrix}\varphi_U \\ \lambda_U \\ a_U\end{bmatrix} = \begin{bmatrix}\varphi_R \\ \lambda_R \\ a_R\end{bmatrix} + \begin{bmatrix}r^{-1} & 0 & 0 \\ 0 & (r\cos\varphi_R)^{-1} & 0 \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}\cos\gamma & -\sin\gamma & 0 \\ -\sin\gamma & -\cos\gamma & 0 \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}u_x \\ u_y \\ u_z\end{bmatrix} \tag{4.4}$$

where $[\varphi_U \ \ \lambda_U \ \ a_U]^T = U$ and $[\varphi_R \ \ \lambda_R \ \ a_R]^T = R$

4.4 Summary

This chapter presents an overview of satellite-based location systems, the causes for inaccuracies, and available augmentation systems. Additionally, it briefly summarizes close-range location systems, which use infra red light, ultrasound, radio waves, and combinations of these for localization. Finally, this chapter proposes the transformation of linear close-range location system coordinates into WGS 84 coordinates to allow the combination of close-range and satellite-based location systems.

5

Vehicle control

This chapter introduces the dynamics of four-rotor helicopters, develops the controllers for flying UAVs, and describes the implemented algorithms for flying UAVs autonomously along a given set course.

5.1 Four-rotor helicopter dynamics

This thesis uses the linearized mathematical model developed in [24]. Equation (5.1) presents the kinematic model in matrix form, where m is the four-rotor helicopter's mass in kilograms and g the gravitational acceleration in kg/m^2. $\ddot{r}_x$, $\ddot{r}_y$, and $\ddot{r}_z$ denote the x-, y-, and z-components of the position vector's second derivative with respect to time in m/s^2. Variables ϕ, θ, and ψ represent the angles roll, pitch, and yaw in radians. T is the total trust of all rotors T_i in Newtons, that is, $T = \sum_{i=1}^{4} T_i$. This model is the result of a small angle approximation and allows angles up to about $\pm\pi/6$ rad.

$$m \begin{bmatrix} \ddot{r}_x \\ \ddot{r}_y \\ \ddot{r}_z \end{bmatrix} = \begin{bmatrix} 1 & \psi & \theta \\ \psi & 1 & \phi \\ \theta & -\phi & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -T \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{5.1}$$

Equation (5.2) shows the dynamic model for a four-rotor helicopter. I_x, I_y, and I_z identify the x-, y-, and z-axis moments of inertia in kg $\cdot$ m^2. $\ddot{\phi}$, $\ddot{\theta}$, and $\ddot{\psi}$ indicate the second derivatives with respect to time of roll, pitch, and yaw in rad/s^2. Because of the symmetric body of the helicopter, all rotor axes have equal distance to the center of gravity, represented by l in meters. Constant K_r describes the proportion between thrust and torque of a rotor. T_1, T_2, T_3, and T_4 stand for the thrusts of each rotor.

$$\begin{bmatrix} I_x\ddot{\phi} \\ I_y\ddot{\theta} \\ I_z\ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & l & 0 & -l \\ l & 0 & -l & 0 \\ K_r & -K_r & K_r & -K_r \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} \tag{5.2}$$

The model consists of six independent second-order differential equations, as shown in (5.1) and (5.2). The three equations in (5.1) affect the design of the position and

altitude controller. The other three equations in (5.2) have an impact on the design of the attitude controllers.

Because the JAviator employs revolution-speed-controlled brushless synchronous motors, the connection between the rotor's thrust and angular speed is of particular interest. [32] defines the constant rotor thrust coefficient formally as

$$C_T = \frac{T}{\frac{1}{2}\rho A \Omega^2 R^2} \tag{5.3}$$

where ρ is the air density in kg/m^3, A is the area swept out by the rotor in m^2, Ω is the angular speed of the rotor in rad/s, and R is the radius of the propeller in meters. Rearranging this equation and solving for the rotor thrust gives

$$T = \frac{1}{2} C_T \rho A R^2 \Omega^2. \tag{5.4}$$

As depicted in Equation (5.4) thrust T is directly proportional to the squared revolution speed, thus the controllers have to adapt to this non-linearity.

5.2 Controller overview

Figure 5.1 visualizes an overview of the proposed controllers including JAviator Plant, GPS receiver, and autopilot. An array of four switches allows changing from manual

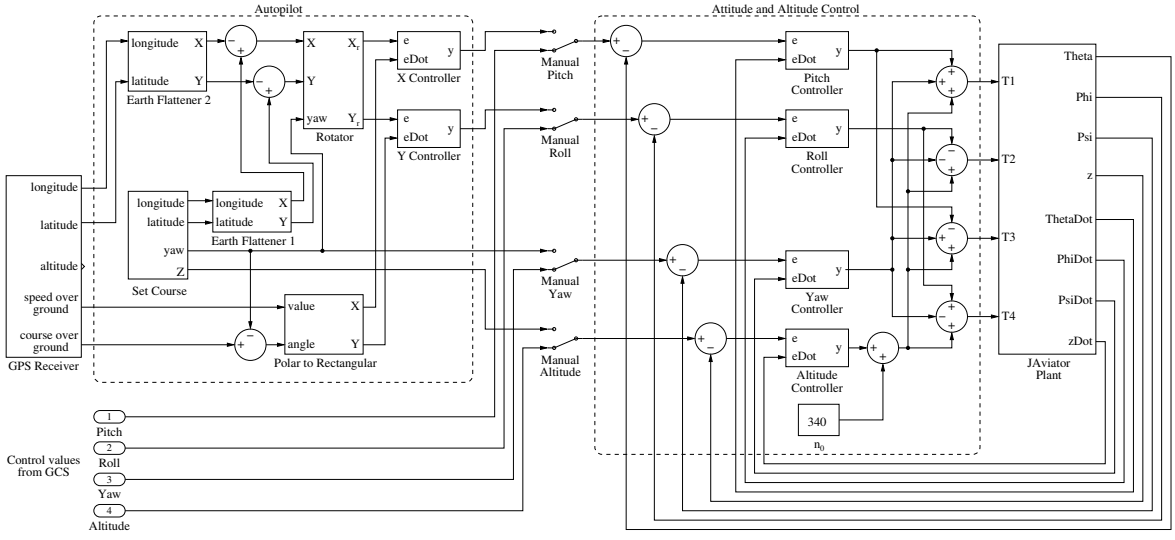

Figure 5.1: Controller overview including JAviator Plant and GPS receiver.

to autonomous flight and vice versa. For manual flights, the GCS delivers the desired values for roll, pitch, yaw, and altitude. When performing autonomous flights, the autopilot issues the values for latitude, longitude, orientation, and height above ground.

The displayed *Earth Flattener* blocks transform the spherical latitude φ and longitude λ values to a x-y-plane by adopting formulas (5.5) and (5.6).

$$r_x = r_e\varphi \tag{5.5}$$
$$r_y = r_e\lambda\cos(\varphi) \tag{5.6}$$

Variables r_x and r_y denote the x- and y-components of the position vector to the vehicle center of gravity in meters. r_e is the average earth radius in meters, φ is the latitude position in radians, and λ is the longitude position in radians. After calculating the difference between the desired position from the autopilot and the current position from the GPS receiver, that is, Δr_x and Δr_y, the *Rotator* considers the desired vehicle yaw ψ from the autopilot by applying Formula (5.7).

$$\begin{bmatrix}\Delta r'_x \\ \Delta r'_y\end{bmatrix} = \begin{bmatrix}\cos(\psi) & -sin(\psi) \\ sin(\psi) & \cos(\psi)\end{bmatrix}\begin{bmatrix}\Delta r_x \\ \Delta r_y\end{bmatrix} \tag{5.7}$$

The presented *Polar to Rectangular* converter transforms signal *Speed over Ground* from the GPS receiver to its rectangular equivalents $\dot{r_x}$ and $\dot{r_y}$ by regarding the difference between the *Course over Ground* and the desired yaw of the vehicle. The controllers for x and y utilize the rotated differences $\Delta r'_x$ and $\Delta r'_y$, as well as the values of $\dot{r_x}$, and $\dot{r_y}$ as inputs.

5.3 Altitude controller design

The differential equation for $\ddot{r_z}$ extracted from Equation (5.1) and combined with Equation (5.4) is as follows

$$\ddot{r}_z = g - \frac{C_T\rho AR^2}{2m}\Omega^2. \tag{5.8}$$

This thesis proposes a PD controller with some modifications in regard to the quadratic dependency of Ω and the available signals. As depicted in Figure 5.2, the controller uses the current altitude z and its first derivative $\dot{z}$, provided by the JAviator Plant.

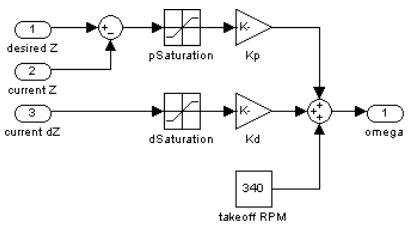

Figure 5.2: Altitude controller.

The controller saturates the difference between desired and current z, as well as $\dot{z}$ in order to allow input signals up to $\pm\infty$ without getting unstable. It also adds the takeoff

revolution speed to the result, to master the quadratic dependency of Ω. In doing so, the controller operates in a rather linear environment and therefore gains stability.

5.4 Attitude controller design

Rearranging Equation (5.1) and solving for the angular accelerations $\ddot{\phi}$, $\ddot{\theta}$, and $\ddot{\psi}$ gives

$$\ddot{\phi} = \frac{l}{I_x}(T_2 - T_4) \tag{5.9}$$

$$\ddot{\theta} = \frac{l}{I_y}(T_1 - T_3) \tag{5.10}$$

$$\ddot{\psi} = \frac{l}{I_z}(T_1 + T_3 - (T_2 + T_4)) \tag{5.11}$$

As Equation (5.9) and Equation (5.10) convey, varying the thrust of opposing rotors differently leads to an acceleration of ϕ and θ, respectively. Equation (5.11) indicates that varying opposing rotors equally, but different from the other rotor pair, leads to an acceleration of ψ. The IMU of the JAviator Plant delivers the current values for ϕ, $\dot{\phi}$, θ, $\dot{\theta}$, ψ, and $\dot{\psi}$. This thesis recommends saturated PD controllers to manage the attitude

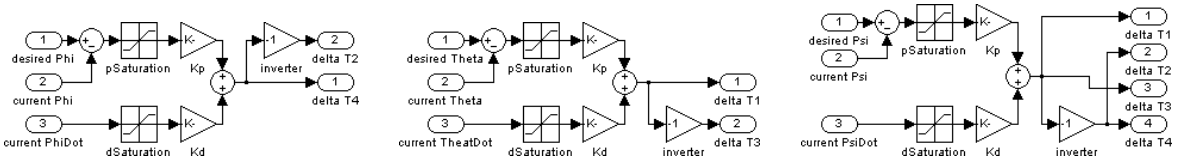

Figure 5.3: Attitude controllers for roll, pitch, and yaw.

stabilization of ϕ, θ, and ψ, as depicted in Figure 5.3. The altitude controller already linearizes the quadratic dependency to Ω and therefore, the design of the attitude controllers disregards this fact.

5.5 Position controller design

Combining Equation (5.1) and Equation (5.4) and solving for $\ddot{r}_x$ and $\ddot{r}_y$ gives

$$\ddot{r}_x = -\theta \frac{C_T \rho A R^2}{2} \Omega^2 \tag{5.12}$$

$$\ddot{r}_y = -\phi \frac{C_T \rho A R^2}{2} \Omega^2. \tag{5.13}$$

As shown in Equation (5.12) and Equation (5.13), non-zero roll and pitch values lead to an acceleration in axes x and y, respectively. A PD controller for each axis is able to handle stabilization. The design of the controllers visualized in Figure 5.4 discounts the

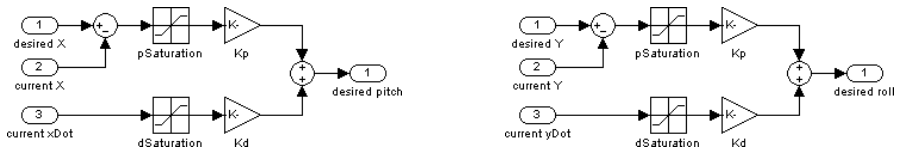

Figure 5.4: Position controllers for x- and y-axis.

quadratic quadratic dependency to Ω, because the altitude controller already considers it sufficiently.

5.6 Path following

Path following is a topic that has been elaborated on extensively in literature. Describing a trajectory is seen differently among the authors and depends also on the particular requirements. Egerstedt et al. [14] describe a trajectory as a path of waypoints, velocities, and accelerations. The authors consider the waypoints as soft constraint, allowing for a trade off between accuracy and smoothness. Conte et al. [11] interconnect the waypoints by applying a cubic spline interpolation, which has second-order continuity at the joints to avoid discontinuity in the helicopter's acceleration. Hoffmann et al. [25] define a trajectory as a series of waypoints interconnected by linear segments of constant velocity.

This thesis develops a flight controller, which follows given waypoints. The developed algorithm defines a trajectory as a sequence of locations the vehicle has to pass. This thesis proposes the durations to traverse the routes between waypoints for defining the dynamic properties of a trajectory. It is this procedure that allows trajectories comprising well defined halts. Two successive waypoints pointing at the same location lock the vehicle into position for the specified traversal time.

Figure 5.5 depicts the trajectory route from waypoint W_i to waypoint W_{i+1}, where $\overline{v}_i$ is the average velocity between these waypoints. The average velocities $\overline{v}_{i-1}$ and $\overline{v}_{i+1}$ belong to the previous and next routes of the trajectory. α_i is the angle between the

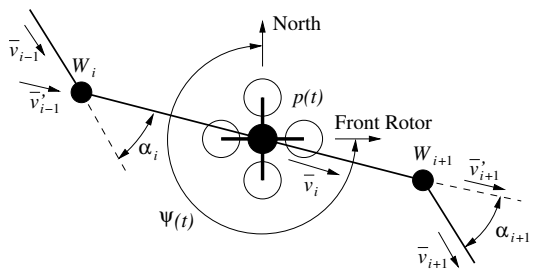

Figure 5.5: Vehicle path definition.

current and the previous route. α_{i+1} is the angle between the current and the next

route. $\overline{v}'_{i-1}$ and $\overline{v}'_{i+1}$ are the components of $\overline{v}_{i-1}$ and $\overline{v}_{i+1}$ in direction of the current route. They are formally defined as $\overline{v}'_{i-1} = \overline{v}_{i-1}\cos(\alpha_i)$ and $\overline{v}'_{i+1} = \overline{v}_{i+1}\cos(\alpha_{i+1})$. $p(t)$ is the position of the vehicle and $\Psi(t)$ its orientation at time t. $\Psi(t)$ defines the orientation of the vehicle as the angle between North and its front rotor.

Inspired by [31], the implemented trajectory controller splits the routes between waypoints into three segments of constant velocity or constant acceleration. Figure 5.6 shows the four possible acceleration scenarios to traverse a given route of length l in the requested duration t_d. $\overline{v}_i$ is the average velocity of the current route defined as

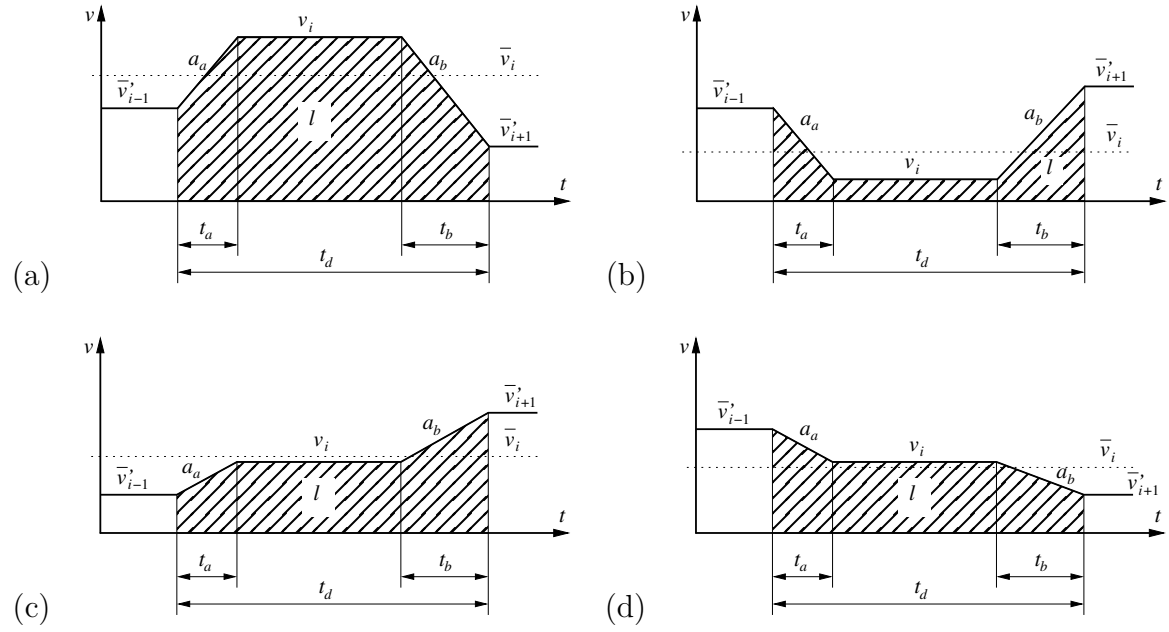

Figure 5.6: Acceleration scenarios.

$\overline{v}_i = \frac{l}{t_d}$. The average speed of the route before the first route and the average velocity of the route after the last route are assumed as zero. Starting from $\overline{v}'_{i-1}$ the vehicle accelerates in the first segment with a_a for time t_a to reach v_i of the middle segment. In the third segment the vehicle accelerates with a_b for time t_b to reach the exit velocity $\overline{v}'_{i+1}$. The route length of the scenarios shown in Figure 5.6 is

$$l = v_i t_d - \frac{(v_i - \overline{v}'_{i-1})^2}{2a_a} + \frac{(\overline{v}'_{i+1} - v_i)^2}{2a_b}. \tag{5.14}$$

Larger values for a_a and a_b allow a bigger spread of route lengths l. By pinning down $|a_a|$ and $|a_b|$ to a positive constant a_{max}, the accelerations are formally defined as

$$a_a = x \cdot a_{max} \qquad and \qquad a_b = y \cdot a_{max} \qquad where \qquad x, y \in \{-1, 1\}. \tag{5.15}$$

Equation (5.16) and Equation (5.17) define the rules for choosing x and y.

$$x = \begin{cases} -1 & \text{for } \overline{v}'_{i-1} > \overline{v}_i \\ 1 & \text{for } \overline{v}'_{i-1} \leq \overline{v}_i \end{cases} \tag{5.16}$$

$$y = \begin{cases} -1 & \text{for } \overline{v}_i > \overline{v}'_{i+1} \\ 1 & \text{for } \overline{v}_i \leq \overline{v}'_{i+1} \end{cases} \tag{5.17}$$

Combining Equation (5.14) and Equation (5.15) gives

$$l = v_i t_d - \frac{(v_i - \overline{v}'_{i-1})^2}{2xa_{max}} + \frac{(\overline{v}'_{i+1} - v_i)^2}{2ya_{max}}. \tag{5.18}$$

Solving Equation (5.18) for the velocity v_i of the middle segment results in two solutions shown in Equations (5.19) and (5.20) below.

$$x \neq y: \quad \begin{aligned} f &= \tfrac{2}{y-x}(a_{max}t_d + x\overline{v}'_{i+1} - y\overline{v}'_{i-1}) \\ g &= \tfrac{1}{y-x}(-2a_{max}l - x\overline{v}'^2_{i+1} + y\overline{v}'^2_{i-1}) \\ v_i &= -\tfrac{f}{2} + \sqrt{\tfrac{f^2}{4} - g} \end{aligned} \tag{5.19}$$

$$x = y: \quad v_i = \frac{-2a_{max}l - y\overline{v}'^2_{i-1} + x\overline{v}'^2_{i+1}}{2(-a_{max}t_d - y\overline{v}'_{i-1} + x\overline{v}'_{i+1}} \tag{5.20}$$

The trajectory controller considers only positive, non-complex values for v_i as valid. Other values for v_i indicate that time t_d is too short or too long to traverse route p_i when applying a_{max} as the maximum allowed acceleration.

5.7 Summary

This chapter explains a simplified dynamic model for four-rotor helicopters, and develops altitude and attitude PD controllers for manual control. The altitude controller takes care of the rotors quadratic thrust characteristic by adding the take off revolution speed to its output. For a smooth behavior at the target value, the presented controllers apply the first derivatives of the current values and not the first derivatives of the deviations.

This chapter also develops a trajectory controller for autonomous flights, which follows given waypoints. The utilized algorithm splits the routes between waypoints into segments of constant velocity or constant acceleration.

6

Software design and implementation

This chapter presents the design and implementation of the JNavigator software. After explaining the basic concepts, it continues with an overview of the functional model, and proceeds with a thorough description of the implemented software modules.

It is the aim of this thesis to provide the capabilities necessary to support manual and autonomous flights of the JAviator, compatible to the existing interfaces to Plant and GCS. In particular the design of the software focuses on flexibility and reusability of components.

6.1 Basic concepts

This thesis establishes two fundamental concepts. The first concept is that a class supports a constructor with a `Properties` object as the only parameter. This allows constructing new objects in a generic object factory and requires that the code for converting configuration parameters to values is part of the constructed class. The second concept is that the structure of the configuration is a tree. Hence, constructing an object implies to identify the subtree belonging to the constructed class in the configuration and initialize a new object by handing over the subtree as a `Properties` object to the class constructor.

6.1.1 Object construction via properties

A class supports a constructor with a `Properties` object as the only parameter. To create an instance of a class, a calling object only has to support a `Properties` object that contains the necessary parameters for construction. The class constructor is responsible for verifying the correctness of the properties and for converting the properties to values of other type for further usage. By applying a generic factory, a caller only has to support a class name and properties for instantiating a new object. For further access, the caller only needs to know a super class or interface of the newly created object. Additionally, the caller remains completely unaware of the concrete implementation.

