Christoph Aldrian, BSc

Entwickeln einer verteilten Anwendung zur Simulation und Optimierung von Modelica-Modellen

Masterarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur / Master of Science

Masterstudium: Softwareentwicklung - Wirtschaft

eingereicht an der

Technische Universität Graz

Betreuer

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Institut for Softwaretechnologie

Groß St. Florian, August 2019

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Kurzfassung

Der weltweite Energieverbrauch steigt kontinuierlich an. Ein signifikanter Anteil des Verbrauchs resultiert aus dem Energiebedarf zur Heizung und zur Kühlung in Haushalten. Für die Nutzung des Einsparungspotenzials gewinnen Faktoren des Konsumentenverhaltens zunehmend an Bedeutung. Jenes Verhalten zu verstehen und zu beeinflussen gilt als effektive Möglichkeit, die Energieeffizienz zu steigern und Energiesparen zu fördern. Gebäudeenergiesysteme werden zum besseren Verständnis mit Hilfe von mathematischen Modellen abgebildet. Ihr reales Verhalten wird durch Simulation der Modelle approximiert.

Diese Arbeit hat sich zum Ziel gesetzt, Computersimulation von Gebäudeenergiesystemen für Konsumenten zugänglicher zu machen. Ein Framework wird entwickelt, das über eine Web API Eingaben zur Parameteroptimierung von voreingestellten Modelica-Modellen entgegennimmt. Anwendungen können diese API nutzen, um Konsumenten über die Auswirkungen ihres Verhaltens auf den Energieverbrauch ihrer Wohneinheiten aufzuklären. Ein beispielhafter Anwendungsfall ist integriert, welcher die optimale Stärke einer zu installierenden Fassadendämmung ermittelt. Miteinbezogen werden sowohl Anschaffungskosten als auch fortlaufende Energieersparnisse.

Das Framework besteht aus vier Hauptbestandteilen. *Webserver* stellen sowohl die Web API als auch eine einfache Webanwendung bereit. Die Daten werden in einer dokumentenorientierten NoSQL-*Datenbank* gespeichert. *Simulationsumgebungen* nehmen Modelldefinitionen, Parameter und Code zum Kompilieren und Simulieren des Modells entgegen. Ein dynamischer *Aufgabenplaner* verteilt Optimierungsaufträge an verfügbare Simulationsumgebungen. Zur einfachen Bereitstellung wird Docker Swarm mode genutzt.

Die Evaluierung bestätigt, dass die entwickelte Lösung unter optimaler Ausnutzung der darunter liegenden Hardware korrekte Ergebnisse berechnet.

Abstract

The world-wide energy consumption is on the rise. A significant portion of the consumption results from households' energy demand for heating and cooling. Studies show a large saving potential that can be exploited by using energy more efficiently. In addition to technological advances, behavioural factors are of great importance. Understanding and changing the energy consumption behaviour of end-users have been identified as effective ways to improve energy efficiency and promote energy saving. In order to better understand building energy systems they are represented as mathematical models which are then simulated for the sake of approximating their real-world behaviour.

This thesis aims to make computer simulation of building energy systems more accessible to consumers. A framework is developed which offers a Web API that accepts inputs for parameter optimizations of pre-defined Modelica models. Third-party applications can use the API to teach its users how their behaviour affects the energy consumption of their buildings. An example use case is implemented which allows to calculate the optimal thickness of a facade insulation to be installed, taking into account the investment costs as well as the energy conservation over time.

The framework consists of four main parts. *Web servers* host the web API as well as a simple web application. The data ist persisted in a document-oriented NoSQL *database*. *Simulation environments* accept model definitions, parameters and code to compile and simulate the model. A *dynamic task scheduler* distributes the optimization requests to the connected simulation environments. For easy setup and deployment the framework makes use of Docker Swarm mode.

The evaluation confirms that the solution calculates correct outputs and makes optimal use of the underlying hardware.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Der wirtschaftliche Aufschwung und die gesellschaftliche Entwicklung begründen eine stetige Zunahme des globalen Energieverbrauchs [1, S. 811]. Der Anstieg des elektrischen Energieverbrauchs seit 2010 beläuft sich dabei auf nahezu das Doppelte des Anstiegs des gesamten Energieverbrauchs [2, S. 3]. Studien haben gezeigt, dass aufgrund von wachsenden Komfortansprüchen, des Klimawandels und der zunehmenden Temperaturen im Sommer zudem eine deutliche Erhöhung des Energiebedarfs in der EU zu erwarten ist. Mehr als die Hälfte des Energieendverbrauchs fällt für Heiz- und Kühlsysteme an. Hier besteht erhebliches Potenzial, Energie einzusparen und in weiterer Folge den CO_2-Ausstoß zu reduzieren. [3]

Ein signifikant hoher Anteil des Mehrverbrauchs ist auf den zunehmenden Energiebedarf in Haushalten zurückzuführen. In den vergangenen Jahrzehnten konnten große Fortschritte in den Bereichen Gebäudeeffizienz, Heizung, Lüftung und Klimatechnik erzielt werden. Heute kommt dem Verstehen des Verhaltens von KonsumentInnen, dem Vermitteln von Einsparungspotentialen und dem Schaffen von Anreizen, dieses Verhalten anzupassen, größere Bedeutung zu [1, S. 811].

1.1 Problemstellung

Die aktuelle Forschung hat sich zum Ziel gesetzt, BewohnerInnen für das Thema Energieoptimierung in ihrem Zuhause zu sensibilisieren sowie durch ihr Mitwirken geeignete Informationen zur verbesserten Planung von Gebäuden und übergeordneten Energiesystemen zu generieren. Es wurde das Projekt *Gamification für die Optimierung des Energieverbrauchs von*

Gebäuden und übergeordneten Systemen, kurz *GameOpSys*[1], ins Leben gerufen. *Gamification* bezeichnet die »Verwendung von Spieltechniken in einem Kontext, der kein Spiel ist« [4]. Durch die Einbindung von Gamification-Konzepten soll ein spielerisches Umfeld für BewohnerInnen bereitgestellt und damit Anreize zur Partizipation geschaffen werden. Studien haben gezeigt, dass Gamification sich positiv auf das Energiesparverhalten auswirken kann, wobei hinsichtlich der Art von Spielen und deren Spielelementen kaum Grenzen bestehen [5].

Um für BewohnerInnen Energieeffizienzsteigerungen ihrer Wohneinheiten zu erzielen, ist es notwendig, diese als Modelle abzubilden. Für derartige Modelle kann in weiterer Folge der Energieverbrauch simuliert und schließlich auch optimiert werden. Es können dabei dynamische Effekte und Trägheiten der Gebäude, wie etwa die Bauteilaktivierung für Heizung und Kühlung, miteinbezogen werden[1].

BewohnerInnen können die Simulationen mit individuellen Angaben beeinflussen. Das bedeutet, dass sie zusätzlich zur Abbildung ihrer Wohneinheiten beispielsweise Angaben hinsichtlich der gewünschten minimalen und maximalen Raumtemperatur machen können. Auch können Art und Anzahl von Haushaltsgeräten, inklusive der Angabe eines Wunschzeitraums für die Nutzung derer, definiert werden[1].

Energieversorger können externe Randbedingungen definieren, sodass Eigeninteressen in der Energieversorgung, wie beispielsweise eine Lastverschiebung zur Vermeidung von Spitzen, ebenfalls Berücksichtigung finden. Durch bedarfsorientierte Preisgestaltung kann auf das KonsumentInnenverhalten Einfluss genommen werden. Die Berücksichtigung von Kriterien beider Seiten soll dem übergeordneten Ziel entsprechen: die Nachfrage und die verfügbare Energie zeitlich, örtlich und quantitativ aufeinander abzustimmen[1].

Die Schnittstelle zwischen den BewohnerInnen und den Energieversorgern bildet eine Webanwendung, welche auch am Smartphone verwendet werden kann[1]. Die Anwendung wird von einer serverseitigen Infrastruktur bereitgestellt und tauscht Benutzerdaten mit ihr aus. Die Aufgabe dieser Masterarbeit ist es, eine serverseitige Infrastruktur (von nun an *Framework*

[1] aus dem Förderungsansuchen des Projekts für das Programm *Stadt der Zukunft*

genannt [6]) als Bereitstellungs- und Kommunikationsschnittstelle zu konzipieren und exemplarisch umzusetzen.

1.2 Zielsetzung

Die Webanwendung sieht das Verwalten des eigenen Benutzerprofils vor. Darin werden neben persönlichen Präferenzen grundlegende Angaben zu den Wohnsitzen gespeichert, wie beispielsweise die geografische Lage oder bauliche Gegebenheiten. Mit diesen Angaben ermöglicht es die Anwendung, bestimmte Berechnungen durchzuführen bzw. Vergleiche aufzustellen, wie eine Anpassung des eigenen Verhaltens sich auf den Energieverbrauch und die damit verbundenen Kosten auswirken kann.

Die zugrunde liegenden numerischen Simulationen werden auf mathematischen Modellen durchgeführt, welche mit der Modellierungssprache Modelica spezifiziert sind. Darauf aufbauend kommen Optimierungsalgorithmen zum Einsatz. Aus Gründen der Benutzerfreundlichkeit hinsichtlich Laufzeit und Energieverbrauch sowie möglicher hardware- und softwareseitiger Einschränkungen werden diese Berechnungen nicht vom Smartphone bewältigt, sondern von einem Verbund an leistungsstarken Rechnern.

Aus dieser Vorgabe leitet sich die Aufgabenstellung für die Arbeit ab: das Konzipieren und Entwickeln eines Frameworks zur Durchführung von Computersimulationen und zur Lösung von Optimierungsproblemen im Zusammenhang mit Gebäuden und deren Energieverbrauch.

Das Ziel dieser Arbeit ist die Beantwortung der Frage, welche Komponenten benötigt werden und wie diese interagieren müssen, um Parameteroptimierungen für Modelica-basierte Gebäudemodelle an einer zentralen, über das Web erreichbaren Stelle durchführen zu können. Die Verwendung moderner Entwicklungswerkzeuge und Technologien, eine hohe Anpassbarkeit an etablierte Bereitstellungsumgebungen sowie eine optimale Ausnutzung vorhandener Ressourcen für die effiziente Verrichtung der Aufgaben, werden implizit vorausgesetzt. Die Zielerfüllung wird anhand der Lösung einer beispielhaften Optimierungsaufgabe an einem vereinfachten Gebäudemodell belegt.

1.3 Aufbau der Arbeit

Das Kapitel 2 befasst sich mit theoretischen Grundlagen zu Modellierung und Simulation. Es wird mit der Definition grundlegender Begriffe eingeleitet und erläutert in weiterer Folge verschiedene Modellierungsansätze. Im Anschluss wird Modelica als Modellierungssprache, sowie ihre Möglichkeiten zur Durchführung von Gebäudesimulationen, vorgestellt.

In Kapitel 3 werden Anwendungsfälle beschrieben, anhand derer die Eignung der konzipierten, serverseitigen Infrastruktur unter Beweis gestellt wird.

Kapitel 4 geht auf das Durchführen von Simulationen mit JModelica ein und eruiert Möglichkeiten zur Leistungssteigerung durch parallelisiertes und verteiltes Rechnen.

Der Aufbau eines Frameworks rund um JModelica, zur Entgegennahme von Parameteroptimierungen via Web API, wird in Kapitel 5 beleuchtet.

In Kapitel 6 werden die Ergebnisse der Leistungsevaluierungen des Frameworks auf verschiedener Hardware zusammengefasst.

Am Ende der Arbeit wird zu den Ergebnissen Stellung genommen und ein Ausblick über mögliche Erweiterungen und Verbesserungen des verwendeten Ansatzes gegeben.

2 Grundlagen

2.1 Modellierung und Simulation

Die Modellbildung hat ihren Ursprung in der Meteorologie und Nuklearphysik aus der Zeit nach dem zweiten Weltkrieg [7]. Hauptzweck der Modellbildung und (Computer-) Simulation ist es, komplexe Systeme zu analysieren und deren Verhalten nachzubilden [8, S. 11f].

Am Beginn einer Simulation steht ein Modell eines physischen Systems. Modelle können verkleinerte, realistische Darstellungen des Originals sein. Sie erlauben es, komplexe Systeme am Computer abzubilden und deren Verhalten zu simulieren, ohne sie in der Realität nachbauen oder erproben zu müssen. Die Herausforderung dabei ist, dass das Modell detailliert genug ist, um das physikalische System ausreichend repräsentieren zu können.
[9, S. 9ff]

Das Verfahren der Modellbildung und Simulation ist ein einheitlicher Ansatz, der in verschiedensten Bereichen der Wissenschaft wie beispielsweise im Maschinenbau, der Astrophysik, der Materialforschung, der Klimaforschung, in der Medizin, der Umweltforschung, der Betriebswirtschaft etc. zur Anwendung kommt [7]. Über Modelle und Simulationen herauszufinden, was mit einem System unter bestimmten Umständen passieren könnte, befriedigt nicht nur die Neugier der Menschheit, sondern kann auch über Leben und Tod entscheiden. Beispiele: Simulation der Entwicklung von Regionen, Klimaveränderungen, Simulation von Fahrzeugverhalten oder medizinischen Geräten. Die Aufgabe der Modellbildung und Simulation ist es, zu relativ sicheren Aussagen über das Verhalten eines Systems unter definierten Umständen zu kommen. Ziel ist eine zuverlässige Beschreibung des Verhaltens eines realen Systems zu erlangen.
[8, S. 11ff]

2.1.1 Grundlegende Begriffsdefinitionen

Im Folgenden werden die zentralen Begriffe im Zusammenhang mit Modellbildung und Simulation, System, Modell und Simulation, näher definiert.

