HOCHSCHULE FÜR TECHNIK STUTTGART

DLR

Bachelorarbeit

Entwicklung und Implementierung einer Schnittstelle zur Kopplung einer 3D-CFD-Simulation mit einer 0D-Verbrennungssimulation zur Berücksichtigung chemischer Reaktionen

vorgelegt zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
im Studiengang Informatik

von
Berk Meriç
Matrikelnummer: 1004509

Hochschule für Technik Stuttgart
in Kooperation mit dem
Deutschen Zentrum für Luft- und Raumfahrt e.V.
Institut für Fahrzeugkonzepte

Erstprüfer: Prof. Dr. Marcus Deininger
Zweitprüfer: Nico Kottwitz, M.Sc.

Abgabedatum: 02. Juli 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CFD Computational Fluid Dynamics

DLR Deutsches Zentrum für Luft- und Raumfahrt e.V.

JSON JavaScript Object Notation

XML Extensible Markup Language

CSV Comma-Seperated Value

TCP Transmission Control Protocol

FVM Finite-Volumen-Methode

FEM Finite-Elemente-Methode

FDM Finite-Differenzen-Methode

UDF User Defined Function

Danksagung

An erster Stelle möchte ich meiner Familie danken, die mich während meines gesamten Studiums unterstützt hat. Besonderer Dank gilt meinem Vater, Hüseyin Meriç, der mich seit Beginn meines Bildungsweges ermutigt und gefördert hat. Seine Unterstützung war entscheidend für den Erfolg meines Studiums.

Mein herzlicher Dank gilt Prof. Dr. Marcus Deininger für die Betreuung dieser Arbeit. Seine fachliche Expertise und wertvollen Ratschläge haben maßgeblich zum Gelingen dieser Bachelorarbeit beigetragen. Die konstruktiven Gespräche und sein Vertrauen in meine Arbeit haben mich während des gesamten Prozesses motiviert.

Besonders danken möchte ich meinem Betreuer am Deutschen Zentrum für Luft- und Raumfahrt, Nico Kottwitz, M.Sc. Durch seine umfassende fachliche Expertise gelang es mir, mich schnell in die komplexe Materie der 3D-CFD-Simulationen einzuarbeiten. Seine geduldige Anleitung und fachliche Unterstützung waren für den Erfolg dieser Arbeit unerlässlich.

Ebenfalls bedanken möchte ich mich bei Dennis Buldt, B.Eng., der mir mit seiner Erfahrung in der 0D-Verbrennungssimulation zur Seite stand. Seine hilfreichen Erklärungen erleichterten mir den Einstieg in dieses spezielle Fachgebiet erheblich.

Allen genannten Personen bin ich für ihre Zeit, ihre Unterstützung und ihr Vertrauen dankbar. Ohne ihre Hilfe wäre diese Arbeit nicht möglich gewesen.

Abstract

The goal of this bachelor thesis is the development and implementation of an interface for coupling a 3D CFD simulation with a 0D combustion simulation. The objective was to combine the detailed flow computation of Ansys Fluent with the fast chemical reaction kinetics of Cantera.

The solution to this matter is based on a client-server architecture, where a UDF in Fluent acts as a client and a Python-based server performs the 0D combustion calculations. Communication is handled via TCP sockets using a JSON-based protocol that enables efficient batch processing of up to 2500 cells per request.

The implementation encompasses both the C-based UDF component and the Python server with multiprocessing support. Special attention was given to system robustness to ensure that errors in the coupling do not compromise the entire 3D CFD simulation.

Initial tests demonstrate that the coupling functions are successful. They enable stable communication between the simulation systems. Batch processing significantly reduces communication overhead, and parallelization on the server side optimally utilizes modern multi-core processors.

The developed solution provides a solid foundation for the coupled simulations and can serve as a basis for future extensions and optimizations.

Disclaimer

In der vorliegenden Arbeit wird darauf verzichtet, bei Personenbezeichnungen sowohl die weibliche als auch die männliche und diverse Form zu nennen. Das generische Maskulinum adressiert alle Leserinnen und Leser und gilt in allen Fällen, in denen dies nicht explizit ausgeschlossen wird, für alle Geschlechter.

1 Einleitung

Diese Bachelorarbeit wurde am Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Fahrzeugkonzepte, in Stuttgart durchgeführt und beschäftigt sich mit der Entwicklung und Implementierung einer Schnittstelle zur Kopplung einer 3D-Computational Fluid Dynamics (CFD)-Simulation mit einer 0D-Verbrennungssimulation zur Berücksichtigung chemischer Reaktionen.

Das DLR ist eine der größten Forschungseinrichtungen Europas auf den Gebieten Luftfahrt, Raumfahrt, Energie, Verkehr, Sicherheit und Digitalisierung (German Aerospace Center, 2025). Das Institut für Fahrzeugkonzepte entwickelt Technologien für umweltfreundliche Fahrzeugantriebe und nachhaltige Mobilität (Schneider, 2021). Im Rahmen der Forschungsaktivitäten zur Entwicklung alternativer Kraftstoffe und optimierter Verbrennungsverfahren entstand die Notwendigkeit, präzise chemische Reaktionsmodelle in bestehende 3D-CFD-Simulationen zu integrieren.

Die vorliegende Arbeit entwickelt eine Kopplungsschnittstelle zwischen Ansys Fluent als 3D-CFD-Simulationssoftware (ANSYS Inc., 2025) und Cantera als spezialisiertem Werkzeug für 0D-Verbrennungssimulationen (Goodwin et al., 2018). Ziel ist es, die detaillierte Strömungsberechnung von Fluent mit den präzisen chemischen Reaktionsmodellen von Cantera zu kombinieren.

1.1 Zielsetzung und Ergebnisse

Das Ziel dieser Arbeit war die Entwicklung und Implementierung einer robusten Schnittstelle zur bidirektionalen Kopplung von ANSYS Fluent mit Cantera. Dabei wurden vier verschiedene Lösungsansätze entwickelt und systematisch evaluiert:

Ansatz 1 nutzt PyFluent (PyFluent Developers, 2025) für die externe Steuerung beider Simulationssysteme. Die 3D-CFD-Simulation und die 0D-Verbrennungssimulation werden von einer übergeordneten Python-Anwendung kontrolliert, ohne dass Ansys Fluent manuell geöffnet werden muss.

Ansatz 2 verwendet eine UDF in Fluent, der über eine Socket-Kommunikation mit einem externen Python-Server kommuniziert. Der Server beinhaltet die 0D-Verbrennungssimulation.

Ansatz 3 verwendet ebenfalls eine UDF. Er tauscht jedoch Daten über Dateien aus,

statt über Netzwerkkommunikation.

Ansatz 4 implementiert die Kopplung direkt in der Fluent-UDF über die Python C API (Python Software Foundation, 2025). Der Python-Code wird dabei in die C-basierte UDF eingebettet und ermöglicht eine nahtlose Integration ohne externe Abhängigkeiten.

Es wurde als Lösung Ansatz 2 gewählt. Die entwickelte Lösung demonstriert erfolgreich die Machbarkeit der Kopplung und zeigt messbare Verbesserungen in der Berechnungseffizienz. Das implementierte System ermöglicht den Austausch von Druck-, Temperatur-, Volumen- und Speziesdaten zwischen beiden Simulationsumgebungen in Echtzeit.

1.2 Aufbau der Arbeit

Diese Arbeit gliedert sich in acht Kapitel, die den Leser durch die Entwicklung und Implementierung der Kopplungsschnittstelle führen:

Kapitel 2 beschreibt die Motivation der Arbeit und definiert die Problemstellung.

Kapitel 3 etabliert die theoretischen Grundlagen von 3D-CFD-Simulationen, 0D-Verbrennungssimulationen und Schnittstellenkonzepten. Es werden die relevanten Konzepte für die Kopplung von der beiden Simulationen erläutert.

Kapitel 4 beschreibt die verwendete Methodik, definiert funktionale und nicht-funktionale Anforderungen und stellt die Lösungsansätze vor.

Kapitel 5 stellt den detaillierten Systementwurf dar, einschließlich der Architektur, Designprinzipien und Komponentenstruktur.

Kapitel 6 dokumentiert die praktische Implementierung der gewählten Lösung mit Code-Beispielen und Integrationsdetails.

Kapitel 7 evaluiert die Ergebnisse durch Tests.

Kapitel 8 dient zum Zusammenfassen der Erkenntnisse und gibt einen Ausblick auf Entwicklungsmöglichkeiten.

2 Motivation und Problemstellung

Die heutigen Motorenentwicklung setzt oft Simulationen ein, um Prozesse zu verstehen, ohne teure Experimente durchführen zu müssen. Besonders die 3D-CFD-Simulation mit Ansys Fluent (ANSYS Inc., 2023b) ist dabei sehr nützlich. Sie erlaubt es, durch ein Rechennetz die Strömungen und Wärmeübertragung im Verbrennungsraum genau darzustellen (Versteeg & Malalasekera, 1995). Auf der anderen Seite kann man mit der 0D-Verbrennungssimulation, die in Python mit Cantera umgesetzt wurde, chemische Reaktionen zwar vereinfacht, aber dafür schnell berechnen (Cantera Developers, 2023).

Ansys kann zwar Verbrennungssimulationen einbauen, stößt aber bei der analytischen Berechnung des chemischen Gleichgewichts an Grenzen. Bereits bei der Verbrennung von Wasserstoff mit Sauerstoff entstehen etwa 20 Reaktionen, die in jedem Zeitschritt berechnet werden müssen. Bei komplexeren Kraftstoffen wie Methan mit Luft können mehrere hundert Reaktionen in jedem Zeitschritt in jeder Zelle auftreten. Sowohl Ansys als auch Cantera nutzen daher vereinfachte Modelle mit Look-up-Tables, die anhand des aktuellen Zustands (Temperatur, Druck, Spezies) nach statistischer Wahrscheinlichkeit die entstehenden Spezies bestimmen.

Der entscheidende Vorteil von Cantera liegt darin, dass es Zugriff auf spezialisierte Verbrennungsmodelle ermöglicht, die in Ansys nicht verfügbar sind. Durch die Zusammenarbeit mit dem Institut für Verbrennungstechnik werden reduzierte Aromatenmodelle für neue Kraftstoffe entwickelt, insbesondere für E-Fuels. Diese Look-up-Tables für neue Verbrennungsprozesse existieren in Ansys noch nicht. Die Alternative wären komplizierte, zeitaufwändige Equilibrium-Berechnungen in jeder einzelnen CFD-Zelle. Die Integration von Cantera ermöglicht es, die reduzierten Algorithmen der DLR-Institute zu nutzen und verspricht eine erhebliche Beschleunigung der Rechenleistung.

Die Hauptidee der Arbeit besteht darin, die Stärken beider Simulationsarten zu kombinieren. Dafür wird eine Schnittstelle entwickelt, die nach jedem Zeitschritt der 3D-CFD-Simulation die wichtigen Daten wie Druck und Temperatur an die 0D-Verbrennungssimulation weitergibt. Die 0D-Verbrennungssimulation verarbeitet diese Daten dann und berechnet, wie die Verbrennung abläuft und welche chemischen Reaktionen stattfinden. Diese Ergebnisse gehen dann zurück an Fluent, damit die 3D-CFD-Simulation mit den aktualisierten Werten weiterlaufen kann.

Um den Ablauf der Schnittstelle zu visualisieren, wurde ein vorläufiges Datenflussdiagramm erstellt, das den Prozess nach einem Zeitschritt verdeutlicht:

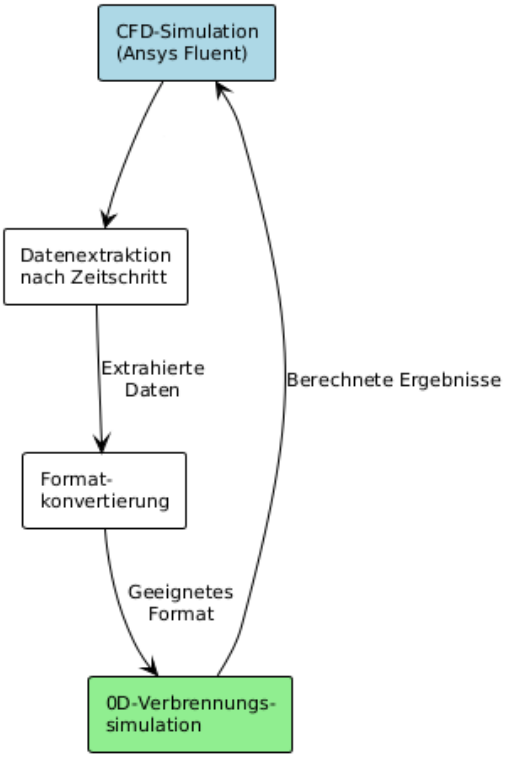

Abbildung 2.1: vorläufiges Datenflussdiagramm

Folgende Herausforderungen wurden identifiziert:

- **Synchronisation:** Fluent muss warten und erst dann mit dem nächsten Schritt weitermachen, wenn die Verbrennungssimulation fertig gerechnet hat und die Ergebnisse zurückgeschickt wurden.
- **Datenmanagement:** Der Datenaustausch zwischen den Programmen muss zuverlässig funktionieren und es dürfen keine Daten verloren gehen.
- **Flexibilität und Erweiterbarkeit:** Die entwickelte Lösung soll nicht nur gut funktionieren, sondern auch in Zukunft noch anpassbar sein, falls sich die Anforderungen ändern.