6.1.2 Hierarchical configuration

The basic idea is organizing the configuration in agreement to the hierarchy of instantiated objects. Hence, an object only creates its child objects and not its grandchild objects. It is the child class that is responsible for creating the grandchild objects. This thesis proposes to implement the hierarchical configuration as Java properties. An object isolates the properties tree of a child and constructs the child object by transferring this sub-tree root to the child class constructor.

Figure 6.1 depicts a configuration, where the configuration of object `A` and its child objects `A1` and `A2` are organized as a tree. Object `A` isolates the sub-tree of `A1` and cre-

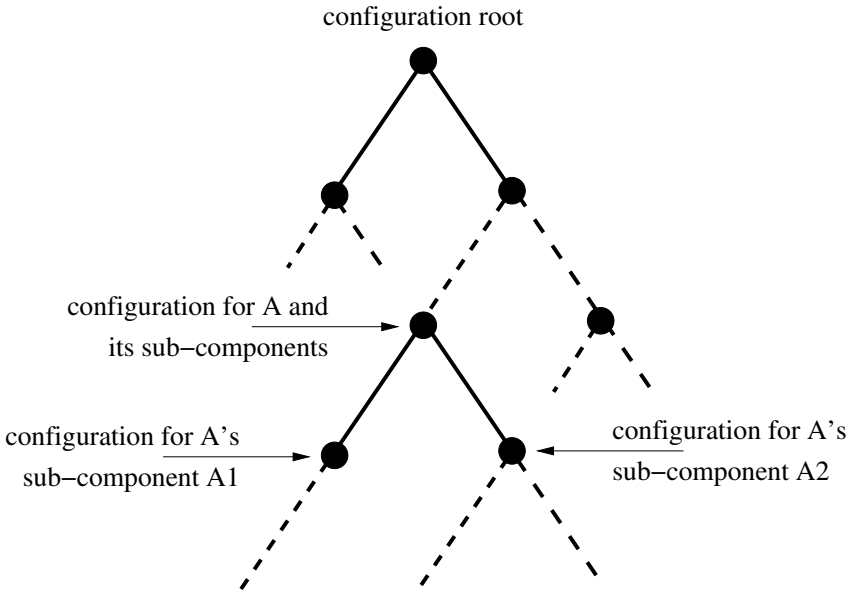

Figure 6.1: Configuration tree.

ates object `A1` by calling its constructor code with the root of the sub-tree. Thereafter, object `A` isolates the sub-tree of `A2` and creates object `A2` by calling its constructor code with the root of the corresponding sub-tree. Section 6.3.2 elaborates on isolating sub-trees and generically creating objects.

6.2 Functional model

Figure 6.2 visualizes the use cases of the implemented JNavigator software, where GCS and Plant are the identified actors. It is the GCS that initiates the use cases for both manual and autonomous flight, whereas the Plant is passive. The manual-flight use case includes use cases for controlling altitude and attitude, respectively. The autonomous-flight use case inherits the functionality of the manual flight and includes the use cases for uploading set courses, starting and aborting autonomous flights, following set courses, and determining the current position of the helicopter. The following subsections describe the use cases in detail.

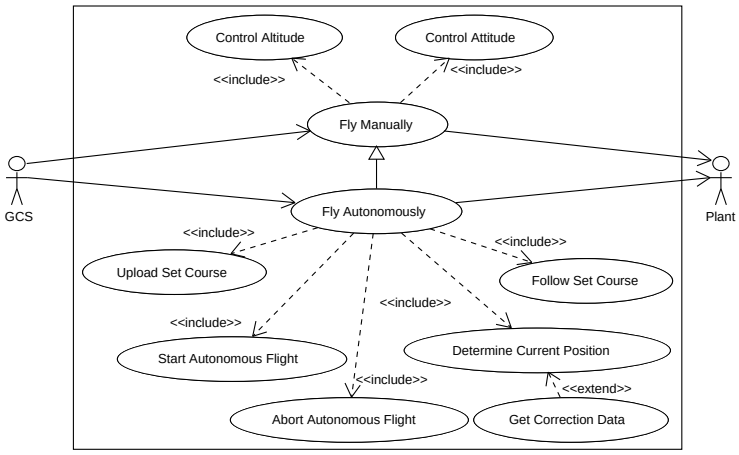

Figure 6.2: Use cases of the JNavigator software.

6.2.1 Use case "Fly Manually"

The FCS periodically sends motor revolution speed nominal values to the Plant. For each received packet the Plant returns sensor data to the FCS, which forwards the data to the connected GCS. In response to sensor data, the GCS transmits attitude and altitude nominal values to the FCS. Applying attitude and altitude controllers to nominal values and sensor data results in new revolution speed nominal values for the motors.

Control Altitude. Controlling altitude implies controlling the height over ground of the UAV. The FCS applies the altitude controller developed in Section 5.3 to the received nominal values, as well as the sensor data for height over ground and its first derivative.

Control Attitude. Controlling attitude means controlling roll, pitch, and yaw of the UAV. The FCS adopts the attitude controllers developed in Section 5.4 to the received nominal values, as well as sensor data for roll, pitch, and yaw plus their first derivatives.

6.2.2 Use case "Fly Autonomously"

This use case adopts the functionality of the manual flight use case. The GCS uploads set courses to the FCS, instead of periodically sending it nominal values. After receiving the start message for conducting a particular set course, the FCS navigates the JAviator along this set course.

Upload Set Course. Currently, uploading a set course is simply copying the file containing the set course data to the Gumstix running the FCS.

Start autonomous flight. The GCS sends a start message for an autonomous flight, which contains the path name of the set course file, to the FCS. After successfully analyzing the set course, the FCS switches from manual to autonomous control and maneuvers the helicopter along this set course by applying use case *Follow Set Course.* If the FCS fails to analyze the set course it remains in manual control.

Abort Autonomous Flight. The GCS aborts an autonomous flight by sending an abort message to the FCS, which switches to manual control instantly.

Determine Current Position. The FCS determines the current position by utilizing a GPS receiver or a close-range location system. It retrieves the current orientation by making use of the Plant's IMU. The FCS forwards available correction data to a GPS receiver by applying use case *Get Correction Data*.

Get Correction Data. This use case currently applies to GPS receivers only. The FCS forwards available RTCM SC-104 correction data to the corresponding GPS receiver. Sources of correction data may provide correction data via GSM or Internet (NTRIP).

Follow Set Course. When following a set course, the FCS calculates the nominal values for position and orientation by considering the elapsed flight time and the analyzed course data. With these nominal values, the current position, and the current orientation, the FCS stipulates nominal values for the altitude and attitude controllers.

6.3 Implemented software modules

As displayed in Figure 6.3, the implemented software modules are the following:

- Input/Output
- Utilities
- Communication
- Course
- GPS
- Location
- JAviator Control
- User Interface

Module *Input/Output* abstracts the calls to the operating system to access TCP/IP and Bluetooth sockets, as well as serial lines. Module *Utilities* provides functionality for manipulating properties and objects. Module *Communication* handles the communication to GCS and Plant. Module *Course* covers all functionality necessary to navigate on planets, as well as to plan and run set courses. Module *GPS* processes messages from GPS receivers and augmentation systems. Module *Location* allows integrating close-range location systems. Module *JAviator Control* executes manual and autonomous control of the helicopter. Module *User Interface* consists of an application to visualize heading, position, and velocity of the helicopter, as well as the time received from a GPS receiver.

Figure 6.3 presents the implemented modules, which the remainder of this chapter thoroughly describes.

6.3.1 Module Input/Output

This module abstracts calls to the operating system to access TCP/IP and Bluetooth sockets, as well as serial lines. It comprises a shared object and a Java archive. The

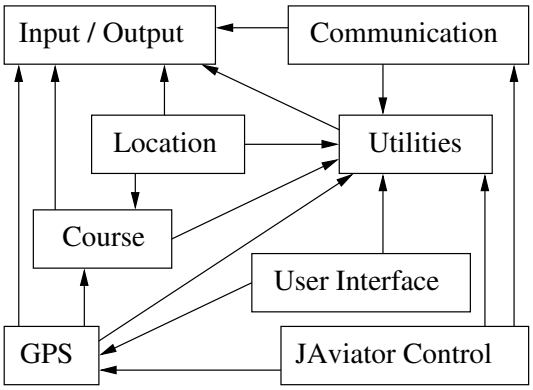

Figure 6.3: Implemented software modules and their dependencies.

shared object just makes the system calls `connect()`, `close()`, `read()`, and `write()` to Bluetooth sockets, as well as the system calls `open()`, `close()`, `read()`, and `write()` to serial lines accessible to Java code. The rest of the implementation resides in the Java archive. The shared object implementation is currently available for Linux systems only.

Figure 6.4 displays the class diagram of module *Input/Output*. Publicly available classes and interfaces are `IConnection`, `TcpSocketServer`, `SocketWrapper`, `TcpSocket`, `BluetoothSocket`, and `SerialLine`. It is interface `IConnection` that nicely abstracts

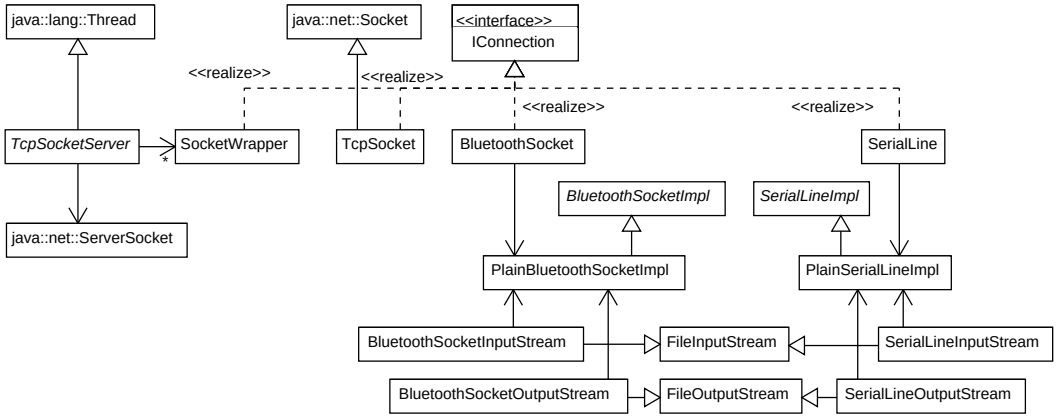

Figure 6.4: Class diagram of Module Input/Output.

connections to allow access to TCP/IP and Bluetooth sockets, as well as serial lines uniformly.

Abstract class `TcpSocketServer` implements a TCP/IP socket server. It opens a `ServerSocket` on the specified port number and waits for clients to connect. It is a separate working thread that takes an incoming connection and deals with the actual receiving and sending of data. A concrete class that inherits the `TcpSocketServer` has

to provide the implementation for starting the working thread. Class `TcpSocketServer` employs class `SocketWrapper` to proxy incoming connections to interface `IConnection`.

Class `TcpSocket` is an envelope for Java Runtime Environment (JRE) class `Socket` for providing TCP/IP connections via an `IConnection` interface. The code below demonstrates how to connect to a TCP/IP service by constructing a `TcpSocket` utilizing a property file. For example, the property file `mytcp.properties` contains the following properties

```
host = 10.10.11.201
port = 3333
```

The code to initiate the connection is:

```
Properties props = new Properties ();
props.load (new FileInputStream("mytcp.properties"));
IConnection connection = (IConnection)(new TcpSocket (props));
```

Class `BluetoothSocket` allows connecting to Bluetooth services and class `SerialLine` facilitates connecting to serial lines. JRE class `Socket` is archetype for the design of the associated Bluetooth and serial line classes. The code for connecting to Bluetooth sockets or serial lines is similar to the code for TCP/IP sockets. Bluetooth sockets need a device address and a channel address for construction. To initiate a serial line connection, class `SerialLine` needs the parameters interface device path, baud rate, parity, number of start bits, and number stop bits.

6.3.2 Module Utilities

This module provides common functionality to all other modules, except module *Input/Output*. It comprises one Java archive, which exports the classes and interfaces `ByteArrayUtils`, `ObjectFactory`, `PropertyUtils`, `IClock`, and `SystemClock`.

Static class `ByteArrayUtils` implements methods to reverse and split byte arrays, as well as methods to convert integer and double data types to byte arrays and vice versa.

Singleton `ObjectFactory` is a generic factory to create objects defined by properties. It instantiates an object as `IConnection`, `InputStream`, or `Object` and it is this class that causes the dependency to module *Input/Output*. To instantiate a TCP/IP connection, an applying code must provide the full name of class `TcpSocket`, as well as the host name and port number to connect to as properties, as shown in the following example.

```
prefix.className = at.uni_salzburg.cs.ckgroup.io.TcpSocket
prefix.host = 10.10.11.201
prefix.port = 3333
```

With the properties loaded in an instance of class `Properties`, the code to instantiate a `TcpSocket` as a reference to interface `IConnection` is as follows.

```
Properties props = ...
ObjectFactory factory = ObjectFactory.getInstance ();
IConnection connection =
      factory.instantiateIConnection ("prefix.", props);
```

The applying code is capable to instantiate an `IConnection` object without being aware of the concrete implementation. The only requirement is that the class specified in property `prefix.className` can be found in the class path of the JVM.

Static class `PropertyUtils` provides methods for loading property files available in the context of the current class loader. Additionally, it allows replacing parts of property keys and extracting property subsets by means of regular expressions.

The code example below illustrates the realization of the hierarchical configuration approach described earlier. It shows the construction of three TCP/IP connections specified in one property file `allmytcp.properties` as follows.

```
confA.prop1 = ...
confA.prop2 = ...
confA.prop3 = ...
confA.conn1.host = 10.10.11.201
confA.conn1.port = 3333
confA.conn2.host = 10.10.11.200
confA.conn2.port = 3334
confA.conn3.host = 10.10.11.199
confA.conn3.port = 3335
confB...
confC...
```

As depicted above, the properties of TCP/IP connections one, two, and tree are subtrees of component A's configuration. Loading the properties and extract the subtree for component A shows the following code.

```
Properties props = new Properties ();
props.load (new FileInputStream("allmytcp.properties"));
Properties propsA = PropertyUtils.extract ("confA.*", props);
propsA = PropertyUtils.replaceFirst ("confA.", "", propsA);
```

Static method `PropertyUtils.extract()` filters the loaded properties and extracts the subset whose keys match regular expression `"confA.*"`. Thereafter, static method `PropertyUtils.replaceFirst()` chops off the first part of the property keys. The code below demonstrates preparing the property subtrees and establishing the TCP/IP connections.

```
Properties props1 = PropertyUtils.extract ("conn1.*", propsA);
Properties props2 = PropertyUtils.extract ("conn2.*", propsA);
Properties props3 = PropertyUtils.extract ("conn3.*", propsA);
```

```
props1 = PropertyUtils.replaceFirst ("conn1.", "", props1);
props2 = PropertyUtils.replaceFirst ("conn2.", "", props1);
props3 = PropertyUtils.replaceFirst ("conn3.", "", props3);
IConnection connection1 = (IConnection)(new TcpSocket (props1));
IConnection connection2 = (IConnection)(new TcpSocket (props2));
IConnection connection3 = (IConnection)(new TcpSocket (props3));
```

The code above demonstrates how the hierarchical configuration approach works. However, employing class `ObjectFactory` for constructing the connections, as depicted in an example earlier, leads to much more readable code.

Interface `IClock` provides the functionality of a simple clock. Some of the classes described later use implementations of this interface, rather than the `System` object. This allows unit tests to employ own mock clocks that are independent of the actual execution time. Class `SystemClock` is a simple wrapper around the `currentTimeMillis()` method of the `System` object and implements the `IClock` interface.

6.3.3 Module Communication

This module implements an upper and a lower communication layer. The upper layer dispatches received packets among notified packet listeners. The lower layer handles the connections to GCS and Plant via `IConnection` links. After verifying a received raw packet, the lower layer converts it to a Data Transfer Object (DTO) and delivers it to the upper layer. When the upper layer forwards a DTO to the lower layer, it is the lower layer that transforms the DTO to a raw packet and sends it to the required link. This thesis considers a DTO as an entity containing arbitrary interrelated values, which conveys information from FCS to Plant and GCS, respectively and vice versa.

Module *Communication* comprises one Java archive, which exports the upper layer communication classes and interfaces `Dispatcher`, `IDataTransferObject`, `IDataTransferObjectListener`, `IDataTransferObjectProvider`, `ISender`, and `DataTransferObjectLogger`, as shown in Figure 6.5. It also exports the DTO classes, which implement interface `IDataTransferObject`, as displayed in 6.9. The exported lower layer communication classes and interfaces, as depicted in Figure 6.5, are `Packet`, `TransceiverAdapter`, `ITransceiver`, `BufferedTransceiver`, `Transceiver`, and `TcpServer`.

Class `Dispatcher` is responsible for distributing DTOs to listeners, which previously had registered with an instance of class `Dispatcher` for particular DTO types. Users of class `Dispatcher` access it via interface `IDataTransferObjectProvider`, which allows registering and deregistering listeners, as well as sending DTOs.

Listeners of class `Dispatcher` implement interface `IDataTransferObjectListener`, which provides receiving objects that implement the `IDataTransferObject` interface of DTOs.

Senders of DTO objects need no registering with instances of class `Dispatcher`. If a sender registers with an instance of class `Dispatcher`, it must supply its reference as

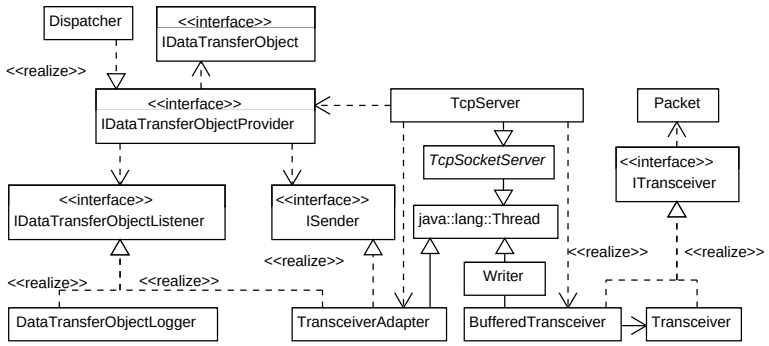

Figure 6.5: Class diagram of Module Communication.

an `ISender` interface when sending DTOs. This prevents DTOs from being sent to their origin.

Class `DataTransferObjectLogger` registers with one instance of class `Dispatcher` for all DTO object types and logs every received DTO detailed to file, including the time stamp in millisecond accuracy.

Class `Packet` represents a raw packet that is employed as the data transfer vehicle between Plant, FCS, and GCS. Figure 6.6 displays the utilized packet format. Class

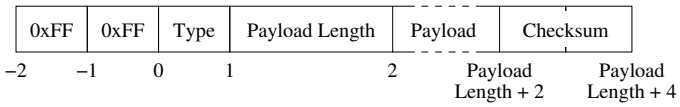

Figure 6.6: Raw packet format for communication between Plant, FCS, and GCS [6].

`Packet` converts arbitrary payload, given as a byte array, and a type into a raw communication packet by adding header, payload length, and checksum. Class `Packet` can also use an `InputStream` instance to initialize its data members. The `Transceiver` class described later uses the latter construction method to deserealize serialized packets received from an `IConnection` link.

Interface `ITransceiver` defines the functionality of a sender and receiver. It abstracts either a stream connection or packet oriented link to a resource. It allows sending and receiving `Packet` objects without bothering about the low level details of the accessed resource. Class `Transceiver` is an implementation of the `ITransceiver` interface that blocks sending and receiving single packets until their transmission is complete. Class `BufferedTransceiver` implements the `ITransceiver` interface too. It applies a ring buffer to store packages temporarily for later transmission, that is, it does not block a sender. An instance of class `BufferedTransceiver`'s inner class `Worker`, running as a separate thread, forwards the packets from the ring buffer to the underlying `Transceiver` instance. Due to the finite number of packets the ring buffer can hold, packets may be lost when sent more rapidly than the link can handle.

Class `TransceiverAdapter` is responsible for converting DTOs to raw communication packets and vice versa. The mapping between DTOs and packet types is part of the configuration and not hard coded into this class. This allows transporting packets of future extensions without changing the already existing code.

Class `TcpServer` implements a TCP/IP server by extending abstract class `TcpSocketServer` of Module *Input/Output*. After a new TCP/IP client connects to the port number of the server, the server creates instances of classes `TransceiverAdapter` and `BufferedTransceiver`. After linking the instance of class `TransceiverAdapter` with the instance of class `BufferedTransceiver` and the already running instance of class `Dispatcher`, the server spawns off the newly created instance of class `TransceiverAdapter` as a separate thread.

Figure 6.7 shows how the `TransceiverAdapter` receives a `Packet` from an `InputStream`, transforms it to a DTO, and delivers it to the `Dispatcher`.

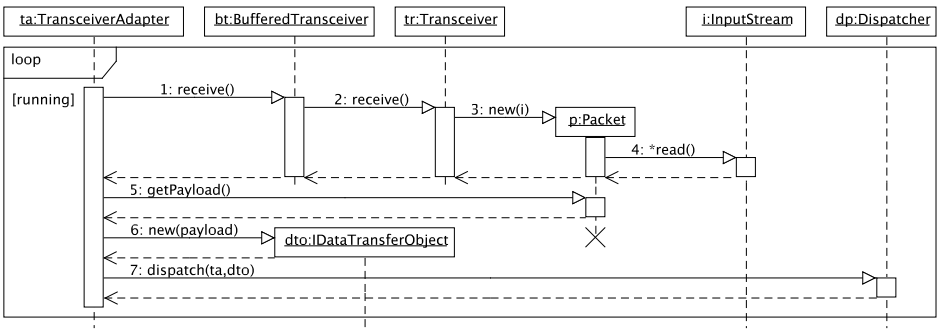

Figure 6.7: Receiving `Packet` objects from an `InputStream`, conversion to DTOs, and dispatching DTOs.

