Abstract

This thesis describes autopilot simulation software designed to let a helicopter fly autonomously in a virtual game world. We continue the work from our previous study "Methods of Real-Time Flight Simulation" and describe a proof of concept implementation. We explore the challenges of state estimation, control logic, virtual sensors, physics and visualization and describe the methods we used. We identify some key concerns when designing software for autopilot simulation and propose a software architecture with tactics that promote modifiability, performance and portability to facilitate future extensions of the work.

We propose a method of autopilot control to safely navigate waypoints and our experiments verified that the autopilot could successfully navigate a variation of scenarios with different levels of state knowledge. We also propose a method for autopilot assisted control that enables a person with no flying experience to safely maneuver a helicopter by joystick.

The results verified the functionality of the GPS/INS Kalman filter implementation that significantly improved the position estimates. However, our position estimates were 40% more accurate than the results in related work when real sensors were used, which indicate a need for more accurate modelling of virtual sensors and physics to simulate autonomous flights with sufficiently realistic results.

The methods and findings in this thesis are viable for reuse in a number of autonomous control applications, including real navigation of unmanned aerial vehicles and dynamic AI control of aircrafts in computer games.

Acknowledgements

We would like to thank professor Torbjørn Hallgren for allowing us to take on this inter-disciplinary thesis and for his time spent on guidance and feedback during both the thesis and the pre-study. We are grateful for access to the high-end equipment and computers that were made available through his dedication to the visualization department at NTNU.

We want to thank co-supervisor Jo Skjermo for his technical insight and assistance in using the laboratory equipment and for feedback on the written reports. Finally, we would like to thank co-supervisor Helge Langseth for assistance on state estimation theory and feedback on the written reports.

Contents

List of Figures

List of Algorithms

Nomenclature

α Yaw angle.

β Pitch angle.

ψ Roll angle.

autopilot Control logic designed to navigate a machine autonomously.

BF Body Frame

BF Body Frame

CEP Circular Error Probable

DIY Do It Yourself UAV, an open source autopilot project.

ENU Earth east-north-up reference frame.

FDM Flight Dynamics Model, a method for describing flight behavior.

FGH Flat Ground Height, a method used by the range finder to estimate height above the ground.

FRU Forward-Right-Up naming convention for the axes in body frame.

Gimbal Lock At certain orientations in 3D two out of three gimbals are in the same plane losing one degree of freedom.

HAG Height Above the Ground

HLSL High Level Shader Language

IMU Inertial Measurement Unit measures change in linear and angular motion.

KF Kalman Filter

NED Earth north-east-down reference frame.

Orientation The direction an object is facing relative to some identity state.

PID A fundamental principle in control theory where proportional, integral and derivative measurements are used to minimize the difference of the current and the desired state.

Rotation A description on how to transition from one orientation to another.

Static Mesh Polygon mesh without animation.

True Observer Condition All sensors are noiseless and sample each timestep.

UAV Unmanned Aerial Vehicle

USV Unmanned Surface Vehicle

WGS-84 World Geodetic System 1984

1. Introduction

1.1. Purpose

This thesis presents our implementation of a simulator software for autonomous flight and the findings of experimenting with different sensor configurations. The implementation is largely based on our previous study "Methods of Real-Time Flight Simulation" [9] and as a proof-of-concept we achieved fully autonomous flight of a helicopter in a virtual game world.

The pre-study holds the details to the concepts of aerodynamics and methods of visualization so we will only mention those briefly where appropriate. Instead we focus here on the implementation of the methods, the software architecture designed to tie the components together and the findings of our experiments. After reading this thesis the reader should be well-equipped to understand the key concepts of regulatory systems and how we applied inter-disciplinary methods to implement a working autopilot.

1.2. Motivation

The field of robotics has gained significant interest over the last two decades and its applications are ever increasing. This project was inspired from the idea that swarms of cheap off-the-shelf model aircrafts could solve tasks safely and efficiently. The costs of issuing a search and rescue helicopter are tremendous and it takes a lot of time to cover a large area. With swarms one could ideally launch hundreds or even thousands of drones capable of navigating and identifying objects by themselves and sending live pictures to a ground station. This way a large area would be covered more quickly and potentially a lot cheaper. For hazardous tasks, such as in irradiated areas or in extreme weather conditions, the drones could be sent instead to avoid putting humans at risk.

In order to develop autopilots safely, efficiently and with minimal risk of failure it is necessary to have a realistic and flexible simulator software to test the autopilots on. Implementing such a simulator is the motivation for this thesis.

1.3. Context

There is a lot of prior work in the field of robotics and autopilots, but much of the available work is academic and difficult to apply in practice without in-depth knowledge. This thesis does an important job in identifying and combining academic methods from multiple disciplines of science such as mathematics, physics, cybernetics and computers. Our academic background was largely from computer science and game technology and this was to our advantage when describing our implementation for entry level readers, since we started out at an entry level in the field of robotics ourselves. The proof-of-concept implementation gives the thesis a practical perspective of things and should prove useful for anyone interested in simulating flight behavior and autonomous vehicles.

1.4. Intended Audience

This thesis is relevant for any reader interested in an entry level introduction to state estimation, autopilot control logic and flight simulation. We will cover the key methods used in our implementation and it is assumed that the reader has an academic background in computer science and a good understanding of graduate level calculus and physics.

1.5. Overview

Chapters 2 to 4 provide a summary of the key findings in the pre-study, similarities and differences of our solution compared to related work and provides a theoretical background required to understand the methods used and the choices made in the implementation.

Chapter 5 describes key implementation details and issues of the autopilot simulator and chapter 6 presents the results of the autopilot performance for different navigation scenarios with different sensor configurations. Chapter 7 discusses shortcomings, issues and how the proof-of-concept implementation matched our expectations and goals. We summarize the key findings in chapter 8 and present ideas for future work and enhancements in chapter 9.

Appendices A to D hold documentation of user manuals, configuration files, sensor datasheets and precision issue details.

2. Previous Work

Prior to this thesis we did a pre-study on methods of real-time flight simulation. The focus here was to identify methods and theory for simulating and visualizing aircraft flight. Here we will summarize the key findings relevant for our implementation.

Introduction to Aerodynamics The pre-study describes key aerodynamic phenomena behind heavier-than-air flight and for our implementation we focus on the forces of *lift* and *drag*. For helicopters, the main source of lift are the rotor blades of the main rotor. Figure 2.1 illustrates a simplified view of how lift is generated by placing a wing at an angle in an airflow. Although the process is more complicated than this, one can think of air as being accelerated downwards by the wing and the reaction force will cause the wing to accelerate upwards. The rotor blades work much in the same way as wings on airplanes and simply put; if the rotor speed is increased or the rotor blades are angled steeper, then the lifting force will increase.

Drag is typically an undesirable property when flying as it resists the motion of bodies through air. There a number of different types of drag, but the most significant types of drag for small model helicopters is *parasite* and *induced* drag. Parasite drag covers all forms of drag that are not related to the generation of lift. The most significant types of parasite drag are *form drag* and *skin friction*. Figure 2.2 shows how blunt body shapes have significant form drag, while streamlined shapes have more skin friction. Intuitively, form friction has a large impact on the magnitude of parasite drag so streamlined wings are preferred.

Induced drag is simply a result of the lift vector being directed up and slightly backwards due to the angle of attack of the wing. The force component that is opposite to the wing motion will then be perceived as drag.

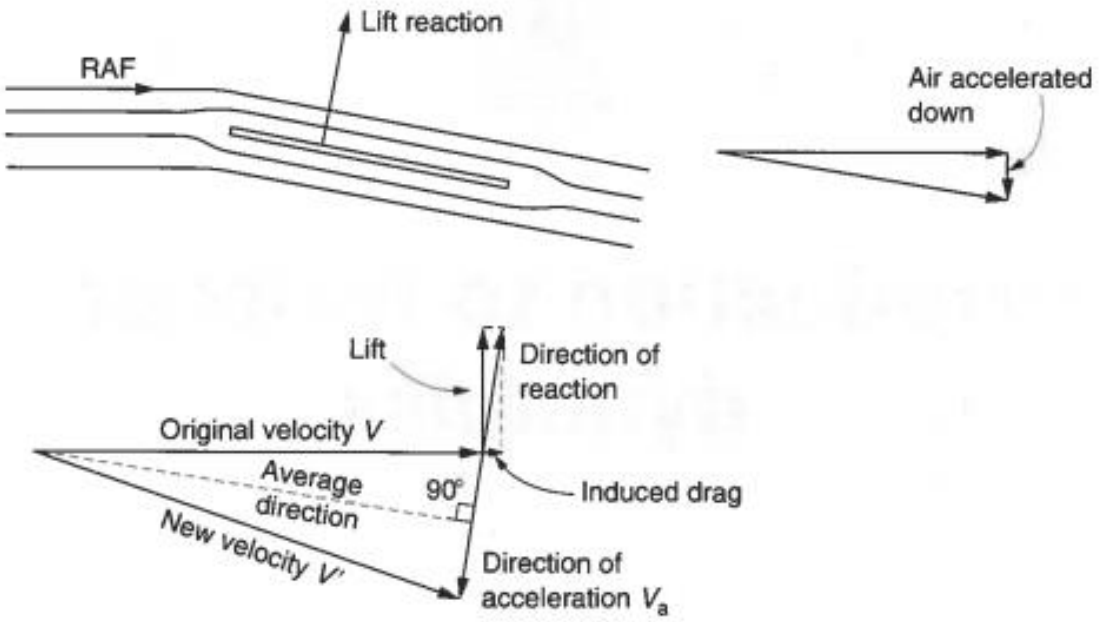


Figure 2.1.: Relative airflow passing over an airfoil and generating lift. Source: [1]

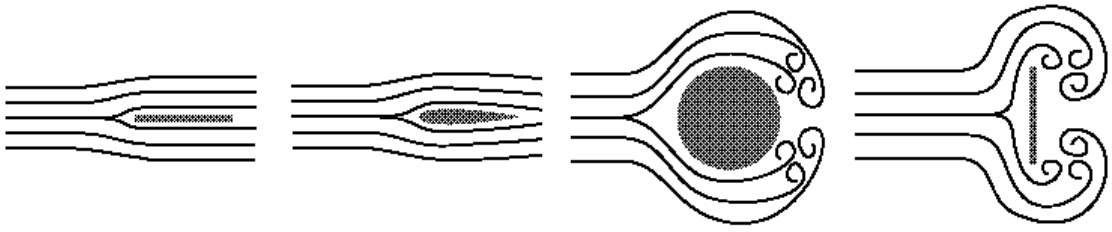


Figure 2.2.: Form drag and skin friction for different types of shapes. Source: [2]

Flight Dynamics Model The pre-study describes methods to analyze the aerodynamical forces for use in flight simulation. From these methods we proposed four types of flight dynamics models (FDMs) to define the flight behavior, as shown in table 2.2. The FDMs are listed as #1 being the most complex and comprehensive method and #4 being the simplest.

#	Method(s)	Model Description
1	CFD	Simulate the interacting forces of air and the aircraft structure by some governing equation for air
2	CFD + Parametric	Determine the lift and drag coefficients by simulating the air in a virtual wind tunnel
3	Parametric	Use published wind tunnel test data for specific aircraft models
4	Parametric	Pick lift and drag coefficients that give reasonable max velocities and control responses

Table 2.2.: Different combinations of computational fluid dynamics and parametric equations to model the flight dynamics.

There are two methods outlined in the table; Computational Fluid Dynamics and parametric equations. CFD attempts to simulate the effects of air around the aircraft by solving equations of fluid dynamics. Any substance that flows is classified as a fluid and this includes gases, such as air. The problem with CDF is that the equations are closely coupled and hard to solve without major simplifications. Solutions today are computationally expensive and require high resolution grids to accurately model the turbulent, high-velocity airflows around an aircraft.

Parametric equations are much simpler and if implemented properly can still present a good approximation. FDM #3 is often used in PC simulator games today and use published wind tunnel test data to realistically model the effects of varying airflows around specific aircraft models. FDM #2 is a combination and also used in modern simulator games. For real-time performance the method solves parametric equations based on CFD or structural analysis of the aircraft body. This analysis can be very time consuming, but allows the model to accurately represent the flight behavior of an arbitrary aircraft design.

In the pre-study we suggested to start out with FDM #4 and later extend to more realistic models if time allowed us to. As we describe in section 5.4, we ended up using

FDM #4 and choosing lift and drag coefficients on an empirical basis. Although the method is simple it proved to be a good approximation to small-scale helicopters with properly chosen coefficients.

3. Related Work

There is a lot of prior work in the field of autopiloting and here we describe a few projects that provided us with the necessary background to approach the task. We will briefly cover parts of the related work that were useful to us, how we made use of this information and discuss if our approach differed in any way.

3.1. GPS/IMU Data Fusion for USV

One article [10] describes a GPS/INS Kalman filter that takes context and validity domains into consideration to further increase the estimate reliability. Large parts of this article was relevant to us to get an understanding of how to apply a Kalman filter for GPS/INS filtering. In particular, we found that their sensor specifications matched up with the ones we were simulating so we could compare the performance of our state estimation to theirs. Their vehicle was not aerial, but as explained in section 5.5 we intentionally limited our vertical estimate error during flight to focus on the horizontal position estimate error. The test results showed that our simulation with similarly specified sensors produced 40% less position estimate errors and we discuss these findings later in chapter 7.

3.2. GPS/INS/Vision Data Fusion for UAV

One paper [11] describes how a vision system mounted on a helicopter could increase the state estimation accuracy of a GPS/INS system. We found this paper useful as an introduction to GPS/INS Kalman filtering and their vision system was a novel addition we had not seen before. They used a laser range finder to measure the vertical distance to the ground and a video camera system to estimate horizontal motion. To accurately

measure motion and height above the ground are some of the biggest weaknesses of GPS and INS systems so the vision system should prove useful for low altitude navigation scenarios as well as landing and take off.

In our implementation we implemented a sonic range finder that served the same purpose as the laser equivalent. This enabled our autopilot to perform low altitude navigation and to follow the curvature of the terrain without crashing.

3.3. Do It Yourself UAV

Do It Yourself UAV [3] (DIY) is an open source project that started in August 2001 and gained a lot of interest in the UAV community even though the project stopped its activities two years later. The goal was to establish a free software base and design a cheap hardware setup that any hobbyist could build and extend themselves for autonomous helicopter flight.

The project spans a working autopilot software, a list of sensors and components to use and complete electronic board designs to assemble the autopilot. The website has a few videos that showcase semi-autonomous outdoor flight, where cyclic and tail is fully controlled by the autopilot to hold a position, while the thrust is controlled by a human operator.

DIY is interesting because a tremendous amount of solid work was put down by a large number of people for the benefit of the community. It was one of the first hobbyist projects to succeed with a home user budget and the do-it-yourself recipe made it a pioneer in the hobbyist community.

The work of DIY is very related to this project and both try to achieve fully autonomous flight using sensors and autopilot software. However, we focus on simulating behavior and benchmarking autopilot performance in a virtual world whereas DIY started experimenting on real helicopters early on in the project.

The advantage of our approach is that it focuses on simulating the autopilot and its sensors to test out the autopilot performance. This allows for safer and cheaper experimenting and should prove more reliable if deployed on a real model helicopter later.

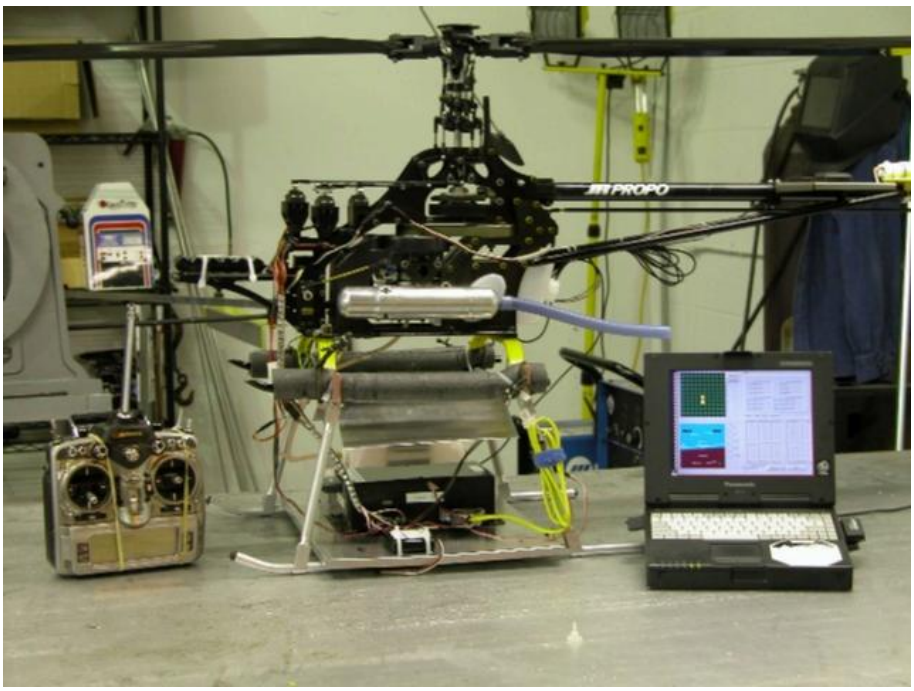


Figure 3.1.: The remote controlled helicopter with sensors mounted and the operator ground station. Source: [3]

3.4. MSc Thesis on UAV INS/GPS Navigation Loop

Sven Rönnbäck wrote a master's thesis [4] in 2000 on combining an inertial navigation system (INS) with a GPS system to estimate the flight path of an airplane drone. The thesis was written as Rönnbäck's contribution to the work of a large team of researchers on autonomous flight, depicted in figure 3.2, and has many similarities to our project. However, his work is solely focused on offline state estimation so there is no autopilot control logic or descriptions on using the state estimates to control a vehicle. Instead his work was based on logs of sensor data from manual test flights with their UAV to evaluate the performance of the state estimator by comparing the estimated and the true flight trajectories.

Our state estimation also relies on the INS/GPS Kalman filter method so his work was of great use for us to learn the ins and outs of aircraft state estimation. In section 5.6 we go into detail on how we implemented the state estimator and how it serves as an information source to our autopilot component.

Figure 3.2.: The research team and the UAV of Rönnbäck's work. Source: [4]

3.5. MSc Thesis on USV Controller Design

Geir Beinset and Jarle Saga Blomhoff wrote a master's thesis [12] in 2007 on the design of a heading autopilot and waypoint navigation system for a propeller-driven boat.

The problem they approached was to determine an accurate representation of the boat in a simulation software by doing actuator-response tests on the full scale rig. This way they could design the autopilot in simulation software before implementing and testing on the USV.

The relevant part of this thesis is how they make use of the control theory discussed in section 4.3 and how they use system identification to determine the boat's behavior for accurate simulation. In particular, the system identification method could prove valuable for our scenario to minimize the error in flight simulation. Interesting properties of a helicopter would be the responsiveness of thrust, cyclic and tail control at different relative air flows.

4. Theoretical Background

The previous chapters summarized the findings of our pre-study and related work. In this chapter we provide a thorough background to relevant theory of reference frames and orientational representations, regulatory systems and state estimation that are essential for the methods used in our implementation.

4.1. Reference Frames

In order to describe position, orientation and scale we need a reference frame and in autopilots we typically operate with two frames. One relative to the vehicle and one for navigation. This section will provide the background to understand the reference frames used in the implementation.

4.1.1. Body Frame

The body frame is the coordinate system relative to the vehicle. The frame is a Cartesian coordinate system and we chose to use a forward-right-up (FRU) naming convention corresponding to the -Z, +X and +Y axes in the XNA Game Studio framework that we implemented our virtual world in.

Body frame is used for the inertial sensors mounted on the vehicle since their values are relative to the mounting. For instance the accelerometer sensor measures acceleration in the body frame, assuming its mounting is aligned with the vehicle. In order to calculate changes in world position and velocity we must first transform the acceleration vector from body frame to navigation frame.

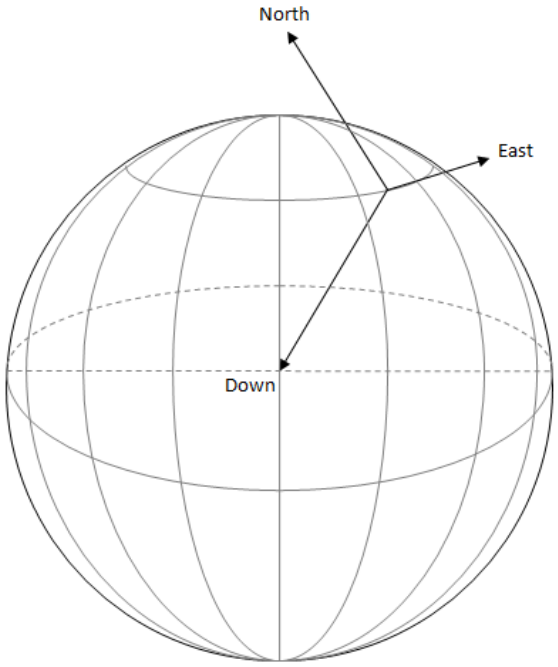


Figure 4.1.: Earth NED navigation frame.

4.1.2. Navigation Frame

In order to describe position, velocity and acceleration in the world we need a navigation frame. This frame can be said to represent a map and serves as a mapping between a position in the navigation frame and a position in the world.

4.1.2.1. Earth ENU/NED Frames

There are a number of common navigation frames, but one covered in [4] is the Earth North-East-Down (NED) frame as shown in figure 4.1. Here a position on the Earth surface is chosen as origin and then the position is described as a local north-east-down vector relative to the origin. When flying most objects of interest are below you so that is why the down vector is chosen in many aviation applications. Alternatively one can use the local east-north-up (ENU) frame, which is preferred for ground tracking applications such as GPS. These frames are typically used for short range navigation where the curvature of the Earth is considered negligible and this fit our navigation scenarios well. In our implementation we used NED.

4.1.2.2. Geodetic Frames

Although we used the NED frame in our implementation there are a number of other frames that one may have to consider for aircraft autopilots. For instance, flights over long distances require us to consider the elliptical shape of the Earth when representing the position, such as the World Geodetic System 1984 (WGS-84) [13]. WGS-84 describes a position by longitude, latitude and height and is widely used in maps and GPS trackers.

4.2. Representing Rotations

We used rotations for a number of objects in the world such as cameras, the helicopter and its main rotor. According to [14] there are three widely used methods for representing rotations; namely matrices, Euler angles and quaternions. Each have its pros and cons and we will identify how these methods fit our implementation.

4.2.1. Clarifying Terms

The terms *orientation* and *rotation* are often used interchangeably, but as noted in [14] there are subtle differences. An orientation can be considered as the direction an object is facing relative to some identity state, while a rotation (or *angular displacement*) describes how to transition from one orientation to another. In other words, applying a rotation to an orientation produces a new orientation. We will stick with this convention here.

4.2.2. Rotation Matrix

A rotation matrix represents a rotation, but more specifically it represents a transformation of vectors from one coordinate space to the other. In 3D the 3x3 matrix simply lists the three basis vectors $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$ and $\hat{\mathbf{w}}$ of one coordinate space expressed in the other coordinate space (XYZ).

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{u}}_x & \hat{\mathbf{u}}_y & \hat{\mathbf{u}}_z \\ \hat{\mathbf{v}}_x & \hat{\mathbf{v}}_y & \hat{\mathbf{v}}_z \\ \hat{\mathbf{w}}_x & \hat{\mathbf{w}}_y & \hat{\mathbf{w}}_z \end{bmatrix} \begin{matrix} (new\, x - axis) \\ (new\, y - axis) \\ (new\, z - axis) \end{matrix}$$

Summary of the rotation matrix representation:

Pros	Description
Explicit basis vectors	Because the rotation matrix is defined explicitly by the three new basis vectors they are readily available.
Concatenation	Multiple rotations can be concatenated into a single rotation matrix, which greatly simplifies nested coordinate space relationships.
Inverse rotation	The rotation from B to A is easily obtained by matrix inversion of the rotation from A to B. Since the basis vectors are orthogonal the inverse is simply the transpose of the matrix.
Graphics APIs	Modern graphics APIs such as OpenGL and DirectX use rotation matrices. This means the representation can be used directly without any transformation.

Cons	Description
Size	This representation is the largest of the three, using 9 numbers to represent an orientation. This is 3 times the cost of Euler angles.
Usability	This representation is not intuitive to work directly with for humans, as we cannot easily imagine a coordinate system by considering their three basis vectors.
Validity	It is possible for a rotation matrix to become invalid, causing its behavior to be unpredictable. If the basis vectors are not orthogonal or not of unit lengths the rotation matrix is not well-defined. This can happen due to floating-point round-off errors by *matrix creep* when concatenating many rotation matrices.

4.2.3. Euler Angles

In contrast to rotation matrices and quaternions the Euler angles explicitly represent an orientation. The idea is to represent an orientation as three sequential rotations around mutually perpendicular axes from an identity state. Rotations are not commutative so the order of rotations matters. Unfortunately there is no universal standard as there are different preferences to axes and orders. For instance, in physics the so-called *x-convention* uses rotations around X, Z and X while XNA uses the order Y, X and Z (yaw, pitch and roll). This means that in an implementation one needs to choose a

convention and stick with it. Since we are using the XNA framework we will be using Y-X-Z Euler angles.

Tait–Bryan angles is a standard convention in aviation used to define the aircraft attitude by yaw, pitch and roll around the Z, Y and X axes. This is identical to our XNA convention except that the names of the axes are chosen differently.

Summary of the Euler angle representation:

Pros	**Description**
Usability	Humans prefer to think in angles so working directly with Euler angles is intuitive.
Size	Using only three numbers to represent an orientation, Euler angles is the smallest representation of the three.
Validity	It is by definition not possible to get an *invalid* Euler angle representation.

Cons	**Description**
Aliasing	Euler angles suffers from *aliasing*; that one orientation is not uniquely represented by Euler angles. Multiples of 360° around any axis and other XYZ order conventions can be used to represent the same orientation.
Gimbal lock	Whenever the second rotation of a Euler angle representation is exactly ±90° the first and third rotations are restricted to rotate around the same axis; known as a *gimbal lock.*
Interpolation	Euler angles are difficult to smoothly interpolate due to their representation.

4.2.4. Quaternions

Quaternions overcome the problems with gimbal lock and aliasing by adding a fourth number to the representation. The math behind quaternions is complicated and largely based on complex numbers, however intuitively we can think of a quaternion as an axis-angle pair. It describes the rotation as a rotation axis $\overrightarrow{v}(x, y, z)$ and a rotation angle w around that axis.

Summary of the quaternion representation:

Pros	Description
Unique	All orientations are represented uniquely and overcomes the above mentioned problems with aliasing and gimbal lock.
Interpolation	Quaternions support smooth interpolations between orientations and [14] describes mathematical methods for both linear and cubic interpolation.
Size	The representation is only four numbers; 33% larger than Euler angles.
Concatenation	A sequence of rotations can be concatenated into a single quaternion.
Speed	According to [14] the performance of concatenation and rotation inversion is significantly faster than for rotational matrices and is faster than Euler angles to construct a rotation matrix from.

Cons	Description
Validity	The accumulated floating-point errors can render the quaternion representation invalid.
Usability	Quaternions are the hardest representation for humans to work directly with.

4.2.5. Converting Rotation Matrix to Euler Angles

We define the rotations yaw, pitch and roll (Y-Z-X) in XNA as the Euler angles α, β and ψ. According to [14] the rotation matrix $\mathbf{R}$ is defined as a product of three rotation matrices for the respective axes.

$$\mathbf{R} = Rot(y, \alpha) \times Rot(x, \beta) \times Rot(z, \psi) \tag{4.1}$$

We can express this matrix in terms of cosine and sine functions. In the matrix, c_x represents $\cos(x)$ and s_x represents $\sin(x)$.

$$\mathbf{R} = \begin{bmatrix} c_\alpha c_\psi - s_\alpha s_\beta s_\psi & -c_\beta s_\psi & c_\psi s_\alpha + c_\alpha s_\beta s_\psi \\ c_\psi s_\alpha s_\beta + c_\alpha s_\psi & c_\beta c_\psi & s_\alpha s_\psi - c_\alpha c_\psi s_\beta \\ -c_\beta s_\alpha & s_\beta & c_\alpha c_\beta \end{bmatrix} \tag{4.2}$$

From the method proposed in [15] we can identify the Euler angles from $\mathbf{R}$ by seeing the following relations.

$$\frac{\mathbf{R}_{31}}{\mathbf{R}_{33}} = -\frac{c_\beta s_\alpha}{c_\alpha c_\beta} = -\frac{s_\alpha}{c_\alpha} = -\tan(\alpha) \tag{4.3}$$

$$\mathbf{R}_{32} = s_\beta \tag{4.4}$$

$$\frac{\mathbf{R}_{12}}{\mathbf{R}_{22}} = -\frac{c_\beta s_\psi}{c_\beta c_\psi} = -\frac{s_\psi}{c_\psi} = -\tan(\psi) \tag{4.5}$$

From this we can derive our yaw, pitch and roll angles. Since there is a choice of four quadrants for the inverse tangent function one can use the $\arctan 2(a, b)$ function found in most math tools that handles this.

$$\alpha = -\arctan 2(\mathbf{R}_{31}, \mathbf{R}_{33}) \tag{4.6}$$

$$\beta = \arcsin(\mathbf{R}_{32}) \tag{4.7}$$

$$\psi = -\arctan 2(\mathbf{R}_{12}, \mathbf{R}_{22}) \tag{4.8}$$

It should be noted that as rotation matrices has an issue with singularities, the conversion to Euler angles also suffers from numeric instability. For instance when the pitch is exactly $+90°$ the yaw angle is no longer defined.

4.2.6. Other conversions

In our implementation we also used other conversions such as from a quaternion to a rotation matrix, but these were not necessary to implement since the XNA framework had methods to do the conversions for us.

4.2.7. Choice of Representations

We covered all three representations in detail since all of them were used in the implementation. Quaternions were used for camera and helicopter orientations in both the physics simulation and the state estimation components since they are fast, small, numerically stable and does not suffer from gimbal locks. The gimbal lock incident on the Apollo 11 moon mission is well known, because it caused critical navigation problems. Rotation matrices were used for rendering since the DirectX API requires us to and Euler angles were used to present the pitch, roll and yaw angles for debugging and autopilot configurations.

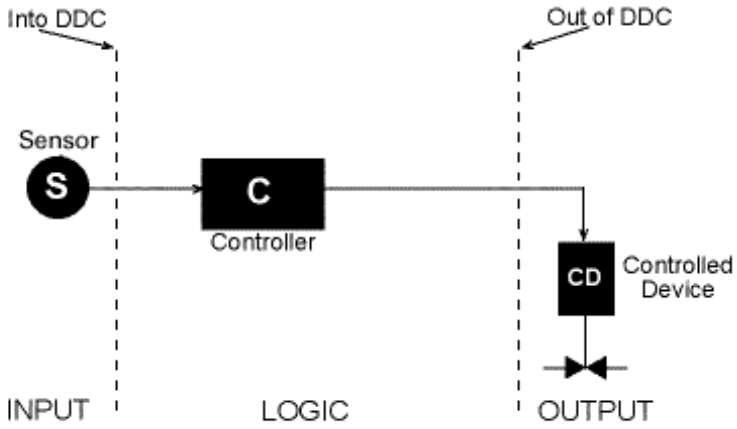


Figure 4.2.: DDC control loop. Source: [5]

4.3. Control Theory

The key problem of our implementation is how to design an autopilot that actually works. To attack this we need a fundamental understanding of control theory, what kinds of sensors exist and how they can be used to estimate the state of the helicopter. This section covers control theory and presents some simple examples on using PID-controllers to control a vehicle given knowledge about the vehicle position. The next section will introduce flight sensors and methods to construct this knowledge from noisy and incomplete measurements.

4.3.1. Direct Digital Control

One of the simplest forms of control is *direct digital control* (DDC). Figure 4.2 illustrates how a simple digital device compares a sensor output with a desired value and regulates the device to minimize the error. One example is using a controller to regulate the room temperature. The logic in the controller can be very straight forward such as increasing the heating effect if the room temperature decreases or it can use more sophisticated methods to better keep the room at a constant temperature. PID loop is one widely used method that allows the logic to be optimally tuned to a specific problem and is a building block for more complex controllers.

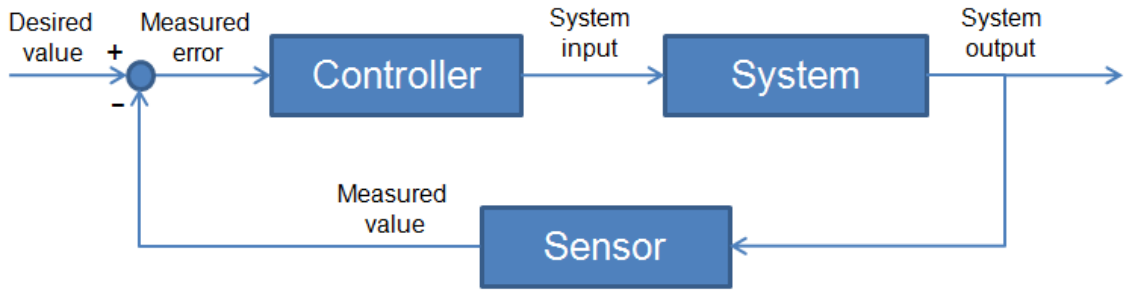


Figure 4.3.: Negative feedback loop.

4.3.2. PID Loop Controller

A PID loop is a type of feedback loop that controls the behavior of the system through negative feedback. The system output is subtracted from the desired value to obtain an error value that is fed back into the controller loop and corrected for, as shown in figure 4.3.

PID loops make use of the *proportional,* the *integral* and the *derivative* of the error, hence the name PID, to produce a control value that reduces the error. A PID controller has the general form [16]

$$u(t) = \underbrace{K_P e(t)}_{P} + \underbrace{K_I \int e(t)dt}_{I} + \underbrace{K_D \frac{d}{dt} e(t)}_{D} \tag{4.9}$$

where the three coefficients K_P, K_I and K_D are used to tune the relative weights of the P, I and D terms. P and D resemble a damped spring where the spring exerts proportional force if it is stretched or compressed from its resting position and the velocity (rate of change in spring length) is used to damp the oscillations and quickly come to rest. The integral term is the sum of error over time and has the effect of gradually amplifying the error reduction if the error persists.

4.3.2.1. Example of an autonomous train

Consider a modern train that is controlled by an autopilot. This example will show how this problem could be solved using a simple PID loop controller, odometer as sensor and throttle as the controlled device. For simplicity we assume the following:

- Throttle control value is clamped by $u(t) \in \mathbb{R}\,[-1, 1]$where positive values accelerate forwards and negative values backwards.
- The train accelerates linearly as a function of throttle, given by $a(u) = 4u(t)$ m/s^2.
- The train suffers from friction as a linear function of velocity, given by $f(v) = 0.1v$.
- The the train moves on rails so we only need to consider throttling in either direction.