Aber was ist der Grund dieser Arbeit? Es soll durch die Kombination der 3D-CFD-Simulation mit der 0D-Verbrennungssimulation eine bessere Methode gefunden werden, um Verbrennungsprozesse zu simulieren. Das hat praktische Vorteile: Je genauer man simulieren kann, desto besser versteht man, was im Motor passiert. Und das hilft letztendlich, leistungsstärkere Motoren zu entwickeln.

3 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen vorgestellt, die zum Verständnis der Arbeit notwendig sind. Es werden zuerst zwei Begrifflichkeiten betrachtet, die in der Informatik eher ungewöhnlich sind: die 3D-CFD-Simulation und die 0D-Verbrennungssimulation. Aber der Schwerpunkt liegt auf der Kopplung dieser Technologien. Deshalb erfolgt keine zu tiefe Betrachtung dieser Spezialgebiete. Aus softwaretechnischer Perspektive sind vor allem die verschiedenen Schnittstellenkonzepte relevant, weshalb dieses Thema ausführlicher behandelt wird.

3.1 3D-CFD-Simulation

Computational Fluid Dynamics (CFD), auch numerische Strömungssimulation genannt, ist eine Methode, mit deren Hilfe man das Verhalten von Fluiden – also Flüssigkeiten und Gasen – rechnerisch abbilden kann. CFD nutzt mathematische Modelle und numerische Verfahren (Versteeg & Malalasekera, 1995). Dadurch kann man komplexe Strömungsprozesse analysieren und vorhersagen, ohne aufwendige reale Experimente durchführen zu müssen (Tu et al., 2018). Ziel einer 3D-CFD-Simulation ist es, Strömungsvorgänge besser zu verstehen, zu visualisieren und zukünftige Zustände möglichst genau vorherzusagen.

Der grundlegende Ablauf einer 3D-CFD-Simulation beginnt stets mit der Definition geeigneter Anfangs- und Randbedingungen. Dies sind beispielsweise Druck, Geschwindigkeit, Temperatur oder geometrische Abmessungen. Anschließend wird das Simulationsgebiet in ein Rechengitter – das sogenannte Mesh – unterteilt. Innerhalb jedes einzelnen Elements dieses Gitters werden mithilfe numerischer Verfahren Berechnungen durchgeführt. Die Ergebnisse dieser Einzelberechnungen ergeben schließlich das gesamte Strömungsfeld (Versteeg & Malalasekera, 1995).

Die mathematische Basis für solche 3D-CFD-Simulation bilden in der Regel Navier-Stokes-Gleichungen (Versteeg & Malalasekera, 1995). Diese Gleichungen beschreiben die physikalischen Prinzipien der Massen-, Impuls- und Energieerhaltung eines strömenden Fluids (Versteeg & Malalasekera, 1995). Sie bestehen aus komplexen partiellen Differentialgleichungen. Hierbei kommen numerische Verfahren zum Einsatz, da diese nur selten analytisch exakt gelöst werden können. Eines der gängigsten Methoden hierfür ist die Finite-Volumen-Methode (FVM) (Versteeg & Malalasekera, 1995). Hier wird der Untersuchungsbereich in diskrete Volumenelemente eingeteilt, um dort die Gleichungen näherungsweise zu lösen (Versteeg & Malalasekera, 1995). Gleichzeitig gibt

es noch weitere numerische Verfahren wie zB. Finite-Elemente-Methode (FEM) oder die Finite-Differenzen-Methode (FDM), auf die hier aber nicht weiter eingegangen wird.

3.1.1 Ansys Fluent als CFD-Simulationswerkzeug

Ansys Fluent ist eine Software für 3D-CFD-Simulationen und ein zentrales Produkt im Portfolio des US-amerikanischen Unternehmens Ansys Inc. Es wurde in den 1980er Jahren eingeführt und später von Ansys im Jahr 2006 übernommen. Seitdem ist Fluent ein Standardwerkzeug in der Strömungssimulation und wird heute in zahlreichen Branchen eingesetzt (ANSYS Inc., 2023b).

Funktionsumfang und Einsatzgebiete

Fluent deckt ein breites Spektrum von Strömungsphänomenen ab und eignet sich für die Simulation komplexer Geometrien und Prozesse. Die Software unterstützt laminare und turbulente Strömungen, Ein- und Mehrphasenströmungen sowie Wärmeübertragung und Verbrennungsprozesse (ANSYS Inc., 2023b). In der Automobilindustrie wird Fluent häufig zur Optimierung der Motoreneffizienz eingesetzt.

Schnittstellen und Erweiterbarkeit

Ansys Fluent ist durch verschiedene Schnittstellen erweiterbar. (ANSYS, Inc., 2023a). Für die Kopplung mit der 0D-Verbrennungssimulation sind folgende Möglichkeiten besonders relevant:

UDF UDFs sind in C geschriebene Funktionen, die über spezielle Makros in den Simulationsprozess eingebunden werden. Für diese Arbeit ist das Makro DEFINE EXECUTE AT END wichtig, das am Ende jedes Zeitschritts einer transienten Simulation ausgeführt wird (ANSYS, Inc., 2023a). Es ermöglicht in Kombination mit anderen Makros die Extraktion von Daten nach einem vollständigen Zeitschritt und die anschließende Kommunikation mit externen Prozessen (ANSYS, Inc., 2023a). UDFs haben direkten Zugriff auf interne Datenstrukturen des Fluent-Solvers, was die Auswertung und Modifikation von Zellwerten wie Druck, Temperatur und Zellvolumen ermöglicht (ANSYS, Inc., 2023a). Die Integration von UDFs in Fluent kann entweder als interpretierter Code oder als kompilierte Shared Library erfolgen (ANSYS, Inc., 2023a).

PyFluent In neueren Versionen von Ansys wurde die Python-Bibliothek PyFluent eingeführt. Durch die Bibliothek wird es möglich, eine Simulation vollständig über ein Python-Programm zu steuern. Der direkte Zugriff auf interne Datenstrukturen des Fluent-Solvers geht nicht verloren. (PyFluent Developers, 2025).

Journal-Dateien und Scheme-Programmierung Neben UDFs und PyFluent bietet

Fluent weitere Erweiterungsmöglichkeiten:

- **Journal-Dateien**: Textdateien, die eine Aufzeichnung von Fluent-Befehlen enthalten und zur Automatisierung von Abläufen verwendet werden können (ANSYS, Inc., 2023a)
- **Scheme-Programmierung**: Fluent unterstützt die Sprache Scheme, ein Lisp-Dialekt, der tiefgreifende Anpassungen der Benutzeroberfläche und der Funktionalität ermöglicht (ANSYS, Inc., 2023a)

Zeitliche Betrachtung in 3D-CFD-Simulationen und Kopplungsrelevanz

Um eine Kopplungslösung finden zu können, ist es wichtig zu verstehen, wie die zeitliche Struktur der 3D-CFD-Berechnung aufgebaut ist. Ein Zeitschritt in einer transienten 3D-CFD-Simulation umfasst:

- Die simultane Berechnung von Erhaltungsgleichungen für alle Zellen im Berechnungsgitter
- Mehrere innere Iterationen zur Konvergenz der Gleichungslöser
- Aktualisierung aller physikalischen Felder (Druck, Geschwindigkeit, Temperatur etc.)

Am Ende eines Zeitschritts liegt ein physikalisch konsistenter Zustand vor (ANSYS Inc., 2023b), der als Ausgangspunkt für Kopplungen mit anderen Simulationen dienen kann.

Die beschriebenen Erweiterungen von Ansys Fluent bieten also mehrere Möglichkeiten, um das Simulationsverhalten anzupassen bzw. zu erweitern, sowie die Integration mit externen Prozessen zu erleichtern.

Neben diesen positiven Aspekten gibt es zudem auch eine Schwäche. Fluent hat zwar grundlegende Verbrennungsmodelle (ANSYS Inc., 2023b), aber für detaillierte Reaktionsmechanismen kann es an seine Grenzen stoßen. Hier kommt ein Tool für effiziente Berechnung komplexer Reaktionskinetik ins Spiel: Eine 0D-Verbrennungssimulation mit Cantera.

3.2 0D-Verbrennungssimulation

Die 0D-Verbrennungssimulation ist ein vereinfachtes numerisches Modell zur Beschreibung thermodynamischer Prozesse im Verbrennungsraum, bei dem räumliche Unterschiede vernachlässigt werden (Cantera Developers, 2023). Der gesamte Brennraum wird dabei als einzelner Punkt betrachtet, was die Simulation zeitlicher Veränderungen

von Zustandsgrößen wie Druck, Temperatur und Gemischzusammensetzung ermöglicht (Cantera Developers, 2023).

Grundlagen und Anwendung

Die 0D-Methode geht von der Homogenität aller Zustandsgrößen im Verbrennungsraum aus (Cantera Developers, 2023). Lokale Effekte wie Temperatur- oder Konzentrationsunterschiede bleiben unberücksichtigt. Trotzdem bieten 0D-Modelle wertvolle Erkenntnisse über das thermodynamische Verhalten. Sie dienen auch oft als Ausgangspunkt für komplexere Simulationen. Sie werden besonders in frühen Entwicklungsphasen von Motoren eingesetzt und finden Anwendung in Echtzeitsimulationen.

Modellierung von Verbrennungsprozessen

Bei der 0D-Verbrennungssimulation werden chemische Reaktionen durch detaillierte Reaktionsmechanismen abgebildet, die den Umwandlungsprozess von Kraftstoff und Oxidator zu Verbrennungsprodukten beschreiben (Cantera Developers, 2023). Die Berechnungen umfassen Zündverzugszeiten, Wärmefreisetzung und Emissionsbildung. Durch die Verbindung mit thermodynamischen Zustandsgleichungen lassen sich Druck- und Temperaturentwicklungen im Zeitverlauf nachvollziehen (Cantera Developers, 2023).

Vorteile und Grenzen Der wesentliche Vorteil der 0D-Verbrennungssimulation liegt in ihrer unkomplizierten Handhabung und dem geringen Rechenaufwand, was schnelle Simulationsdurchläufe ermöglicht (Cantera Developers, 2023). Die Ergebnisse fallen allerdings weniger detailliert aus und können aufgrund der Vernachlässigung räumlicher Effekte keine lokalen Phänomene abbilden (Cantera Developers, 2023). Daher eignen sich diese Modelle vor allem dann, wenn das übergeordnete Verhalten des Verbrennungsprozesses im Mittelpunkt steht und eine hohe Rechengeschwindigkeit gefordert ist.

3.2.1 Cantera

Cantera ist eine Open-Source-Softwarebibliothek, die sich auf thermodynamische Zustandsberechnungen, chemische Kinetik und Transportprozesse spezialisiert hat (Cantera Developers, 2023). Sie wurde ursprünglich am California Institute of Technology entwickelt und wird heute von einer aktiven Entwicklergemeinschaft betreut (Wikipedia-Autoren, 2009).

Funktionsumfang und Relevanz für die Kopplung

Cantera bietet umfassende Funktionalitäten für die Modellierung chemischer Reaktionssysteme mit Schwerpunkt auf Verbrennungsprozessen (Cantera Developers, 2023). Zu den Kernfunktionen zählen die Berechnung thermodynamischer Zustandsgrößen, die Simulation chemischer Gleichgewichte und Reaktionskinetik sowie die Modellierung verschiedener Reaktortypen (Cantera Developers, 2023). Die Software wird in vielfältigen Anwendungsbereichen eingesetzt, darunter Verbrennung, Detonation, elektrochemische Energieumwandlung und -speicherung, Brennstoffzellen, Batterien und Plasmaprozesse (Cantera Developers, 2023).

Im Gegensatz zu vollständigen CFD-Lösungen wie Ansys Fluent konzentriert sich Cantera auf die detaillierte chemische und thermodynamische Modellierung, ohne die komplexe Strömungsdynamik räumlich aufzulösen (Cantera Developers, 2023). Dies ermöglicht die Verwendung umfangreicher Reaktionsmechanismen bei vertretbarem Rechenaufwand, was für die geplante Kopplung mit der 3D-CFD-Simulation von entscheidender Bedeutung ist.

Softwarearchitektur und Schnittstellen

Cantera wurde in C++ implementiert, bietet jedoch Schnittstellen für verschiedene Programmiersprachen (Cantera Developers, 2023). Die Python-Schnittstelle ist heute am häufigsten verwendet und ermöglicht eine einfache Integration mit dem wissenschaftlichen Python-Ökosystem (PyData, 2025). Die objektorientierte Struktur basiert auf verschiedenen Klassen, die physikalische Entitäten abbilden (Cantera Developers, 2023):

- **Solution**: Repräsentiert eine Mischung von Substanzen
- **Reactor**: Implementiert verschiedene Reaktortypen wie konstantes Volumen oder konstanten Druck
- **ReactorNet**: Verbindet mehrere Reaktoren zu einem Netzwerk

Reaktormodelle

Für die Kopplung mit der CFD-Simulation ist besonders der ideale Gasreaktor (IdealGasReactor) relevant, der einen homogenen Reaktor mit variabler Temperatur und Druck darstellt (Cantera Developers, 2023). Dieser eignet sich gut für die Simulation von Zündvorgängen und Verbrennung in den einzelnen Zellen der CFD-Simulation.

