Implementierung eines Frameworks für generische Flugzeugsimulationen in C++

Abschlussbericht für das Modul Effizient programmieren I & II

von

Jan Olucak

durchgeführt am

Institut für Aerodynamik und Gasdynamik
der Universität Stuttgart

Stuttgart, im Juli 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Ausgangslage

Der Prozess des Flugzeugentwurfs erstreckt sich über mehrere Iterationen in denen das Flugzeug immer detaillierter ausgearbeitet wird. Um das Leistungsvermögen früh im Entwicklungsprozess zu beurteilen, können numerische Simulationen genutzt werden. Zum Einen werden mathematische Modelle benötigt, um das physikalische Verhalten von spezifischen Domänen des Flugzeuges abzubilden. Zum Anderen werden Parameter benötigt, um besagte Modelle zu beschreiben. Viele Parameter sind erst im Laufe des Entwurfsprozesses verfügbar. Um das Flugverhalten dennoch abzubilden, wird eine Simulation mit verschiedenen Ausbaustufen und Fehlermodellen benötigt.

In der Arbeit nach [1] wurde eine Matlab basierendes Programm entwickelt, mit dessen Hilfe Simulationsmodelle für Flugzeuge modelliert werden können. Die fertigen Modelle werden sowohl in Text-Files, als auch komprimierten mat-files gespeichert. Um die Modelle zu validieren wurde zudem eine Simulation mit 6 Freiheitsgraden implementiert. Das Programm Aircraft Designer kann jederzeit um weitere Methoden ergänzt werden, um so den Entwurfsprozess fortzusetzen. Mittels dem semi-empirischen Tool DataCompendium (DATCOM) wird die Aerodynamik des Flugzeuges für zuvor definierte Flugzustände berechnet. In einem automatisierten Prozess wird ein Vorgaberegler für die Flugzustände entworfen. Ist der Entwurfsprozess abgeschlossen kann durch die zuvor erwähnte Simulation die Güte der Modelle beurteilt werden und gegebenenfalls durch Anpassung der Parameter modifiziert werden. Die Laufzeit ist mitunter eine sehr kritische Größe. Zudem ist eine objektorientierte Programmierung nur bedingt möglich, was den Ausbau zu einer generischen Simulation erschwert.

1.2 Zielsetzung

Innerhalb dieser Ausarbeitung soll eine generische Flugzeugsimulation in der Hochsprache C++ implementiert werden. Generisch bedeutet in diesem Zusammenhang, dass die Simulationsumgebung nicht auf einen spezifischen Entwurf bezogen ist, sondern verschiedene Entwürfe mit dem in [1] entstandenen Aircraft Designer simulierbar sind. Somit sollen die Methoden sehr allgemein gehalten werden. Jede Ausbaustufe des Entwurfsprozesses soll simulierbar sein, ohne dass dabei der Code verändert wird. Über Input-Files sollen verschiedene Domänen-Modelle ausgewählt und die Simulation gesteuert werden können. Ein modularer Aufbau soll zudem eine einfache Erweiterung des Simulation gewährleisten. Zudem soll die Performance der Simulation hinsichtlich Rechenzeit gegenüber Matlab verbesert und im Nachgang weiter optimiert werden. Um das Simulations-Framework zu validieren, wird die in [1] implementierte Simulation in C++ übersetzt. Der gesamte Entwicklungsprozess stützt sich dabei auf die in [2] vorgestellten Methoden und Anregungen.

2 Werkzeuge für effiziente Code-Entwicklung

In diesem Kapitel werden die verwendeten "Werkzeuge"näher erläutert, die bei der Code-Entwicklung unterstützt haben. Dabei wird auch auf Aspekte der Vorlesungsreihe Effizient programmieren [2] eingegangen.

2.1 Entwicklungsumgebung und Versionsmanagement

Wie in [2] beschrieben können Entwicklungsumgebungen bei der Entwicklung und Implementierung von Programmen unterstützen. Aufgrund des Windows-Betriebssystems ist die Entscheidung auf Microsoft Visual Studio 2017 gefallen. Für die Reproduzierbarkeit wurde ein Repository bei GitHub angelegt. Da es sich bei dieser Arbeit nicht um ein Gruppenprojekt handelt, wurde lediglich ein Trunk angelegt. Es wurde stets darauf geachtet, dass die Version nur comitted und gepushed wurde, wenn das Programm lauffähig war und zuvor getestet wurde [2].

2.2 Code Dokumentation und User Manual

Ein weit verbreitetes Problem ist die Dokumentation von Programmen. Das Problem besteht darin, dass ein zusätzliches Dokument gepflegt werden muss. Um dieses Programm zu dokumentieren wird Doxygen genutzt. Mit speziellen Kommentarzeilen erzeugt Doxygen automatisch ein HTML oder LaTeXDokument. Zusätzlich werden die Abhängigkeiten der Klassen zueinander visualisiert. Somit entsteht nur ein geringer Mehraufwand.
Für die Installation und Nutzung der Simulation wurde eigens ein User Manual erstellt.

2.3 Verwendung von Open Source Bibliotheken

Nach [3] ist es nicht immer notwendig bei gewissen Funktionalitäten von Grund auf neu zu beginnen, sondern auf bereits bestehendes Aufzubauen. Da in dieser Arbeit die Simulation im Vordergrund steht und nicht die Implementierung von grundlegenden Funktionen, wurden neben den C++ Standardbibliotheken ebenfalls Open Source Bibliotheken eingebunden. Aufgrund dessen, dass besagte Simulation auf dem Matlab Programm nach [1] beruht, ist die Lineare Algebra unverzichtbar. Zudem wurden Daten häufig in komprimierten .mat-Files gespeichert. Es liegt also nahe diese grundlegenden Funktionen mithilfe von bereits vorhanden Bibliotheken nach Möglichkeit abzudecken.
Die in diesem Projekt verwendeten Open Source Bibliotheken werden in Tabelle 2.1 kurz beschrieben.

Name der Bibliothek	**Kurzbeschreibung**
Eigen	C++ Template Bibliothek für lineare Algebra [4]
matio	C/C++ Bibliothek, die Funktionen bereitstellt, um Daten aus und in .mat-Files zu lesen/schreiben [7]

Tabelle 2.1: Übersicht über verwendete Open Source Bibliotheken

Wie aus 2.1 ersichtlich handelt es sich bei der Eigen-Bibliothek um Templates. Nachdem Eigen in das Programm eingebunden wird, kann sofort auf die Funktionen zugegriffen werden. Zum Einlesen der .mat-Files wurden aufbauend auf den Funktionen aus der matio-Bibliothek selbst eine Klasse implementiert, die diese Funktionalität bereitstellt. Um matio zu verwenden werden sei darauf verwiesen, dass zusätzlich noch die Bibliotheken HDF5 und zlib benötigt werden.