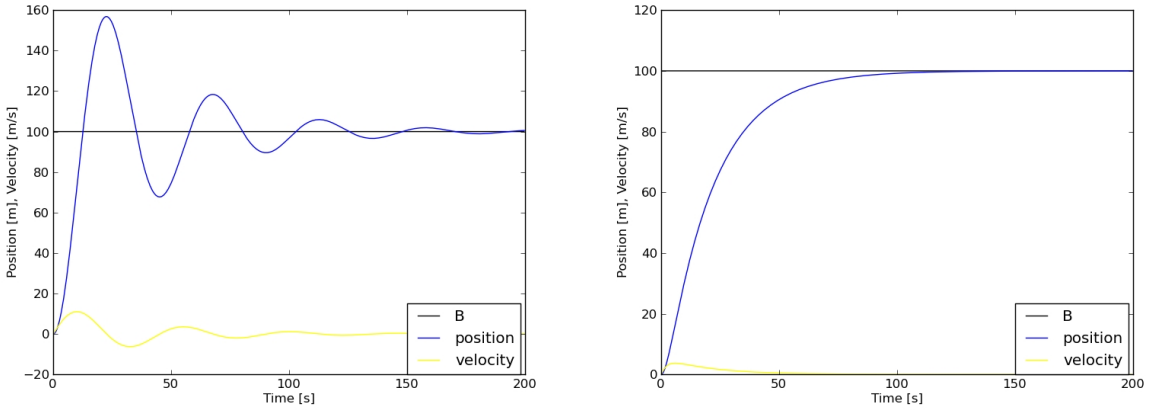
P-controller We will start by moving the train from A to B using only the P term. A is positioned at 0 meters and B at 100 meters and the initial error e(0) is -100m. K_p is chosen as $\frac{-1}{100m}$ so the throttle is immediately floored by the P term. When the train starts moving towards B the throttle is gradually relaxed and when we reach our destination the error becomes zero and the throttle is set to neutral. However, because the train has momentum it will overshoot B as figure 4.4(a) shows and oscillate back and forth across B until the train eventually comes to rest due to the damping effect of friction.

The oscillating behavior it not desired since the passengers will pass the train station several times before stopping. Although friction already dampens the oscillation we would like to overdamp the function so we stop at B on the first attempt. The D-term provides the damping we need, so let us examine how it works.

PD-controller With sufficiently large K_D the D-term gradually relaxes the throttle and starts breaking as we are approaching B at a velocity. Let us keep K_P and choose $K_D = \frac{-1}{5m/s}$.

It can be seen from figure 4.4(b) how the P coefficient is strong near the starting condition where the error is large and this accelerates the train quickly. Near 50 seconds the proportional error is starting to get small compared to the velocity and the effect of the D-term damping becomes dominant causing the train to slow down. The P-term will continue to "pull" the train towards B as a spring and the D-term will prevent the train from overshooting.

The train is now able to transport passengers smoothly from A to B because the coefficients are tuned for the behavior of this train, but what happens if the environment changes? Let us finally consider the I-term of the PID-controller.



(a) P-controller: Train overshoots its destination. (b) PD-controller: Train stops smoothly at its destination.

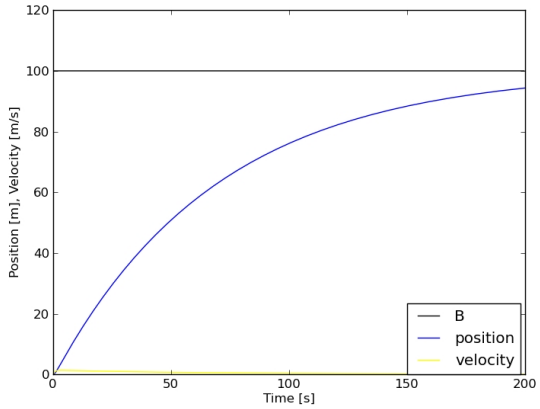
Figure 4.4.: The effects of applying P and PD controllers with appropriately tuned coefficients.

PID-controller In a perfect world the PD coefficients would be chosen to fit each application and achieve perfect behavior. However, in the real world of non-deterministic processes this is not so. Windy conditions, wear and tear on mechanical parts and uphill slopes complicate the problem of choosing coefficients that get the train to its destination in a quick, accurate and smooth manner. Figure 4.5(a) shows how an unforeseen increase in friction to $f(v) = 0.5v$ results in a dramatic delay. In this particular case it prolongs the traveling time from 150 to 400 seconds since the throttle is not applied efficiently. Note how the large friction alone is sufficient for the train to come to a stop so that the throttle never has to go negative.

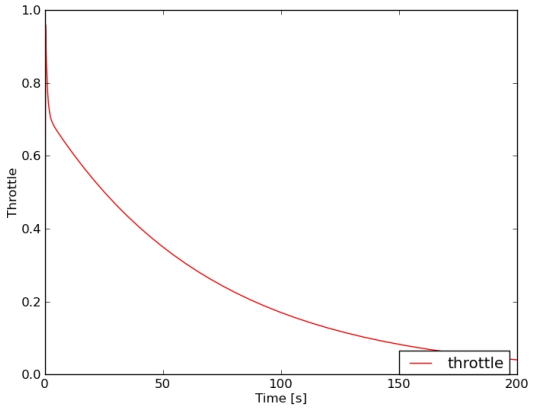
The I-term sums the error in position over time and accelerates the reduction of error the longer the error persists. This means a good choice of K_I will get the train to its destination more quickly if the train is starting to run late. K_P and K_D must be reconfigured to avoid oscillating behavior introduced by the I-term, so we choose $K_P = -1$, $K_D = -4$ and $K_I = \frac{1}{100000}$.

It should be seen in figure 4.6 how the throttle control is maxed by the P and I terms until the train nears its target at such a speed the D term begins to relax the throttle. As a result the travel time of the train has improved from 400 to 70 seconds with the same amount of friction.

Although this is a very simple example the nature of PID controllers have proven very reliable and applicable to most control problems in the real world. The difficulty of PIDs

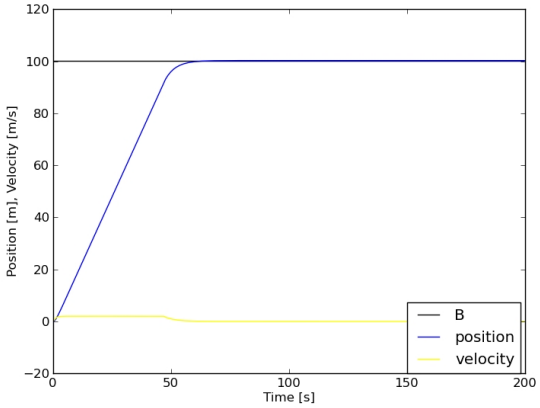


(a) PD-controller: An increase in friction dramatically delays the train.

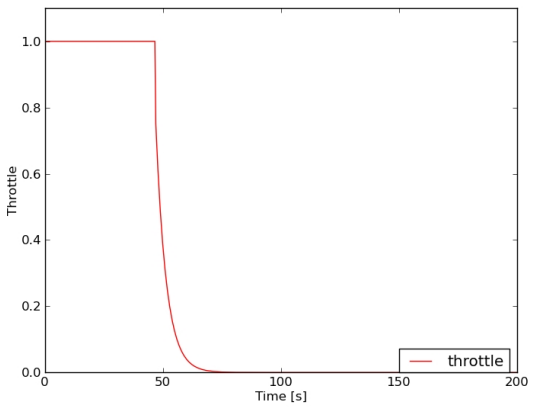


(b) PD-controller: The throttle is not used to its potential.

Figure 4.5.: The PD controller may not suffice if the environment changes from what it was tuned for.



(a) PID-controller: The I-term accelerates the train to minimize the travel time



(b) PID-controller: Throttle is more aggressive and overcomes the extra friction

Figure 4.6.: Overcoming dynamic changes in the environment using a PID controller.

are to choose good coefficients that perform satisfactory and for this we need methods to tune the coefficients.

4.3.3. PID Tuning

Before fiddling with the coefficients we should first realize that a PID-controller is not always the best fit. Getting all three PID coefficients to work in harmony is anything but trivial and we may successfully omit one or two to achieve better results. In [6, p. node63.html] the behavior of different controller types are discussed and figure 4.7 shows how they perform to each other in one particular scenario. Here the uncontrolled steady-state converges towards $y_\infty = 1$ and the controller's target value is $y_{ref} = 0$. The figure can be summarized as follows:

- P-controller: high overshoot, long settling time and large steady-state error
- I-controller: highest overshoot, no steady-state error
- PI-controller: high overshoot, no steady-state error
- PD-controller: medium overshoot, medium steady-state error
- PID-controller: smallest overshoot, no steady-state error

At first glance it would seem obvious that the PID-controller is superior to the other controllers, but that is not necessarily true for other scenarios. The fact is that PIDs are difficult to tune properly and with untuned coefficients the controller will suffer from poor performance or go unstable.

For instance, if our top priority is minimal steady-state error then an I- or PI-controller may suffice or to achieve fast response a P or PD controller might work well. The advantage with simpler controllers is that the coefficients are much easier to tune and to keep stable. So first we need to decide what the optimal response properties are, then choose a matching controller and finally tune the coefficients. In section 5.7.2 we explain how we use P-controllers to control the helicopter attitude, while the throttle required a PID-controller for stable altitude control.

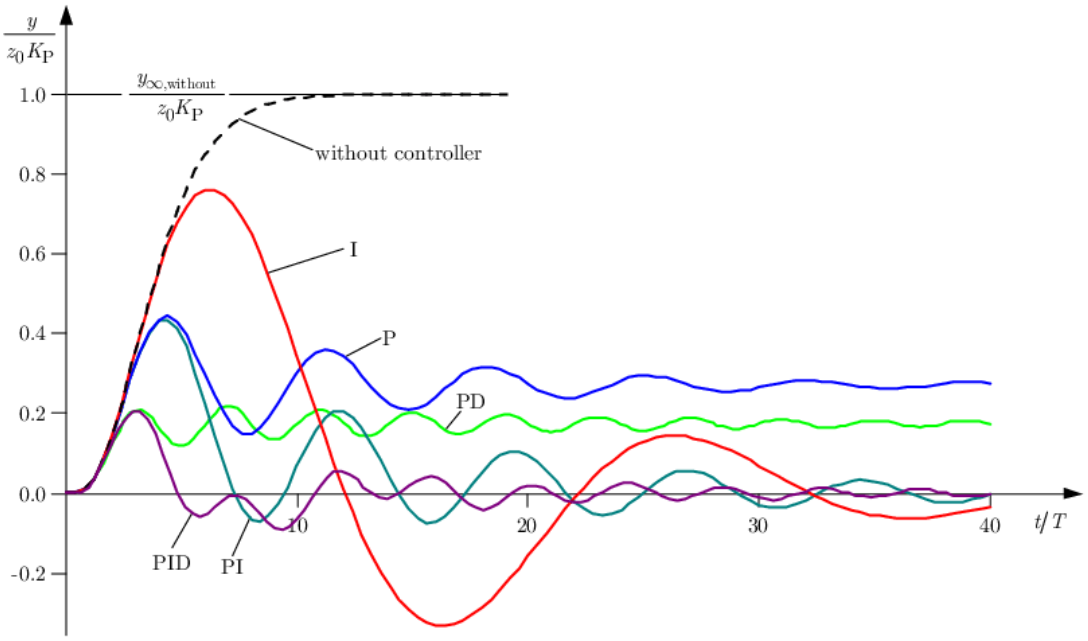


Figure 4.7.: Comparison of P, I, PI, PD and PID controllers. Source: [6, p. node63.html]

4.3.3.1. Trial and Error

The most straight forward method of tuning is to tweak each property and see if the result gets any better. The following 5 steps [16] are the basis of manually tuning a PID controller:

1. Set $K_P = K_I = K_D = 0$.
2. Increase K_P until the output oscillates, then reduce the K_P until the amplitudes reduce by approximately 75% each period.
3. Increase K_I until any offset is corrected for within sufficient time.
4. Increase K_D until the output converges to an acceptable error within sufficient time.

    a) A speedy configuration use smaller values of K_D and typically overshoots slightly to converge more quickly.

    b) If overshoots are not acceptable (do not pass the train station) then increase until the output is sufficiently over-damped as seen in figure 4.4(b).

5. Experiment by tweaking the values near this base setting to improve the result.

It is often required to manually fine-tune the PID and table 4.1 summarizes the effects of the three coefficients. The controller can be sucessfully optimized using trial and error with a little bit of experience and knowing how these effects interact.

Parameter	**Rise time t_r**	**Overshoot M_p**	**Settling time t_ε**	**Steady-state error**
K_P	Decrease	Increase	-	Decrease
K_I	Decrease	Increase	Increase	Eliminates
K_D	-	Decrease	Decrease	-

Table 4.1.: Effects of increasing PID coefficients. Source: [17]

Ziegler-Nichols Method Ziegler and Nichols derived a table based on their experience with trial and error that helps create a stable base setting for different controllers. Although the values are derived from assumptions on the environment model, this approach has been widely accepted in the industry as a standard in control systems practice.

4.4. State Estimation

Now that we have a understanding of control theory and regulatory systems we need to provide a feedback to the autopilot controller logic. The feedback is typically the variables such as the position, velocity and orientation of the helicopter and this section discusses how to estimate these from incomplete and noisy sensor measurements.

4.4.1. Flight Sensors

In our implementation we use sensors to determine the current state of the helicopter and this subsection covers what state information is required for autonomous flight and what types of sensors are available to provide this.

State Information Required For Helicopter Maneuvering The primary objective of our autopilot is to avoid crashing into the ground or into obstacles. We may also want it to navigate from A to B as long as that does not conflict with the first objective. To

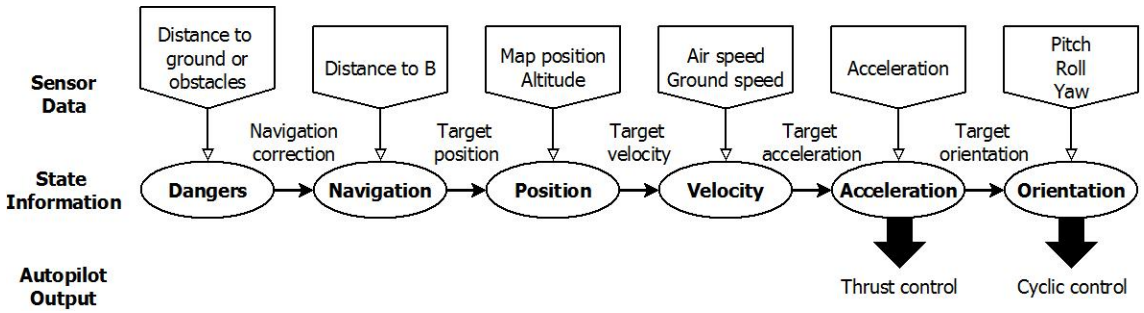


Figure 4.8.: Hierarchy of information and chain of control for a helicopter autopilot.

achieve either one we need to know at the very least where we are, where we are headed and how to control these two.

A simplified hierarchy of state information is shown in figure 4.8 and this illustrates how a chain of control may be applied for a helicopter autopilot. In the end the only means of maneuvering a helicopter is to apply thrust and cyclic, as discussed in [9, ch. 2.1].

The autopilot output is typically determined by a basic navigation state unless there are immediate dangers that override it. In order to navigate to B the autopilot must minimize the positional error and intuitively we can achieve this by controlling the velocity. The velocity is in turn a result of forces that act on the helicopter over time. Although we cannot control the environmental forces we are able to apply thrust and direct it using the cyclic to move towards B.

The same basic principles apply for any aircraft. Now that we have an understanding of what information is required for an autopilot to function we can look at what sensors exist to provide this.

Actual and Inferred Measures We distinguish between two types of sensor output; actual and inferred. All sensors measure one or more physical properties directly such as relative air speed and altitude. However, most measurements can also infer extra information using mathematics and physics. The derivative and integral relationships between position, velocity and acceleration is well known, but the integration suffers from numerical diffusion. For instance, small errors in acceleration measurements have a large impact on positional estimates by dead-reckoning, because the errors are double-integrated over time.

A list of flight sensors relevant for autopilot systems are listed in table 4.2. In general the quality of a sensor can be described by four properties:

Sensor	Measures	Inferred Information
Altimeter	Height over a reference point	Height over the ground
3D Compass	Relative magnetic field fluxes	Orientation
GPS	Global position, velocity	Height over the ground
Range Finder	Distance to nearest surface in a direction	Height over the ground, distance to obstacle
IMU	Acceleration, angular velocity	Position, velocity, orientation

Table 4.2.: Flight sensor types; what they measure and what information can be inferred

- Accuracy: its ability to measure a physical property.
- Precision: its ability to numerically represent the measurement.
- Frequency: number of samples per second.
- Noise.

Obviously we want to optimize these properties, but for a small airborne drone this is limited by cost, size and weight. This means we need to look at methods of estimating the state based on inaccurate, infrequent and noisy samples of the world and the Kalman filter is a good fit for just that.

4.4.2. Kalman Filter

The Kalman filter is a recurring choice in automation tasks due to its ability to fuse the information from multiple sensors by modelling the sensor properties. In [7] the Kalman filter (KF) is described as a recursive stochastic technique that estimates the state of a dynamic system from a set of incomplete and noisy observations. In our case of navigation this can be used to estimate the helicopter position, velocity and orientation from unreliable sensor measurements in the world.

Formally, KF solves the problem of estimating a true state $\mathbf{x}_k$ that is governed by a linear stochastic difference equation

$$\mathbf{x}_k = \mathbf{A}_{k-1}\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k, \tag{4.10}$$

given an observation $\mathbf{z}_k$ per state by

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k. \tag{4.11}$$

Table 4.3 summarizes the symbols. Figure 4.9 illustrates how the real state $\mathbf{x}$ is hidden from us and we see the that the following information is available:

- We define a state transition model $\mathbf{A}_{k-1}$ for how we expect the previous state $\mathbf{x}_{k-1}$ to propagate on its own with no control input.
- We know the current autopilot output $\mathbf{u}_k$ and define a control model $\mathbf{B}_k$ of how that output is expected to change the state.
- We observe the current true state $\mathbf{x}_k$ as $\mathbf{z}_k$ and define an observation model $\mathbf{H}_k$ that transforms the state to the respective observations.
- The process noise $\mathbf{w}$ and the observation noise $\mathbf{v}$ can be modelled if their statistical distributions are known.

It has been proven [18] that KF estimates a state with minimum mean square error for linear systems given that:

1. The process model noise is white and Gaussian with distribution $\mathbf{w} \sim \mathcal{N}(0, \mathbf{Q})$.
2. The observation model noise is white and Gaussian with distribution $\mathbf{v} \sim \mathcal{N}(0, \mathbf{R})$.
3. The two are independent of each other.
4. The initial condition (initial estimated state) is set to the actual initial system state excluding process noise.

In other words, KF provides an optimal guess of the current flight state given a reliable starting point and unreliable flight sensor data up to that point.

Prediction The KF algorithm is described in [7] as having two steps and is illustrated by figure 4.10. The first step calculates a predicted (a priori) next state $\hat{\mathbf{x}}_k^-$ from its previous estimated state $\hat{\mathbf{x}}_{k-1}$ based on how we have modelled the state to propagate on its own (state transition model $\mathbf{A}_{k-1}$) and how the current autopilot output will affect the next state (control model $\mathbf{B}_k$ and control input $\mathbf{u}_k$) from equation 4.10. This prediction, as one can see, is made without any observations and simply follows the modelled process.

$$\hat{\mathbf{x}}_k^- = \mathbf{A}_{k-1}\hat{\mathbf{x}}_{k-1} + \mathbf{B}_k\mathbf{u}_k \tag{4.12}$$

$$\mathbf{P}_k^- = \mathbf{A}_{k-1}\mathbf{P}_{k-1}\mathbf{A}_{k-1}^T + \mathbf{Q}_k \tag{4.13}$$

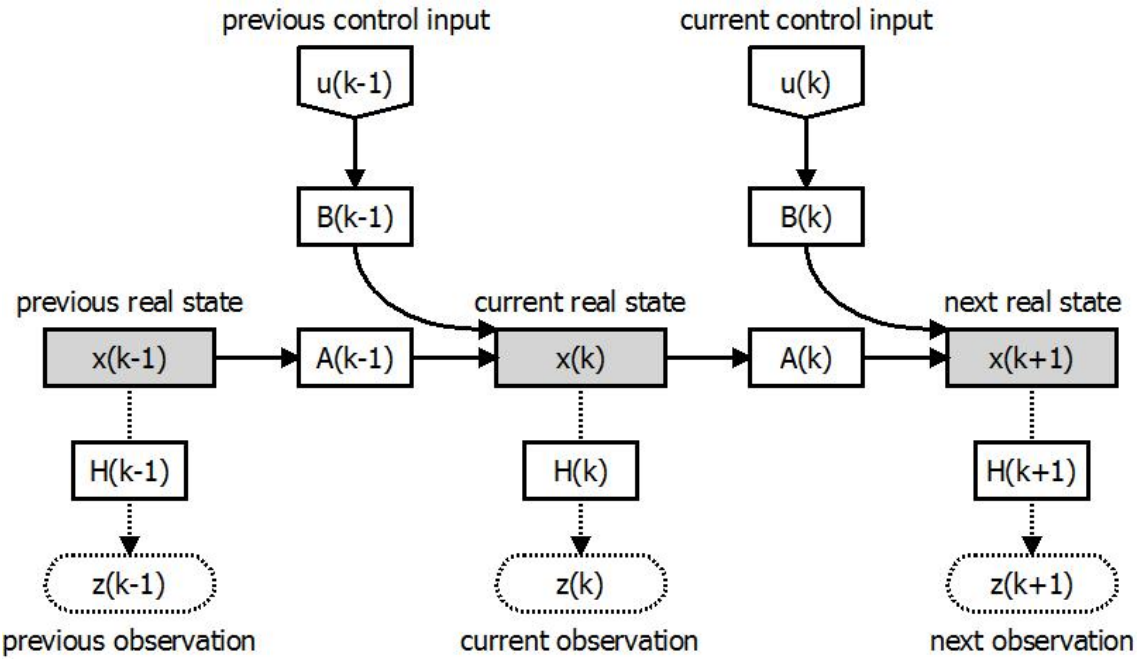


Figure 4.9.: System states $\mathbf{x}$, control inputs $\mathbf{u}$ and state observations $\mathbf{z}$. Noise is omitted for clarity.

Symbol	General Description	Contextual Interpretation
$\mathbf{A}_k$	State transition model that evolves a state by its underlying process model	Equations that govern world physics
$\mathbf{B}_k$	Control model that maps control input into changes in true state space	Equations that map autopilot output to changes in helicopter state
$\mathbf{u}_k$	Control input vector	Autopilot output in terms of pitch, roll, yaw and thrust
$\mathbf{H}_k$	Observation model that maps true state space into observed space	Equations that map helicopter position, velocity and orientation to respective sensor measurements
$\mathbf{w}_k$	Process noise	The world physics is not entirely represented by the modelled state transition A
$\mathbf{v}_k$	Observation noise	Noise in the observations due to sensor precision and accuracy

Table 4.3.: Description of the symbols in equations 4.10 and 4.11.

$\hat{\mathbf{x}}_k^-$ Predicted state estimate based on the previous state estimate, the previous control input and the known process model.

$\mathbf{P}_k^-$ Predicted estimate covariance describes the uncertainty of the predicted estimate.

Correction When the system enters state k an (unreliable) observation $\mathbf{z}_k$ is made. This observation is then used to correct the former prediction of the state and establish a corrected (a posteriori) estimate $\hat{\mathbf{x}}_k$. An important part of the correction step is to compute a *Kalman gain* $\mathbf{K}_k$ that minimizes the covariance of the mean squared estimate error. The gain can be considered a weighting function that balances its trust between the predicted state $\hat{\mathbf{x}}_k^-$ and the observation $\mathbf{z}_k$.

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^- \tag{4.14}$$

$$\mathbf{S}_k = \mathbf{H}_k\mathbf{P}_k^-\mathbf{H}^T + \mathbf{R}_k \tag{4.15}$$

$$\mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^T\mathbf{S}_k^{-1} \tag{4.16}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k\tilde{\mathbf{y}}_k \tag{4.17}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_k^- \tag{4.18}$$

$\tilde{\mathbf{y}}_k$ Innovation of the observation versus the expected observations for the predicted state.

$\mathbf{S}_k$ Covariance of innovation.

$\mathbf{K}_k$ Optimal Kalman gain describes its distribution of trust between the predicted state and the observed state.

$\hat{\mathbf{x}}_k$ Corrected state estimate is the final estimate for state k.

$\mathbf{P}_k$ Corrected estimate covariance is the uncertainty for estimated state $\hat{\mathbf{x}}_k$.

See in figure 4.10 how the first step predicts the next state prior to observation and then later corrects this estimate by the error between predicted and observed values. The corrected value is then used to predict the next state and so it continues cyclically. The iterative operation of the two steps and the background for these equations are covered in detail in [7] and [18]. Note that there are some differences in how the index k is applied depending on the models.

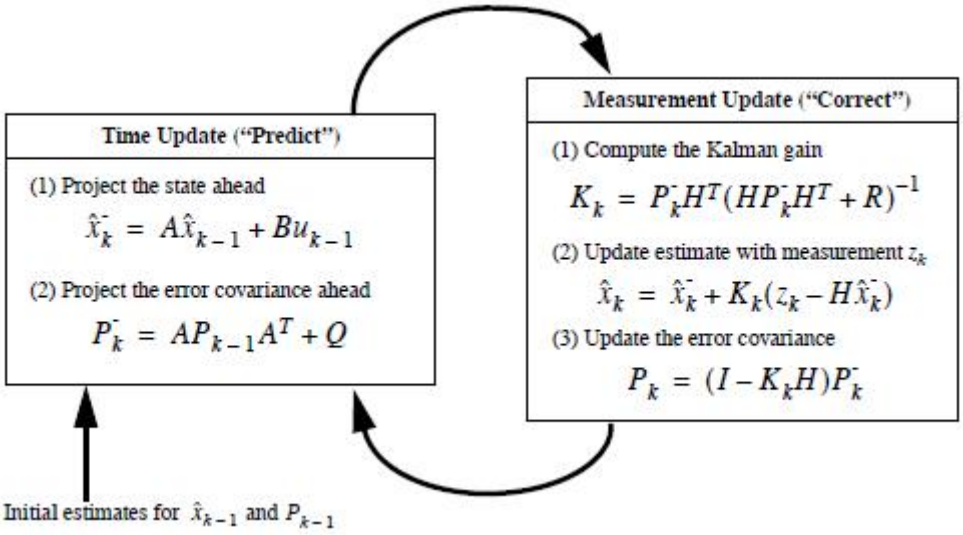


Figure 4.10.: The two steps in the Kalman filter operation. A and H are shown as constants here. Source: [7]

Kalman Filter in Practice The motivation for using KF is to achieve reliable estimates from noisy and incomplete data. This section presents an example where the 2D position of an object is being tracked by a positioning sensor. Figure 4.11 shows the position observations as red dots along the real path of the object.

Figure 4.12 shows how the filter is able to crudely match its estimates with the real trajectory and it is a huge improvement over the original plots with significantly less noise. But it should also be seen that the path generated by the Kalman filter is still very rugged and the sudden changes in estimated velocity would probably cause the controller to become jittery. Smoothing the estimates would help, but autonomous vehicles need to navigate based on the most recent estimate so any smoothing is restricted to the history of estimates.

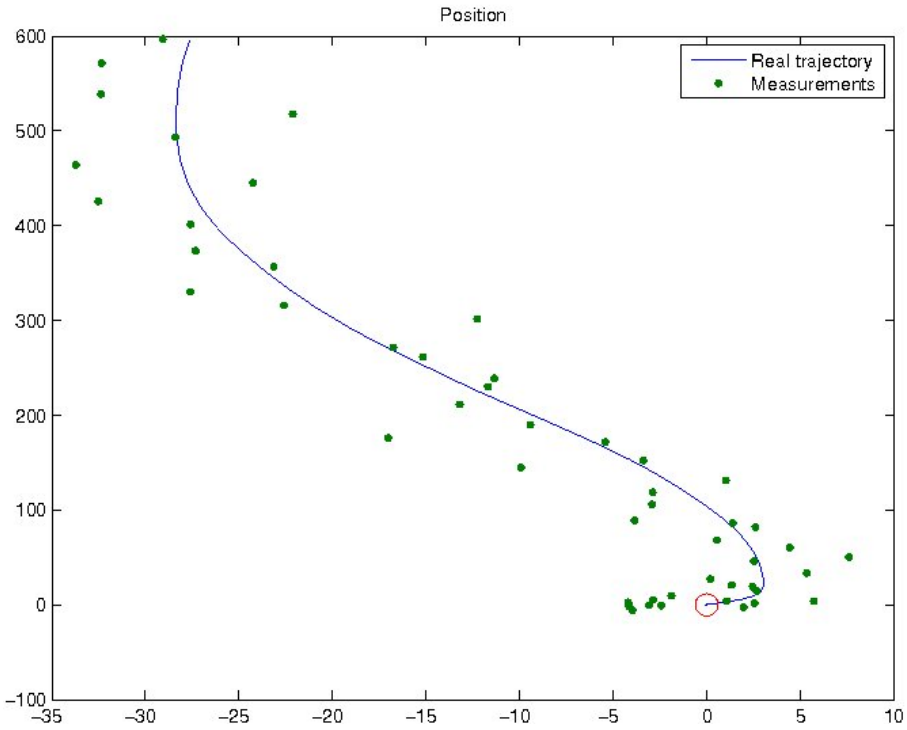


Figure 4.11.: Measured position as dots and actual path as a curved line. Source: [8]

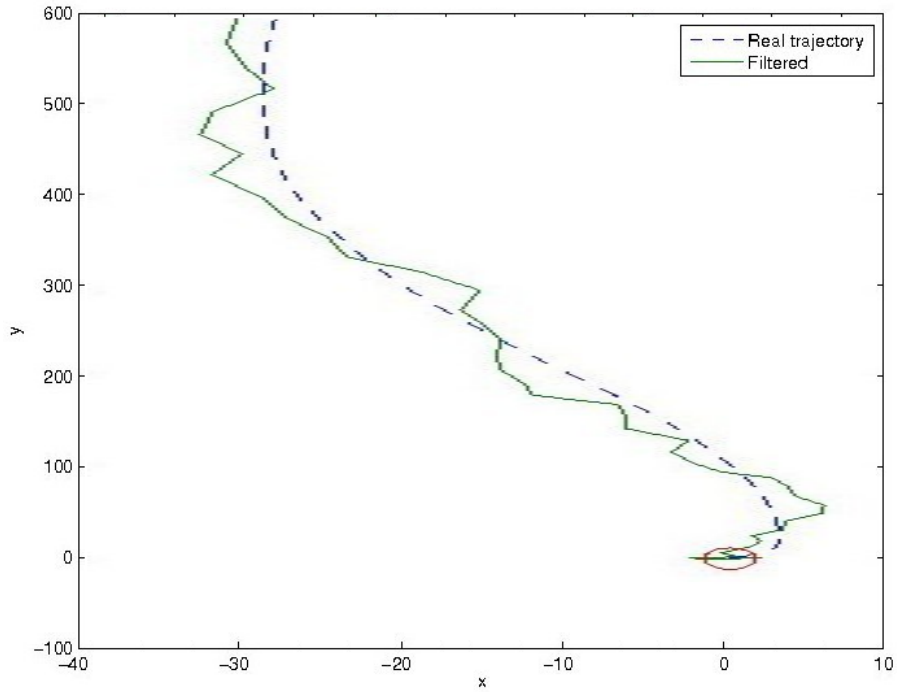


Figure 4.12.: Estimated position using Kalman filter. Source: [8]

Obviously there is no substitute for more accurate sensor data, but the Kalman filter does a good job in filling the gap. It has proven valuable in minimizing the estimate error and fusing overlapping information from several sensor types to more accurately estimate information about the state. In particular, [10] discusses how an inertial measurement unit (IMU) and a GPS may be combined in a Kalman filter to more accurately estimate the position. The IMU has a high accuracy in temporal motion but tends to drift over time, while the GPS is infrequent and inaccurate but provides absolute measurements that counter the drift. In section 5.6 we discuss how we implemented a GPS/INS Kalman filter for the helicopter state estimation.

Non-Linear Systems The Kalman filter is limited to linear systems and will perform poorly otherwise. There are a number of variations that try to linearize the problem such as the Extended Kalman Filter, Unscented Kalman Filter and Potter Square Root Filter, but we will not cover them here. As explained in section 5.6 we did not have time to add orientation estimation to the Kalman filter and omitted the problem of non-linearity in our implementation.

5. Implementation

This section covers all the important aspects of our implementation. An autopilot simulator is very comprehensive and involves physics simulation, virtual sensors, autopilot logic and visualization techniques. In this section we discuss each part and the software architecture that tie these components together.

5.1. Dependencies

We used a number of freely available code libraries to aid in the development and the following are required for the simulator to compile and run.

XNA Game Studio 3.1 XNA Game Studio (EULA) is a framework and a set of tools to help develop 2D and 3D games on the Microsoft platform; including PC, Xbox 360 and Zune. The XNA framework is a managed code wrapper for the DirectX API and makes it easy to use DirectX from managed code languages such as C# and Visual Basic. Managed code simplifies the development process since the allocated resources are automatically garbage collected and there are powerful refactoring tools available to help maintain the code.

Math.NET Iridium v2008.8.16.470 Iridium (LGPL license) is a math library that offers basic numerics, linear algebra, random generators and more. In our implementation we used Iridium to perform matrix operations in the Kalman filter.

NUnit 2.5.3 NUnit (BSD license) is an open source unit testing framework for all .NET languages and was originally ported from JUnit for Java. We used NUnit to create unit tests in our implementation in order to ensure the correctness of pieces of code.

Swordfish Charts 0.6 Swordfish charts (BSDlicense) were used for dynamic graphing and illustrating flight trajectories of the helicopter. This .NET graphing tool uses WPF vectorized graphics and allows one to zoom without rasterization artifacts.

Jitter Physics Engine 0.1.0 (beta) Jitter (free for non-commercial use) is a lightweight .NET managed physics engine that is designed for use in .NET and XNA. The engine originates from the widely used open source JiglibX library and has been extended with new features. Jitter is not open source, but is free to use for non-commercial applications. In our implementation we use Jitter to handle collisions between the helicopter and the terrain to prevent the helicopter from flying through the ground and to enable the helicopter to land.

VrpnNet 1.1.1 The Virtual Reality Peripheral Network (VRPN) (MIT license) [19] is a public-domain software released by the University of North Carolina. The software includes both server and client and allows for easy retrieval of virtual reality sensor data over a network. VrpnNet is a library that allows .NET applications to easily interface with VRPN and the connected sensors.

5.2. Software Architecture

When designing the software architecture we needed to consider the major concerns for the quality of the software. A good architecture not only eases the effort of working with the code, but also promotes desired system qualities that is considered important for the overall quality. In this section we will identify key requirements and tactics and give an overview of the architecture we used for our implementation.

5.2.1. Requirements

Being a thesis the implementation was subject to future extensions and reuse so *modifiability* was a big concern. Also the application was intended to run a real-time simulation so *performance* was an obvious concern. Finally the implementation was intended for future use in embedded systems so we also needed to account for *portability.* We isolated

a few key quantitative and qualitative requirements to identify tactics and promote the three aspects of the software.

#	**Requirement Description**
R1	It should be easy to add new sensor types and modify sensor configurations.
R2	The physics simulation should be replaceable and easy to modify.
R3	Modifying the autopilot behavior should not affect other components.

Table 5.1.: Modifiability requirements.

#	**Requirement Description**
R4	Simulation should run at interactive frame rates ($>$ 10 fps) on the test computer setup[a] .
R5	Sensors should be able to run at different sampling rates up to 100 Hz.