System

Im Wesentlichen ist ein System ein Objekt oder eine Menge von Objekten dessen Eigenschaften erforscht werden sollen. Ein System ist beispielweise das Universum, ein Flugzeug oder ein Gebäude. Ein System kann dabei aus Sub- oder Teilsystemen bestehen. [10, S. 3f]

Ein System weist bestimmte Merkmale auf:

- Es erfüllt eine bestimmte Funktion, Systemzweck.
- Es besteht aus Elementen und Wirkungsverknüpfungen (Relationen).
- Es ist nicht teilbar. Würde man ein Element herauslösen, wäre es zerstört und der ursprüngliche Systemzweck nicht mehr erfüllt. Beispiel: ein Stuhl. Fällt ein Bein weg, ist der Stuhl zerstört und der Systemzweck ist nicht mehr erfüllt.

Bei der Modellbildung und Simulation werden die Wirkungsverknüpfungen im System herausgearbeitet. [8, S. 16]

Systeme lassen sich in natürliche oder künstlich erzeugte unterscheiden. Ein natürlich entstandenes System ist das Universum. Ein künstliches System ist ein Haus oder ein Flugzeug. Systeme können aber auch beides vereinen. Ein Beispiel dafür ist ein System, das als Subsystem ein natürliches, wie etwa die Sonne, und ein künstliches, wie ein Haus, in einem Gesamtsystem vereint.

Die Festlegung was ein System ist, ist willkürlich bzw. frei wählbar. In der Definition eines Systems muss sehr selektiv und abhängig von den zu untersuchenden Aspekten vorgegangen werden.
[10, S. 4f]

Modell

Der präziseste Weg, das Verhalten eines Systems zu analysieren, ist es, dieses unter verschiedenen Bedingungen zu beobachten [8, S. 27]. Den Prozess des Variierens mit Eingabeparametern und das Beobachten und Kontrollieren der Ausgaben nennt man experimentieren. Um an einem System experimentieren zu können, muss dieses also steuerbar und beobachtbar sein [10, S. 5f]. Dies ist aber oft aus verschiedensten Gründen nicht möglich. In einem solchen Fall ist der einzige Weg, das Verhalten des Systems zu erforschen, dies nicht am Realsystem, sondern an einem Modell durch Simulationen zu erarbeiten [8, S. 27]. Modelle füllen demzufolge die Lücke zwischen Theorie und Experiment [9, S. 9ff].

Modelle repräsentieren ein System in einer vereinfachten Weise. Es gibt verschiedene Arten von Modellen: mentale, verbale, physikalische und mathematische. Letztere sind für diese Arbeit von Relevanz. Mathematische Modelle beschreiben ein System, in dem Beziehungen zwischen Variablen des Systems in mathematischen Formeln ausgedrückt werden. Variablen können dabei messbar sein wie Größe, Länge etc. [10, S. 6]

Die Abbildung eines Systems als Modell hat viele Vorteile. Einige davon sollen hier genannt werden: es sind keine Experimente am Original notwendig, es besteht keine Gefahr für das reale System, es ist möglich schneller eine größere Anzahl an Messergebnissen zu erzielen etc. Letztgenanntes ermöglicht es außerdem, ein breiteres Spektrum an Verhalten abzudecken und die Kosten dabei verhältnismäßig gering zu halten.

Es gibt aber auch Nachteile, die die Abbildung eines Systems über ein Modell mit sich bringt. Hier kann im Wesentlichen die ständige Unsicherheit genannt werden, ob das Verhalten in der Simulation in allen Aspekten dem realen System entspricht.
[8, S. 27]

Die Richtigkeit eines Modells lässt sich nicht beweisen. Daher wird auch nicht von der Richtigkeit gesprochen, sondern von der Gültigkeit eines Modells für einen bestimmten Modellzweck [8, S. 36].

Computersimulation

Computer Simulation ist im Allgemeinen das Experimentieren am Modell analog dem Experimentieren am realen System. Beispiele, in denen Simulationen zur Anwendung kommen, sind:

- das Finden von Prozessoptimierungen in einem industriellen Fertigungsprozess,
- das Verhalten eines Fahrzeuges unter bestimmten Umweltbedingungen nachzustellen,
- in IT Systemen das Verhalten der Server unter verschiedenen Lasten nachzustellen, um die Performance entsprechend verbessern zu können etc.

Dabei ist es so, dass die Beschreibung des Modells und die Beschreibung des Experiments zwei getrennte Aspekte darstellen, die einander jedoch bedingen. Ein Modell kann beispielsweise nur für bestimmte Experimente Gültigkeit haben.

Liegt das mathematische Modell ausführbar auf einem Computer vor, können Simulationen mittels numerischen oder nicht numerischen Experimenten durchgeführt werden. Es ist eine einfache und sichere Art des Experimentierens, mit dem Vorteil, dass alle Variablen des Modells beobachtbar und kontrollierbar sind. Die Aussagekraft und der Wert der Ergebnisse sind jedoch davon abhängig, wie gut das Modell konzipiert wurde und dem tatsächlichen System entspricht.

Ergebnisse sind entsprechend nicht automatisch zuverlässig. Es muss viel Aufwand und Expertise in die Modellierung und die Simulation einfließen, um beurteilen zu können, welche Ergebnisse glaubwürdig sind und welche nicht.

Die Einfachheit der Verwendung von Simulationen birgt gleichzeitig die größte Gefahr, wenn auf Einschränkungen und Bedingungen, unter denen die Simulation Gültigkeit hat, vergessen wird und die falschen Schlüsse aus den Ergebnissen der Simulation gezogen werden. Um dies zu vermeiden, soll immer versucht werden, Ergebnisse der Simulation mit experimentellen Ergebnissen am realen System zu vergleichen.
[10, S. 7f]

2.1.2 Erweiterte Begrifflichkeiten

Dynamische Systeme

Wenn sich innerhalb eines für uns relevanten Zeitraums der Zustand eines Systems ändert, zeigt es *dynamisches Verhalten*. Bei der Beobachtung dieses dynamischen Verhaltens kann nicht davon ausgegangen werden, dass Veränderungen wesentlicher Eigenschaften des Systems von außen erkennbar sind. Oft muss der innere Zustand des Systems erfasst werden, um weitere Informationen über das Systemverhalten feststellen zu können. Die Eigenschaften, die den tatsächlichen Systemzustand beschreiben, werden Zustandsgrößen genannt. Durch sie lässt sich der Zustand des Systems jederzeit vollständig erfassen. Sie spielen bei der Systemanalyse, Modellbildung und Simulation deshalb eine wesentliche Rolle. [8, S. 17]

Ein Modell muss selbst ein solches dynamisches Verhalten erzeugen können. Es muss eine Wirkungsstruktur mit Systemparametern aufweisen und auf Einwirkungen der Umgebung reagieren können. Oft handelt es sich hierbei um mathematische Formeln, von denen sich durch Festlegen von Eingaben ein Systemverhalten ableiten lässt. [8, S. 27]

Systemgrenzen

Systeme haben eine Systemumgebung, die äußeren Einfluss auf die Entwicklung des Systems nehmen kann. Aber auch das System selbst kann die Systemumwelt beeinflussend verändern.

Eine völlige Isolation des Systems ist nicht möglich. Es ist daher notwendig, Systemgrenzen zu ziehen. Damit lässt sich die Komplexität einer Untersuchung erheblich reduzieren. Das Ziehen dieser Grenze ist jedoch oftmals schwierig und bedarf besonderer Aufmerksamkeit.
[8, S. 17f]

Systemverhalten und Wirkungsstrukturen

Einwirkungen von außen, aber auch Prozesse innerhalb des Systems, können zu Zustandsänderungen führen und damit die Zustandsgrößen beeinflussen. Das Systemverhalten entsteht durch systemunabhängige Einwirkungen und durch Rückwirkungen im System selbst. Beide Arten von Wirkungen werden über die systeminterne Wirkungsstruktur weitergegeben und verändert. Für die Beschreibung des Systemverhaltens muss diese Wirkungsstruktur bekannt sein. [8, S. 20]

Modularisierung

Bei der Untersuchung eines komplexen Systems aus mehreren Teilsystemen ist es sinnvoll und einfacher, nach Ziehen der Grenzen zwischen den Teilsystemen diese und ihr Verhalten auf Außeneinwirkungen getrennt zu untersuchen. Ist die Wirkungsstruktur der Teilsysteme bekannt, kann das Verhalten des Gesamtsystems untersucht werden. [8, S. 22]

2.1.3 Modellbildung, Modellentwicklung

Am Beginn der Modellentwicklung steht die Definition des Modellzwecks. Je genauer der Zweck definiert ist, desto präziser kann die Modellformulierung erfolgen. Ein System kann für unterschiedliche Modellzwecke durch unterschiedliche Modelle repräsentiert werden. [8, S. 28]

Eine Modellbildung bedarf vieler Entscheidungsvorgänge. Es sind daher umfassende Gültigkeitsprüfungen und Falsifikationsversuche notwendig, um den Grad an Subjektivität in der Modellbildung möglichst zu minimieren. [8, S. 36]

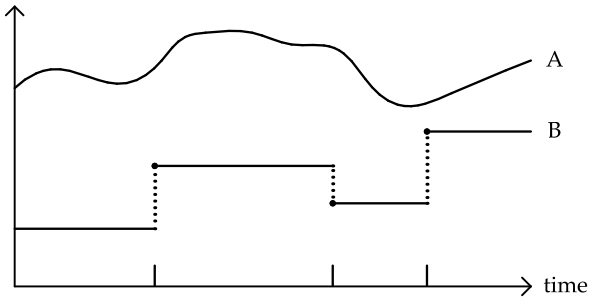


Abbildung 2.1: Zeitkontinuierlich (A) vs zeitdiskret (B) [10, S. 13ff]

Arten mathematischer Modelle

Abhängig vom Verhalten eines Systems, können verschiedene Arten mathematischer Modelle mit verschiedenen Unterscheidungskriterien definiert werden [10, S. 13].

Zeitkontinuierliche vs. zeitdiskrete dynamische Modelle Zeitkontinuierlich bedeutet, dass das System zu jedem beliebigen Zeitpunkt definiert und messbar ist, zeitdiskrete Systeme hingegen nur zu bestimmten Zeitpunkten (siehe Abbildung 2.1). Computer arbeiten zeitdiskret. Für die Darstellung zeitkontinuierlicher Modelle werden Berechnungen in eine Reihe zeitdiskreter aufgeteilt (sehr kleine Treppenkurven), die nahe an das kontinuierliche Verhalten herankommen. [8, S. 38]

Arten der Modellentwicklung

Die Wahl des geeigneten Modellierungsansatzes muss gut überlegt und der zu simulierenden Situation angepasst sein. Es gibt eine Reihe von Unterscheidungskriterien für die Wahl des geeigneten Modellierungsansatzes. Im Folgenden werden verschiedene genannt bzw. beschrieben.

Black Box- vs. White Box- vs. Grey Box-Modellierung Bei der *Black Box*-Modellierung ist die Wirkungsstruktur des Originalsystems nicht von Interesse. Eine Beschreibung des historischen Verhaltens unter bestimmten

Umweltbedingungen lässt auf Verhalten unter gleichen Bedingungen in der Zukunft schließen. Das System wird sozusagen lediglich von außen betrachtet.

Black Box-Modellierung wird auch datengetriebene oder statistische Modellierung genannt. Es müssen große Mengen an Daten durch die Beobachtung des Systemverhaltens generiert werden. Bei der Modellierung nach dem Black Box-Verfahren werden Eingabe-Ausgabe-Relationen beschrieben und das Modell lernt anschließend aus der Fülle der vorhandenen Daten (maschinelles Lernen, künstliche Intelligenz).

Es handelt sich hierbei um eine eher unflexible Methodik, die eine Erweiterung, Modifizierung oder Verbesserung des Modells schwierig macht. Ein Vorteil dieser Variante ist jedoch, dass die Simulation sehr effizient ist.
[8, S. 29ff]

Datengetriebene Modelle werden häufig verwendet um beispielweise Raumtemperaturen, Wetterbedingungen oder den Energieverbrauch für Heizungs-, Lüftungs- und Klimatechniksysteme (*HVAC*) vorherzusagen.
[11]

White Box-Modellierung beschreibt die Nachbildung eines Systems. Das nachgebildete Modell zeigt das gleiche Systemverhalten wie das Original. Es handelt sich um ein Modell des Systems und nicht ein Modell des Systemverhaltens. Die Wirkungsstruktur muss hierfür erkannt und verstanden werden. Für diese Art der Modellentwicklung sind keine Verhaltensbeobachtungen erforderlich. Sehr wohl muss jedoch die Systemstruktur bekannt sein, um daraus alle, das System beschreibenden, physikalischen Gleichungen und in weiterer Folge das Systemverhalten ableiten zu können.