3 Simulations-Framework

Im Nachfolgenden Kapitel soll die Grundidee des Simulations-Frameworks erläutert werden. Zum Einen soll ein modularer Aufbau umgesetzt werden, sodass im späteren Verlauf die Module nach dem Baukastenprinzip zu einer Gesamtsimulation zusammengesetzt werden und jederzeit um weitere Klassen ergänzt werden können. Zum Anderen soll der Nutzer in der Lage sein, in den Modulen verschiedene Methoden zu nutzen, ohne dabei den Code zu verändern.

3.1 Aufbau der Module

Jedes Modul beschreibt eine spezifische Domäne des Flugzeuges und besitzt mehrere Klassen, die im Nachfolgenden allgemein erläutert werden. Eine Klasse ist für die Auswahl und den Aufruf der gewünschten Modelle zuständig und besitzt den gleichen Namen wie das Modul (Modulklasse). Sie wird im späteren Verlauf in der Simulation aufgerufen und dient sozusagen als Schnittstelle zu den anderen Modulen. Um nun eine gewünschte Methoden aufrufen zu können, ohne dabei den Code zu verändern, wird auf das Prinzip der Vererbung zurückgegriffen. Die Kind-Klassen können von der Elternklasse oder von einem Kind abgeleitet werden. Der allgemeine Aufbau eines Moduls wird in Abbildung exemplarisch 3.1 dargestellt.

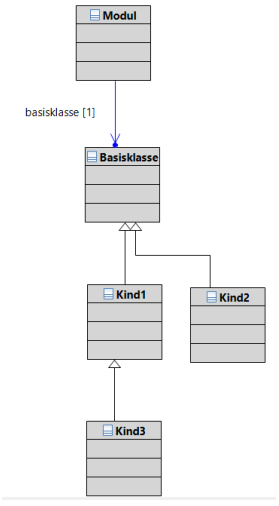

Abbildung 3.1: Allgemeiner Aufbau eins Moduls

Zusätzlich besitzen alle Klassen neben Konstruktor und Destruktor auch Standard-Methoden, die in Tabelle 3.1 beschrieben werden.

Standard-Methode	**Bedeutung**
init(...)	Die für die jeweilige Domäne benötigten Parameter werden aus den Input-Files eingelesen und Variablen initialisiert.
update(...)	Methode in der die spezifische Domäne simuliert wird

Tabelle 3.1: Standard-Methoden der Simulation

Die Standard-Methoden sollen einen einheitlichen Funktionsaufruf gewährleisten. Neben besagten Standard-Methoden können die Klassen auch noch zusätzliche Methoden besitzen, worauf später noch näher eingegangen wird.
Im Speziellen wird hier Polymorphie mithilfe von virtuellen Funktionen genutzt. Dazu werden die Standard-Methoden der Basisklasse als virtuelle Funktionen deklariert, was exemplarisch im Listing 3.1 zu sehen ist.

```
        virtual initBasisklasse();

        virtual updateBasisklasse();
```

Listing 3.1: Aufbau des Header-Files der Basisklasse

Aus diesem Grund lässt sich die Basisklasse auch als polymorphe Klasse bezeichnen [5].
Um nun zu gewährleisten, dass das gewünschte Modell des Moduls ausgewählt wird, wird ein ein Basisklassenzeiger verwendet. Der Zeiger kann auf jedes Objekt adressiert werden, dass von ihm abgeleitet wurde [5]. Die eigentliche Initialisierung findet in der Modulklasse statt. Dazu wird neben den Standard-Methoden noch die Methode **SelectModuleType** implementiert, wobei **Module** für das jeweilige Modul steht.

```
void Modulklasse::SelectModuleType(int Type)

switch(Type){
        case 1:
                BasePtr= new Kindklasse1;
                break;

        case 2:
                BasePtr = new Kindklasse2;
                break;
}
```

Listing 3.2: Basisklassenzeiger im Modul.cpp File

Durch das Schlüsselwort **virtual** in der Basisklasse wird der Compiler veranlasst, zum entsprechenden Objekt die dazugehörige Methode aufzurufen, wenn das Objekt über den Zeiger auf die Basisklasse angesprochen wird [5].
Im nächsten Schritt können nun in der Modulklasse die Standard-Methoden der Basisklasse aufgerufen werden, wie im Lisiting 3.3 dargestellt.

```
void Modulklasse::initModulklasse()
{
        BasePtr->initBasisklasse();
}

void Modulklasse::updateModulklasse()
{
        BasePtr->updateBasisklasse();
}
```

Listing 3.3: Aufruf der Standard-Methoden im Modul.cpp File

Somit können die Methoden der Kind-Klassen, wo die Modelle implementiert wurden, aufgerufen werden, in dem die Standard-Methoden der Modulklasse aufgerufen werden. Der Nutzer muss lediglich den spezifische Modell-Parameter im Input-File ändern, um ein anderes Model auszuwählen.

3.2 Aufbau und Ablauf der Ausbaustufen

Durch die Beschreibung der Flugzeugdomänen mithilfe der Module, ist man nun in der Lage die eigentliche Simulation im Baukastenprinzip aufzubauen.
In Tabelle 3.2 werden alle Module, die für die Ausbaustufen der Simulation benötigt werden, kurz umschrieben. Dabei wird lediglich die allgemeine Funktion des jeweiligen Moduls beleuchtet, anstatt deren Implementierung, um einen Überblick zu erhalten.

Modulname	**Kurzbeschreibung**
Actuator	Simuliert die Rudermaschinen des Flugzeuges
Aerodynamic	Berechnung der aerodynamischen Kräfte und Momente
Airframe	Enthält die Bewegungsgleichungen des Flugzeuges
Atmosphere	US Standard-Atmosphäre 1976
Autopilot	Vorgaberegler
Engine	Berechnung der Schubkräfte und -momente
Guidance	Flugführungssystem
IMU	Inertialsensorik
Navigation	Berechung der Navigationslösung für das GNC.