3.3 Schnittstellenkonzepte

Bei der Kopplung unterschiedlicher Systeme stellen Schnittstellen die entscheidende Verbindung dar. Sie definieren, wie Informationen zwischen den Systemen ausgetauscht, interpretiert und verarbeitet werden. Im Kontext dieser Arbeit ist die Betrachtung von Schnittstellenkonzepten essenziell, da sie die Kopplung zwischen der 3D-CFD- und der 0D-Verbrennungssimulation ermöglichen müssen.

Die Wahl geeigneter Datenformate und Kommunikationsprotokolle hat direkten Einfluss auf die Leistungsfähigkeit, Robustheit und Wartbarkeit des Gesamtsystems. Unterschiedliche Datenstrukturen und zeitliche Abläufe müssen miteinander harmonisiert werden. Daher werden in den folgenden Abschnitten Konzepte vorgestellt, die für diese Arbeit von Bedeutung sind.

3.3.1 Datenformate und Kommunikationsprotokolle

Datenformate

Datenformate legen fest, wie Informationen strukturiert, codiert und interpretiert werden. Für die Simulationskopplung müssen oft gegensätzliche Anforderungen erfüllt werden: Einerseits sollten die Formate flexibel genug sein, um komplexe Datenstrukturen abzubilden, andererseits aber auch effizient in der Verarbeitung und Übertragung sein.

Comma-Seperated Value (CSV) ist ein einfaches und etabliertes Format, bei dem jede Zeile einen Datensatz repräsentiert und die Werte durch Kommata getrennt werden. (Shafranovich, 2005) Die Vorteile liegen in der Einfachheit und universellen Unterstützung (PromptCloud, 2024). Nachteile werden spürbar, wenn komplexere Datenstrukturen oder hierarchische Beziehungen dargestellt werden müssen (Knack, 2024). Bei großen Datenmengen macht sich zudem der Overhead durch die textbasierte Darstellung negativ bemerkbar.

Extensible Markup Language (XML) wurde entwickelt, um die Einschränkungen einfacher Formate zu überwinden. XML bietet hohe Flexibilität bei der Darstellung komplexer, hierarchischer Datenstrukturen und ermöglicht durch XML-Schemas verbindliche Strukturdefinitionen und Validierung (Amazon Web Services, 2025).

JavaScript Object Notation (JSON) ist ein textbasiertes Datenformat für den Austausch strukturierter Daten zwischen Anwendungen. Es kann primitive Datentypen wie Strings und Zahlen sowie strukturierte Typen wie Objekte und Arrays darstellen (Bray, 2014). Aufgrund seiner einfachen Syntax und weiten Verbreitung eignet sich JSON gut für die Datenübertragung zwischen der CFD-Simulation und der Verbrennungsberechnung.

Kommunikationsstrategien

Während Datenformate die Struktur der Informationen festlegen, definieren Kommunikationsprotokolle die Regeln für deren Austausch. Sie bestimmen, wie Verbindungen hergestellt, Daten gesendet und empfangen sowie Fehler behandelt werden. Die folgenden Kommunikationsansätze werden kurz vorgestellt: ein spezifisches Socket-basiertes Protokoll und die etablierten Standards REST und SOAP.

Die Socket-basierte Kommunikation ist ein proprietäres Protokoll und bildet das Fundament vieler Kommunikationsansätze. Sockets stellen eine Abstraktion der netzwerkbasierten Kommunikation dar und ermöglichen den Datenaustausch zwischen

Prozessen, unabhängig davon, ob diese auf demselben Rechner oder über ein Netzwerk verteilt laufen (Stevens et al., 2003). Socket-basierte Kommunikation bietet hohe Flexibilität und Effizienz (Stevens et al., 2003), erfordert jedoch Verständnis in der Netzwerkprogrammierung und eine sorgfältige Implementierung der Fehlerbehandlung.

REST (Representational State Transfer) ist ein nicht-proprietäres, also ein vorgegebenes Protokoll und basiert auf HTTP und folgt grundlegenden Prinzipien wie Ressourcenorientierung, Zustandslosigkeit und einer einheitlichen Schnittstelle (Fielding, 2000). Für die Kopplung von Simulationssystemen bietet REST vor allem dann Vorteile, wenn eine lose Kopplung gewünscht ist oder die Simulationskomponenten über ein Netzwerk verteilt sind. Die HTTP-basierte Kommunikation ist einfach zu implementieren und zu debuggen. Einschränkungen ergeben sich bei echtzeitkritischen Anwendungen durch den relativ hohen Overhead des HTTP-Protokolls (Fielding, 2000).

SOAP (Simple Object Access Protocol) ist auch ein vorgegebenes Protokoll, das für den Austausch strukturierter Informationen in verteilten Umgebungen, das vollständig auf XML basiert. Eine SOAP-Nachricht besteht aus einem Envelope-Element, das einen optionalen Header und einen obligatorischen Body-Abschnitt enthält (Gudgin et al., 2007).

3.3.2 Synchronisationsmechanismen

Synchronisationsmechanismen stellen sicher, dass Daten zwischen gekoppelten Simulationen zum richtigen Zeitpunkt und in der korrekten Reihenfolge ausgetauscht werden (Jefferson, 1985). Ihre Aufgabe ist es, den zeitlichen Ablauf der Kommunikation zu koordinieren und dadurch die Konsistenz und Zuverlässigkeit der Simulation zu gewährleisten (Mattern, 1993). Gerade bei der Kopplung unterschiedlicher Simulationen, wie der 3D-CFD-Simulation mit einer 0D-Verbrennungssimulation, spielt Synchronisation eine entscheidende Rolle. Unzureichende oder falsche Synchronisation kann zu inkonsistenten Ergebnissen oder sogar zu erheblichen Fehlern führen (Jefferson, 1985).

Zeitgesteuerte Synchronisation erfolgt nach festgelegten Zeitintervallen (Mattern, 1993). Diese Methode ermöglicht eine genaue zeitliche Steuerung des Datenflusses. Ein Nachteil ist jedoch die eingeschränkte Flexibilität. Vor allem, wenn der Datenaustausch nicht regelmäßig oder vorhersehbar ist.

Ereignisgesteuerte Synchronisation findet nur bei Eintritt von Ereignissen oder nach Auftreten bestimmter Zustände statt (Jefferson, 1985). Dadurch ist dieser Ansatz flexibler, aber es verlangt eine komplexere Steuerung, um z. B. Datenverluste zu verhindern. Die blockierende Socket-Kommunikation ist ein praktisches Beispiel für ereignisgesteuerte Synchronisation.

3.3.3 Fehlerbehandlung und Robustheit

Netzwerkunterbrechungen, fehlerhafte Daten oder Systemausfälle sind keine Seltenheit. Deshalb müssen Schnittstellen mit solchen unerwarteten Situationen umgehen können (Avižienis et al., 2004). Eine gute Fehlerbehandlung muss unerwartete Zustände frühzeitig erkennen, diagnostizieren und korrigieren können. Zudem muss die Schnittstelle robust sein. Die Robustheit ist die Eigenschaft eines Systems, trotz auftretender Fehler weiterhin zuverlässig zu funktionieren (Avižienis et al., 2004). Dabei sind Timeout-Mechanismen oder automatische Wiederverbindungsversuche wichtige Strategien, die zum Einsatz kommen sollten.

4 Methodik und Vorgehensweise

Dieses Kapitel beschreibt die systematische Vorgehensweise zur Entwicklung der Kopplungsschnittstelle zwischen der 3D-CFD-Simulation und der 0D-Verbrennungssimulation. Zunächst werden die funktionalen und nicht-funktionalen Anforderungen an das System definiert und der Datenfluss zwischen den beteiligten Komponenten analysiert. Anschließend werden verschiedene Lösungsansätze entwickelt, bewertet und der geeignetste Ansatz ausgewählt.

4.1 Anforderungs- und Datenflussanalyse

Bevor mit der Entwicklung der Kopplungslösung begonnen werden kann, müssen die grundlegenden Anforderungen an das System klar definiert und der Datenfluss zwischen den beteiligten Modulen analysiert werden.

Funktionale Anforderungen

Die funktionalen Anforderungen decken den Datenaustausch und die Datenintegrität ab. Folgende funktionalen Anforderungen wurden identifiziert:

- Die Extraktion und das Senden relevanter Zustandsdaten (Druck, Temperatur, Zellvolumen und Zusammensetzung) nach jedem Zeitschritt aus der 3D-CFD-Simulation
- Die Durchführung der 0D-Verbrennungssimulation mit den Zustandsdaten aus Fluent
- Die Gewährleistung, dass die 3D-CFD-Simulation erst dann mit dem nächsten Zeitschritt fortfährt, wenn die 0D-Verbrennungssimulation ihre Berechnungen vollständig abgeschlossen und die Ergebnisse zurückgeliefert hat
- Die blockierende Kommunikation zwischen der 3D-CFD-Simulation und der 0D-Verbrennungssimulation
- Das Übertragen der der Ergebnisse zurück an Fluent

Nicht-Funktionale Anforderungen

Neben den funktionalen Anforderungen sind auch nicht-funktionale Anforderungen zu berücksichtigen. Diese decken die Skalierbarkeit und den modulareren Aufbau ab. Folgende nicht-funktionalen Anforderungen wurden identifiziert:

- Die leichte Anpassung an unterschiedlichen Betriebssysteme. Die Kopplung muss sowohl auf Windows als auch auf Linux Umgebungen laufen können.
- Die Beibehaltung der Leistungsfähigkeit bei steigenden Datenmengen
- Die Erweiterbarkeit für verschiedene Reaktormodelle

Datenflussanalyse

Bei der Datenflussanalyse geht es um die Umsetzung des Informationsaustauschs zwischen den Simulationsmodulen. Der angestrebte Ablauf lässt sich in vier wesentliche Schritte unterteilen:

Als erstes erfolgt die Datenextraktion in Fluent, nachdem ein Zeitschritt beendet wurde. Es werden relevante Zustandsdaten wie Druck, Temperatur, Zellvolumen und Zusammensetzung aus der 3D-CFD-Simulation extrahiert. Diese Daten werden dann über eine Schnittstelle in einem standardisierten Format an die 0D-Verbrennungssimulation gesendet.

Die 0D-Verbrennungssimulation empfängt die Daten und führt die notwendigen Berechnungen durch. Die Ergebnisse werden dann zurück an Fluent übertragen. Fluent muss währenddessen blockierend auf die Rückmeldung warten. Der nächste Zeitschritt wird also erst eingeleitet, wenn alle Ergebnisse vorliegen.

Ein grobes Datenflussdiagramm zwischen zwei Zeitschritten wurde in Abbildung 2.1 dargestellt.

4.2 Lösungsansätze

Die Kopplung zwischen der 3D-CFD-Simulation und der 0D-Verbrennungssimulation kann auf verschiedene Arten realisiert werden. Nach einer Analyse wurden vier grundlegende Ansätze identifiziert und evaluiert.

4.2.1 Ansatz 1: PyFluent mit direkter Integration

Dieser Ansatz nutzt die Python-Schnittstelle PyFluent (PyFluent Developers, 2025), um die 0D-Verbrennungssimulation direkt in den Simulationsprozess zu integrieren.

Beschreibung:

- Die gesamte Steuerung erfolgt über Python
- Die 0D-Verbrennungssimulation wird direkt im selben Python-Prozess ausgeführt
- PyFluent dient als API zur Kontrolle und Steuerung der Fluent-Simulation (PyFluent Developers, 2025)

Vorteile:

- Einfache Implementierung, da alles in Python bleibt
- Direkte Datenübergabe ohne Netzwerk- oder Dateikommunikation
- Einheitliche Programmiersprache für beide Simulationskomponenten

Nachteile:

- Erfordert, dass die Ingenieure ihren Workflow ändern
- Nicht ideal für HPC-Umgebungen, wo Fluent typischerweise manuell gestartet wird
- Weniger Integration mit vorhandenen Automatisierungsprozessen

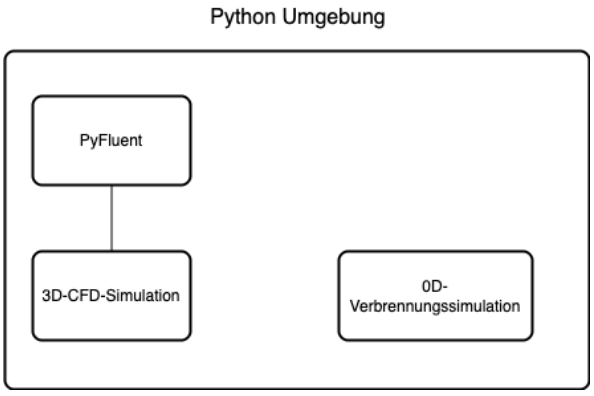

Abbildung 4.1: Aufbau Ansatz 1

Wie in Abbildung 4.1 dargestellt, erfolgt die gesamte Steuerung über Python, wobei PyFluent als zentrale Koordinationsschicht fungiert.

