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 Schleifen 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.
Aus den meisten Ingenieursanwendungen ist MATLAB nicht mehr wegzudenken. Grund hierfür ist das breite Anwendungsspektrum und die Vielzahl an zusätzlichen Toolboxen. Dabei handelt es sich um eine proprietäre Programmiersprache die auf dem jeweiligen Rechner interpretiert wird. Trotz der vielseitigen Anwendungsmöglichkeiten benötigt MATLAB für komplexere Anwendung eine größere Laufzeit, im Vergleich zu Programmiersprachen, welche direkt in Maschinensprache übersetzt werden. Die Laufzeit ist mitunter einer sehr kritische Größe bei der Nachweisführung und Leistungsrechnung. Zudem ist eine objektorientierte Programmierung nur bedingt möglich, was den Aufbau 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 Simulation in der Lage ist, verschiedene, mit dem Aircraft Designer modellierte Flugzeuge simulieren zu können. Jede Ausbaustufe des Entwurfsprozesses soll simulierbar sein, ohne dass dabei der Code verändert wird. Über Input-Files sollen verschiedene Domänen-Modelle ausgewählt werden können. Ein modularer Aufbau soll zudem eine einfache Erweiterung des Simulation gewährleisten. Um das Simulations-Framework zu verifizieren, wird die in [5] implementierte Simulation in C++ übersetzt. Der gesamte Entwicklungsprozess stützt sich dabei auf die in [2] vorgestellten Methoden und Anregungen.

1.3 Kurzvorstellung Programm Aircraft Designer

In der Arbeit nach [5] 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 verifizieren 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 Zustandsregler 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.

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

2.3 Verwendung von Open Source Bibliotheken

Nach [3] ist es nicht immer notwendig bei gewissen Funktionalitäten immer bei Null 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 [5] 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 [6]
matio	C/C++ Bibliothek, die Funktionen bereitstellt, um Daten aus und in .mat-Files zu lesen/schreiben

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 selber 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. 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 3.1 dargestellt.

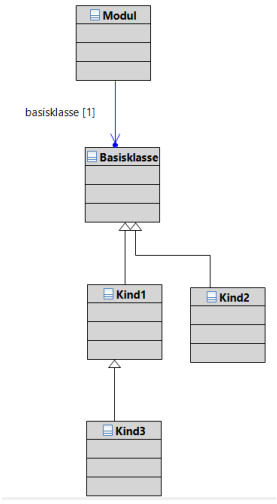

Abbildung 3.1: Allgemeiner Aufbau eins Moduls

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 eigentliche Berechnung/Simulation der spezifischen Domäne für jeden Zeitschritt durchgeführt 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 diese Grund lässt sich die Basisklasse auch als polymorphe Klasse bezeichnen [7].
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. 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.
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	Die aerodynamischen Kräfte und Momente des Flugzeuges werden berechnet
Airframe	Enthält die Bewegungsgleichungen des Flugzeuges
Atmosphere	Hier ist die Standard-Atmosphäre hinterlegt.
Autopilot	Hier ist die Flugregelung implementiert
Engine	Die Triebwerke werden simuliert
Guidance	Stellt die Lenkung zur Verfügung
IMU	Simuliert die Inertialsensoren
Navigation	Hier findet die Berechung der Navigationslösung für das GNC statt.

Tabelle 3.2: Module der generischen Flugzeug-Simulation

An dieser Stelle sei angemerkt, dass es noch weitere Module gibt, 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.
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 3.2 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 3.2 zu sehen.

Abbildung 3.3: Aktivitätsdiagramm der Trajektorie mit 6 Freiheitsgraden

Die Kindklasse wird um die rotatorischen Bewegungsgleichungen ergänzt. Außerdem erhält sie ein Flugführungssystem, das sich aus einer Lenkung und einem Flugzustandsregler zusammensetzt. In den beiden bisherigen Ausbaustufen wird das Systemverhalten als ideal angesehen. Das bedeutet, dass weder Messfehlern 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 3.2 zu sehen.

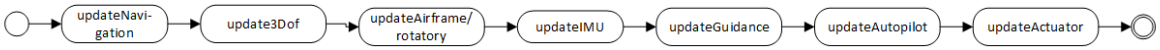

Abbildung 3.4: Aktivitätsdiagramm der Trajektorie mit 6 Freiheitsgraden unter Berücksichtigung von Fehlermodellen

Die höchste Ausbaustufe erbt von der zweiten 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 3.3 aufgezeigt, wobei angemerkt sei, dass sämtliche Objekte bereits initialisiert wurden.