Diese Form der Modellentwicklung ermöglicht im Gegensatz zur Verhaltensbeschreibung auch Aussagen über noch nicht beobachtetes, zukünftiges Verhalten. Anwendungsbeispiele hierfür sind Wettervorhersagen oder Flugsimulatoren.

Als Nachteile der White Box-Modellierung können genannt werden: der hohe Bedarf an Know-how, zeitintensives, jedoch erforderliches Validieren sowie eine aufwendigere Simulation, die oft langsamer ist und viele Systemparameter aufweist, die unklar, nicht bekannt oder nicht beobachtbar sind.
[8, S. 29ff]

Aufgrund der soeben angeführten Vor- und Nachteile beider Modellierungsverfahren werden diese häufig kombiniert, zur sogenannten *Grey Box*-Modellierung. Bei der Grey Box-Modellierung werden beispielsweise einige Blöcke als Black Boxes abgebildet und einige andere, für die Eingriffe und Modifizierungsmöglichkeiten gegeben sein müssen, als White Boxes. Es entsteht somit ein vereinfachtes physikalisches Modell, wie beispielsweise eines Hauses und dessen HVAC-Systems. Physikalische Parameter werden aggregiert und teilweise werden Schätzwerte, unter Zuhilfenahme von Parameterschätzverfahren, angenommen.

Die Nachteile des Grey Box-Modellierungsverfahren entsprechen im Wesentlichen jenen des White Box-Verfahrens.
[12]

Akausale vs. kausale Modellierung White Box- und Grey Box-Modellierung können weiter untergliedert werden in *kausale* und *akausale* Modellierungsansätze. Ein akausales Modell besteht aus der Definition von Variablen und ihren Beziehungen zueinander. Es behält die Struktur vom physikalischen System bei und ist gut für eine schnelle Prototypenentwicklung auf Komponentenebene geeignet. Die zugrundeliegenden mathematischen Formeln sind implizite differential-algebraische Gleichungen (*DAE*). Diese sind einfach zu lesen und erhöhen die Wiederverwendbarkeit, Erweiterbarkeit und Anpassbarkeit des Modells. Ein Nachteil dieser Variante ist, dass sie entfernter von der Lösung des Algorithmus ist.

In der akausalen Modellierung liegt der Fokus auf dem möglichst einfachen und natürlichen Modellierungsprozess, der das Ableiten von Code für Simulationen dem Simulationswerkzeug überlässt. Es gibt im Wesentlichen zwei Methoden zur Simulation von akausalen Modellen: mit Hilfe vorkompilierter Komponenten sowie mit Hilfe von Methoden, in denen Gleichungen global zur Integration zur Verfügung stehen.
[13, S. 6ff]

Ein kausales Modell basiert auf Ein- und Ausgaberelationen. Eingaben stellen aus der Systemumgebung kommende Daten dar, Ausgaben sind Daten, die an die Systemumgebung zurückgegeben werden. Es liegt die Darstellung

von Ursache-Wirkungs-Beziehungen im Fokus. Die zugrundeliegenden mathematischen Formeln sind explizite gewöhnliche Differentialgleichungen (*ODE*). [3, S. 2]

Die kausale Modellierung verfolgt den Ansatz, dass ein System in Blockdiagramme, mit kausalen Interaktionen zwischen diesen Blöcken, zerlegbar ist. Diese Kausalität muss künstlich hergestellt werden, um notwendige Bedingungen für die Simulation in konventionellen sequentiellen Rechnern schaffen zu können. Oft ist jedoch ein enormer Analyseaufwand sowie ein analytischer Transformationsprozess nötig, um ein Problem bzw. System in dieser Art und Weise darzustellen. Es ist sehr aufwendig, fehleranfällig und erfordert hohe technische Fähigkeiten. [14, S. 1ff]

In kausalen Modellen wird die Auflösung von Gleichungen nicht dem Computer überlassen, sondern es muss der Weg der Berechnung vorgegeben werden. In der akausalen Modellierung beschreiben die Modellkomponenten direkt die Gleichungen, und nicht den Algorithmus zur Lösung der Gleichungen. [15, S. 1,4,5]

Ein Vorteil von kausalen Modellen ist, dass sie deshalb sehr nahe an der Lösung des Algorithmus liegen. Ein Nachteil ist wiederum, dass das Verbinden verschiedener Blöcke die Berechnungsprozeduren widerspiegelt, und nicht die Strukturen des zugrundeliegenden Modells. Daher sind kausale Modelle schwerer zu lesen, sind in der Wiederverwendbarkeit eingeschränkt und schwerer zu warten. Meist führen bereits kleine Änderungen in der physikalischen Struktur zu großen Änderungen im blockorientierten Modell. [3, S. 2]

Ein wesentlicher Unterschied der beiden Modelle liegt des Weiteren in der Art, Komponenten miteinander zu verbinden. Bei der akausalen Modellierung wird ein Modell erstellt, ohne zuvor Implementierungsdetails wissen zu müssen [14, S. 1f]. Der Datenfluss zwischen Komponenten ist bidirektional, was kausale Modelle nicht standardmäßig mitbringen [3, S. 10].

Abbildung 2.2 zeigt je ein Beispiel eines kausalen und eines akausalen Modells.

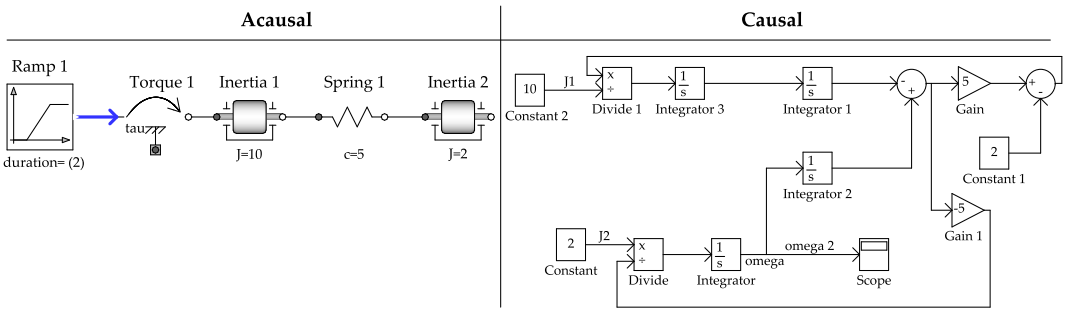


Abbildung 2.2: Beispiele eines akausalen und eines kausalen Modells [13, S. 8]

2.2 Modellierungssprache Modelica

Die mathematische Modellierung und Simulation hat sich in den vergangen Jahrzehnten als Schlüsseltechnologie erwiesen, um Probleme in Technik und Wissenschaft zu beschreiben und Lösungsmöglichkeiten zu finden. Aufgabenstellungen wurden stets komplexer und computergestützte Werkzeuge wurden benötigt, um für zukünftige Anforderungen gerüstet zu sein.

Ende der 1990er Jahre entstand ausgehend vom Projekt *Simulation in Europe Basic Research Working Group (SiE-WG)* die Idee einer neuen Modellierungssprache für komplexe physikalische Systeme, um die unterschiedlichen Schwächen vorhandener Sprachen und proprietärer Simulationssoftware wie ACSL, SIMULINK, HSPICE, DADS zu lösen. [16]

Ein Hauptkritikpunkt vieler früherer Sprachen war die jeweils sehr spezifische Anwendbarkeit. Domainübergreifender Einsatz war kaum möglich und auch das Fehlen von gemeinsamen Standards, wie beispielsweise einer einheitlichen Syntax oder Semantik, wurde als Schwachstelle identifiziert. [17]

Innerhalb der *Federation of European Simulation Societies (EUROSIM)* wurde von einem multidisziplinären Team rund um Hilding Elmqvist die Idee einer neuen, leistungsfähigen und universell einsetzbaren Modellierungssprache weiterentwickelt. Im September 1997 wurde die Definition von *Modelica 1.0* offiziell vorgestellt. [16]

Im Jahr 2000 bildete sich die *Modelica Association*, eine gemeinnützige Organisation mit dem Ziel, die Modelica-Modellierungssprache für Modellierung, Simulation und Programmierung von physikalischen und technischen Systemen und Prozessen zu entwickeln und voranzutreiben [18]. Modelica ist seitdem unter der Open-Source-Lizenz *Modelica Lizenz 2.0* in der Version 3.4, mit Stand April 2017, frei verfügbar. [19].

2.2.1 Definition

Die Modelica-Sprache ist eine nicht proprietäre, objektorientierte, auf Gleichungen basierende Sprache, um auf bequeme Weise komplexe physikalische Systeme zu modellieren, die beispielsweise mechanische, elektrische, elektronische, hydraulische, thermische, Steuerungs-, Strom- oder prozessorientierte Unterkomponenten enthalten [20].

2.2.2 Funktionsmerkmale

Die wesentlichsten Funktionsmerkmale von Modelica sind die Anwendung von differential-algebraischen Gleichungen, die Objektorientierung, der Multi-Domain-Ansatz und das umfassende Softwarekomponentenmodell:

Gleichungsbasiert

Mit Modelica ist die Nutzung von differential-algebraischen Gleichungen zur Erstellung von akausalen Modellen möglich. [16]

Das grundlegende Problem von Simulationslösungen auf Basis von kausalen Modellen ist, dass sie Systeme basierend auf physikalischen Gesetzen und Zusammenhängen nur unzureichend abbilden können bzw. dies nur mit sehr hohem analytischen Transformationsaufwand möglich ist. [21, S. 35ff]

Mit den differential-algebraischen Gleichungen als Basis unterscheidet sich Modelica auch grundlegend von klassischen Programmiersprachen, die Zuweisungen verwenden, und damit an einen bestimmten Prozessfluss gebunden sind. [10, S. 19ff]

In einem Modelica-Modell werden bekannte und unbekannte Variablen erst im Rahmen eines Experiments eines physikalischen Systems als Randbedingungen bestimmt und automatisch aufgelöst. Es sind keine manuellen Eingriffe oder Vorarbeiten notwendig. Dabei bleibt die Routine so effizient, dass auch große Modelle mit mehr als 100.000 Gleichungen berechnet werden können. [21, S. 35ff] [20]

Der akausale Ansatz, zusammen mit der Objektorientierung von Modelica, wirkt sich sehr positiv auf die Wiederverwendbarkeit von Modellen aus. [10, S. 19ff] [17] [22]

Objektorientiert

Einen großen Einfluss auf die Wiederverwendbarkeit von Modelica-Modellen hat die Objektorientierung der Sprache. In Modelica dient die Objektorientierung vor allem der Strukturierung, die die Komplexität des Modells besser beschreiben lässt. [10, S. 26]

Dynamische Modelleigenschaften werden in Modelica ebenfalls deklarativ als Gleichungen definiert. Die Implementierung des in klassischen Programmiersprachen üblichen Datenaustauschs mittels Zuweisung und Methodenaufruf ist in Modelica nicht notwendig. Dieser erfolgt automatisch durch den Modelica-Compiler. Modelica setzt auf einer höheren Abstraktionsebene als übliche objektorientierte Programmierung an. [10, S. 26ff]

Multi-Domain-Ansatz

Bereits in den ersten Überlegungen für Entwicklungen von Modelica stand die Verwendbarkeit und Interoperabilität der neuen Modellierungssprache für verschiedene Fachbereiche im Mittelpunkt.

Der akausale, gleichungsbasierte Ansatz von Modelica, zusammen mit der Objektorientierung, ermöglicht die Abbildung von physikalischen Modellen basierend auf realen physikalischen Gesetzmäßigkeiten oder Objekten, die für verschiedenste technische Fachbereiche, wie Mechanik, Maschinenbau, Elektrotechnik und Elektronik, Thermodynamik, Hydraulik und Pneumatik,

Regelungs- und Prozesstechnik gleichermaßen gültig und damit nutzbar sind. [16]

Softwarekomponentenmodell

Modelica bietet ein umfassendes Softwarekomponentenmodell, welches sich gut zur Abbildung von komplexen, physikalischen Systemen in einer Modellierungssprache eignet. [10, S. 19ff] Da sich Modelica an realen Gegebenheiten orientiert, wird die Erstellung von Modellen und Modellkomponenten sowie deren Verschaltung unter Verwendung eines grafischen Editors als besonders einfach beschrieben. [17]

Die einzelnen Komponenten werden mittels Konnektoren verbunden. Konnektoren wiederum werden selbst als Gleichungen bzw. als physisches Konstrukt beschrieben, wie beispielsweise eine elektrische Leitung oder mechanische Verbindungen. Mittels Konnektoren kann außerdem eine hierarchische Struktur von Komponenten und Subkomponenten definiert werden, wie etwa zwischen den Reifen und dem Fahrzeug. [10, S. 38],

Die Modellkomponenten werden in Bibliotheken (*Libraries*) zusammengefasst. Die *Modelica Standard Library (MSL)* bildet dabei mit 1600 Modellkomponenten und 1350 Funktionen die Grundlage, bezogen auf den jeweiligen Modelica-Sprachstandard [23]. Zusätzlich gibt es spezifische kommerzielle und nicht-kommerzielle Bibliotheken für bestimmte technische Fachbereiche [24].

2.2.3 Ausführung von Modelica-Modellen

Im Folgenden wird beschrieben, wie Modelica Modelle zur Ausführung kommen und welche Schritte dazu notwendig sind (siehe Abbildung 2.3).

