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 Domänen zu beschreiben. Viele dieser 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 [7] 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 in 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. Mithilfe des semi-empirischen Tools *Data Compendium* (DATCOM) wird die Aerodynamik des Flugzeuges für zuvor definierte Flugzustände berechnet. In einem automatisierten Prozess werden Vorgaberegler 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 von Matlab 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 [7] 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. Aus diesem Grund, sollen über Input-Files 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 [7] implementierte Simulation in C++ übersetzt. Der gesamte Entwicklungsprozess soll sich dabei auf die in [4] und [5] vorgestellten Methoden und Anregungen stützen.

2 Werkzeuge für effiziente Code-Entwicklung

In diesem Kapitel werden die verwendeten "Werkzeuge" kurz erläutert, die bei der Code-Entwicklung unterstützt haben. Dabei wird auf Aspekte aus [4] eingegangen.

2.1 Entwicklungsumgebung und Versionsmanagement

Wie in [4] 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 eingecheckt wurde, wenn das Programm lauffähig war und zuvor getestet wurde [4].

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 den Entwickler. Für die Installation und Nutzung der Simulation wurde eigens ein User Manual erstellt. Es soll dem Nutzer die Bedienung der Simulation erläutern. Die zusätlichen Dokumente dieser Ausarbeitung befinden sich im Ordner *Documents*. In Abschnitt A.2 wird dessen Ordnerstruktur aufgezeigt.

2.3 Open Source Bibliotheken

Nach [5] 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 [7] 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 [11]
matio	C/C++ Bibliothek, die Funktionen bereitstellt, um Daten aus .mat-Files zu lesen und diese zu erstellen [3]

Tabelle 2.1: Übersicht der verwendeten Open Source Bibliotheken

Wie aus Tabelle 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 sei darauf verwiesen, dass zusätzlich noch die Bibliotheken *HDF5* [2] und *zlib* [9] 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 Methoden der Basisklasse werden in der Modulklasse aufgerufen. Die Kindklassen können sowohl von der Basisklasse, als auch von einem anderen Kind abgeleitet werden. Der allgemeine Aufbau eines Moduls wird in Abbildung 3.1 exemplarisch dargestellt.

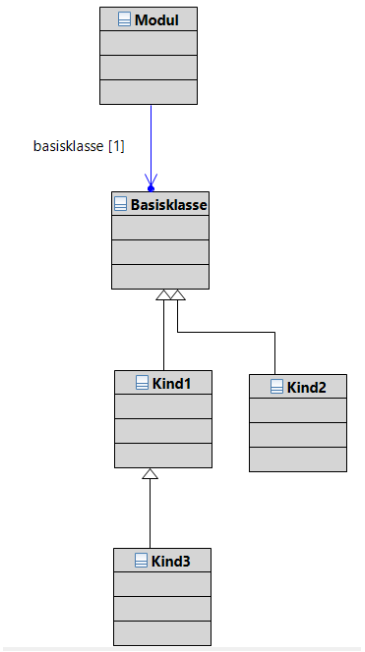

Abbildung 3.1: Allgemeiner Aufbau eins Moduls

Alle Klassen besitzen 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 [13].
Um nun zu gewährleisten, dass das gewünschte Modell des Moduls ausgewählt wird, wird ein ein Basisklassenzeiger verwendet. Der Basisklassenzeiger kann auf jedes Objekt adressiert werden, dass von ihm abgeleitet wurde [13]. Die eigentliche Initialisierung findet in der Modulklasse statt. Dazu wurde dort neben den Standard-Methoden noch die Methode *SelectModuleType* implementiert, wobei *Module* für das jeweilige Modul steht, was im Listing 3.2 zu sehen ist.

```
Basisklasse *BasePtr;

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

switch(Type){
        case 1:
                // Initialisierung der ersten Methode
                BasePtr= new Kindklasse1;
                break;

        case 2:
                // Initialisierung der zweiten Methode
                BasePtr = new Kindklasse2;
                break;
}
```

Listing 3.2: Basisklassenzeiger im Modul.cpp File

Der Nutzer muss lediglich den spezifische Modell-Parameter (*int Type*) im Input-File ändern, um ein anderes Model auszuwählen. Die Funktion *SelectModuleType* wird immer im Konstruktor der jeweiligen Modulklasse aufgerufen. 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 [13].

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.
Soll ein Modul eine neue zusätzliche Klasse (neues Modell) erhalten, so muss der Nutzer lediglich die Klasse selbst implementieren und in *SelectModuleType* des jeweiligen Moduls einen neuen *Case* in der *switch*-Funktion mit der richtigen Initialisierung des Basisklassenzeigers hinzufügen.

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. Die einzelnen Module mit ihren Klassen werden in eigenen Projektordnern hinterlegt.
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	Bewegungsgleichungen des Flugzeuges
Atmosphere	US Standard-Atmosphäre 1976
Autopilot	Vorgaberegler
Engine	Berechnung der Schubkräfte und -momente
GPS	GPS-Modul
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, wie zum Beispiel *Fahrwerk*, wäre denkbar.
Wie bereits in Abschnitt 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 Aktivitätsdiagramme beziehen sich dabei immer auf die *update* Standard-Methoden der Modulklassen und beruhen auf den in [14] aufgezeigten Simulationen. Die niedrigste Ausbaustufe ist eine Simulation mit 3 Freiheitsgraden. Es werden nur die translatorischen Bewegungsgleichungen berücksichtigt. Die Aufruf-Reihenfolge wird in Abbildung 3.2 aufgezeigt.