4.2.2 Ansatz 2: UDF mit Socket-Kommunikation

Dieser Ansatz verwendet eine UDF in Fluent, die über Socket-Kommunikation mit einem externen Python-Server kommuniziert.

Beschreibung:

- Für Fluent wird eine UDF in C implementiert, die am Ende jedes Zeitschritts ausgeführt wird
- Die UDF extrahiert relevante Daten und sendet sie über TCP/IP-Sockets an einen externen Server
- Ein Python-Server empfängt die Daten, führt die 0D-Verbrennungssimulation durch und sendet Ergebnisse zurück
- Die UDF wartet auf die Antwort, bevor Fluent mit dem nächsten Zeitschritt fortfährt

Vorteile:

- Minimale Änderung des bestehenden Workflows
- Klare Trennung der Systeme
- Erlaubt verteilte Ausführung (3D-CFD-Simulation und 0D-Verbrennungssimulation könnten auf verschiedenen Systemen laufen)

Nachteile:

- Komplexere Implementierung der Socket-Kommunikation in C (Stevens et al., 2003)
- Potenzieller Overhead durch Netzwerkkommunikation
- Erfordert zusätzliche Fehlerbehandlung für Netzwerkprobleme

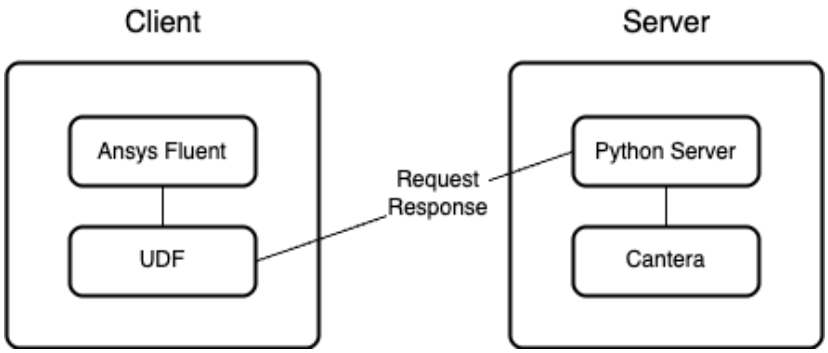

Abbildung 4.2: Aufbau Ansatz 2

4.2.3 Ansatz 3: UDF mit Datei-Kommunikation

Dieser Ansatz nutzt ebenfalls eine UDF in C, tauscht aber die Daten über Dateien anstatt über Netzwerkkommunikation aus.

Beschreibung:

- Die UDF schreibt Daten in Dateien, die von der 0D-Verbrennungssimulation eingelesen werden
- Nach der Berechnung schreibt die 0D-Verbrennungssimulation ihre Ergebnisse in Dateien zurück
- Die UDF wartet auf das Erscheinen der Ausgabedateien, bevor der nächste Zeitschritt beginnt

Vorteile:

- Einfache Implementierung
- Keine Netzwerkkonfiguration nötig
- Gute Nachvollziehbarkeit

Nachteile:

- Gegebenfalls langsamer aufgrund von Festplattenzugriffen
- Erfordert gemeinsamen Dateizugriff zwischen Fluent und der 0D-Verbrennungssimulation.

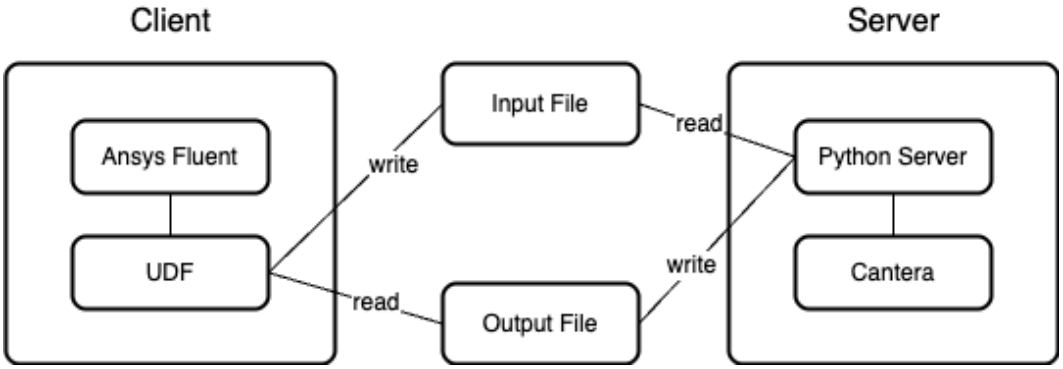

Abbildung 4.3: Aufbau Ansatz 3

4.2.4 Ansatz 4: UDF mit eingebetteter Python-Umgebung

Dieser Ansatz nutzt die Python C-API, um eine Python-Umgebung direkt in die UDF zu integrieren.

Beschreibung:

- Die UDF initialisiert eine Python-Umgebung und führt die 0D-Verbrennungssimulation. direkt aus
- Datenaustausch erfolgt direkt im Speicher, ohne externe Kommunikation
- Alle Berechnungen laufen im selben Prozess

Vorteile:

- Hohe Integration, alles in einem Paket
- Schneller Datenaustausch ohne externe Kommunikation
- Keine Netzwerk- oder Dateikommunikation nötig

Nachteile:

- Komplexe Implementierung der Python C-API
- Erfordert Installation der Python-Umgebung auf den Simulationsrechnern
- Fehler im Python-Code können die gesamte Simulation zum Absturz bringen

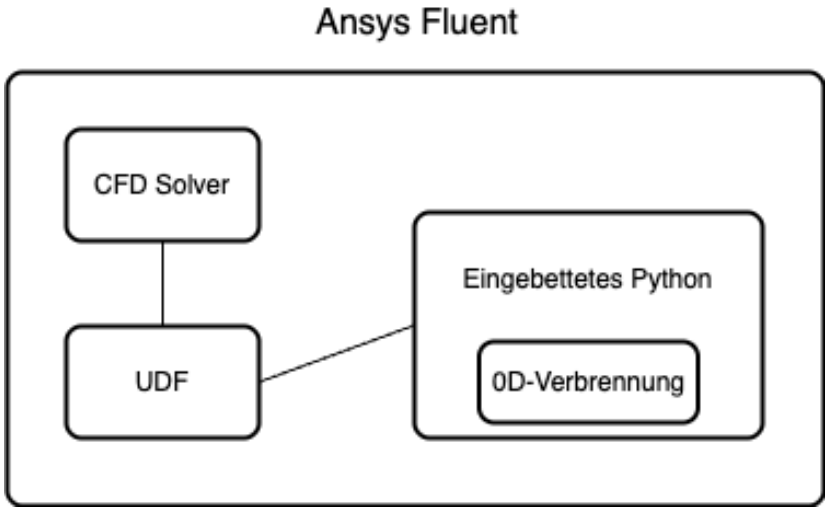

Abbildung 4.4: Aufbau Ansatz 4

4.3 Auswahl und Begründung des gewählten Ansatzes

Die 4 Ansätze wurden prototypisch implementiert und abgewägt. Es wurde für diese Arbeit der **Ansatz 2: UDF mit Socket-Kommunikation** ausgewählt. Die Entscheidung fiel aus mehreren Gründen auf diesen Ansatz, die im Folgenden näher gebracht werden.

4.3.1 Bewertungskriterien

Für die Auswahl wurden die Ansätze anhand gewichteter Kriterien bewertet:

Leistungsbezogene Kriterien (45%)

- Latenz (25%): Durchschnittliche Antwortzeit pro Zeitschritt
- Durchsatz (10%): Anzahl verarbeiteter Anfragen pro Sekunde
- Overhead (10%): Zusätzlicher Ressourcenverbrauch

Qualitätsbezogene Kriterien (35%)

- Zuverlässigkeit (20%): Stabilität und Robustheit des Systems
- Wartbarkeit (15%): Einfachheit der Pflege und Fehlerbehandlung

Implementierungsbezogene Kriterien (20%)

- Implementierungsaufwand (10%): Entwicklungszeit und Komplexität
- Debugging-Möglichkeiten (5%): Nachverfolgbarkeit von Fehlern
- Erweiterbarkeit (5%): Anpassung an zukünftige Anforderungen

Bewertet wurde auf einer Skala von 1 (unzureichend) bis 5 (ausgezeichnet).

Zur objektiven Bewertung wurden prototypische Implementierungen aller vier Ansätze entwickelt. Dabei wurden mit 200 Zeitschritten Leistungsmessungen durchgeführt. Die Messdaten umfassten sowohl C-seitige als auch Python-seitige Zeitmessungen, wobei die C-Messung die gesamte Verarbeitungszeit einschließlich der Python-Berechnung erfasste.

Für die qualitätsbezogenen und implementierungsbezogenen Kriterien erfolgte die Bewertung anhand praktischer Erfahrungen während der Entwicklungsphase sowie der Analyse von Wartbarkeit, Debugging-Möglichkeiten und Erweiterbarkeit der jeweiligen Lösungsansätze.

Bewertungsmatrix

Kategorie	Kriterium	Gewichtung	Datei	Socket	PyFluent	C-API
Leistung	Latenz	25%	2	4	1	4
	Durchsatz	10%	2	4	1	5
	Overhead	10%	3	4	2	4
Qualität	Zuverlässigkeit	20%	4	4	4	3
	Wartbarkeit	15%	5	3	4	2
Implementierung	Aufwand	10%	5	3	4	2
	Debugging	5%	5	4	2	3
	Erweiterbarkeit	5%	3	4	4	4
Bewertung		**100%**	**3.45**	**3.75**	**2.65**	**3.35**

Tabelle 4.1: Gewichtete Bewertung der Kategorien Leistung, Qualität und Implementierung

Der Socket-basierte Ansatz (Ansatz 2) wurde aufgrund der höchsten Gesamtbewertung von 3.75 Punkten ausgewählt. Es bietet eine ausgewogene Balance zwischen Performance, Robustheit und Implementierungsaufwand.

4.3.2 Praktische Umsetzung

In der praktischen Umsetzung wird der Socket-basierte Ansatz folgendermaßen implementiert:

1. **Fluent-seitig**: Eine UDF wird entwickelt, die am Ende jedes Zeitschritts ausgeführt wird (`DEFINE_EXECUTE_AT_END`). Diese UDF extrahiert Daten, wie Druck und Temperatur, öffnet eine Socket-Verbindung zum 0D-Verbrennungssimulationsserver, sendet die Daten und wartet auf die Antwort.

2. **Python-seitig**: Ein Python-Server wird implementiert, der auf eingehende Verbindungen wartet, die empfangenen Daten mit Cantera verarbeitet und die Ergebnisse zurücksendet.

3. **Synchronisation**: Die blockierende Natur der Socket-Kommunikation gewährleistet, dass Fluent erst mit dem nächsten Zeitschritt fortfährt, wenn die 0D-Verbrennungssimulation abgeschlossen ist.

4. **Datenformat**: Für den Datenaustausch wird das JSON-Format verwendet, das eine gute Balance zwischen menschlicher Lesbarkeit und maschineller Verarbeitbarkeit bietet.

4.4 Testkonzept

Ein Testplan wurde entwickelt, um den Betrieb des gewählten Ansatzes zu überprüfen. Der geplante Test umfasst vier Phasen:

4.4.1 Komponententests

In dieser Phase untersuchen wir selektiv jedes einzelne Element der Verbindungslösung:

- Server-Funktionalitätstest: Validierung der Python-Server-Komponente einschließlich Netzwerkkommunikation, JSON-Verarbeitung und Cantera-Integration
- UDF-Strukturtests: Überprüfung der UDF-Implementierung auf Vollständigkeit der Fluent-Makros, Socket-Kommunikation und Datenstrukturen
- Kommunikationstests: Validierung des Datenaustauschs in den Kommunikationsverbindungen zwischen den gekoppelten Komponenten.

4.4.2 Integrationstests

In dieser Phase wird die Kombination verifiziert:

- End-to-End-Kommunikation: Überprüfung des vollständigen Datenflusses von der CFD-Simulation zur 0D-Simulation unter Verwendung des implementierten Protokolls
- Batch-Verarbeitung: Tests der Effizienz und Korrektheit bei verschiedenen Batch-Größen
- Datenintegrität: Sicherstellung der korrekten Übertragung und Verarbeitung von Simulationsdaten

4.4.3 Validierungstests

Diese Tests demonstrieren die praktische Anwendbarkeit:

- Funktionsnachweis: Verifikation der grundlegenden Kopplungsfunktionalität mit CFD-Parametern
- Species-Verarbeitung: Tests verschiedener Verbrennungsmischungen zur Validierung der chemischen Reaktionskinetik

Die Implementierung der Tests erfolgt mit automatisierten Skripten zur Gewährleistung der Reproduzierbarkeit. Aufgrund der Komplexität der Fluent-Integration werden für bestimmte Testaspekte auch manuelle Validierungsverfahren eingesetzt. Die detaillierte Durchführung und Ergebnisse der Testevaluation werden in Kapitel 7 dokumentiert.