Als ersten Schritt wird der Modelica Quellcode geparst und in eine interne Repräsentation umgewandelt. Diese Struktur wird dann weiter adaptiert, mit dem Ziel, eine flache Zusammenstellung von Gleichungen, Konstanten, Variablen und Funktionen zu erhalten. Dazu werden Klassen vererbt

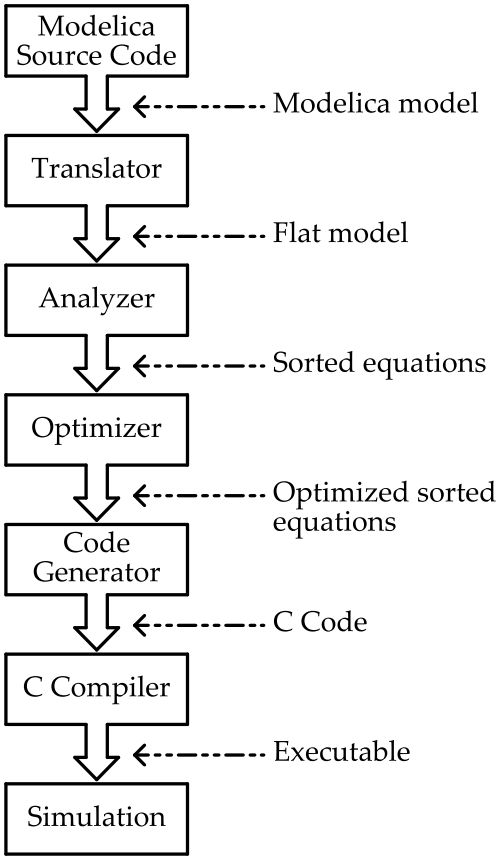


Abbildung 2.3: Kompilieren und Ausführen von Modelica-Modellen [25]

und erweitert, Adaptierungen und Instanziierungen durchgeführt sowie Konnektorengleichungen in Standardgleichungen umgewandelt.

Die Gleichungen werden nun topografisch in Datenstromrichtung sortiert. In einem weiteren Schritt wird ein Optimizer aufgerufen, der zur Vereinfachung verschiedene mathematische Methoden anwendet, sodass sich die Zahl der Gleichungen drastisch reduziert.

Anschließend wird das nun stark reduzierte und vereinfachte Gleichungssystem in C-Programmcode überführt und mittels numerischem Solver gelöst. Anfangswerte werden gesetzt, Parameterspezifikationen vorgenommen und Variablen berechnet. Das entstandene ausführbare Programm kann nun für die Simulation verwendet werden. [10, S. 64],

2.2.4 Modelica-Umgebungen

In den vorherigen Kapiteln dieser Arbeit wurde Modelica als eine Sprache für die Simulation physikalischer Systeme beschrieben. Es handelt sich dabei letztlich um Code, der mit einem Editor bearbeitet werden kann. Um eine Simulation durchführen zu können, wird eine entsprechende Softwareumgebung zur Interpretation des Codes und zur grafischen Aufbereitung der Ergebnisse benötigt. Hierzu gibt es eine Reihe verfügbarer kommerzieller und nicht-kommerzieller Softwareprodukte, die im Folgenden grob beschrieben werden. [21]

Dymola

Dymola, eine kommerzielle Modelica-Simulationssoftware, eignet sich besonders für sehr große Systeme, mit mehr als 100.000 Gleichungen. Der Schwerpunkt liegt im Speziellen auf effiziente Echtzeitsimulation [26]. Zusätzliche Funktionsmerkmale von Dymola sind beispielsweise die Möglichkeit einer grafischen 3D-Animation oder die Integration für Versionierungssysteme. [10, S. 1055ff]

JModelica

JModelica ist eine auf Modelica basierende Open-Source-Plattform für Optimierung, Simulation und Analyse komplexer, dynamischer Systeme. Es bietet eine Art virtuelles Labor für die Algorithmusentwicklung und die Algorithmusforschung an. JModelica beinhaltet einen Compiler, sowohl für Modelica als auch für *Optimica*. Letzeres ist eine Erweiterung von Modelica zur Formulierung dynamischer Optimierungsprobleme.

Weitere nennenswerte Funktionen von JModelica sind die verfügbaren Eclipse-Plugins oder die Möglichkeit der Python Integration, was JModelica zu einem sehr mächtigen Werkzeug macht. JModelica eignet sich somit gut, um Simulationen eingebettet in Python Code laufen zu lassen.
[21]

Die JModelica.org Plattform ist verfügbar unter der GPL open source license. Teile der Software sind auch als CPL license verfügbar. [26]

OpenModelica

OpenModelica ist ein Open-Source-Produkt mit dem Ziel, für Forschungs- und Lehrzwecke, aber auch zur industriellen Nutzung, eine vollständige Umgebung zur Modellierung, Kompilierung und Simulation von Modelica-Modellen bereit zu stellen. OpenModelica steht frei zur Verfügung und darf in andere, proprietäre Softwareprodukte integriert werden. OpenModelica war die erste verfügbare Open-Source-Modelica-Umgebung.

Mit OpenModelica wird die grafische Entwicklung von Modellen und Simulationsexperimenten unterstützt. Auch eine grafische Analysemöglichkeit sowie eine Exportierfunktion stehen zur Verfügung. Die aktuelle Version von OpenModelica unterstützt bereits den Großteil der Sprachelemente von Modelica, inklusive Gleichungen, Algorithmen, Ereignisbehandlung, Funktionen und Pakete. Es gibt außerdem ein Eclipse-Plugin für fortgeschrittene Softwareentwickler. Aktuelle Projekte befassen sich mit der Optimierung des Compilers, einer allgemeinen Solver-Schnittstelle, erweiterter grafischer Unterstützung und Modelica-UML-Integration in Eclipse. [26], [21]

Bestimmte Funktionsmerkmale sind essenziell und deshalb in allen aufgelisteten Umgebungen vorzufinden:

- Modelica-Compiler oder -Interpreter, um das Modell zu übersetzen, inklusive eines Optimizers zur Reduktion der Anzahl an Gleichungen
- Ausführungsmodul oder Ablaufumgebung, inklusive einem numerischen Solver für DAE-Gleichungen
- Texteditor, um das Modelica Modell zu bearbeiten
- Grafischer Editor für komponentenbasiertes Modelldesign

[10, S. 1033]

2.2.5 Functional Mock-up Interface

Functional Mock-up Interface (FMI) ist ein von Modelica-Umgebungen unabhängiger Standard, der den Austausch und die Kopplung von Modellen unterschiedlicher Umgebungen unterstützt. FMI wurde 2010 erstmals in der Version 1.0 veröffentlicht. Die Daimler AG hat die Entwicklung initiiert, um den Austausch von Modellen zwischen Herstellern und Zulieferern zu ermöglichen. Seit 2014 steht die Version 2.0 zur Verfügung.

Inzwischen wird FMI von vielen Tools unterstützt und sowohl in der Automobilbranche, als auch in vielen anderen Bereichen weltweit eingesetzt.
[10, S. 1159f]

Model Exchange vs. Co-Simulation

FMI unterstützt zwei Arten der Koppelung von dynamischen Modellen: *Model Exchange* und *Co-Simulation*.

Model Exchange bedeutet, dass Modelle, die in unterschiedlichen Simulationstools erstellt wurden, ausgetauscht, vereint und gemeinsam simuliert werden können. Eine Modellierungsumgebung erstellt hierzu aus dem dynamischen Systemmodell C-Programmcode, der in anderen Modellierungs- und Simulationsumgebungen ausgeführt werden kann.

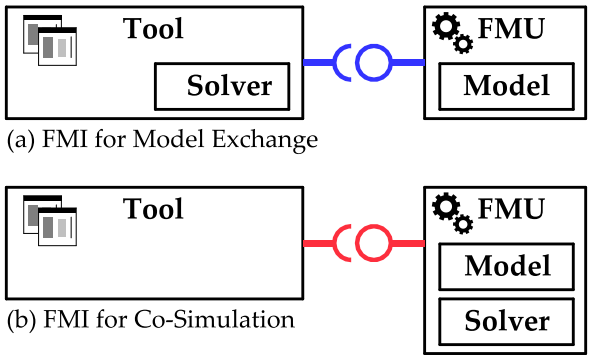


Abbildung 2.4: FMI für Model Exchange vs FMI für Co-Simulation [27]

Co-Simulation bedeutet, dass es möglich ist, mehrere Simulationswerkzeuge miteinander zu koppeln. Der Datenaustausch zwischen den Subsystemen ist beschränkt auf definierte Kommunikationspunkte. Die Subsysteme werden voneinander unabhängig von ihren individuellen Solvern aufgelöst. Ein Masteralgorithmus steuert den Datenaustausch zwischen den Subsystemen und die Synchronisation aller Simulationssolver. Der Master Algorithmus ist jedoch kein Teil von FMI.
[10, S. 1159]

Der Unterschied der beiden beschriebenen Ansätze ist in Abbildung 2.4 grafisch veranschaulicht. In der Co-Simulation-Variante wird das Modell vom eigenen Solver aufgelöst, bei Model Exchange macht dies der Solver des Simulationstools.

Functional Mock-up Unit

Die Erzeugung einer *Functional Mock-up Unit (FMU)* ist notwendig, um den oben beschriebenen Modellaustausch oder eine Co-Simulation zu ermöglichen. Der Export einer solchen FMU aus einem bestehenden Modell wird von der zugrundeliegenden Simulationssoftware, die den FMI Standard unterstützt, angeboten.

Eine FMU ist im Wesentlichen eine Datei im ZIP-Format mit der Namenserweiterung *.fmu*. Sie enthält u.a. folgende Daten:

- XML-Datei mit allen zur Simulation benötigten Informationen wie beispielsweise Meta-Informationen zum Modell oder eine Liste aller notwendigen Variablen für den Datenaustausch zwischen dem Simulator und der FMU
- Gleichungen, die für die Interaktion zwischen Modell und Simulator benötigt werden, sie können als Code oder als dynamische Programmbibliothek in Binärform (*DLL*) zur Verfügung gestellt werden
- Optionale Daten wie Dokumentation, Parametertabellen etc.

[10, S. 1160ff] [28]

2.3 Gebäude-Energiesysteme

Laut einem Bericht der internationalen Energieagentur (*IEA*) werden 36% des globalen Energiebedarfs für Gebäude und deren Errichtung aufgewendet. Daraus resultieren 40% der globalen CO2 Emissionen. [29]

Die IEA beschreibt in ihrem *Efficient World Scenario* die Entwicklung des Energieverbrauchs bei steigender Bevölkerungs- und Gebäudezahl bis zum Jahr 2040. Sie gehen dabei von der Umsetzung verschiedenster, kostenneutraler Optimierungen und Effizienzsteigerungen aus. Trotz einer Steigerung der Gebäudeflächen um 60% wird die mögliche Energieeffizienzsteigerung auf 40% eingeschätzt, wobei der größte Teil die Raumtemperaturregulierung betrifft. [29]

Um diese Effizienzsteigerungen im Neubau oder in der Revitalisierung von Altbestand umsetzen zu können, ist es notwendig, »Gebäude-Energiesysteme« zu verstehen. Darauf aufbauend können, unter zu Hilfenahme von Gebäude-Simulationssoftware (*BESP*), Simulationen zur Darstellung der Abhängigkeiten und Wechselwirkungen der Faktoren dargestellt werden. Darauf aufbauend können Optimierungen vorgenommen werden. [30], [31]

2.3.1 Definition

Als *Gebäude-Energiesysteme (BES)*, werden Systeme bezeichnet, die für den Energieverbrauch in Gebäuden verantwortlich sind. Diese können jede Art von physischer Ausstattung, technischer Geräte, ein Prozess oder der Kombination aus diesen sein. [30]

Die relevantesten Energiesysteme im Gebäudeumfeld sind die HVAC-Systeme, Beleuchtung, elektrische Motoren und andere Geräte mit hoher Leistungsaufnahme [30], wobei der größte Teil auf die Raumklima- bzw. Temperaturregulierung entfällt. Damit ist dies ein guter Ansatzpunkt für Optimierungen.

Was die Simulation des Raumklimas jedoch verkompliziert sind innere, äußere und bauliche Einflussfaktoren. Als innerer Einflussfaktor zählt z.B. die Anzahl an Personen, die sich im Gebäude aufhalten. Als äußere Einflussfaktoren zählen z.B. die Außentemperatur, die relative Feuchtigkeit, die Sonneneinstrahlung und der Wind. Mit baulichen Einflussfaktoren sind z.B. Wandbeschaffenheiten (Dicke, Material, Dämmung etc.), Fenster, Dach und Wärmeleitung gemeint. [30], [32]

Im Zusammenhang mit dem Raumklima stellt Heizen die Zufuhr und Kühlen den Entzug von Energie dar, um das Gebäude im gewünschten Temperaturbereich zu halten. Dafür verantwortlich ist das HVAC-System. Der entstehende Energieaufwand wird als *Building Space Load (BSL)* bezeichnet, der in einem späteren Schritt optimiert werden soll. Die oben genannten inneren, äußeren und baulichen Einflussfaktoren haben ebenfalls Einfluss auf den BSL. [30], [32]

Abbildung 2.5 stellt die Energieflüsse in einem geheizten und gekühlten Gebäude schematisch dar.