[a]See section 6.1 for details.

Table 5.2.: Performance requirements.

#	**Requirement Description**
R6	The autopilot code and its dependencies can be written in the programming language C.

Table 5.3.: Portability requirements.

All requirements were satisfied in the implementation except for R5. It was problematic to simulate high-rate sensors and sensors running at different rates than the game loop. This is explained in detail section 5.5.1.

5.2.2. Tactics

#	Req.	Tactic Description
T1	R1,R2,R6	Loose coupling by unified interface
T2	R1,R2,R3,R6	Minimize simulator dependencies
T3	R2,R3,R6	High cohesion by grouping code into package assemblies
T4	R6	Minimize platform specific dependencies (.NET and XNA)
T5	R1, R2	Use inheritance for variations of the same component type
T6	R5	Run updates separately from rendering in game loop

Table 5.4.: Tactics to promote modifiability, performance and portability.

Not all requirements can be solved by architectural tactics. R4 is typically limited by the render pipeline for outdoor scenes and so the frame rate will be controlled by adjusting the number of scene objects and their detail levels.

5.2.3. Trade-Offs

The most significant trade-off in our architecture tactics is compromising performance to gain modifiability and portability. There is some overhead in boxing and unboxing data structures for each method call and loosely coupled code as promoted by tactics T1 and T3 have longer call stacks and will suffer slightly in performance from this. Calling methods through interfaces and virtual methods as proposed by T1 and T5 and converting .NET and XNA types to portable primitives as proposed by T4 will also introduce some overhead. However, we consider the final performance hit negligible for our simulation purposes and that the gains in modifiability and portability justify the costs.

5.2.4. Reference Architecture

There a number of major patterns for robot architecture patterns, including *control loop* [20], *layered* [21]and *blackboard* [22]. Each pattern also has a number of widely used reference architectures to inspire the design. The main advantage of blackboard over

layered was efficiency by parallel processing, which was not relevant to us, so we chose a combination of control loop and layered patterns.

The layered pattern was used as an overall architectural design to separate components into abstraction levels. Elfes [21] is one widely used reference architecture, illustrated by figure 5.1. The mapping from Elfes to our implementation is also shown in the figure where each layer lists its corresponding classes. It can be seen that we deviated slightly from the reference in that our Autopilot spans both Supervisor and Global Planning layers and that OutputController spanned Control and Navigation layers. This was mainly due to the smaller size of the implementation and it did not make sense to add the extra levels of abstraction.

The layered architecture allows one to reuse layers in future variations of the autopilot and to comply with standard interfaces for interoperability with off-the-shelf components. Any code changes are isolated to within individual layers and this is a good fit for requirement R3. On the downside it can be difficult to establish a correct granularity of layers and to decide what components belong to each layer. Also, once the architecture is designed it can be problematic to later redefine the behavior of a layer as there are a number of dependencies that rely on it.

A variation of the control loop pattern was already described in section 4.3.1 and figure 4.2 illustrates how a controller computes output from input. We used the control loop pattern in our OutputController component, where input was the measured world state and output was the joystick output computed by the navigation logic and PID controllers.

5.2.5. Class Diagram

The final architectural design is illustrated by a class diagram and a component data flow diagram in figures 5.2 and 5.3. The class diagram shows how we divided key components into separate packages to preserve high cohesion as dictated by tactic T3. Tactic T1 is achieved by interfaces for state providers, physics components and sensors. Tactic T2 is achieved by using PhysicalHeliState as the coupling between our physics simulation and the portable SensorEstimatedState state estimator. The remaining tactics are implementation details and not illustrated here.

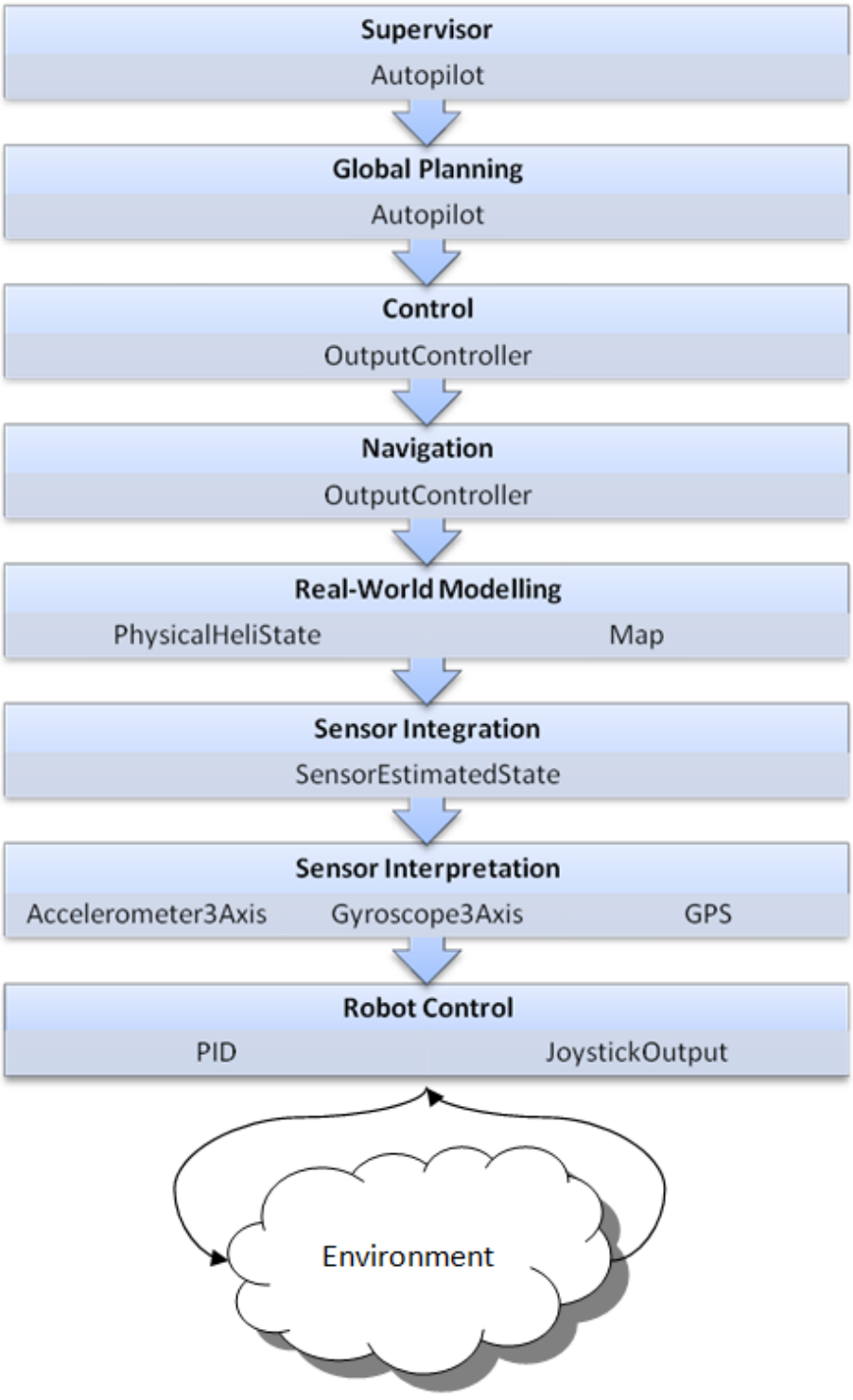


Figure 5.1.: Elfes reference architecture.

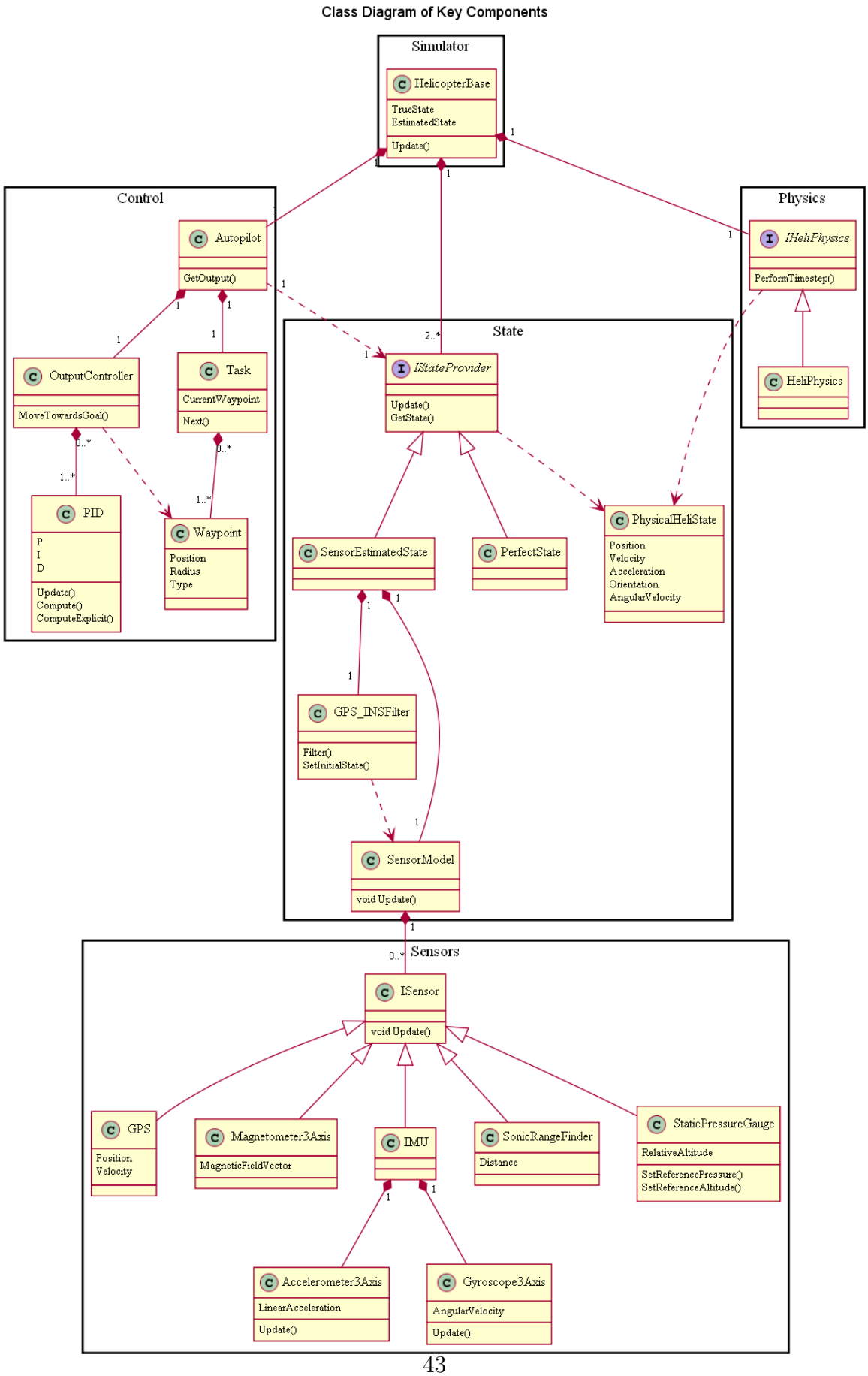


Figure 5.2.: Class diagram highlighting the coupling of key components and their packages.

5.2.6. Component Data Flow Diagram

The data flow can be considered cyclic since it runs in a game loop. Starting with the autopilot, figure 5.3 shows how the autopilot has an initial guess of its current state. Depending on the state the autopilot decides how to issue joystick output to maneuver the helicopter. Optionally the autopilot can be overridden and flown manually by joystick. The joystick output is then input to our helicopter physics, which uses a flight dynamics model and collision handling to decide the new helicopter state at the end of the timestep. At this point the renderer will draw the helicopter at its current position and orientation.

Now that we have simulated the timestep we can simulate the sensors. For the state estimator to be synchronized with the physics simulation we must provide inertial measurements from the start of the timestep and GPS measurements from the end of the timestep. This way the GPS/INS Kalman filter can predict the change in state based on the IMU and correct the estimated state based on the GPS observations. The estimated state is then fed back into the autopilot and this concludes the cycle, which is repeated over and over.

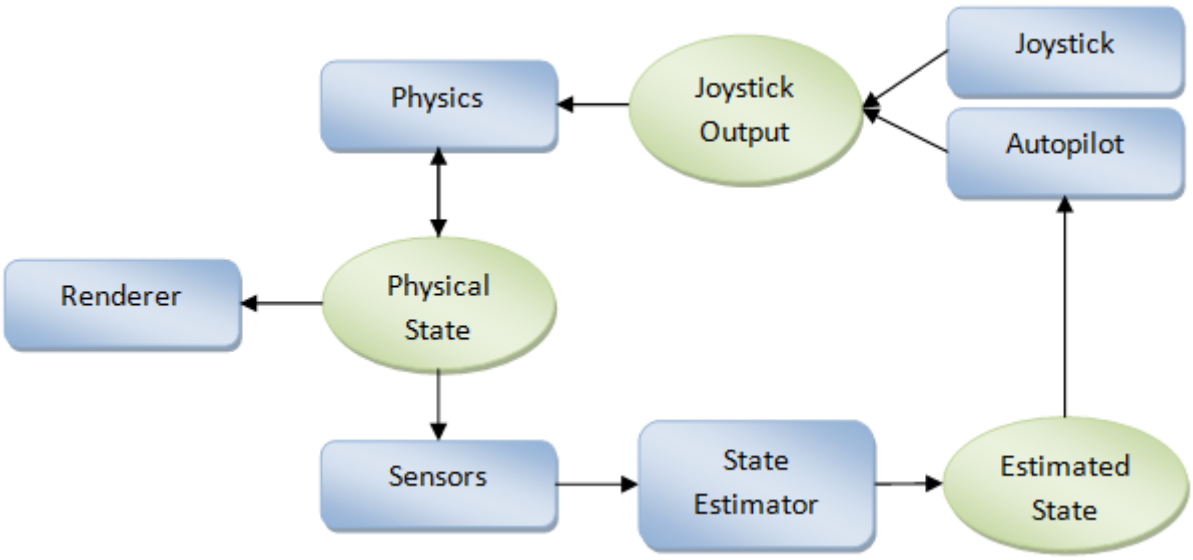


Figure 5.3.: Component data flow.

5.3. Visualization

To give a believable illusion of flying in the virtual world we implemented some visualization methods on our own and reused some open source components. This section covers the most significant parts of our implementation that were involved in the visualization.

5.3.1. Skydome and Sunlight

The skydome and the sunlight effect were reused from the open source component Atmospheric Scattering V2 [23] to give the illusion of a sky surrounding the terrain. In contrast to typical skyboxes and skydomes there was no geometry involved in rendering the sky. Instead a HLSL shader was used to render the sky directly, which was very quick and produced believable results. The effect was largely based on blending color gradients between day, sunset and night as well as rendering a sun according to the time of day. The skydome component allowed us to easily manipulate the time of day by changing the parameters of the shader accordingly as shown in figures 5.4 and 5.5.

The same shader could also with some modifications be applied to scene objects to light them according to the color, intensity and direction of the sunlight. For outdoor scenes this is an important illusion since the sun is the main light source, so if the objects are not lit in the same manner this will look very unnatural. Finally, the shader used fog in order to blend the scene objects nicely into the horizon at long distances and at sunset this gave an illusion of atmospheric scattering, as shown in figure 5.4.

Since we were rendering outdoor scenes this sunlight effect was reused for all our scene objects to give a more natural appearance. The major modifications we needed to apply was concerned with objects that were already using shaders, such as the terrain and the trees. Those changes are described in their respective sections. For all other textured static mesh objects we created a SimpleModel class that applied the sunlight automatically.

5.3.2. Helicopter

Although most simple objects in the world were rendered by the SimpleModel class there were some objects that required more work. For instance the helicopter needed to let

Figure 5.4.: Screenshot of the world at sunset.

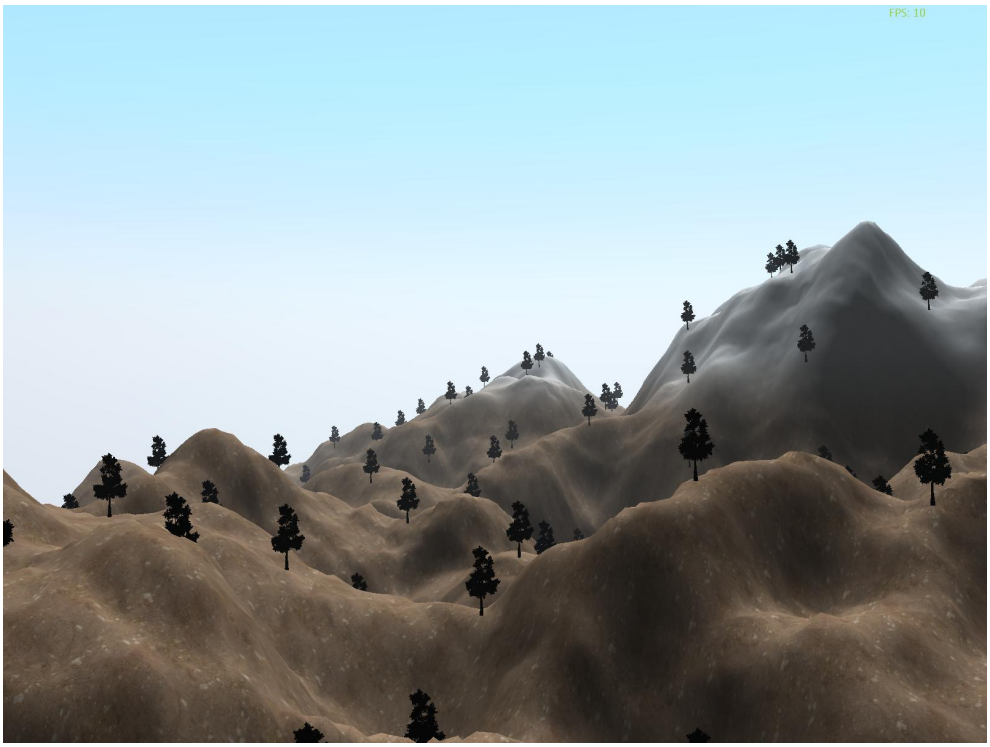


Figure 5.5.: Screenshot of the world in daylight.

Algorithm 5.1 Algorithm for applying local rotor rotation to the rotor mesh and rendering motion blur instances of the rotors.

i = 0 to motionBlurCount
$rotorBaseAngle = revolutionsPerSecond \cdot 2\pi \cdot dt$
$localRotorAngle = rotorBaseAngle + i \cdot motionBlurSpacing$
$localRotorRotation = CreateQuaternionFromAxisAngle(Vector3.Up, localRotorAngle)$
$worldRotorRotation = worldHelicopterRotation \times localRotorRotation$

the rotor to spin and we also needed a "ghost" version of the helicopter that illustrated the position and rotation of the estimated helicopter state to help debug the autopilot.

Main Rotors To give the illusion of the rotors rotating we split the mesh in two parts. In 3D Studio the rotor polygons were separated from the body to form two separate meshes. The two meshes were named accordingly so that in XNA we could distinguish the two meshes by name.

XNA has classes for working with quaternions and the rotation of the helicopter is represented by a quaternion. Applying rotation to the rotors was then a matter of quaternion multiplication of helicopter rotation and local rotor rotation. Algorithm 5.1 illustrates how the rotors are rotated as a function of time and how motion blur instances are positioned to give the illusion that the rotors are moving very fast. This effect is illustrated by 5.6 where the opacity is gradually reduced for each successive motion blur instance.

Helicopter Ghost To aid in the debugging we needed to visualize the estimated state of the helicopter. By rendering the helicopter a second time with some specific render states we achieved a ghost-like effect. As seen in figure 5.7, the helicopter is transparent and white and this way we could easily see the errors in estimation alongside the real helicopter and clearly distinguish the two. These are the render states we set prior to rendering the helicopter with its normal render states.

```
GraphicsDevice.RenderState.AlphaBlendEnable = true;
GraphicsDevice.RenderState.SourceBlend = Blend.One;
GraphicsDevice.RenderState.DestinationBlend = Blend.One;
```

Figure 5.6.: An illusion of motion blur is achieved by rendering gradually more transparent rotor instances.

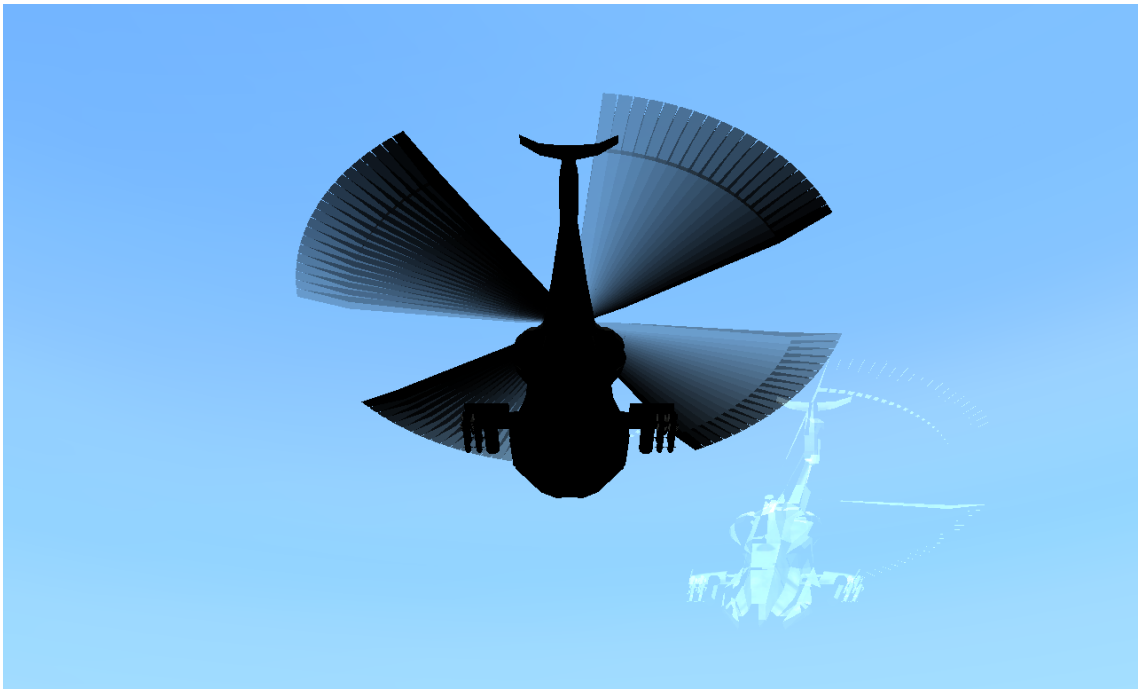


Figure 5.7.: State estimation error visualized by a ghost helicopter.

5.3.3. Terrain

For terrain we used an open source heightmap generator from an article on terrain generation [24] and wrote a custom shader to render it. The shader blends textures of snow, grass, stones and dirt as a function of terrain height and adds the sun lighting as described in 5.3.1. The end result is shown in figure 5.4. Although the visualization is far from realistic it is easy on the eye and provides a believable representation of terrain. Even so, we would like to see more realistic details in the diffuse and heightmap textures as well as add bump mapping or parallax mapping to up the realism and the sense of scale.

5.3.4. Trees

Trees were rendered by the open source XNA component LTree 2.0a [25]. The component generated trees by the Lindenmayer system and utilized the XNA content pipeline to easily generate different kinds of trees from XML definition files. The component had a custom shader to render the trunk and branches as geometry and the leaves as billboards. The shader also supported simple skeleton animation to give the illusion of the trees swaying in the wind.

The shader had to be modified to support our sunlight effect. The geometry shader was straight forward to modify, but the billboard shader was harder to get right. The problem was that the billboards are flat and always rotate to face the camera. In order to give the illusion of geometric leaves being lit by the sun we made a simplification where we constructed a normal vector directed towards the camera for each leaf billboard. This way when observing the tree from its shadow side it would look unlit and when observing it from towards the sunlit side it would gradually look more lit as shown in figure 5.8. Naturally, this uniform lighting of all the leaves did not look very realistic. However, as figure 5.4 shows the sunlight effect did blend the trees naturally in with the rest of the sunlit terrain at distances.

5.3.5. Stereo Rendering

We decided to implement stereo rendering to give the user a heightened sense of depth in the virtual world. This was not only useful to better determine distances and the scale

Figure 5.8.: A sunlit tree seen from different angles.

of things, but it was also very fun to use. In the lab we had a head mounted display (HMD) with head tracking and dual high-resolution monitors that were used to try out the stereo rendering effect.

The implementation was fairly straight-forward as we already had camera classes providing the camera position, the look-at position and the up vector. Extending the camera implementations was simply a matter of rendering the world twice to different monitors and offsetting the camera position slightly according to the eye separation distance. We also had to take the up vector into account for rolling cameras such as the cockpit camera that utilized head tracking. The camera offsets would then be set along an axis perpendicular to the look-at vector and the up vector.

Eye Separation Distance For a realistic 3D effect the camera offsets should match the true eye separation distance. This distance can vary a few centimeters between persons, but an average for men is about 6.5 cm. We experimented with different values to see how they affected the 3D effect. In our implementation we used 10 cm, because it increased the 3D experience and was still comfortable to use. Beyond 10 cm we started noticing how it was hard to focus on different parts of the picture and how we could sustain only short periods of time before tiring and starting to get headaches. We experimented with values as high as 50 cm and could see parts of the picture clearly, but due to the focus depth nearer and more distant objects were hard to focus on.

Focus Point In our implementation we used a look-at point fixed at a 10 meter distance from the camera. This meant that objects near that point would be clear, while objects away from that point would be harder to focus on. This issue became more evident when increasing the eye separation distance. We realized that there were two reasons to why stereo rendering was hard to get right.

1. The cameras must focus on what the user is looking at.
2. Everything else should to some degree be out of focus.

The first issue could be solved by a dynamic look-at point so that both eye cameras point towards what user is looking at. A simple solution would be to cast a ray along the non-stereo look-at vector and use the intersection point as a focus point for the stereo cameras. This would focus the cameras towards the nearest object roughly in the centre of the image. The problem was that the method does not take into account what part of the picture the user is looking at. Even with head-tracking and HMD some of the stereo illusion was lost since objects are harder to focus on if looking at areas off the picture centre.

We could prevent this artifact to some extent by applying the depth-of-field effect commonly used in modern games. If a dynamic look-at point was used then we could use a fast post-processing depth of field shader that blurs out the parts of the image that is out of focus. This way the out of focus effect on objects off the image center would not be as evident and it guides the focus of the user towards the center of the image. Unfortunately we did not have enough time to implement this.

Cross-Converged View Since we already had the the code to render in stereo it was a short step to implement what is known as cross-converged viewing. The same effect is often used in 3D-image books and the advantage is that no special equipment or multiple monitors are required. The method is to render two cameras onto the same monitor by dividing the monitor in a left and right half as illustrated in figure 5.10. To achieve the stereo effect the user will cross his or her eyes by focusing on a point between the person and the monitor. With some exercise the two images will slide to form a third picture in the center, which comes into focus with the stereo effect. It helps to use the hands to cover the peripheral vision so that only the center image is visible. Unfortunately this method is very straining on the eyes and the horizontal field of view is halved.

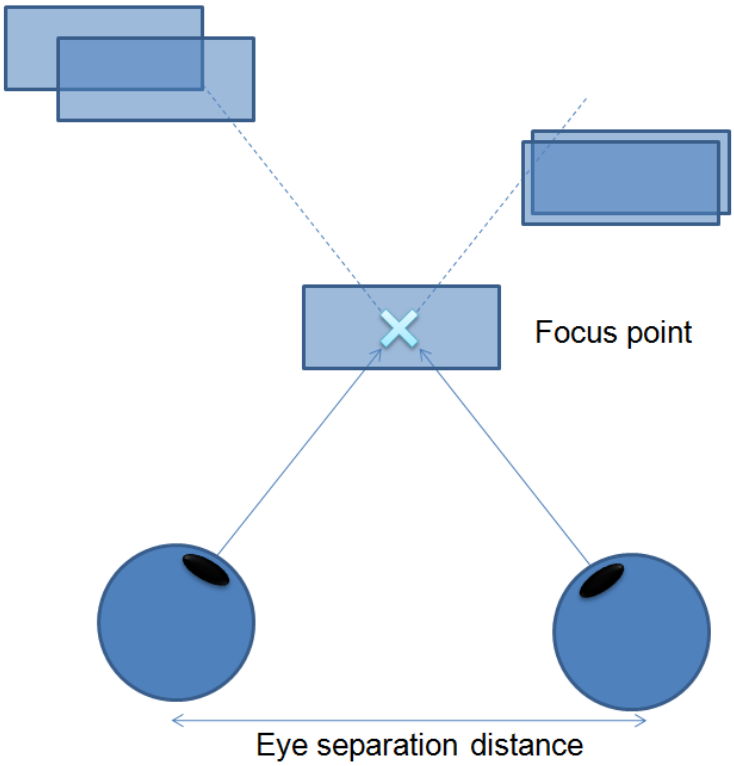


Figure 5.9.: Objects near the focus point become clear, while objects away from the point are harder to focus on.

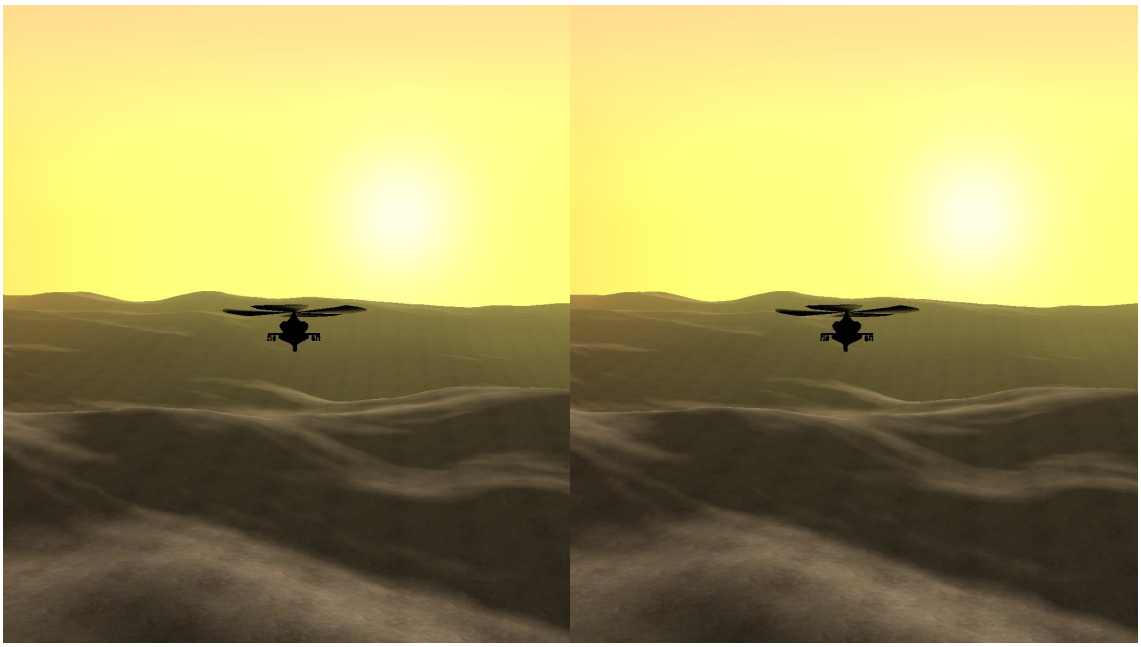


Figure 5.10.: Stereoscopic effect on single monitor by cross-converged viewing.

5.3.6. Cockpit Camera

Since we already had an HMD that supported head tracking we decided to create a cockpit camera. This allowed the user to fly the helicopter while sitting in a virtual 3D cockpit and to look around by turning his or her head. This was both intuitive and fun and truly immersed the user in the virtual world unlike any conventional PC game today. The implementation problem was threefold. First we needed to obtain the head rotation, then apply the rotation to our camera and finally render the virtual cockpit.

Reading head rotation by VRPN VRPN was configured according to the lab wikipage [26] for settings such as sensor coordinate system and device names. To obtain the head rotation we used a .NET wrapper for the VRPN client (VrpnNet-1.1.1) that greatly simplified the usage in the C# programming language. With the correct setup reading sensory data was a trivial task.

Transforming Rotation to World Coordinates The second issue was that the sensor coordinate system did not match up with XNA. The IS-900 system in the laboratory is set up with a right-handed system so that X points north, Y points east (towards the windows) and Z points down. From a sensor point of view north, east and down is +X, +Y and +Z respectively.

XNA is also right-handed, but here we defined north, east and down as -Z, +X and -Y. Following the "A simpler approach for orientation" on the Flock of Birds ERT wikipage [26] we have the following equation to transform the measured HMD orientation into XNA world frame.

$$Rot_{XNA} = A_{xref} \times Rot_{sensor} \times A_{xref}^{T} \tag{5.1}$$

A_{xref}^{T} is the matrix found by comparing the sensor's reference frame to the XNA frame. It can be seen that the sensor +X axis (north) maps to the XNA -Z axis, sensor +Y (east) maps to XNA +X and sensor +Z (down) maps to XNA -Y. From this we define the following matrix:

$$A_{xref}^{T} = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \begin{matrix} (north) \\ (east) \\ (down) \end{matrix}$$

Here A_{xref} is simply the matrix transpose of A_{xref}^{T}.

Figure 5.11.: Virtual cockpit

Rendering a Virtual Cockpit To complete the illusion of sitting inside the helicopter we needed a 3D cockpit. This was achieved by rendering a textured static mesh aligned with the helicopter and then positioning the camera at a fixed position relative to the mesh at the virtual pilot's head. The result is shown in figure 5.11.

5.3.7. Other Cameras

Although the cockpit camera was the most comprehensive we also implemented three simpler camera types that were useful for both manual and autonomous flights.

Free Camera One multi-purpose camera that was implemented is the free camera. The mouse is used to look around, while the keyboard moves the camera in the direction one is looking. This camera allows the user to move and look around freely as if flying a fictional spaceship. Holding down the SHIFT key increases the speed at which the camera is moving to cross large distances more quickly.

Fixed Camera When testing the autopilot behavior it was often useful to observe the flight pattern from a fixed point of view. The fixed camera has a fixed position and will always look at a particular scene object, such as the helicopter. This camera was often used to overview precision navigation scenarios to get a better sense of scale and motion.

Chase Camera A chase camera was convenient for observing flights over large distances. The code was largely reused from an XNA sample at [27] and incorporates a spring to smoothly follow the moving object. The spring helps visualize changes in altitude or horizontal motion, since the camera will lag behind for a short while before catching up with the helicopter.

5.4. Simulation of Physics

To challenge the autopilot with semi-realistic navigation scenarios we needed a physics simulation. This section covers how we implemented the flight behavior and collision handling to allow the helicopter to land on the ground.

5.4.1. Flight Dynamics Model

Following the recommendation from our previous work [9] we decided to go with the simplified flight dynamics model (FDM) #4, as described in section 2. This model uses parametric equations for drag and lift with empirically chosen coefficients to get a reasonable maximum velocity and control response.

Angular Motion One major simplification was to let change in orientation be a direct function of joystick output. A more realistic simulation would model angular motion by torque from a number of sources, such as airflows towards the fuselage, main rotor inertia and angular velocity, cyclic controls and the varying lifting force of each individual rotor blade over a revolution.