Abbildung 3.2: Aktivitätsdiagramm der Trajektorie mit 3 Freiheitsgraden (3 Dof)

Eine solche Ausbaustufe könnte beispielsweise schon sehr früh genutzt werden, wenn zum Beispiel noch keine Trägheitsmomente des Flugzeuges bekannt sind.
Die nächst höhere Ausbaustufe besitzt 6 Freiheitsgrade und erbt von der Ausbustufe mit 3 Freiheitsgraden. Der Ablauf ist in Abbildung 3.3 zu sehen.

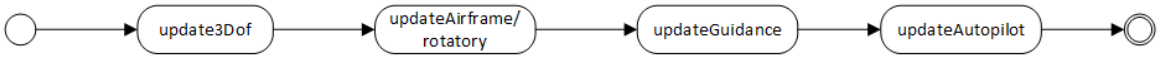

Abbildung 3.3: Aktivitätsdiagramm der Trajektorie mit 6 Freiheitsgraden (6 Dof)

Die Kindklasse wird um die rotatorischen Bewegungsgleichungen ergänzt. Außerdem erhält sie ein Flugführungssystem und einen Vorgaberegler. Eine solche Ausbaustufe könnte beispielsweise genutzt werden, um die Flugregelung auf dem nichtlinearen Modell zu testen und zu beurteilen. In den beiden bisherigen Ausbaustufen wird das Systemverhalten als ideal angesehen. Das bedeutet, dass weder Messfehler durch Sensorik oder Verzögerung durch Aktuatorik berücksichtigt werden.
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 weitere Ausbaustufe benötigt. Diese ist in Abbildung 3.4 zu sehen.

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

Die höchste Ausbaustufe erbt ebenfalls von der ersten Ausbaustufe mit 3 Freiheitsgraden. Anstatt der idealen Integration der Kinematik wird ein Navigations-Modul verwendet. Dort wird im Regelfall eine Navigationslösung erzeugt, in dem die Sensormessungen genutzt werden um durch ein 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. Aktuell sind keine Fehlermodelle implementiert. Für Testzwecke wird das Verhalten dieser Module als fehlerfrei angesehen.

3.3 Gesamtsimulation

Die zuvor beschriebenen Ausbaustufen können genutzt werden, um die Trajektorie 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 3.5 aufgezeigt, wobei angemerkt sei, dass sämtliche Objekte bereits initialisiert wurden und *updateTrajectory* für die vom Nutzer ausgewählte Ausbaustufe der Simulation steht.

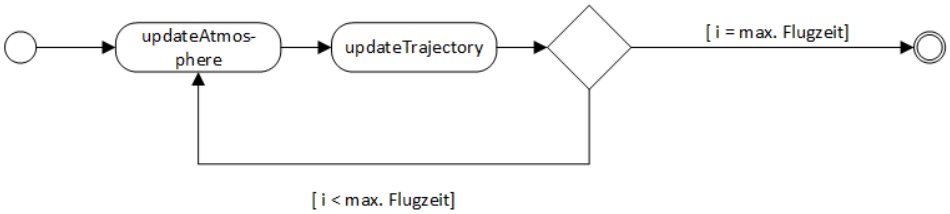

Abbildung 3.5: Aktivitätsdiagramm der Gesamtflugzeug-Simulation

Das *Aircraft*-Modul wird im *Executive* initialisiert und aufgerufen.

4 Durchführung der Implementierung

4.1 Implementierungs-Prozess

Für die Validierung des Frameworks wird die Matlab-Simulation nach [7] 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.
Das Simulationsframework 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, wie in Abbildung 4.1 dargestellt.

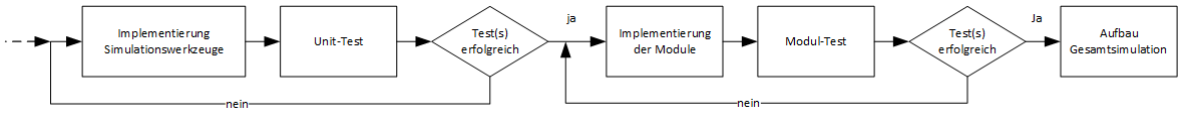

Abbildung 4.1: Flussdiagramm des Implementierungs-Prozess

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 4.1 zusammengefasst.