5 Systementwurf

Die Entwicklung der Kopplungsarchitektur erforderte eine systematische Herangehensweise, um sowohl die technischen Anforderungen als auch die praktischen Einschränkungen der beiden Simulationsumgebungen zu berücksichtigen. Das folgende Kapitel beschreibt die konzeptionelle Architektur des Systems, bevor in Kapitel 6 auf die konkreten Implementierungsdetails eingegangen wird.

5.1 Architekturüberblick

Bei der Konzeption der Gesamtarchitektur stand die Frage im Vordergrund, wie eine stabile und effiziente Kopplung zwischen zwei grundlegend verschiedenen Simulationssystemen realisiert werden kann. Die Herausforderung bestand darin, eine Lösung zu entwickeln, die beide Systeme optimal nutzt, ohne deren bestehende Funktionalität zu beeinträchtigen.

5.1.1 Grundlegende Designprinzipien

Die Architektur folgt bewährten Software-Engineering-Prinzipien. Das wichtigste Prinzip war die strikte Trennung der Verantwortlichkeiten zwischen den verschiedenen Systemkomponenten.

Separation of Concerns: Die 3D-CFD-Simulation konzentriert sich ausschließlich auf die Strömungsberechnung, während die chemische Reaktionskinetik vollständig von der 0D-Verbrennungsimulation übernommen wird. Diese klare Abgrenzung ermöglicht es, beide Teilsysteme unabhängig voneinander zu optimieren und weiterzuentwickeln.

Loose Coupling: Statt einer engen Integration wurde bewusst eine lose gekoppelte Architektur gewählt. Die beiden Simulationsumgebungen kommunizieren ausschließlich über eine definierte Schnittstelle, wodurch Änderungen an einem System minimale Auswirkungen auf das andere haben.

5.1.2 Client-Server-Architektur

Für die Umsetzung wurde das Client-Server-Muster gewählt, bei dem die UDF als Client und die 0D-Verbrennungssimulation als Server fungiert.

Das Client-Server-Modell bietet eine natürliche Rollenverteilung: Der Client initiiert die Kommunikation genau dann, wenn Berechnungsergebnisse benötigt werden, während der Server bereitsteht und auf Anfragen reagiert. Diese asymmetrische Kommunikation passt gut zu den unterschiedlichen Charakteristika der beiden Simulationssysteme.

Ein wichtiger Vorteil dieser Architektur ist die Möglichkeit der verteilten Ausführung. In der aktuellen Implementierung laufen beide Komponenten auf demselben System. Aber es kann auch bei Bedarf der Server auf einen anderen Rechner ausgelagert werden.

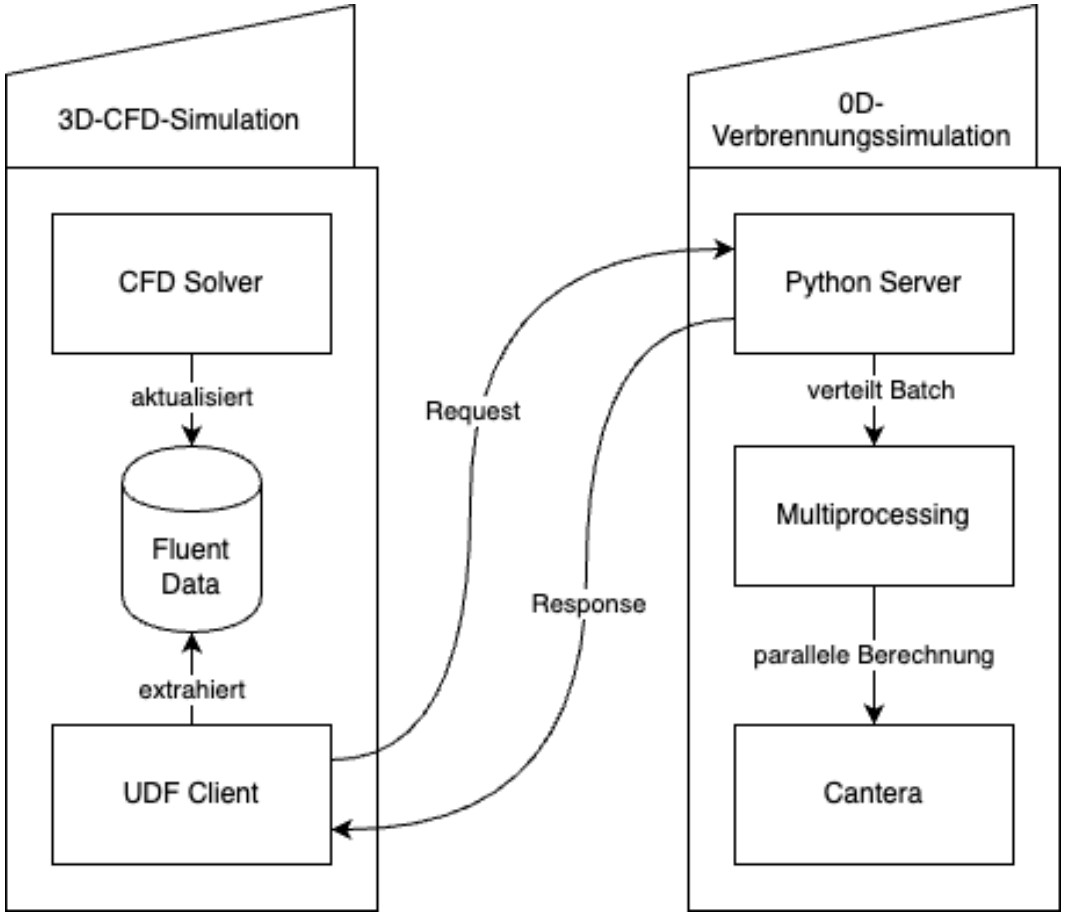

Abbildung 5.1: Komponentendiagramm der Kopplungsarchitektur

Abbildung 5.1 zeigt, dass sich die Gesamtarchitektur in zwei Hauptbereiche aufteilt: die 3D-CFD-Simulation mit der UDF-Komponente in C und der 0D-Verbrennungsimulation. Diese logische Struktur ermöglicht klare Verantwortlichkeiten.

5.2 Komponentenentwurf

Der Entwurf der einzelnen Systemkomponenten orientiert sich an den funktionalen Anforderungen und den Erkenntnissen aus der Lösungsanalyse. Dabei wurde besonderer Wert auf die Schnittstellen zwischen den Komponenten gelegt, um eine saubere Entkopplung zu gewährleisten.

5.2.1 UDF-Client-Architektur

Die UDF-Komponente bildet die Schnittstelle zwischen der 3D-CFD-Simulation und der externen 0D-Verbrennungsberechnung. Ihre Architektur muss sowohl die Integration in Fluent als auch die effiziente Kommunikation mit dem Server ermöglichen.

Der ursprüngliche Entwurf sah eine modulare Struktur mit separaten Komponenten vor (siehe Abbildung 5.2).

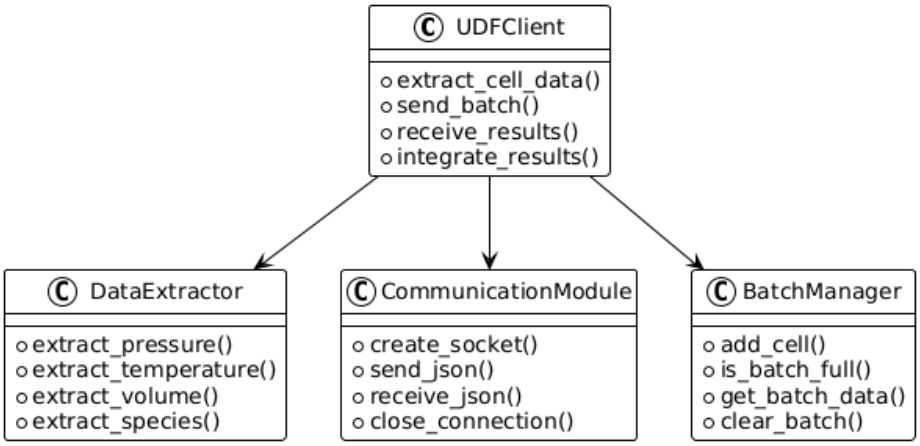

Abbildung 5.2: Ursprünglich geplante Klassenstruktur der UDF-Komponente

Nach eingehender Analyse der Fluent-UDF-Anforderungen und Kompatibilitätsaspekte wurde jedoch eine Standalone-Implementierungsstrategie gewählt. Diese Entscheidung basiert auf technischen Gegebenheiten des Fluent-Build-Systems:

- **Build-System-Limitierungen:** Fluents UDF-Compiler arbeitet mit einem verkettungsbasierten Kompilierungsansatz, der mehrere Dateien zu einer einzigen Shared Library zusammenfasst. Diese Architektur bietet keine echte Modularität, sondern erzeugt lediglich eine pseudo-monolithische Kompilierung (ANSYS, Inc., 2023a)
- **Namespace-Konflikte:** Alle Funktionen aus verschiedenen UDF-Dateien teilen sich denselben globalen Namespace innerhalb der generierten Library. Dies verhindert die erwarteten Kapselungsvorteile einer modularen Struktur (ANSYS, Inc., 2023a).

- **Compiler-Kompatibilität:** Die offiziellen Ansys-Dokumentationen enthalten mehrfache Warnungen vor Komplexitätsproblemen bei der Dateiorganisation. Besonders bei parallelen Ausführungen treten systematische Kompilierungsfehler auf, die sich bei modularen Ansätzen verstärken (ANSYS, Inc., 2023a).

Standalone-Architektur

Die finale UDF-Implementierung organisiert die ursprünglich geplanten Komponenten in logisch getrennte Funktionsbereiche:

Datenextraktion: Verantwortlich für die Extraktion der relevanten Simulationsdaten aus den Fluent-Zellen. Es abstrahiert den Zugriff auf die internen Fluent-Datenstrukturen und stellt eine einheitliche Schnittstelle für die nachgelagerten Funktionsbereiche bereit.

Kommunikationssteuerung: Verwaltet die Socket-Verbindungen zum Server und implementiert das Nachrichtenprotokoll. Es kapselt alle netzwerkspezifischen Details und bietet eine einfache API für die Datenübertragung.

Batch-Management: Ein zentraler Aspekt der UDF-Architektur ist die Batch-Verarbeitung. Um die Kommunikationseffizienz zu optimieren, werden mehrere Zellen zu Batches zusammengefasst, bevor eine Übertragung initiiert wird. Das Batch-Management koordiniert diese Sammlung und gewährleistet die korrekte Zuordnung der Ergebnisse.

Die Standalone-Architektur behält die funktionale Trennung der ursprünglich geplanten Komponenten bei, während sie gleichzeitig die praktischen Anforderungen der Fluent-Integration erfüllt. Diese Designentscheidung gewährleistet sowohl die Wartbarkeit des Systems als auch die zuverlässige Funktion in der Produktionsumgebung.

5.2.2 Server-Architektur

Der Server implementiert die chemische Reaktionsberechnung und muss sowohl einzelne Anfragen als auch Batch-Verarbeitungen effizient handhaben können.

Netzwerkschicht: Die Netzwerkschicht verwaltet die eingehenden Client-Verbindungen und implementiert das Transmission Control Protocol (TCP)-Socket-Interface. Sie abstrahiert die plattformspezifischen Details der Netzwerkkommunikation und stellt eine einheitliche Schnittstelle für die Anwendungslogik bereit.

Request-Handler: Der Request-Handler verarbeitet die eingehenden JSON-Anfragen und koordiniert die Weiterleitung an die entsprechenden Simulationsmodule. Er implementiert auch die Fehlerbehandlung und Datenvalidierung.

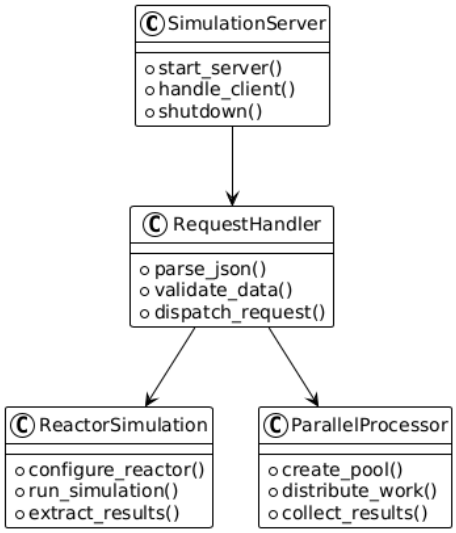

Abbildung 5.3: Klassenstruktur der Server-Komponente

Simulationsmodul: Das Simulationsmodul kapselt die Cantera-Integration und implementiert die eigentliche 0D-Reaktorberechnung. Es abstrahiert die Details der Cantera-API und stellt eine einfache Schnittstelle für die Reaktorsimulation bereit.

Parallelisierungsmanager: Für die effiziente Verarbeitung von Batch-Anfragen implementiert der Parallelisierungsmanager die Verteilung der Einzelberechnungen auf mehrere CPU-Kerne. Er verwaltet den Process Pool und koordiniert die Aggregation der Ergebnisse.