In the following, the sequence depicted in Figure 6.7 is explained in more detail.

1. `TransceiverAdapter` *ta* runs a separate thread that is responsible for receiving `Packet` objects from an `InputStream`. It is this thread that calls the `receive()` method of the associated `BufferedTransceiver` *bt*.

2. `BufferedTransceiver` *bt* calls the `receive()` method of the underlying `Transceiver` *tr*.

3. `Transceiver` *tr* creates a new `Packet` *p* by calling the `new()` operator with the `InputStream` *i* to read from as parameter.

4. The `Packet` constructor calls the `read()` method of the provided `InputStream` *i* for each byte of the new packet to be received. Then, the `Packet` constructor verifies the integrity of the received package.

5. `TransceiverAdapter` *ta* retrieves the payload from the newly received `Packet` object *p*.

6. `TransceiverAdapter` *ta* creates a DTO by calling its constructor. The parameter provided is the payload of the received `Packet` *p*. The DTO's constructor now verifies the payload and initializes the member variables of object *dto*.

7. `TransceiverAdapter` *ta* forwards the DTO to `Dispatcher` *dp*, which routes the DTO to all interested listeners. Thereafter, `TransceiverAdapter` *ta* starts from the beginning by calling the `receive()` method of `BufferedTransceiver` *bt* to get the next `Packet` object.

The sequence discussed shows how bytes, read from an `InputStream` instance, are converted into `Packet` objects, transformed to DTOs, and forwarded to the `Dispatcher`.

Figure 6.8 visualizes how the `TransceiverAdapter` receives a DTO, converts it to a `Packet` object, and sends it to an `OutputStream`.

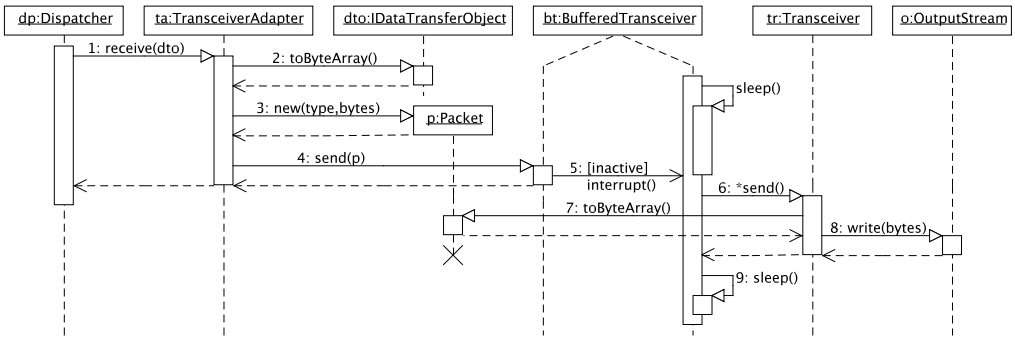

Figure 6.8: Sending DTOs to an `OutputStream`.

The procession is in detail as follows.

1. `Dispatcher` *dp* forwards a DTO to `TransceiverAdapter` *ta*.

2. `TransceiverAdapter` *ta* transforms the DTO to an array of bytes by calling the DTO's `toByteArray()` method.

3. `TransceiverAdapter` *ta* creates a new `Packet` object. The provided parameters are the packet type and the DTO as an array of bytes.

4. `TransceiverAdapter` *ta* calls the `send()` method of `BufferedTransceiver` *bt*, which stores the new `Packet` in an internal ring buffer to prevent blocking the caller.

5. `BufferedTransceiver` *bt* sends an `interrupt()` event to its inactive sending thread.

6. `BufferedTransceiver` *bt* runs a separate thread for sending the `Packet` objects stored in its internal ring buffer. For each object in the ring buffer, this thread calls the `send()` method of the underlying `Transceiver` *tr*.

7. `Transceiver` *tr* converts `Packet` *p* to an array of bytes by calling method `toByteArray()` of *p*.

8. `Transceiver` *tr* calls method `write()` of `OutputStream` *o* to write the array of bytes as a whole to *o*.

9. After the sender thread has sent all `Packet` objects to `OutputStream` *o*, it waits for the arrival of new objects.

The discussed progression ensures that the `Dispatcher` is not blocked by sending DTOs to both Plant and GCS.

Figure 6.9 visualizes the DTO classes, which realize interface `IDataTransferObject`. Class `ActuatorData` holds the motor revolution speed nominal values for the Plant,

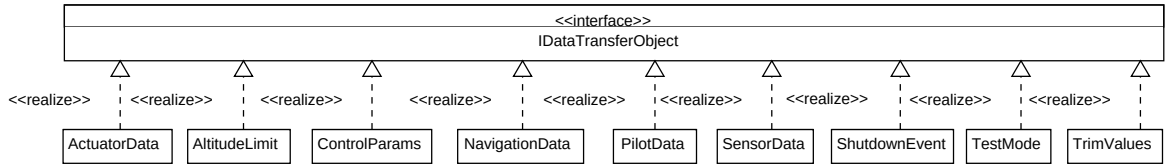

Figure 6.9: Class diagram of the Data Transfer Objects.

class `AltitudeLimit` conveys the maximum altitude the JAviator is allowed to fly, class `ControlParams` defines the parameters of the implemented controllers, class `NavigationData` stipulates the nominal values for manual flight, class `PilotData` allows starting and stopping autonomous flights, class `SensorData` transports the current sensor values from Pant to FCS, class `ShutdownEvent` immediately stops any flight, class `TestMode` initiates the test sequence while the helicopter still rests on the ground, and class `TrimValues` carries the motors revolution speed offset values.

6.3.4 Module Course

This module provides functionality to convert between coordinate systems and to plan set courses. It comprises one Java archive, which exports the classes and interfaces `AdvancedCoursePlanner`, `CartesianCoordinate`, `CourseData`, `CourseUtils`, `ICoursePlanner`, `IGeodeticSystem`, `IPositionProvider`, `ISetCourseSupplier`, `PolarCoordinate`, `PrePlanningSetCourseSupplier`, `SectionFlightPlan`, `Section`, `SetCourse`, `SphericEarth`, `VehicleStatus`, `WGS84`.

Class `CartesianCoordinate` implements a 3-tuple (x, y, z) to describe a position in an orthogonal coordinate system. It provides functionality for adding, subtracting, multiplying, and normalizing vectors. Additionally, this class is able to calculate the length of the vector.

Class `PolarCoordinate` implements a 3-tuple (latitude, longitude, and altitude) to describe a position in a polar coordinate system.

Interface `IPositionProvider` covers the functionality of a geodetic position provider. An implementation of this interface must support the current position as WGS 84 [33] coordinates.

Figure 6.10 depicts the diagram of the main classes and interfaces of Module *Course*. Interface `IGeodeticSystem` defines functionality necessary to convert polar coordinates of planets in rectangular coordinates and vice versa. Additionally, it offers functionality to estimate distance, speed, elevation, and course between two given polar coordinates. Instances of class `CourseData` act as a transport object of the estimated values. Class `SphericEarth` implements interface `IGeodeticSystem` as a spherical Earth and class `WGS84` implements the WGS 84 [33].

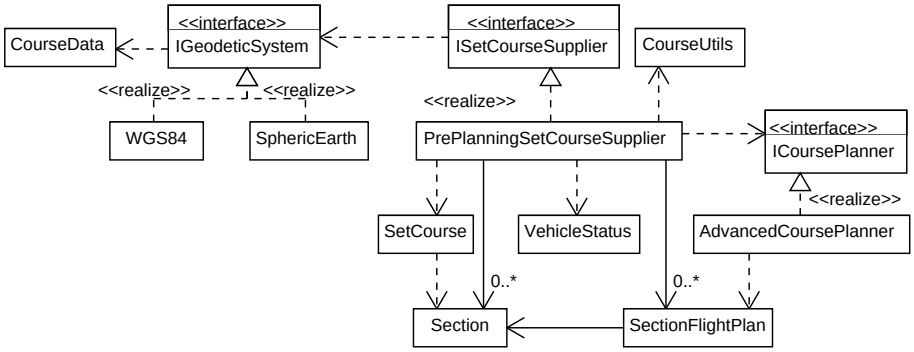

Figure 6.10: Class diagram of Module Course.

Class `VehicleStatus` implements a 5-tuple to describe a vehicle's status. It contains position, total speed, course over ground, elevation, and orientation of the vehicle. Ground is considered as a plain, normal to the nadir-zenith axis. The elevation is the angle between ground and motion vector of the vehicle.

An instance of class `Section` describes one section of a set course, which is the planned linear motion path of the vehicle from one set course way point to the next within a certain time. A section also defines the desired orientation of the vehicle for the start and end positions.

Static class `SetCourse` loads the set course data from an `InputStream` and allows access to it by providing several getter methods. The loading method interprets the course data line by line. A line starting with '#' is considered as a comment and therefore skipped. Empty lines are ignored. Every other line must contain the values for latitude, longitude, altitude, duration, and orientation. Semicolons (';') separate the fields from each other.

Static class `CourseUtils` allows interpolating between two angles by weighting between 0% and 100%. 0% results in the first and 100% in the second angle.

Interface `ICoursePlanner` describes the functionality of a course planner in general. A course planner gets an array of `Section` objects and an instance of interface `IGeodeticSystem` and returns an array of `SectionFlightPlan` objects.

Class `AdvancedCoursePlanner` implements interface `ICoursePlanner`. It splits each section in three subsections of constant velocity or constant acceleration and tries to smoothen the transition between course sections. The planning result is an array of `SectionFlightPlan` objects. Section 5.6 elaborates on the details of the implemented algorithm.

Interface `ISetCourseSupplier` specifies the functionality of a set course supplier, which returns the set course position to a given point in time since the start of an autonomous flight.

Class `PrePlanningSetCourseSupplier` implements interface `ISetCourseSupplier` by pre-planning a set course. It loads the set course data by employing class `SetCourse` and plans the set course by applying an instance of `ICoursePlanner`. A user of this class may query a required `VehicleStatus` to a given point in time. The implementation of class `PrePlanningSetCourseSupplier` will search the corresponding `SectionFlightPlan` object and estimate the required vehicle state. Figure 6.11 visualizes the steps for loading set course data and preparing flight plans.

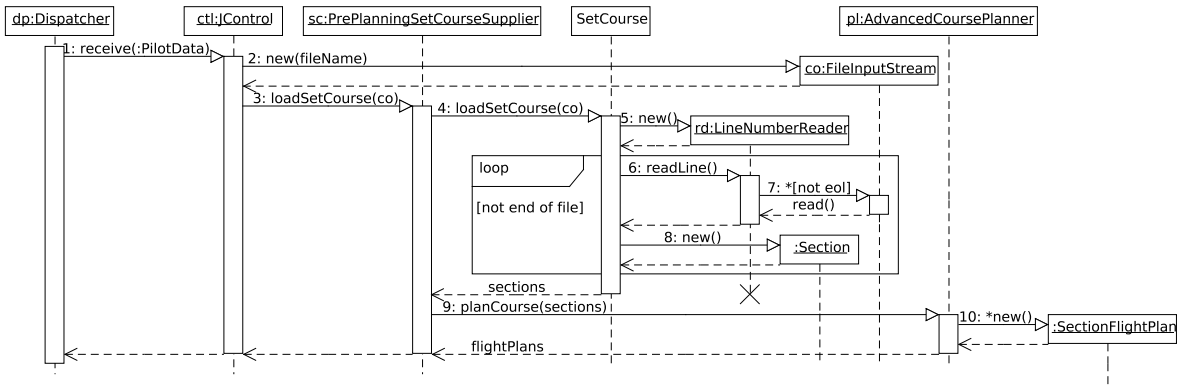

Figure 6.11: Set course data loading and flight plan preparing.

The chronology shown in Figure 6.11 is outlined below.

1. `Dispatcher` *dp* forwards a `PilotData` start message, which contains the path name of the set course data file, to JAviator controller *ctl*.
2. JAviator controller *ctl* opens the file containing the set course data by creating `FileInputStream` *co*.
3. By calling method `loadSetCourse()`, JAviator controller *ctl* instructs `PrePlanningSetCourseSupplier` *sc* to load the set course data.
4. To achieve this, *sc* calls `loadSetCourse()` of static class `SetCourse`.
5. Class `SetCourse` takes input stream *co* and wraps `LineNumberReader` *rd* around it for easier handling.
6. For each line in the set course data file, class `SetCourse` invokes method `readLine()` of reader *rd*.

7. Method `readLine()` of *rd* reads bytes from input stream *co* until it reaches the end of the current line.

8. Class `SetCourse` instantiates a `Section` object for each line read.

9. After successfully reading the set course data file, *sc* plans the course by applying method `planCourse()` of `AdvancedCoursePlanner` *pl* to the `Section` objects.

10. `AdvancedCoursePlanner` *pl* establishes one `SectionFlightPlan` object for each `Section` object.

The sequence discussed does not consider errors. Method `receive()` of JAviator controller *ctl* takes care of exceptions caused by errors in the set course data. In case of errors, *ctl* remains in manual flight control mode.

After successfully loading set course data, JAviator controller *ctl* periodically polls the set course supplier for set course positions, as detailed in Figure 6.12.

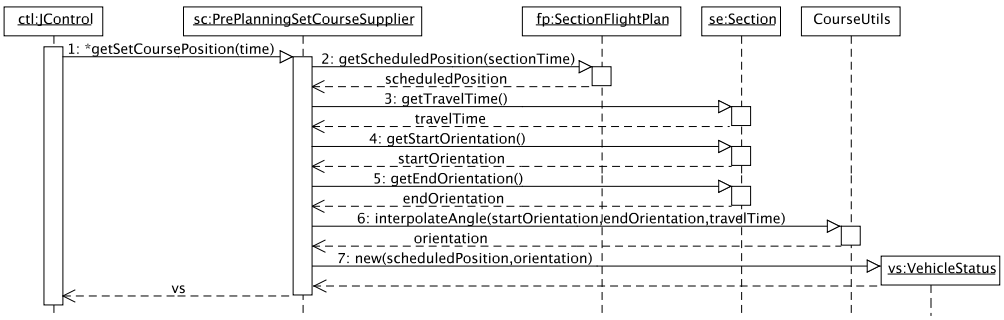

Figure 6.12: Retrieving the set course vehicle state for a given flight time.

In the following, the sequence depicted in Figure 6.12 is explained in detail.

1. JAviator controller *ctl* queries the set course position and orientation corresponding to the current flight *time* by invoking method `getSetCoursePosition()` of `PrePlanningSetCourseSupplier` *sc*.

2. In consideration of the current flight time, *sc* searches the corresponding `SectionFlightPlan` fp and computes the flight time from the beginning of plan fp. With this time as parameter, *sc* invokes `getScheduledPosition()` of plan fp to get the new set course position.

3. To get the travel time of the current set course section, *sc* calls method `getTravelTime()` of the associated `Section` *se*.

4. Then, *sc* determines the vehicle's orientation at section start by accessing method `getStartOrientation()` of *se*.

5. Thereafter, *sc* retrieves the vehicle's orientation at section end by invoking method `getEndOrientation()` of *se*.

6. In order to get the vehicle's current orientation nominal value, *sc* executes method `interpolateAngle()` of static class `CourseUtils`.

7. With the new set course position and the orientation nominal value, *sc* creates a new `VehicleStatus` object *vs*, and returns it to JAviator controller *ctl*.

Method `getSetCoursePosition()` of class `PrePlanningSetCourseSupplier` caches the index to `SectionFlightPlan` *fp* and `Section` *se* for later use. Subsequent invocations of method `getSetCoursePosition()` will use this index as a start to find the fitting instances of classes `SectionFlightPlan` and `Section`. With the current JAviator controller cycle time of 20 ms and section travel time spans of at least several seconds, the correct index value will be the same one as the last in nearly 100% of all invocations. If the flight time of a section ends, the new index will be the current index plus one to get the eligible flight plan and section objects. Hence, the execution time of method `getSetCoursePosition()` has a constant upper limit independent of the number of set course sections.

6.3.5 Module GPS

This module implements the range of functions to handle the NMEA 0183 [1] messages of a GPS receiver, the functionality to forward available RTCM SC-104 [2] correction data streams to the same GPS receiver, and the functionality necessary to simulate a GPS receiver.

Module *GPS* comprises one Java archive, which exports classes and interfaces of general GPS functionality, as well as classes and interfaces to handle NMEA 0183 messages, RTCM SC-104 correction data streams, and GPS simulation.

Figure 6.13 shows the main classes and interfaces of Module *GPS*. Instances of class `GpsPositionProvider` receive NMEA 0183 Course Over Ground and Ground Speed (VTG), Global Positioning System Fixed Data (GGA), and Recommended Minimum Specific GNSS Data (RMC) messages as `Nmea0183Message` objects from instances of class `Nmea0183MessageProvider` and convert the received positions to objects of type `PolarCoordinate`. Interested objects may retrieve the current vehicle location from a position provider by accessing the `IPositionProvider` interface.

Class `GpsAdapter` extends class `GpsPositionProvider` by adding initialization code that allows construction from given properties. To achieve this, class `GpsAdapter` employs classes `GpsDaemonBuilder` and `GpsDaemon`.

An instance of class `GpsDaemon` runs as a separate thread that connects to a GPS receiver by means of an `IConnection` link, parses the incoming bytes, and routes detected NMEA 0183 messages as `Nmea0183Message` objects to registered instances of interface `Nmea0183MessageListener`. Interested objects use the `Nmea0183MessageProvider` interface for registering with the daemon. If available, the daemon forwards RTCM SC-104 correction data messages to the attached GPS receiver.

As depicted in Figure 6.14, the initialization code of class `GpsDaemonBuilder` instantiates one `GpsDaemon` and, optionally, one RTCM SC-104 message provider from given properties by applying the `ObjectFactory`.

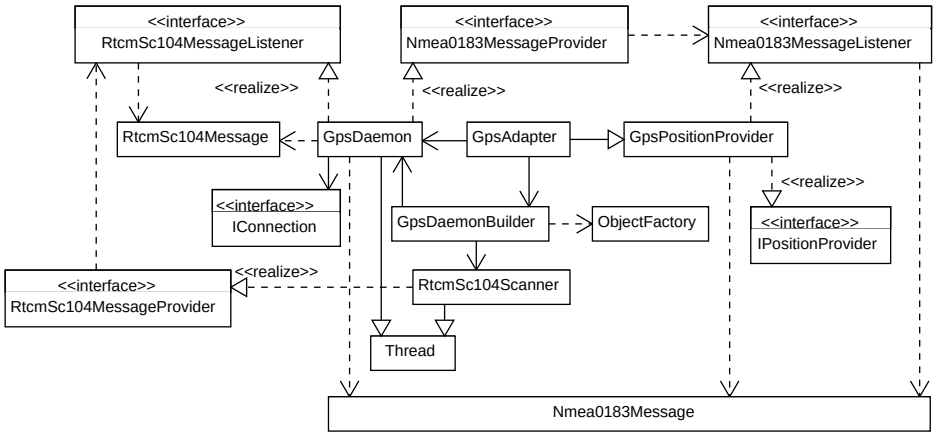

Figure 6.13: Class diagram of Module GPS.

An instance of class `RtcmSc104Scanner` runs as a separate thread that reads RTCM SC-104 correction data from a provided `InputStream`. After receiving a complete correction data packet, the scanner converts it to an object of type `RtcmSc104Message`. Subsequently, the scanner forwards this message object to registered instances of class `RtcmSc104MessageListener`. Interested objects register with the scanner by accessing its `RtcmSc104MessageProvider` interface.

The chronology shown in Figure 6.14 is outlined below.

1. `GpsAdapter` *ga* instantiates object *db* of class `GpsDaemonBuilder`.
2. `GpsDaemonBuilder` *db* invokes the `instantiateConnection()` method of `ObjectFactory` *of* to open a connection to the GPS receiver.
3. `ObjectFactory` *of* creates a new `IConnection` object *gps* and returns the reference to it to `GpsDaemonBuilder` *db*.
4. `GpsDaemonBuilder` *db* constructs a new `GpsDaemon` object *gpsd*.
5. `GpsDaemonBuilder` *db* starts the *gpsd* thread, which immediately scans the data available from the GPS receiver *gps*.
6. If the configuration supports parameters of a RTCM SC-104 correction data stream, `GpsDaemonBuilder` *db* calls the `instantiateInputstream()` method of `ObjectFactory` *of* to connect to this data stream.
7. `ObjectFactory` *of* creates a new `InputStream` object *rtcm* and returns the reference to it to `GpsDaemonBuilder` *db*.
8. If the correction data stream needs the approximate location of the GPS receiver before sending correction data, it also implements the `Nmea0183MessageListener` interface. In this case `GpsDaemonBuilder` *db* registers stream *rtcm* with GPS daemon *gpsd*.

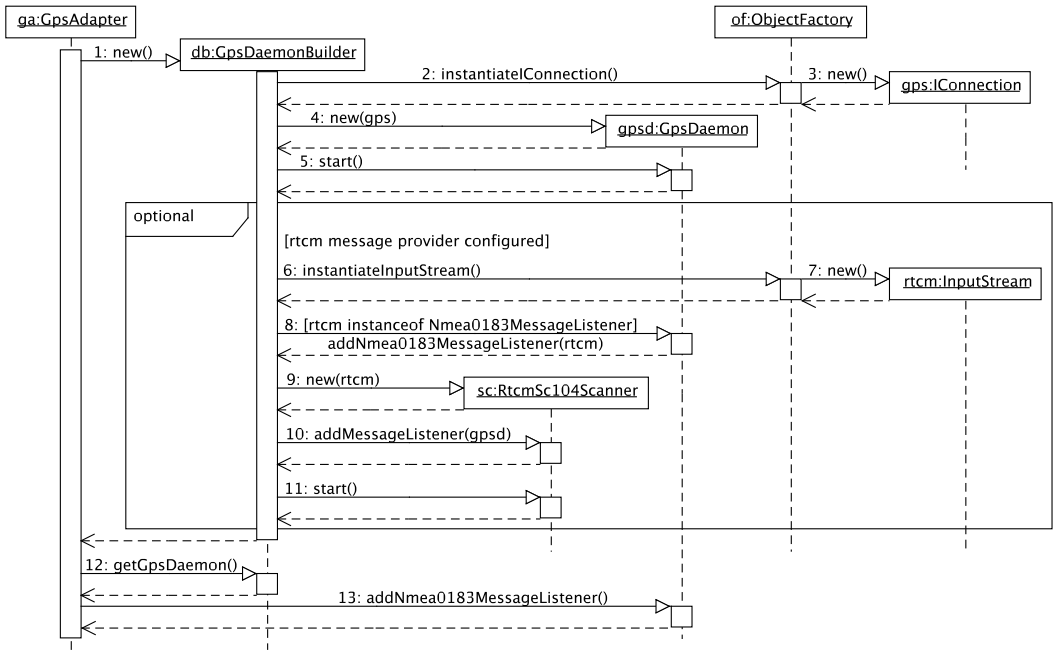

Figure 6.14: Initialization sequence of class `GpsAdapter`.