Funktion/Klasse	**Kurzbeschreibung**
Constants.h	physikalische Konstanten
DataLogger.h	Erzeugung von Text-Files (Output der Simulation)
LinearInterpolation.h	Ein- und zweidimensionale Interpolation
MatFileReader.h	Funktionen zum Einlesen von .mat-File
ODESolver.cpp	Template(s) für numerische Integration
readInData.h	Funktionen zum Einlesen von Parametern aus Text-Files
Transformation.h	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 definiert. Dazu wurde das von Visual Studio bereitgestellte Microsoft-Komponententest-Frameworks für C++ genutzt, um die Tests manuell durchzuführen. 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 initialisiert 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 verglichen. Wird eine Übereinstimmung festgestellt, erhält 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 [11] 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 [3] beruhenden *MatFileReader*-Klasse wird nur der selbst geschrieben Code getestet. Eine Außnahme bei den Unit-Test stellt das Header-File *Constants.h*, bei der lediglich physikalische und mathematische Konstanten hinterlegt sind. Die Tests wurden im gleichnamigen Modul implementiert.

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 [7] 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 beispielsweise das *Autopilot* und *Guidance* Modul. Für beide Module werden Parameter von anderen Modulen benötigt z.B. die Flugzustände.
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 4.2 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.
Guidance	Es wird lediglich getestet, dass für den jeweiligen Zeitschritt die richtige Soll-Querbeschleunigung und Trajektorie vorliegt. Das Modul selbst kann erst im Gesamtsystemtest vollständig getestet werden, da die jeweiligen Flugzustände benötigt werden.
Trajectory	Der Trajectory-Modul Test testet die korrekte Berechnung der Flugbahn. Es werden die Parameter der Matlab Simulation genutzt.

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. Wie bereits weiter oben erwähnt waren teilweise Modul-Tests nicht oder nur bedingt möglich. Dort wurde ebenfalls *Code Reading* als Testmethode verwendet. Es ist offensichtlich, dass das Trajektorien-Modul erst getestet werden konnte, nachdem die dafür benötigten Module bereits erfolgreich getestet wurden. Wie auch bei den Unit-Tests wurden die Modul-Tests in einem eigenen Modul implementiert. Die Überprüfung der Tests erfolgte rein qualitativ und die Ergebnisse wurden im Ordner *Test Results* hinterlegt. Dies dient als Referenz für zukünftige Änderungen.

4.4 Validierung des Simulations-Framework

Abschluss des Implementierung stellt der Gesamtsystemtest des Simulations-Framework. Dazu wird das *Aircraft* Modul mit den Simulationsparametern nach [7] initialisiert. Ziel von [7] war es, den Vorgaberegler zu testen. Dem Vorgaberegler wurden Querbeschleunigungen für jeden Zeitschritt als Soll-Wert vorgegeben. Ziel war es, dass das Flugzeug den Querbeschleunigungen folgt.
In Abbildung 4.2 sieht man, wie 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. Die Trimmpunkte werden durch Linearisierung gefunden, was einen Modellfehler mit sich zieht.

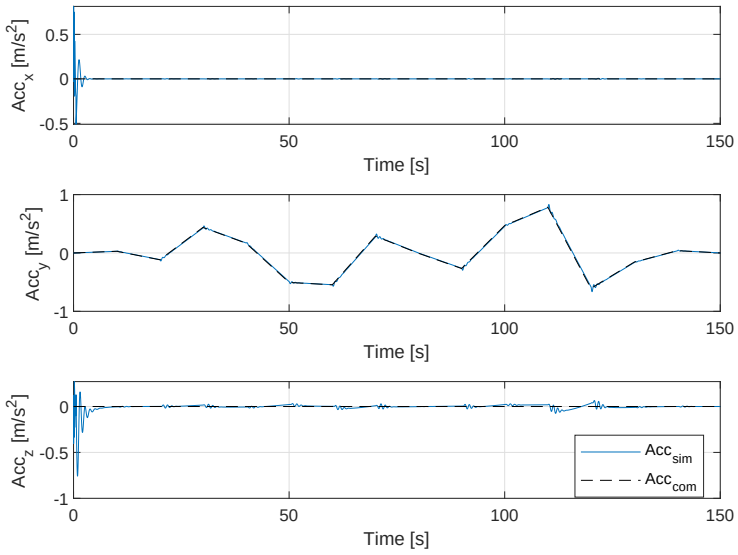

Abbildung 4.2: Validierung der Simulation

Da kein Bahnregler implementiert wurde, ist es weniger zielführend die Trajektorie selbst zu betrachten. Weitere Simulationsergebnisse sind im Anhang unter A.1 dargestellt.

Nachfolgend werden die Ergebnisse von Matlab und C++ verglichen.