5.2.3 Protokollarchitektur

Das Protokoll basiert auf dem Request-Response-Muster mit synchroner Kommunikation. Diese Architektur gewährleistet die erforderliche zeitliche Synchronisation zwischen den Simulationen und vereinfacht die Implementierung erheblich.

Synchrone Kommunikation: Die UDF wartet blockierend auf die Antwort des Servers, bevor sie mit dem nächsten Zeitschritt fortfährt. Dies gilt nur für transiente Simulationen, was unserem Fall entspricht.

Request-Response-Zyklus: Jede Kommunikation folgt einem klar definierten Ablauf: Request-Erstellung, Übertragung, Server-Verarbeitung, Response-Generierung und Response-Verarbeitung. Dieser strukturierte Ablauf erleichtert die Fehlerbehandlung und das Debugging.

Zustandslosigkeit: Das Protokoll ist zustandslos gestaltet - jede Anfrage enthält alle notwendigen Informationen für die Verarbeitung.

5.2.4 Datenformat

Wie schon in Kapitel 4 erwähnt, wurde JSON als Nachrichtenformat gewählt.

Das Datenformat unterstützt sowohl Einzel- als auch Batch-Anfragen. Batch-Anfragen enthalten ein Array von den relevanten Zelldaten, während Einzelanfragen direkt die Zellparameter übertragen. Das Datenmodell dazu sieht wie folgt aus:

```
{
  "cells": [
    {
      "pressure": -1922.56,
      "temperature": 307,
      "volume": 6.24e-10,
      "species": {"H2": 0.5, "O2": 0.5}
    }
  ]
}
```

Erweiterbarkeit: Das JSON-Format ermöglicht eine einfache Erweiterung um zusätzliche Parameter, ohne die Abwärtskompatibilität zu beeinträchtigen. Neue Felder können optional hinzugefügt werden, während bestehende Implementierungen unbekannte Felder ignorieren.

5.2.5 Kommunikationsablauf

Der detaillierte Ablauf der Kommunikation folgt einem standardisierten Muster, das die korrekte Synchronisation und Fehlerbehandlung gewährleistet.

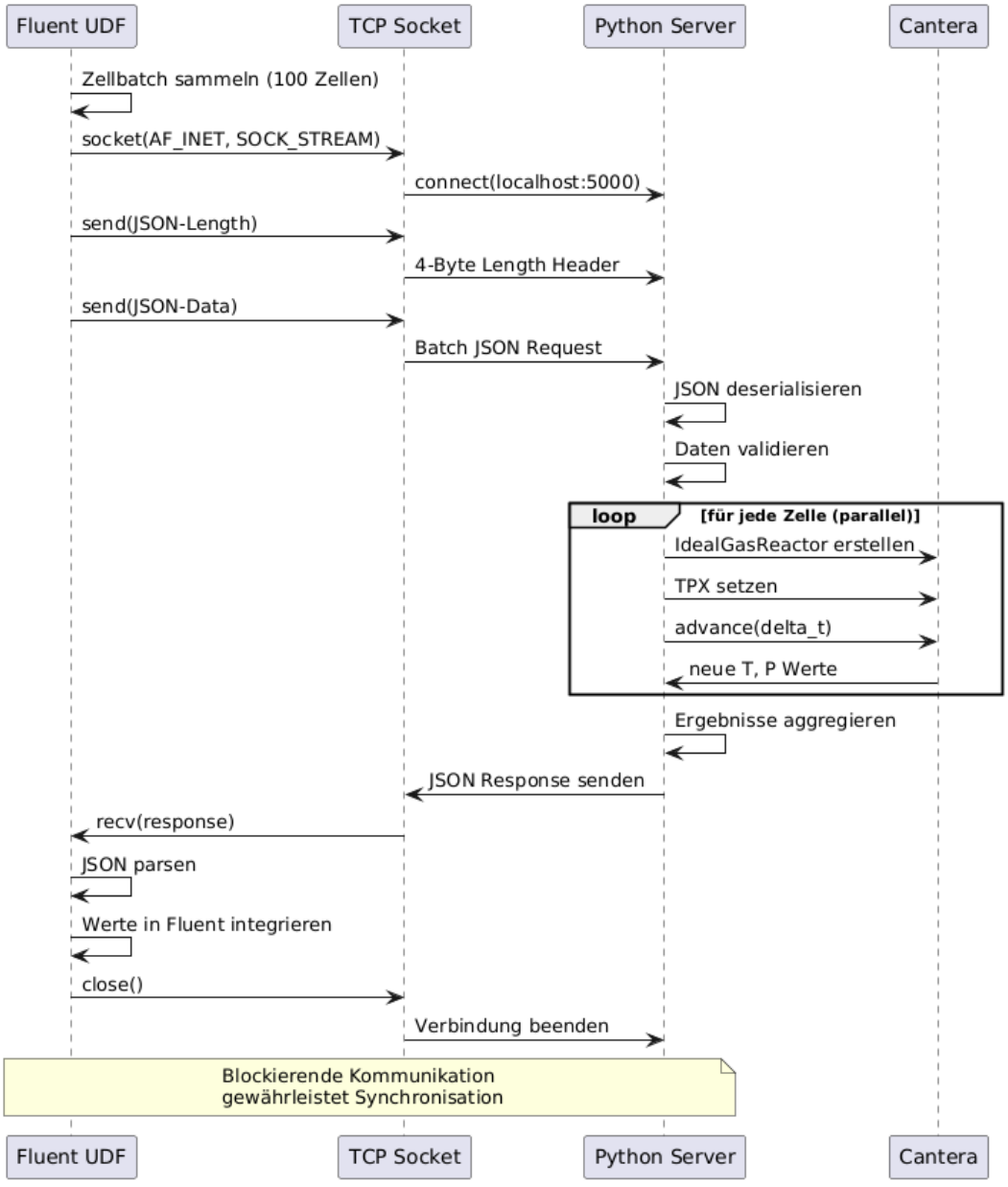

Abbildung 5.4: Sequenzdiagramm der Socket-Kommunikation

Abbildung 5.4 zeigt, dass jeder Kommunikationszyklus mit dem Verbindungsaufbau durch die UDF beginnt. Nach dem erfolgreichen Verbindungsaufbau wird erstmal die Länge der JSON-Nachricht übertragen. Danach die eigentliche Nachricht mit den Daten. Der Server verarbeitet die Anfrage und sendet eine entsprechende Antwort zurück, bevor die Verbindung wieder geschlossen wird.

5.3 Datenfluss-Architektur

Die Architektur des Datenflusses definiert, wie Informationen zwischen den verschiedenen Systemkomponenten ausgetauscht werden und stellt sicher, dass die physikalische Konsistenz der gekoppelten Simulation gewährleistet bleibt.

5.3.1 Systemweiter Datenfluss

Der Datenfluss folgt einem zyklischen Muster, das sich in jedem Zeitschritt der 3D-CFD-Simulation wiederholt. Dieser Zyklus ist so gestaltet, dass er nahtlos in den bestehenden Fluent-Workflow integriert werden kann.

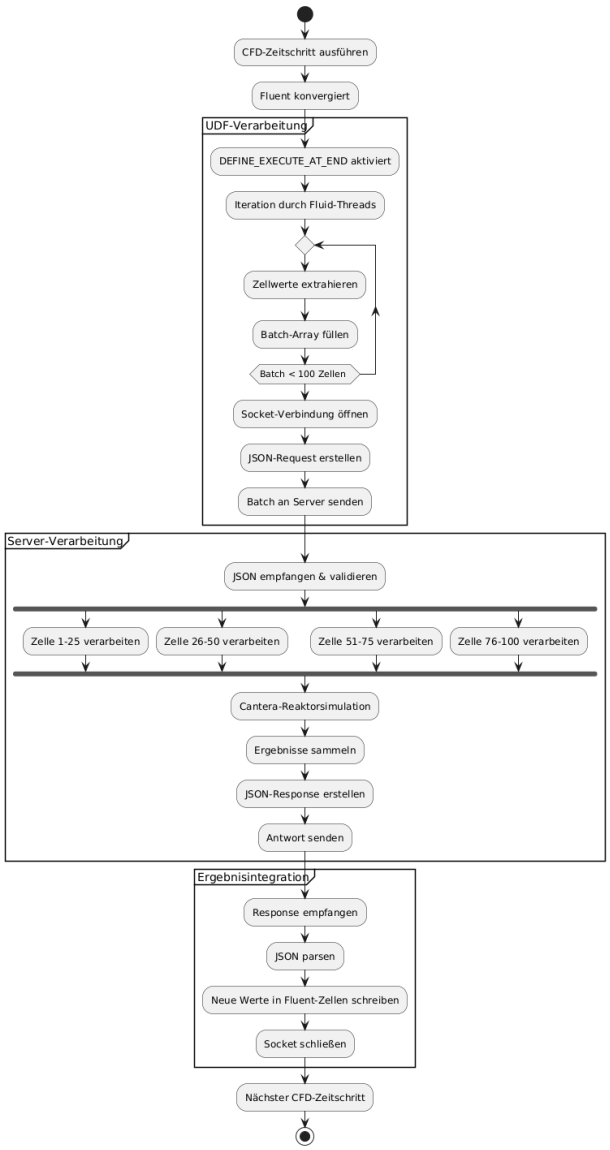

Abbildung 5.5: Aktivitätsdiagramm des Datenfluss-Zyklus

Abbildung 5.5 zeigt den Ablauf eines Kopplungszyklus für eine Batch-Größe von 100 Zellen. Der Prozess beginnt mit der Ausführung eines 3D-CFD-Zeitschritts in Fluent. Nach der Konvergenz werden die relevanten Zellwerte extrahiert und zu Batches zusammengefasst. Diese Batches werden parallel auf dem Server verarbeitet, bevor die Ergebnisse in die 3D-CFD-Simulation integriert werden.

Triggering-Mechanismus: Die UDF wird durch das `DEFINE_EXECUTE_AT_END` Makro automatisch am Ende jedes Zeitschritts aktiviert. Dieser Zeitpunkt gewährleistet, dass alle physikalischen Felder der 3D-CFD-Simulation konvergieren und konsistent sind.

Batch-Formation: Die extrahierten Zellwerte werden zu Batches von maximal 2500 Zellen zusammengefasst.

Parallel Processing: Auf der Serverseite werden die Batch-Anfragen parallelisiert verarbeitet, um die verfügbaren CPU-Ressourcen optimal zu nutzen. Die Parallelisierung erfolgt auf der Ebene einzelner Zellen innerhalb eines Batches.

5.3.2 Datenkonvertierung und -validierung

Ein wichtiger Aspekt der Datenfluss-Architektur ist die korrekte Behandlung der unterschiedlichen Datenformate und Referenzsysteme der beiden Simulationsumgebungen.

Einheitenkonvertierung: Fluent und Cantera verwenden teilweise unterschiedliche Referenzsysteme und Einheiten. Die Architektur sieht vor, dass diese Konvertierungen zentral auf der Serverseite durchgeführt werden, um Konsistenz zu gewährleisten.

Datenvalidierung: Jede übertragene Nachricht wird sowohl syntaktisch als auch semantisch validiert. Diese mehrstufige Validierung gewährleistet die Datenintegrität und ermöglicht eine frühzeitige Fehlererkennung.

5.4 Fehlerbehandlungskonzept

Die Architektur des Fehlerbehandlungssystems ist darauf ausgelegt, verschiedene Fehlerklassen zu erkennen und darauf zu reagieren. Dabei soll die Gesamtsimulation nicht gefährdet werden.

5.4.1 Fehlerklassifikation

Das Konzept der Fehlerbehandlung unterscheidet zwischen verschiedenen Fehlertypen, welche auf unterschiedlicher Art und Weise behandelt werden können.

Kommunikationsfehler: Das sind z. B. Netzwerkprobleme, Timeout-Situationen und

Verbindungsabbrüche. Sie werden durch Retry-Mechanismen und Fallback-Strategien behandelt.

Datenfehler: Ungültige oder inkonsistente Daten werden durch Validierungsroutinen erkannt und durch Standardwerte oder vorherige gültige Werte ersetzt.

Simulationsfehler: Fehler in der Cantera-Berechnung werden abgefangen und führen zur Beibehaltung der ursprünglichen Fluent-Werte für die betroffenen Zellen.

5.4.2 Graceful Degradation

Durch Graceful Degradation bleibt das System auch bei partiellen Fehlern funktionsfähig.

Isolation: Fehler in einzelnen Komponenten werden isoliert und beeinträchtigen nicht die Funktion anderer Systemteile.

Fallback-Mechanismen: Bei kritischen Fehlern werden automatisch Fallback-Strategien aktiviert, die eine Fortsetzung der Simulation ermöglichen.

Monitoring: Ein umfassendes Monitoring-System protokolliert alle Fehler und ermöglicht eine nachträgliche Analyse der Systemstabilität.

Die beschriebene Architektur bildet eine Grundlage für die praktische Implementierung der Kopplungsschnittstelle. Die modulare Struktur und die klaren Schnittstellen ermöglichen eine schrittweise Entwicklung und erleichtern spätere Erweiterungen des Systems.