9. `GpsDaemonBuilder` *db* instantiates object *sc* of class `RtcmSc104Scanner`.
10. `GpsDaemonBuilder` *db* registers `GpsDaemon` *gpsd* with scanner *sc*.
11. `GpsDaemonBuilder` *db* starts the *sc* scanner thread, which forwards the correction data as `RtcmSc104Message` objects to GPS daemon *gpsd*.
12. `GpsAdapter` *ga* retrieves a reference to GPS daemon *gpsd* for further usage.
13. `GpsAdapter` *ga* registers itself with GPS daemon *gpsd*.

The sequence discussed shows how the connection to a GPS receiver and an optional RTCM SC-104 correction data stream is initialized.

After initialization, the GPS daemon parses the data provided from the GPS receiver and forwards it as `Nmea0183Message` objects to registered listeners, as visualized in Figure 6.15. The series of invocations shown in this picture follows in detail.

1. GPS daemon *gpsd* calls its own `readLine()` method to retrieve the next message from GPS receiver *gps*.
2. Method `readLine()` reads bytes from GPS receiver `InputStream` *gps* until the *end of line* characters arrive, that is, character *carriage return* followed by character *linefeed*.
3. With the received sequence of bytes as parameter, GPS daemon *gpsd* creates a new `Nmea0183Message` object *msg*.

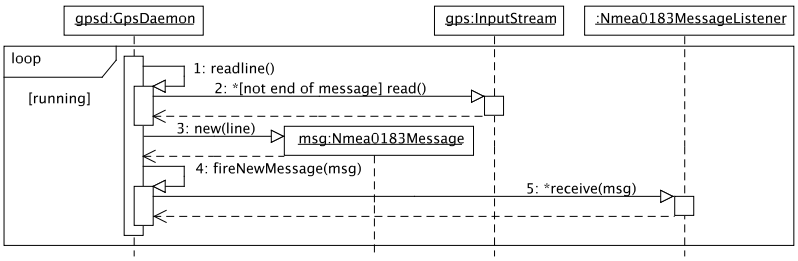

Figure 6.15: `GpsDaemon` forwards NMEA 0183 messages to registered listeners.

4. GPS daemon *gpsd* calls its own `fireNewMessage()` method.

5. Method `fireNewMessage()` distributes newly arrived message *msg* to all registered listeners by calling method `receive()` of their implemented interface `Nmea0183MessageListener`.

If the received sequence of bytes does not represent a well-formed NMEA 0183 message, the constructor of class `Nmea0183Message` throws an exception, which the main loop of GPS daemon *gpsd* catches. Thereafter, *gpsd* continues reading messages from GPS receiver *gps*.

Figure 6.16 views how a `RtcmSc104Scanner` instance reads RTCM SC-104 messages from a correction data stream and forwards them to registered listeners that implement interface `RtcmSc104Listener`.

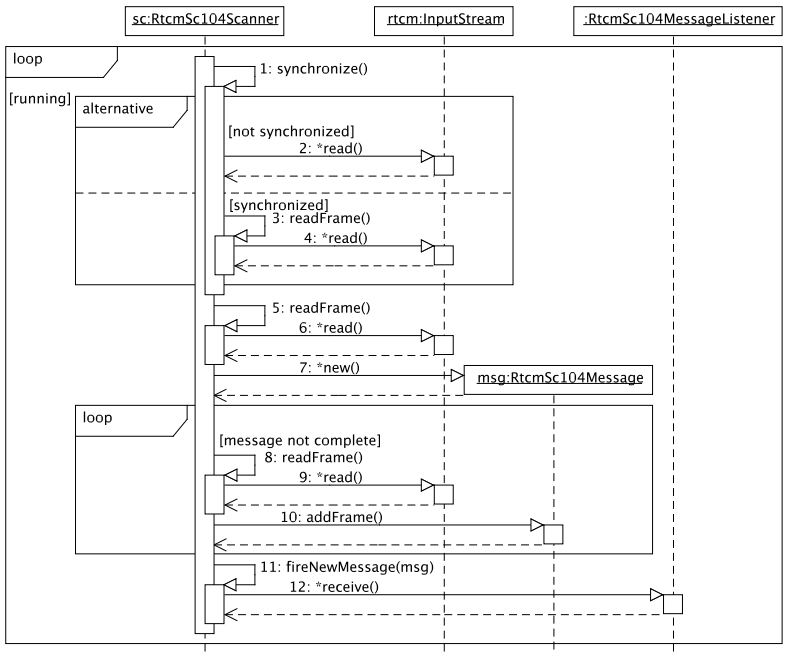

Figure 6.16: `RtcmSc104Scanner` forwards RTCM SC-104 messages to listeners.

In the following, the sequence shown in Figure 6.16 is explained in more detail.

1. `RtcmSc104Scanner` *sc* calls its `synchronize()` method to get in sync with the correction data stream.
2. If scanner *sc* is not yet in sync with correction data stream *rtcm*, method `synchronize()` repeatedly reads bytes from the stream and tries to find the preamble of a RTCM SC-104 message. Then, method `synchronize()` reads the rest of the first message frame and returns.
3. If the scanner is already synchron to the correction data stream it reads the next message frame, which has to be a message header, by calling its `readFrame()` method.
4. Method `readFrame()` reads a complete frame from correction data stream *rtcm*.
5. With the first frame available, scanner *sc* reads the second frame of the message by calling its `readFrame()` method.
6. Method `readFrame()` reads a complete frame from correction data stream *rtcm*.
7. Now the first two frames of the message are available. These two frames enable scanner *sc* to determine the message type and length, thus scanner *sc* creates a new `RtcmSc104Message` object *msg*.
8. Scanner *sc* reads the remaining frames of the message from stream *rtcm* by repeatedly calling its `readFrame()` method.
9. Method `readFrame()` reads a complete frame from correction data stream *rtcm*.
10. Scanner *sc* adds the newly read message frame to `RtcmSc104Message` *msg*.
11. Scanner *sc* calls its method `fireNewMessage()` for every newly received and well-formed RTCM SC-104 message.
12. Method `fireNewMessage()` sends message *msg* to all registered listeners by calling method `receive()` of their interface `RtcmSc104MessageListener`.

Whenever the `RtcmSc104Scanner` instance recognizes parity errors, it assumes not to be in sync with the correction data stream and tries to re-synchronize.

Figure 6.17 illustrates how a `GpsDaemon` instance forwards received `RtcmSc104Message` objects to the GPS receiver.

The procession depicted in Figure 6.17 is in detail as follows.

1. `RtcmSc104Scanner` *sc* sends a new RTCM SC-104 message to `GpsDaemon` *gpsd* by invoking method `receive()` of its interface `RtcmSc104MessageListener`.
2. `GpsDaemon` *gpsd* calls method `getBytes()` of message *msg* to retrieve its content as an array of bytes.

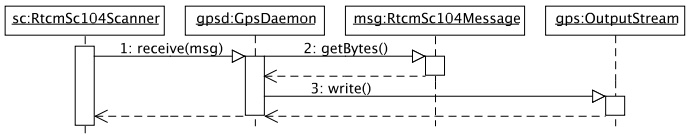

Figure 6.17: `GpsDaemon` forwards RTCM SC-104 messages to the GPS receiver.

3. `GpsDaemon` *gpsd* writes this array of bytes at once to `OutputStream` *gps* that represents the GPS receiver's input for correction data.

The Austrian Positioning Service (APOS) [8] of the Austrian Bundesamt für Eich- und Vermessungswesen (BEV) provides RTCM SC-104 [2] correction data streams over GSM or Mobile Internet (NTRIP [4] via General Packet Radio Service (GPRS)/ Universal Mobile Telecommunication System (UMTS)). To keep the particularities of APOS hidden from the `RtcmSc104Scanner`, the classes exposed in Figure 6.18 provide the APOS correction data stream as an object of type `InputStream`.

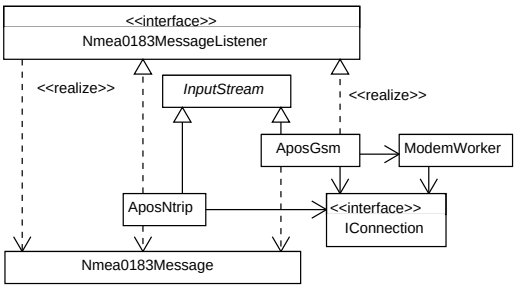

Figure 6.18: Diagram of the classes accessing the APOS [8] service.

Class `AposNtrip` is responsible for accessing the NTRIP service of APOS. When calling method `read()` of an `AposNtrip` instance demands the first byte of correction data, this method blocks until connecting to the NTRIP service succeeds and the first byte of correction data is available. `AposNtrip` implements the `Nmea0183MessageListener` interface to receive the approximate current vehicle position from the GPS receiver, which the APOS service requires before sending correction data. `AposNtrip` establishes the connection to APOS after the approximate current vehicle position is available. Figure 6.19 shows the outlined behavior in detail.

The following describes the sequence depicted in Figure 6.19.

1. `RtcmSc104Scanner` *sc* calls method `read()` of the `InputStream` interface implemented by `AposNtrip` object *rtcm*.

2. If not yet connected to an NTRIP data stream, `AposNtrip` object *rtcm* calls its `connect()` method.

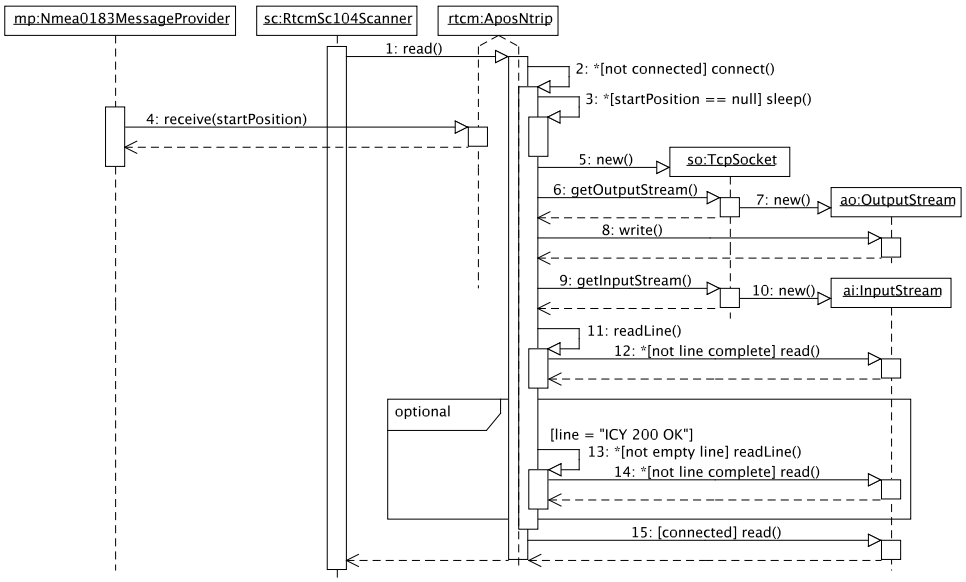

Figure 6.19: APOS NTRIP connection establishment.

3. Method `connect()` waits until the approximate location of the GPS receiver is available.

4. `Nmea0183MessageProvider` *mp*, that is, the `GpsDaemon` instance, hands over the current approximate location of the GPS receiver to `AposNtrip` object *rtcm* by calling method `receive()` of its interface `Nmea0183MessageListener`.

5. Thereafter, `AposNtrip` instance *rtcm* connects to the NTRIP data stream by creating a new `TcpSocket` *so*.

6. `AposNtrip` object *rtcm* requests of `TcpSocket` *so* its associated `OutputStream` object.

7. `TcpSocket` *so* creates a new `OutputStream` *ao* and returns it.

8. `AposNtrip` instance *rtcm* assembles the NTRIP request to initialize the correction data stream and writes the entire request to `OutputStream` *ao* of the TCP/IP connection.

9. Now, `AposNtrip` object *rtcm* queries of `TcpSocket` *so* its associated `InputStream` object.

10. `TcpSocket` *so* creates a new `InputStream` object *ai* and returns it.

11. `AposNtrip` object *rtcm* reads the first line of the NTRIP data stream by calling its own `readLine()` method.

12. Method `readLine()` reads bytes from `InputStream` *ai* until it reaches the end of the current line.

13. If the retrieved line does not equal to `"ICY 200 OK"`, method `connect()` returns to method `read()`, which tries to reconnect. If the retrieved line equals to `"ICY 200 OK"`, connecting to the NTRIP data stream succeeded. Method `connect()` repeatedly calls method `readLine()`, to read all header lines sent from the NTRIP server until an empty line arrives.

14. Method `readLine()` reads bytes from `InputStream` *ai* until it reaches the end of the current line.

15. The transmission of correction data starts with the next byte read from `InputStream` *ai*. `AposNtrip` object *rtcm* reads a byte and returns it to the `RtcmSc104Scanner` *sc*.

The sequence discussed ensures that a `RtcmSc104Scanner` instance gets RTCM SC-104 correction data from the APOS NTRIP caster only if the approximate location of the GPS receiver is available and the connection to the caster succeeded.

Class `AposGsm` works similar to `AposNtrip`, but establishes the connection to APOS via a GSM mobile phone modem accessed by serial line or Bluetooth socket. To access the modem, `AposGsm` employs instances of class `ModemWorker`.

Figure 6.20 displays the diagram of the classes involved in simulating GPS messages. Class `GpsReceiverSimulator` estimates new coordinates by reading position data from an instance of `IPositionProvider` and modifying this data with inaccuracies. In order

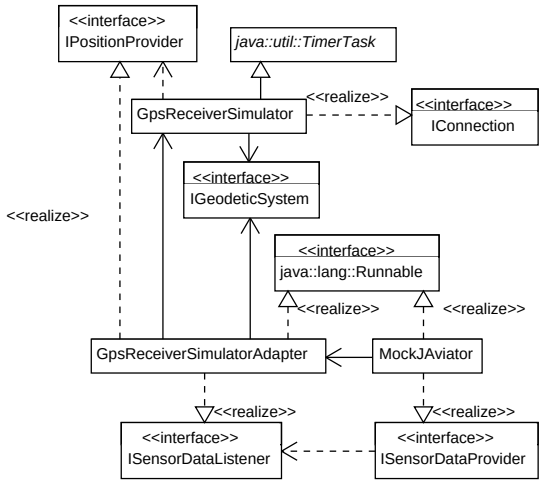

Figure 6.20: Class diagram of the GPS simulation.

to simulate satellite coherent inaccuracies, the simulator utilizes deviation information from recorded GPS position data. Class `GpsReceiverSimulator` converts newly estimated coordinate, course, and velocity data into NMEA 0183 messages, which it caches in an internal ring buffer. Class `GpsReceiverSimulator` implements interface `IConnection` to grant access to the buffered messages by means of input streams. By doing so, the simulator mimics the behavior of an opened connection to a GPS receiver. Users may read the messages as quick as they like, because the input stream picks up

a message from the ring buffer and delivers it byte-by-byte until the whole message has been sent. Slow readers will loose messages this way, but they will always get complete messages. Every time the instance of the `MockJAviator` gets new actuator data from the FCS, it notifies registered `ISensorDataListener` objects of new sensor data. An instance of class `GpsReceiverSimulatorAdapter` is one of the registered listeners, which runs as a separate thread. It converts received sensor data packets to WGS 84 [33] coordinates and publishes them via the `IPositionProvider` interface to the GPS receiver simulator. Additionally, class `GpsReceiverSimulatorAdapter` implements a TCP/IP server that forwards the data stream of the GPS receiver simulator to connected TCP/IP clients. Figure 6.21 shows the described functionality.

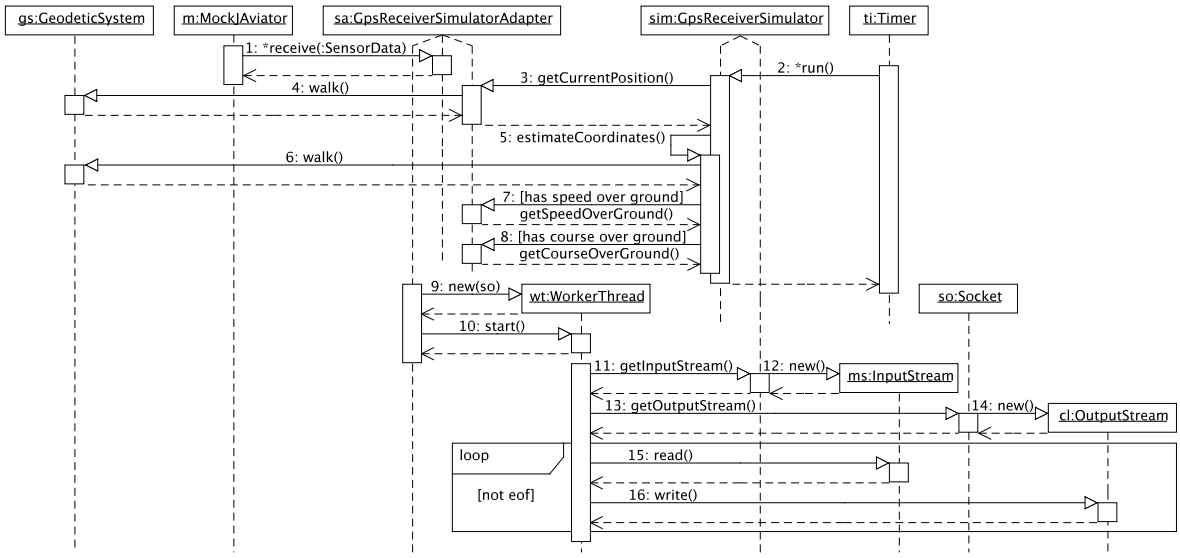

Figure 6.21: GPS receiver simulator scenario.

The following details on the sequence depicted in Figure 6.21.

1. `MockJAviator` *m* cyclically sends to `GpsReceiverSimulatorAdapter` *sa* `SensorData` objects, which contain *x*-, *y*-, and *z*-coordinates of the simulated JAviator Plant, as well as their first derivatives. Additionally, the `SensorData` objects convey IMU provided values, especially the orientation of the vehicle over ground.

2. `Timer` *ti* invokes method `run()` of `GpsReceiverSimulator` *sim* each time it should generate a new set of NMEA 0183 messages.

3. By calling method `getCurrentPosition()`, `GpsReceiverSimulator` *sim* queries the current location from `GpsReceiverSimulatorAdapter` *sa*.

4. `GpsReceiverSimulatorAdapter` *sa* uses a configured reference position as an origin. It calculates the current position by adding the content of the recently received `SensorData` object to the origin by employing `IGeodeticSystem` *gs*.

5. `GpsReceiverSimulator` *sim* prepares the strings necessary to compose the NMEA 0183 messages by calling its method `estimateCoordinates()`.

6. By invoking method `walk()` of the current `IGeodeticSystem` instance *gs*, `GpsReceiverSimulator` *sim* applies recorded deviations to the position fetched from `GpsReceiverSimulatorAdapter` *sa*.

7. If available, `GpsReceiverSimulator` *sim* queries the current speed over ground by calling `GpsReceiverSimulatorAdapter` *sa*'s method `getSpeedOverGround()`. `GpsReceiverSimulator` *sim* will simulate the value of the current speed over ground, if necessary.

8. `GpsReceiverSimulator` *sim* queries the current course over ground by invoking `GpsReceiverSimulatorAdapter` *sa*'s method `getCourseOverGround()`. If not available, `GpsReceiverSimulator` *sim* will simulate the value of the current course over ground. Thereafter, method `run()` assembles a new set of NMEA 0183 messages and caches them in an internal ring buffer of limited length. Always the oldest messages in this ring buffer will be overwritten by new ones.

9. For every connecting TCP/IP client, `GpsReceiverSimulatorAdapter` *sa* creates a separate `WorkerThread` *wt*.

10. Subsequently, `GpsReceiverSimulatorAdapter` *sa* starts new thread *wt*.

11. To gain access to the simulated NMEA 0183 messages, `WorkerThread` *wt* requests a new `InputStream` from `GpsReceiverSimulator` *sim* by invoking method `getInputStream()`.

12. Method `getInputStream()` instantiates `InputStream` *ms* that has access to the ring buffer cache of simulator *sim*.

13. Then, `WorkerThread` *wt* queries the `OutputStream` of `Socket` *so* by calling `getOutputStream()`.

14. Method `getOutputStream()` creates `OutputStream` *cl*.

15. `WorkerThread` *wt* reads one byte from `InputStream` *ms*.

16. `WorkerThread` *wt* writes the newly read byte to `OutputStream` *cl*. Now *wt* jumps back to Step 15 to read the next byte, as long as *ms* produces new bytes.

The sequence discussed ensures that new NMEA 0183 messages are available at each time `Timer` *ti* invokes `GpsReceiverSimulator` *sim*'s method `run()`, which always uses the most recently received `SensorData` object for estimations. The sequence above also ensures that an arbitrary number of clients can access the NMEA 0183 messages via TCP/IP. The usage of a ring buffer cache allows clients to read messages at any speed they like without interfering with the creation of messages or the other clients. However, slow clients may miss messages.