Tabelle 3.2: Module der generischen Flugzeug-Simulation

An dieser Stelle sei angemerkt, dass noch weitere Module vorhanden sind , die allerdings vielmehr eine sekundäre Rolle für die eigentliche Simulation einnehmen. Auf besagte Module wird im späteren Verlauf dieser Arbeit noch eingegangen. Zudem sei noch erwähnt, dass nur Domänen implementiert wurden, die für eine nichtlineare flugmechanische Simulation benötigt werden. Eine Erweiterung um weitere Domänen wäre denkbar.
Wie bereits in 1.1 beschrieben, werden verschiedene Ausbaustufen der Simulation benötigt. Unter Ausbaustufe ist hierbei die Anzahl an Freiheitsgraden und Detaillierungsgrad gemeint. Die Ausbaustufen werden im Modul **Trajectory** implementiert. Der Aufbau und der Aufruf erfolgt wie in Abschnitt 3.1 beschrieben. Je nach Ausbaustufe werden innerhalb des Trajektorien-Moduls die in Tabelle 3.2 beschriebenen Module über ihre Modulklasse aufgerufen. Insgesamt sind drei Ausbaustufen vorgesehen, die im Folgenden erläutert werden.

Die niedrigste Ausbaustufe ist eine Simulation mit 3 Freiheitsgraden. Es werden nur die translatorischen Bewegungsgleichungen berücksichtigt. Die Aufruf-Reihenfolge wird in Abbildung ?? aufgezeigt.

Abbildung 3.2: Aktivitätsdiagramm der Trajektorie mit 3 Freiheitsgraden

Eine solche Ausbaustufe könnte beispielsweise schon sehr früh genutzt werden, wenn zum Beispiel noch keine Trägheitsmomente bekannt sind.
Die nächst höhere Ausbaustufe besitzt 6 Freiheitsgrade. Sie ist eine Kindklasse der zuvor beschriebenen Ausbaustufe. Der Ablauf ist in ?? zu sehen.

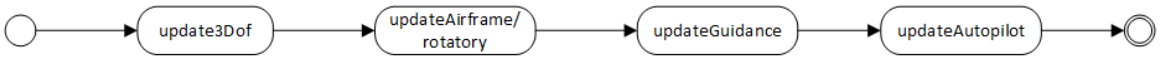

Abbildung 3.3: Aktivitätsdiagramm der Trajektorie mit 6 Freiheitsgraden

Die Kindklasse wird um die rotatorischen Bewegungsgleichungen ergänzt. Außerdem erhält sie ein Flugführungssystem und einen Vorgaberegler. In den beiden bisherigen Ausbaustufen wird das Systemverhalten als ideal angesehen. Das bedeutet, dass weder Messfehler durch Sensorik oder Verzögerung durch Aktuatorik berücksichtigt werden. Eine solche Ausbaustufe könnte beispielsweise genutzt werden, um die Flugregelung auf dem nichtlinearen Modell zu testen und zu beurteilen.
Unabhängig von der Güte der physikalischen und mathematischen Modelle ist es dennoch wichtig, ab einem gewissen Grad des Entwicklungsprozesses Fehlermodelle für Subsysteme zu berücksichtigen. Somit wird eine weiter Ausbausstufe benötigt. Diese ist in Abbildung ?? zu sehen.

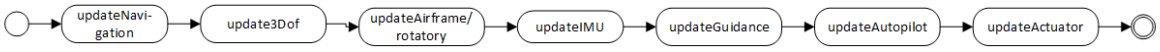

Abbildung 3.4: Aktivitätsdiagramm der Trajektorie mit 6 Freiheitsgraden unter Berücksichtigung von Fehlermodellen

Die höchste Ausbaustufe erbt von der ersten Ausbaustufe. Anstatt der idealen Integration der Kinematik wird ein Naviagations-Modul verwendet. Dort wird im Regelfall eine Navigationslösung erzeugt, in dem die Sensormessungen genutzt werden um durch einen Kalman-Filter Position und Lage zu schätzen. Zudem wird die Messung durch Inertialsensoren berücksichtigt. Verzögerungen, die beispielsweise durch die Rudermaschinen verursacht werden, können im Akutator-Modul modelliert werden.

3.3 Gesamtsimulation

Die zuvor beschriebenen Ausbaustufen können genutzt werden, um die Flugbahn eines Flugzeuges zu simulieren. Somit ist es naheliegend das Gesamtflugzeug in einem eigenen Modul zu beschreiben, dem **Aircraft**-Modul.
Neben dem Trajektorien-Modul wird dort ebenfalls die Atmosphäre initialisiert. Der Ablauf des Gesamtflugzeuges wird in Abbildung **??** aufgezeigt, wobei angemerkt sei, dass sämtliche Objekte bereits initialisiert wurden.

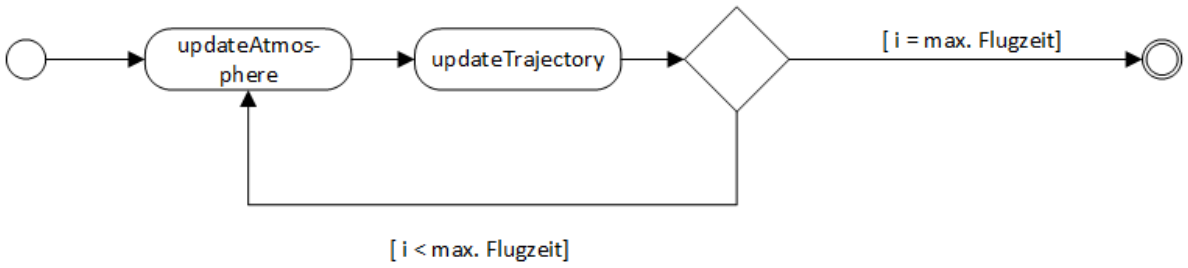

Abbildung 3.5: Aktivitätsdiagramm der Gesamtflugzeug-Simulation

4 Durchführung der Implementierung

4.1 Implementierungs-Prozess

Für die Validierung des Frameworks wird die Matlab-Simulation nach [1] verwendet. Da es hier lediglich um eine Konvertierung des Codes handelt, werden die mathematischen Modelle nicht explizit erklärt. Vielmehr wird das Vorgehen der Implementierung erläutert, um zu zeigen, wie das Programm effizient entwickelt wurde. Gleiches gilt für die Implementierung der Simulationswerkzeuge.
Die Simulations-Umgebung wurde sukzessive aufgebaut. Es wurde stets darauf geachtet, dass nach der Implementierung einer Funktion nach Möglichkeit direkt getestet wurde. Erst mit der nachgewiesenen Funktionalität, wurde der nächste Schritt im Prozess durchgeführt. In Abbildung **??** wird dieser Prozess dargestellt.