2.3.2 Gebäudesimulation mit JModelica

Modelica ist, wie bereits unter 2.2 Modellierungssprache Modelica erwähnt, eine akausale Modellierungssprache, die sich besonders für die Modellierung von physikalischen Systemen eignet. Außerdem ist sie zur Modellie-

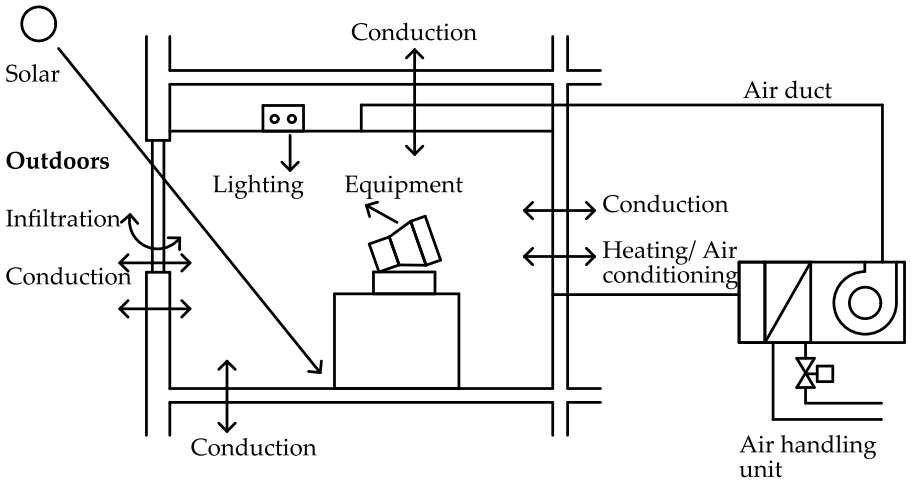


Abbildung 2.5: Schematische Darstellung eines Gebäudeenergiesystems [30]

rung von Multi-Domain-Systemen geeignet und verfügt mit der Modelica Standard Library über eine umfangreiche, frei zugängliche Sammlung von Komponenten. Aufgrund dessen wird Modelica auch zur Gebäudesimulation eingesetzt. [33]

Anfang der 2000er Jahre wurde erstmals über die Verwendung von Modelica bei Gebäude- und Energiesimulationen berichtet [34]. Bis 2012 wurden verschiedene Modelica-Bibliotheken für diesen speziellen, technischen Fachbereich entwickelt. Sie waren jedoch aufgrund von unterschiedlichen Designentscheidungen und Modellierungsprinzipien untereinander nicht kompatibel. [35]

Zu diesen Bibliotheken zählen:

- *AixLib*[1], von der Rheinisch-Westfälischen Technischen Hochschule Aachen
- *Buildings*[2], vom Lawrence Berkeley National Laboratory in Berkeley, CA, USA
- *BuildingSystems*[3], von der Universität der Künste Berlin

[1] https://github.com/RWTH-EBC/AixLib
[2] http://simulationresearch.lbl.gov/modelica
[3] http://www.modelica-buildingsystems.de

- *IDEAS*[4], von der Katholieke Universiteit Leuven in Belgien

Von 2012 bis 2017 hat die IEA zusammen mit 42 Instituten aus 16 Ländern den *IEA EBC Annex 60* entwickelt. Ziel war es, eine gemeinsame Basis für Simulations- und Berechnungswerkzeuge zu schaffen. Dabei sollten sich diese auf die folgenden, bereits existierenden, offenen Standards stützen [36], [35]:

- Modelica, siehe 2.2
- Functional Mockup Interface, siehe 2.2.5
- *Industry Foundation Classes (IFC)*, einer Definition eines Lebenszyklusmodells für Gebäude, als offener, internationaler Standard (ISO 16739) [36], [35]

Mit dem Annex 60 wurden auch die vier genannten Bibliotheken auf die gemeinsame Basis gestellt, wobei jede ihre Besonderheiten und spezifischen Stärken behalten hat. Nach Abschluss des Annex 60 wurde nun bereits das Folgeprojekt *IBPSA Project 1* gestartet, das die Entwicklung weiter vorantreiben soll. [37], [38]

2.3.3 Optimierung von Gebäudesimulationen

Durch Gebäudesimulationen werden die Zusammenhänge im Energiesystem *Gebäude* sichtbar und nachvollziehbar. Um daraus Effizienzsteigerungen und in weiterer Folge Kosteneinsparungen ableiten zu können, müssen die Parameter immer wieder angepasst werden. So nähert man sich einer besseren Lösung an, man optimiert das Ergebnis. [31]

Eine *Building Performance Simulation (BPS)* ist ein automatisierter Prozess, welcher ausschließlich auf numerischer Simulation und mathematischer Optimierung beruht [31].

Generell kann man BPS in drei Phasen einteilen:

[4] `https://github.com/open-ideas/IDEAS`

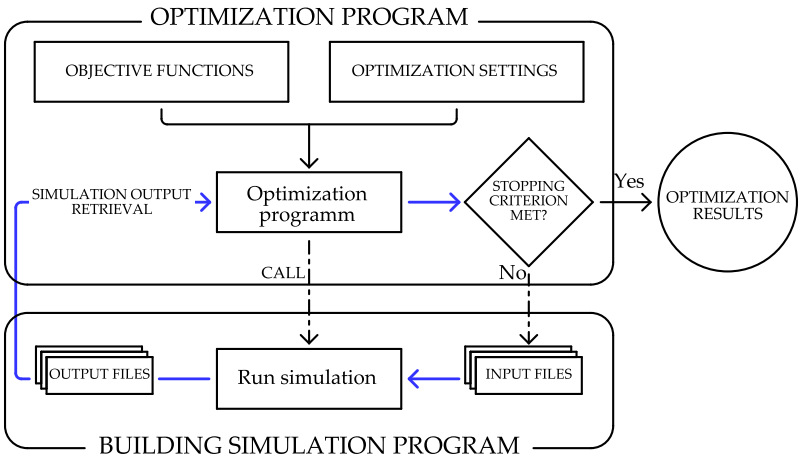


Abbildung 2.6: Kreislauf zwischen Simulation und Optimierungsprogramm [31]

1. *Vorverarbeitung (pre-processing)*: Die Grundlage jeder BPS bildet die Formulierung eines Optimierungsproblems. Des Weiteren wird das Gebäudemodell vereinfacht, um den Optimierungsprozess zu beschleunigen.
2. *Laufzeitoptimierung (running optimization)*: Während der Optimierung muss der Prozess laufend überwacht werden um etwaige Fehler zu identifizieren. Außerdem wird beobachtet, ob eine Konvergenz hergestellt wurde, der Algorithmus also zu seinem finalen Ergebnis gekommen ist. Dies bedeutet aber nicht zwingend, dass das globale Minimum identifiziert wurde. Die Abbruchbedingungen können dabei variieren.
3. *Nachbearbeitung (post-processing)*: In der Nachbearbeitungsphase werden die Optimierungsergebnisse aufbereitet (z.B. mittels Punktwolken-, Konvergenz- oder Balkendiagrammen) und interpretiert. Es wird zudem empfohlen, das Ergebnis mit zusätzlichen Methoden und Analysen auf dessen Tragfähigkeit zu verifizieren.

Abbildung 2.6 stellt den sich wiederholenden Kreislauf von Simulation und Optimierung im Rahmen der Building Performance Simulation dar.
[31]

Optimierung von Gebäudesimulationen mit Modelica

Modelica wurde für simulationsbasierte Analysen ohne angeschlossene Optimierung konzipiert. Erst vor einigen Jahren wurde die Spracherweiterung *Optimica* eingeführt, die eine direkte Modelica-basierte Optimierung erlaubt. [33]

Vorteile der Optimierung mit Optimica sind auf grundlegende Modelica-Eigenschaften zurückzuführen, wie den Multi-Domain-Ansatz, die physikalische Verankerung der Modelle und die akausale Modellierung. Hinderlich in der weiteren Verbreitung von Optimica für die Optimierung von Gebäudesimulationen ist, dass aktuell in den verschiedenen Modelica unterstützenden Programmen die implementierten Solver nur in bestimmten Klassen von Optimierungsproblemen und für nicht-hybride Modelle verwendet werden können. [33]

3 Anwendungsfälle

Für die Beschreibung der Anforderungen an das zu entwickelnde Framework werden zunächst zwei triviale Modelica-Modelle verwendet. In der abschließenden Evaluierung der Software kommt ein vereinfachtes Gebäudemodell zum Einsatz. Die drei Modelle werden im Folgenden knapp beschrieben.

3.1 Modell der Rosenbrock-Funktion

Zur Evaluierung des Leistungsverhaltens von Optimierungsalgorithmen werden häufig mathematische Testfunktionen eingesetzt [39]. Die von Howard H. Rosenbrock im Jahr 1960 eingeführte Funktion [40] ist besonders beliebt, da sie ein langes, schmales und gekurvtes Tal darstellt (siehe Abbildung 3.1), für das Algorithmen unter Umständen eine Vielzahl an Iterationen benötigen, um ihr Minimum zu finden [41]. Das Minimum ist analytisch ermittelbar und liegt für $n = 2$ bei (1,1).

$$f(x) = \sum_{i=1}^{n} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$

Aufgrund ihrer Beliebtheit für diese Art der Problemstellung kommt die Rosenbrock-Funktion als erster Anwendungsfall im Zuge der Entwicklung des Simulationsframeworks zum Einsatz. Es wird ein sehr einfaches Modelica-Modell erstellt, das neben den Ein- und Ausgabeparametern lediglich die Gleichung der Funktion spezifiziert (siehe Listing 3.1). Es werden keine zusätzlichen Bibliotheken benötigt.

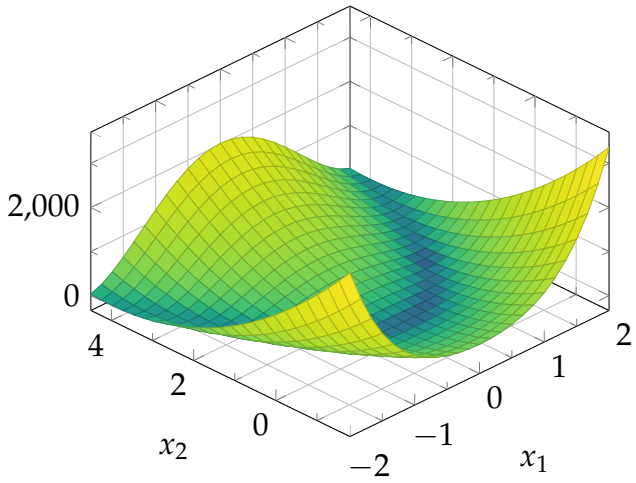


Abbildung 3.1: Grafische Oberflächendarstellung der Rosenbrock-Funktion

```
1 model Rosenbrock "Rosenbrock"
2   input Real x1(start=10);
3   input Real x2(start=10);
4   output Real u;
5 equation
6   u = 100 * (x2 - x1^2)^2 + (1 - x1)^2;
7 end Rosenbrock;
```

Listing 3.1: Modell der Rosenbrock-Funktion

Eine beispielhafte Simulation des Modells der Rosenbrock-Funktion wird unter 4.1 beschrieben.

3.2 Modell der Rastrigin-Funktion

Im Rahmen dieser Arbeit wird ein weiteres Modell mit einer mathematischen Funktion verwendet. Die Funktion wurde von L. A. Rastrigin eingeführt und stellt eine besondere Herausforderung an Optimierungsalgorithmen dar. Im Gegensatz zur Rosenbrock-Funktion, welche nur ein Minimum besitzt, hat die Rastrigin-Funktion im Bereich $x_i \in [-5.12, 5.12]$ eine parabolische Landschaftsstruktur (siehe Abbildung 3.2). Ihr globales

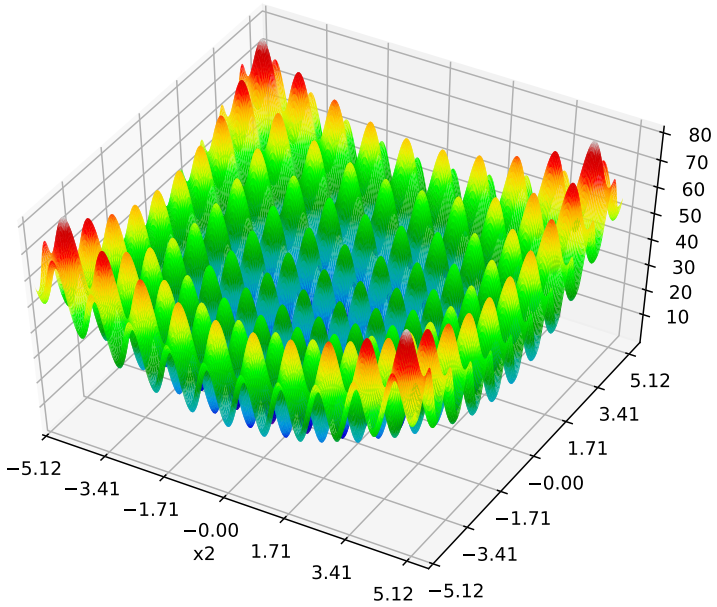


Abbildung 3.2: Grafische Oberflächendarstellung der Rastrigin-Funktion

Minimum befindet sich bei (0,0) und ist von lokalen Minima umgeben. [42]

$$f(x) = \sum_{i=1}^{n} [x_i^2 - 10 * cos(2\pi x_i) + 10]$$

Das Modell der Rastrigin-Funktion wird in 4.2 mehrfach sequenziell und parallel simuliert. Durch eine Streuung der Startwerte, in bestimmter Dichte über den zu untersuchenden Bereich, kann das globale Minimum gefunden werden (siehe Abbildung 3.3).