However, from our own experience in flying model helicopters our method was a good approximation. There a number of reasons for this. First, their small scale creates very little moment of inertia when rotated so the motion is quickly damped. Second, the main

rotor will stabilize the orientation in the same manner as a spinning wheel will resist reaction to outside forces. Third, the tail rotor is designed to counter any unwanted change in heading angle, such as by winds or by the torque of the main rotor. Added to our own experience in flying model helicopters we assumed that the angular momentum was insignificant enough to be ignored in our simplified physics simulation.

Linear Motion Now that angular motion is a function of joystick output we are left with forces of linear motion. Although the helicopter is an advanced mechanical structure that operate by complex aerodynamic phenomena we can simplify the behavior using a model of lift and drag forces. Figure 5.12 illustrates the forces at work in our flight dynamics model. In [9] the equations of lift and drag were described as:

$$F_L = \frac{1}{2}\rho u^2 A C_L \tag{5.2}$$

$$F_D = \frac{1}{2}\rho u^2 A C_D \tag{5.3}$$

ρ	mass density of the fluid
u	relative airflow velocity
A	reference area (typically the square of the mean chord length for a wing)
C_D	drag coefficient
C_L	lift coefficient

Drag Force Using these equations we had to determine the coefficients and constants. Mass density of air is defined in [28] as $\rho = 1.204$ at sea level and 20 $°C$. The reference area A was chosen as a constant rectangle of 0.1×0.2 m, which approximates the cross-section of the model helicopter in forward flight.

We know that the body shape affects the drag and that the drag coefficient should reflect this. [29] proposes that a cube has $C_D \approx 1.05$ while the C_D of a streamlined body approaches zero. We chose $C_D = 1$ since it produced a maximum velocity of around 80 km/h; a realistic speed for model helicopters and in line with the FDM #4 definition.

Relative airflow velocity u is simply the length of a combination of the helicopter velocity vector and the wind vector. An improved model would take the air flow angle towards the fuselage into account and determine an appropriate reference area and drag coefficient.

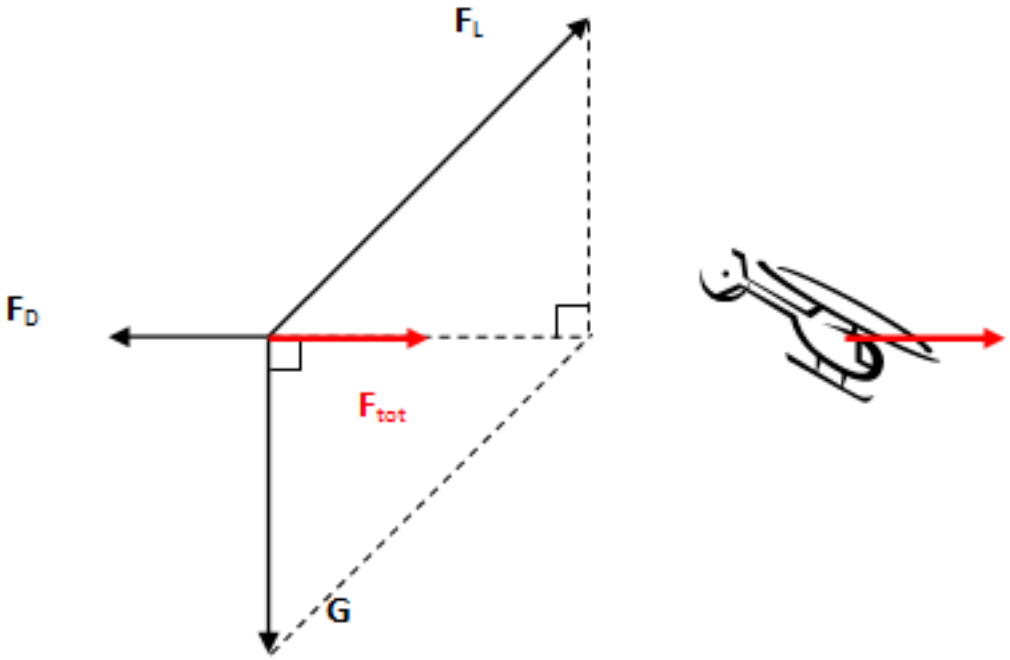


Figure 5.12.: Forces at work in linear motion of helicopter according to FDM #4.

Lifting Force Based on empirical experiments we found that we wanted the maximum lift force to be approximately 70% greater than the force of gravity. Again we ignore the inertia of the model helicopter main rotor due to its small scale and define relative airflow velocity u of the main rotor blades as a function of joystick throttle h. We then reduce equation 5.2 to $F_L = 1.7Gh$.

h joystick throttle $\in [0, 1]$

G force of gravity

5.4.2. Collision Handling

In order to prevent the helicopter from flying through the terrain and to distinguish between landing and crashing we used the Jitter physics engine. Integrating Jitter with our heightmap terrain and helicopter polygonmesh introduced a few problems.

First, Jitter did not support heightmaps so we had to construct an indexed vertexbuffer of polygons from the heightmap. Second we already had a physics component that

dealt with the flight dynamics so we had to re-route parts of our physics code to Jitter. The Jitter integration was solved by first calculating all the linear forces and angular velocities in our flight dynamics model, then passing those to Jitter for each timestep. This way we could add collision handling without any constraints on the flight dynamics. Third, we just broke the *True Observer Condition* in our state estimator as described in section 5.6.3. This could only be overcome by disabling collision handling when verifying the correctness of the state estimation implementation.

With those problems solved the collision handling was simply a matter of defining a proper bounding volume that represented the mass and volume of the helicopter. After some difficulties trying to use more accurate compound objects for the skids, fuselage and rotors we ended up using a simple rectangular prism that matched the dimensions of the helicopter. This way to land the helicopter without tipping over was sufficiently difficult.

5.5. Simulation of Sensors

Sensors are an important part of autopilots as they are the only means of observing the environment. To simulate the behavior of an autopilot as realistically as possible the sensors need to be simulated in a realistic manner as well. This section covers the sensors in our implementation and how they match up with the specifications of commercially available sensors. The choice of sensors was based on quality, price and availability for future applications. All sensor datasheets are included in appendix C.

5.5.1. Sampling Rate

There are several issues with simulating sensors at different sampling rates as proposed by requirement R5 in section 5.2.1. Our real-time simulation has a game loop that runs at approximately 60 Hz. Each iteration we simulate physics, update the world state and render. Having a different sampling rate requires the game loop to run a lot faster since the world state must update before taking new sensor measurements. Figure 5.13 shows how having two different sampling rates can require the world state to update twice as often as the ordinary real-time game loop, even though each individual sensor has a lower sampling rate than the original game loop.

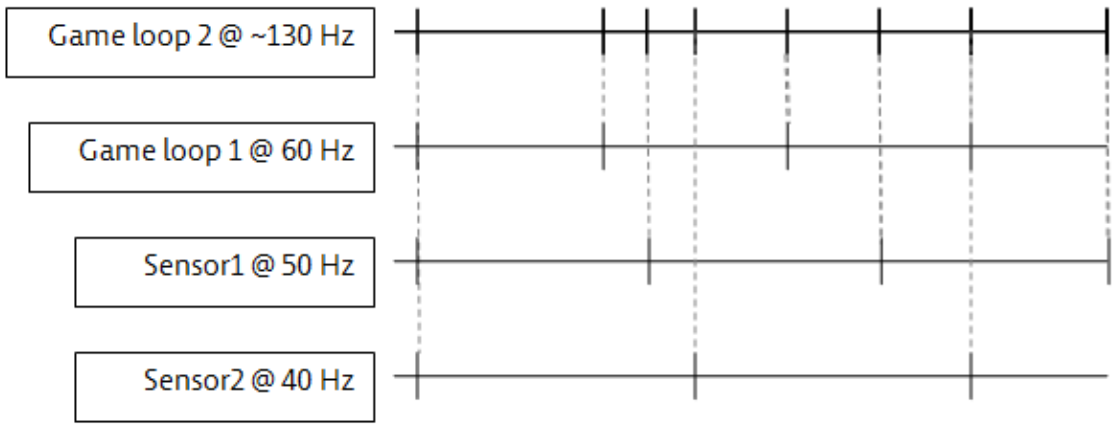


Figure 5.13.: Different sampling rates increases the game loop frequency.

Before adding sensors the render pipeline was typically the bottleneck of the game loop, but with different sampling rates the physics simulation now has to run multiple times for each render pass and the frame rate easily becomes CPU bound. We also noted that the problem is not readily parallelizable, since each timestep requires the results from the previous timestep. The problem escalates when modelling high-rate sensors such as accelerometers and gyroscopes that can run at more than 1000 Hz in UAV applications.

To achieve real-time simulation we decided to simplify and sample each sensor only once per game loop iteration. The advantage was that we could isolate sampling rate when verifying the correctness of the state estimator, as described in section 5.6.3, since the state estimator was now synchronized with the simulator. Different sampling rates and undersampling was achieved by dropping samples, but high-rate sensors were not feasible in real-time so we consider requirement R5 to be only partially met.

5.5.2. GPS

A Global Positioning System (GPS) is a natural part of any navigation system. The main advantage is that we get an absolute measurement of the position on the planet so any errors in the measurements will not accumulate over time. The key issues with GPS systems are low sampling rates and that measurements are often off by several meters. This means that GPS alone will not suffice for autonomous navigation of a flying vehicle. In our implementation we simulate the FV-M8 GPS from SANAV.

Key Specifications of FV-M8

Rate: $1 \sim 5\,Hz$

Horizontal position accuracy: 3.3 m CEP (approx. $\mathcal{N}(0,\, 4.9\,m)$ radius)

Vertical position accuracy: N/A (assumed equal to horizontal)

Velocity accuracy: 0.1 Knot RMS (approx. $\mathcal{N}(0,\, 0.05\,m/s)$ radius)

Typically GPS sensors only deliver position and velocity measurements once every second and that is the sampling rate we used. The accuracy was specified for aidless tracking and can be further increased by DGPS solutions. The vertical positional accuracy was not described in the specifications, so we assume it is equal to the horizontal accuracy here. The tight velocity accuracy is also useful to correct the velocity estimate of the IMU, but we suspect that the specifications are not as accurate in practice.

Implementation A GPS sensor is trivial to implement. The true world position is already known from simulation so we only need to add navigation frame noise to the measurements. An accuracy of 3.3 m CEP (Circular Error Probable) means that 50% of all measurements fall within this radius. We assume here that the noise follows a Gaussian distribution with zero mean and from tables in [30, pp. 146-154] we convert CEP to a standard deviation of $\sigma_{pos} \approx 4.9\,m$. Velocity accuracy is already expressed in standard deviation so we simply convert to metric and find $\sigma_{vel} \approx 0.05\,m/s$.

To implement the random position error we generated a random normalized 3D vector and multiplied it with a distance $d \sim \mathcal{N}(0,\, \sigma_{pos})$. The GPS position is then obtained by adding the random position error vector to the true position vector. The GPS is then modelled to let 68% of the position measurements be within 4.9 m of the truth. The random velocity error vector was calculated in a similar manner.

5.5.3. IMU

Inertial Measurement Unit (IMU) is a widely used concept in inertial navigation systems (INS). The typical configuration consists of accelerometers and gyroscopes arranged to measure linear acceleration and angular velocity of a body. Given a known starting state we can estimate the position and orientation by dead-reckoning, however in practice this estimate will quickly drift away from truth due to noise, inaccuracy, precision and discretization. This section covers the implementation of the two sensors required for an IMU.

5.5.3.1. Accelerometer

An accelerometer measures linear acceleration along one or more axes and we use this to calculate the velocity and position of the helicopter. Accelerometers can typically run at very high rates and accurately captures changes in linear motion for short intervals of time. In our implementation we simulate the ADXL330 accelerometer from Analog Devices.

Key Specifications of ADXL330

Rate: Max. 1600 Hz

Range: $\pm 3\,g$

Noise density forward/right: 280 $\mu g/\sqrt{Hz}RMS$

Noise density up: 350 $\mu g/\sqrt{Hz}RMS$

The noise is specified to resemble Gaussian behavior at all frequencies. In order to translate density to an actual noise level we must first determine what bandwidth we require the sensor to run at. The bandwidth decides what frequencies to encompass and according to [31] a good choice for UAV navigation is a bandwidth of 0-400 Hz, for a number of mechanical and electrical reasons. This gives noise RMS levels of:

$$\begin{aligned} \sigma_{right} = \sigma_{fwd} &= 280 \times 10^{-6}\sqrt{400} = 7.1E-3\,g \\ \sigma_{up} &= 350 \times 10^{-6}\sqrt{400} = 8.9E-3g \end{aligned}$$

Implementation The sensor measures acceleration in the body reference frame, while both the physics simulation and the state estimator operates in the navigation frame. To implement an accelerometer we need to transform the simulated world acceleration vector to body frame. Then the state estimator needs to transform the accelerometer vector back to navigation frame, which should ideally equal the simulated vector. Unfortunately there is a precision loss in converting between reference frames so that the state estimator would be off even if no noise or inaccuracy were added to the sensors.

In section 5.6 we discuss how we needed to verify the correctness of the state estimator by zero deviation when no noise was added to the sensors. For this case we cheated and passed the world acceleration vector directly to the state estimator. This way only the noise vector would need to be transformed and so the loss in precision would only be a negligible addition to the noise. Obviously this is not possible using real sensors and is only useful in a simulation scenario to verify the validity of the state estimator. Details of the transformation precision loss is found in appendix D.2.

5.5.3.2. Gyroscope

A gyroscope measures the angular velocity of one or more axes and we use it to estimate the orientation of the helicopter. Gyroscopes run at high rates and accurately captures changes to the orientation over short periods of time, but will drift away from truth over time. In our implementation we simulate the LYPR540AH 3-axial gyroscope from ST.

Key Specifications of LYPR540AH

Rate: Max. 140 Hz

Range: $\pm 400\,^\circ/s$

Noise density: $0.02\,^\circ/\sqrt{Hz}RMS$

The range of this sensor should suffice for our autopilot. Helicopters are an unstable platform so we want to avoid sudden changes to the orientation to ensure we do not lose control of the vehicle. Once more noise is defined as a density function of bandwidth and for this sensor we use a bandwidth of 0-140 Hz. This gives a noise RMS level for each axis of:

$$\sigma_{axis} = 0.02\sqrt{140} \approx 0.24\,^\circ/s$$

Implementation The gyroscope was implemented by calculating the angular velocity each timestep. First we used the straight forward solution of calculating the angular displacement as $d_i = \triangle\theta_i$ and angular velocity as $\omega_i = \frac{d_i}{\triangle t}$ for Euler angles $\theta_1, \theta_2, \theta_3$.

Later we discovered that this delayed the state estimator by one iteration since the angular velocity would reflect the change from the previous to the current point of time instead of the current angular velocity. In section 5.4 we describe how the orientation of the helicopter is simply a function of the joystick output in the physics simulation. This way we can simply formulate the measured angular velocity as a function of joystick output.

5.5.4. Range Finder

The range finder was added to the setup to enable low flight scenarios such as flying along terrain. GPS and inertial navigation systems suffer from inaccuracy of up to several meters and easily confuses the autopilot logic due to large sudden jumps in estimated position. The range finder typically uses sonar, LASER or RADAR technologies to measure the distance to an object with great accuracy. In our implementation we simulate the LV-MaxSonar-EZ0 sonar range finder from MaxBotix.

A typical mounting for the range finder is shown in figure 5.14. Here the sensor points in the down direction of the vehicle and returns the range to nearest object within the range and width of its *beam*. We use this beam primarily to measure the height above the ground and its high accuracy surpasses the INS/GPS estimates for this task. By telling the autopilot to hold a fixed height above the ground we can now navigate just a few metres above the terrain with much less risk of crashing.

Key Specifications of LV-MaxSonar-EZ0

Rate: 20 Hz

Range: 0 m - 6.45 m (extended to 10 m)

Resolution: 2.54 cm

Noise: N/A

One obvious limitation here is the range of just about 6 meters. The autopilot can only follow the terrain as long as it can measure the distance to the ground, so naturally we are restricted to flying no higher than roughly 5 meters above the terrain. This leaves little room for error and would never be acceptable for real outdoor navigation. However, for our simulation it still serves a purpose so we decided to slightly extend the

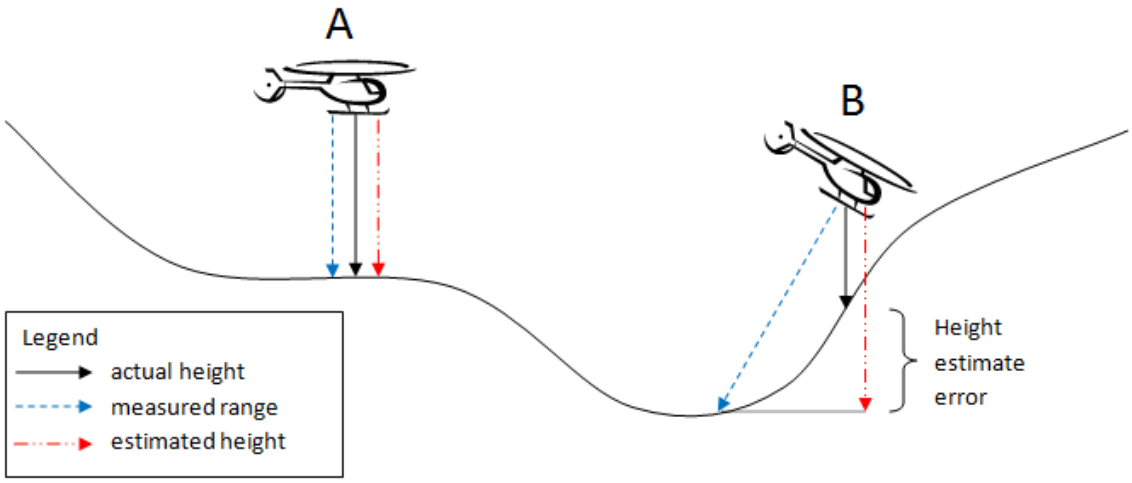


Figure 5.14.: Measuring flat ground height.

range to 10 m to allow for more interesting testing scenarios. The resolution, noise levels and measurement rates was expected to prove sufficient for our application and the final results are found in section 6.5. The sensor datasheet does not mention any noise so we simply considered the resolution of 2.54 cm as the only uncertainty in our application.

Flat Ground Height There is one big issue with using a range finder to measure height above the ground. Figure 5.14 shows how we measure what we call *Flat Ground Height* (FGH). When estimating the height above the ground we assume the ground is flat and simply use the vertical component of the measured range to find FGH. Helicopter A is pointing its sensor straight down and so FGH equals height above the ground at that point. Helicopter B, however, is pitching its nose down in order to accelerate forwards and consequently the sensor is now pointing at an angle. Intuitively we can see that FGH no longer equals height above the ground and we get an estimate error depending on the curvature of the terrain and the orientation of the helicopter. Later in the experiments section we show that it was indeed possible to navigate along the terrain using datasheet sensors and the FGH method.

One a side note, it is possible to mount the sensor to always point straight down to overcome the problem of measuring height in the first place. This could be arranged by a simple PID-controller and two servos, since it has an estimate of the current orientation. However, for small remote controlled helicopters this setup may not be feasible due to limitations in weight, batteries and space so we chose the former method.

Implementation Another issue was how to properly implement the sensor. We needed to simulate the fact that the sensor is pointing at an angle towards the terrain, so we decided to use a form of raycasting for this. The terrain was defined as a heightmap and since most raycast implementations work with polygons and bounding boxes we implemented our own raycaster using the binary search algorithm.

The idea is very simple and the recursive function is started by providing the helicopter position, the sensor beam world direction and the max range of the beam. This range is then cut in two halves and each subrange is then recursively checked for intersection with the heightmap. The problem is that we don't have sorted heights along the ray so we must determine if an intersection exists by other means. One simplification is to compare the height above the ground (HAG) for the start and end points of that range. GetAltitude(point) looks up the positions in the heightmap map by bi-linear interpolation and returns the HAG for a world point. If both are on the same side of the terrain (above or below) we assume that line segment does not intersect with the terrain. If not, the line segment does intersect and we recursively search for it. The function returns the midpoint of a line segment as the intersection point once the length of the line segment is less than the specified resolution of 2.54 cm.

The method worked well in our scenario and should be faster than linear searching raycasting algorithms. However, our method does suffer from corner cases that could break the functionality. Figure 5.15 shows how the measured range can become undefined if the terrain curvature is hilly and the helicopter is flying low with a significant tilt. Here our intersection check fails since both the start and end points are above the terrain. The tests, however, did not indicate any problems with this case since the autopilot is configured to not exceed 10° of pitch or roll and since our terrains were not as extreme as depicted in the figure.

5.6. State Estimation

The state estimator provides the only input to our autopilot about where it currently is and where it is headed. This section explains how we implemented a Kalman filter to estimate position, velocity and orientation of the helicopter from noisy and incomplete sensor measurements and what issues we encountered.

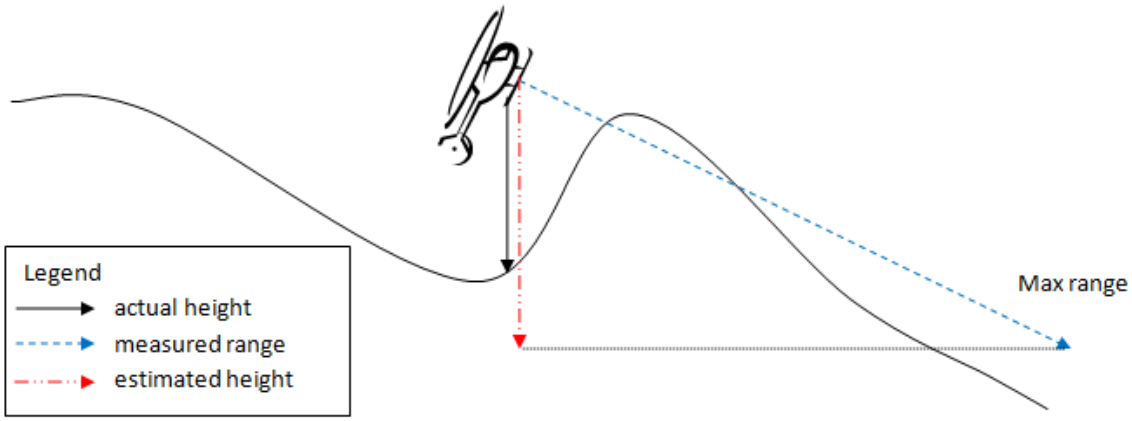


Figure 5.15.: Corner case of the binary search raycasting algorithm.

Algorithm 5.2 Pseudo-code for raycasting implemented by binary search.

```
float FindIntersectionDistance
(float minRange, float maxRange, float maxError, Vector3 position, Vector3 worldDirection)
{
  if (distance(minRange, maxRange) <= maxError)
    return (minRange + maxRange) / 2;

  float halfRange = minRange + (maxRange - minRange) / 2;
  Vector3 rayStartPosition = position + minRange * worldDirection;
  Vector3 rayHalfPosition = position + halfRange * worldDirection;
  Vector3 rayEndPosition = position + maxRange * worldDirection;
  float startGroundAltitude = GetAltitude(rayStartPosition);
  float halfPointGroundAltitude = GetAltitude(rayHalfPosition);
  float endGroundAltitude = GetAltitude(rayEndPosition);

  // Height Above Ground
  float startHAG = rayStartPosition.Y - startGroundAltitude;
  float halfPointHAG = rayHalfPosition.Y - halfPointGroundAltitude;
  float endHAG = rayEndPosition.Y - endGroundAltitude;

  if (startHAG > 0 && halfPointHAG <= 0)
  return BinarySearchTerrainRayIntersection(minRange, halfRange, maxError, position,  worldDirection);

  if (halfPointHAG > 0 && endHAG <= 0)
  return BinarySearchTerrainRayIntersection(halfRange, maxRange, maxError, position, worldDirection);

 return float.NaN;
}
```

5.6.1. GPS/INS Kalman Filter

GPS/INS Kalman filter is a widely used method for autonomous outdoor navigation. An inertial navigation system (INS) provides position and orientation estimates by dead reckoning from the IMU measurements, while the GPS provides absolute position and velocity estimates from satellite signal processing.

We mentioned in section 4.4.1 how absolute and relative measurements differ and that is why the GPS and INS complement each other. The INS a high sampling rate and excel at representing motion over short periods of time and capturing sudden changes in motion. However, due to the integration of angular velocity and the double integration of linear acceleration these estimates quickly diverge from the truth. This is where the GPS becomes useful. The GPS has a very low sampling rate (typically 1 Hz) and an inaccuracy of up to several meters, however the error is near-Gaussian and centered at the truth so we can use it to prevent the INS estimate from diverging too much.

Omitting the Orientation Estimate Ordinarily the INS/GPS filter should estimate position, velocity and orientation from noisy and incomplete sensor measurements, but expressing equations for orientation in the process model and the observation model and constructing the covariance matrices $\mathbf{Q}$ and $\mathbf{R}$ was not easily achieved. Although we had several sources ([4, 32, 33]) that proposed solutions, we did not find enough time to get the implementation right, so in this section we have left out the angular velocity measurements from the control input vector $\mathbf{u}$ and the orientation quaternion $\mathbf{q}$ from the state model for simplicity. Some descriptions and figures will still include these two to illustrate the intended function of the filter. In our flight experiments we compensated for the lack of uncertainty by adding an error bias to the true orientation.

Overview Figure 5.16 illustrates how the Kalman filter relies on the INS in-between GPS updates and the circles denote how the INS estimate uncertainty increases over time.

The two key Kalman filter equations we need to consider here is the process model 5.4 and the observation model 5.5. Refer to section 4.4 for the background theory.

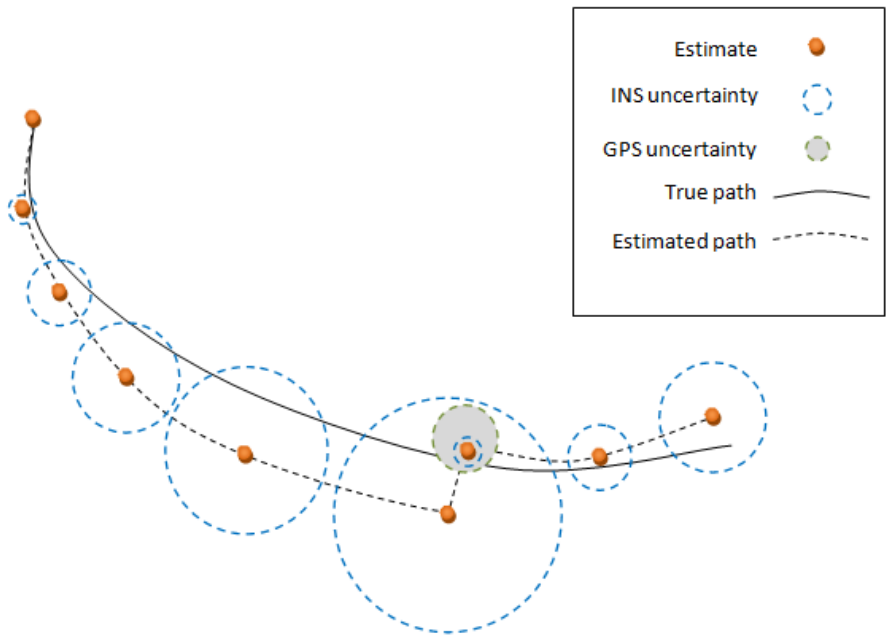


Figure 5.16.: The accuracy of the INS estimate degrades over time and its error is corrected by the more reliable GPS measurement.

$$\mathbf{x}_k = \mathbf{A}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k \tag{5.4}$$

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{5.5}$$

Figure 5.17 illustrates how we have configured the Kalman filter to estimate position, velocity and orientation of the helicopter. Each timestep we fuse sensor information from the GPS and the IMU. The filter then integrates the linear acceleration and angular velocities to produce an INS state estimate, which is then compared to the GPS position and velocity observations by their magnitudes of uncertainty.

Symbols

$\mathbf{p}$	position in navigation frame
$\dot{\mathbf{p}}$	velocity in navigation frame
$\ddot{\mathbf{p}}$	linear acceleration in navigation frame
t	timestep duration in fraction of seconds

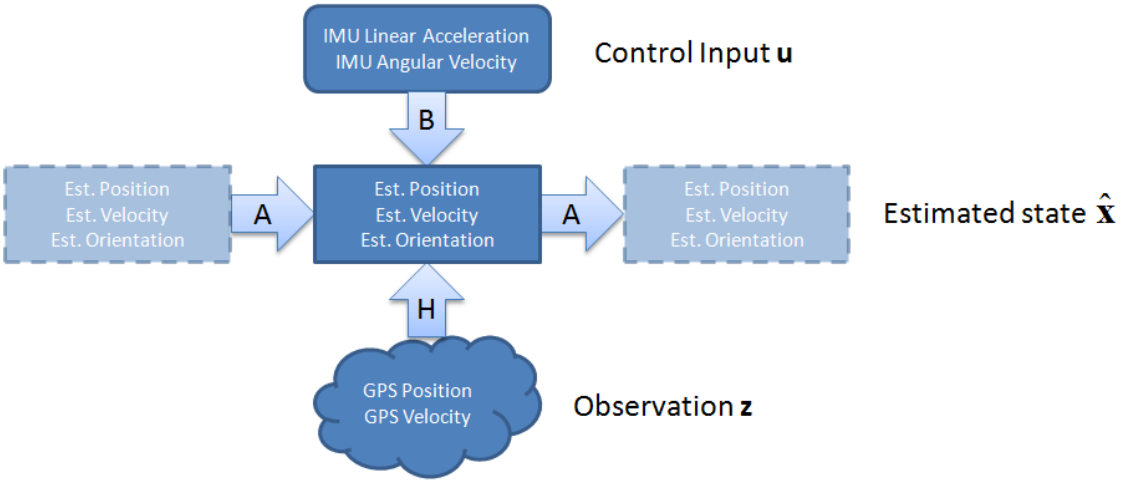


Figure 5.17.: Overview of the GPS/INS Kalman filter configuration.

State Model Since we have omitted the orientation filtering we are left with estimating the position and velocity of the helicopter. In our implementation we used the north-east-down (NED) navigation frame and chose origo at $\mathbf{p}_{ned} = \begin{pmatrix} 0, & 0, & 0 \end{pmatrix}$. We define our state as:

$$\mathbf{x} = \begin{bmatrix} p_n & p_e & p_d & \dot{p}_n & \dot{p}_e & \dot{p}_d \end{bmatrix}^T$$

State Transition Model Part of the process model is defined by the state transition model $\mathbf{A}$, which describes how we believe the true state $\mathbf{x}$ will propagate by time without any control input. From the equation of linear motion $\mathbf{p} = \mathbf{p}_0 + \mathbf{v}t$ we formulate our transition model $\mathbf{A}$ so that

$$\mathbf{Ax} = \begin{bmatrix} 1 & 0 & 0 & t & 0 & 0 \\ 0 & 1 & 0 & 0 & t & 0 \\ 0 & 0 & 1 & 0 & 0 & t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_n \\ p_e \\ p_d \\ \dot{p}_n \\ \dot{p}_e \\ \dot{p}_d \end{bmatrix}$$

describes the change in position due to velocity over time.

Control Input Model The control input $\mathbf{u}$ is typically a vector of variables controlled by the autopilot. How the control input is expected to change the state is defined by the control input model $\mathbf{B}$. According to [4] we can use measured linear acceleration and angular velocity as control input to fuse INS and GPS measurements. Note that the angular velocity input is omitted here for clarity, as already explained, so we define our control input vector:

$$\mathbf{u} = \begin{bmatrix} \ddot{p}_n & \ddot{p}_e & \ddot{p}_d \end{bmatrix}^T$$

From the equations of linear motion $\mathbf{p} = \mathbf{p}_0 + \frac{1}{2}\mathbf{a}t^2$ and $\mathbf{v} = \mathbf{v}_0 + \mathbf{a}t$ we define our control input model $\mathbf{B}$ so that

$$\mathbf{Bu} = \begin{bmatrix} \frac{1}{2}t^2 & 0 & 0 \\ 0 & \frac{1}{2}t^2 & 0 \\ 0 & 0 & \frac{1}{2}t^2 \\ t & 0 & 0 \\ 0 & t & 0 \\ 0 & 0 & t \end{bmatrix} \begin{bmatrix} \ddot{p}_n \\ \ddot{p}_e \\ \ddot{p}_d \end{bmatrix}$$

describes the change in position and velocity due to linear acceleration.

Process Model The process model is given by equation 5.4 and denotes how a combination of the state transition model $\mathbf{A}$ and the control input model $\mathbf{B}$ follows the modelled process with some process noise $\mathbf{w}$. The main source of noise in our process model is the IMU sensor measurements in the control input $\mathbf{u}$ and the process noise $\mathbf{w} \sim \mathcal{N}(0, \mathbf{Q})$ is assumed to be drawn from a zero mean multivariate normal distribution with covariance $\mathbf{Q}$. The predicted estimate covariance $\mathbf{P}_k^-$ can then model the linear increase in uncertainty of the INS estimate over time by equation 4.13. This is also illustrated in figure 5.16 by the dotted circles.

From the sensor specifications in section 5.5.3 we have the standard deviations

$$\begin{aligned}\mathbf{S}_Q &= \begin{bmatrix} \sigma_{accel,right} & \sigma_{accel,up} & \sigma_{accel,right} \end{bmatrix}^T \\ &= \begin{bmatrix} 7.1E-3\,g & 8.9E-3\,g & 7.1E-3\,g \end{bmatrix}^T\end{aligned}$$

for the control input $\mathbf{u}$ and define the process model covariance matrix:

$$\mathbf{Q} = \mathbf{B}\mathbf{B}^T\mathbf{S}_Q^2 = \begin{bmatrix} \frac{1}{4}t^4 & 0 & 0 \\ 0 & \frac{1}{4}t^4 & 0 \\ 0 & 0 & \frac{1}{4}t^4 \\ \frac{1}{2}t^3 & 0 & 0 \\ 0 & \frac{1}{2}t^3 & 0 \\ 0 & 0 & \frac{1}{2}t^3 \end{bmatrix} \begin{bmatrix} (7.1E-3)^2 \\ (8.9E-3)^2 \\ (7.1E-3)^2 \end{bmatrix}$$

Observation Model The GPS sensor measures position and velocity with known noise distributions. We define our observation vector:

$$\mathbf{z} = \begin{bmatrix} p_n & p_e & p_d & \dot{p}_n & \dot{p}_e & \dot{p}_d \end{bmatrix}^T$$

From equation 4.11 we define our observation model $\mathbf{H}$ so that

$$\begin{aligned}\mathbf{z} = \mathbf{H}\mathbf{x} + \mathbf{v} &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_n \\ p_e \\ p_d \\ \dot{p}_n \\ \dot{p}_e \\ \dot{p}_d \end{bmatrix} + \mathbf{v} \\ \mathbf{v} &\sim \mathcal{N}(0, \mathbf{R})\end{aligned}$$

describes the observed position and velocity with some observation noise $\mathbf{v}$. The noise is assumed to be drawn from a zero mean multivariate normal distribution with covariance

R. Filling in the values from the sensor specifications in section 5.5.2 we get the standard deviations

$$\begin{aligned}\mathbf{S}_R &= \begin{bmatrix} \sigma_{GPS,pos,n} & \sigma_{GPS,pos,e} & \sigma_{GPS,pos,d} & \sigma_{GPS,vel,n} & \sigma_{GPS,vel,e} & \sigma_{GPS,vel,d} \end{bmatrix}^T \\ &= \begin{bmatrix} 2.83\,m & 2.83\,m & 2.83\,m & 0.05\,m/s & 0.05\,m/s & 0.05\,m/s \end{bmatrix}^T\end{aligned}$$

for our observation **z** and define the GPS noise covariance matrix:

$$\mathbf{R} = \mathbf{S}_R\mathbf{S}_R^T = \begin{bmatrix} 7.98 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7.98 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7.98 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.5E-3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.5E-3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.5E-3 \end{bmatrix}$$

Applying the Kalman Filter in the Simulation Now that we have defined the process model, the observation model and their covariances from sensor datasheets solving the Kalman filter is simply a matter of applying the equations described in 4.4.2. We implemented the Kalman filter in the class GPSINSFilter, which computes state estimates each game loop iteration from GPS and INS measurements.