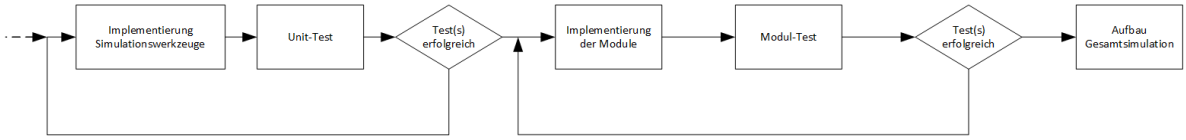

Abbildung 4.1: Flussdiagramm des Implementierungs-Prozess

4.2 Simulationswerkzeuge

Unter Simulationswerkzeugen sind Funktionen zu verstehen, die in allen Teilen des Simulations-Frameworks vorkommen. Dies umfasst beispielsweise das Einlesen und Schreiben von Simulationsparametern oder mathematische Funktionen wie Integration oder Interpolation. Es ist offensichtlich, dass besagte Funktionen als erstes implementiert werden müssen, damit die Module im späteren Verlauf auf diese zurückgreifen können. Diese Werkzeuge wurden eigens in das Modul **Tools** implementiert. Auch die in 2.3 beschriebenen Bibliotheken fallen unter diese Werkzeuge. Die implementierten Simulationswerkzeuge werden in Tabelle **??** zusammengefasst.

Funktion/Klasse	**Kurzbeschreibung**
Constants	physikalische Konstanten
DataLogger	Erzeugung von Text-Files (Output der Simulation)
LinearInterpolation	Ein- und zweidimensionale Interpolation
MatFileReader	Funktionen zum Einlesen von .mat-File
ODESolver	Template(s) für numerische Integration
readInData	Funktionen zum Einlesen von Parametern aus Text-Files
Transformation	Matrizen für Koordinatentransformation

Tabelle 4.1: Simulationswerkzeuge der generischen Flugzeugsimulation

4.2.1 Unit-Tests

Um sicherzustellen, dass die Werkzeuge korrekt implementiert wurden, wurden Unit-Test implementiert. Dazu wurde das von Visual Studio bereitgestellte Microsoft-Komponententest-Frameworks für C++ genutzt. Die Unit-Tests wurden in einen gleichnamigen Projektordner ausgelagert. An dieser Stelle werden nicht alle Test erklärt. Vielmehr soll das Test-Schema erläutert werden.
In dieser Arbeit wird das AAA-Schema (Arrange, Act, Assert) nach [6] genutzt. Dazu werden die Tests in die zuvor genannten Abschnitte aufgeteilt. Im einzelnen haben die Abschnitte folgende Bedeutung [6]:

- Arrange dient der Initialisierung des Tests. Die benötigten Objekte werden initalisiert und Parameter für den Testfall festgelegt.
- Innerhalb von Act wird die zu testende Methode mit den zuvor definierten Testfall/Parametern aufgerufen.
- Im Abschnitt Assert werden die Referenzwerte mit den Werten aus dem eigentlichen Test vergleichen. Wird eine Übereinstimmung festgestellt, erhählt der Testfall seine Bestätigung durch das Framework.

Die Referenzwerte wurden entweder selbst festgelegt (z.B. Einlesen eines Parameters) oder es wurde Matlab genutzt, um Referenzwerte zu generieren (z.B. Interpolation).
Nach [4] handelt es sich bei eigen um eine vollständig getestete Bibliothek. Aus diesem Grund wurden nur Unit-Tests für einige wichtige Funktionen durchgeführt. Bei der auf [7] beruhenden MatFileReader-Klasse wird nur der selbst geschrieben Code getestet. Eine Außnahme bei den Unit-Test stellt das Header-File Constants, bei der lediglich physikalische und mathematische Konstanten hinterlegt sind.

4.3 Implementierung und Testen der Module

In Abschnitt 3.1 wurde bereits der Aufbau der Module erläutert und in Tabelle 3.2 die für die Simulation benötigten Module aufgelistet. Um deren Funktionalität nachzuweisen, wurden Modul-Tests durchgeführt. Wie schon erwähnt, wird die 6 Dof-Simulation nach [1] genutzt, um das Simulations-Framework zu testen. Somit liegen nur zu den dort verwendeten Modellen Referenzwerte vor. In diesem Abschnitt beziehen sich die Tests nicht auf das jeweilige Gesamtmodul, sondern lediglich auf die bisher implementierten Klassen. Die Schwierigkeit, die mit den Modul-Tests einhergeht, ist die Definition der Testfälle. Ziel ist es eine größtmögliche Testabdeckung zu haben, um zu garantieren, dass das Modul wie gewünscht funktioniert. Bei einigen Modulen ist das Testen nur bedingt oder nur im Gesamtsystem-Test möglich. Dies betrifft beispielweise das **Autopilot** und **Guidance** Modul. Für beide Module werden Parameter von anderen Modulen benötigt.
Eine weitere Problematik besteht bei den verschiedenen Modellen der Ausbaustufen. Für die Simulation mit 3 Freiheitsgraden liegt keine Referenz vor. Ähnlich verhält sich die 6 Dof-Simulation mit Fehlermodellen. Es liegen aktuell keine Modelle für Sensorik und Aktuatorik vor. Somit können an dieser Stelle keine Modul-Tests für die zusätzlichen Module erfolgen. In Tabelle **??** werden die Module und deren Testfälle aufgezeigt, für die ein Modul-Test in Betracht kommt.

Modul	**Beschreibung Testfall**
Atmosphere	Bei dem hier hinterlegten Modell handelt es sich um die US Standard-Atmosphäre von 1976. Temperatur, Druck, Dichte und Schallgeschwindigkeit sind hierbei abhängig von der Flughöhe. Somit werden die zuvor beschriebenen Größen von 0-10000m berechnet und mit Daten der Standard-Atmosphäre verglichen.
Aerodynamic	Hier wird ein Windtunnel bei konstanter Höhe simuliert. Es werden Machzahl, Anstellwinkel und Höhenruder-Winkel variiert. Es ist ersichtlich, dass nur die Längsbewegung betrachtet wird. Dies ist auf mangelnde Referenzwerte für die Seitenbewegung zurückzuführen. Ziel dieses Tests ist es die Polare des Flugzeuges abzufahren und mit den Matlab-Daten zu vergleichen.
Autopilot	Die Parameter für das Gain Scheduling werden eingelesen und die Zustände auf Arbeitspunkt (Trimmpunkt) initialisiert. Es wird überprüft, ob die Scheduling Parameter und Stellgrößen nach der Initialisierung mit denen aus Matlab übereinstimmen.
Trajectory	Der Trajectory-Modul Test testet die korrekte Berechnung der Flugbahn. Hier wird die Matlab-Simulation simuliert. Im Anschluss können die Ergebnisse verglichen werden.