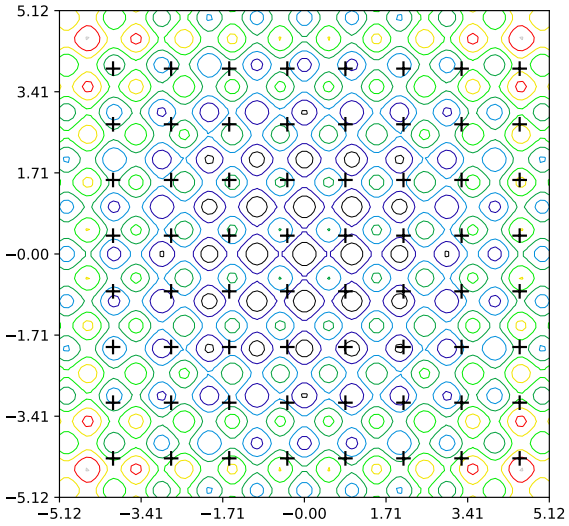


Abbildung 3.3: Raster aus 8x8 Startwerten für die Parameteroptimierung

3.3 Modell eines Gebäudes zum Optimieren der Dämmstärke

Im Rahmen einer Lehrveranstaltung wurde ein vereinfachtes Gebäudemodell entworfen und für die Entwicklung und Evaluierung des Frameworks dieser Arbeit angepasst. Es handelt sich dabei um ein Grey-Box Modell, das aus Komponenten der Bibliotheken AixLib und IDEAS zusammengesetzt ist. Als beispielhafter Anwendungsfall wird die Amortisation einer zu installierenden Dämmung berechnet.

Das Modell bietet eine Vielzahl an konfigurierbaren Parametern, darunter Eigenschaften von Mauern und Fenstern, Wetterdaten sowie Ober- und Untergrenzen für die gewünschte Raumtemperatur. Durch das Einbeziehen von Strompreisen für Heizung und Kühlung und das Gegenüberstellen von geschätzten Investitionskosten der Dämmung wird eine optimale Stärke der Dämmung berechnet.

Abbildung 3.4 zeigt den Aufbau des Modells überblicksmäßig. Die vier darin gekennzeichneten Teilbereiche werden im Folgenden näher erläutert:

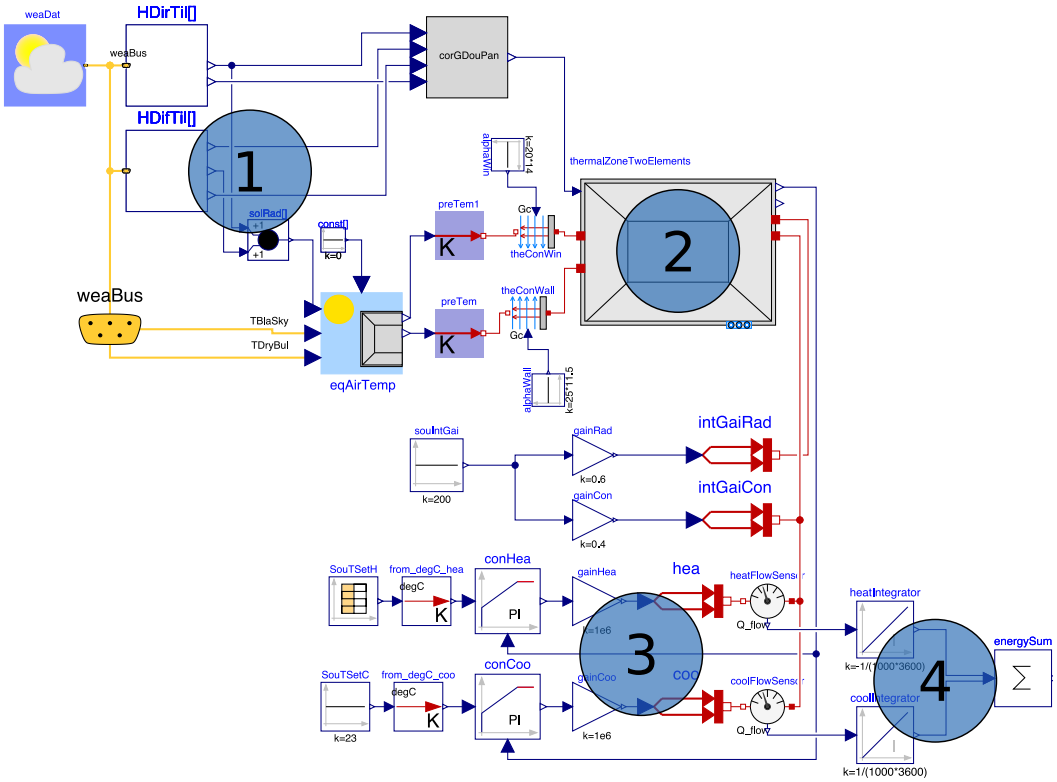


Abbildung 3.4: Schematische Dartstellung des vereinfachten Gebäudemodells

1. Einlesen von Wetterdaten:
   Das Modell erlaubt das Einlesen von Wetterdaten im Format des typischen, meteorologischen Jahres (TMY). Durch Einbeziehen der geografischen Lage eines Gebäudes und dem Abrufen von ortsbezogenen Wetterdaten kann der Realitätsgrad der Simulation erhöht werden.

2. Thermische Zone:
   Dieser Bereich bildet das Gebäude ab. Es werden unter anderem Außen- und Innenwände, Dämmung, Fenster, Sonneneinstrahlung und innere Lufttemperatur miteinbezogen. Hier könnten die Benutzerangaben zu den individuellen Wohneinheiten berücksichtigt werden.

3. Temperaturregelung:
   Regler vergleichen die gewünschten Mindest- und Maximalraumtemperaturen zu jedem Zeitpunkt mit den Modellwerten der thermischen Zone und aktivieren bei Unter- oder Überschreitung eine Kühlung oder Heizung.

4. Integrator
   Am Abschluss der Komponentenkette werden die Leistungswerte zum Gesamtenergieverbrauch aufintegriert und können als Ausgabeparameter abgerufen werden.

Für eine realitätsnahe Simulation ist es notwendig, ein komplexeres Modell sowie eine Mehrzahl an Parametern zu variieren. Dadurch ergibt sich eine mögliche Vervielfachung des Rechenaufwands für eine Parameteroptimierung verglichen mit der Optimierung der Dämmstärke in diesem Modell.

Im folgenden Kapitel werden deshalb Möglichkeiten analysiert, wie moderne Computerhardware besser genutzt werden kann, um die Dauer von derartigen Auswertungen in Grenzen zu halten.

4 Verteilte Simulationsumgebung

Bei der Leistungssteigerung von Computerprozessoren hat man sich in den vergangenen Jahren zunehmend an die physikalischen Grenzen angenähert [43]. Die aktuellen Entwicklungen bewegen sich in Richtung der Verwendung von mehreren Prozessorkernen zur Bewältigung von rechenintensiven Aufgaben. In der Regel kommen in den heute verbreiteten Simulationswerkzeugen jedoch nur einzelne, zentrale Solver zum Einsatz. Trotz Fortschritten in der Entwicklung von Algorithmen und Software bleibt jedoch die grundsätzliche Problematik der schlechten Skalierbarkeit bestehen: die Ausführungszeit der Simulation steigt mit der Größe des Systems überlinear an. [44]

Die Idee der Nutzung mehrerer Rechenkerne für die Computersimulation ist nicht neu. Bereits lange vor der Einführung von Modelica ist es gelungen, durch die Aufteilung eines Modells in Submodelle mit eigenen Solvern, nahezu lineare Skalierbarkeit zu erzielen [44]. Die unter *Transmission Line Modelling* bekannten Technologien können auch mit Modelica eingesetzt werden und versprechen drastische Leistungsverbesserungen. Eine andere Variante zur Nutzbarmachung von mehreren Rechenkernen für Simulationen ist das automatisierte Übersetzen von sequenziellem Simulationscode in eine parallele Version. Mit diesem Bestreben konnte eine 2,5-fache Beschleunigung durch den Einsatz eines 8-Kern Computerclusters erzielt werden [45].

In dieser Arbeit wird Parallelisierung auf einer höheren Ebene eingebracht. Das zu entwickelnde Framework wird eine Standardinstallation von JModelica als Simulationsumgebung verwenden, jedoch wird daraus Nutzen gezogen, dass für die Lösung von Optimierungsproblemen meist eine hohe Anzahl an Simulationsdurchläufen ausgeführt werden, die sich nur

durch unterschiedliche Eingabeparameter unterscheiden. Meist können diese Simulationen voneinander unabhängig, und somit parallel, ausgeführt werden. [46]

Ein weiterer Aspekt der Parallelisierung ergibt sich daraus, dass das Framework vielen Benutzern gleichzeitig zur Verfügung steht und deshalb parallel Optimierungsprobleme mehrerer Benutzer lösen kann. Es soll keine harte Obergrenze hinsichtlich der Anzahl gleichzeitiger Benutzer haben, sondern skalierbar konzipiert werden, sodass das System auch mit vielen Benutzern stets stabil läuft. Analog dazu sollen die verfügbaren Ressourcen optimal zur schnellstmöglichen Verarbeitung der anstehenden Aufgaben genutzt werden können.

Im Folgenden werden Grundlagen für den Aufbau einer skalierbaren, verteilten Simulationsumgebung beleuchtet.

4.1 Simulieren von Modelica-Modellen in JModelica

Die Simulationsumgebung des Optimierungsframeworks dieser Arbeit besteht im Wesentlichen aus einer Installation von JModelica und einer Anzahl an Softwarepaketen, die vorausgesetzt werden. Dazu zählen beispielsweise ein C- und ein Fortran-Compiler, eine Python-Laufzeitumgebung sowie verschiedene Bibliotheken [47]. Ein mitgeliefertes Shell-Skript zum Starten des Python-Interpreters sorgt dafür, dass Umgebungsvariablen zum Auffinden von Python-Modulen und anderen Abhängigkeiten gesetzt werden. Mittels Python-Code können daraufhin Modelica-Modelle als Textdateien oder eingebettet in Functional Mock-up Units importiert, exportiert sowie simuliert und optimiert werden.

JModelica bietet eine Vielzahl an Konfigurationsmöglichkeiten. Es ist daher nicht das Ziel dieser Arbeit, eine Abstraktion über die bestehende Programmierschnittstelle zu bauen. Vielmehr wird eine Basis geschaffen, die sich mit geringer Anpassung für verschiedene Zwecke einsetzen lässt. Der Programmcode, der Modelle lädt, kompiliert und simuliert, ist austauschbar und wird auf die Art und die Parameter der Modelle abgestimmt.

Listing 4.1 zeigt beispielhaft, dass das Laden, Kompilieren und Simulieren des Modells der Rosenbrock Funktion aus 3.1 mit nur wenigen Zeilen Python-Code bewerkstelligt werden kann.

```
1  def simulate(x, model):
2      # set input, simulate, retrieve and return output
3      output = model.simulate(final_time=0, \
4          input=(['x1', 'x2'], lambda t: x))
5      return output['f'][-1]
6
7  if __name__ == "__main__":
8      # compile the model and store it as FMU
9      from pymodelica import compile_fmu
10     fmu_path = compile_fmu("Rosenbrock", "rosenbrock.mo")
11
12     # load FMU for simulation
13     from pyfmi import load_fmu
14     model = load_fmu(fmu_path)
15
16     f = simulate([1, 1], model)
17     print(f) # prints 0.0
```

Listing 4.1: Simulieren des Rosenbrock-Modells

4.2 Parallele Programmierung in Python

Der eingangs in diesem Kapitel erwähnten Problematik der schlechten Skalierbarkeit von zentralen Solvern wird mit Möglichkeiten der parallelen Programmierung in Python entgegengewirkt. Das Ziel ist es, die Startparameter x_i des Modells der Rastrigin-Funktion aus 3.1 im Wertebereich $x_i \in [-5.12, 5.12]$ zu verteilen und durch Anwenden einer Minimierungsfunktion auf jede Startparameterkombination mindestens einmal auf das globale Minimum zu stoßen. Diese Vorgänge sind voneinander unabhängig und können unter in Anspruchnahme mehrerer Rechenkerne ganz oder teilweise parallel ausgeführt werden.

4.2.1 Parameteroptimieren mit Multi-Threading

Die am häufigsten verwendete Methode der parallelen Programmierung ist Multi-Threading. Threads bzw. Ausführungsfäden können im Allgemeinen auf mehrere Prozessorkerne verteilt und parallel abgearbeitet werden. Sie teilen sich die Ressourcen und den Adressraum ihres Prozesses und können diesen zum Datenaustausch nutzen. Der Wechsel zwischen Thread-Kontexten geht viel schneller vonstatten als der Kontextwechsel zwischen Prozessen.