First the filter must be initialized to an initial guess of the starting condition. In our helicopter test scenarios we assumed that the starting condition was well known and set $\hat{\mathbf{x}}_0$ to reflect the true position, velocity and orientation of the helicopter. When the initial state is well known we can also set the initial estimate covariance $\mathbf{P}_0$ to a zero matrix, which will let the filter consider this initial estimate as certain.

For each game loop iteration the filter receives noisy IMU measurements of linear acceleration and angular velocity. As described in section 4.4.2 the filter then *predicts* the new state $\hat{\mathbf{x}}_k^-$ based on the previous state estimate $\hat{\mathbf{x}}_{k-1}$, the current control input $\mathbf{u}_k$ and the known process model outlined by the prediction equations 4.12 and 4.13.

GPS measurements of position and velocity are received once per second. When observations are available then the former prediction is *corrected* by an optimal Kalman gain

factor that distributes its trust between the process model estimate and the observation model estimate. The trust distribution is based on the modelled covariances of the INS and GPS estimates and the accumulated uncertainty of the INS estimate versus the uncertainty of the GPS measurements. The corrected estimates for state $\hat{\mathbf{x}}_k$ and covariance $\mathbf{P}_k$ are the final estimates for state k and in the next game loop iteration those estimates are used to once more predict state $k+1$.

As we already noted, our implementation of the Kalman filter omits the angular velocities in the control input vector $\mathbf{u}$ and simply inserts the simulated orientation into the estimated state vector $\hat{\mathbf{x}}$. Errors in estimated orientation will greatly affect the INS position uncertainty over time, so to compensate we periodically added a proper random error to the yaw, pitch and roll angles.

Fusion of Asynchronous Sensors One big issue with sensors is that they often run at very different sampling frequencies. In our GPS/INS filter the GPS samples once per second, while the IMU is configured to sample 60 times per second. To model the filter to trust entirely on INS estimates in-between GPS updates we set the observation model matrix $\mathbf{H}$ to a zero matrix the 59 times per second when no new GPS observation was available. This caused the optimal Kalman gain matrix $\mathbf{K}$ to zero out the row-column pairs for the observed position and the velocity state variables and in effect ignore those observations in the corrected estimate $\hat{\mathbf{x}}$.

Range Finder for HAG Measurements The range finder in section 5.5.4 was added to accurately measure the height above the ground (HAG). This was necessary because the INS/GPS was too inaccurate for low level flights along the curvature of the terrain. Ideally one should filter the HAG, but due to the way our range finder measured HAG its noise distribution was no longer Gaussian and that violated one of the Kalman filter assumptions in section 4.4.2. Instead we simply replaced the GPS/INS altitude estimate by the range finder HAG measurement and let the autopilot maintain a certain height above the ground during navigation.

5.6.2. Separating Linear and Angular State Updates

We had to take special care to model the Kalman filter process model exactly identical to the simulation physics model to avoid estimate deviations due to subtle implementation

differences. Unfortunately, as described in section 5.6.1, it proved difficult to model the change in angular state in the Kalman filter so we had to make a simplification. Figure 5.18 shows how the simulation first updates the linear state and then updates the angular state. This way the two are isolated and much easier to model in the Kalman filter. We assumed the effects of this simplification would be negligible since the small timesteps would minimize the error and we were not using a very realistic physics model to begin with.

5.6.3. Estimation Validity

One very strict requirement to our implementation was that the simulation and the state estimator should produce the exact same results if a certain condition was met. A virtual sensor is said to be *perfect* when no noise is added to the measurement and a sample is read each timestep. If we simulate using only perfect sensors then the simulated linear force vector and the angular velocities should be perfectly reconstructed by the IMU each timestep. We call this the *True Observer Condition*.

This requirement was crucial to ensure that any deviation between the true and the estimated states was due to noisy and incomplete measurements and not errors in the implementation. Without this requirement the estimation errors would not be useful to evaluate the autopilot performance. The test results for configuration #1 in section 6.5 shows that we were able to satisfy this requirement using perfect virtual sensors.

Validity Broken by Precision Issue The True Observer Condition required the physics simulation and the state estimator to produce identical results. Here we discovered a problem that the simulated and the estimated orientation deviated after many iterations even when the condition was met. The deviation in estimated orientation caused the position estimate to deviate by 3 cm after 30 seconds of flight and quickly diverge as the velocity error accumulated.

After some research we isolated the error to two pieces of code that calculated the new state in the simulator and in the state estimator. Given identical inputs these should produce exactly the same results. The reason was a subtle syntactical difference illustrated by the pseudo-code in algorithm 5.3.

Here result1 and result2 was expected to produce equal results, but their precision differed due to technical details covered in appendix D.1. The workaround was to ensure

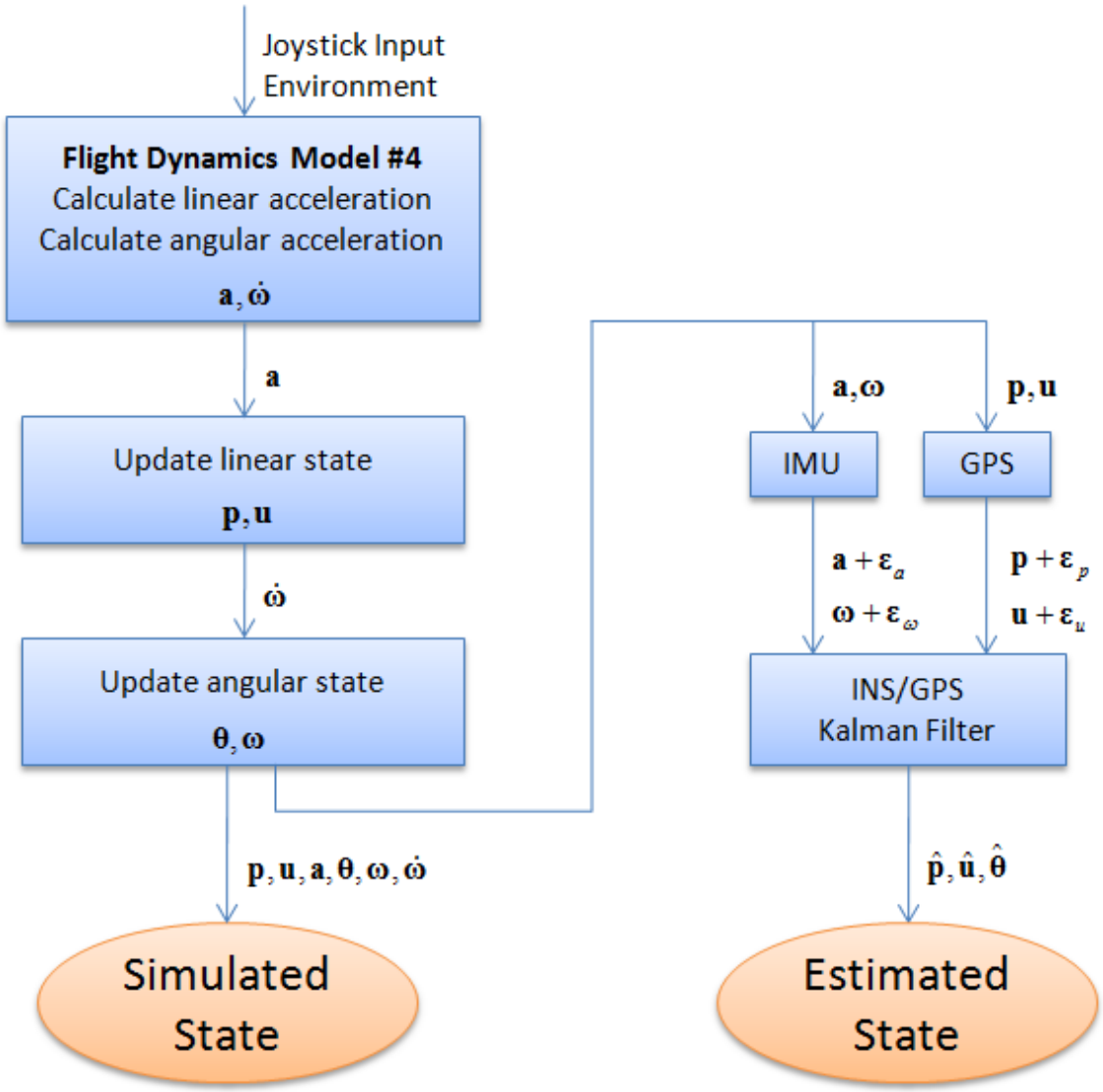


Figure 5.18.: Update sequence of simulation and state estimator.

Algorithm 5.3 Pseudo-code for floating point precision issue.

```
const float A, B, C, dt; // Any non-zero values
float result1 = (A*B)*dt;
float result2 = (A*B);
result2 *= dt;
```

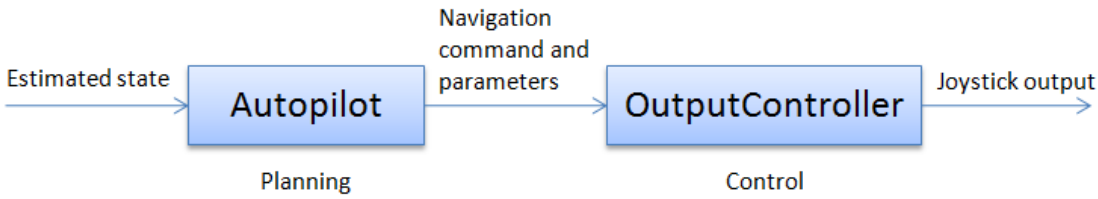


Figure 5.19.: The autopilot and output controller components.

that both the simulation and the state estimator used the exact same piece of code to calculate the change in orientation.

Validity Broken by Collision Handling When we introduced collision handling with Jitter the physics library we realized that the validity requirement could no longer be met. The physics calculations were now largely done by the library and we could no longer guarantee that state estimator would compute the exact same results. This was very unfortunate and was not discovered before at the end of the project. This meant that while using collision handling we could no longer prove that any deviation in the state estimation was a result of imperfections in the sensor data alone. The solution was to disable collision handling by Jitter if we later needed to re-verify the validity of the state estimation.

5.7. Autopilot

Now that we have an understanding of how physics simulation, virtual sensors and state estimation work we can approach the task of getting the helicopter to fly on its own.

5.7.1. Overview

The autopilot control logic is divided in two. The autopilot class handles the navigation planning and breaks it down into simple navigation commands, which the output controller then transforms into joystick outputs as shown in figure 5.19.

The autopilot is very comprehensive so an overview is given in figure 5.20 on how the the output controller, Kalman filter and PID configurations are set up according to the velocity controlled cyclic navigation method we ended up using, explained later in this section.

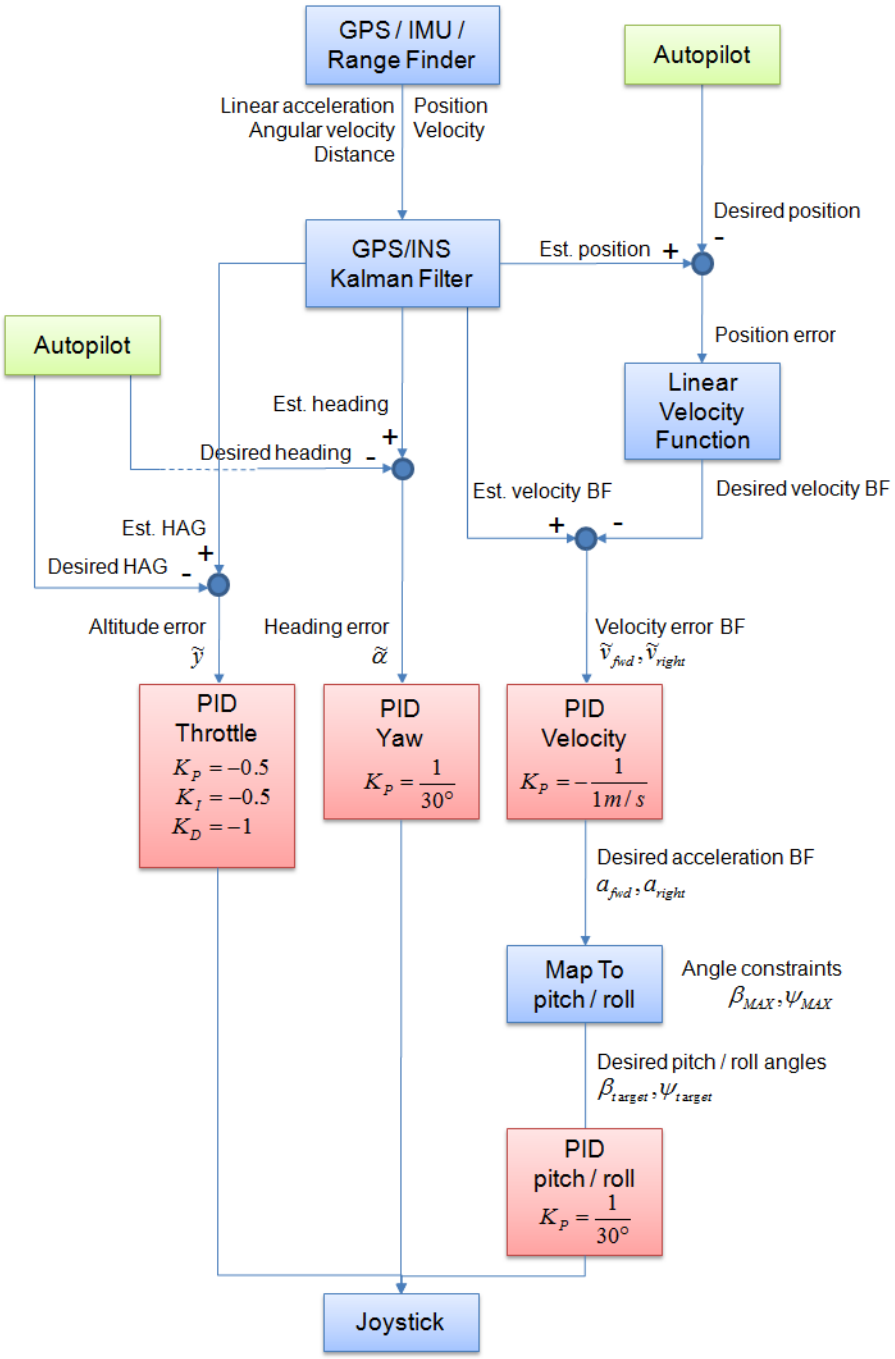


Figure 5.20.: Overview of the autopilot and the PID configurations for the velocity controlled cyclic navigation method. BF denotes body frame.

5.7.2. Autopilot Class

The autopilot is primarily concerned with how to avoid crashing and how to get from A to B. When the autopilot is initialized it is given a navigation task, which holds a series of waypoints. It is the responsibility of the autopilot to keep track of the task progression and command the output controller so that all waypoints are visited in sequence. We implemented three commands; en route, hover and recovery. En route tells the output controller to move towards B without crashing, while hover attempts to keep the helicopter at a fixed position in the air. Recovery is issued if the autopilot detects that we are in risk of crashing and will override all logic to gain altitude as quickly as possible.

5.7.3. State Provider Classes

The only input to the autopilot is the current state of the helicopter, given by any IStateProvider implementation as seen in figure 5.2. The helicopter state mainly consists of its position, orientation, their derivatives and the navigation state. This was an architectural choice that allowed us to swap on-the-fly between a perfect state provider and a sensor-based state provider with different sensor configurations. This was not only useful during development, but also critical to accomplish the automated testing system described in section 5.8.

5.7.4. Output Controller Class

The output controller is concerned with producing joystick output from the current state and the navigation command given by the autopilot. Helicopters are very hard to develop navigation logic for since they are unstable by nature and will quickly drift out of control. In this section we describe a few methods we developed that enabled the helicopter to maneuver safely. We assume a perfect state provider is used unless otherwise noted to leave the concerns of state inaccuracy out of the details.

En Route Command The command we spent the most time developing logic for was for navigating from A to B. More accurately, this method is only concerned with getting to B from where it currently believe it is. One simplification we found natural to do

was to think navigation in the horizontal plane, as if looking at map. We simplified any vertical navigation to the concept of ascending and descending. By assuming the helicopter would always maintain near-level flight we could isolate this part to only control the throttle output. For navigation we then only needed to think in the horizontal plane and we will use the h-vector notation here to underline this.

Throttle The throttle control logic is very straight forward. As long as the pitch and roll angles are constrained to near level flight increasing the throttle will increase the upwards lift component. Controlling the throttle is then a matter of defining altitude error in the navigation frame as:

$$\tilde{y} = p_d - y \tag{5.6}$$

p_d helicopter altitude

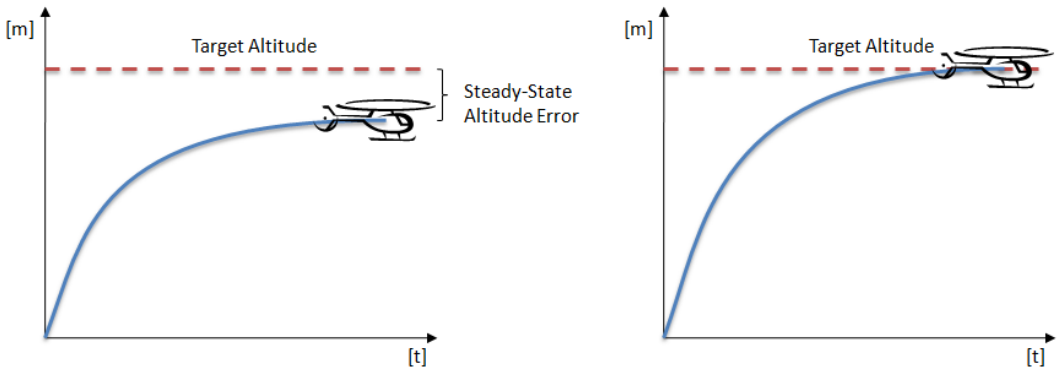
y target altitude

Our problem is now essentially to let $\tilde{y}$ approach zero to reach our target altitude. We assigned a PID-controller $PID_{throttle}$ to regulate this and we insert the altitude error into equation 4.9 and get:

$$PID_{throttle} = K_P \cdot \tilde{y}(t) + K_I \cdot \int \tilde{y}(t)dt + K_D \cdot \frac{d}{dt}\tilde{y}(t),\ PID_{throttle} \in [0,1] \tag{5.7}$$

To implement the equation we discretize into timesteps and formulate a recursive equation to compute the PID output. By accumulating the error in f_n we only need store values for the current and the previous timestep.

$$\begin{aligned} PID_{throttle} &= K_P e_n + K_I f_n + K_D \frac{e_n - e_{n-1}}{t_n - t_{n-1}},\ PID_{throttle} \in [0,1] \\ e_n &= \tilde{y}(n) \\ f_n &= f_{n-1} + e_n \\ f_0 &= 0 \\ n &\geq 1 \end{aligned}$$



(a) PD-controller: The proportional gain is not sufficient to overcome the constant gravity.

(b) PID-controller: The integral term helps increase the throttle to overcome gravity .

Figure 5.21.: PD- vs. PID-controller for altitude control loop.

n	timestep
e_n	altitude error at timestep n
f_n	accumulated altitude error at timestep n
t_n	time at timestep n
K_P	proportional gain
K_I	integral gain
K_D	derivative gain

When the error e_n is negative the helicopter is below its target altitude and the proportional term will apply a thrust to produce lift. Due to gravity we will typically not reach this altitude, because the lifting force by the proportional term is eventually outweighed by the force of gravity as shown in figure 5.21(a). As time passes the integral term will then accumulate enough error to raise the thrust just enough to reach the target altitude. If we configure the K_P and K_I gains aggressively to more quickly reach the target we are likely to overshoot the the target and oscillate back and forth across the target. The derivative term will then dampen this motion as a dampened spring and allows the helicopter to quickly and smoothly reach its target altitude and stay there as shown in figure 5.21(b). The PID settings used in the implementation are listed in table 6.1.

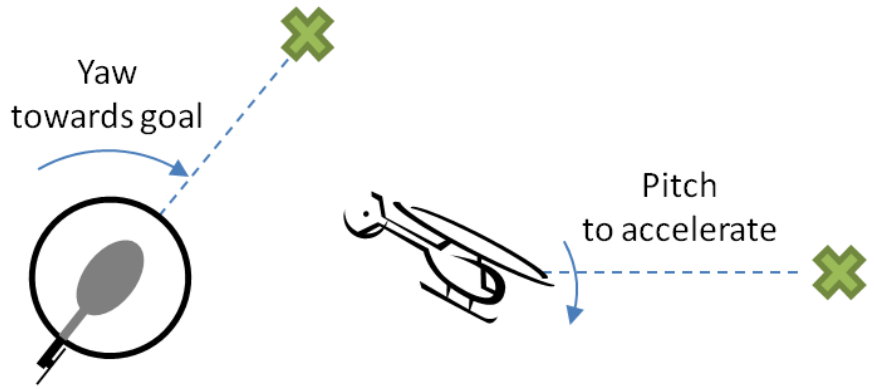


Figure 5.22.: Nose-on navigation by yawing and pitching.

Nose-On Navigation Now that throttle control is out of the way we can concentrate on the horizontal navigation. The first method we attempted was *nose-on navigation*. We realized that we could accelerate forwards in the horizontal plane by controlling the pitch angle. Applying negative pitch (nose down) was the equivalent of accelerating along the h-forward vector and we could direct this acceleration by yawing the nose to point where we wanted to go. Figure 5.22 illustrates this.

The problem was that in most cases the helicopter starts with the nose pointing in an arbitrary direction. Since the helicopter must yaw the nose towards the goal the acceleration vector will lag behind and often be directed off the target. The faster the helicopter was configured to move the larger this error became and it often missed the waypoint radius entirely. In addition, if the helicopter overshot its waypoint position it had to turn 180° in order to accelerate in the opposite direction. This would typically oscillate back and forth around the waypoint and never come to rest. For sensor-based state providers this method performed particularly bad due to jumps in the estimated position.

Acceleration Controlled Cyclic Navigation A far better method that we implemented used *cyclic navigation*. The idea was that it is possible to navigate horizontally using pitch and roll, described in [9] as the *cyclic controls*. Pointing the nose in a particular direction is not necessary for maneuvering a helicopter. This method was slightly more intricate as we needed to transform our desired acceleration vector $\mathbf{a}_{desired}$ towards the target into a combination of desired pitch and roll angles as shown in figure 5.23. First we define the position error in body frame as $\tilde{p}_{fwd}$ and $\tilde{p}_{right}$ by projecting the position error vector

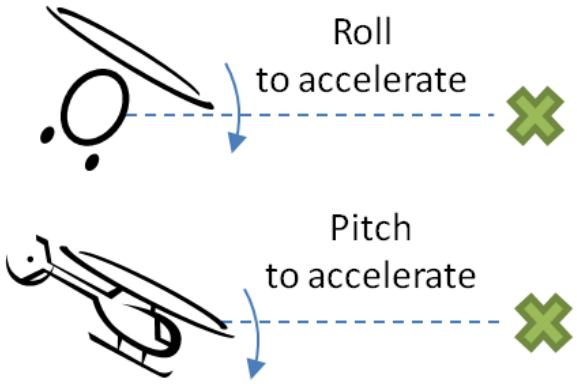


Figure 5.23.: Cyclic navigation by pitching and rolling to accelerate in the horizontal plane.

$$\tilde{\mathbf{p}} = \mathbf{p} - \mathbf{p}_{target} \tag{5.8}$$

onto the body frame basis vectors frame basis vectors $\mathbf{e}_{fwd}$ and $\mathbf{e}_{right}$ by the scalar projection function:

$$\tilde{p}_{fwd} = project(\tilde{\mathbf{p}}, \mathbf{e}_{fwd}) \tag{5.9}$$

$$\tilde{p}_{right} = project(\tilde{\mathbf{p}}, \mathbf{e}_{right}) \tag{5.10}$$

Now we have formulated our problem to minimize the errors $\tilde{p}_{fwd}$ and $\tilde{p}_{right}$ in order to reach our target position. We then assign two PID-controllers PID_{pitch} and PID_{roll} to control the pitch and roll angles so that the resulting *acceleration* minimizes the body frame position error. From the PID equation 4.9 we define our pitch and roll PID outputs:

$$PID_{accel,fwd} = K_P \cdot \tilde{p}_{fwd} + K_D \cdot \frac{inv_{fwd} \cdot \triangle\tilde{p}_{fwd}}{\triangle t},\ PID_{accel,fwd} \in [-1, 1] \tag{5.11}$$

$$PID_{accel,right} = K_P \cdot \tilde{p}_{right} + K_D \cdot \frac{inv_{right} \cdot \triangle\tilde{p}_{right}}{\triangle t},\ PID_{accel,right} \in [-1, 1] \tag{5.12}$$

$$inv_{fwd} = \begin{cases} \tilde{p}_{fwd} \geq 0, & 1 \\ \tilde{p}_{fwd} < 0, & -1 \end{cases} \tag{5.13}$$

$$inv_{right} = \begin{cases} \tilde{p}_{right} \geq 0, & 1 \\ \tilde{p}_{right} < 0, & -1 \end{cases} \tag{5.14}$$

We interpret the proportional term so that the greater the distance to the target, the more it will attempt to increase pitch and roll angles to accelerate towards it. The derivative term will dampen the approach to avoid overshoot by decelerating the helicopter as the proportional term becomes less significant. The inverse factors are needed because acccelerating when the target is in front (positive forward error) requires a negative pitch angle (nose down) and when the target is behind the helicopter (negative forward error) it must be positive (nose up). The same concept applies to roll. The integral term is not used and is omitted here.

The PID outputs are clamped to -1 and 1. If we consider the PID output as a desire to accelerate along each axis, with 1 and -1 being maximum acceleration in either direction along the axis, then we can simply multiply this value by some maximum angle to obtain our pitch and roll target angles. The yaw angle is not necessary for navigation and omitted here. The target angles are then fed into a P-controller that applies the proper joystick outputs to correct the pitch and roll angles.

$$\beta_{target} = \beta_{MAX} \cdot PID_{accel,fwd},\ \beta_{target} \in [-\beta_{MAX}, \beta_{MAX}] \tag{5.15}$$

$$\psi_{target} = \psi_{MAX} \cdot PID_{accel,right},\ \psi_{target} \in [-\psi_{MAX}, \psi_{MAX}] \tag{5.16}$$

$$PID_{pitch} = K_P \cdot (\beta - \beta_{target}),\ PID_{pitch} \in [-1, 1] \tag{5.17}$$

$$PID_{roll} = K_P \cdot (\psi - \psi_{target}),\ PID_{roll} \in [-1, 1] \tag{5.18}$$

We already specified that max throttle would produce a lifting force of 1.7G. We wanted to preserve a vertical lift force component of 1.5G at all times and from $1.7G\cos^2\theta = 1.5G$ we find that the pitch and roll angles should never exceed 20°. In our implementation we used 10° to be on the safe side. When the helicopter was far away from its target it would pitch and roll up to 10° to gain speed towards the target. As it approached its destination the derivative term would gradually become dominant and decelerate the helicopter until it came to rest at its target. We successfully achieved nose-independent navigation using this method and proved this by forcing the helicopter to continuously yaw clockwise while navigating through waypoints.

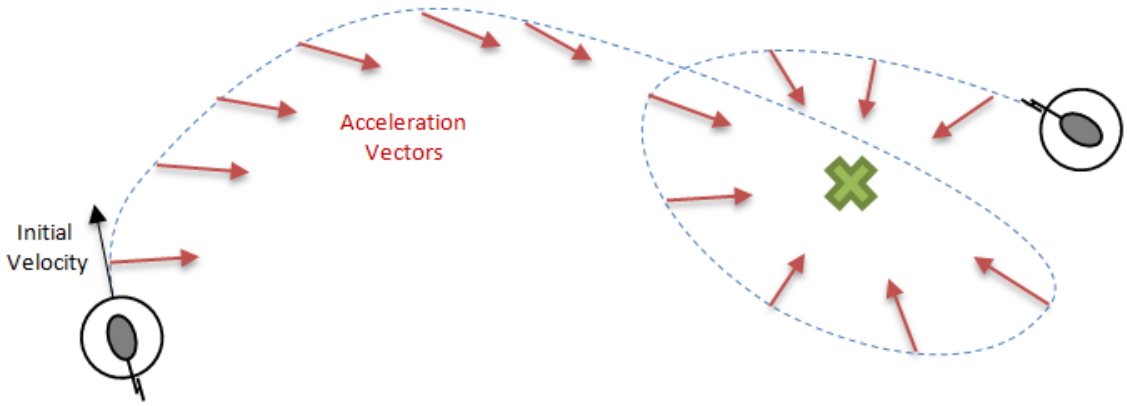


Figure 5.24.: Accelerating towards the target results in circular motions near the target.

Velocity Controlled Cyclic Navigation The cyclic navigation method did perform well for large parts of the project, but later we wanted to challenge the autopilot precision by decreasing the radius of the waypoints and hovering at fixed positions. That is when we discovered that our method would often miss the waypoints slightly and would never seem to come to a full stop, but rather move around the spot in a circular motion.

Both problems are related to the fact that we are always accelerating *towards* the target position. Although it seemed like a good idea at first it became evident that this was the same as a particle attached to a pivot by a string. If the particle has a velocity then the string acts a centripetal force on the particle towards the pivot, equivalent to our acceleration by pitch and roll towards the target. Figure 5.24 shows how this acceleration becomes orthogonal to the circular motion velocity and that is why it will keep circling the target.

Our final method overcomes this problem by realizing that we need to control our position in terms of desired *velocity* and not the desired acceleration. An overview of this method is illustrated by figure 5.20. We reformulate our problem so that we want our h-velocity vector $\mathbf{v}$ to point towards the target. First we define velocity error in body frame as $\tilde{v}_{fwd}$ and $\tilde{v}_{right}$ by projecting the navigation frame velocity error

$$\tilde{\mathbf{v}} = \mathbf{v} - \mathbf{v}_{desired} = \mathbf{v} - v_{nav}\frac{\mathbf{p}_{goal} - \mathbf{p}}{|\mathbf{p}_{goal} - \mathbf{p}|} = \mathbf{v} - v_{nav}\frac{-\tilde{\mathbf{p}}}{|-\tilde{\mathbf{p}}|} = \mathbf{v} + v_{nav}\frac{\tilde{\mathbf{p}}}{|\tilde{\mathbf{p}}|} \tag{5.19}$$

onto the body frame basis vectors frame basis vectors $\mathbf{e}_{fwd}$ and $\mathbf{e}_{right}$ as shown in figure 5.25:

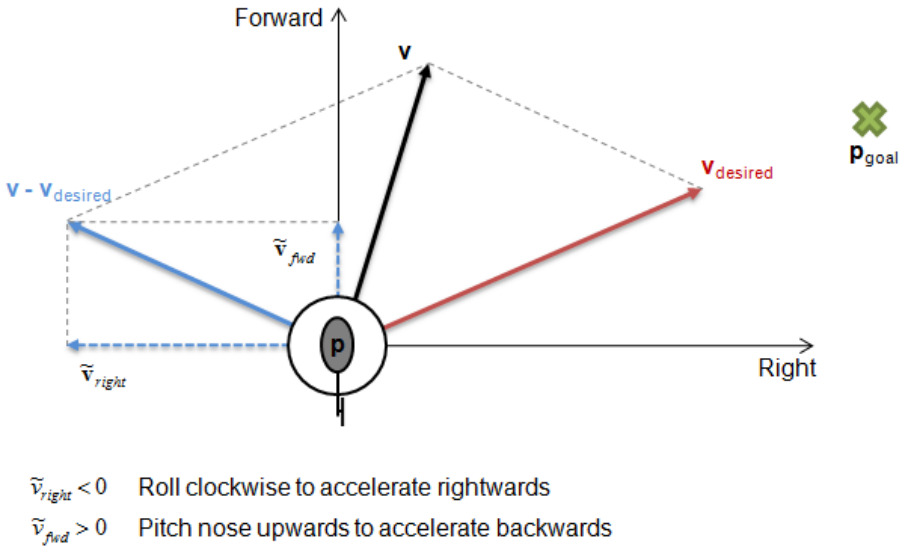


Figure 5.25.: Pitch and roll is controlled to minimize the error velocities $\tilde{v}_{fwd}$ and $\tilde{v}_{right}$ to get $\mathbf{v}$ to approach $\mathbf{v}_{desired}$.

$$\tilde{v}_{fwd} = project(\tilde{\mathbf{v}}, \mathbf{e}_{fwd}) \tag{5.20}$$

$$\tilde{v}_{right} = project(\tilde{\mathbf{v}}, \mathbf{e}_{right}) \tag{5.21}$$

In addition to directing our velocity vector towards the target we also need to specify how fast we should be moving towards it. We used a simple algorithm where the desired velocity magnitude $|\mathbf{v}_{desired}|$ was a linear function of distance and clamped at a max horizontal speed $v_{h,max}$. This way we could smoothly decelerate the helicopter the last few meters towards the target.

Similar to our previous method we have formulated our problem to let errors $\tilde{v}_{fwd}$ and $\tilde{v}_{right}$ approach zero in order to reach our target position. Again we use PID-controllers, but this time we want to direct our *velocity* towards the target by controlling the pitch and roll angles. From equations 4.9, 5.15 and 5.16 we get equations for our target pitch and roll angles. The target angles are then fed into a P-controller that applies the proper joystick outputs to correct the pitch and roll angles. The P-controller for the optional yaw control is also listed here.

$$PID_{velocity,fwd} = K_P \cdot \tilde{v}_{fwd},\ PID_{velocity,fwd} \in [-1, 1] \tag{5.22}$$

$$PID_{velocity,right} = K_P \cdot \tilde{v}_{right},\ PID_{velocity,right} \in [-1, 1] \tag{5.23}$$

$$\beta_{target} = \beta_{MAX} \cdot PID_{velocity,fwd},\ \beta_{target} \in [-\beta_{MAX}, \beta_{MAX}] \tag{5.24}$$

$$\psi_{target} = \psi_{MAX} \cdot PID_{velocity,right},\ \psi_{target} \in [-\psi_{MAX}, \psi_{MAX}] \tag{5.25}$$

$$PID_{pitch} = K_P \cdot (\beta - \beta_{target}),\ PID_{pitch} \in [-1, 1] \tag{5.26}$$

$$PID_{roll} = K_P \cdot (\psi - \psi_{target}),\ PID_{roll} \in [-1, 1] \tag{5.27}$$

$$PID_{yaw} = K_P \cdot (\alpha - \alpha_{target}),\ PID_{yaw} \in [-1, 1] \tag{5.28}$$

With this method the integral and derivative terms are not required to control the pitch and roll angles since the velocity function will ensure that the helicopter approaches the target quickly and smoothly. Also, as explained in section 5.4, the orientation of the model helicopter is quite stable and will quickly damp angular velocity on its own. This means that a simple P-controller should be able to control the pitch, roll and yaw angles with very little oscillation. The yaw angle is also controlled by the autopilot, but it is not required for navigation and omitted here.