Tabelle 4.2: Auflistung der Modultests

Aufgrund der Einfachheit des aktuell hinterlegten Schub-Models (Engine-Modul) wurde auf einen Modul-Test verzichtet und der Nachweis über Code-Reading durchgeführt. Es ist offensichtlich, dass das Trajektorien-Modul erst aufgebaut werden konnte, nachdem die dafür benötigten Module bereits erfolgreich getestet wurden. Die Ergebnisse wurde rein qualitativ überprüft.

4.4 Validierung des Simulations-Framework

Ziel von [1] war es, den Flugzustandsregler zu testen. Dem Vorgaberegler wurden Querbeschleunigungen als Soll-Wert vorgegeben. Er hat das Ziel diesen Querbeschleunigungen zu folgen.

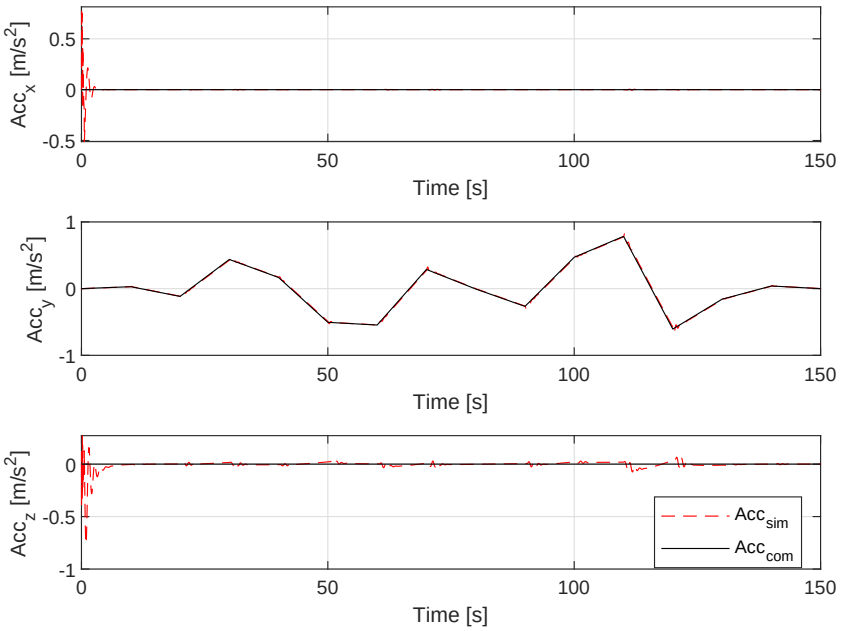

Abbildung 4.2: Validierung der Simulation

In Abbildung 4.2 sieht man, wie der der Ist-Wert des Reglers den Soll-Werten der Guidance sehr gut folgt. Die anfänglichen Schwingungen sind auf die Initialisierung der Simulation am Trimmpunkt zurückzuführen. Da die Arbeitspunkte durch Linearisierung gefunden werden, entsteht ein Modellfehler. Nachfolgend werden die Ergebnisse von Matlab und C++ verglichen.

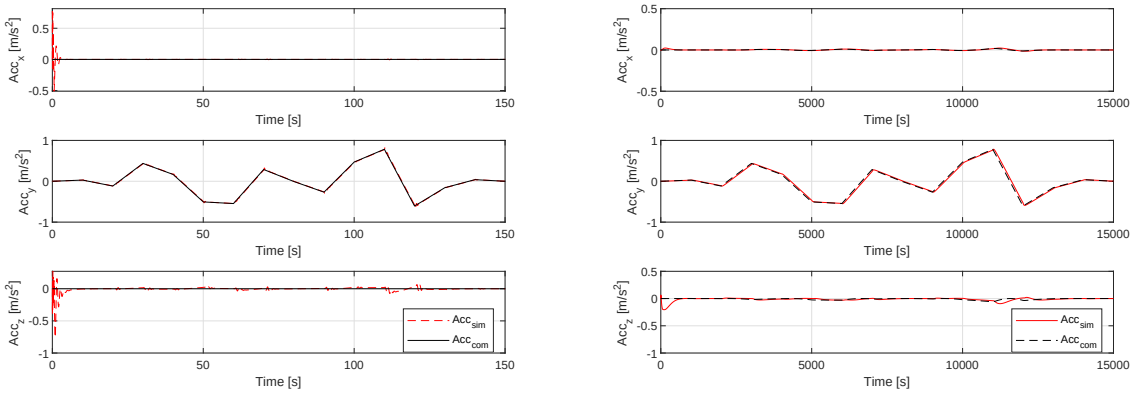

Abbildung 4.3: Vergleich der Simulationsergebnisse, links C++, rechts Matlab

Bei beiden Simulationen sind die Initialisierungsfehler zu erkennen, wobei in C++ die anfänglichen Schwingungen größere Amplituden besitzen. Zudem sind auch kleinere numerische Unterschiede zu erkennen. Nach [Thread-Stackoverflow] sind diese numerischen Unterschied auf Berechnungsungenauigkeit von trigonometrischen Funktionen zwischen Matlab und C/C++ normal. Es ist also zu erwarten, dass die beiden Ergebnisse nicht vollständig übereinstimmen. Die Simulation nach [1] diente nur der Validierung und nicht der Verifikation. Das Ziel, dass das Flugzeug den Querbeschleunigungsvorgaben folgt, wurde erfüllt. Die höchste Ausbaustufe liefert die gleichen Ergebnisse wie die Ausbaustufe ohne Fehlermodelle. Dies ist darauf zurückzuführen, dass speziellen Methoden implementiert wurden, die ein fehlerfreies Verhalten, zum Beispiel der Navigation, simulieren.

4.4.1 Direktvergleich Rechenzeit Matlab und C++

In Abbildung 4.4 wird die Rechenzeit der Simulation in Matlab und C++ gegenübergestellt. Der Performance Gewinn in C++ ist offensichtlich zu erkennen.

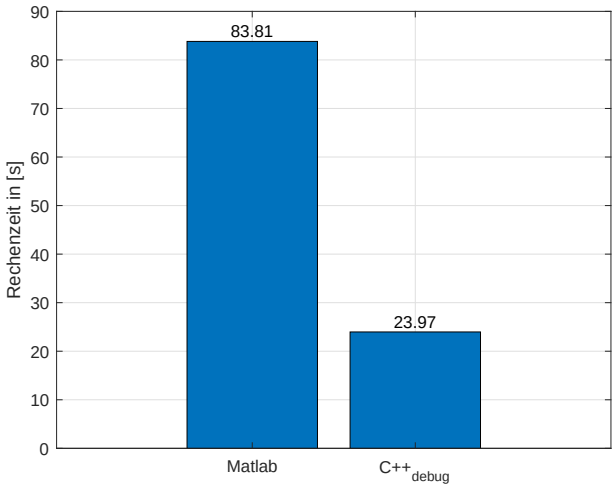

Abbildung 4.4: Direktvergleich Matlab und C++

Betrachtet man das Ergebnis von der quantitativen Seite und berechnet den relativen Rechenzeitunterschied ist das C++ Programm 71,4% schneller als Matlab. Dieses Ergebnis ist weniger verwunderlich, da C++ das Programm in Maschinencode übersetzt. Matlab ist eine proprietäre Programmiersprache die auf dem jeweiligen Rechner interpretiert wird und kann dementsprechend die Ressourcen des Computers nicht voll ausnutzen.
Mit diesem Ergebnis wurde ein Ziel dieser Arbeit, die Rechenzeit zu reduzieren, erreicht.