6 Implementierung

Nach dem in Kapitel 5 beschriebenen Systementwurf ging es an die praktische Umsetzung der Kopplungsschnittstelle. Dabei stellten sich verschiedene Herausforderungen, die von der Auswahl geeigneter Entwicklungstools bis hin zu plattformspezifischen Problemen reichten. Dieses Kapitel beschreibt die konkreten Schritte der Implementierung und die dabei aufgetretenen praktischen Schwierigkeiten.

6.1 Technologie-Stack und Entwicklungsumgebung

Die Auswahl der Technologien war größtenteils durch die Anforderungen von Fluent und Cantera vorgegeben. Es mussten trotzdem verschiedene Entscheidungen bezüglich der Entwicklungstools und Build-Systeme getroffen werden.

6.1.1 UDF-Entwicklungsumgebung

Die Entwicklung von UDFs für Fluent erwies sich als deutlich komplexer als ursprünglich erwartet. Fluent erfordert spezifische Compiler-Konfigurationen, die je nach Betriebssystem stark variieren. Unter Windows musste Microsoft Visual Studio in einer kompatiblen Version installiert werden, während unter Linux der GCC-Compiler verwendet wurde.

Build-Integration: Fluent bietet eigene Build-Mechanismen, die automatisch die korrekten Include-Pfade und Compiler-Flags setzen. Ein Versuch, externe Libraries direkt einzubinden, scheiterte an Fluents geschlossener Architektur. Dies führte dazu, dass die gesamte Socket-Kommunikation mit den Standard-C-Libraries implementiert werden musste.

Debugging-Problematik: Eine der größten Herausforderungen war das Debugging der UDF. Standard-Debugger können nicht verwendet werden, da die UDF im Fluent-Prozess läuft. Stattdessen musste umfangreich mit der `Message()` Funktion gearbeitet werden, um Laufzeitinformationen zu erhalten. Dies führte zu einem iterativen Entwicklungsprozess mit vielen Compile-Test-Zyklen. Beim kleinsten Fehler im C-Code stürzte Fluent ab. Man musste also Fluent jedesmal neustarten und die Simulation laden.

Plattformkompatibilität: Die Notwendigkeit, sowohl Windows- als auch Linux-Systeme zu unterstützen, machte extensive Verwendung von Präprozessor-Direktiven

erforderlich. Besonders die Socket-APIs unterscheiden sich zwischen den Plattformen:

```
#ifdef _WIN32
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/socket.h>
#include <arpa/inet.h>
#endif
```

6.1.2 Python-Server-Entwicklung

Für die Serverseite wurde eine minimalistische Herangehensweise gewählt. Python 3.12 bildete die Basis, wobei die Anzahl der externen Dependencies niedrig gehalten wurde.

Die Entwicklung erfolgte parallel zur UDF-Implementierung, wobei zunächst einfache Test-Clients verwendet wurden, um die Server-Funktionalität isoliert zu testen. Dies erwies sich als sehr effektiv, da Probleme in der Netzwerkkommunikation unabhängig von Fluent-spezifischen Problemen debuggt werden konnten.

6.2 UDF-Implementierung

Die Implementierung der UDF brachte verschiedene Herausforderungen mit sich, die sowohl die Fluent-Integration als auch die Netzwerkprogrammierung in C betrafen.

Es gab während der Implementierung der UDF verschiedene Herausforderungen. Diese betrafen unter anderem die Fluent-Integration, aber auch die Programmierung in C.

6.2.1 Fluent-Datenstrukturen und Makros

Durch von Fluent bereitgestellte Makros war es möglich, auf interne Daten zuzugreifen. Dies war doch nicht immer intiutiv. Besonders die Iteration über Zellen und Threads erforderte einige Experimente, um die korrekte Syntax zu finden.

Zellzugriff: Die Extraktion von Zelldaten funktioniert über spezielle Makros wie `C_P()` für Druck, `C_T()` für Temperatur und `C_VOLUME()` für Zellvolumen. Dabei stellte sich heraus, dass diese Makros nur für Fluid-Threads gültig sind. Die korrekte Überprüfung erfolgt mit `FLUID_THREAD_P()`.

Species-Extraktion: Die Extraktion von Spezieskonzentrationen erwies sich als besonders problematisch, da die entsprechenden Makros nicht in allen Fluent-Konfigurationen verfügbar sind. An die Species erlangt man mit dem Macro `C_YI(c, t, i)`, wobei `c` für Zelle, `t` für Thread und `i` für den Index der Specie steht. Problematisch war es, wenn

man auf einen Species-Index zugriff, der nicht existiert. Fluent stürzte hierbei ab. Ebenso musste man bei der Extraktion die Reihenfolge der Species kennen, um sie richtig zu benennen und in die Verbrennungssimulation senden zu können. Als Fallback-Lösung wurden Standardwerte verwendet, wenn die Speziesextraktion fehlschlägt.

Timing-Kontrolle: Die Verwendung des `DEFINE_EXECUTE_AT_END` Makros war entscheidend für die korrekte zeitliche Koordination. Dieser wird automatisch nach jedem Zeitschritt der transienten Simulation ausgeführt. Der neue Zeitschritt beginnt erst, wenn die Funktion endet.

6.2.2 Socket-Kommunikation in C

Die Implementierung der Netzwerkkommunikation in C stellte eine erhebliche Herausforderung dar, besonders im Hinblick auf Fehlerbehandlung und Plattformkompatibilität.

Verbindungsmanagement: Jede Batch-Übertragung verwendet eine neue Socket-Verbindung, die nach der Transaktion wieder geschlossen wird. Diese Strategie vereinfacht die Fehlerbehandlung erheblich, da bei Problemen nur der aktuelle Batch betroffen ist.

JSON-Serialisierung: Da keine externen JSON-Libraries verwendet werden konnten, musste die JSON-Erstellung manuell mit `sprintf()` implementiert werden.

```
strcpy(json_data, "{\"cells\": [");
for (i = 0; i < count; i++) {
    sprintf(temp_str,
            "{\"pressure\": %.2f, \"temperature\": %.2f, \"volume\":
    %.12e, \"species\": \"%s\"}",
            pressures[i], temperatures[i], volumes[i], species_data[i])
    ;
    strcat(json_data, temp_str);

    if (i < count - 1) {
        strcat(json_data, ", ");
    }
}
strcat(json_data, "]}");
```

Fehlerbehandlung: Ein kritischer Aspekt war die Implementierung von Timeout-Mechanismen. Die UDF darf nicht unbegrenzt auf eine Server-Antwort warten, da dies die gesamte CFD-Simulation blockieren würde. Gleichzeitig müssen die Timeouts großzügig genug bemessen sein, um auch bei hoher Systemlast korrekte Berechnungen zu ermöglichen.

6.2.3 Batch-Verarbeitung

Die Implementierung der Batch-Verarbeitung ist entscheidend für die Performance der Kopplung. Einzelzell-Übertragungen hätten einen prohibitiv hohen Kommunikationsoverhead verursacht.

Batch-Größenoptimierung: Für die Entwicklungsphase wurde eine Batchgröße von 100 Zellen ausgewählt. Dies erleichterte die Entwicklung und das Debugging drastisch. Für die Produktionsumgebung wurde entschieden, eine Batch-Größe von 2500 Zellen zu wählen, da sich nach verschiedenen Versuchen eine Batch-Größe von 2500 Zellen als optimal erwies. Kleinere Batches führten zu hohem Kommunikationsoverhead, während größere Batches zu Speicherproblemen und längeren Wartezeiten führten.

Speicherverwaltung: Alle Batch-Arrays werden als lokale Variablen auf dem Stack allokiert, was die Speicherverwaltung vereinfacht. Die feste Batch-Größe gewährleistet einen vorhersagbaren Speicherverbrauch.

6.3 Server-Implementierung

Die Python-basierte Server-Implementierung konzentrierte sich auf die effiziente Verarbeitung der Batch-Anfragen und die Integration mit Cantera.

6.3.1 Cantera-Integration

Die Integration mit Cantera erwies sich als unkompliziert, da die Python-API gut dokumentiert und intuitiv ist. Dennoch traten folgende praktische Probleme auf.

Druckkonvertierung: Ein kritischer Punkt war die korrekte Behandlung der Druckwerte. Fluent arbeitet mit Relativdruck (Gauge Pressure), während Cantera absolute Drücke benötigt. Die Umrechnung erfolgt durch Addition des Atmosphärendrucks:

```
P1_absolut = 101325 + pressure  # 1 atm + Relativdruck
```

Reaktorkonfiguration: Die Konfiguration der Cantera-Reaktoren erforderte einige Experimente mit verschiedenen Parametern. Besonders die Toleranzwerte für die numerische Integration mussten angepasst werden, um stabile Ergebnisse zu erhalten.

Fehlerbehandlung: Cantera wirft bei ungeeigneten Eingabewerten Exceptions. Diese werden abgefangen und führen zur Rückgabe der ursprünglichen Eingabewerte. Das führt zur Stabilität der gekoppelten Simulation.

6.3.2 Parallelverarbeitung

Die Implementierung der parallelen Verarbeitung nutzt Pythons `multiprocessing` Modul, um die verfügbaren CPU-Kerne optimal auszunutzen.

Process Pool Management: Die Anzahl der Worker-Prozesse wird dynamisch an die verfügbaren CPU-Kerne angepasst, mit einer Obergrenze von 8 Prozessen. Diese Begrenzung verhindert eine Überlastung des Systems, besonders in Shared-HPC-Umgebungen.

Memory Management: Die Verwendung separater Prozesse statt Threads vermeidet Probleme mit Pythons Global Interpreter Lock (GIL) und ermöglicht echte Parallelverarbeitung. Gleichzeitig wird jeder Worker-Prozess nach Abschluss automatisch beendet, was Speicher-Leaks verhindert.

6.3.3 JSON-Protokoll-Implementierung

Die Implementierung des JSON-basierten Kommunikationsprotokolls nutzt Pythons eingebaute `json` Bibliothek, was die Entwicklung erheblich vereinfachte.

Request-Parsing: Der Server erkennt automatisch anhand der Nachrichtenstruktur, ob es sich um eine Einzel- oder Batch-Anfrage handelt. Dies geschieht durch Überprüfung des `cells` Feldes in der JSON-Nachricht.

Response-Format: Die Antworten folgen im JSON Format mit Status-Feldern, die eine einfache Fehlerbehandlung auf der Client-Seite ermöglichen.

```
{
  "results": [
    {
      "success": true,
      "new_pressure": -1845.23,
      "new_temperature": 315.8,
      "error_message": ""
    },
    {
      "success": true,
      "new_pressure": -1920.45,
      "new_temperature": 312.1,
      "error_message": ""
    }
  ],
  "batch_success": true,
  "total_processed": 2
}
```

7 Evaluation und Ergebnisse

In diesem Kapitel wird die entwickelte Kopplungsschnittstelle zwischen der 3D-CFD-Simulation und der 0D-Verbrennungssimulation systematisch evaluiert. Die Validierung erfolgt durch ein mehrstufiges Testkonzept, das sowohl die einzelnen Komponenten isoliert als auch das Gesamtsystem im Verbund überprüft. Zunächst wird die Grundfunktionalität des Python-Servers getestet, anschließend die UDF-Komponente auf strukturelle Korrektheit überprüft und schließlich die vollständige End-to-End-Kommunikation zwischen beiden Komponenten validiert. Dabei liegt besonderes Augenmerk auf der Batch-Verarbeitung, die einen zentralen Aspekt der Implementierung darstellt. Die Ergebnisse der Tests werden hinsichtlich Funktionalität, Systemrobustheit und praktischer Anwendbarkeit bewertet.

7.1 Testmethodik

Zur Validierung der entwickelten Kopplungslösung wurde ein systematisches Testkonzept implementiert, das die verschiedenen Komponenten sowohl isoliert als auch im Verbund überprüft. Die Evaluation erfolgte in einer kontrollierten Umgebung mit Windows 10 und Python 3.12.7.

Das Testkonzept gliedert sich in drei aufeinander aufbauende Phasen: Zunächst wurde die Grundfunktionalität des Python-Servers isoliert getestet, anschließend die UDF-Komponente auf strukturelle Korrektheit überprüft und schließlich die vollständige End-to-End-Kommunikation zwischen beiden Komponenten validiert. Diese schrittweise Herangehensweise ermöglichte es, potenzielle Fehlerquellen gezielt zu identifizieren.

Für die Tests wurden realistische Testszenarien mit standardisierten Testdaten entwickelt, die typische Anwendungsfälle der Kopplung abbilden. Dabei wurde besonderes Augenmerk auf die Batch-Verarbeitung gelegt, da diese einen zentralen Aspekt der Implementierung darstellt.

7.2 Funktionsvalidierung

7.2.1 Server-Grundfunktionalität

Die Evaluation der Server-Komponente erfolgte durch automatisierte Tests verschiedener Aspekte der Netzwerkkommunikation. Der entwickelte Test ermittelte zunächst einen verfügbaren Port aus einem vordefinierten Bereich, um Konflikte mit anderen Diensten zu vermeiden. Diese Flexibilität erwies sich als praktisch, da der Standard-Port 5000 in der Testumgebung bereits belegt war.