Scalar Projection Function Our scalar projection function $project(\mathbf{a}, \mathbf{b})$ returns the length of $\mathbf{a}$ projected onto $\mathbf{b}$. The dot product for three dimensional vectors is defined by $\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$ and its geometric interpretation $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}|\,|\mathbf{b}| \cos\theta$. It is readily seen that our scalar projection is defined by the length of $\vec{\mathbf{a}}$ and its angle θ to $\mathbf{b}$:

$$project(\mathbf{a}, \mathbf{b}) = |\mathbf{a}| \cos\theta \tag{5.29}$$

We shuffle the above equations and obtain the scalar projection function.

$$project(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{b}|} = \frac{a_x b_x + a_y b_y + a_z b_z}{|\mathbf{b}|} \tag{5.30}$$

Hover Command Our initial attempts to hover at a position was to tell the output controller to maintain level flight ($\alpha = 0, \beta = 0$) and to maintain the current altitude. The problem was that if the helicopter already had a velocity the helicopter would move a long distance before drag dampened the motion to a full stop. Fortunately, once we had a working method for navigation then hovering became very simple. We now simply tell the output controller to move to the position we were at the time the command was issued and let the navigation method deal with all the details.

Recovery Command The implementation for this command is also very simple. Assuming the recovery command is issued if there is an immediate danger of crashing into the ground we simply tell the output controller to maintain level flight ($\alpha = 0$, $\beta = 0$) and to maintain full throttle. Hopefully this will generate enough lift to prevent the crash. Obviously there are more elegant solutions, such as avoiding trees and walls, but in our simple terrain navigation scenarios this worked well.

5.7.5. Autopilot Assisted Control

The autopilot assisted control (AAC) was a concept we came up with when we observed that most persons attempting manual flight in our simulator ended up crashing a lot. We realized that having a working autopilot made it possible to let the autopilot handle all the details of keeping the helicopter stable and simply follow navigational commands of the user on where to go. This would enable persons with no flying experience to maneuver the helicopter safely and efficiently. We already had a working method for autonomous flight so adding assisted control was simply a matter of extending our method to accept joystick commands from the user.

We designed the AAC so that if the joystick was let go the helicopter would safely come to a hover at its current position in the air. Holding the joystick forwards would make the helicopter move forwards and the same for sideways and backwards. The joystick throttle slider was used to tell the autopilot what altitude above the ground to maintain and the yaw controls (pedals) were used change the direction of the helicopter nose during flight.

By extending our velocity controlled cyclic navigation method we only had to convert joystick outputs into values for desired velocity vector $\mathbf{v}_{desired}$, target height above the ground y and heading angle ψ. Our navigation method already had PID configurations to let the autopilot maneuver the helicopter accordingly so the implementation of AAC was straight forward. Initial tests in our simulation indicated that with sufficiently accurate state estimates the users were almost unable to crash the helicopter. We observed safe flights as low as 2 meters above the ground at speeds up to 36 km/h over curved terrains when there were no trees or obstacles to crash in.

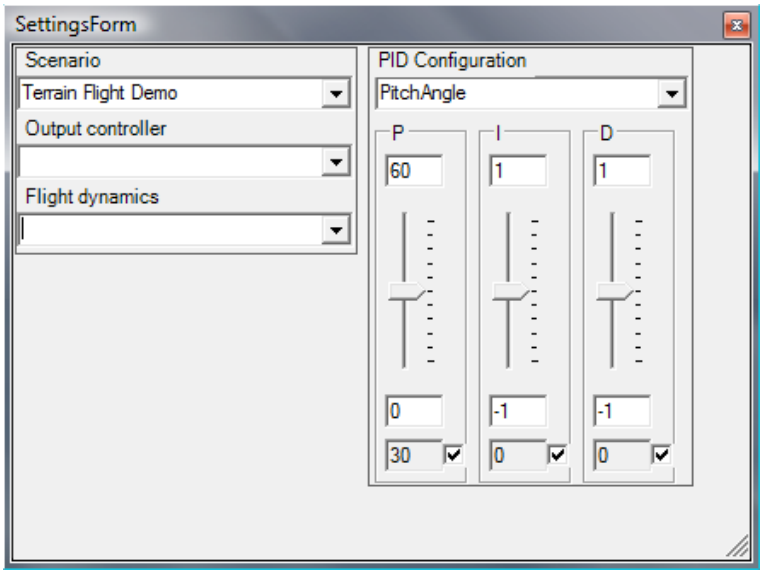


Figure 5.26.: PID settings configuration dialog.

5.7.6. PID Settings

The output controller relies largely on the use of PID-controllers to control the motion of the helicopter. For this to work we must accurately tune the PID settings to fit the flight dynamics of the helicopter to achieve safe and smooth navigation. In section 4.3 we discussed methods to tune the PID coefficients so that the helicopter reaches its destination quickly with minimal overshoot and oscillation. Since we were in a simulator environment we found that the most effective way of tuning the coefficients was by trial and error so we developed a GUI component (figure 5.26) that let us manipulate the PID settings during autonomous navigation to see the effects on-the-fly. Also, given a sufficiently accurate sensor and physics simulation this should provide a good starting point for real flight experiments in future applications. The final PID settings used in our experiments are found in table 6.1.

5.8. Automated Testing

The experiments we performed in section 6.5 required us to test a large number of combinations of autopilot settings, sensor specifications, navigation tasks and terrain configurations. Having to re-compile or change configuration files for each combination was not an option so we implemented an automated testing system.

A test configuration file as described in appendix B.1 lists a set of test scenarios and a set of autopilot settings. The scenarios are also defined in a configuration file as described in appendix B.2 and holds all the remaining information about starting state, sensors, navigation, terrain and world objects to be loaded. Each scenario is then run a number of times to test out all the autopilot settings for that scenario. The test scenario ends when the helicopter crashes, completes the navigation task or exceeds the timeout limit and a flight log is exported to file for review. These logs were then used to present the results in the experiments section.

5.9. Portability to Embedded Solutions

It was the intention that any autopilot logic developed in this simulator should be easily portable to a microcontroller for real navigation in future applications. We noted this in the software architecture design in section 5.2 by requirement R6 that said the autopilot program code and its dependencies should be portable to the programming language C. There were a number of tactics (T1-T4) that helped achieve this, such as minimizing simulator and platform dependencies and isolating the autopilot code to a black box with inputs and outputs. The only dependencies of our autopilot code is an open source matrix math library and a few references to data structures for helicopter state and sensor configurations; so we consider the requirement to be met.

5.10. Audio

XNA has native support for sound effects and music. One can either load audio files directly or use XACT, the included audio tool for creating banks of sounds and adding effects to the sounds.

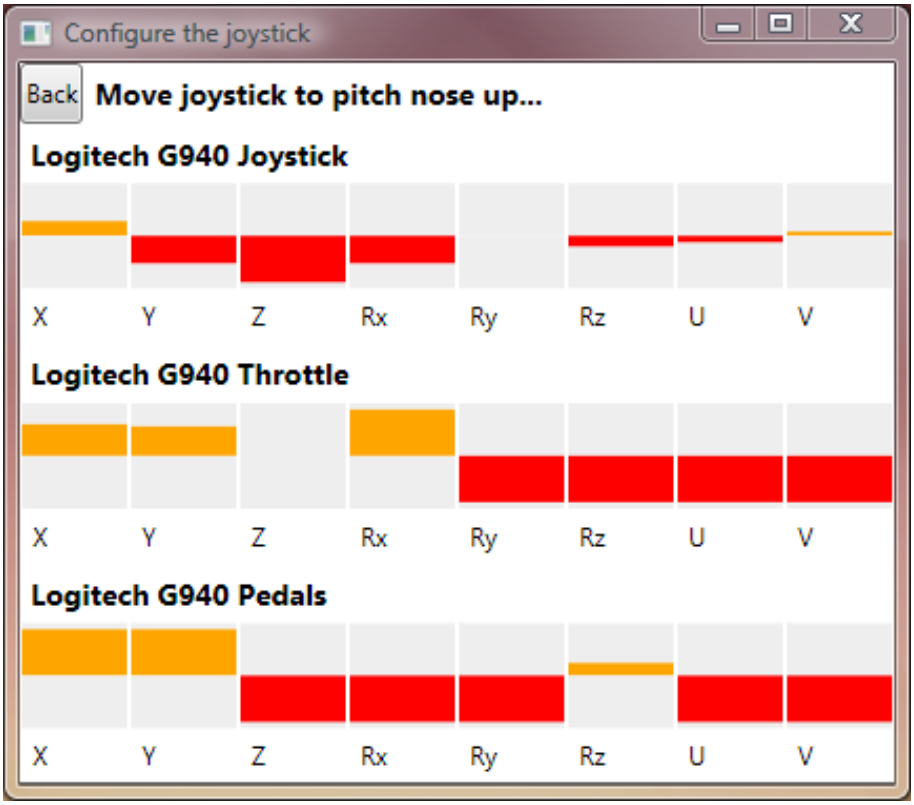
We created an engine sound by extracting a three second audio clip from a recording of a helicopter engine running at a fixed speed. With some careful edge-trimming the clip could be looped to create a continuous engine sound without hearing the loop points. To give an illusion of the engine running faster when applying throttle we used the XNA audio framework to modify the audio pitch on the fly. Raising the pitch by a few notes gave the impression of the engine speeding up and the end result was quite believable.

5.11. Joystick Support

XNA supports mouse, keyboard and gamepads, but not joysticks. To achieve this we interfaced with the DirectInput framework of DirectX. In the early stages of our implementation we reused an open source component to read the the axis and button outputs, but when the lab introduced the Logitech G940 Flight System we ran into an issue. The joystick had 9 joystick axes, 9 axis trims and a large number of buttons and HAT-switches. Any single DirectInput device cannot support more than 8 analog values used for axes and trims so this is why the G940 system is detected as three separate game controllers; joystick, throttle and pedals.

First we had to write our own JoystickSystem class to handle multiple joystick devices. Second there was a need to keep the mapping of functions to different buttons and axes of a joystick in a separate file. Third we had to create a small application to help configure the mapping. We started on creating a wizard to aid the user in setting up a new joystick, but this took a lot of time so figure 5.27(b) shows how we instead list all the connected devices and show their live outputs. Mapping the joystick is then a matter of assigning function names to the corresponding axis names in the configuration file, explained in detail in appendix B. This way our implementation supports any PC joystick no matter what axes or number of devices it has.

(a) Logitech G940 Flight System



(b) Application for mapping functions to axes and buttons.

Figure 5.27.: The G940 joystick system required us to create a joystick configuration application.

6. Results

As a proof of concept we conducted autonomous flight experiments with different levels of state knowledge, from full simulation knowledge to state estimates with realistically modelled IMU and GPS sensors. In this section we list the results of the experiments and all the computer and configuration details required to reproduce them.

6.1. Computer Configuration

Below is the test computer setup we used to produce the results in our flight experiments.

OS: Windows Vista™ Business 64-bit

CPU: Intel® Core™ i7 920 @ 2.66 GHz

GPU: NVIDIA® Quadro® FX 5600 4GB

Memory: 12GB

6.2. Autopilot Configuration

The autopilot configuration variables used throughout the experiments are listed in table 6.1. Note that some variables are defined in code and are not readily configurable. The velocity controlled cyclic navigation method in section 5.7.2 was used throughout all the experiments. The sensor configurations are defined in their respective experiment sections.

Conf. Variable	Value	Description
β_{MAX}	10 °	Max pitch angle.
ψ_{MAX}	10 °	Max roll angle.
HAG_{target}	5 m	Target height above ground.
$v_{h,max}$	3.6 - 36 km/h	Max horizontal velocity.
t_{dec}	1 s	Deceleration reaction time.
s_{dec}	$t_{dec} \cdot v_{h,max}$ m	Distance to start decelerating.
r_{max}	5[a] m	Max distance to pass a waypoint.
PID_{pitch}	$P = 30$, $I = 0$, $D = 0$[b]	PID settings for pitch angle.
PID_{roll}	$P = 30$, $I = 0$, $D = 0$	PID settings for roll angle.
PID_{yaw}	$P = 30$, $I = 0$, $D = 0$	PID settings for yaw angle.
$PID_{throttle}$	$P = -0.5$, $I = -0.5$, $D = -1$	PID settings for throttle.
$PID_{velocity}$	$P = -1$, $I = 0$, $D = 0$	PID settings for velocity.

[a]The precision scenario uses 0.5 m.
[b]The actual coefficients are inverted, such as $K_P = \frac{1}{P}$.

Table 6.1.: The autopilot configuration used in experiments.

6.3. Flight Experiments

Configuration 1: Perfect Knowledge The first experiment we designed gave the autopilot direct insight to the simulation information. This means the helicopter knew its exact position and orientation in the virtual world and the autopilot performance was only limited by the autopilot logic itself. This way we set the bar for the optimal autopilot performance and any deviation from these results would be a direct consequence of having to estimate the state from uncertain information. This configuration was also used to verify that the INS had zero deviation when its IMU sensor data was perfect. The results are listed in table 6.5. Since no sensors were involved for navigation we omit the sensor specifications here.

Configuration 2: Perfect Sensors The second experiment was designed to reveal the effects of precision issues when transforming between reference frames. We have eliminated any noise in the sensors and set the accelerometer to sample infinitely fast (once every simulation timestep). Only the INS estimate was used so the GPS and range finder measurements were ignored to isolate the error sources to the loss of precision when

transforming simulated measurements between navigation and body reference frames, as described in appendix D.

Specification	Value
GPS Freq.	-
GPS 3D Position RMS	-
GPS 3D Velocity RMS	-
Accelerometer Rate	Inf. Hz
Accelerometer Fwd. RMS	0 g
Accelerometer Right RMS	0 g
Accelerometer Up RMS	0 g
Range Finder RMS	-
INS Euler Angle RMS	0 °

Table 6.2.: Sensor specifications for experiment 2.

Configuration 3: Datasheet Sensors This experiment was designed to let the autopilot navigate by realistic magnitudes of sensor noise and state uncertainty. We used the sensor specifications from the datasheets directly as outlined in table 6.3. Since we did not use angular velocity measurements to filter orientation we approximated this uncertainty by periodically adding a random offset to the INS yaw, pitch and roll angles.

Specification	Value
GPS Freq.	1 Hz
GPS 3D Position RMS	4.9 m
GPS 3D Velocity RMS	0.05 m/s
Accelerometer Rate	60 Hz
Accelerometer Fwd. RMS	0.0071 g
Accelerometer Right RMS	0.0071 g
Accelerometer Up RMS	0.0089 g
Range Finder RMS	0.025 m
INS Euler Angle RMS	2 °

Table 6.3.: Sensor specifications for experiment 3.

Configuration 4: Unreliable Sensors The final experiment challenges the autopilot with very high noise levels to see the effects on state estimation and flight behavior.

The sensor specifications were simply chosen by increasing the errors in the datasheet specifications. An interesting point was to halve the accelerometer sampling rate to undersample the truth just as real sensors would. The noise in the orientation estimate was also doubled to promote uncertainty in the INS estimate.

Specification	**Value**
GPS Freq.	1 Hz
GPS 3D Position RMS	10 m
GPS 3D Velocity RMS	1 m/s
Accelerometer Rate	30 Hz
Accelerometer Fwd. RMS	0.71 g
Accelerometer Right RMS	0.71 g
Accelerometer Up RMS	0.89 g
Range Finder RMS	0.5 m
INS Euler Angle RMS	4 °

Table 6.4.: Sensor specifications for experiment 4.

6.4. Navigation Scenarios

Each experiment runs through a set of test scenarios that describes the navigation task and the terrain configuration. Each test scenario will only succeed if the helicopter reaches its destination before timeout and without crashing. If the helicopter crashes the autopilot will try a number of lesser $v_{h,max}$ configurations until it succeeds. If none of them succeeds the test scenario is marked as failed. The complete list of test scenarios and their configurations are listed in appendix B.1.

Outline of Scenarios Each scenario is presented here by a short description and a figure that illustrates the navigation task by visual flight logs. The scale of the navigation tasks makes it impractical to present any details and many of the lines are not distinguishable, so the intention of the figures is merely to give an overview of the different navigation scenarios. Each scenario has three graphs that show the horizontal navigation, the altitude over time and the measured acceleration over time. A legend for the graphs is shown in figure 6.1.

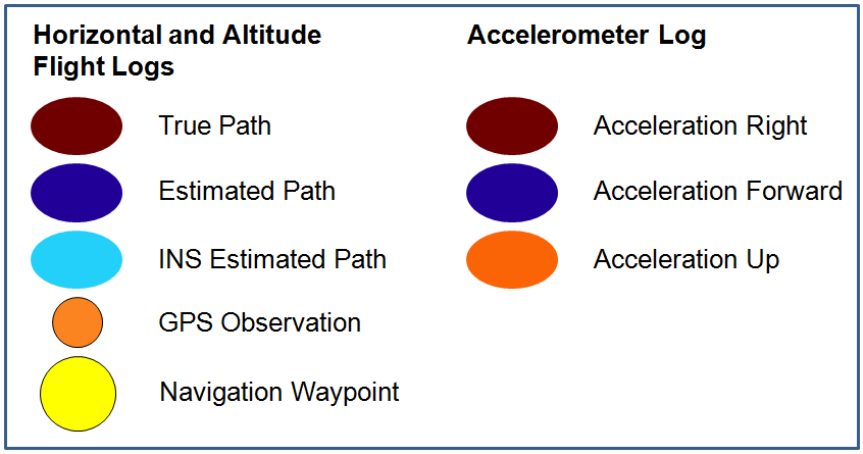


Figure 6.1.: Legend for flight log graphs.

6.4.1. Scenario 1: A-B Short Flat

This is the simplest scenario, which involves a short straight navigation from A to B over flat ground. It is expected that all experiments will pass this scenario unless the noise levels are increased beyond realistic levels. See outline in figure 6.2.

6.4.2. Scenario 2: Circle Medium Sloped

This is the most realistic navigation scenario, which involves passing four waypoints placed in a wide circle. The terrain has significant curvature and the length of navigation is almost 200 meters. The difficulty of passing this scenario is considered medium, but passable for realistically modelled sensors or better. See outline in figure 6.3.

6.4.3. Scenario 3: Circle Large Hilly

This is a very difficult scenario designed to challenge the autopilot. The circle is over 300 meters and the terrain is much steeper so we did not expect sensor-based autopilots to be able to pass this one. See outline in figure 6.4.

6.4.4. Scenario 4: Circle Precision Short Flat

This is also an artificially difficult scenario designed to challenge the precision of the autopilot. While all the other scenarios uses 5 m waypoint radiuses this one uses 0.5 m. The navigation needs to be very accurate and we did not expect sensor based experiments to pass this. See outline in figure 6.5.

6.5. Test Results

This section describes the results of the flight experiments we conducted on the four test scenarios. The experiments were designed to measure the impact of unreliable knowledge in the autopilot performance over different navigation scenarios. We conducted four experiments ranging from perfect simulation knowledge to poor sensor data and the results are listed in tables 6.5 to 6.8 and figures 6.6 to 6.9.

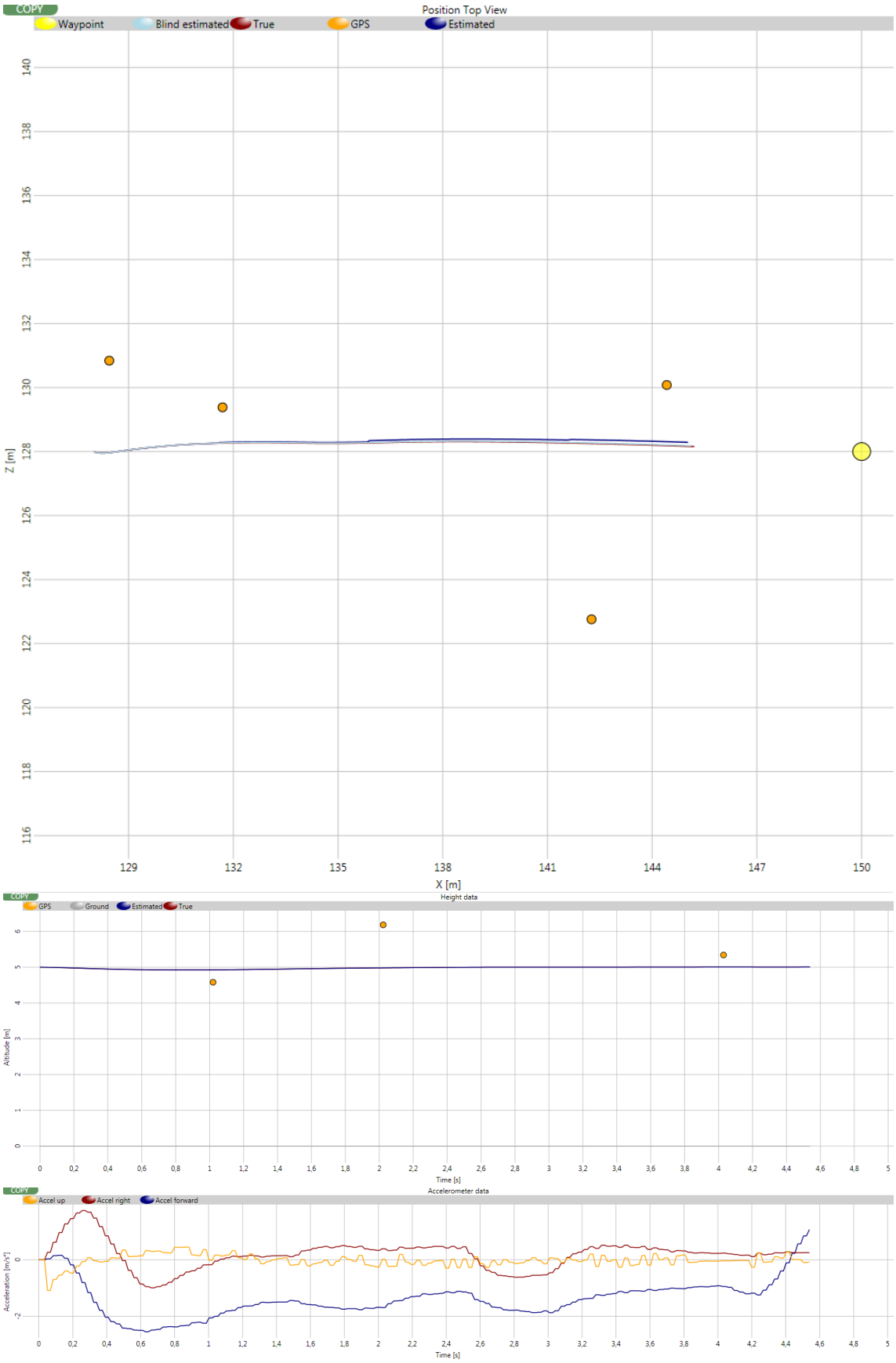


Figure 6.2.: Outline of scenario by flight log from configuration #3 in scenario #1.

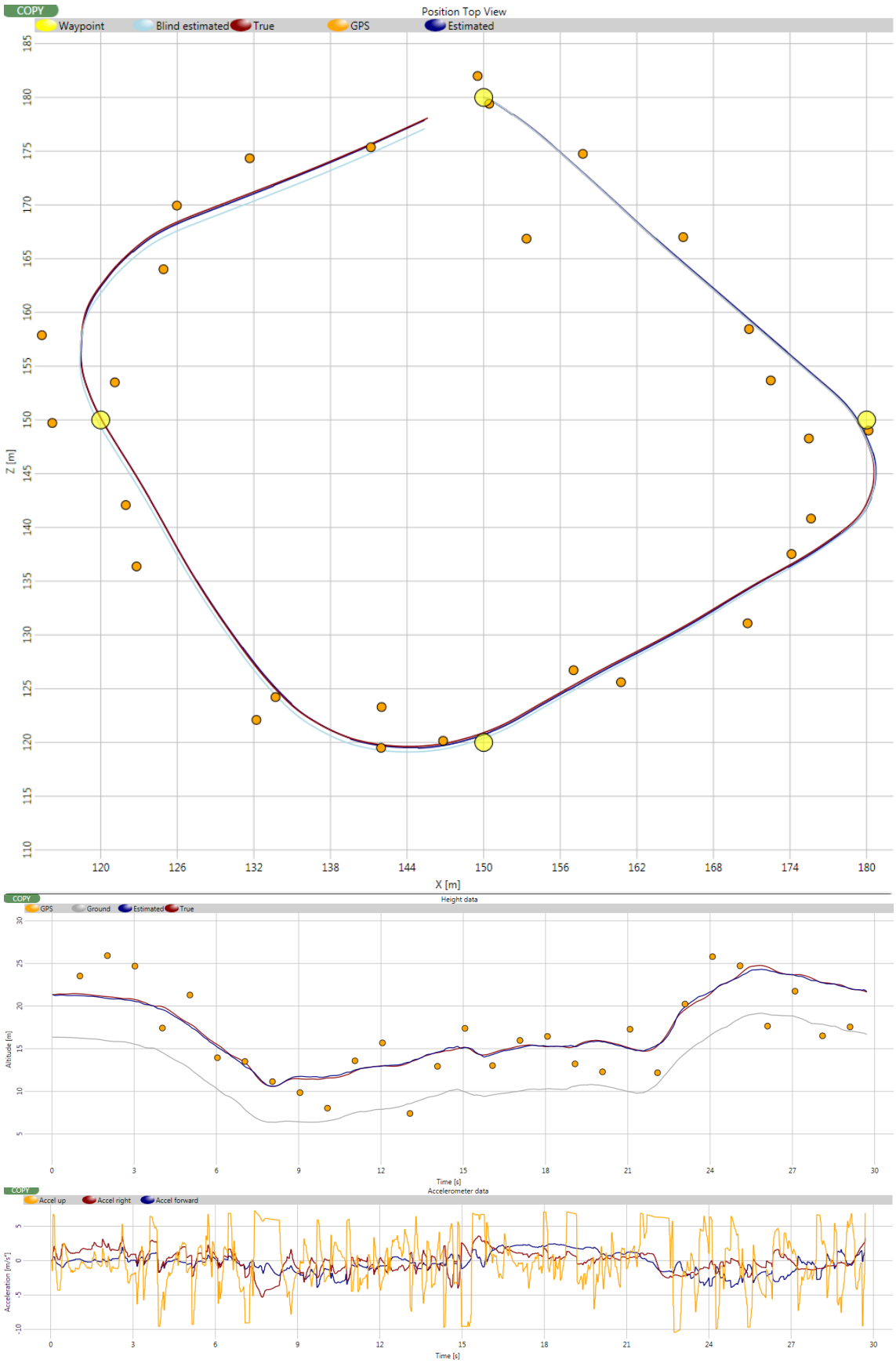


Figure 6.3.: Outline of scenario by flight log from configuration #3 in scenario #2.

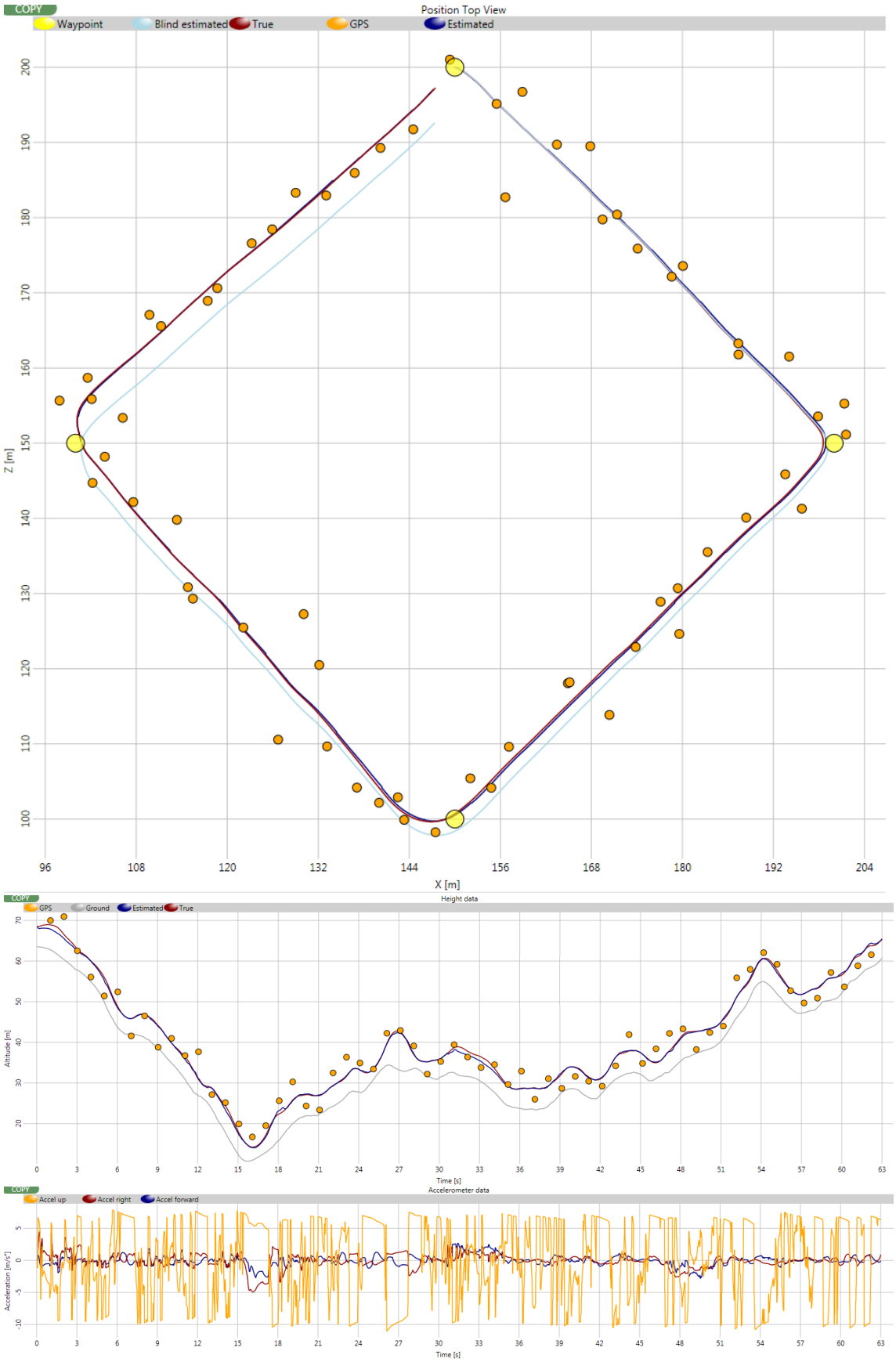


Figure 6.4.: Outline of scenario by flight log from configuration #3 in scenario #3.

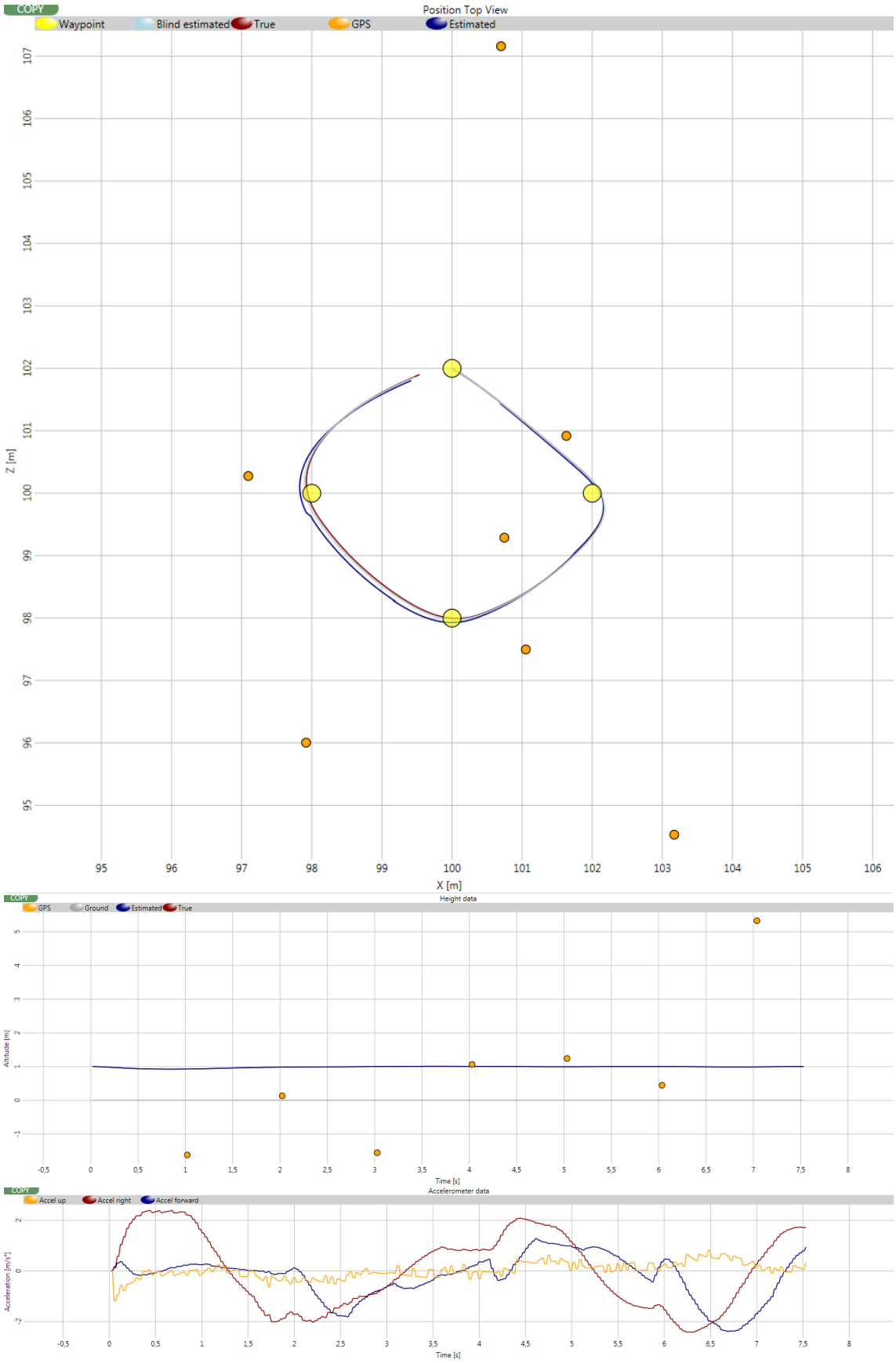


Figure 6.5.: Outline of scenario by flight log from configuration #3 in scenario #4.