Die Referenzimplementation des Python-Interpreters *CPython* bringt im Zusammenhang mit Multi-Threading jedoch einen gravierenden Nachteil mit: sie ist nicht threadsicher und erlaubt deshalb lediglich einem Thread die Ausführung von Bytecode zu einem Zeitpunkt. Viele Entwickler greifen aus diesem Grund zu Multi-Processing zurück.
[48]

Python bringt in der Standardbibliothek das Modul *threading* mit. Der zentralen Klasse *Thread* übergibt man im Konstruktor unter anderem die Zielfunktion sowie optionale Argumente. Durch Aufruf der Instanzmethode *start* wird mit der Ausführung des Threads begonnen. Mit der Methode *join*, wartet man die Beendigung des Threads ab[1].

Im Folgenden werden besondere Codeausschnitte zum parallelen Optimieren des Rastrigin-Modells aus 3.2 gezeigt.

Wählen der Startparameter

Die Minima der Rastrigin-Funktion sind symmetrisch angeordnet. Das globale Minimum befindet sich genau im Zentrum des definierten Bereichs, an der Position $x = (0, 0)$. Als triviale Herangehensweise, dieses mit Optimierungsmethoden zu finden, wird ein Raster über die Fläche der Funktion gespannt. Jeder Kreuzungspunkt des Rasters dient als Startparameter für einen Optimierungsdurchlauf.

[1] `https://docs.python.org/2.7/library/threading.html`

In Listing 4.2 wird das Berechnen von Startwerten über den definierten Funktionsbereich veranschaulicht. Mit diesem Vorgehen wird das globale Minimum ab einer Verteilung von 8x8 Startwerten gefunden.

```
    # define the grid of the initial guesses x0
    division = int(sys.argv[1]) if len(sys.argv) > 1 else 8
    x1 = numpy.linspace(-4, 4.5, division)
    x2 = numpy.linspace(-4.3, 3.9, division)
    X1, X2 = numpy.meshgrid(x1, x2)
    x0s = zip(X1.flatten(), X2.flatten())
    # x0s: [(-4.0, -4.298), (-2.786, -4.298), ...]
```

Listing 4.2: Erzeugen eines Netzes aus Startparametern

Simulieren des Modells

Listing 4.3 zeigt eine für Multi-Threading angepasste Variante der in Listing 4.1 verwendeten Wrapper-Funktion *simulate*. Zum einen wird die je Thread erzeugte Modellinstanz im Normalfall mehrfach zur Simulation verwendet und muss deshalb vor jedem Simulationsvorgang zurückgesetzt werden. Zum anderen muss die Option *"result_handling"* angepasst werden, um das Erstellen von kollidierenden, temporären Dateien zu verhindern.

```
def simulate(x, model):
    model.reset()
    options = model.simulate_options()
    # don't create result files in order not to interfere with
       other threads
    options['result_handling'] = 'memory'
    # suppress solver output
    options["CVode_options"]["verbosity"] = 50 # quiet
    input_ = (['x1', 'x2'], lambda t: x)
    output = model.simulate( \
        final_time=0, input=input_, options=options)
    return output['f'][-1]
```

Listing 4.3: Simulationsfunktion adaptiert für parallele Programmierung

Minimieren der Parameter

Aus Gründen der übersichtlichen Programmstruktur wird der Aufruf und das Setzen der Argumente der Optimierungsfunktion in eine einfache Funktion mit dem Namen *optimize* eingebettet (siehe Listing 4.4). In dieser Funktion werden außerdem die minimierten Parameter sowie der dazugehörige Funktionswert aus dem Ergebnisobjekt extrahiert und retourniert.

```
1 def optimize(x0, model):
2     from scipy.optimize import minimize
3     result = minimize(simulate, \
4         x0, \
5         args=model, \
6         method='nelder-mead', \
7         tol=1e-6)
8     return result.x, result.fun
```

Listing 4.4: Wrapper-Funktion zum Optimieren eines Modells

Initialisieren von Multi-Threading

Nachdem die Startwerte der Parameteroptimierungen in einer Liste vorliegen, werden zwei Variablen zur Speicherung des Ergebnisses deklariert. Da die Threads direkt darauf schreiben können, wird zudem ein Objekt zur Zugriffssperre verwendet. Jeder Thread entnimmt dieser Liste Wert für Wert und führt Optimierungen durch. Während seiner Laufzeit speichert er das kleinste gefundene Minimum in lokalen Variablen. Erst wenn alle Startwerte konsumiert wurden, gleichen die Threads ihre gefundenen Minima mit den globalen Variablen ab.

Für das Entnehmen eines Startwerts und das abschließende Synchronisieren des Ergebnisses beantragen die Threads exklusiven Zugriff über das Sperrobjekt (siehe Listing 4.5). Obwohl dies nur am Beginn und am Ende der Laufzeit eines Threads passiert, zeigt sich in den Ergebnissen unter 6.1, dass die Parallelisierung mittels Multi-Threading für diesen Anwendungsfall keinen Vorteil bringt. Tatsächlich verlängert sich die Laufzeit gegenüber der sequenziellen Variante.

```
    # global variables and a lock object
    global_min_x, global_min_f = None, None
    lock = Lock()

    def thread_init():
        # load model once in each thread
        model = pyfmi.load_fmu(fmu_path)

        # use thread-local variables, only sync at the end
        local_min_f, local_min_x = None, None
        while True:
            # take a start value, escape if none left
            with lock:
                if len(x0s) == 0:
                    break
                x0 = x0s.pop(0)
            # perform the optimization
            x, f = optimize(x0, model)
            # keep track of the local minimum
            if local_min_f is None or f < local_min_f:
                local_min_x, local_min_f = x, f

        # merge the local results with the global ones
        global global_min_f, global_min_x
        if local_min_f is not None:
            with lock:
                if global_min_f is None or \
                local_min_f < global_min_f:
                    global_min_x = local_min_x
                    global_min_f = local_min_f

    # create and start one thread per physical cpu core
    thread_count = min(cpu_count(logical=False), division)
    threads = [Thread(target=thread_init)
        for t in xrange(thread_count)]
    [thread.start() for thread in threads]
    [thread.join() for thread in threads]
```

Listing 4.5: Ausführen der Parameteroptimierung mit Multi-Threading

4.2.2 Parameteroptimieren mit Multi-Processing

Eine Alternative zu Multi-Threading ist Multi-Processing. Anstatt leichtgewichtige Threads zur nebenläufigen Abarbeitung des Programms zu erzeugen, werden eigenständige Prozesse mit separatem Adressraum gestartet [48]. Auf Linux-Systemen wird dafür der Elternprozess kopiert, wodurch Variablenwerte in die Kindprozesse übertragen werden.

Python bietet in seiner Standardbibliothek das Modul *multiprocessing*[2] an. Die Programmierschnittstelle ist der des Moduls *threading* sehr ähnlich. Für die Kommunikation zwischen Prozessen stehen die Klassen *Pipe* und *Queue* zur Verfügung. Mit Pipes kann man bidirektionale Datenströme zwischen zwei Prozessen aufbauen. Queues stellen eine prozessübergreifende Datenstruktur dar, die auch von mehr als zwei Prozessen zum Datenaustausch genutzt werden kann.

Für die Anwendung von Multi-Processing zum Simulieren des Rastrigin-Modells sind nur geringfügige Anpassungen gegenüber der Multi-Threading-Variante notwendig. Die Start-Funktion eines Kindprozesses erhält die Instanz des Modells sowie zwei Queue-Instanzen, zum Abrufen von Startparametern und zum Zurückgeben des Ergebnisses (siehe Listing 4.6).

```
1  def process_init(x0_queue, result_queue, model):
2      # use local variables to keep track of the minimum
3      min_x, min_f = None, None
4      while True:
5          try:
6              # take a start value if available and optimize
7              x0 = x0_queue.get_nowait()
8              x, f = optimize(x0, model)
9              if min_f is None or f < min_f:
10                 min_x, min_f = x, f
11         except Empty:
12             # return the local result to the parent process
13             result_queue.put((min_x, min_f))
14             break
```

Listing 4.6: Initialisieren eines Kindprozesses zur Parameteroptimierung

[2] https://docs.python.org/2/library/multiprocessing.html

Durch das Duplizieren des Elternprozesses beim Erstellen der Kindprozesse braucht das Modell lediglich einmal am Programmstart geladen zu werden. Eine Queue-Instanz wird mit den gestreuten Startwerten befüllt und steht allen Prozessen gemeinsam zur Verfügung. Für die Anzahl der zu erstellenden Kindprozesse wird die Anzahl an logischen Prozessorkernen, multipliziert mit dem Faktor *1,5*, herangezogen (siehe Listing 4.7). Dies verlängert unter Umständen die Dauer einzelner Teiloptimierungen, da bestimmte Rechenkerne zwischen der Abarbeitung zweier Aufgaben wechseln müssen, es reduziert jedoch die Fälle, bei denen gegen Ende der Programmlaufzeit einige Kerne unausgelastet sind, während wenige Kerne die verbleibenden Teiloptimierungen abarbeiten.

Nachdem die Kindprozesse ihre ermittelten Minima in eine weitere Queue-Instanz abgelegt und die Ausführung beendet haben, wird das kleinste Minimum vom Elternprozess zum globalen Minimum gekürt.

```
1      # compile and load the model in the parent process
2      fmu_path = pymodelica.compile_fmu( \
3          "Rastrigin", "rastrigin.mo")
4      model = pyfmi.load_fmu(fmu_path)
5
6      # use queues for start parameters and for results
7      x0_queue = Queue()
8      [x0_queue.put(x0) for x0 in x0s]
9      result_queue = Queue()
10
11     # create one process per logical cpu core
12     process_count = min(int(cpu_count(logical=True)*1.5),
           division)
13     processes = [Process(target=process_init, \
14         args=(x0_queue, result_queue, model))
15         for p in xrange(process_count)]
16     [process.start() for process in processes]
17     [process.join() for process in processes]
18
19     # merge the results of the child processes
20     min_x, min_f = None, None
21     while not result_queue.empty():
22         x, f = result_queue.get_nowait()
23         if min_f is None or f < min_f:
24             min_x, min_f = x, f
```

Listing 4.7: Ausführen der Optimierung mit Multi-Processing

Die Ergebnisse unter 6.1 belegen eine deutliche Effizienzsteigerung des Programmes in der Multi-Processing-Variante gegenüber der Multi-Threading und der sequenziellen Variante.

4.3 Verteilte Anwendungen mit Python

Für bestimmte Anwendungen stellen die Kapazitäten einzelner Computer einen limitierenden Faktor dar. Abhilfe schafft in diesesm Fall das Konzept eines *verteilten Systems*. Ein verteiltes System ist ein Zusammenschluss einer potenziell hohen Anzahl an über Netzwerk verbundenen, autonomen Recheneinheiten, die gegenüber ihrer AnwenderInnen als einziges, einheitliches System erscheinen. Eine Recheneinheit kann ein Hardwaregerät oder aber auch ein Softwareprozess sein.

In der Literatur werden verteilte Systeme weiter unterteilt in *Cluster Computing*, *Grid Computing* und *Cloud Computing*. Cluster Computing beschreibt eine Zusammenstellung von gleichartigen Arbeitsplatzrechnern, die durch ein Hochgeschwindigkeitsnetzwerk miteinander verbunden sind. In Grid Computing sind die einzelnen Einheiten etwas ungebundener, können unter verschiedene Verwaltungsbereiche fallen, sowie unterschiedliche Hardware, Software und Netzwerktechnologien nutzen. Cloud Computing bezeichnet das Konzept, Einrichtungen bereitzustellen, mit denen die benötigte Infrastruktur aus verschiedenen Diensten zusammengestellt werden kann.
[6]

Die abschließende Aufgabe dieses Kapitels ist es, die Parameteroptimierung des Rastrigin-Modells weiter zu parallelisieren, sodass sie auf mehreren Computern verteilt ausgeführt werden kann.

4.3.1 Programmbibliotheken für verteiltes Rechnen

In den vergangenen Jahren ist eine Vielzahl an Programmbibliotheken für verteiltes Rechnen entstanden. Unterschiede sind hinsichtlich verschiedener Charakteristiken auszumachen, wie etwa:

- Funktionsumfang
- Popularität, Verbreitung, Community
- Unterstützung von Programmiersprachen und Betriebssystemen
- Abhängigkeit von Drittanbietersoftware
- Aktive oder bereits eingestellte Entwicklung

Die Anforderungen für die Parameteroptimierung der Modelica-Modelle an eine derartige Programmbibliothek sind überschaubar: es muss, wie in den Varianten mit Multi-Threading und Multi-Processing, lediglich möglich sein, Arbeitsschritte (*Tasks*) zur alsbaldigen Ausführung in Auftrag zu geben und deren Ergebniswerte abwarten zu können. Im Folgenden werden zwei Projekte, die für diese Aufgabe geeignet sind, vorgestellt:

Ray

Ray ist ein verteiltes System zur Bewältigung von Herausforderungen im Zusammenhang mit aufkommenden KI-Anwendungen. Es wurde mit der Absicht geschaffen, Millionen von nebenläufigen Aufgaben pro Sekunde starten und verwalten zu können und dabei Latenzzeiten im Millisekundenbereich zu bewahren. Die Länge einzelner Arbeitsschritte ist unerheblich und kann von wenigen Millisekunden bis zu mehreren Stunden betragen.