6.3.6 Module Location

This module converts linear coordinates provided by non-GPS positioning systems to WGS 84 [33] coordinates, by applying Equation (4.4). Moreover, it calculates course and speed over ground. A close-range location system must support coordinates in the form of sentences. The general sentence format is:

```
$hhhhhh,d1,d2,...*ss<CR><LF>
```

The first six letters succeeding the $ are the message protocol header, which is followed by a number of data fields, separated by commas, and a checksum. The delimiter between the last data field and the checksum is an asterisk (*). Each sentence ends with *carriage return/linefeed*. The checksum is the exclusive OR of all characters beginning with the $ up to the asterisk. Table 6.1 describes the current implemented message format for receiving position updates from close-range location systems.

Name	**Example**	**Units**	**Description**
$			Start of sentence
Message ID	LOCPNQ		Message protocol header
Tag type	ULocationIntegration::Tag		Location sensor type
Tag ID	00000000000020000021176		Location sensor ID
UTC time	2009-05-24 00:08:50.462		
Precision	0.00849146	meters	standard deviation
Status	1		0=data not valid or 1=data valid
x	0.117011	meters	X coordinate
y	3.18593	meters	Y coordinate
z	0.84653	meters	Z coordinate
a	1.0	meters	Quaternion real part
b	0	meters	Quaternion imaginary part
c	0	meters	Quaternion imaginary part
d	0	meters	Quaternion imaginary part
Checksum	4D		
<CR><LF>			End of message termination

Table 6.1: Message format for attaching close-range location systems.

Module *Location* requires the location of two sensors to calculate the course. The message format described in Table 6.1 allows conveying a quaternion for the movement direction, but handling this is part of future enhancements.

Module *Location* comprises one Java archive, which exports the classes and interfaces `LocationMessage`, `ILocationMessateProvider`, `ILocationMessageListener`, `LocationDaemon`, and `PositionProvider`.

Figure 6.22 displays the classes and interfaces of Module *Location*. An instance of class `LocationDaemon` is responsible for receiving position updates from a close-range location system and converting correct updates into `LocationMessage` objects, which it

forwards to registered `ILocationMessageListener` instances. Interested objects employ the `ILocationMessateProvider` interface to register with the daemon. Instances of class `PositionProvider` receive the position updates of two different location sen-

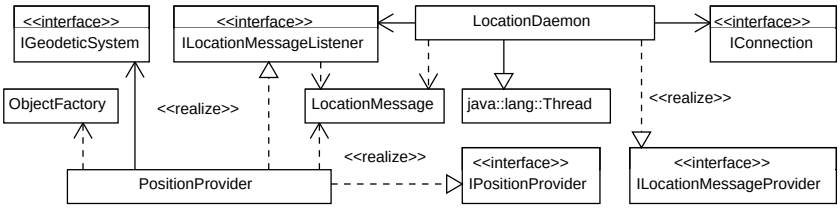

Figure 6.22: Class diagram of Module Location.

sors from the `LocationDaemon` and calculate course and speed over ground, as well as the mean position between the location sensors. Other objects may poll the current position, course, and speed via interface `IPositionProvider`. Class `PositionProvider` uses an instance of interface `IGeodeticSystem` for the calculations described earlier, which it instantiates by means of static class `ObjectFactory`.

6.3.7 Module User Interface

This module implements a graphical user interface to visualize information that is not shown in the GCS yet. It receives NMEA 0183 messages from the GPS receiver and displays the current values of position, course over ground, speed over ground, GPS time, as well as a birds-eye view of the trajectory. The design of this module allows running the graphical user interface in a separate JVM, irrespective of the GCS.

Module *User Interface* comprises one Java archive, which exports the classes and interfaces `Altimeter`, `BirdsEyeFrame`, `BirdsEyeView`, `Clock`, `Compass`, `IAltitudeView`, `ICoordinateView`, `ICourseView`, `INavigatorView`, `ISatelliteView`, `ISpeedView`, `ITimeView`, `Nmea0183MessageForwarder`, `NavigationFrame`, `NavigationMain`, `Speedometer`.

Figure 6.23 shows the diagram of the classes and interfaces of Module *User Interface*. Class `NavigationMain` is the master routine of this module, which implements a static `main()` method. When started as a program, it creates an instance of class `GpsDaemon` by means of class `GpsDaemonBuilder`. Additionally, it creates instances of classes `BirdsEyeFrame` and `NavigationFrame`. The instance of `GpsDaemon` connects to the configured GPS receiver and forwards received NMEA 0183 messages to registered instances of interface `Nmea0183MessageListener`. An instance of class `Nmea0183MessageForwarder` receives the NMEA 0183 messages and distributes the content among instances of interface `INavigatorView`. The instantiated class `Nmea0183MessageForwarder` knows, which of the interfaces derived from `INavigatorView` a registered instance implements. The `Nmea0183MessageForwarder` sends polar coordinates to instances of `ICoordinateView`, altitude values to instances of `IAltitudeView`, speed values to instances of `ISpeedView`, course values to instances

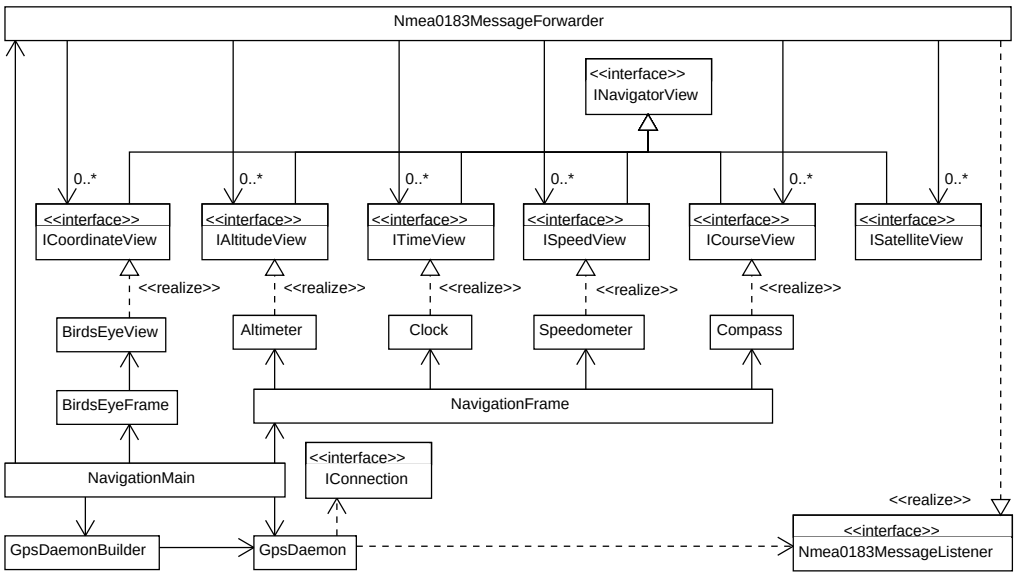

Figure 6.23: Class diagram of Module User Interface.

of `ICourseView`, and satellite updates to instances of `ISatelliteView`. To instances of `ITimeView` and to all other descendants of interface `INavigatorView` the instantiated `Nmea0183MessageForwarder` sends the timestamp of the corresponding NMEA 0183 message.

An instance of `BirdsEyeFrame` embeds one instance of class `BirdsEyeView`, which displays the last 100 position updates of the trajectory as a set of dots. Instances of `NavigationFrame` visualize instances of classes `Altimeter`, `Clock`, `Speedometer`, and `Compass`. It is the running `NavigationMain` that registers the instantiated views of `BirdsEyeFrame` and `NavigationFrame` with the `Nmea0183MessageForwarder` instance. Hence, the frames and views are completely unaware of the providers of the data they are visualizing.

6.3.8 Module JAviator Control

This module implements the controllers for manual and autonomous flights of the JAviator, as developed in Chapter 5. It is the main module of the FCS running in the JVM on the JAviator's Gumstix, together with Modules *Input/Output*, *Utilities*, *Communication*, *Course*, *GPS*, and *Location*.

Module *JAviator Control* comprises one Java archive, which exports the classes and interfaces `IControlAlgorithm`, `IController`, `JControl`, `JControlMain`, `PDDController`, and `PositionControlAlgorithm`.

Figure 6.24 visualizes the classes and interfaces of Module *JAviator Control*. Class `JControlMain` implements the `main()` routine to run the FCS. At startup it instantiates a `Dispatcher` for dispatching messages, an `ISetCourseSupplier` to handle set

courses, an `IPositionProvider` for querying the current position, a `JControl` object for flight control, a `TransceiverAdapter` to connect to the Plant, and a `TcpServer` for incoming TCP/IP connections from the GCS. Thereafter, `JControlMain` registers the instances of `TransceiverAdapter`, `JControl`, and `TcpServer` with the instance of `Dispatcher`. Finally, `JControlMain` spawns the `TcpServer` object off as a separate thread and starts the timer to periodically invoke the `JControl run()` method. Class `JControl` is responsible for instantiating the configured `IControlAlgorithm`.

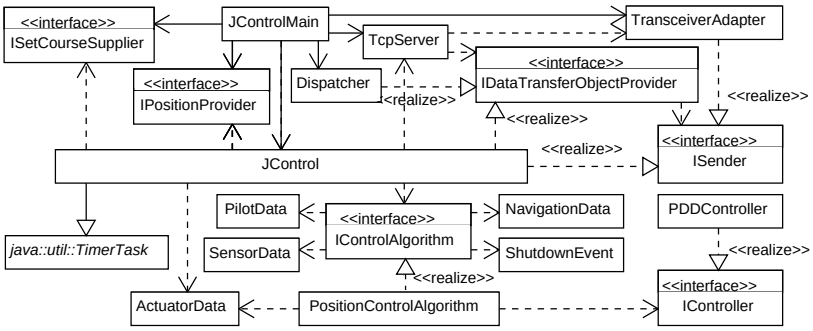

Figure 6.24: Class diagram of Module JAviator Control.

Instances of class `JControl` receive `SensorData` objects from the Plant, as well as `NavigationData`, `PilotData`, and `ShutdownEvent` objects from the GCS. When performing manual flights, the `JControl` instance forwards the `NavigationData` and `SensorData` objects to the `IControlAlgorithm` instance, which returns the new motor nominal values as an `ActuatorData` object. The GCS sends `PilotData` objects to switch from manual to autonomous flights and vice versa. In autonomous-flight mode, the `JControl` instance forwards sensor data, position, course over ground, speed over ground, and set course data to the `IControlAlgorithm` instance, which also returns an `ActuatorData` object. Every time the timer invokes the `run()` method, the `JControl` instance sends the current `ActuatorData` object to the `Dispatcher` instance, which routes it to the Plant.

Class `PositionControlAlgorithm` is currently the only implementation of interface `IControlAlgorithm`. It is this class that realizes the controllers depicted in Figure 5.1 by creating `IController` instances for the roll, pitch, yaw, altitude, x position, and y position as required by its configuration.

Class `PDDController` realizes interface `IController` by implementing a PD^2 controller. It estimates the second derivative of the control variable internally, if not available. The implementation allows configuring a controller saturation of both proportional and derivative parts.

Figure 6.25 shows the initialization sequence of class `JControlMain`, which the following enumeration explains in more detail.

1. JAviator control main class `JControlMain` instance *cm* creates a new `Dispatcher` object *dp*.

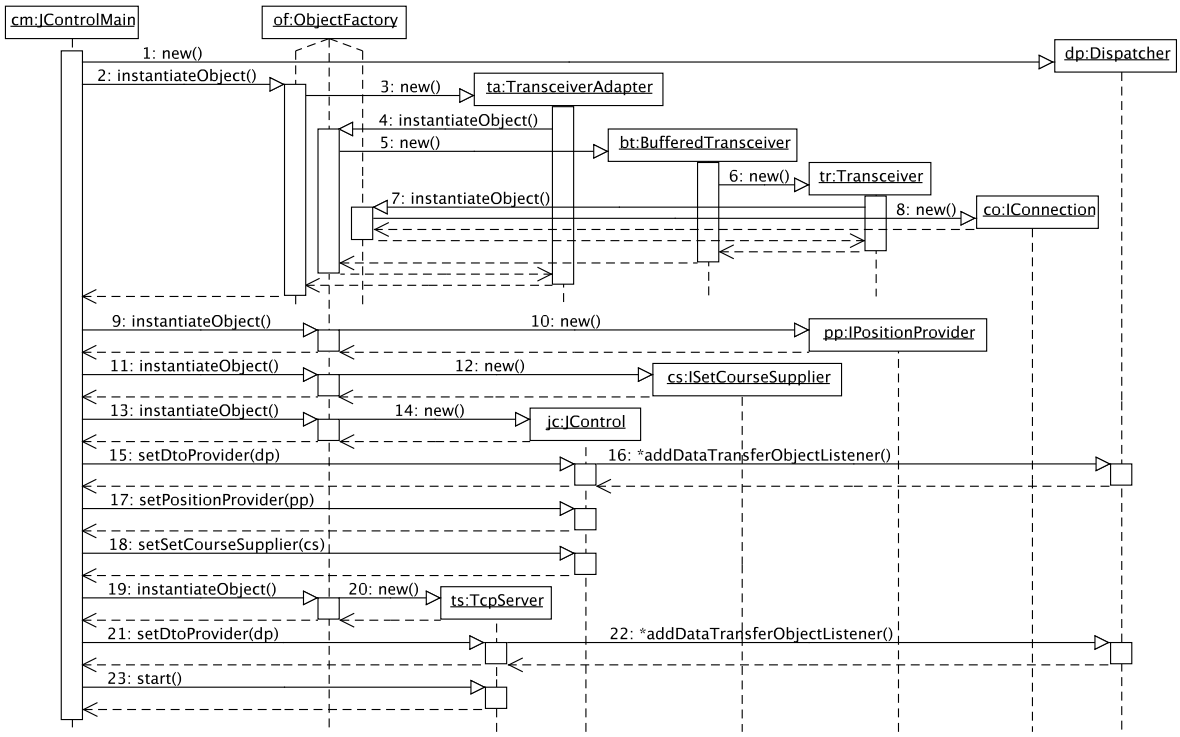

Figure 6.25: Initialization sequence of class `JControlMain`.

2. By employing `ObjectFactory` *of*, `JControlMain` *cm* creates a new instance of class `TransceiverAdapter` to connect to the JAviator Plant.

3. `ObjectFactory` *of* instantiates a new `TransceiverAdapter` *ta*.

4. `TransceiverAdapter` *ta* uses `ObjectFactory` *of* to construct a new instance of class `BufferedTransceiver`.

5. `ObjectFactory` *of* creates a new `BufferedTransceiver` *bt*.

6. `BufferedTransceiver` *bt* instantiates its underlying `Transceiver` *tr*.

7. `Transceiver` *tr* opens the connection to the JAviator Plant by calling method `instantiateObject()` of `ObjectFactory` *of*.

8. `ObjectFactory` *of* constructs a new `IConnection` object *co* that represents the connection to the JAviator Plant.

9. `JControlMain` *cm* instantiates the configured position provider by calling method `instantiateObject()` of `ObjectFactory` *of*.

10. `ObjectFactory` *of* creates the requested position provider *pp*.

11. `JControlMain` *cm* constructs the configured set course supplier by calling method `instantiateObject()` of `ObjectFactory` *of*.

12. `ObjectFactory` *of* creates the required set course supplier *cs*.

13. `JControlMain` *cm* builds the configured JAviator controller by calling method `instantiateObject()` of `ObjectFactory` *of*.
14. `ObjectFactory` *of* instantiates JAviator controller *jc*.
15. `JControlMain` *cm* registers `Dispatcher` *dp* with `JControl` *jc*.
16. `JControl` *jc* registers with `Dispatcher` *dp* for all DTO types it needs to receive.
17. `JControlMain` *cm* registers `IPositionProvider` *pp* with `JControl` *jc*.
18. `JControlMain` *cm* registers `ISetCourseSupplier` *cs* with `JControl` *jc*.
19. `JControlMain` *cm* builds the configured `TcpServer` instance by calling `ObjectFactory` *of*'s method `instantiateObject()`.
20. `ObjectFactory` *of* constructs the requested `TcpServer` object.
21. `JControlMain` *cm* registers `Dispatcher` *dp* with `TcpServer` *ts*.
22. `TcpServer` *ts* registers with `Dispatcher` *dp* for all DTO types it needs to forward.
23. `JControlMain` *cm* starts `TcpServer` *ts* as a separate thread.

The sequence discussed initializes all objects necessary to control the JAviator Plant for manual and autonomous flights.

Figure 6.26 depicts the sequence for manual and autonomous flight control.

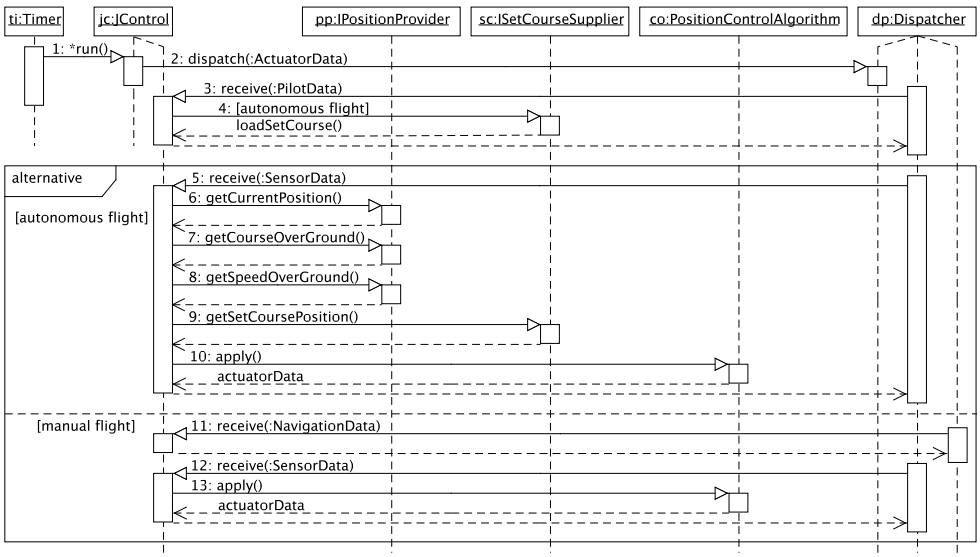

Figure 6.26: Flight control sequence.

In the following, the sequence shown Figure 6.26 is explained in more detail.

1. `Timer` *ti* periodically invokes method `run()` of `JControl` object *jc*. The invocations cycle time of 20 ms is part of the configuration.
2. `JControl` instance *jc* sends the currently available actuator data, that is, the revolution speed nominal values for the Plant's motors, to the `Dispatcher` *dp* for transmission to the JAviator Plant.
3. `Dispatcher` *dp* delivers a `PilotData` DTO to `JControl` object *jc*.
4. If the received `PilotData` DTO is a message to start an autonomous flight, `JControl` object *jc* instructs `ISetCourseSupplier` instance *sc* to load the corresponding set course. If *sc* succeeds in loading the requested set course, *jc* switches to autonomous-flight mode.
5. Steps 5 to 10 apply to autonomous flight control only. `Dispatcher` *dp* forwards a `SensorData` DTO to `JControl` instance *jc*.
6. `JControl` instance *jc* queries the current location by calling method `getCurrenPosition()` of `IPositionProvider` *pp*.
7. `JControl` instance *jc* retrieves the current course over ground by invoking method `getCourseOverGround()` of `IPositionProvider` *pp*.
8. `JControl` instance *jc* determines the current speed over ground by accessing method `getSpeedOverGround()` of `IPositionProvider` *pp*.
9. `JControl` instance *jc* ascertains the current set course position by calling method `getSetCoursePosition()` of `ISetCourseSupplier` *sc*.
10. `JControl` instance *jc* calculates new actuator data by applying both the current and the desired values, retrieved for position, speed, and course, to `PositionControlAlgorithm` *co*.
11. Steps 11 to 13 apply to manual flight control only. `Dispatcher` *dp* delivers a `NavigationData` DTO to `JControl` instance *jc*. Object *jc* caches the DTO for further use.
12. `Dispatcher` *dp* routes a `SensorData` DTO to `JControl` instance *jc*.
13. `JControl` instance *jc* computes new actuator data by applying the received objects of type `NavigationData` and `SensorData` to `PositionControlAlgorithm` *co*.

The sequence discussed presents the details of manual and autonomous flight control of the JAviator FCS developed by this thesis.

6.4 Summary

This chapter presents the design and implementation of the JNavigator software. After outlining the basic concepts, it explains the functional model and describes the implemented software modules in detail.

7

Evaluation

This chapter exemplifies time responses of the designed controllers, interprets the path following results, and investigates the FCS JVM behavior. It focuses on autonomous flights, because a significant amount of work has been done already in the domain of model UAVs.