Die Serverkomponente bestand alle grundlegenden Funktionalitätstests erfolgreich. Die Dateisystemprüfung bestätigte das Vorhandensein aller erforderlichen Module. Der Verbindungstest validierte die TCP-Socket-Implementierung und die korrekte Behandlung eingehender Client-Verbindungen.

Der Test der JSON-Kommunikation war besonders relevant für die praktische Anwendung. Hierbei wurde ein Einzelzellen-Request im Format der UDF gesendet und die Server-Antwort analysiert. Der Server verarbeitete die Anfrage erfolgreich und lieferte eine strukturell korrekte Antwort im erwarteten Format zurück.

7.2.2 UDF-Strukturvalidierung

Die Untersuchung der UDF-Komponente erfolgte durch statische Code-Analyse. Die entwickelte `coupling_udf.c` wurde erfolgreich lokalisiert und auf das Vorhandensein aller erforderlichen Fluent-spezifischen Makros und Funktionen überprüft.

Die Analyse bestätigte die korrekte Implementierung der wesentlichen UDF-Komponenten: Die `DEFINE_EXECUTE_AT_END`-Funktion als Hauptkopplungsschnittstelle war vollständig implementiert, ebenso wie die erforderlichen Header-Dateien für Socket-Kommunikation und Fluent-Integration. Die Konfigurationsparameter für Server-Verbindung und Batch-Verarbeitung wurden als korrekt identifiziert.

Die implementierten Datenstrukturen für die Batch-Verarbeitung entsprechen den Anforderungen für eine effiziente Datenübertragung. Die vorgesehene Batch-Größe von 2500 Zellen stellt einen sinnvollen Kompromiss zwischen Kommunikationseffizienz und Speicherverbrauch dar.

7.2.3 End-to-End-Kommunikationsvalidierung

Die End-to-End-Validierung simulierte die vollständige Kommunikation zwischen UDF und Server unter Verwendung des exakten Protokolls aus der `coupling_udf.c`. Diese Tests demonstrierten die erfolgreiche Implementierung der Kopplungsschnittstelle.

Der Einzelzellen-Kommunikationstest bestätigte die korrekte Funktionsweise des JSON-basierten Protokolls. Eine Testzelle mit den Parametern Druck (101325 Pa), Temperatur (300 K), Volumen ($1.23 \times 10^{-6}\,\mathrm{m}^3$) und Species-Zusammensetzung (H2:0.5, O:0.5) wurde erfolgreich verarbeitet. Der Server lieferte modifizierte Werte zurück, was die grundsätzliche Funktionsfähigkeit der 0D-Verbrennungssimulation bestätigt.

Die Batch-Kommunikationstests validierten die Funktionsfähigkeit des Systems mit verschiedenen Batch-Größen von 5 bis 50 Zellen. Alle Batches wurden erfolgreich verarbeitet, was die Wirksamkeit des gewählten Batch-Verarbeitungsansatzes bestätigt.

Die Tests mit verschiedenen Species-Zusammensetzungen bestätigten die Robustheit der Cantera-Integration. Alle getesteten H2/O2-Verhältnisse wurden erfolgreich verarbeitet, einschließlich extremer Zusammensetzungen. Dies zeigt, dass die 0D-Simulationskomponente verschiedene Verbrennungsbedingungen handhaben kann.

7.3 Systemrobusheit

Die Fehlerbehandlungstests bestätigten die Stabilität des implementierten Systems. Verschiedene Fehlerszenarien wurden getestet, darunter ungültiges JSON, fehlende Datenfelder und physikalisch unrealistische Werte. In allen Fällen reagierte der Server graceful, ohne Abstürze oder unvorhersagbare Zustände zu verursachen.

Wichtig für die Anwendung ist, dass Kommunikationsfehler die Fluent-Simulation nicht beeinträchtigen würden. Die UDF-Implementierung ist so konzipiert, dass sie bei Server-Ausfällen oder Netzwerkproblemen den Simulationsablauf nicht blockiert.

7.4 Diskussion der Ergebnisse

7.4.1 Erreichte Ziele

Die durchgeführte Evaluation bestätigt, dass die wesentlichen Projektziele erfolgreich erreicht wurden. Die entwickelte Kopplungsschnittstelle ermöglicht eine funktionsfähige Kommunikation zwischen Ansys Fluent und Cantera. Das implementierte Client-Server-Modell mit JSON-basiertem Protokoll erweist sich als praktikabel und erweiterbar.

Die Batch-Verarbeitung reduziert den Kommunikations-Overhead signifikant und zeigt, dass das System für realistische CFD-Anwendungen prinzipiell geeignet ist.

7.4.2 Verbesserungsmöglichkeiten

Basierend auf den Evaluationsergebnissen lassen sich mehrere Ansatzpunkte für Verbesserungen identifizieren. Die Integration zusätzlicher Validierungslogik für physikalische Plausibilität der übertragenen Daten würde die Robustheit des Systems erhöhen.

Eine adaptive Batch-Größe, die sich dynamisch an die Systemlast oder die Gesamtzellenanzahl anpasst, könnte die Effizienz weiter optimieren. Die Erweiterung auf umfangreichere Reaktionsmechanismen würde das System für realistische Anwendungen qualifizieren.

7.4.3 Einordnung der Ergebnisse

Die erzielten Ergebnisse bestätigen die Machbarkeit der angestrebten Kopplung und liefern eine solide Grundlage für weiterführende Entwicklungen. Das implementierte System stellt einen funktionsfähigen Proof-of-Concept dar, der die grundlegenden Herausforderungen der CFD-Verbrennung-Kopplung erfolgreich adressiert.

Die während der Evaluation gewonnenen Erkenntnisse über Systemverhalten bilden eine wertvolle Basis für die Weiterentwicklung zu einem produktionsreifen System. Die identifizierten Verbesserungsmöglichkeiten bieten klare Ansatzpunkte für zukünftige Arbeiten in diesem Bereich.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Bachelorarbeit wurde eine Schnittstelle entwickelt, die eine 3D-CFD-Simulation in Ansys Fluent mit einer 0D-Verbrennungssimulation in Cantera koppelt. Das Hauptziel war es, die Stärken beider Simulationsumgebungen zu kombinieren: Fluents Strömungsberechnung mit Canteras Behandlung komplexer chemischer Reaktionsmechanismen.

Zu Beginn der Arbeit wurden vier verschiedene Kopplungsansätze untersucht und bewertet. Nach einer systematischen Analyse erwies sich der Socket-basierte Ansatz als die beste Lösung. Dieser Ansatz verwendet eine Client-Server-Architektur, bei der eine UDF in Fluent als Client fungiert und ein Python-Server die 0D-Verbrennungsberechnungen mit Cantera durchführt. Die Kommunikation erfolgt über TCP-Sockets mit einem JSON-basierten Protokoll.

Während der Implementierung stellten sich einige unerwartete Herausforderungen heraus. Das Fluent Build-System erwies sich als deutlich restriktiver als ursprünglich angenommen. Die geplante modulare UDF-Architektur musste zugunsten einer Standalone-Lösung aufgegeben werden, da Fluents Compiler keine echte Modularität unterstützt. Auch das Debugging gestaltete sich schwieriger als erwartet, da Standard-Debugger nicht mit UDFs funktionieren. Stattdessen musste die Entwicklung iterativ mit der Message()-Funktion erfolgen.

Ein zentraler Aspekt der Lösung ist die Batch-Verarbeitung. Anstatt jede Zelle einzeln zu übertragen, werden mehrere Zellen zu Batches zusammengefasst, was den Kommunikations-Overhead erheblich reduziert. Die Tests zeigten, dass eine Standard-Batch-Größe von 2500 Zellen einen guten Kompromiss zwischen Effizienz und Speicherverbrauch darstellt.

Die Evaluierung bestätigte die Funktionsfähigkeit der entwickelten Lösung. Alle Tests, von der Server-Grundfunktionalität über die UDF-Strukturvalidierung bis hin zur End-to-End-Kommunikation, verliefen erfolgreich. Besonders erfreulich war, dass das System robust auf Fehlerszenarien reagiert und die CFD-Simulation auch bei Kommunikationsproblemen nicht beeinträchtigt wird.

Die Arbeit zeigt, dass die Kopplung von 3D-CFD- und 0D-Verbrennungssimulationen technisch machbar ist und praktische Vorteile bietet. Das entwickelte System ermöglicht den zuverlässigen Austausch von Druck-, Temperatur-, Volumen- und Species-Daten

zwischen beiden Simulationsumgebungen.

8.2 Ausblick

Es gibt für die entwickelte Lösung noch einige Bereiche, die man verbessern kann. Eine adaptive Batch-Größe, die sich an die Summe der Zellen anpasst, könnte die Performance weiter optimieren.

Für weiterführende Forschungsarbeiten wäre eine Validierung mit experimentellen Daten interessant. Bisher wurde die Funktionalität der Kopplung gezeigt, aber ein Vergleich mit Messdaten würde die Genauigkeit der gekoppelten Simulation bewerten.

Auch die Erweiterung der Fehlerbehandlung bietet Potenzial. Das aktuelle System reagiert robust auf die meisten Fehlerszenarien, aber eine detailliertere Protokollierung und Analyse von Fehlern könnte bei der Systemoptimierung helfen.

Diese Arbeit bildet einen wichtigen Grundstein für die praktische Umsetzung einer funktionsfähigen Kopplung. Die erarbeiteten Erfahrungen bezüglich technischer Schwierigkeiten und deren Bewältigung lassen sich als Ausgangspunkt für nachfolgende Forschungsarbeiten in diesem Fachgebiet verwenden.

Literaturverzeichnis

Amazon Web Services. (2025). JSON vs XML - Difference Between Data Representations. Verfügbar 10. April 2025 unter https://aws.amazon.com/compare/the-difference-between-json-xml/

ANSYS Inc. (2025). ANSYS Fluent | Fluid Simulation Software. Verfügbar 10. März 2025 unter https://www.ansys.com/products/fluids/ansys-fluent

ANSYS, Inc. (2023a). *ANSYS Fluent Customization Manual, Release 2023 R1*.

ANSYS Inc. (2023b). *ANSYS Fluent Theory Guide, Release 2023 R1*. ANSYS, Inc.

Avižienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11–33.

Bray, T. (2014, März). *The JavaScript Object Notation (JSON) Data Interchange Format* (Request for Comments Nr. RFC 7159). Internet Engineering Task Force. https://doi.org/10.17487/RFC7159

Cantera Developers. (2023). Cantera Documentation. Verfügbar 20. März 2025 unter https://cantera.org/documentation

Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Verfügbar 22. April 2025 unter https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

German Aerospace Center. (2025). The German Aerospace Center (DLR). Verfügbar 8. März 2025 unter https://www.dlr.de/en

Goodwin, D. G., Speth, R. L., Moffat, H. K., & Weber, B. W. (2018). Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes. https://doi.org/10.5281/zenodo.1174508

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., & Lafon, Y. (2007, April). *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* (W3C Recommendation). World Wide Web Consortium (W3C). https://www.w3.org/TR/soap12-part1/

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, *7*(3), 404–425.

Knack. (2024). CSV Formatting: The Ultimate Guide & Examples. Verfügbar 5. April 2025 unter https://www.knack.com/blog/csv-formatting/

Mattern, F. (1993). Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, *18*(4), 423–434.

PromptCloud. (2024). Various Data Delivery File Formats and their Pros and Cons. Verfügbar 2. April 2025 unter https://www.promptcloud.com/blog/data-delivery-formats-pros-cons/

PyData. (2025). Cantera | PyData. Verfügbar 28. März 2025 unter https://pydata.org/project/cantera/

PyFluent Developers. (2025). PyFluent documentation 0.32.2 — PyFluent. Verfügbar 15. März 2025 unter https://fluent.docs.pyansys.com/

Python Software Foundation. (2025). Embedding Python in Another Application [Python 3.13.5 documentation]. Verfügbar 18. März 2025 unter https://docs.python.org/3/extending/embedding.html

Schneider, S. J. (2021). *Beitrag zur systemischen Entwicklung und experimentellen Untersuchung eines Freikolbenmotors in Gegenkolben-Bauweise* [Dissertation]. Universität Stuttgart [Angefertigt am Institut für Fahrzeugkonzepte, Deutsches Zentrum für Luft- und Raumfahrt].

Shafranovich, Y. (2005). *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files* (RFC Nr. 4180). RFC Editor. https://tools.ietf.org/html/rfc4180

Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). *UNIX Network Programming, Volume 1: The Sockets Networking API* (3. Aufl.). Addison-Wesley Professional.

Tu, J., Yeoh, G. H., & Liu, C. (2018). *Computational Fluid Dynamics: A Practical Approach* (3. Aufl.). Butterworth-Heinemann.

Versteeg, H. K., & Malalasekera, W. (1995). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Longman Scientific; Technical.

Wikipedia-Autoren. (2009, August). Cantera. Verfügbar 18. Mai 2025 unter https://de.wikipedia.org/wiki/Cantera#:~:text=Die%20Software%20wurde%20ab%201998,II%20mit%20einigen%20zus%C3%A4tzlichen%20Erweiterungen.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Weiterhin erkläre ich, dass die Arbeit nicht anderweitig veröffentlicht oder an anderer Stelle als Prüfungsleistung vorgelegt wurde.

Stuttgart, den

Berk Meriç