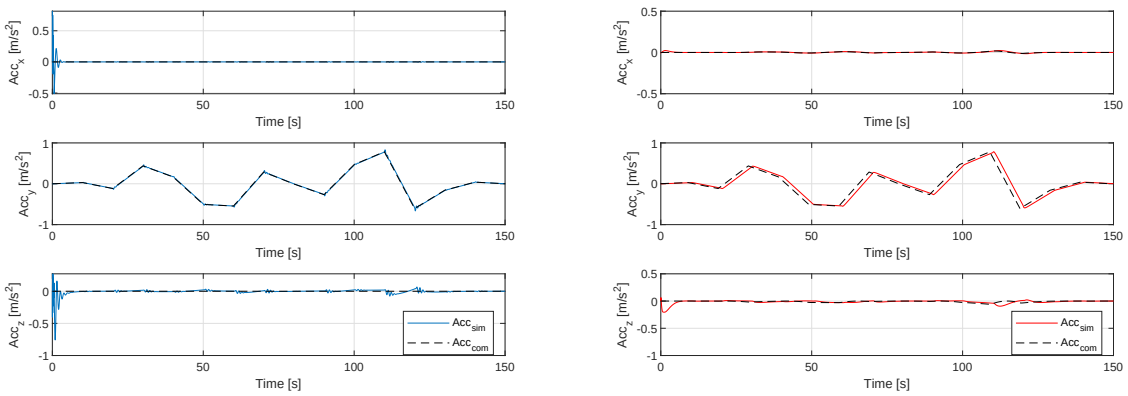

Abbildung 4.3: Vergleich der Simulationsergebnisse, links C++, rechts Matlab

Bei beiden Simulationen sind die Initialisierungsfehler zu erkennen, wobei in C++ die anfänglichen Schwingungen größere Amplituden besitzen. Offensichtlich folgt die Simulation in C++ der Soll-Werten besser als in Matlab. Zudem sind auch numerische Unterschiede zu erkennen. In [10] und [12] wurden diese numerischen Unterschied zwischen Matlab und C/C++ diskutiert. Beruhend auf diesen Quellen ist es zu erwarten, dass die beiden Ergebnisse nicht vollständig übereinstimmen und die Berechnung in C++ genauer ist. Die Simulation nach [7] diente nur der Validierung und nicht der Verifikation. Das Ziel, dass das Flugzeug den Querbeschleunigungsvorgaben folgt, wurde erfüllt. Die Ausbaustufe mit 6 Freiheitsgraden konnte somit validiert werden.
Die höchste Ausbaustufe liefert die gleichen Ergebnisse wie die Ausbaustufe ohne Fehlermodelle. Wie in Abschnitt 3.2 beschrieben, sind nur Methoden implementiert, die ein fehlerfreies Verhalten simulieren. Somit konnte auch diese Ausbaustufe erfolgreich getestet werden. Auf eine Visualisierung wurde an dieser Stelle verzichtet.

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.

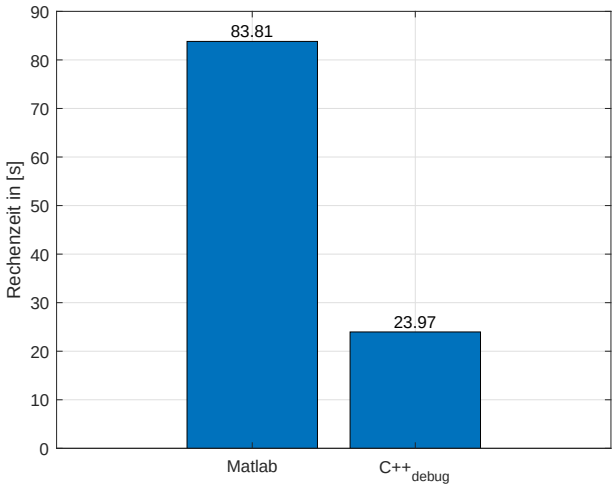

Abbildung 4.4: Direktvergleich Matlab und C++

Betrachtet man das Ergebnis quantitativ 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:

- 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 Leistungsprofilerstellung

Mithilfe des in Visual Studio integrierten Profilerstellungstools 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.

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

Tabelle 5.1: Ergebnisse des Leistungsprofils

Es ist offensichtlich, dass das Erzeugen der Output-File den größten Einfluss auf die Gesamtrechenzeit 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 möglichen Ursachen und Lösungsansätze gegenüber gestellt.

Nr.	Ursachen	Lösungsansatz
1	-nicht benötigte Daten werden geloggt -std::to_string (25,5%)	-Reduktion des Outputs -alternative Methode finden (Benchmarking)
2	Eigen::Library	alternative Bibliotheken
3	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 unter Punkt 1 der Output der einzelnen Module reduziert. Da das Hauptaugenmerk auf der Überprüfung der Flugregelung steht, wurden Parameter vernachlässigt, die wenig Aussagekraft darüber besitzen. Zusätzlich wird im nächsten Abschnitt nach Alternativen für die *std::to_string* Methode gesucht.
Der Flugzustandsregler (Punkt 2) 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 [11] gibt es spezielle Einstellungen für Visual Studio, um die Performance der Bibliothek zu steigern, welche letztlich umgesetzt wurden.
Für die Tabellen-Interpolation (Punkt 3) werden Stützstellen benötigt. Diese Stützstellen zu finden ist mit großem Aufwand verbunden, da dass gesamte Array (Tabelle) durchsucht werden muss. Es wurde versucht, diese Suche zu beschleunigen. Diese algorithmische Optimierung hat sich als äußerst komplex herausgestellt und konnte nicht gelöst werden. Das Ergebnis des ersten Teils der Optimierung ist in Abbildung 5.1 zu sehen.