Figure 7.1 shows the test arrangement used with JVMs for Plant, FCS, and GCS. The Plant JVM executes instances of `MockJAviator` and `GpsReceiverSimulator`. Every time an actuator data packet arrives at instance `MockJAviator`, it calculates the new state of the virtual helicopter and sends a sensor data packet to both FCS and `GpsReceiverSimulatorAdapter`. Every 100 ms, instance `GpsReceiverSimulator`

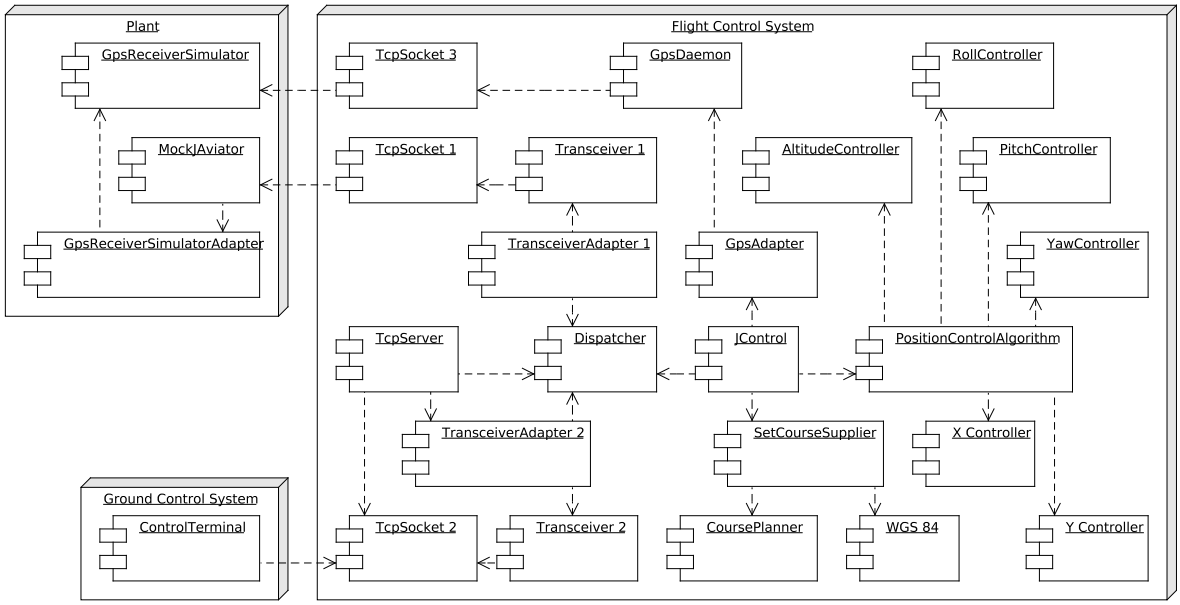

Figure 7.1: Test arrangement comprising separate JVMs for Plant, FCS, and GCS.

polls the current position from instance `GpsReceiverSimulatorAdapter` and creates a new set of NMEA 0183 messages, which the FCS accesses via a TCP/IP connection.

Object `GpsReceiverSimulator` imitates an ideal location system and, for some tests, applies deviation data to emulate position inaccuracies. The GCS JVM runs instance `ControlTerminal`, which connects to the FCS also via TCP/IP.

`JControl` is the main component in the FCS. It employs `PositionControlAlgorithm`, `GpsAdapter`, and `SetCourseSupplier` instances to compute new actuator data for the Plant in response to received sensor data. The communication between `JControl` and Plant uses `Dispatcher` for delivering DTOs, `TransceiverAdapter_1` for transforming packets to DTOs and back again, `Transceiver_1` for packet assembling and disassembling, and `TcpSocket_1` for TCP/IP communication. Instance `ControlTerminal` visualizes the current vehicle status by presenting gauges for altitude, roll, pitch, and yaw, as well as a 3-D view of the vehicle. The communication path from GCS to `JControl` comprises `TcpServer` for connection establishment, `TcpSocket_2` for actually interchanging data, `Transceiver_2` for packet assembling and disassembling, `TransceiverAdapter_2` for transforming packets to DTOs and back again, and `Dispatcher` for delivering DTOs. Object `SetCourseSupplier` uses instances `WGS 84` and `CoursePlanner` to calculate nominal positions for autonomous flights. To accomplish its task, the `PositionControlAlgorithm` utilizes instances of class `PDDController` as sub controllers named `RollController`, `PitchController`, `YawController`, `AltitudeController`, `X_Controller`, and `Y_Controller`.

7.1 Altitude flight control

The altitude controller is responsible for holding the helicopter at a certain height. It should respond fast, but not overshoot the target value to avoid unexpected crashes to ground and ceiling. It is critical to harmonize altitude and attitude controllers, because a slow altitude controller and fast attitude controllers cause the vehicle to lose height temporarily when the vehicle's attitude changes. The parameters of the controller were found by experimenting with the values for proportional and derivative gains. Figure 7.2 shows the helicopter's responses to altitude steps (a) from 0.5 m to 1.5 m and (b) back again.

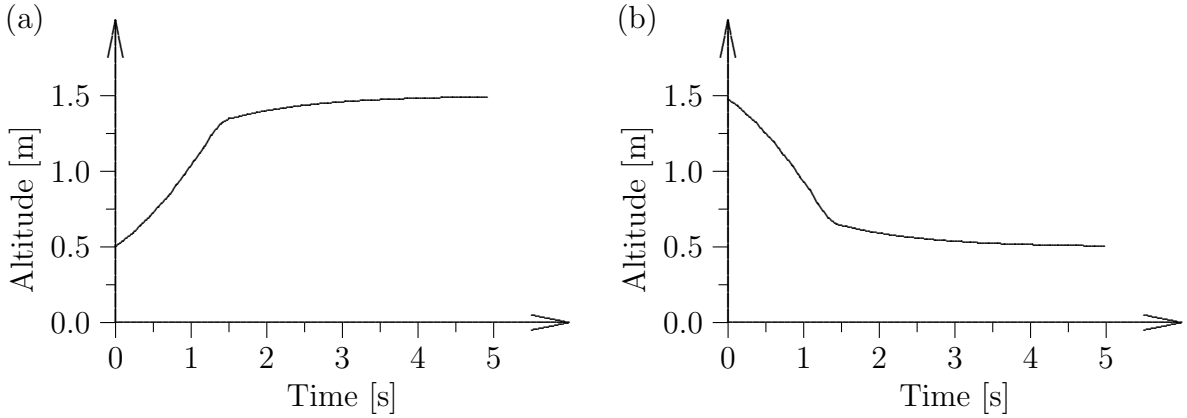

Figure 7.2: The helicopter's responses to a step input command (a) from 0.5 m to 1.5 m and (b) from 1.5 m to 0.5 m.

As visualized, the altitude controller adjusts 85% of the height difference within 1.5 s and reaches 99% after 4.6 s.

7.2 Attitude flight control

The attitude controllers are concerned with stabilizing the angles roll, pitch, and yaw. In order to quickly respond to new desired angle values, the controller may overshoot up to 30%. This causes the helicopter to sway a little, but nevertheless makes no odds. It is essential that the attitude controllers swiftly adapt to new nominal values for rapidly following a set course. For the attitude controller tests, the virtual helicopter evenly hovers at 0.5 m for some 10 s, before the attitude nominal values change. Again, the controller parameters were found by experimenting with the values for proportional and derivative gains. Figure 7.3 visualizes the time responses to (a) a 10° roll, (b) a 10° pitch, and (c) a 20° yaw step input command. Moreover, Figure 7.3 presents the

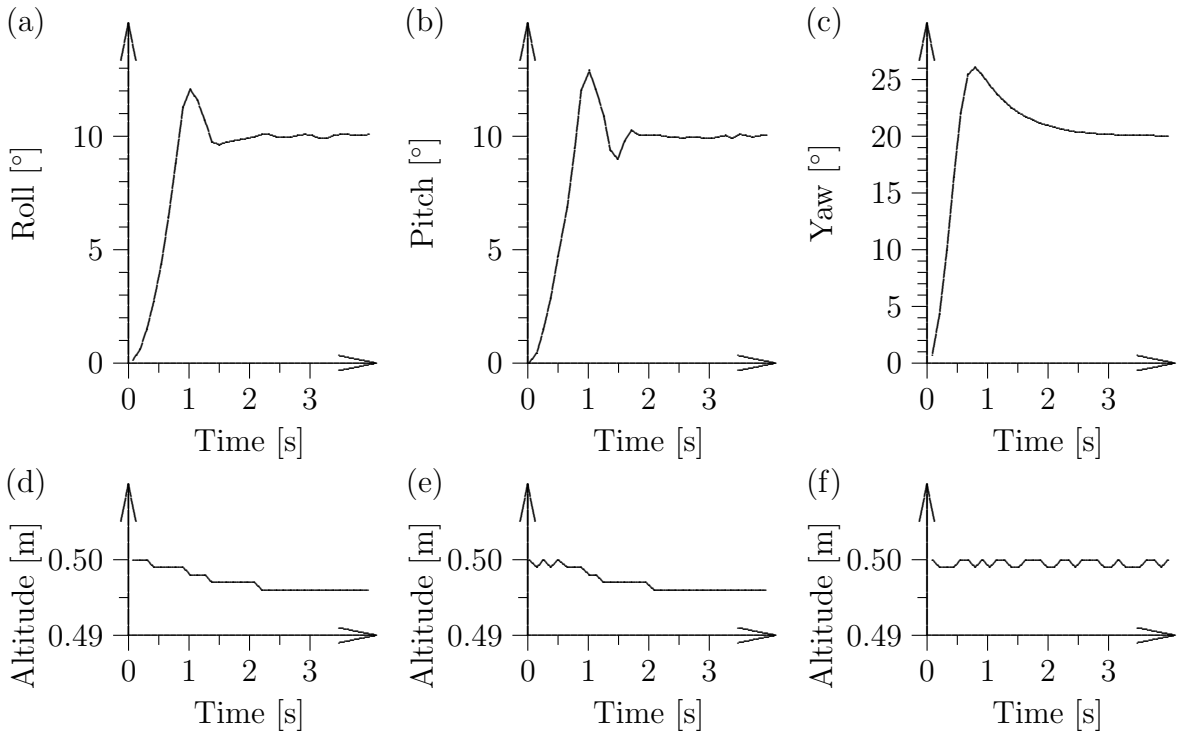

Figure 7.3: The helicopter's attitude responses to (a) roll, (b) pitch, and (c) yaw step input commands, as well as the corresponding altitude responses.

corresponding altitude time responses. The roll and pitch controllers compensate their steps better than 90% within 1.5 s and reach 99% before 2 s. Within 1.6 s, the yaw controller adjusts to the new nominal value better than 10% divergence and manages 1% before 3 s. Subfigures (d) and (e) indicate that the helicopter's altitude wanes no more than 5 mm on a 10° attitude change. Subfigure (f) visualizes that yaw changes have no more than 1 mm impact to the JAviator's altitude.

7.3 Position flight control

As developed in Chapter 5, two independent controllers manage the east and north position of the helicopter. Once again, the parameters of the controllers were found by experimenting with the values for proportional and derivative gains. Figure 7.4 visualizes the time responses for (a) a 10 m move eastwards and (b) a 10 m move northwards, without overshooting the desired target position. Subfigures (c) and (d) visualize roll and pitch of the helicopter while moving eastwards and northwards, respectively. As expected, the roll and pitch controllers cause the helicopter to sway a little. Subfigures (e) and (f) indicate that the vehicle's altitude falls off no more than 5 mm on a 10 m move. The east and north controllers reach 99% of the 10 m move within 10 s.

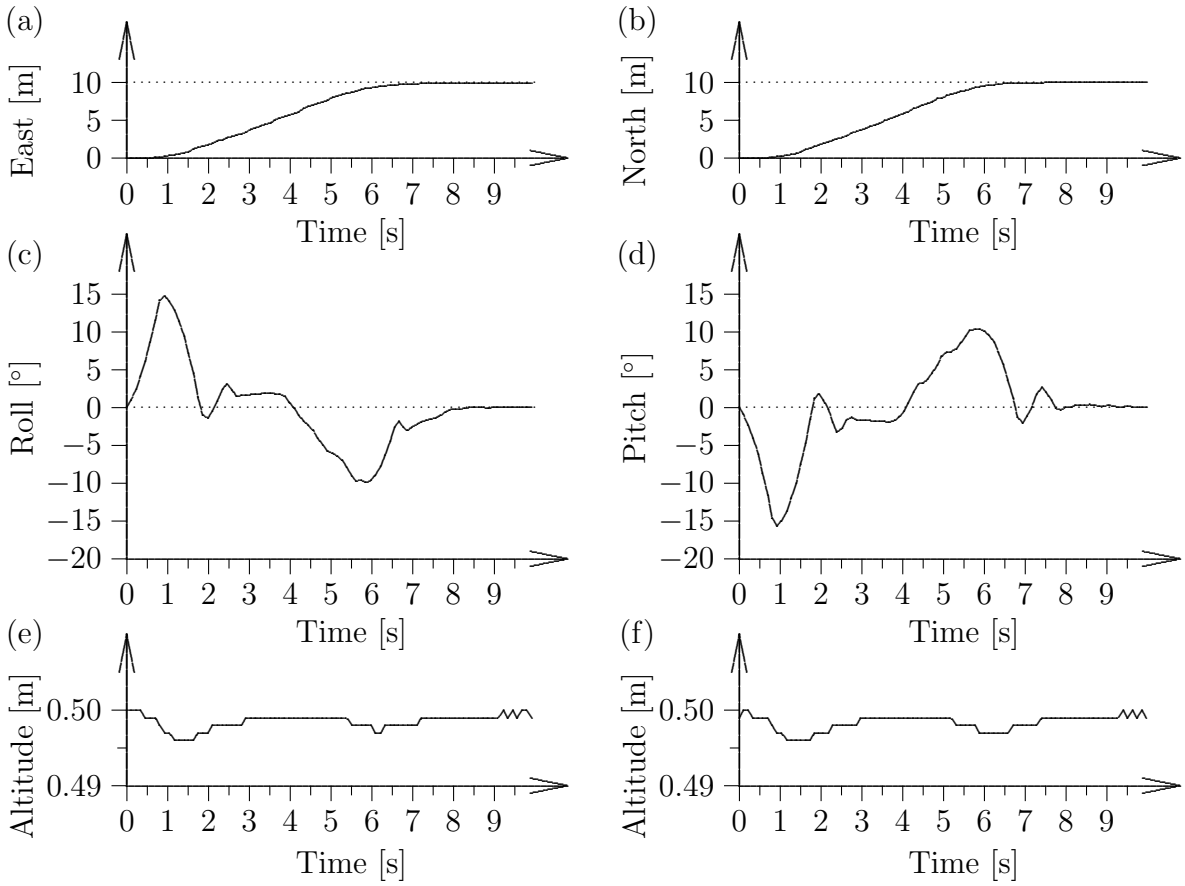

Figure 7.4: The helicopter's position, attitude, and altitude responses to a 10 m step input command for moving eastwards (left) and northward (right), respectively.

In addition to the time responses shown above, the controller ability to pin the JAviator to a particular point is crucial. The following paragraphs investigate the performance of the position controllers utilizing both ideal and inaccurate location systems.

When provided an ideal positioning system, the east and north position controllers are able to keep the vehicle hovering within a 1.8 cm deviation circle. Figure 7.5 visualizes the recorded 34 s trajectory. The presented trajectory is rather square, due to the fact that the simulated Plant, that is, the `MockJAviator`, provides position data only

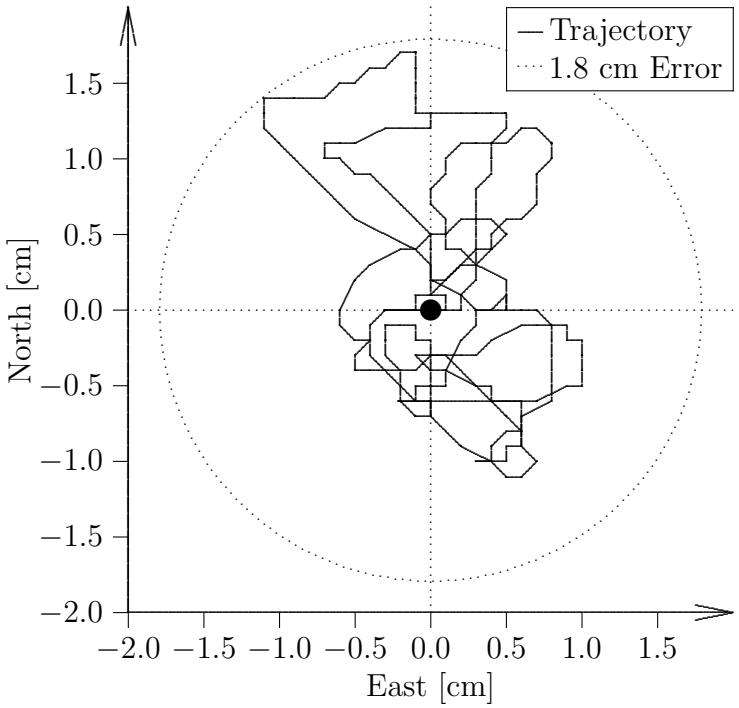

Figure 7.5: Position controller performance with an ideal positioning system applied (1.8 cm error circle, 34 s flight time).

in millimeter precision. The real JAviator employs a GPS receiver that provides a position accuracy of eight decimal digits, which is approximately the same precision.

To simulate inaccuracies of real GNSS, the GPS receiver simulator applies location deviations recorded by real GPS receivers. It is important that the period of data recording is equal to the period of simulating, because this enables the GPS receiver simulator to deliver authentic data. The midpoint of the recorded locations becomes deviation data origin. The data provided to the GPS receiver simulator are deviations of latitude, longitude, and altitude in meters. At startup time, the simulator loads its deviation data into a ring list. Each time the simulator receives a location data packet from the `MockJAviator`, it adds the current deviation data entry to the newly arrived position and delivers the result. For each subsequent received location data packet, the simulator advances one pace to the next deviation entry in the list to apply it. Figure 7.6 visualizes a 60 s DGPS rectified recording of the JAviator's GPS receiver at 10 Hz update rate. The largest deviation between reported and actual positions is 1.4 m.

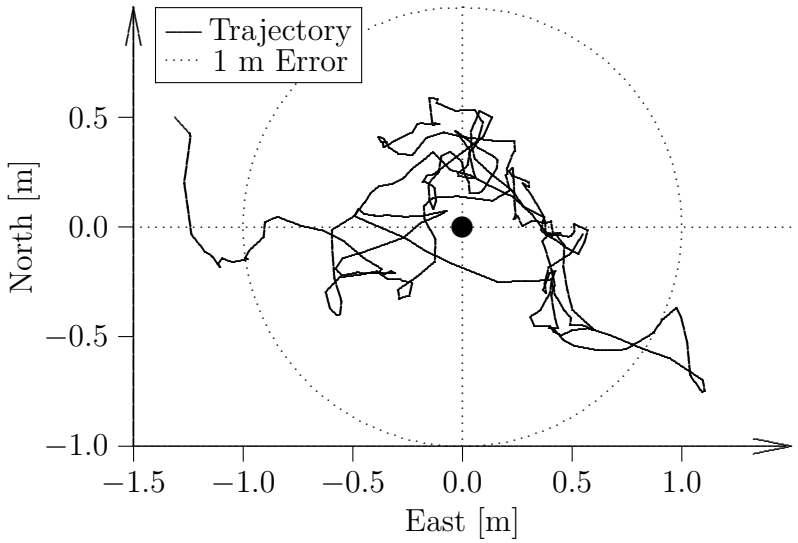

Figure 7.6: 60 s DGPS corrected deviations recorded by the JAviator's GPS receiver at 10 Hz update rate for emulating inaccuracies (1 m error circle, 1.4 m maximum error).

Figure 7.7 shows a 60 s trajectory of the `MockJAviator` as the GPS receiver simulator applies the deviation data depicted in Figure 7.6. As displayed, the helicopter's trajectory does not fit at all to the utilized deviation data. It is the inertia of the position

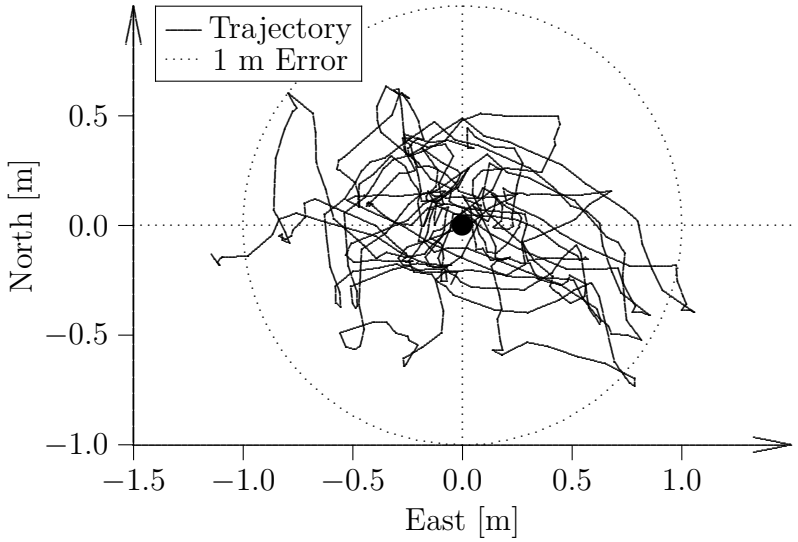

Figure 7.7: Position controller hover performance as the GPS receiver simulator applies the recorded deviation data shown in Figure 7.6 (1 m error circle, 60 s trajectory, 1.15 m maximum error).

controller that causes these discrepancies. However, the helicopter keeps within the 1 m error circle better than the recorded deviation data suggest. The largest difference between reported and actual positions is 1.15 m.

7.4 Autonomous flight control

This section presents results of three autonomous-flight experiments. The first two experiments investigate how accurate the implemented position controller moves the helicopter along a given set course dependent upon ideal and inaccurate location systems. The third experiment verifies the limitations of the position controller.

The applied set course is the same for all of the following experiments. It starts at the origin and proceeds (1) 10 m eastwards, (2) 10 m northwards, (3) 20 m westwards, (4) 20 m southwards, (5) 20 m eastwards, and (6) diagonally back to the origin.