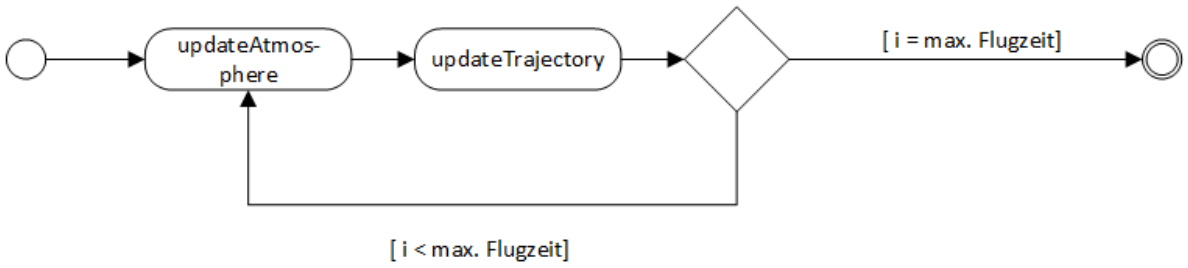

Abbildung 3.5: Aktivitätsdiagramm der Gesamtflugzeug-Simulation

4 Durchführung der Implementierung

4.1 Implementierungs-Prozess

Für die Verifikation des Frameworks wird die Matlab-Simulation nach [5] 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 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 möglich direkt getestet wurde. Erst mit der nachgewiesenen Funktionalität, wurde der nächste Schritt im Prozess durchgeführt. In Abbildung 4.1 wird dieser Prozess dargestellt.

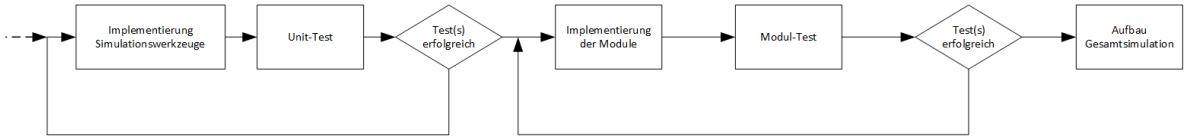

Abbildung 4.1: Flussdiagramm des Implementierungs-Prozess

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 4.2 zusammengefasst.

Funktion/Klasse	**Kurzbeschreibung**
Constants	physikalische Konstanten
DataLogger	Klassen, um Parameter in ein Text-File zu schreiben
LinearInterpolation	Ein- und zweidimensionale Interpolation bereitstellt
MatFileReader	Funktionen zum Einlesen von .mat-File
ODESolver	Templates für numerische Integration
readInData	Funktionen zum Einlesen von Parametern aus Text-Files
Transformation	Matrizen für Koordinaten-Transformation

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 in 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 [4] genutzt. Dazu werden die Tests in die zuvor genannten Abschnitte aufgeteilt. Im einzelnen haben die Abschnitte folgende Bedeutung [4]:

- 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 [6] handelt es sich bei eigen um eine vollständig getestete Bibliothek. Aus diesem Grund wird auf eine Unit-Tests verzichtet. Bei der auf [1] 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 3.1 wurde bereits der Aufbau der Module erläutert und in 3.2 die für die Simulation benötigten Module aufgelistet. Um die Simulation wie im Baukastenprinzip zusammenzusetzen, ist es erforderlich, dass jedes Teil-Modul korrekt funktioniert.
Die Schwierigkeit, die mit den Modul-Tests einhergeht, ist die Definition der Testfälle. Ziel ist es eine größtmögliche Code-Coverage 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. Wie schon erwähnt, wird die 6 Dof-Simulation nach [5] genutzt, um das Simulations-Framework zu testen. Somit liegen nur zu den dort verwendeten Modellen Referenzwerte vor. 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 4.3 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 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.

4.4 Verifikation des Simulations-Framework

5 Optimierung der Simulation

5.1 Grundlegende Rechnerdaten

An dieser Stelle wird der Rechner und genutzte Tools, womit die Implemetierung durchgeführt wurde, kurz aufgezeigt. Die grundlegenden Daten des Rechners sind in Tabelle 5.1 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 5.1: Rechnerdaten

5.2 Vergleich der Rechenzeit Matlab und C++

5.3 Code Optimierung

5.4 Parallelisierung

6 Fazit

Literaturverzeichnis

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

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

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

[4] Microsoft. Grundlagen zum Komponentest, 2018.

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

[6] TuxFamily. eigen, 2018.

[7] 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.