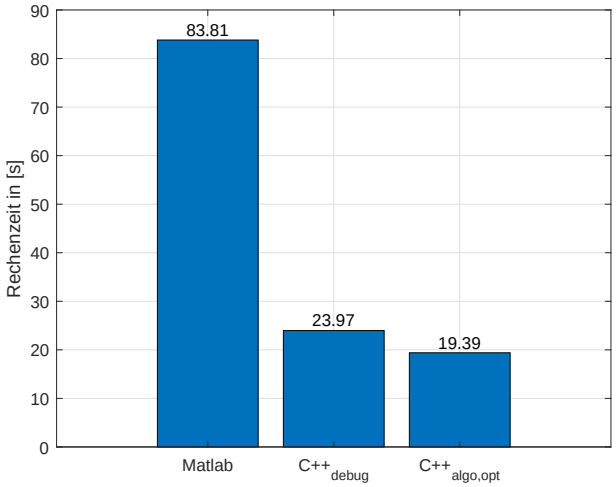

Abbildung 5.1: Gegenüberstellung nach Code Optimierung Teil 1

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 Tabelle 5.2 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 [1] genutzt. Besagte Bibliothek wurde zur Produktivitätssteigerung in C++ erschaffen. Zum Anderen wurde die C++ String Toolkit Library (StrTk) von [8] 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: 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.

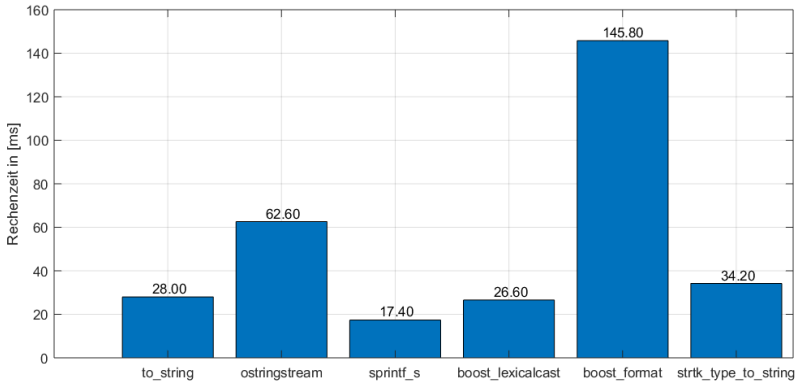

Abbildung 5.2: Ergebnisse des Benchmarkings

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.

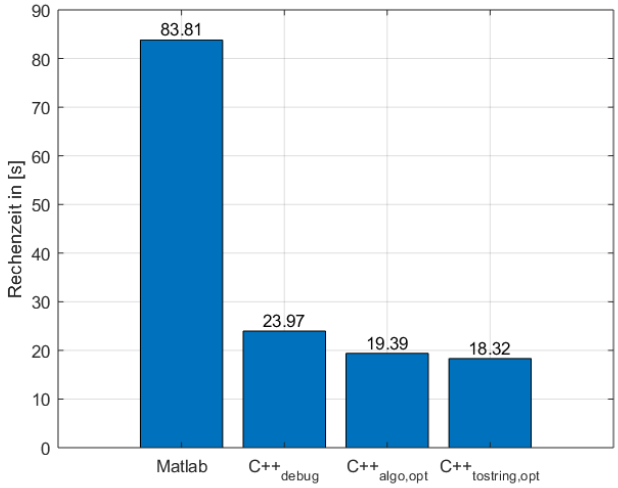

Abbildung 5.3: Gegenüberstellung nach Code Optimierung Teil 2

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 [5]. 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 werden an dieser Stelle die Daten des Rechners, womit die Implementierung durchgeführt wurde, 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 zweiten Code Optimierung aus Abschnitt 5.2.1.

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.

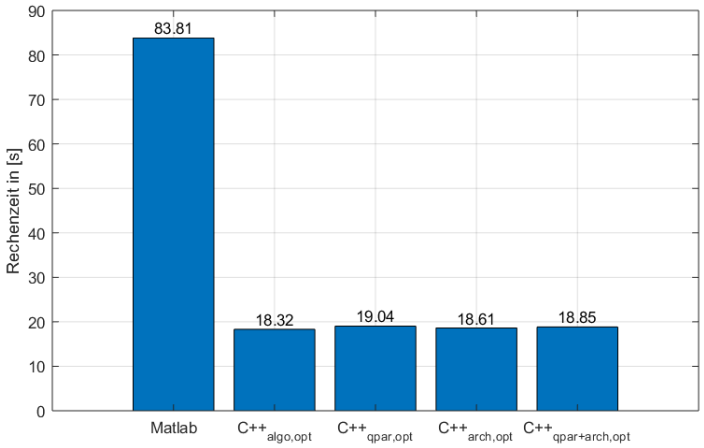

Abbildung 5.4: Ergebnisse der automatischen Parallelisierung und Vektorisierung

Durch die automatische Parallelisierung und Vektorisierung wurde die Performance geringfügig schlechter. Dieses Ergebnis ist nach [5] 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 Zusammenfassung 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 ausgewählt und simuliert werden können, ohne dass der Nutzer den Code selbst verändern muss. Durch das umfangreiche Testen, konnte das Framework für die Simulation mit 6 Freiheitsgraden validiert werden. 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.