6.5.1. Results of Scenario 1

Configuration	#1	#2	#3	#4
Passed	Yes	Yes	Yes	Yes
Duration	4.7 s	4.7 s	4.5 s	4.9 s
Attempts	1	1	1	1
Final $v_{h,max}$	36 km/h	36 km/h	36 km/h	36 km/h
Max velocity	23 km/h	23 km/h	24 km/h	24 km/h
Avg. velocity	13 km/h	13 km/h	14 km/h	13 km/h
Max HAG	5.00 m	5.00 m	5.00 m	5.01 m
Avg. HAG	4.98 m	4.98 m	4.98 m	4.98 m
Min HAG	4.93 m	4.93 m	4.92 m	4.93 m
Max pos. est. err.	0.00 m	0.00 m	0.22 m	1.35 m
Avg. pos. est. err.	0.00 m	0.00 m	0.1 m	0.53 m

Table 6.5.: Results of experiments on test scenario 1.

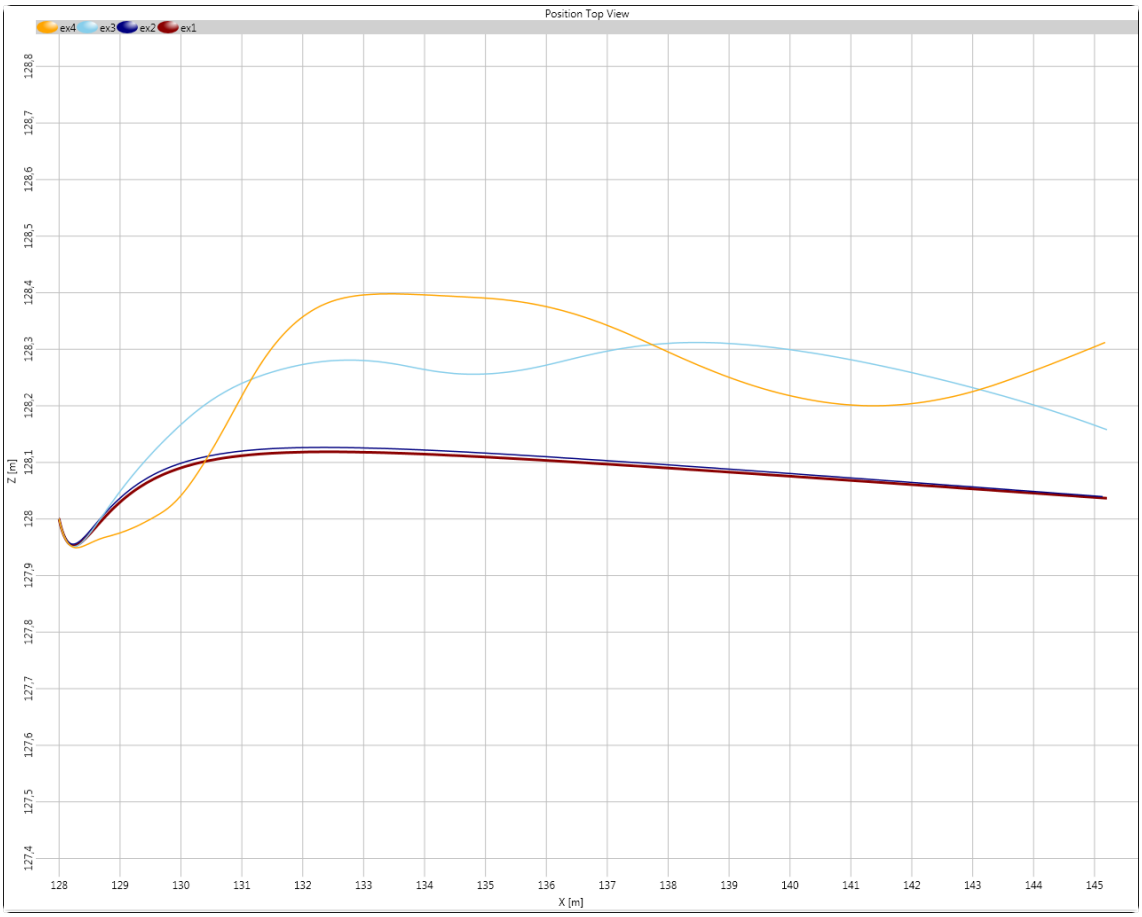


Figure 6.6.: Flight logs from experiments on scenario 1.

6.5.2. Results of Scenario 2

Configuration	#1	#2	#3	#4
Passed	Yes	Yes	Yes	Yes
Duration	29.8 s	29.8 s	29.7 s	40.8 s
Attempts	1	1	1	1
Final $v_{h,max}$	36 km/h	36 km/h	36 km/h	36 km/h
Max velocity	32 km/h	33 km/h	35 km/h	36 km/h
Avg. velocity	22 km/h	22 km/h	22 km/h	19 km/h
Max HAG	5.27 m	5.29 m	5.64 m	5.95 m
Avg. HAG	5.00 m	5.00 m	5.04 m	5.02 m
Min HAG	4.70 m	4.69 m	4.18 m	3.71 m
Max pos. est. err.	0.00 m	0.00 m	0.55 m	6.83 m
Avg. pos. est. err.	0.00 m	0.00 m	0.3 m	3.62 m

Table 6.6.: Results of experiments on test scenario 2.

Figure 6.7.: Flight logs from experiments on scenario 2.

6.5.3. Results of Scenario 3

Configuration	#1	#2	#3	#4
Passed	Yes	Yes	Yes	Yes
Duration	63.3 s	63.3 s	63.0 s	81.5 s
Attempts	2	2	2	2
Final $v_{h,max}$	18 km/h	18 km/h	18 km/h	18 km/h
Max velocity	33 km/h	33 km/h	35 km/h	38 km/h
Avg. velocity	19 km/h	19 km/h	20 km/h	17 km/h
Max HAG	6.17 m	6.24 m	9.53 m	8.84 m
Avg. HAG	5.00 m	5.00 m	5.15 m	5.07 m
Min HAG	3.67 m	3.61 m	2.40 m	0.90 m
Max pos. est. err.	0.00 m	0.00 m	1.48 m	9.65 m
Avg. pos. est. err.	0.00 m	0.00 m	0.49 m	3.35 m

Table 6.7.: Results of experiments on test scenario 3.

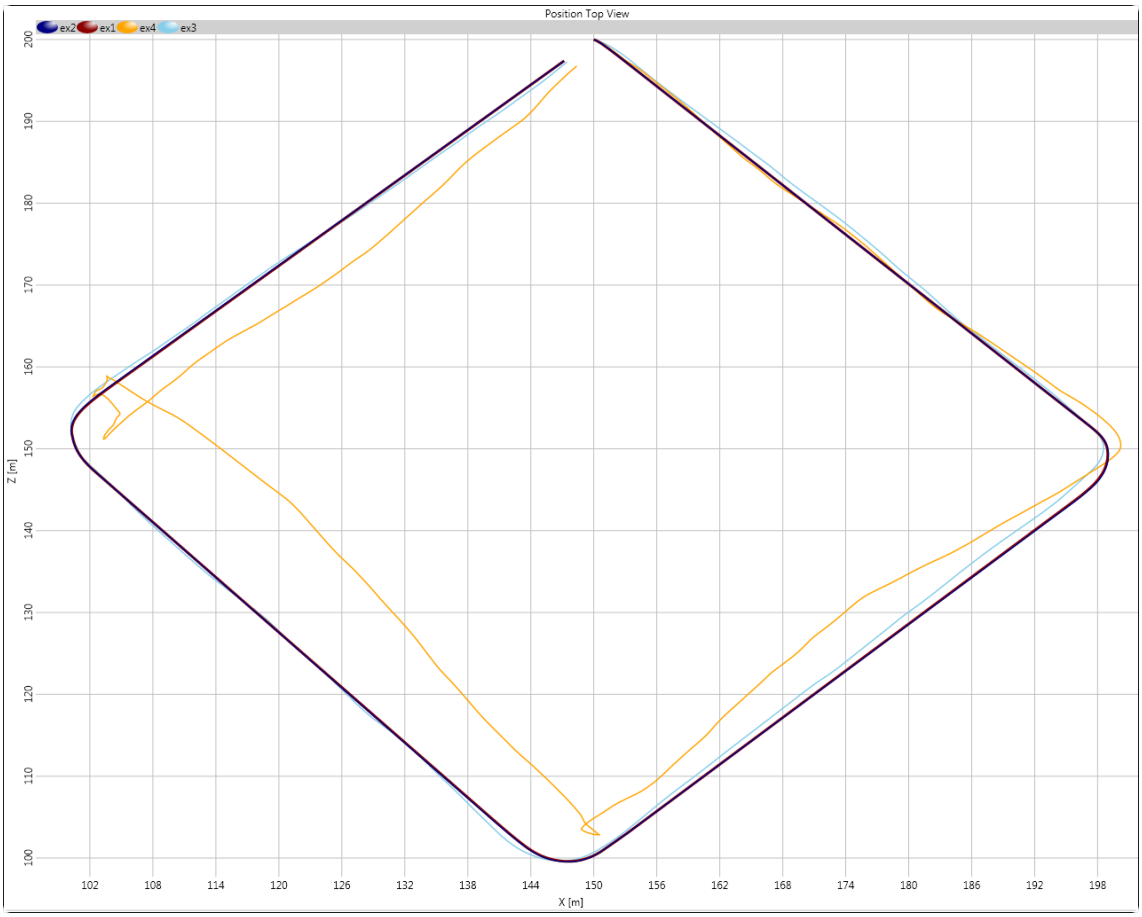


Figure 6.8.: Flight logs from experiments on scenario 3.

6.5.4. Results of Scenario 4

Configuration	#1	#2	#3	#4
Passed	Yes	Yes	Yes	No
Duration	7.8 s	7.8 s	7.5 s	Timed out
Attempts	1	1	1	1
Final $v_{h,max}$	36 km/h	36 km/h	36 km/h	36 km/h
Max velocity	7 km/h	7 km/h	8 km/h	9 km/h
Avg. velocity	5 km/h	5 km/h	6 km/h	4 km/h
Max HAG	1.01 m	1.01 m	1.01 m	1.01 m
Avg. HAG	0.99 m	0.99 m	0.99 m	1.00 m
Min HAG	0.93 m	0.93 m	0.92 m	0.92 m
Max pos. est. err.	0.00 m	0.00 m	0.19 m	8.31 m
Avg. pos. est. err.	0.00 m	0.00 m	0.08 m	3.88 m

Table 6.8.: Results of experiments on test scenario 4.

Figure 6.9.: Flight logs from experiments on scenario 4.

7. Discussion

The proof-of-concept implementation employs methods from a number of disciplines and here we discuss the methods and choices made for each aspect of the software. We will use the test results from autonomous flight experiments to verify that we reached the goals we set out to achieve and we will discuss whether the results achieved here are realistic and applicable for real navigation scenarios.

7.1. Software Architecture

The simulator was intended for future extensions so we decided it was important to design a proper software architecture and to document it. We put significant effort into promoting modifiability, performance and portability throughout the code and we consider 5 out of 6 requirements to be met. The physics and virtual sensor components can easily be extended or replaced with more accurate models later and the autopilot logic is portable from the simulator environment to a microcontroller solution with real sensors. Requirement R5 was only partially met, because it proved difficult to maintain real-time simulation with high-speed sensors running as fast as 100 Hz. We did, however, achieve different sensor sampling rates by downsampling the IMU and GPS measurements.

The focus was to develop a working prototype to investigate how the state estimation error was affected by different configurations of sensor noise and frequency. Since we were not going to port our autopilot to a microcontroller as a part of this thesis we did take some shortcuts. First, the GPS/INS Kalman filter uses a large math library to deal with matrix operations and this dependency is not directly portable to a microcontroller. Second, we did not take restrictions of a microcontroller solution into account when designing the state estimator and autopilot components. Issues such as state estimation performance and sampling timing were left out of the scope of this implementation.

An interesting side-effect of the architectural design was that it also promoted reusability of certain components. The OutputController class has few simulation dependencies and the control logic contained within it could be viable for reuse in the control of helicopters in games. With properly tuned PID settings the AI would only need to provide navigational commands and the current world state of the game in order for the helicopter to move according to the AI planning.

7.2. Visualization

We implemented a wide range of visualization techniques to add to the value of the implementation as both an autopilot development tool and a flight simulation application. The goal was to create believable and useful graphics that the user could relate to as an approximation to real flight in outdoor environments. In that aspect we feel we succeeded.

The visualization focused on representing the current position in the virtual world and to give the user a sense of scale and motion to make manual flight easier and to better observe the autopilot flight trajectories. The outdoor environment was accomplished by utilizing open source XNA components for terrains, trees and skydome rendering. We saved a lot of effort in using the XNA game framework here, which is designed to easily reuse XNA components and content created by others. Manual flight became a lot easier and much more fun after implementing the cockpit camera. The user could now fly as if sitting in a real cockpit and looking out the windows. In addition the head-tracking and stereoscopy rendering was both practical and an entertaining experience that is not available in PC games today.

Although the visualization is practical and intuitive it does lack in realism. The components we reused for outdoor rendering were designed for games and were fast, but simple and not very realistic. The trees alone were useful to give a sense of scale to the world, but the billboarding technique looked unnatural up close and we did not generate forests or formations of trees as one expects to find in the nature. Without growths and grass the terrain looks flat and artificial and it becomes more difficult to determine our velocity and the distance to certain objects when flying manually. We did reuse an open source sunlight shader and modified it to apply it to our terrain and world objects as if they were lit by the sun. Far away objects would then fade into the horizon color and this was both appealing and increased the realism, but as a serious game we would

like to see more realistic skydomes with moving clouds, improved lighting and objects casting shadows to raise the visual standard to that of modern games.

7.3. Physics Simulation

Due to time limitations we had to settle for a very simple flight dynamics model. Even so, we were satisfied with the overall feel of flying the helicopter by joystick and based on our own experience in flying model helicopters we considered it a good approximation. We will discuss the most prominent shortcomings here and suggest future enhancements.

One simplification we made was to omit torque and angular momentum. Instead we let angular velocity of the main rotor and the helicopter body be a direct function of joystick output. We justified this by the small scale and mass of model helicopters, but it is evident that incorporating moment would allow us to better model aerodynamic phenomena such as how the streamlined fuselage will rotate to align with the relative air flow from winds and high velocity flight. Modelling moment of inertia would allow us to more realistically simulate the way it takes time for an engine to accelerate and decelerate the main rotor to a certain rotor revolutions per minute. As we discovered when adding the Jitter physics engine to handle collisions it was hard to integrate with due to the way we simplified our angular motion, so to better support plug-in physics we need to model this properly. Finally, small model helicopters are very sensitive to gusts of winds and turbulence and it requires tight control to maintain safe and stable flight. It would be very interesting to challenge the autopilot with such conditions so that is another incentive to improve the physics simulation.

Although our physics simulation was very simple, it did behave much in the same manner as we expected it to. We consider the main limitations of our implementation as being too stable compared to our experience with real flight and that we did not compare the error in our flight behavior to videos of real helicopter flight where the joystick outputs were known.

We do not consider the current physics simulation to be sufficiently realistic, but adding a more accurate modelling of angular motion, inertia, winds and turbulence and comparing that flight behavior to real flights should go a long way on approaching a solution for real autopilot development.

7.4. State Estimation

The state estimation method uses a Kalman filter to fuse measurements from the IMU and GPS sensors. This method is widely used for autonomous navigation applications and it can be proven that the filter provides an optimal estimate given certain assumptions of the sensors and the problem. The test results show that our implementation of the filter performed well and a lot better than trusting GPS or INS estimates alone. However, the Kalman filter is only suitable for estimating linear processes and this means we could not easily estimate orientation from IMU and digital compass measurements as we would like to. There was not enough time to implement this so we simplified the uncertainty of orientation. Having orientational uncertainty was necessary, because even just a few degrees of error in pitch or roll would quickly accumulate to large errors in position. Without this, the state estimation would become artificially accurate.

Another simplification we made was to model the range finder sensor without noise so that the height above the ground estimate in configurations 3 and 4 would be limited to the inaccuracy of the sensor and the Flat Ground Height method proposed in section 5.5. This was fair, because the datasheet did not specify noise for the range finder and since the FGH method was very inaccurate to start with. This way we avoided crashing so much since we were flying just a few meters from the ground in our test scenarios. Also, since the error was mainly horizontal we could compare our results to related work on GPS/INS filtering for land vehicles.

From the test results in configuration 1 we see that there was zero deviation in the state estimation when the sensor output was perfect and not subject to precision loss. All four scenarios verified this and we conclude that the estimation implementation is valid. As we noted in section 5.6, this was only true as long as we did not use the Jitter physics engine to handle collisions. Unfortunately, this also meant that we were not able to create scenarios that involved take-off and landing, because this would introduce significant estimation errors. In a future application this issue needs to be resolved for simulating realistic scenarios where helicopters start and stop on the ground.

7.5. Autopilot Logic

In our implementation we proposed three different methods of maneuvering the helicopter from A to B. We could not find any academic material to base those methods

on so the development was largely a process of trial and error. The final method, velocity controlled cyclic navigation, performed very well and the test results showed that it successfully completed a number of combinations of navigation scenarios and sensor configurations. It was designed to minimize overshoot by decelerating towards B and to minimize unwanted velocity components in order to hit the target dead on and avoid circular motion near the target position. Based on the simulation results we believe that this method is sound and stable for navigation with uncertain states and should be a good candidate for real autonomous flight experiments in the future.

An interesting possibility with a working autopilot is the concept of autopilot assisted control. Since helicopters are very unstable they require a lot of experience to fly safely. The autopilot could then enable persons with minimal training to maneuver the helicopter by commanding it to move forwards, sideways and to increase or decrease the height above the ground. The autopilot will then deal with all the effort required to maneuver the helicopter, hold it steady in windy conditions and avoid crashing. This should prove very useful for scenarios such as search operations where swarms of autonomous helicopters are sent out in the wild to look for missing people. The helicopters send live pictures back to a ground station monitored by human operators who can assume manual control over a helicopter to further investigate certain areas. We tested out our implementation of assisted control on people with no prior flying experience. In manual flight, most people would crash many times before getting to grips with the controls and the technique. With assisted control, they successfully flew the helicopter at high velocities and low altitudes over curved terrains without crashing.

7.6. Flight Experiments

The experiments were designed to challenge both the autopilot control logic and the state estimation. When we designed the experiments and test scenarios we had certain expectations to what the results would look like. An autopilot with perfect knowledge should easily complete the navigation without crashing, while causing problems for autopilots with less accurate measurements. We then set out to investigate how lesser accurate knowledge performed and whether the specifications of cheap off-the-shelf sensors were sufficient for outdoor navigation.

We expected all four experiments to pass scenario 1 since it was very short and simple. This scenario was added to distinguish between configurations that were able to fly

and those that did not fly at all. Section 6.5 lists all the test results and we see how all four experiments easily passed this scenario. Even the worst configured experiment performed satisfactory with a max estimation error of 1.35 m. This was natural since there is not enough time for the INS error to accumulate significantly. It should be noted that the altitude error is artificially accurate for all configurations, since we decided to ignore noise in the range finder implementation. The datasheet had no specifications on noise and this way we could concentrate on state estimation errors in the horizontal plane.

The second scenario was the most realistic navigation scenario. It featured curved terrains and the waypoints stretched for 200 m. We were uncertain whether our sensor based configurations would be able to pass this scenario since a lot of error will accumulate in the INS over 30 seconds of navigation. This meant that the GPS/INS filter had to use the GPS observations more actively to prevent the INS position and velocity estimates from diverging significantly. The results show that the GPS/INS filter does indeed work as intended and the error does not diverge out of control. The datasheet sensors were only off by half a meter, while the unreliable sensors reached almost 7 meters of position inaccuracy. It can be seen from the flight log that the fourth experiment suffered so much from uncertainty that it missed the third waypoint and had to return back for it.

The third scenario featured extremely hilly terrain and we did not expect any sensor based configurations to complete it. As it turned out the scenario may have been overly difficult as even with perfect knowledge the autopilot did not manage to avoid crashing on the first attempt. The second attempt reduced the max horizontal speed from 36 to 18 km/h and this time experiments 1 and 2 completed without problem. To our surprise experiments 3 and 4 did so too. It seemed that the accurate range finder was working a little too well since it prevented all configurations from crashing even when their state was uncertain. On the other side, we see that both sensor based configurations were much more unstable in maintaining a height above the ground of 5 meters due to the error of the flat ground height method. Their HAG varied 7.13 and 7.94 meters respectively and scenario 4 was only 90 cm shy of crashing into the ground, while scenarios 1 and 2 had variations of only 2.5 and 2.63 meters respectively despite moving as fast as 33 km/h across hilly terrains. Once more we see that scenario 4, due to its uncertainty of up to 9.65 m, had trouble hitting waypoints 2 and 3 and has to turn around to get within the radius of 5 m.

The fourth scenario featured precision navigation and required the helicopter to pass

within 0.5 meters of the waypoints. We did not expect the sensor based configurations to successfully navigate all four waypoints, but the test results show that configuration 3 with datasheet sensors did pass this test. This might be due to the navigation spanning only 8 seconds, but then again in scenario 3 the datasheet configuration had an average accuracy of just 0.49 meters over 63 seconds of navigation, which indicated its uncertainty would not grow too large to pass scenario 4. Configuration 4 failed as expected and timed out after 100 seconds. The flight log illustrates the chaotic flight pattern as it struggles to approach waypoint 3 when the error in the INS has accumulated so much that the GPS is not able to correct it sufficiently to get within the waypoint radius.

Overall we see that navigation by state estimation works very well, but seeing the results from using datasheet specifications in configuration 3 we we suspect that the state estimation is performing unrealistically well. Comparing with the results of [10] we see that they achieved an average of 0.81 m for a 300 second horizontal navigation scenario using similarly specified GPS and IMU sensors, while our test results showed an average with 0.49 m over 63 seconds where the contribution from vertical error was inherently small. We do not consider an improvement of 40% over related work to be realistic and this supports our suspicions that the state estimation may be performing too well. A few explanations are that the sensor specifications may be optimistic. In our own experience with GPS we have yet to see that 50% of the measurements fall within 3.3 meters. Also, as we explained in section 5.5, the IMU sensors sampled each simulation time step so the measurements would not suffer from undersampling, which is the case for real sensors trying to sample real world processes. Finally the sensors are modelled by applying Gaussian noise, whereas real sensors will often have a bias and non-Gaussian noise. We tried to compensate for this by adding more noise and undersampling the IMU sensor in configuration 4 to get more realistic levels of uncertainty and as the test results show it was the only configuration that did not pass test scenario 4. Seeing the test results in light of related work, we do not consider the current physics and sensor modelling sufficiently realistic to evaluate autopilot logic for real navigation scenarios.

8. Conclusion

We set out to develop a working autopilot for small model helicopters and to design a simulator software to verify its correctness. The implementation spans several academic disciplines and required us to do a thorough research on regulatory systems, sensor specifications, state estimation and physics simulation. The thesis continued the work from our pre-study on methods of flight simulation, which provided some insight into aerodynamics and visualization techniques. Combined with our background in computer science this enabled us to build a simulator software to verify the correctness of the autopilot logic.

We proposed three methods for helicopter autopilot logic and the method of velocity controlled cyclic navigation proved stable and is considered a good candidate for future extensions of this work. We also proposed a method for autopilot assisted control that successfully enabled any person with little or no flying experience to safely maneuver the helicopter by joystick. This should prove very useful in the outlined search-and-rescue scenario, where swarms of autonomous helicopters aid in the search and send live pictures back to a station monitored by human operators. If an operator spots something of interest then he or she can assume manual control over a helicopter to further investigate certain areas.

Due to the scope of the project we had to make some simplifications and this resulted in state estimates that were 40% more accurate than the results of related work. The virtual sensors are not modelled to incorporate issues such as environmental influences, undersampling, biased measurements and non-Gaussian noise that would increase the uncertainty. The GPS/INS Kalman filter only supports linear estimation so were not able to model orientation by IMU measurements. Instead we inserted random errors in the orientation to compensate for the missing uncertainty contribution. In addition the realism of the physics model suffered from using a simple flight dynamics model with empirically chosen coefficients for drag and lift.

The final implementation was a proof-of-concept that, although with major simplifications in both physics and sensor modelling, the test results clearly indicate that the autopilot is capable of controlled flight and the Kalman filter improves the state estimation significantly compared to relying on individual sensor measurements. We did measure the autopilot performance when modelling the sensors by datasheets, but due to the simplifications we do not consider the results sufficiently realistic to evaluate whether the autopilot could function in a real navigation scenario or not. However, the results prove that the autopilot works on a conceptual basis. With future improvements to physics and sensor modelling we believe the simulator could be used to develop autopilots for real autonomous navigation. Also, the autopilot component should prove viable for reuse in physics oriented games to enable realistic maneuvering of AI-controlled aircrafts.

9. Future Work

There were a number of shortcomings in the implementation that we would like to see improved in future extensions. The virtual sensors need to model noise and measurements in a realistic manner by accounting for external influences, undersampling of truth, precision, inaccuracy, biasing and non-Gaussian noise. For example, in real world applications the accelerometers would suffer greatly from vibration in the fuselage during flight and introduce a lot more uncertainty in the INS estimate.

The physics model should model angular velocity from the torque generated by cyclic controls, rotor velocity, rotor inertia, winds, turbulence and other advanced aerodynamic phenomena. For example, model helicopters are known to become unstable when flying near the ground due to turbulent flow of the main rotor downwash being recycled back into the inflow. Also, the lift and coefficients should depend on the relative airflow angle and velocity. A future extension to the physics should also be compared with real flight experiments to further refine the realism.

We would like to see different configurations of helicopter flight dynamics to challenge the autopilot with different flight behavior. It should be possible to automatically calibrate the autopilot to each configuration by performing specific maneuvers and measuring how joystick outputs over time causes change in position, orientation and their derivatives. When properly calibrated the autopilot should be able to navigate blindly for short periods of time, such as when the GPS loses reception or any sensors become faulty. This could then be used to perform emergency landing procedures.

The autopilot component should be ported to a microcontroller for testing with real sensors. Only then can one with confidence measure what levels of uncertainty to expect during real flights. These results will then be used to further enhance the realism of the modelled sensors and the autopilot simulation.

Bibliography

[1] J. Watkinson, Art of the Helicopter, Butterworth-Heinemann, 2004.

[2] Wikipedia, Drag (physics), http://en.wikipedia.org/wiki/Drag_(physics), visited 29.10.2009.

[3] autopilot: Do it yourself uav, http://autopilot.sourceforge.net, 20.10.2009.

[4] S. Rï¿œnnbï¿œck, Development of a ins/gps navigation loop for an uav, Master's thesis, Luleï¿œ Tekniska Universitet (2000).

[5] Introduction to direct digital controls, http://www.ddc-online.org/intro/intro_chapt01.aspx, visited 23.09.2009.

[6] C. Schmid, Course on dynamics of multidisplicinary and controlled systems, http://www.atp.ruhr-uni-bochum.de/rt1/syscontrol/node1.html, visited 12.10.2009.

[7] G. Welch, G. Bishop, An introduction to the kalman filter, http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf, visited 20.10.2009.

[8] H. U. of Technology, TKK / LCE models and methods: 2D CWPA-model, http://www.lce.hut.fi/research/mm/ekfukf/cwpa_demo.shtml, visited 11.10.2009.

[9] A. Larsen, Methods of real-time flight simulation (December 2009).

[10] F. Caron, E. Duflos, D. Pomorski, P. Vanheeghe, Gps/imu data fusion using multisensor kalman filtering: introduction of contextual aspects, Inf. Fusion 7 (2) (2006) 221–230. doi:http://dx.doi.org/10.1016/j.inffus.2004.07.002.

[11] J. Wang, M. Garratt, A. Lambert, J. J. Wang, S. Hana, D. Sinclair, Integration of gps/ins/vision sensors to navigate unmanned aerial vehicles, in: ISPRS Congress

Beijing 2008. Vol. XXXVII. Part B1. Commission I., The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Beijing, 2008, pp. 963–970.

[12] G. Beinset, J. S. Blomhoff, Controller design for an unmanned surface vessel, Master's thesis, NTNU (2007).

[13] J. A. Farrell, M. Barth, The Global Positioning System & Inertial Navigation, McGraw-Hill Professional, 1998.

[14] F. Dunn, I. Parberry, 3D Math Primer for Graphics and Game Development (Wordware Game Math Library), Jones & Bartlett Publishers, 2002.

[15] G. G. Slabaugh, Computing euler angles from a rotation matrix, `http://www.gregslabaugh.name/publications/euler.pdf`, visited 16.04.2010 (August 1999).

[16] B. Copeland, The design of pid controllers using ziegler nichols tuning, `http://www.eng.uwi.tt/depts/elec/staff/copeland/ee27B/Ziegler_Nichols.pdf`, visited 16.09.2009 (March 2008).

[17] Q.-C. Zhong, Robust Control of Time-delay Systems, Springer, 2006.

[18] A. H. Jazwinski, Stochastic Processes and Filtering Theory, Dover Publications, 2007.

[19] VRPN, `http://www.cs.unc.edu/Research/vrpn/`, visited 19.03.2010.

[20] T. Lozano-Pérez, Spatial planning: a configuration space approach (1990) 259–271.

[21] A. Elfes, Sonar-based real-world mapping and navigation (1990) 233–249.

[22] S. A. Shafer, A. Stentz, C. E. Thorpe, An architecture for sensor fusion in a mobile robot, Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco, CA, 1986, pp. 2002–2011.

[23] A. U. Alvarez, Componente skydome, `http://xnacommunity.codeplex.com/wikipage?title=Componente%20SkyDome`, visited 18.04.2010.

[24] Nerdy inverse focus on terrain, `http://www.ziggyware.com`, visited 14.04.2010 (page no longer available).

[25] XNA procedural ltrees, `http://ltrees.codeplex.com/Wikipage`, visited 16.04.2010.

[26] NTNU Visualization Lab Wiki, `http://mediawiki.idi.ntnu.no/wiki/vis/index.php/Development:Development`, visited 19.03.2010.

[27] XNA chase camera sample, `http://creators.xna.com/en-us/sample/chasecamera`, visited 19.03.2010.

[28] B. Atkinson, Dynamical Meteorology: An Introductory Selection (1st Edition), Routledge, 1981.

[29] G. Palmer, Physics for Game Programmers, Apress, 2005.

[30] J. Farrell, Aided Navigation: GPS with High Rate Sensors, McGraw-Hill Professional, 2008.

[31] B. Premerlani, User manual for uav v2 development platform, `http://www.sparkfun.com/datasheets/GPS/GPSUAV2Manual_v29.pdf`, visited 28.04.2010 (2009).

[32] J. Marins, X. Yun, E. Bachmann, R. B. McGhee, M. J. Zyda, An extended kalman filter for quaternions-based orientation using marg sensors, Vol. 4 of Proceedings of the IEEE International Conference on Intelligent Robots and Systems, Maui, Hawaii, 2001, pp. 2003–2011 Vol. 4.

[33] P. Zhang, J. Gu, E. E. Milios, P. Huynh, Navigation with imu/gps/digital compass with unscented kalman filter, Vol. 3 of Proceedings of the IEEE International Conference on Mechatronics and Automation, 2005, pp. 1497–1502 Vol. 3.

A. User Manual

A.1. Running the Flight Demo

Included in the digital content is both source code and pre-compiled binary files. To run the flight demo simply run the shortcut located at */bin/Run Demo*. To enable head tracking of the HMD for cockpit camera mode one must run the local VRPN server prior to running the simulator, located at *C:\vrpn\vrpn\pc_win32\server_src\vrpn_server\Release\vrpn_server* on the test computer. All configurations for the demo are found in the files as specified in appendix B.

A.1.1. Keyboard and Mouse Shortcuts

Listed are keyboard and mouse shortcuts that can be accessed when running the simulator.

Key / Mouse	Function
1	Chase camera
2	Cockpit camera
3	Fixed camera
4	Free camera
Space	Reset / Next test scenario
L	Open live flight logger
PrntScrn	Screenshot
Up/Down	Change time of day
Right-Click	Open PID configuration

A.1.2. Reproducing the Test Results

To run the tests simply execute the batch file located at */bin/Run Tests.bat*. This will run all four test configurations in sequence and save the results to */bin/Test Results/*. Note that the results may differ slightly due to random processes in the simulation. The output files are structured as follows.

test_results.txt Statistics for all the test scenarios run by that specific test configuration.

TestConfiguration.xml Configuration of sensors, autopilot and test scenarios.

flightlog_<scenario>.xml The flight log data.

To visualize a flight log simply run the review tool located at */bin/Flight Log Review Tool/Review.exe* and drag and drop the flight log .xml file to the window. Comparing flight logs for a specific test scenario is more difficult. There must be exactly four .xml files named "ex1.xml" to "ex4.xml" corresponding to the flight logs for configurations 1-4 on the same test scenario. Drag these four files simultaneously to the window for comparison. See *docs/Masterprosjekt/test results/* for an example on how we organized our files to easily compare them.

Note that test configurations 1 and 2 requires changes in program code to reproduce the results. Configuration 1 needs changes in code to use world state instead of estimated state. Configuration 2 requires the autopilot to navigate by INS estimation only and we did not make this a configurable option. Configurations 3 and 4, however, are reproducible by using the batch file system.

B. Configuration Files

To let the user change configurations without having to re-compile the program code we created some configuration files. We will briefly describe the function and syntax of the configuration files here.

B.1. TestConfiguration.xml

An example test configuration is listed in table B.1. The first part defines standard deviations for the Gaussian modelled noise of accelerometer and GPS measurements. OrientationAngleNoiseStdDev defines the distribution of noise for each yaw, pitch and roll angles to compensate for not using gyroscopes to estimate orientation.

The list of MaxHVelocity elements tells the autopilot what max horizontal speed (in m/s) to use when navigating. If the helicopter crashes in a test scenario it will retry with velocity settings in sequence until it either passes or fails for every setting. The list of ScenarioName describes the scenarios that the autopilot will run and their definitions are listed in the Scenarios.xml file.

```
<root>
  <TestConfiguration>
    <SensorSpecifications>
      <Accelerometer>
       <Frequency>60</Frequency>
       <NoiseStdDev>
         <Forward>7.08E-3</Forward>
         <Right>7.08E-3</Right>
         <Up>8.85E-3</Up>
       </NoiseStdDev>
      </Accelerometer>

      <GPSPositionAxisStdDev>3.3</GPSPositionAxisStdDev>
      <GPSVelocityAxisStdDev>0.05</GPSVelocityAxisStdDev>
      <OrientationAngleNoiseStdDev>2</OrientationAngleNoiseStdDev>
    </SensorSpecifications>

    <MaxHVelocity>10</MaxHVelocity>
    <MaxHVelocity>5</MaxHVelocity>
    <MaxHVelocity>2</MaxHVelocity>
    <MaxHVelocity>1</MaxHVelocity>

    <ScenarioName>A-B Short Flat</ScenarioName>
    <ScenarioName>Circle Large Hilly</ScenarioName>
    <ScenarioName>Circle Medium Sloped</ScenarioName>
    <ScenarioName>Circle Precision Short Flat</ScenarioName>
  </TestConfiguration>
</root>
```

Table B.1.: TestConfiguration.xml example.

B.2. Scenarios.xml

This file holds the definitions of all the scenarios that can be run. We used these to define manual flight scenarios as shown in the first example and autopilot test scenarios as shown in the latter example. Table B.2 lists the possible elements and their values. Examples shown in tables B.3 and B.4.

B.3. PIDSetups.xml

We tuned our PID settings by the methods described in section 4.3.3 and the final configuration is shown in table B.5. Note that the PID coefficients are inverted in the configuration because this was easier to relate to. Actual coefficients used in calculations are found by $K_P = \frac{1}{P}$ and similar.