5 Optimierung der Simulation

In der vorangegangen Kapiteln wurde die Umsetzung der Simulation erläutert und gezeigt, dass ein funktionsfähiges Framework validiert werden konnte. Im letzten Schritt dieser Ausarbeitung geht es um die Performance Optimierung hinsichtlich der Rechenzeit. Somit soll die Effizienz gesteigert und Schwachstellen im Code korrigiert werden. An dieser Stelle werden die Randbedingungen kurz erläutert, um gleiche Testbedingungen zu garantieren und um die Ressourcen des Rechners vollständig auszunutzen:

- Um statistisch aussagekräftige Ergebnisse bezüglich der Rechenzeit zu erhalten werden 5 Läufe durchgeführt und am Schluss gemittelt,
- Hintergrundprozesse sollen vermieden werden,
- Der Rechner (Laptop) soll an einer Stromquelle angeschlossen sein.

5.1 Profiling

Mithilfe des integrierten Leistungsprofilers wird das Laufzeitverhalten der Simulation untersucht. Somit sollen Problembereiche aufgezeigt werden, die kritisch hinsichtlich der Rechenzeit sind. In dieser Arbeit wird der Profiler genutzt, um die Anzahl der Aufrufe und Durchläufe von Funktionen festzustellen. Somit kann das Hauptaugenmerk auf Funktionen gerichtet werden, die durch ihre Aufrufe viel Zeit in Anspruch nehmen. In Tabelle 5.1 werden besagte Funktionen mit ihrer jeweiligen CPU-Zeit aufgelistet.

Methode	**Kurzbeschreibung**	**CPU-Zeit**
Trajectory6Dof::log6DofData()	Daten-Logging der einzelnen Module	50,79%
updateAutopilot::updateStateController(...)	Zustandsregler	22,62%
LinearInterpolation::linearInterpolation2D(...)	Tabellen Interpolaton	13,18%

Tabelle 5.1: Ergebnisse des Leistungsprofilers

Es ist offensichtlich, dass das erzeugen der Output-File den größten Einfluss auf die gesamt Rechenzeit hat. Im nächsten Schritt müssen mögliche Ursachen gefunden werden, die für die jeweiligen Funktionen die Rechenzeit beeinflussen. Im Anschluss muss eine Lösung für die Rechenzeit-Reduktion gefunden werden. In Tabelle 5.2 werden die Ursachen und die vermeintlichen Lösungen gegenüber gestellt.

Methode	**Ursachen**	**Lösungsansatz**
...log6DofData()	nicht benötigte Daten werden geloggt	Reduktion des Outputs
	std::to_string (25,5%)	alternative Methode finden (Benchmarking)
...updateStateController(...)	Eigen::Library	alternative Bibliotheken
...linearInterpolation2D(...)	Index suche im Array	Suchmethoden anpassen

Tabelle 5.2: Gegenüberstellung Ursachen und Lösungsansätze der Rechenzeit

5.2 Code Optimierung

Wie in Tabelle 5.2 beschrieben, wurde der Output der einzelnen Module reduziert. Da das Hauptaugenmerk auf der Überprüfung der Flugregelung steht, wurden Parameter vernachlässigt, die wenig Aussagekraft darüber besitzen.
Der Flugzustandsregler selbst beruht in großen Teilen auf linearer Algebra. In dieser Ausarbeitung wurde die lineare Algebra durch die Open Source Bibliothek Eigen realisiert. Eine Effizienzsteigerung wäre mithilfe einer alternativen Bibliothek denkbar, wurde aber nicht weiter betrachtet. Der Aufwand, sämtliche mathematischen Operationen in der Gesamtsimulation abzuändern wäre zu groß. Nach [4] gibt es spezielle Einstellungen für Visual Studio, um die Performance der Bibliothek zu steigern, welche letztlich umgesetzt wurden.
Für die Interpolation werden Stützstellen benötigt. Diese Stützstellen zu finden ist mit großem Aufwand verbunden, da dass gesamte Array durchsucht werden muss. Es wurde versucht, diese Suche zu beschleunigen. Diese algorithmische Optimierung hat sich als äußerst komplex rausgestellt und konnte nicht gelöst werden. Das Ergebnis des ersten Teils der Optimierung ist in Abbildung 5.1 zu sehen.

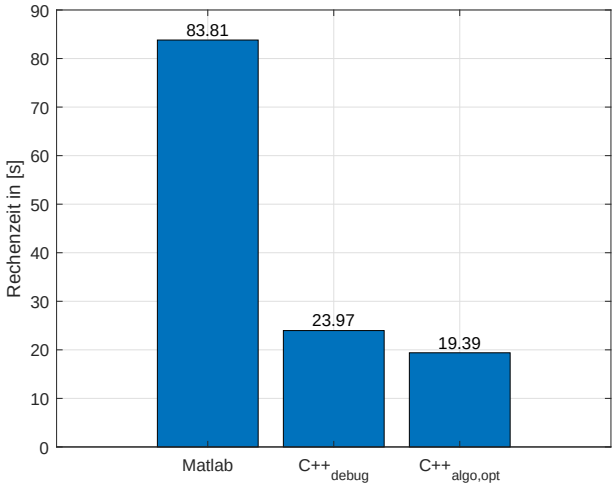

Abbildung 5.1: Gegenüberstellung nach Code Optimierung Teil 1

Durch kleinere Maßnahmen im Code konnte die Rechenzeit nochmals um rund 4 Sekunden reduziert werden, was in etwa 19% zum ursprünglichen C++ Code und 76,86% zu Matlab schneller ist.