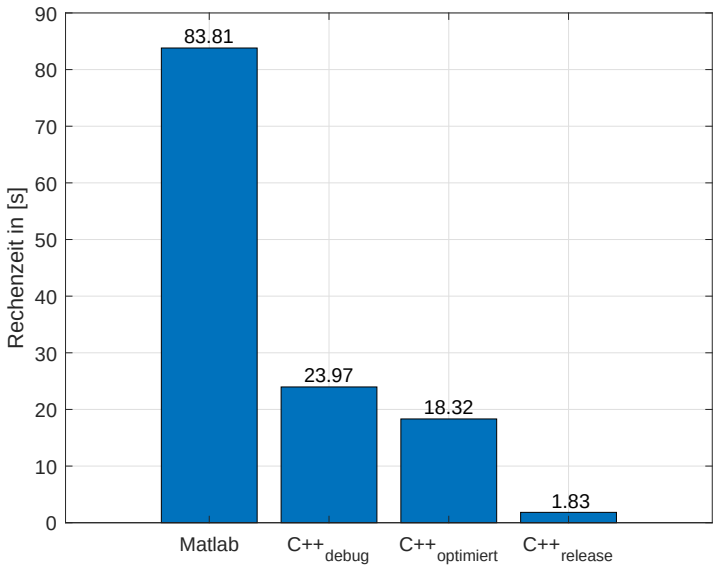

Abbildung 6.1: Vergleich der Rechenzeiten

Durch die Kompilierung als Release, konnte nochmals ein erheblicher Performancegewinn erzielt werden. Quantitative ergibt sich dadurch eine Rechenzeiteinsparung von 97,2% gegenüber Matlab und 90% gegenüber dem optimierten C++ Code.

6.2 Ausblick und Fazit

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 der Module könnten in Zukunft implementiert werden. Derzeit werden die Unit- und Modultests nur manuell durchgeführt. Eine Automatisierung der Tests wäre denkbar.

Die in dieser Arbeit gesetzten Ziele wurde mit Erfolg erfüllt. Die Verwendung von Polymorphie wird als sehr effektiv angesehen, um ein Programm modular aufzubauen. Zudem haben sich die in [4] und [5] 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 und mögliche Fehlerquellen beim Aufbau des Frameworks ausgeschlossen werden. Dennoch ist es sehr schwierig die Tests so zu definieren, um eine große Codeabdeckung zu gewährleisten. Die Leistungsprofilerstellung 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 effiziente Simulationsumgebung für Flugzeuge zur Verfügung gestellt wird.

Anhang A

A.1 Vergleich der Simulationsergebnisse

Es werden die Ergebnisse der Simulationen aus C++ und Matlab gegenübergestellt. In Abbildung A.2 ist klar zu erkennen, dass kein Bahnregler vorhanden ist. In beiden Simulationen folgt das Flugzeug den Querbeschleunigungen. Im Weiteren werden die Zustandsgrößen der Längs- und Seitenbewegung dargestellt. Wie bereits erwähnt sind numerische Unterschiede zu erkennen.