7.4.1 Experiment 1 - ideal location systems

In the first experiment, an ideal location system is connected to the position controller. This test shows how exactly the controller stirs the vehicle along a given set course. The average set course velocity is 0.5 m/s in this experiment. Figure 7.8 depicts the position controller performance when utilizing an ideal location system. The controller

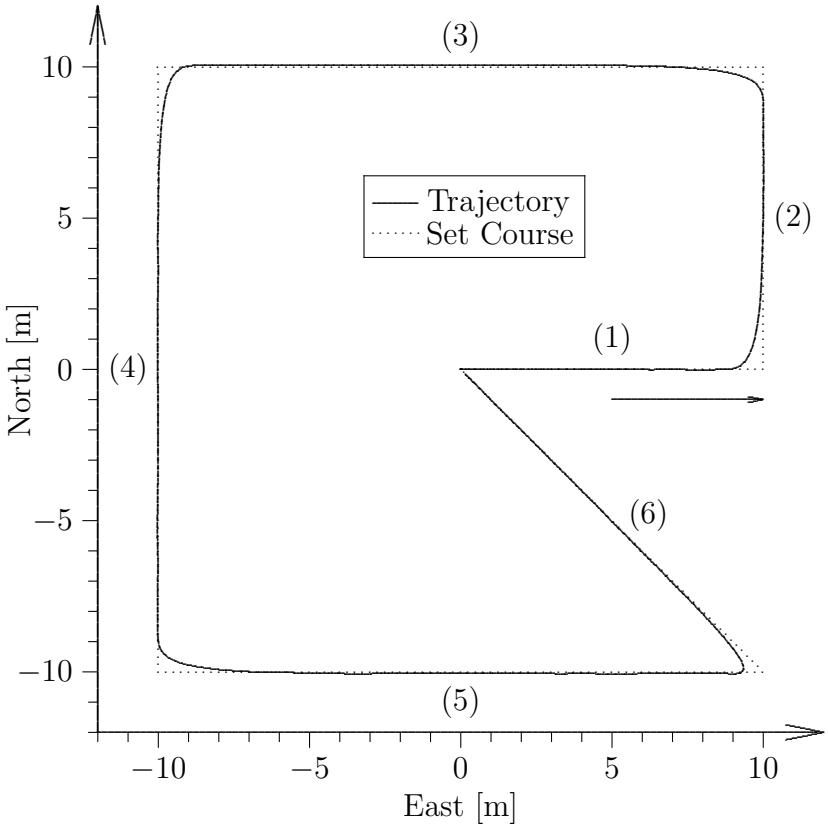

Figure 7.8: Path following at 0.5 m/s average velocity utilizing an ideal location system.

is able to keep the vehicle's flight path within a few centimeters of lateral difference at following a linear set course. The helicopter's trajectory does not reach the corners stipulated by the set course, due to the inertia of the position controllers.

In addition to lateral variances, the deviation between set course position and vehicle position is of particular interest. It is this deviation that rounds the vehicle's trajectory at square set courses. Figure 7.9 visualizes the follow-up error of the first experiment regarding to time. As displayed, the deviation between helicopter and set course is less than 2 m at an average speed of 0.5 m/s.

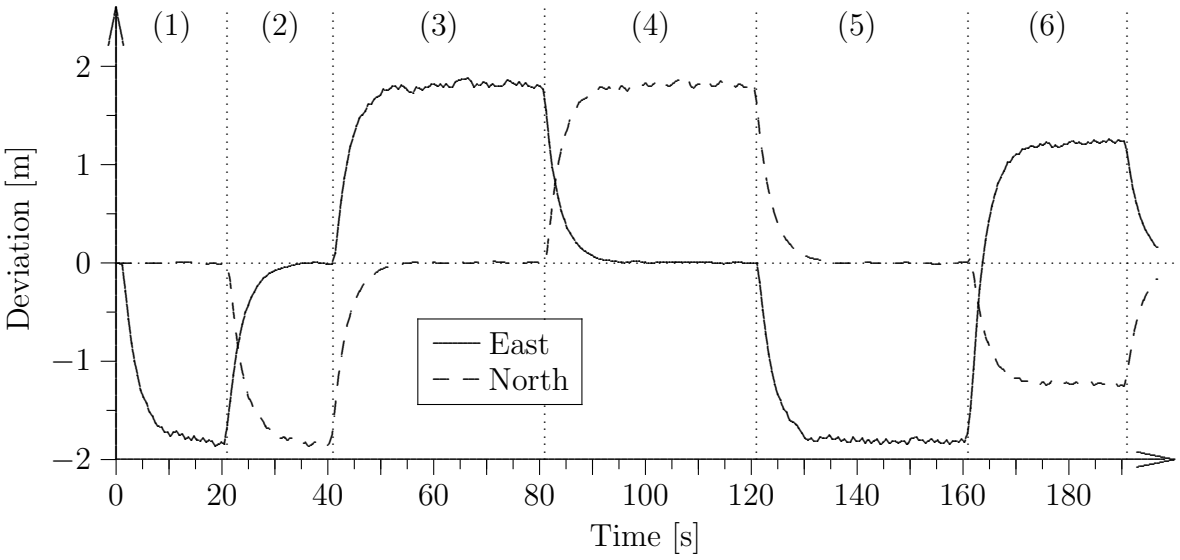

Figure 7.9: East and north deviations during path following at 0.5 m/s.

As the set course position accelerates in Segment (1), the east deviation increases to 1.8 m after 21 s and the north deviation keeps smaller than 15 mm. Now, the set course position enters Segment (2), but the UAV has not yet mastered Segment (1). While the set course position advances north, the UAV cuts the corner and tries to follow. This causes the east deviation to shrink to less than 15 mm after 13 s. However, the north deviation grows to 1.9 m at the end of Segment (2). This scenario repeats for the other segments, as plotted in Figure 7.9.

Besides position errors, larger deviations between set course position and vehicle cause also timing errors. Because of the follow-up error, the vehicle is not at the desired position in time. Hence, larger distances cause longer latencies. In this experiment the set course ends after 191 s of flight time, but the vehicle is 1.1 m east and 1.1 m north away from its target position. The vehicle arrives at its target position 13 s late.

7.4.2 Experiment 2 - inaccurate location systems

In the second experiment, an inaccurate positioning system delivers location updates. The employed GPS receiver simulator performs inaccuracy emulation by utilizing the deviation data shown in Figure 7.6. As visualized in Figure 7.10, the UAV trajectory shows deviations of about approximately the same size as displayed in Figure 7.7. Despite location system inaccuracies and controller inertness, the position controller

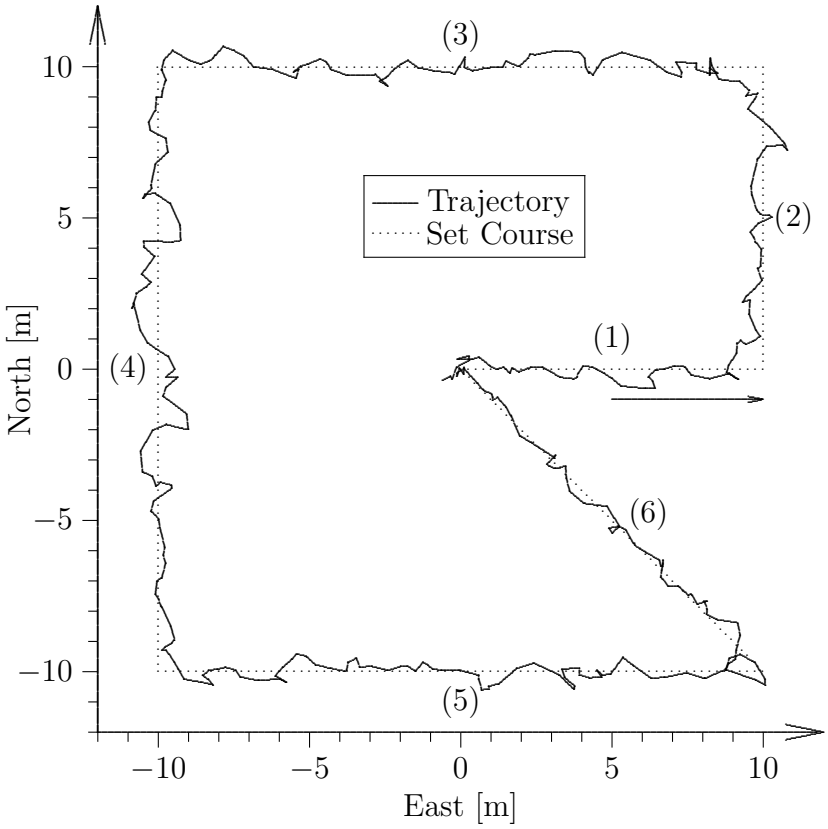

Figure 7.10: Path following at 0.5 m/s average velocity utilizing an inaccurate location system.

is able to stir the helicopter along the given set course. Because of the inaccuracies of the applied location system, this experiment delivers no further insight into the performance of the position controllers.

7.4.3 Experiment 3 - controller limitations

The third experiment investigates the interrelation between control deviation and set course velocity. This test uses the same set course as the previous experiments, but increases average segment speeds. Segment (1) starts at 0.625 m/s, Segment (2) continues with 1 m/s, Segment (3) proceeds with 1.3 m/s, Segment (4) advances to 2 m/s, Segment (5) progresses to 2.2 m/s, and Segment (6) slows down to 1.4 m/s.

Figure 7.11 exhibits the helicopter's trajectory corresponding to this accelerating set course. Furthermore, it exemplifies the deviation between vehicle and set course at certain positions. As displayed, the deviation between helicopter and set course increases

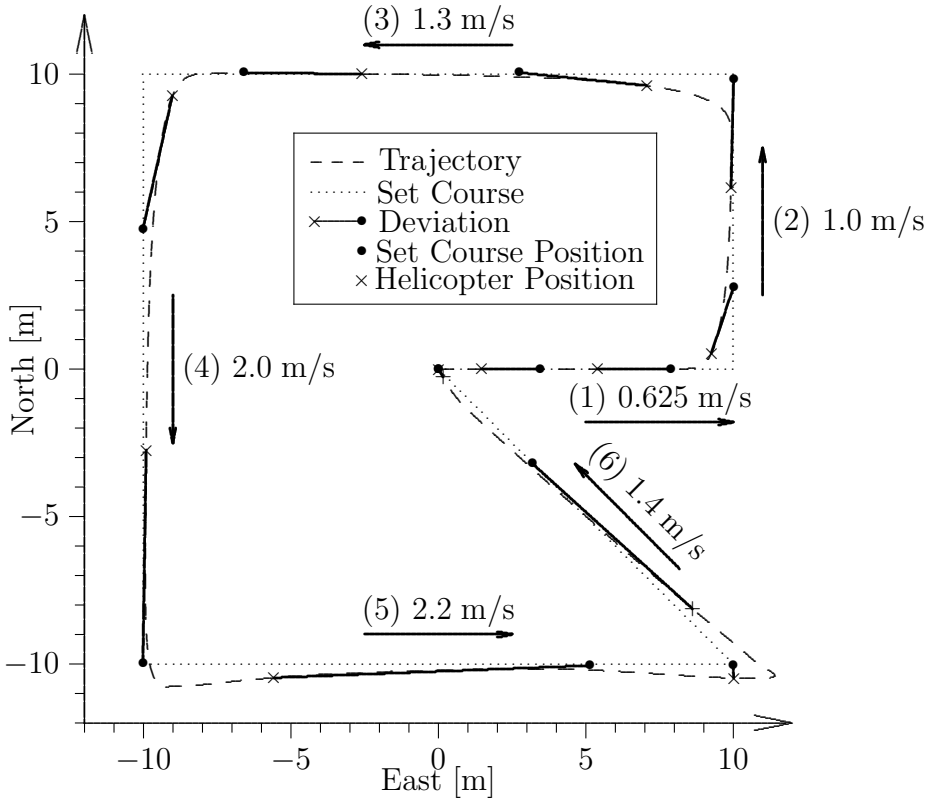

Figure 7.11: Path following at miscellaneous velocities.

as the average velocity of the corresponding segment waxes. Figure 7.12 depicts this observation in detail. In Segment (1), the position controller keeps the north deviation below 1.3 cm, but allows the east deviation grow up to 2.5 m. While the north deviation increases to 3.8 m in Segment (2), the east deviation shrinks to 10 cm at its end.

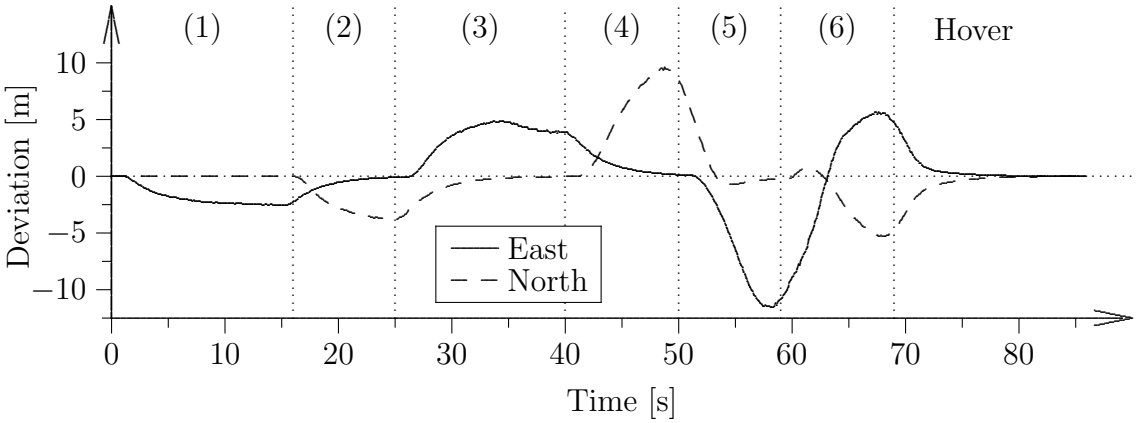

Figure 7.12: Deviation during path following at miscellaneous velocities.

Segment (3) causes the east deviation wax to 4.8 m after two thirds, while the north deviation dwindles to 1 cm. In Segment (4) the east deviation diminishes to 13 cm, as the north deviation rises to 9.6 m. Segment (5) allows the north deviation to drop off faster than in the segments before, but causes an overshoot of 0.7 m. Meanwhile, the east deviation jumps to 11.5 m. Because of the lower average speed in Segment (6), the east and north deviations *only* reach 5.6 m and 5.3 m, respectively.

7.5 FCS JVM behavior

Figure 7.13 shows the experimental test bed used including an IBM ThinkPad T60p laptop, a 100 Mbit/s network switch, and a 600 MHz Gumstix Verdex XL6P. The laptop uses a standard Ubuntu 8.04.3 [9] installation and executes the MockJAviator as well as the GCS in different SUN 1.5.0_16-b2 JVMs [36]. On the Gumstix, gumstix-buildroot revision 1541 with real-time patches applied runs the FCS in IBM's WSRT JVM [27]. To avoid side effects of slow flash memory, both FCS and JVM have been installed on RAM disk. A 100 Mbit/s network switch connects the Gumstix to the laptop.

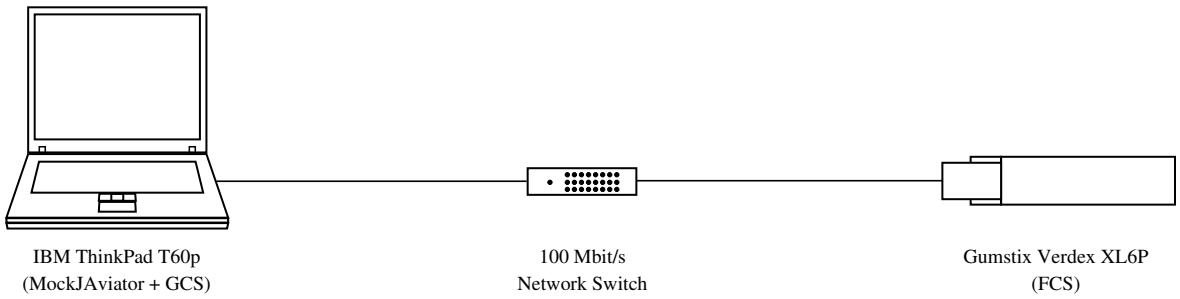

Figure 7.13: Experimental test bed.

To study the JVM behavior, the FCS should control the MockJaviator to hover at a particular location. Garbage collecting should consume only 5% of available computing power to reduce its effect to a minimum. Additionally, the JVM is granted 52 MiB of heap memory.

Figure 7.14a visualizes IBM's WSRT JVM memory utilization when running the FCS in autonomous-flight mode. At time 0 s in the diagram, the JVM executes the FCS about 30 s. The JVM initiates GCs at 93 s, 226 s, and 423 s. The first two GCs were triggered at approximately 50% of the granted memory. IBM's WSRT JVM GC Metronome decreases memory utilization quickly, but not abruptly to reduce its influence. After 423 s, the memory consumption reaches its upper limit and Metronome can no more guarantee its CPU utilization when collecting garbage. In other words, Metronome needs more computing power to succeed.

Figure 7.14b shows nominal and actual controller cycle times with respect to autonomous-flight time. The nominal cycle time is 20 ms, but the FCS only manages approximately 125 ms until 304 s of autonomous flight. Thereafter, the average cycle time slumps to 50 ms until 348 s of flight time and memory utilization increases marginally faster. After 348 s the average cycle time decreases to 30 ms, but memory utilization ascends rapidly. After 423 s of autonomous flight, memory utilization reaches its upper limit of 52 MiB and Metronome collects all the garbage at once. Thereafter, the cycle time averages again at 125 ms. Cycle times that long also indicate that over 80% of the actuator packets have been dropped.

An aggravating factor is that after starting the FCS the system utilization reaches 100% immediately, even without a connected GCS. There is no more computing power left

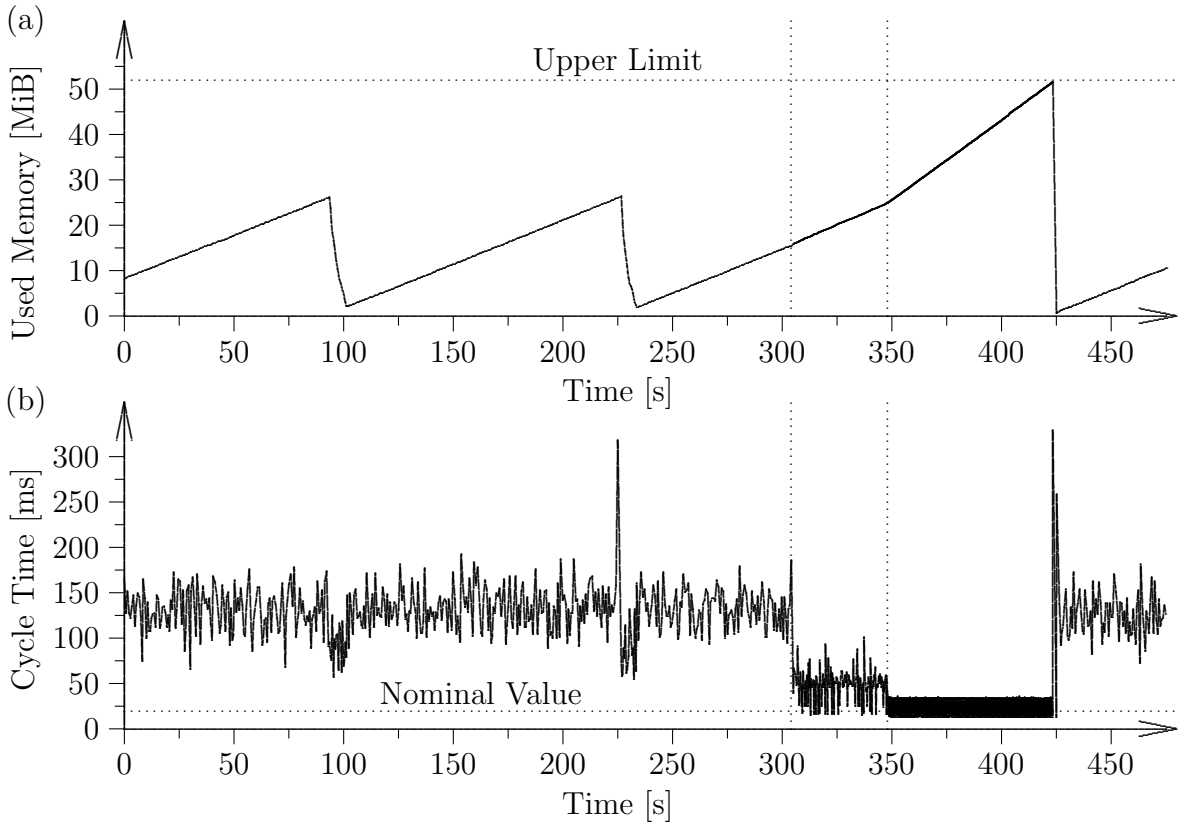

Figure 7.14: IBM's WSRT-JVM running the FCS on Gumstix Verdex XL6P.

that Metronome may claim. Hence, this experiment failed, because of system overload. Garbage collecting is not the problem, but there are two other reasons for overload. First, the FCS intensively uses double-precision arithmetic and Gumstix Verdex XL6P boards lack a Floating Point Unit (FPU). Second, together with the real-time-patched Linux kernel, IBM's WSRT JVM applies sophisticated algorithms to obtain predictable response times, which cause additional overhead.

To show that the implemented FCS works properly, the Gumstix has been replaced by a standard 3.3 GHz Intel Celeron PC for further experiments. On this hardware, a non-real-time IBM Java 1.4.2 JVM executes the FCS on Ubuntu 8.04.3 [9]. The initial JVM heap size is 16 MiB and the maximum heap size is 128 MiB. The following experiment shows that the implemented FCS works properly if provided enough computing power. The subsequent experiment shows that memory utilization has an upper limit during the JAviator's maximum flight time, that is, about 40 min.

Figure 7.15 visualizes (a) JVM memory utilization and (b) controller cycle time of a 480 s autonomous flight of the FCS in the described non-real-time environment. At time 0 s in the diagram, the JVM executes the FCS for about 10 s. Memory utilization has its lower limit at approximately 1.5 MiB and its upper limit at approximately 16 MiB, as shown in Subfigure (a). Although the JVM is allowed to use much more heap memory, its upper memory consumption is about the initial heap size. JVM initiated GCs repeat at a cycle time of approximately 42 s.