5.2.1 Benchmarking String-Typumwandlung

Wie in Tabelle5.2 angedeutet, beschäftigt sich dieser Abschnitt mit der Untersuchung von verschiedenen Möglichkeiten, die Typumwandlung in Strings umzusetzen. Neben den Standardbibliotheken werden an dieser Stelle auch Open Source Bibliotheken hinzugezogen. Zum Einen wurden zwei Methoden der Boost-Bibliothek genutzt. Besagte Bibliothek wurde zur Produktivitätssteigerung in C++ erschaffen. Zum Anderen wurde strtk (string Toolkit) genutzt. Dabei handelt es sich um eine Bibliothek, die speziell für den Umgang von Strings implementiert wurde. In Tabelle 5.3 werden die unterschiedlichen Methoden aufgelistet.

Methode
std::ostringstream
sprintf_s
std::to_string
boost::lexical_cast<std::string>(...)
boost::str(boost::format("%.4f") % (value))
strtk::type_to_string<double>(value)

Tabelle 5.3: Gegenüberstellung der Methoden für die Typumwandlung in String

Das Ergebnis des Benchmarkings ist in Abbildung 5.2 zu sehen. In diesem Fall erwies sich die Methode sprintf_s als sehr effizient.

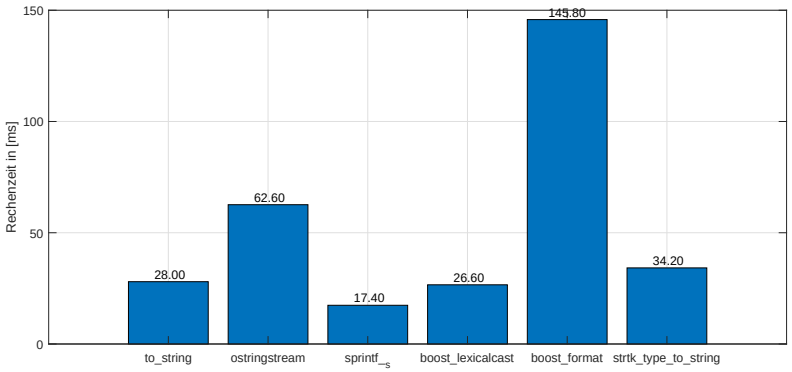

Abbildung 5.2: Ergebnisse des Benchmarkings

Die Methoden der Open Source Bibliotheken enttäuschen hingegen und werden aus diesem Grund in dieser Arbeit nicht weiter betrachtet. Aufgrund des Benchmarking-Ergebnisses wurde die std::to_string Methode durch sprintf_s ersetzt. Das Ergebnis der zweiten Code Optimierung ist in Abbildung 5.3 zu sehen.

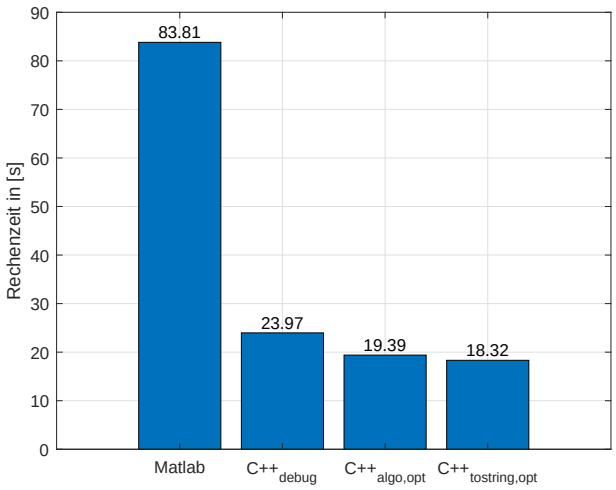

Abbildung 5.3: Gegenüberstellung nach Code Optimierung Teil 2

Auch wenn die Rechenzeit sich nur minimal verbessert hat, so ist dies dennoch als gutes Ergebnis zu betrachten. Quantitativ ausgedrückt, hat sich die Rechenzeit gegenüber der ersten Optimierung konnten nochmals 5,5% verbessert. Stand hier wurde die Rechenzeit gegenüber Matlab um 78,1% verbessert.

5.3 Parallelisierung und Vektorisierung

Eine weitere Möglichkeit, neben der Code Optimierung, die Performance zu steigern ist die Zuhilfenahme von automatischer und gesteuerter Parallelisierung [3] die Performance weiter zu verbessern. Zusätzlich wurde auch noch der Einfluss von automatischer Vektorisierung untersucht. Die Ergebnisse der Parallelisierung sind auf den jeweiligen Rechner bezogen, wo diese durchgeführt wurde. Aus diesem Grund wird an dieser Stelle der Rechner, womit die Implementierung durchgeführt wurde, kurz aufgezeigt. Die grundlegenden Daten des Rechners sind in Tabelle 5.4 zusammengefasst.

Eigenschaft	**Beschreibung/Wert**
Betriebssystem	Windows 10
Prozessor	Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz 2.81GHz
RAM	8 GB
Systemtyp	64-Bit

Tabelle 5.4: Rechnerdaten

Ausgangslage für die weitere Optimierung stellt das Ergebnis der Code Optimierung.

5.3.1 Automatische Parallelisierung und Vektorisierung

Die automatische Parallelisierung und Vektorisierung kann schnell und einfach über Compiler-Flags durchgeführt werden. Der Compiler entscheidet selbstständig, ob er beispielsweise einen Abschnitt parallelisiert oder nicht. Die in dieser Arbeit genutzten Flags von Visual Studio werden in Tabelle 5.5 aufgezeigt.

Flag-Name	**Verwendung**
/Qpar	automatische Parallelisierung
/arch	automatische Vektorisierung

Tabelle 5.5: Compiler-Flags für automatische Parallelisierung und Vektorisierung

Die Flags werden sowohl einzeln als auch gemeinsam betrachtet. In Abbildung 5.4 sind die Ergebnisse dargestellt.

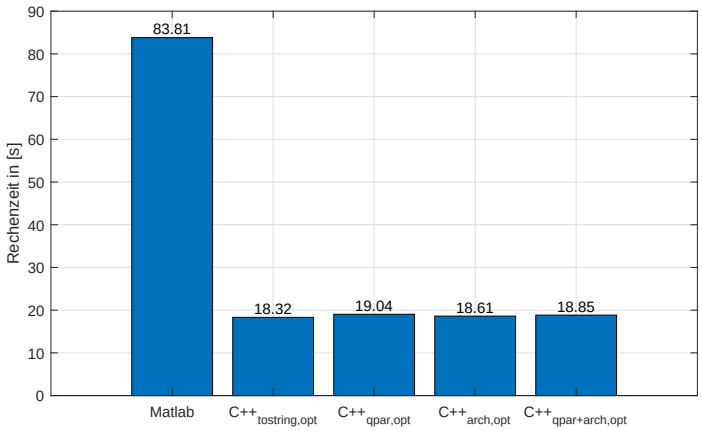

Abbildung 5.4: Ergebnisse der automatischen Parallelisierung und Vektorisierung

Durch die automatische Parallelisierung und Vektorisierung wurde die Performance geringfügig schlechter. Dieses Ergebnis ist nach [3] durchaus möglich. Aufgrund des fehlenden Performance-Gewinn, wird auf die Flags im Weiteren verzichtet.