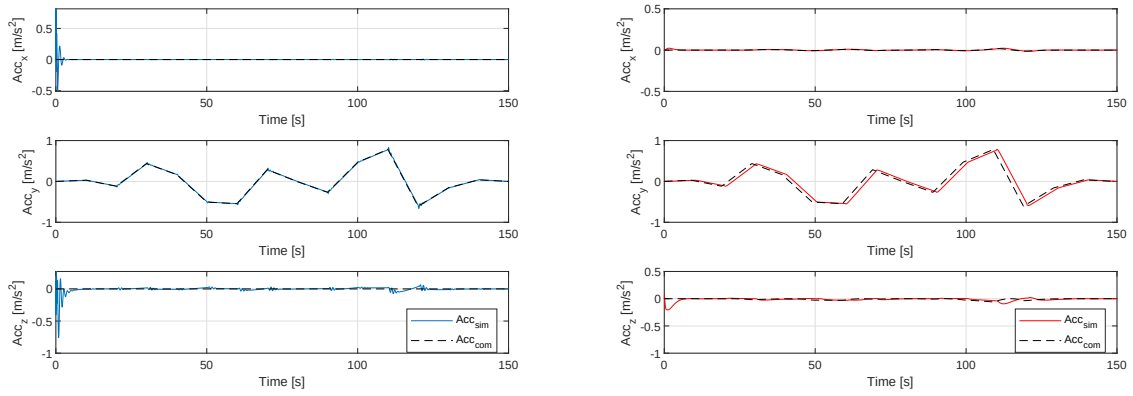

Abbildung A.1: Vergleich Querbeschleunigungen, links C++, rechts Matlab

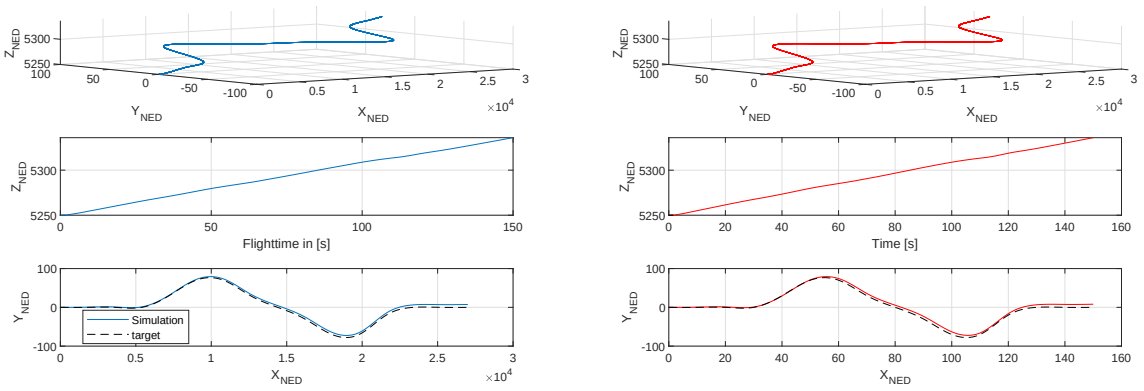

Abbildung A.2: Vergleich der Trajektorien, links C++, rechts Matlab

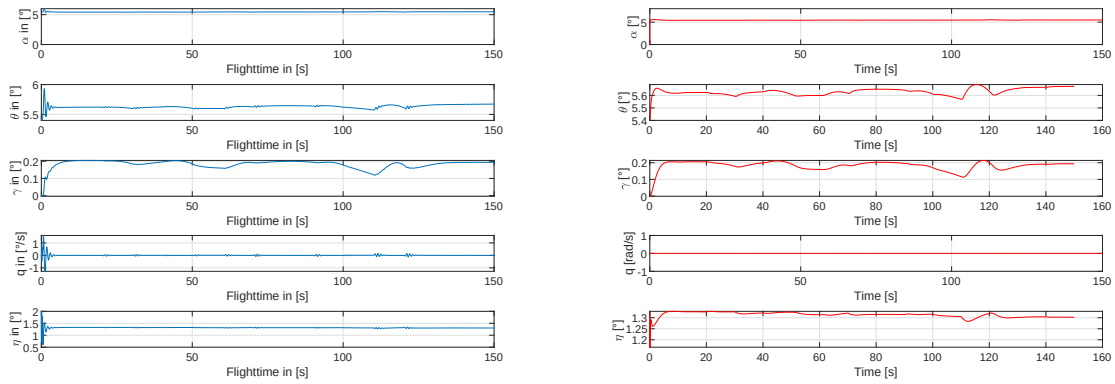

Abbildung A.3: Vergleich der Längsbewegung, links C++, rechts Matlab

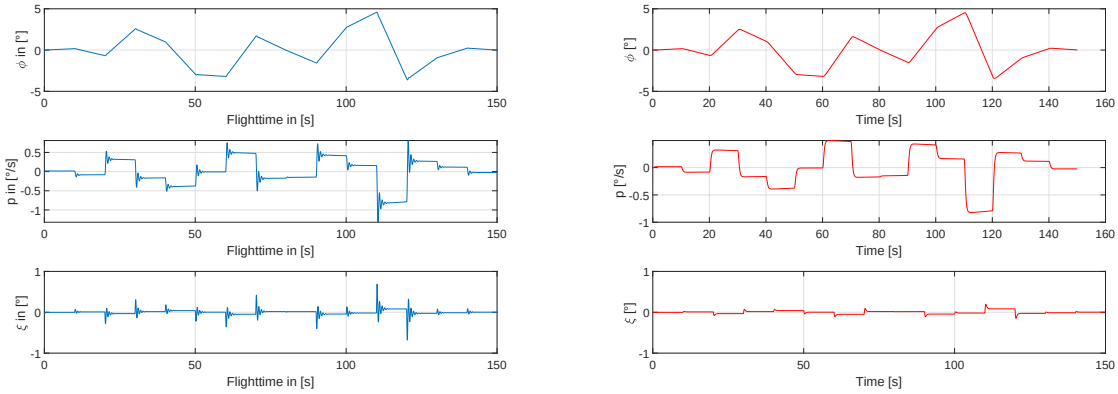

Abbildung A.4: Vergleich der Seitenbewegung (Rollen), links C++, rechts Matlab

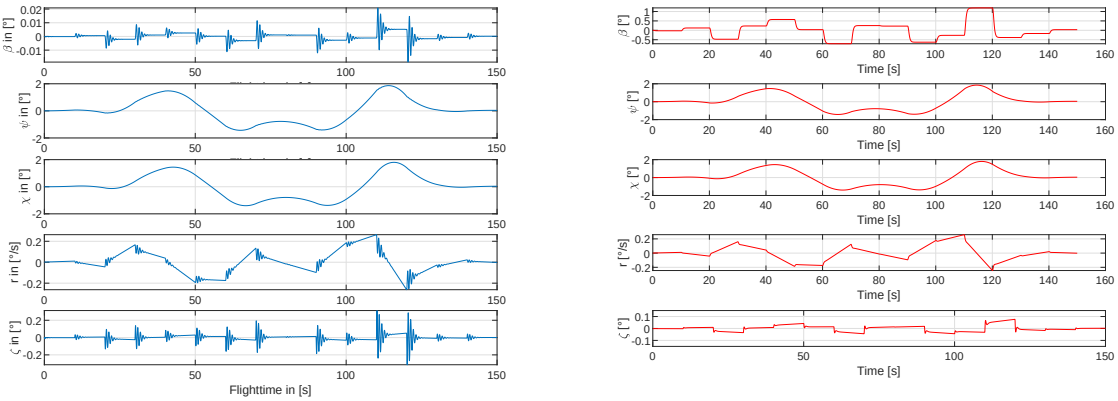

Abbildung A.5: Vergleich der Seitenbewegung (Gieren), links C++, rechts Matlab

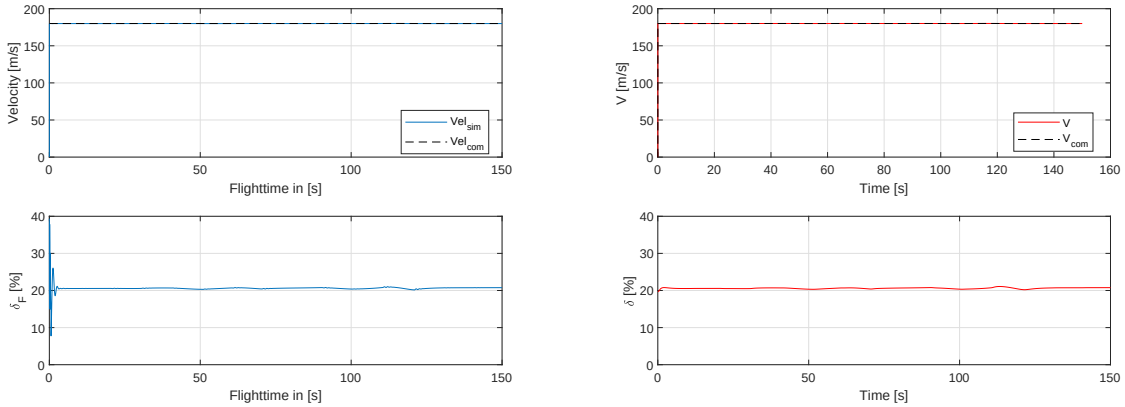

Abbildung A.6: Vergleich der Geschwindigkeit, links C++, rechts Matlab

A.2 Übersicht Dokumente

Neben diesem Abschlussbericht, der das Vorgehen und den Aufbau der Simulation erläutern soll, sind im Ordner *Documents* noch weitere Dokumente hinterlegt. Diese sollen dem Nutzer bei der Bedienung unterstützen.

```
Documents
├── Algorithm Description Documents (Code Dokumentation mit Doxygen)
│   ├── Latex
│   ├── html
│   └── doxyfile
├── User Manuel
└── Abschlussbericht
```

Literaturverzeichnis

[1] C++ Standards Committee's: *Boost C++ Libraries*. https://www.boost.org/. Version: 2018

[2] HDF Group: *HDF5*. https://support.hdfgroup.org/HDF5/. Version: 2018

[3] Hulbert, Christopher C.: *matio*. https://github.com/tbeu/matio. Version: 2013