XML Element	Description / Possible Values
PreSelectedScenario	Name of scenario to load on start-up (not test mode).
SwapStereo	Swap left and right eyes in stereo mode.
RenderMode	Rendering technique. *Normal, StereoCrossConverged, Stereo*
Scenario	
CameraType	*Chase, Fixed, Free, Cockpit*
Scene	List of world objects to load. *Terrain, Skydome, Forest, Ground, Barrels, CurrentWaypoint.*
Helicopter	
EngineSound	Toggle audio. *true, false*
PlayerControlled	Toggle autopilot or manual flight. *true, false*
StartPosition	Start position of helicopter in meters.
Task	
Loop	Toggle loop of waypoints. *true, false*
HoldHeightAbove..	Target altitude above ground in meters.
DefaultWaypointRadius	Max distance to pass a waypoint in meters.
Waypoint	
Type	The waypoint function in navigation. *Intermediate, Hover, TestDestination, Land*
Position	Position of waypoint in meters.
Radius	Overrides the default radius.

Table B.2.: Fields and values for Scenarios.xml.

```
<root>
  <PreSelectedScenario>Circle Medium Sloped</PreSelectedScenario>

  <Scenario Name="Terrain Flight Demo">
    <CameraType>Chase</CameraType>
    <Scene>
      <Terrain/>
      <Skydome />
    </Scene>
    <Helicopter>
      <EngineSound>false</EngineSound>
      <PlayerControlled>true</PlayerControlled>
      <StartPosition X="128" Y="70" Z="128" />
      <Task>
        <Loop>false</Loop>
        <Waypoint>
          <Type>Land</Type>
          <Position X="128" Y="70" Z="128" />
        </Waypoint>
      </Task>
    </Helicopter>
  </Scenario>
```

Table B.3.: Scenarios.xml example 1 - Manual flight.

```
<Scenario Name="Circle Medium Sloped">
    <TimeoutSeconds>100</TimeoutSeconds>
    <Scene>
      <Terrain Width="256" MinHeight="0" MaxHeight="30" />
      <Skydome />
      <CurrentWaypoint />
    </Scene>
    <Helicopter>
      <StartPosition X="150" Y="-1" Z="180" />
      <Task>
        <HoldHeightAboveGround>5</HoldHeightAboveGround>
        <WaypointRadius>5</WaypointRadius>
        <Waypoint>
          <Type>Intermediate</Type>
          <Position X="180" Y="-1" Z="150" />
        </Waypoint>
        <Waypoint>
          <Type>Intermediate</Type>
          <Position X="150" Y="-1" Z="120" />
        </Waypoint>
        <Waypoint>
          <Type>Intermediate</Type>
          <Position X="120" Y="-1" Z="150" />
        </Waypoint>
        <Waypoint>
          <Type>TestDestination</Type>
          <Position X="150" Y="-1" Z="180" />
        </Waypoint>
      </Task>
    </Helicopter>
  </Scenario>
</root>
```

Table B.4.: Scenarios.xml example 2 - Autonomous flight.

```
<root>
  <PIDSetup Name="Stable_v1">
    <PID Name="PitchAngle" P="30" I="0" D="0" />
    <PID Name="RollAngle" P="30" I="0" D="0" />
    <PID Name="YawAngle" P="30" I="0" D="0" />
    <PID Name="Throttle" P="-0.5" I="-0.5" D="-1" />
    <PID Name="Velocity" P="0" I="0" D="-1" />
  </PIDSetup>
</root>
```

Table B.5.: PIDSetups.xml example.

```
<root>
  <JoystickSetup Name="Microsoft SideWinder Precision 2">
    <JoystickDevice Name="SideWinder Precision 2 Joystick">
      <Axis Name="X" Inverted="false">Roll</Axis>
      <Axis Name="Y" Inverted="false">Pitch</Axis>
      <Axis Name="Z" Inverted="false"></Axis>
      <Axis Name="Rx" Inverted="false"></Axis>
      <Axis Name="Ry" Inverted="false"></Axis>
      <Axis Name="Rz" Inverted="false">Yaw</Axis>
      <Axis Name="U" Inverted="true">Throttle</Axis>
      <Axis Name="V" Inverted="false"></Axis>
    </JoystickDevice>
  </JoystickSetup>
</root>
```

Table B.6.: JoystickSetups.xml example - Microsoft SideWinder

B.4. JoystickSetups.xml

In our final implementation we configured two joysticks; Microsoft SideWinder Precision 2 and Logitech G940 Flight System. Both are shown in tables B.6 and B.7. Note how the Logitech uses data from three separate joystick devices. Each JoystickSetup has one or more JoystickDevices. Each device has a 8 axes that can be mapped to a function in the simulator such as *Roll, Pitch, Yaw, Throttle* and optionally invert the axis. To find what logical axis on the joystick corresponds to a physical axis use the joystick application located in the */bin/Joystick Configuration Tool/* folder.

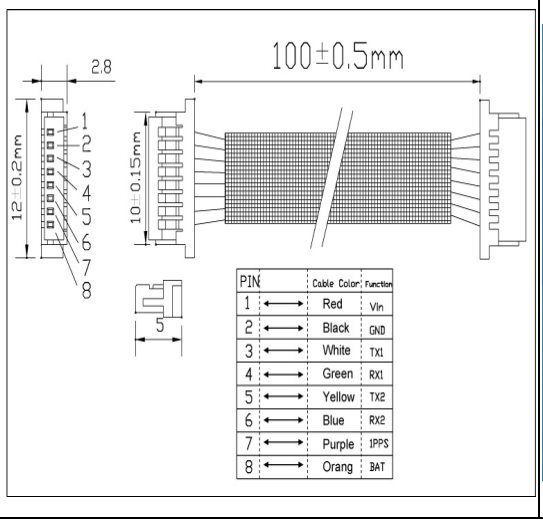
```
<root>
  <JoystickSetup Name="Logitech G940 Flight System">
    <JoystickDevice Name="Logitech G940 Joystick">
      <Axis Name="X" Inverted="false">Roll</Axis>
      <Axis Name="Y" Inverted="false">Pitch</Axis>
      <Axis Name="Z" Inverted="false"></Axis>
      <Axis Name="Rx" Inverted="false"></Axis>
      <Axis Name="Ry" Inverted="false"></Axis>
      <Axis Name="Rz" Inverted="false"></Axis>
      <Axis Name="U" Inverted="false"></Axis>
      <Axis Name="V" Inverted="false"></Axis>
    </JoystickDevice>
    <JoystickDevice Name="Logitech G940 Throttle">
      <Axis Name="X" Inverted="true">Throttle</Axis>
      <Axis Name="Y" Inverted="false"></Axis>
      <Axis Name="Z" Inverted="false"></Axis>
      <Axis Name="Rx" Inverted="false"></Axis>
      <Axis Name="Ry" Inverted="false"></Axis>
      <Axis Name="Rz" Inverted="false"></Axis>
      <Axis Name="U" Inverted="false"></Axis>
      <Axis Name="V" Inverted="false"></Axis>
    </JoystickDevice>
    <JoystickDevice Name="Logitech G940 Pedals">
      <Axis Name="X" Inverted="false"></Axis>
      <Axis Name="Y" Inverted="false"></Axis>
      <Axis Name="Z" Inverted="false"></Axis>
      <Axis Name="Rx" Inverted="false"></Axis>
      <Axis Name="Ry" Inverted="false"></Axis>
      <Axis Name="Rz" Inverted="false">Yaw</Axis>
      <Axis Name="U" Inverted="false"></Axis>
      <Axis Name="V" Inverted="false"></Axis>
    </JoystickDevice>
  </JoystickSetup>
</root>
```

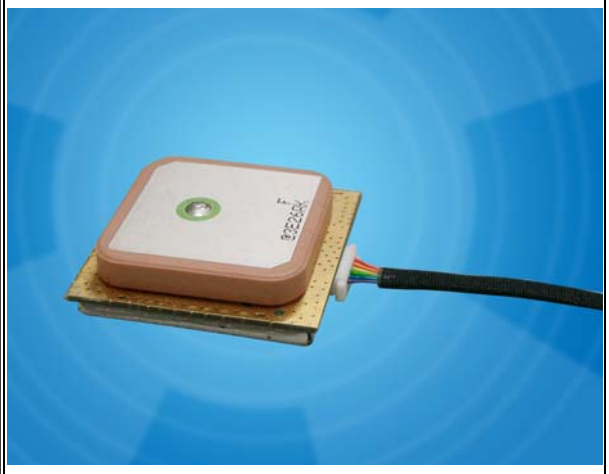
Table B.7.: JoystickSetups.xml example - Logitech G940

C. Sensor Datasheets

GPS Engine Board

Model: FV-M8





Specifications:

PHYSICAL CONSTRUCTION		PERFORMANCE		
Dimension	L30mm*W30mm*H8.6mm	Built-in Antenna	Highly-reliable ceramic patch	
Weight	15 grams	Sensitivity	-158dbm	
		SBAS	1 channel (Support WAAS, EGNOS, MSAS)	
		DGPS	RTCM Protocol	
Receiving frequency	1575.42MHZ; C/A code	Receiver architecture	32 parallel channels	
Connector	8pin connector with 1.0mm pitch	Start-up time	Hot start	1 sec. typical
			Warm start	35 sec. typical
			Cold start	41sec. typical
Mounting	Soldering	Position accuracy	Without aid	3.3 m CEP
			DGPS (RTCM)	2.6 m

San Jose Technology, Inc. | 11F., No.2, Sec. 4, Jhongyang Rd., Tucheng City, Taipei County 236, Taiwan (R.O.C.) | Tel: 886-2-22694456 | Fax: 886-2-22694451 | sanav@sanav.com

Construction	Full EMI Shielding	Velocity accuracy	0.1 Knot RMS steady state
ENVIRONMENTAL CONDITIONS		Update Rate	1 ~ 5Hz
Temperature	Operating: -30 ~ +80 ℃	Power Supply	3.3~5V +- 5%
	Storage: -40 ~ +85 ℃	Current Consumption	Acquisition: 63mA
COMMUNICATION			Tracking: 59mA (first 5 minutes)
			42mA (after 5 minutes)
Protocol	NMEA V3.01		33mA (after 20minutes)
Signal level	UART @ 2.8V * 2		
INTERFACE CAPABILITY		Baud Rate	4800 bps (default) & 4800/9600/38400/57600/11520 0 bps are adjustable
Standard Output Sentences	Default: RMC, GGA, GSV*5, VTG, GSA*5		
	Optional: GLL, ZDA		

Small, Low Power, 3-Axis ±3 *g* *i*MEMS® Accelerometer

ADXL330

FEATURES

3-axis sensing
Small, low-profile package
4 mm × 4 mm × 1.45 mm LFCSP
Low power
180 µA at V_S = 1.8 V (typical)
Single-supply operation
1.8 V to 3.6 V
10,000 *g* shock survival
Excellent temperature stability
BW adjustment with a single capacitor per axis
RoHS/WEEE lead-free compliant

APPLICATIONS

Cost-sensitive, low power, motion- and tilt-sensing applications
Mobile devices
Gaming systems
Disk drive protection
Image stabilization
Sports and health devices

GENERAL DESCRIPTION

The ADXL330 is a small, thin, low power, complete 3-axis accelerometer with signal conditioned voltage outputs, all on a single monolithic IC. The product measures acceleration with a minimum full-scale range of ±3 *g*. It can measure the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion, shock, or vibration.

The user selects the bandwidth of the accelerometer using the C_X, C_Y, and C_Z capacitors at the X_{OUT}, Y_{OUT}, and Z_{OUT} pins. Bandwidths can be selected to suit the application, with a range of 0.5 Hz to 1600 Hz for X and Y axes, and a range of 0.5 Hz to 550 Hz for the Z axis.

The ADXL330 is available in a small, low profile, 4 mm × 4 mm × 1.45 mm, 16-lead, plastic lead frame chip scale package (LFCSP_LQ).

FUNCTIONAL BLOCK DIAGRAM

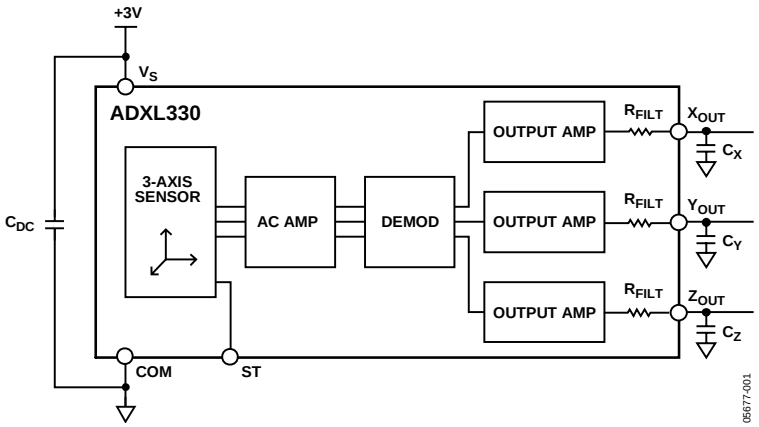


Figure 1.

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 781.329.4700 www.analog.com
Fax: 781.461.3113

SPECIFICATIONS

T_A = 25°C, V_S = 3 V, C_X = C_Y = C_Z = 0.1 µF, acceleration = 0 *g*, unless otherwise noted. All minimum and maximum specifications are guaranteed. Typical specifications are not guaranteed.

Table 1.

Parameter	Conditions	Min	Typ	Max	Unit
SENSOR INPUT	Each axis				
Measurement Range		±3	±3.6		*g*
Nonlinearity	% of full scale		±0.3		%
Package Alignment Error			±1		Degrees
Interaxis Alignment Error			±0.1		Degrees
Cross Axis Sensitivity[1]			±1		%
SENSITIVITY (RATIOMETRIC)[2]	Each axis				
Sensitivity at X_{OUT}, Y_{OUT}, Z_{OUT}	V_S = 3 V	270	300	330	mV/*g*
Sensitivity Change Due to Temperature[3]	V_S = 3 V		±0.015		%/°C
ZERO *g* BIAS LEVEL (RATIOMETRIC)	Each axis				
0 *g* Voltage at X_{OUT}, Y_{OUT}, Z_{OUT}	V_S = 3 V	1.2	1.5	1.8	V
0 *g* Offset vs. Temperature			±1		m*g*/°C
NOISE PERFORMANCE					
Noise Density X_{OUT}, Y_{OUT}			280		µ*g*/√Hz rms
Noise Density Z_{OUT}			350		µ*g*/√Hz rms
FREQUENCY RESPONSE[4]					
Bandwidth X_{OUT}, Y_{OUT}[5]	No external filter		1600		Hz
Bandwidth Z_{OUT}[5]	No external filter		550		Hz
R_{FILT} Tolerance			32 ± 15%		kΩ
Sensor Resonant Frequency			5.5		kHz
SELF TEST[6]					
Logic Input Low			+0.6		V
Logic Input High			+2.4		V
ST Actuation Current			+60		µA
Output Change at X_{OUT}	Self test 0 to 1		−150		mV
Output Change at Y_{OUT}	Self test 0 to 1		+150		mV
Output Change at Z_{OUT}	Self test 0 to 1		−60		mV
OUTPUT AMPLIFIER					
Output Swing Low	No load		0.1		V
Output Swing High	No load		2.8		V
POWER SUPPLY					
Operating Voltage Range		1.8		3.6	V
Supply Current	V_S = 3 V		320		µA
Turn-On Time[7]	No external filter		1		ms
TEMPERATURE					
Operating Temperature Range		−25		+70	°C

[1] Defined as coupling between any two axes.
[2] Sensitivity is essentially ratiometric to V_S.
[3] Defined as the output change from ambient-to-maximum temperature or ambient-to-minimum temperature.
[4] Actual frequency response controlled by user-supplied external filter capacitors (C_X, C_Y, C_Z).
[5] Bandwidth with external capacitors = $1/(2 \times \pi \times 32\ k\Omega \times C)$. For C_X, C_Y = 0.003 µF, bandwidth = 1.6 kHz. For C_Z = 0.01 µF, bandwidth = 500 Hz. For C_X, C_Y, C_Z = 10 µF, bandwidth = 0.5 Hz.
[6] Self-test response changes cubically with V_S.
[7] Turn-on time is dependent on C_X, C_Y, C_Z and is approximately 160 × C_X or C_Y or C_Z + 1 ms, where C_X, C_Y, C_Z are in µF.

ABSOLUTE MAXIMUM RATINGS

Table 2.

Parameter	Rating
Acceleration (Any Axis, Unpowered)	10,000 *g*
Acceleration (Any Axis, Powered)	10,000 *g*
V_S	−0.3 V to +7.0 V
All Other Pins	(COM − 0.3 V) to (V_S + 0.3 V)
Output Short-Circuit Duration (Any Pin to Common)	Indefinite
Temperature Range (Powered)	−55°C to +125°C
Temperature Range (Storage)	−65°C to +150°C

Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only; functional operation of the device at these or any other conditions above those indicated in the operational section of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

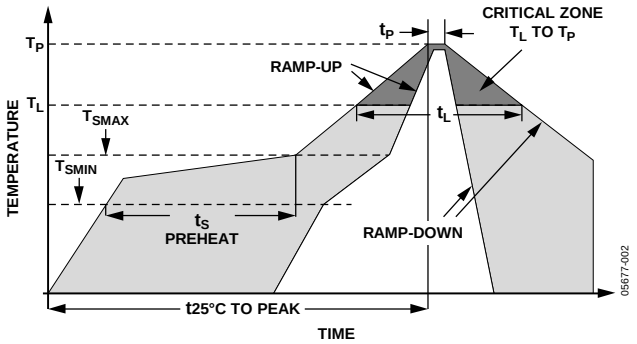


Figure 2. Recommended Soldering Profile

Table 3. Recommended Soldering Profile

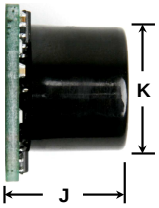
Profile Feature	Sn63/Pb37	Pb-Free
Average Ramp Rate (T_L to T_P)	3°C/s max	3°C/s max
Preheat		
Minimum Temperature (T_{SMIN})	100°C	150°C
Maximum Temperature (T_{SMAX})	150°C	200°C
Time (T_{SMIN} to T_{SMAX}), t_S	60 s to 120 s	60 s to 180 s
T_{SMAX} to T_L		
Ramp-Up Rate	3°C/s max	3°C/s max
Time Maintained Above Liquidous (T_L)		
Liquidous Temperature (T_L)	183°C	217°C
Time (t_L)	60 s to 150 s	60 s to 150 s
Peak Temperature (T_P)	240°C + 0°C/−5°C	260°C + 0°C/−5°C
Time within 5°C of Actual Peak Temperature (t_P)	10 s to 30 s	20 s to 40 s
Ramp-Down Rate	6°C/s max	6°C/s max
Time 25°C to Peak Temperature	6 minutes max	8 minutes max

ESD CAUTION

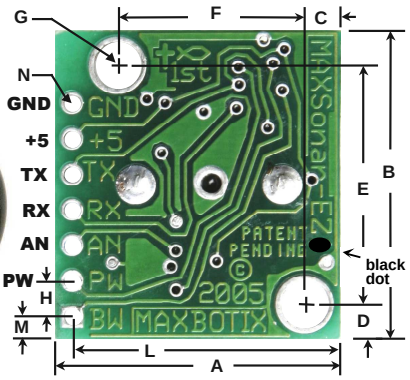
ESD (electrostatic discharge) sensitive device. Electrostatic charges as high as 4000 V readily accumulate on the human body and test equipment and can discharge without detection. Although this product features proprietary ESD protection circuitry, permanent damage may occur on devices subjected to high energy electrostatic discharges. Therefore, proper ESD precautions are recommended to avoid performance degradation or loss of functionality.

LV-MaxSonar®-EZ0™ High Performance Sonar Range Finder

With 2.5V - 5.5V power the LV-MaxSonar®-EZ0™ provides very short to long-range detection and ranging, in an incredibly small package. The LV-MaxSonar®-EZ0™ detects objects from 0-inches to 254-inches (6.45-meters) and provides sonar range information from 6-inches out to 254-inches with 1-inch resolution. Objects from 0-inches to 6-inches range as 6-inches. The interface output formats included are pulse width output, analog voltage output, and serial digital output.



approximately actual size



A	0.785"	19.9 mm
B	0.870"	22.1 mm
C	0.100"	2.54 mm
D	0.100"	2.54 mm
E	0.670"	17.0 mm
F	0.510"	12.6 mm
G	0.124" dia.	3.1 mm dia.

H	0.100"	2.54 mm
J	0.645"	16.4 mm
K	0.610"	15.5 mm
L	0.735"	18.7 mm
M	0.065"	1.7 mm
N	0.038" dia.	1.0 mm dia.
weight, 4.3 grams		

values are nominal

Features

- Continuously variable gain for beam control and side lobe suppression
- Object detection includes zero range objects
- 2.5V to 5.5V supply with 2mA typical current draw
- Readings can occur up to every 50mS, (20-Hz rate)
- Free run operation can continually measure and output range information
- Triggered operation provides the range reading as desired
- All interfaces are active simultaneously
  - Serial, 0 to Vcc
    - 9600Baud, 81N
  - Analog, (Vcc/512) / inch
  - Pulse width, (147uS/inch)
- Learns ringdown pattern when commanded to start ranging
- Designed for protected indoor environments
- Sensor operates at 42KHz
- High output square wave sensor drive (double Vcc)

Benefits

- Very low cost sonar ranger
- Reliable and stable range data
- Sensor dead zone virtually gone
- Lowest power ranger
- Quality beam characteristics
- Mounting holes provided on the circuit board
- Very low power ranger, excellent for multiple sensor or battery based systems
- Can be triggered externally or internally
- Sensor reports the range reading directly, frees up user processor
- Fast measurement cycle
- User can choose any of the three sensor outputs

Beam Characteristics

The LV-MaxSonar®-EZ0™ has the most sensitivity of the MaxSonar product line, yielding a controlled wide beam with high sensitivity. Sample results for measured beam patterns are shown below on a 12-inch grid. The detection pattern is shown for;

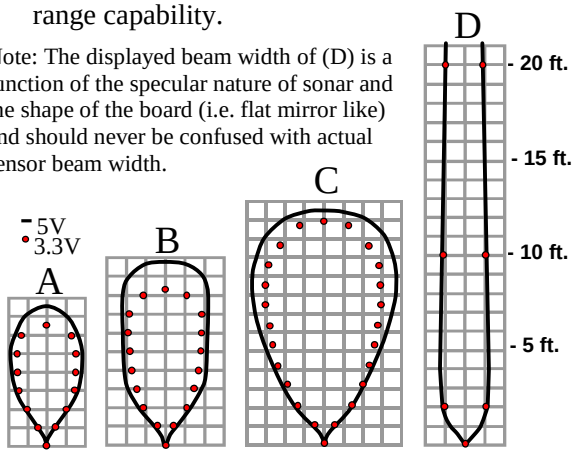
(A) 0.25-inch diameter dowel, note the narrow beam for close small objects,

(B) 1-inch diameter dowel, note the long narrow detection pattern,

(C) 3.25-inch diameter rod, note the long controlled detection pattern,

(D) 11-inch wide board moved left to right with the board parallel to the front sensor face and the sensor stationary. This shows the sensor's range capability.

Note: The displayed beam width of (D) is a function of the specular nature of sonar and the shape of the board (i.e. flat mirror like) and should never be confused with actual sensor beam width.



beam characteristics are approximate

Email: info@maxbotix.com
Web: www.maxbotix.com

LV-MaxSonar®-EZ0™ Pin Out

GND – Return for the DC power supply. GND (& Vcc) must be ripple and noise free for best operation.

+5V –Vcc – Operates on 2.5V - 5.5V. Recommended current capability of 3mA for 5V, and 2mA for 3V.

TX – When the *BW is open or held low, the TX output delivers asynchronous serial with an RS232 format, except voltages are 0-Vcc. The output is an ASCII capital "R", followed by three ASCII character digits representing the range in inches up to a maximum of 255, followed by a carriage return (ASCII 13). The baud rate is 9600, 8 bits, no parity, with one stop bit. Although the voltage of 0-Vcc is outside the RS232 standard, most RS232 devices have sufficient margin to read 0-Vcc serial data. If standard voltage level RS232 is desired, invert, and connect an RS232 converter such as a MAX232. When BW pin is held high the TX output sends a single pulse, suitable for low noise chaining. (no serial data).

RX – This pin is internally pulled high. The EZ0™ will continually measure range and output if RX data is left unconnected or held high. If held low the EZ0™ will stop ranging. Bring high for 20uS or more to command a range reading.

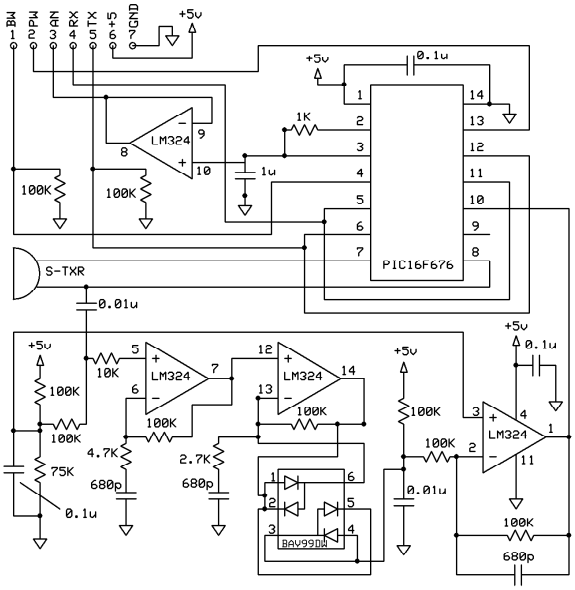
AN – Outputs analog voltage with a scaling factor of (Vcc/512) per inch. A supply of 5V yields ~9.8mV/in. and 3.3V yields ~6.4mV/in. The output is buffered and corresponds to the most recent range data.

PW – This pin outputs a pulse width representation of range. The distance can be calculated using the scale factor of 147uS per inch.

BW – *Leave open or hold low for serial output on the TX output. When BW pin is held high the TX output sends a pulse (instead of serial data), suitable for low noise chaining.

LV-MaxSonar®-EZ0™ Circuit

The LV-MaxSonar®-EZ0™ sensor functions using active components consisting of an LM324, a diode array, a PIC16F676, together with a variety of passive components.



LV-MaxSonar®-EZ0™ Timing Description

250mS after power-up, the LV-MaxSonar®-EZ0™ is ready to accept the RX command. If the RX pin is left open or held high, the sensor will first run a calibration cycle (49mS), and then it will take a range reading (49mS). Therefore, the first reading will take ~100mS. Subsequent readings will take 49mS. The LV-MaxSonar®-EZ0™ checks the RX pin at the end of every cycle. Range data can be acquired once every 49mS.

Each 49mS period starts by the RX being high or open, after which the LV-MaxSonar®-EZ0™ sends thirteen 42KHz waves, after which the pulse width pin (PW) is set high. When a target is detected the PW pin is pulled low. The PW pin is high for up to 37.5mS if no target is detected. The remainder of the 49mS time (less 4.7mS) is spent adjusting the analog voltage to the correct level. When a long distance is measured immediately after a short distance reading, the analog voltage may not reach the exact level within one read cycle. During the last 4.7mS, the serial data is sent. The LV-MaxSonar®-EZ0™ timing is factory calibrated to one percent at five volts, and in use is better than two percent. In addition, operation at 3.3V typically causes the objects range, to be reported, one to two percent further than actual.

LV-MaxSonar®-EZ0™ General Power-Up Instruction

Each time after the LV-MaxSonar®-EZ0™ is powered up, it will calibrate during its first read cycle. The sensor uses this stored information to range a close object. It is important that objects not be close to the sensor during this calibration cycle. The best sensitivity is obtained when it is clear for fourteen inches, but good results are common when clear for at least seven inches. If an object is too close during the calibration cycle, the sensor may then ignore objects at that distance.

The LV-MaxSonar®-EZ0™ does not use the calibration data to temperature compensate for range, but instead to compensate for the sensor ringdown pattern. If the temperature, humidity, or applied voltage changes during operation, the sensor may require recalibration to reacquire the ringdown pattern. Unless recalibrated, if the temperature increases, the sensor is more likely to have false close readings. If the temperature decreases, the sensor is more likely to have reduced up close sensitivity. To recalibrate the LV-MaxSonar®-EZ0™, cycle power, then command a read cycle.

Product / specifications subject to change without notice. For more info visit www.maxbotix.com/MaxSonar-EZ1_FAQ

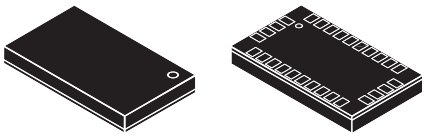
LYPR540AH

MEMS motion sensor: 3 axis analog output gyroscope

Preliminary data

Features

- Analog supply voltage 2.7 V to 3.6 V
- Wide extended operating temperature range (-40°C to 85°C)
- 3 indipendent angular rate channels
- ±400 dps and ±1600 dps full-scale
- High shock survivability
- Embedded self-test
- ECOPACK® RoHS and “Green” compliant (see *Section 5*)



LGA-28L (4.4x7.5x1.1 mm)

Application

- Motion and man machine interface
- Gaming and virtual reality input devices
- Fitness and wellness
- Pointing device and remote controllers
- Industrial and robotics
- Personal navigation devices

Description

The LYPR540AH is a three axis yaw, pitch and roll analog gyroscope featuring three separate analog output channels.

LYPR540AH provides amplified (±400 dps full scale) and not amplified (±1600 dps full scale) outputs for each sensible axis available at the same time through dedicated pins, and is capable of detecting rates with a -3 dB bandwidth up to 140 Hz.

ST 3 axis gyroscope family leverages on robust and mature manufacturing process already used for the production of hundreds milion micromachined accelerometers with excellent acceptance from the market.

Sensing element is manufactured using specialized micromachining processes, while the IC interfaces are realized using a CMOS technology that allows to design a dedicated circuit which is trimmed to better match the sensing element characteristics.

The LYPR540AH is available in plastic Land Grid Array (LGA) package. Several years ago ST pioneered successfully the usage of this package for accelerometers. Today ST has the widest manufacturing capability and strongest expertise in the world for production of sensors in plastic LGA package.

Table 1. Device summary

Order code	Temperature range [°C]	Package	Packing
LYPR540AH	-40 to +85	LGA-28L	Tray
LYPR540AHTR	-40 to +85	LGA-28L	Tape and reel

2 Module specifications

2.1 Mechanical characteristics

Vdd = 3V, T = 25 °C unless otherwise noted[a]

Table 3. Mechanical characteristics

Symbol	Parameter	Test conditions	Min.	Typ.[1]	Max.	Unit
FS	Measurement range	Not amplified output (X,Y,Z)		±1600		dps
FSA		Amplified output (4xX,4xY,4xZ)		±400		
So	Sensitivity	Not amplified output (X,Y,Z)		0.8		mV/dps
SoA		Amplified output (4xX,4xY,4xZ)		3.2		
SoDr	Sensitivity change vs. temperature			0.07		%/°C
Voff	Zero-rate level			1.5		V
VoffDR	Zero rate level drift over temperature			0.08		dps/°C
NL	Non linearity[2]	Best fit straight line		±1		% FS
BW	Bandwidth[3]			140		Hz
Rn	Rate noise density			0.02		dps/√Hz
Top	Operating temperature range		-40		+85	°C

1. Typical specifications are not guaranteed.
2. Guaranteed by design.
3. The product is capable of measuring angular rates extending from DC to the selected BW.

a. The product is factory calibrated at 3 V. The operational power supply range is from 2.7 V to 3.6 V.

D. Precision Issues

D.1. Non-deterministic Behavior of Floating-Point Calculations

As mentioned in section 5.6.3 the values of result1 and result2 in the following code are not always equal.

```
const float A, B, C, dt; // Any non-zero values
float result1 = (A*B)*dt;
float result2 = (A*B);
result2 *= dt;
```

Here is the results after running the below unit test.

```
X
  Expected: 0.275153548f
  But was:  0.275153786f
```

The estimated orientation error is very small and only presented itself after many iterations in the actual implementation. In simulations, as a consequence, the positional estimation reached about 3cm estimation error after 30 seconds of flight and diverged quickly due to errors in velocity estimates.

The reason we got this behavior was because the compiler may or may not decide to use the x86 high-precision flag when performing floating point calculations. In our pseudo-code, result2 may lose precision compared to result1 because it stores a temporary calculation to an intermediary single precision floating point variable, which truncates any extra precision.

Unit test code listing:

```
[TestFixture]
internal class QuaternionPrecisionTest
{
[Test]
public void Test()
{
    JoystickOutput output;
    output.Pitch = 0.312312432f;
    output.Roll = 0.512312432f;
    output.Yaw = 0.912312432f;

    const float dt = 0.017001f;
    float pitchRate = output.Pitch * PhysicsConstants.MaxPitchRate;
    float rollRate = output.Roll * PhysicsConstants.MaxRollRate;
    float yawRate = output.Yaw * PhysicsConstants.MaxYawRate;

    Quaternion orient1 = Quaternion.Identity;
    Quaternion orient2 = Quaternion.Identity;
    for (int i = 0; i < 10000; i++)
    {

      float deltaPitch = (output.Pitch * PhysicsConstants.MaxPitchRate) * dt;
      float deltaRoll =  (output.Roll * PhysicsConstants.MaxRollRate) * dt;
      float deltaYaw =   (output.Yaw * PhysicsConstants.MaxYawRate) * dt;

      // Add deltas of pitch, roll and yaw to the rotation matrix
      orient1 = VectorHelper.AddPitchRollYaw(
                               orient1, deltaPitch, deltaRoll, deltaYaw);

      deltaPitch = pitchRate * dt;
      deltaRoll = rollRate * dt;
      deltaYaw = yawRate * dt;
      orient2 = VectorHelper.AddPitchRollYaw(
                               orient2, deltaPitch, deltaRoll, deltaYaw);
    }

    Assert.AreEqual(orient1.X, orient2.X, "X");
    Assert.AreEqual(orient1.Y, orient2.Y, "Y");
    Assert.AreEqual(orient1.Z, orient2.Z, "Z");
    Assert.AreEqual(orient1.W, orient2.W, "W");
  }
}
```

D.2. Loss of Precision when Transforming Between Body Frame and Navigation Frame

We discovered a precision loss when transforming a vector between reference frames and this was particularly noticeable when the simulated acceleration vector in navigation frame had to be transformed to the body frame accelerometer and then transformed back

to navigation frame in the INS. We designed a test class FrameConversionPrecisionTest to measure the loss for different orientations of the helicopter and here are the results.

Although the error is typically in the order of 1E-6 for each roundtrip transformation this resulted in significant state estimation errors that accumulated over time.

```
Body frame pitch(0) roll(0) yaw(0) gave differences of {X:0 Y:0 Z:0}.
Body frame pitch(0) roll(0) yaw(1) gave differences of {X:9,536743E-07 Y:0 Z:0}.
Body frame pitch(1) roll(0) yaw(0) gave differences of {X:0 Y:9,536743E-07 Z:1,907349E-06}.
Body frame pitch(1) roll(1) yaw(0) gave differences of {X:1,907349E-06 Y:0 Z:3,814697E-06}.
Body frame pitch(0,6) roll(1,7) yaw(2,1) gave differences of {X:-2,861023E-06 Y:-1,430511E-06 Z:0}.
```