Subfigure (b) visualizes that until 42 s flight time, the cycle time vigorously fluctuates around an average of 22 ms. Then, the cycle time settles down to 21 ms in average and

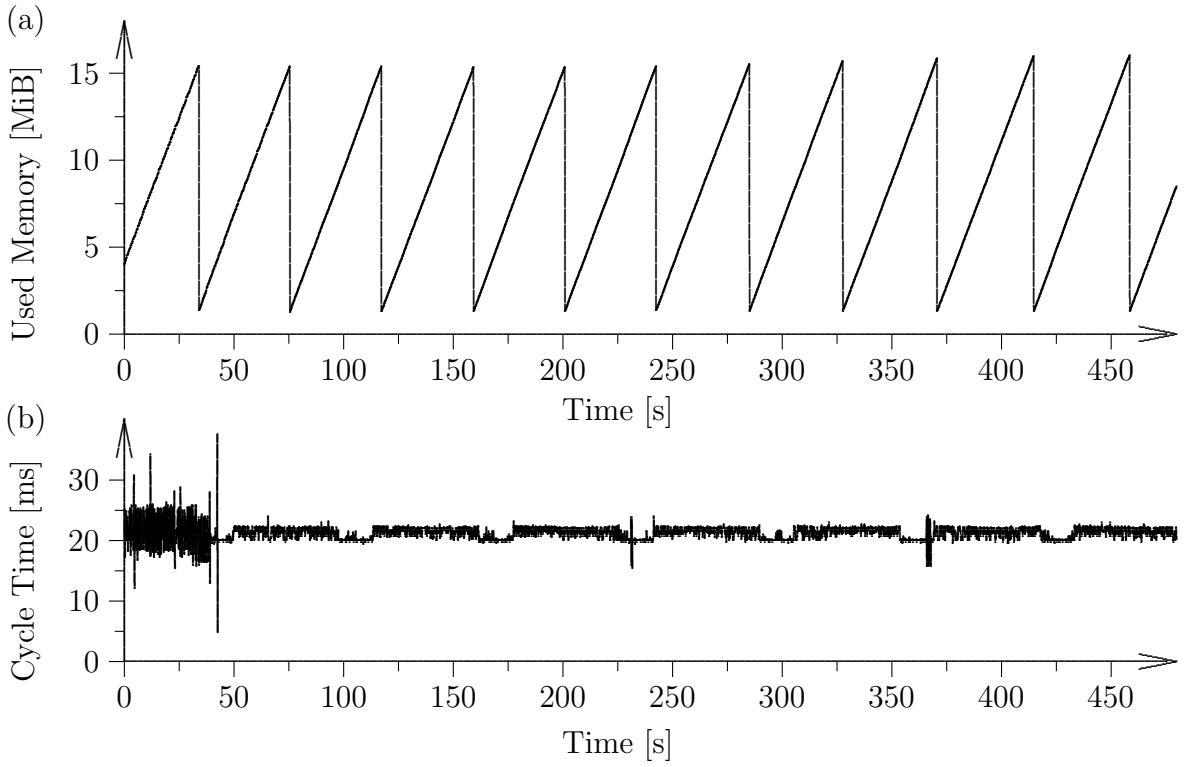

Figure 7.15: Memory usage and cycle time when running the FCS on IBM's Java 1.4.2 JVM on a 3.3 GHz Intel Celeron PC (non-real-time) for 480 s.

fluctuates much less. During this experiment the FCS induced average CPU utilization is 1.1%.

Figure 7.16 displays the JVM memory utilization during an 1 h autonomous flight, which is 50% more than the real JAviator's flight time.

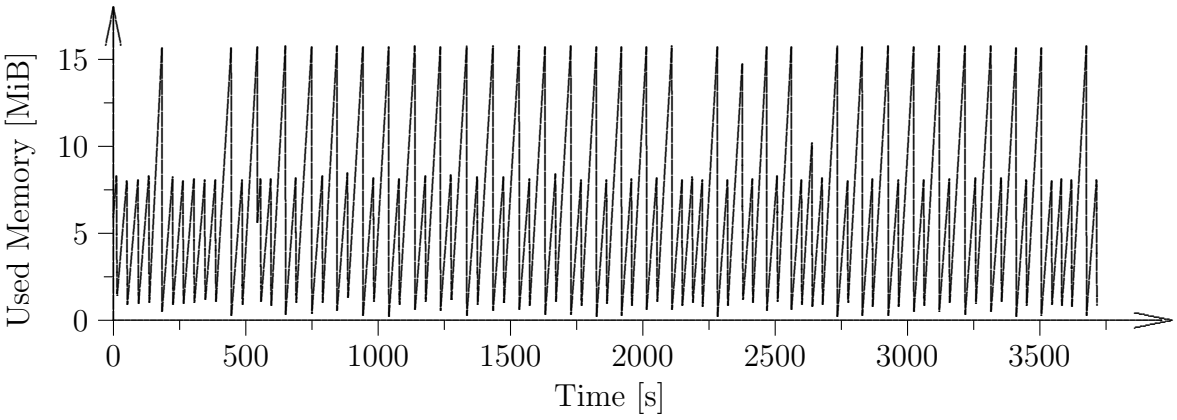

Figure 7.16: Memory usage when running the FCS on IBM's Java 1.4.2 JVM on a 3.3 GHz Intel Celeron PC (non-real-time) for approximately 1 h.

Again, the upper limit is about the initial heap size of 16 MiB. However, more than 50% of the GCs were triggered at about 8 MiB of allocated memory.

Although this two experiments ran on a non-real-time Linux system, they showed that the implemented FCS is capable of controlling the JAviator if provided enough

computing power. These experiments depict that the required memory has an upper limit at approximately the initial heap size of 16 MiB. The last two experiments were successful, because the system running the FCS was utilized only to 1.1% in average. Controller cycle time inaccuracies are originated in non-real-time scheduling algorithms of the applied Linux system.

To run the implemented FCS on an embedded system mounted on the JAviator, more double-precision computing power is necessary. The employed Gumstix and Intel Celeron systems are too dissimilar from each other to allow a reliable estimation of required computing power for an embedded system without an FPU.

It is this non-real-time setup of the last two experiments that delivered the results of the previous sections in this chapter.

7.6 Summary

This chapter presents time responses of the developed controllers, analyzes the autonomous-flight results, and explores the behavior of the FCS JVM.

The developed controllers are able to control altitude, attitude, and position of the simulated helicopter in combination with both precise and inaccurate positioning systems. The controllers show their shortcomings when applied to "speedy" set courses. The deviation between set course and helicopter increases with the set course velocity.

The implemented FCS needs more resources than supported by a 600 MHz Gumstix Verdex XL6P when executing IBM's WSRT JVM on a Linux system with real-time patches applied. A non-real-time test setup employing a 3.3 GHz Intel Celeron PC shows that the implemented FCS works properly, but does not allow a reliable estimation of required computing power regarding an embedded system without an FPU.

8

Conclusion

This thesis has presented the design, implementation, and evaluation of the JNavigator autopilot software. This chapter concludes the thesis by summarizing the current implementation, outlining design strengths and weaknesses, and providing an outlook to future work.

8.1 Current implementation

The software design focuses on flexibility and extensibility. It covers not only the JAviator's requirements, but also easily adapts to other UAVs. A hierarchical configuration approach allows arbitrary reuse of components that are either included in the JNavigator or provided externally from additional libraries using the JNavigator programming interfaces.

JNavigator substitutes the existing FCS and controls both manual and autonomous flights. The GCS can interrupt an ongoing autonomous flight at any time. Interrupting an autonomous flight is currently challenging for an operator, because she must provide an appropriate vehicle altitude nominal value to avoid unintended crashes.

JNavigator verifies that the JAviator can fly a loaded set course and prepares an acceleration plan before starting an autonomous flight. Creating a set course is presently a laborious task, because low acceleration limits narrow the range of values for both velocity and distance of sections in a flight plan.

For the determination of the current JAviator position, JNavigator employs the NMEA 0183 [1] messages from a GPS receiver. For enhanced precision, JNavigator can forward an available RTCM SC-104 version 2.3 [2] correction data stream to the GPS receiver. The correction data stream may be provided by an NTRIP [4] caster, a GSM service or a directly connected RTCM SC-104 receiver (TCP/IP socket, Bluetooth socket, or serial line). A newly implemented extension to the JAviator simulator allows determining the current position via NMEA 0183 messages. This extension can also be applied to convert coordinates of arbitrary positioning systems to NMEA 0183 messages.

Apart from eight calls to the operating system to access serial lines and Bluetooth sockets, JNavigator is entirely written in the Java programming language. Calls to the

operating system are nicely abstracted to allow access to TCP/IP sockets, Bluetooth sockets, and serial lines uniformly.

8.2 Design strengths and weaknesses

JNavigator's experimental configuration described in Chapter 7 is a list of 92 parameters. Many of these parameters are for initialization of instances of the same class, like, for example, classes `TcpSocket` and `PDDController`. Employing a singleton object that reads and verifies all configuration parameters for later use seemed unattractive, because it adds unnecessary complexity and dependencies. This thesis suggested that each component is provided a `Properties` object containing its configuration and verifies the parameters itself in its initialization code. This applies to subcomponents also, which receive their part of the configuration from their parent components. It is this hierarchical configuration approach that allows arbitrarily reuse of components, which are either included in the JNavigator or externally provided from additional libraries using the JNavigator programming interfaces.

Another design strength is that components only need to know subcomponent interfaces. A generic object factory takes care of extracting subcomponent configurations from the parent configuration and instantiating subcomponents via their class names. This allows adding new components without adapting and rebuilding existing code. Only the configuration changes. Besides of more flexibility in usage, components become easily testable, because components and subcomponents are loosely coupled via interfaces. To test a component, unit tests only have to support class names for mock subcomponents in the component configuration.

The next design strength is that every component performs only one task. Components employ subcomponents to execute complex jobs. This approach leads to components of limited complexity. Together with reasonable component names, this approach results in understandable, testable, and maintainable code.

JNavigator uses ring buffers for data exchange between threads. This is a design strength, because threads can act independently from each other without being blocked by others. JNavigator uses separate threads for reading and writing of each connection stream, that is, there are two threads for handling the Plant interface, two threads for handling the ground control interface, and two threads for handling the GPS receiver interface. Each reading thread receives bytes from its connection stream. Whenever a package is complete, a reading thread forwards this packet to the dispatcher, which routes the packet to all interested listeners. Thereafter, a reading thread continues receiving bytes from its stream. Another thread cyclically distributes actuator data packets to the dispatcher, which routes the packets to all interested listeners like the writer threads of the Plant and GCS connections. The writer threads employ ring buffers to allow the dispatcher passing packets without being blocked. A writer thread takes an entire message from its ring buffer and sends it byte by byte. Reader and writer threads are blocked when performing I/O operations, but one thread never blocks another thread.

Another design strength is that DTOs are not changed after initialization. This allows multiple threads to access a single DTO without the need of synchronization.

One design weakness is the resource utilization of this Java implementation, which requires powerful onboard computers. The implemented software intensively needs double-precision calculation power, which causes overload on embedded systems lacking an FPU. Algorithms to enable real-time functionality in Java introduce additional overhead.

8.3 Future work

There are several enhancements that this thesis leaves for future work. First of all, a state estimator that integrates a location system and inertial sensors, as proposed in [5], would provide better position information. This would allow faster position controllers without introducing instabilities.

A future state estimator should also be able to handle two or more different location systems concurrently. This would offer autonomous flights from inside to outside of buildings and vice versa.

The position controller needs optimization to reduce the deviation between nominal location and vehicle position when executing "speedy" set courses.

As mentioned earlier, creating set courses is a laborious task. Low acceleration limits narrow the range of values for both velocity and distance of sections in a flight plan. A future planning tool should enable an operator to quickly compose a valid set course. Furthermore, this tool should provide a graphical user interface that allows planning set courses via maps.

In contrast to set courses, a Vehicle Control Language (VCL) would provide a higher degree of freedom when assembling autonomous-flight plans. A future planning tool could emit VCL scripts to be uploaded to the FCS for execution. [40] exemplifies a simple but effective VCL implementation.

Currently, interrupting an ongoing autonomous flight raises high demands on an operator. To avoid crashing the helicopter, *synchronizing* to the set course is necessary before switching to manual flight. A future enhancement could mix manual and autonomous control as follows. Whenever an operator wants to interfere an autonomous flight, she could use the GCS-attached joystick. The FCS would recognize this and would switch to manual flight. Moreover, the FCS would dampen altitude changes until the operator gets a grip on controlling the altitude.

A nice extention to the GCS user interface would be an attitude indicator as known from aircrafts. Another beneficial extention to the GCS user interface would be a bird's eye view that allows zooming.

References

[1] *NMEA 0183, Standard for Interfacing Marine Electronic Devices*, 1998. `http://www.nmea.org`.

[2] *RTCM Recommended Standards for Differential GNSS Service*, 1998.

[3] *GLOBAL NAVIGATION SATELLITE SYSTEM GLONASS, Interface Control Document (version 5.0)*. `http://www.glonass-ianc.rsa.ru/i/glonass/ICD02_e.pdf`, 2002.

[4] *RTCM Recommended Standards for Networked Transport of RTCM via Internet Protocol (Ntrip)*, 2004.

[5] Andrews, A. P., Grewal, S. M., and Weill, L. R.: *Global Positioning Systems, Inertial Navigation, and Integration.* John Wiley and Sons, Inc., New York, 2nd edition, 2007.

[6] Auerbach, J., Bacon, D., Craciunas, S., Iercan, D., Kirsch, C., Rajan, V., Röck, H., and Trummer, R.: *The JAviator Project*. `http://javiator.cs.uni-salzburg.at` (7.9.2009), 2006–2009.

[7] Auerbach, Joshua, Bacon, David F., Iercan, Daniel, Kirsch, Christoph M., Rajan, V. T., Röck, Harald, and Trummer, Rainer: *Low-latency time-portable real-time programming with exotasks*. Trans. on Embedded Computing Sys., 8(2):1–48, 2009, ISSN 1539-9087.

[8] Bundesamt für Eich- und Vermessungswesen (BEV): *APOS - Austrian Positioning Service*. `http://www.bev.gv.at/portal/page?_pageid=713,1571538&_dad=portal&_schema=PORTAL` (7.9.2009), April 2009.

[9] Canonical Ltd.: *Ubuntu*. `http://www.ubuntu.com` (7.9.2009), September 2009.

[10] CISRO ICT Centre: *CSIRO Precision Location Technology (PLT)*. `http://www.ict.csiro.au/page.php?did=67&print=print` (7.9.2009), 2009.

[11] Conte, Gianpaolo, Duranti, Simone, and Merz, Torsten: *Dynamic 3d path following for an autonomous helicopter.* In *Proceedings of the IFAC Symposium on Intelligent Autonomous Vehicles.*, 2004.

[12] Craciunas, S., Kirsch, C., Payer, H., Röck, H., Sokolova, A., Stadler, H., and Staudinger, R.: *The Tiptoe Project*. `http://tiptoe.cs.uni-salzburg.at` (7.9.2009).

[13] Crosby, Graeme K., Kraus, Donna K., Ely, William (Bill) S., Cashin, Timothy P., McPherson, Keith W., Bean, Kevin W., Stewart, Joy M., and Elrod, Dr. Bryant D.: *A Ground-based Regional Augmentation System (GRAS) - The Australian Proposal*. Presented at ION GPS2000, Salt Lake City, UT, September 2000.

[14] Egerstedt, M., Hoffmann, F., Koo, T. J., and Sastry, S.: *Path planning and flight controller scheduling for an autonomous helicopter*. Lecture Notes in Computer Science, 1569:91–102, 1999.

[15] EUROCONTROL: *Ground-Based Augmentation System (GBAS)*. `http://www.ecacnav.com/Future_Applications/GBAS` (7.9.2009), 2000–2008.

[16] EUROCONTROL: *Satellite-Based Augmentation Systems (SBAS)*. `http://www.ecacnav.com/Future_Applications/SBAS/SBAS_Home.html` (7.9.2009), 2000–2009.

[17] European Commission, Transport: *Galileo*. `http://ec.europa.eu/transport/galileo/index_en.htm` (7.9.2009), September 2009.

[18] European Space Agency (ESA): *The present - EGNOS Navigation*. `http://www.esa.int/esaNA/egnos.html` (7.9.2009), 2000–2009.

[19] Federal Aviation Administration: *Local Area Augmentation System - How It Works*. `http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/laas/howitworks/` (7.9.2009), August 2009.

[20] Fregene, K., Lai, G., and Wang, D.: *A control structure for autonomous model helicopter navigation*. In *Canadian Conference on Electrical and Computer Engineering*, volume **1**, pages 103–107, Portoroz, Slowenien, April 2000.

[21] Gumstix, Inc. `http://www.gumstix.com/` (7.9.2009), August 2009.

[22] Hoffmann, F., Koo, T. J., Shim, H., Sinopoli, B., and Sastry, S.: *Hybrid control of an autonomous helicopter*. In *IFAC Workshop on Motion Control*, pages 285–290. AMS Press, 1998.

[23] Hoffmann, Frank, Koo, Tak John, and Shakernia, Omid: *Evolutionary design of a helicopter autopilot*. In *3rd On-line World Conf. on Soft Computing (WSC3)*, pages 201–214, 1998.

[24] Hoffmann, Gabriel M., Jang, Jung Soon, Tomlin, Claire J., and Waslander, Steven L.: *Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning*. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 468–473, Edmonton, AB, Canada, August 2005.

[25] Hoffmann, Gabriel M., Tomlin, Claire J., and Waslander, Steven L.: *Quadrotor helicopter trajectory tracking control*. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Honolulu, HI, August 2008. AIAA Paper Number 2008-7410.

[26] International Business Machines Corp.: *Metronome*. http://researchweb.watson.ibm.com/metronome/ (7.9.2009), June 2008.

[27] International Business Machines Corp.: *WebSphere Real Time*. http://www.ibm.com/software/webservers/realtime (7.9.2009), August 2009.

[28] Johnson, Eric N. and DeBitetto, Paul A.: *Modeling and simulation for small autonomous helicopter development*. AIAA, 1997.

[29] Kim, H. J., Shim, D. H., and Sastry, S.: *Flying robots: Sensing, Control and Decision Making*. In *ICRA '02. IEEE International Conference on Robotics and Automation*, volume **1**, pages 66–71, Washington, DC, USA, May 2002.

[30] Kim, H. Jin and Shim, David H.: *A flight control system for aerial robots: algorithms and experiments*. Control Engineering Practice, **11**(12):1389–1400, December 2003.

[31] Kottmann, Markus: *Software for Model Helicopter Flight Control technical report 316*. http://www.uav.ethz.ch/research/publications/316.pdf (7.9.2009), March 1999.

[32] Leishman, J. G.: *Principles of Helicopter Aerodynamics*. Cambridge University Press, New York, 2nd edition, 2006.

[33] NIMA: *DoD World Geodetic System 1984 - Its Definition and Relationships with Local Geodetic Systems*. Technical Report TR8350.2, National Geospatial-Intelligence Agency, 2000. http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf (7.9.2009).

[34] Priyantha, N., Chakraborty, A., and Balakrishnan, H.: *The Cricket Location-Support system*. In *Proceedings of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM)*, Boston, MA, August 2000.

[35] Singh, Arjun: *GAGAN - A visionary approach*. http://www.mycoordinates.org/gagan-july-05.php (7.9.2009), July 2005.

[36] Sun Microsystems: *Sun Developer Network*. http://java.sun.com (7.9.2009), February 2009.

[37] Ubisense: *Ubisense Precise Real-time Location*. http://www.ubisense.net/ (7.9.2009), 2009.

[38] U.S. Department of Homeland Security: *GENERAL INFORMATION ON GPS*. http://www.navcen.uscg.gov/gps/default.htm (7.9.2009), September 2009.

[39] Ward, A., Jones, A., and Hopper, A.: *A New Location Technique for the Active Office*. IEEE Personal Communications, **4**(5):42–47, October 1997.

[40] Williams, Adam: *Vicacopter autopilot*. http://coptershyna.sourceforge.net/ (7.9.2009), June 2009.

[41] Yasuda, A.: *Japan: Augmenting navigation.* `http://www.mycoordinates.org/japan_augmenting_navigation.php` (7.9.2009), February 2008.

[42] Zogg, Jean Marie: *Essentials of Satellite Navigation (u-blox Compendium).* `http://www.u-blox.com/technology/GPS-X-02007.pdf` (7.9.2009), April 2007.

List of abbreviations

3-D	3-dimensional
APOS	Austrian Positioning Service
BEV	Bundesamt für Eich- und Vermessungswesen
DGPS	Differential GPS
DTO	Data Transfer Object
EGNOS	European Geostationary Navigation Overlay System
FCS	Flight Control System
FPU	Floating Point Unit
GAGAN	GPS Aided Geo Augmented Navigation
GBAS	Ground-Based Augmentation System
GC	Garbage Collection
GCS	Ground Control System
GGA	Global Positioning System Fixed Data
GLONASS	Globalnaja Nawigazionnaja Sputnikowaja Sistema
GNSS	Global Navigation Satellite System
GPRS	General Packet Radio Service
GPS	Global Positioning System
GRAS	Ground-Based Regional Augmentation System
GSM	Global System for Mobile Communications
IMU	Inertial Measurement Unit
JRE	Java Runtime Environment
JVM	Java Virtual Machine

LAAS	Local Area Augmentation System
MSAS	MTSAT Satellite Augmentation System
MTSAT	Multifunctional Transport Satellite
NMEA	National Marine Electronics Association
NTRIP	Networked Transport of RTCM via Internet Protocol
RMC	Recommended Minimum Specific GNSS Data
RTCM	Radio Technical Commission for Maritime Services
RTK	GPS Real-Time Kinematics
SBAS	Space-Based Augmentation System
TCP/IP	Transmission Control Protocol / Internet Protocol
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
UMTS	Universal Mobile Telecommunication System
UTC	Coordinated Universal Time
VCL	Vehicle Control Language
VTG	Course Over Ground and Ground Speed
WAAS	Wide Area Augmentation System
WGS 84	Department of Defense World Geodetic System 1984
WLAN	Wireless Local Area Network
WSRT JVM	WebSphere Real Time Java Virtual Machine