5.3.2 Gesteuerte Parallelisierung

Bei der gesteuerten Parallelisierung wird der Compiler durch Direktiven im Code angewiesen eine Parallelisierung durchzuführen. In dieser Ausarbeitung wurde der OpenMP Standard genutzt.
Es war leider nicht möglich, ein sinnvolles Ergebnis zu erzeugen. Grund hierfür liegt im Ablauf der Simulation selbst. Die Berechnung der Trajektorie findet lediglich in einer for-Schleife statt. Somit sind die Ergebnisse der direkt aufeinanderfolgenden Zeitschritte abhängig voneinander. Aus rein physikalischer Sicht ist es beispielsweise nicht möglich, dass ein Thread schon den nächsten Zeitschritt berechnet, bevor der vorangegangene Zeitschritt berechnet wurde. Somit wird auch die gesteuerte Parallelsierung nicht weiter betrachtet.

6 Schlussbetrachtung

6.1 Diskussion des Gesamtergebnis

Eines der Ziele, ein generisches Simulations-Framework für Flugzeuge in C++ zu implementieren wurde dadurch erreicht, dass sämtliche Parameter aus Input-Files eingelesen werden und die Methoden sehr allgemein gehalten wurden. Durch den modularen Aufbau können die einzelnen Module jederzeit um weitere Klassen erweitert werden. Die Simulation selber kann über ein Input-File gesteuert werden, womit unterschiedliche Modelle und Ausbaustufen simuliert werden können, ohne dass der Nutzer den Code selbst verändern muss. In Kapitel 5 wurde die Performance Optimierung der Simulation erläutert. Das Ziel, die Rechenzeit gegenüber Matlab zu verbessern und im Nachgang weiter zu optimieren wurde erfüllt, wie in Abbildung 6.1 zu sehen ist.

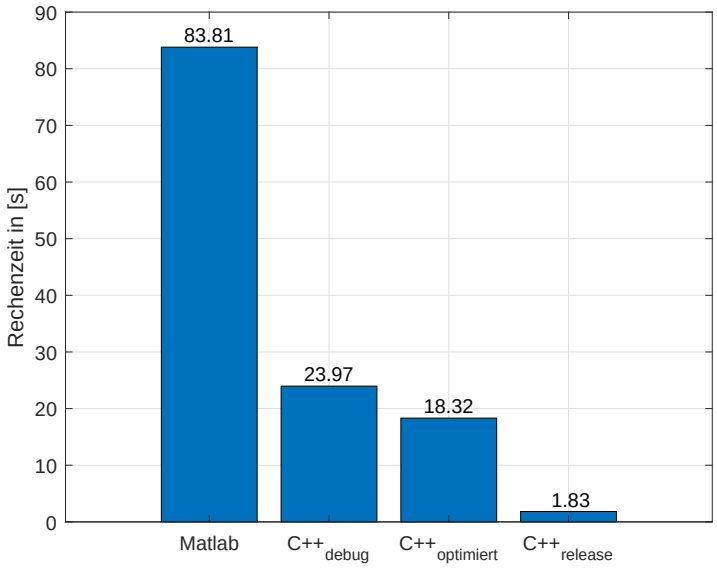

Abbildung 6.1: Vergleich der Rechenzeiten

Durch die Kompilierung als Release, konnte nochmals ein erheblicher Performancegewinn erzielt werden. Quantitative ergibt sich dadurch eine Rechenzeiteinsparung von 97,2% gegenüber Matlab.

6.2 Erweiterungsmöglichkeiten

Das Simulationsframework kann aufgrund des modularen Aufbaus jederzeit um weitere Klassen ergänzt werden. Es wurde bereits erwähnt, dass die höchste Ausbaustufe derzeit noch keine Fehlermodelle der Subsysteme besitzt. Diese und auch andere Erweiterungen könnten in Zukunft implementiert werden. Derzeit werden die Unit- und Modultests nur manuell durchgeführt. Eine Automatisierung der Tests wäre denkbar.

6.3 Fazit

Die in dieser Arbeit gesetzten Ziele wurde mit Erfolg erfüllt. Die Verwendung von Polymorphie wird als sehr effektiv angesehen, um ein Programm modular aufzubauen und um Code zu sparen. Zudem haben sich die in [2] vorgestellten Tools zur effizienten Programmierung bewährt. Die zusätzlich Codedokumentation erwies sich als zusätzliche Last und fordert eine gewisses Maß an Selbstdisziplin. Durch die Durchführung von Unit- und Modultests, konnte die Funktion einiger Methoden nachgewiesen werden. Dennoch ist es sehr schwierig die Tests so zu definieren, um eine große Codeabdeckung zu gewährleisten. Der Leistungsprofiler ist sehr empfehlenswert, um Schwachstellen im Code zu erkennen und nachträglich algorithmisch zu optimieren. Die Möglichkeit der Parallelisierung konnte in dieser Ausarbeitung aufgrund des physikalischen Hintergrunds der Simulation nicht genutzt werden. Zusammenfassend lässt sich sagen, dass dem Nutzer eine funktionsfähige Simulationsumgebung für Flugzeuge zur Verfügung gestellt wird.

Literaturverzeichnis

[1] Jan Olucak. *Modellierung und Implementierung eines Starflüglermodells zur Untersuchung von Außenlasten*. Bachelorarbeit, Technische Hochschule, Ingolstadt, 2017-02-15.

[2] Manuel Kessler. Effizient Programmieren I, Sommersemester 2017.

[3] Manuel Kessler. Effizient Programmieren II, Wintersemester 2017/18.

[4] TuxFamily. eigen, 2018.

[5] Jürgen Wolf. *C++: Das umfassende Handbuch ; [das Lehr- und Nachschlagewerk zu C++ ; inkl. der neuerungen von C++11 ; Sprachgrundlagen, OOP, Multithreading u.v.m ; CD-ROM: Quellcode der Beispiele und Entwicklunsumgebungen]*, volume 2021 of *Galileo Computing*. Galileo Press, Bonn, 3., aktualisierte aufl. edition, 2014.

[6] Microsoft. Grundlagen zum Komponentest, 2018.

[7] Christopher C. Hulbert. matio, 2013.