[4] Kessler, Manuel: *Effizient Programmieren I*. Stuttgart, Sommersemester 2017

[5] Kessler, Manuel: *Effizient Programmieren II*. Stuttgart, Wintersemester 2017/18

[6] Microsoft ; Microsoft (Hrsg.): *Grundlagen zum Komponentest*. https://msdn.microsoft.com/de-de/library/hh694602.aspx#BKMK_Writing_your_tests. Version: 2018

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

[8] Partow, Arash: *C++ String Toolkit Library (StrTk)*. http://www.partow.net/programming/strtk/. Version: 2018

[9] Roelofs, Greg: *zlib*. https://zlib.net/. Version: 2018

[10] Stackoverflow-Thread ; Stackoverflow (Hrsg.): *precision differences in matlab and c++*. https://stackoverflow.com/questions/11151609/precision-differences-in-matlab-and-c. Version: 2012

[11] TuxFamily: *eigen*. http://eigen.tuxfamily.org/index.php?title=Main_Page#Contributing_to_Eigen. Version: 2018

[12] Unbekannt ; Matlabcentral (Hrsg.): *Identical math operations in C++ and matlab differ by ~1% (10^11 in this case!)*. https://de.mathworks.com/matlabcentral/answers/19736-identical-math-operations-in-c-and-matlab-differ-by-1-10-11-in-this-case. Version: 2011

[13] Wolf, Jürgen: *Galileo Computing*. Bd. 2021: *C++: Das umfassende Handbuch*. 3., aktualisierte Aufl. Bonn : Galileo Press, 2014

[14] Zipfel, Peter H.: *Modeling and simulation of aerospace vehicle dynamics*. 2nd ed. Reston, Va. : American Institute of Aeronautics and Astronautics, 2007