Auf Anwendungsebene gibt es drei Arten von Prozessen: *Driver*, *Worker* und *Actor*. Der Driver-Prozess führt das Anwenderprogramm aus. Worker-Prozesse erhalten von der Systemebene Aufgaben zugeteilt. Actor-Prozesse sind zustandsorientiert und werden von Driver- oder Worker-Prozessen explizit beauftragt.

Die Systemebene besteht aus einem *Distributed Scheduler*, einem *Distributed Object Store* und einem *Global Control Store*. Der Aufgabenplaner ist für die Verteilung der Aufgaben an die Netzwerkknoten zuständig. Die beiden Datenspeicherknoten dienen der Ausfallsicherheit und der Minimierung der Latenzzeiten. [49]

Dask

Dask wurde mit der Intention geschaffen, weit verbreitete Programmbibliotheken für wissenschaftliche Anwendungen durch kompatible Nachbildungen abzulösen. Bestimmte Datenstrukturen und Programmschnittstellen von *NumPy*[3], *Pandas*[4] und *PyToolz*[5] wurden repliziert, um von verteilten Systemen und moderner Hardware profitieren zu können. Dask ist leichtgewichtig und kann einfach mit dem Paketverwaltungsprogramm *pip* installiert werden. [50]

Im Programmpaket enthalten ist ein zentral verwalteter, verteilter, dynamischer Aufgabenplaner, genannt *Dask.distributed*[6]. Ähnlich wie Ray enthält die Architektur von Dask einen *Scheduler*-Prozess, mehrere *Worker*-Prozesse und einen Einsprungpunkt, die Klasse *Client*. Die Kommunikation zwischen den Prozessen erfolgt mit Netzwerkprotokollen. [51]

Weitere Pakete zur Aufgabenverwaltung

Im Zuge der Recherche wurden einige weitere, für die Aufgabenstellung potenziell anwendbare Programmpakete identifiziert. Darunter zählen *Celery*[7], *SCOOP*[8], *Luigi*[9], *dramatiq*[10], *huey*[11] und *dispy*,[12]. Mit Dask und Ray kamen jedoch bereits zwei ausgezeichnet geeignete Projekte in die engere Auswahl. Schlussendlich fiel die Entscheidung aufgrund bisheriger Erfahrungen des GameOpSys-Projektteams auf die Verwendung von Dask.

[3] https://numpy.org
[4] https://pandas.pydata.org
[5] https://toolz.readthedocs.io
[6] https://distributed.dask.org
[7] http://www.celeryproject.org
[8] https://github.com/soravux/scoop
[9] https://github.com/spotify/luigi
[10] https://github.com/Bogdanp/dramatiq
[11] https://github.com/coleifer/huey
[12] https://pgiri.github.io/dispy

4.4 Parameteroptimieren mit Dask

Im Folgenden wird beschrieben, wie die zuvor mit Multi-Threading und Multi-Processing umgesetzte Parameteroptimierung des Rastrigin-Modells mittels Dask.distributed verteilt ausgeführt werden kann.

4.4.1 Dask-Scheduler als Aufgabenplaner

Die zentrale Komponente des Experiments stellt eine Instanz des Schedulers dar. Für gewöhnliche Anwendungsfälle reicht es aus, die mit dem Paket installierte, eigenständige Kommandozeilenanwendung ohne Angabe von Argumenten auf einem der Computer auszuführen.

```
$ dask-scheduler
```

Alternativ wäre es möglich, eine Scheduler-Instanz von einem Python Programm aus zu konfigurieren und zu starten.

4.4.2 Dask-Worker als JModelica-Umgebung

Die tatsächliche Abarbeitung der Arbeitsschritte wird von Workern durchgeführt. Sie stellen zunächst eine Netzwerkverbindung zum Scheduler her und bekunden ihre Verfügbarkeit. Auf ihnen müssen alle zur Simulation benötigten Softwarekomponenten installiert und die benötigten Umgebungsvariablen gesetzt sein. Auch für diese Komponente wird von Dask eine eigenständige Kommandozeilenanwendung mitgeliefert.

Zum Starten des Workers als Simulationsumgebung wird das von JModelica installierte Startskript lediglich dahingehend angepasst, dass anstatt des Python-Interpreters die Anwendung *dask-worker* ausgeführt wird. Als Startargument wird zumindest die Netzwerkadresse der Scheduler-Instanz erwartet. Es besteht auch die Möglichkeit, die Anzahl der verwendeten Threads vorzugeben. Ohne Angabe wird laut Dokumentation als Standardwert die Anzahl der verfügbaren Rechenkerne verwendet. Aufgrund der Erkenntnisse aus dem Versuch mit Mutli-Threading liegt es jedoch nahe,

die Anzahl der Threads auf maximal einen einzigen zu reduzieren. Im Gegenzug wird für eine optimale Ressourcennutzung jedoch die Anzahl der zu verwendeten Prozesse an die Anzahl der logischen Prozessorkerne angepasst (siehe Listing 5.5).

4.4.3 Verteiltes Ausführen der Parameteroptimierung

Die letzte, jedoch nicht minder wichtige Komponente zur verteilten Parameteroptimierung ist das Anwenderprogramm, welches dem Scheduler die auszuführenden Aufgaben erteilt. Im Unterschied zu den vorangegangenen Varianten wird das Kompilieren des Modells zu einer FMU nicht vom Einstiegsprogramm selbst, sondern bereits von einem Worker-Prozess durchgeführt.

Bevor Dateien von Workern gelesen werden können, müssen sie explizit an den Scheduler zur Verteilung hochgeladen werden. Dies trifft in diesem Fall für die Modelldefinition *rastrigin.mo* sowie die FMU zu und wird mit der Funktion *upload_file* der Klasse *dask.distributed.Client* bewerkstelligt.

Im Anschluss an die Verteilung der FMU wird die Optimierung ausgelöst. Hierzu wird die Liste der Startparameter unter Zuhilfenahme einer Abbildungsfunktion *map* auf eine Hilfsroutine angewandt (siehe Listing 4.8).

```
1  if __name__ == '__main__':
2      scheduler = sys.argv[1]
3      print(str.format('scheduler: {}', scheduler))
4
5      # connect to the dask scheduler
6      from dask.distributed import Client
7      client = Client(sys.argv[1])
8
9      # upload the model file to the scheduler
10     client.upload_file('rastrigin.mo')
11
12     # perform the compilation on a worker, it will upload the
           FMU to the scheduler
13     fmu_file_name = client.submit(compile_and_upload_fmu, \
14         'Rastrigin', 'rastrigin.mo').result()
15
```

```
16      # calculate the list of start values and distribute the call
            of load_fmu_and_optimize for each one
17      division = int(sys.argv[2]) if len(sys.argv) > 1 else 8
18      x0s = calculate_x0s(division=division)
19      future = client.map(load_fmu_and_optimize, x0s, \
20          fmu_file_name=fmu_file_name)
21      results = client.gather(future)
22      client.close()
23
24      # find the global minimum
25      min_x, min_f = min(results, key=lambda res: res[1])
26
27      print(str.format('minimum f = {} found at x = {}', \
28          round(min_f, 3), \
29          [round(min_xi, 3) for min_xi in min_x]))
```

Listing 4.8: Hauptroutine zum Starten der Parameteroptimierung

Die vom Scheduler verteilten Dateien werden auf den Workern in temporären Verzeichnissen gespeichert. In der Funktion *compile_and_upload_fmu* wird dafür gesorgt, dass die Modelldefinition einmalig kompiliert und die FMU danach jedem Worker zugespielt wird (siehe Listing 4.9).

```
1  def compile_and_upload_fmu(class_name, model_file_name):
2      from dask.distributed import get_worker, get_client
3      worker_dir = get_worker().local_dir
4      model_path = os.path.join(worker_dir, model_file_name)
5      fmu_path = pymodelica.compile_fmu(class_name, \
6          model_path, \
7          compile_to=worker_dir)
8
9      # distribute the fmu to all workers
10     get_client().upload_file(fmu_path)
11
12     fmu_file_name = os.path.basename(fmu_path)
13     return fmu_file_name
```

Listing 4.9: Kompilieren der Modelldefinition im Worker-Prozess

Die Funktion *load_fmu_and_optimize* wird bereits von jedem Worker abgearbeitet. Sie übernimmt das Laden der empfangenen FMU in den Speicher und das Aufrufen der Optimierungsfunktion mit den erhaltenen Startparametern (siehe Listing 4.10).

```
1 def load_fmu_and_optimize(x0, fmu_file_name):
2     # load model at each worker
3     from dask.distributed import get_worker
4     worker_dir = get_worker().local_dir
5     fmu_path = os.path.join(worker_dir, fmu_file_name)
6     model = pyfmi.load_fmu(fmu_path)
7     # optimize and return result
8     x, f = optimize(x0, model)
9     return x, f
```

Listing 4.10: Initialisieren eines Worker-Prozesses

Für die Simulation des Modells und die Optimierung der Parameter kommen die bereits vorgestellten Funktionen *simulate* und *optimize* unverändert zum Einsatz (siehe Listing 4.3 und 4.4).

Im Gegensatz zur Multi-Threading-Variante entsprechen die Leistungsdaten der Multi-Processing-Variante den Erwartungen. Details sind unter 6.2 zusammengefasst.

5 Skalierbares Optimierungsframework

Das aktuelle Kapitel befasst sich mit dem Aufbau eines Frameworks, das über eine *Web-Programmierschnittstelle (Web API)* Aufträge zur Parameteroptimierung von Modelica-Modellen entgegennimmt, sie auf den verfügbaren Recheneinheiten verteilt ausführt und deren Ergebnisse nach Fertigstellung zum Abruf bereithält.

5.1 Skalierbarkeit und Cloud Computing

Bei der Konzeption des Frameworks, dessen Aufbau und dem Zusammenwirken der verschiedenen Komponenten, wird besonderes Augenmerk auf den Aspekt der *Skalierbarkeit* gerichtet. Skalierbarkeit ist in modernen Anwendungen von zentraler Bedeutung und deshalb ein wichtiger Erfolgsfaktor für viele Unternehmen [52]. Ausreichend Ressourcen bereitzustellen, die auch Lastspitzen bewältigen, kann nicht nur schwer vorausplanbar, sondern auch sehr kostspielig sein.

Mit Cloud Computing hat sich ein mächtiges Datenverarbeitungsparadigma etabliert. Eingehende Aufgaben werden in Rechenzentren einer Kombination aus Netzwerkverbindungen, Software und Netzwerkdiensten zugeteilt. Der Rechenaufwand wird von großangelegten, verteilten Computersystemen gestemmt. AnwenderInnen steht Rechenleistung auf Supercomputerniveau zur Verfügung, und das jederzeit auf Abruf. Zusammen mit flexiblen Preismodellen, die eine Verrechnung auf Basis von tatsächlich genutzten Ressourcen je Zeiteinheit erlauben, stellt Cloud Computing eine attraktive

Alternative zur Vor-Ort-Infrastruktur sowie zur Einmietung bei Hosting-Anbietern dar. Die optimale Nutzung dieser flexiblen Infrastruktur, bei der je nach Bedarf neue Ressourcen hinzugefügt oder entfernt werden können, bedarf einer geeigneten, auf Skalierbarkeit ausgelegeten Softwarearchitektur.

Im professionellen Umfeld werden Anwendungen häufig *dynamisch* skalierbar ausgelegt. Dazu werden bestimmte Kennwerte laufend überwacht und anhand von Indikatoren die Entscheidungen gefällt, zu welchem Zeitpunkt und in welchem Ausmaß das System zur Laufzeit erweitert oder reduziert wird. In dieser Arbeit wird auf eine dynamische Skalierung verzichtet. Das Framework ist flexibel hinsichtlich der Anzahl der redundanten Komponenten, jedoch muss die Struktur einmalig beim Start festgelegt und kann zur Laufzeit nicht mehr angepasst werden.

5.2 Containervirtualisierung

Hochleistungsrechenzentren sind für das Durchführen von anspruchsvollen Rechenaufgaben sehr gefragt. Das Einrichten und Administrieren eines Rechnerverbunds zum Betrieb eines verteilten Systems ist traditionell jedoch ein anspruchsvoller und zeitintensiver Vorgang. In den vergangenen Jahren hat sich dies durch das Etablieren des Konzepts der Containervirtualisierung geändert. Bei Containervirtualisierung wird eine Software zusammen mit einem Dateisystem in eine Abbild-Datei eingebettet. Das Dateisystem beinhaltet alle zur Ausführung des Programmes notwendigen Komponenten: Programmcode, Laufzeitumgebung, Systemwerkzeuge und Systembibliotheken [52].

Das quelloffene Paket *Docker*[1] zählt aktuell zu den populärsten Vertretern im Bereich der Containervirtualisierung. Es existiert bereits eine Vielfalt an öffentlich verfügbaren Containerabbildern (*Docker Images*) oder Abbildefinitionen (*Dockerfiles*), die verschiedene Softwarepakete miteinander bündeln. Zudem bringt Docker mit *Docker Swarm mode* eine freie Lösung für die Orchestrierung der Container auf mehreren Rechnern mit, die nicht von

[1] `https://www.docker.com`

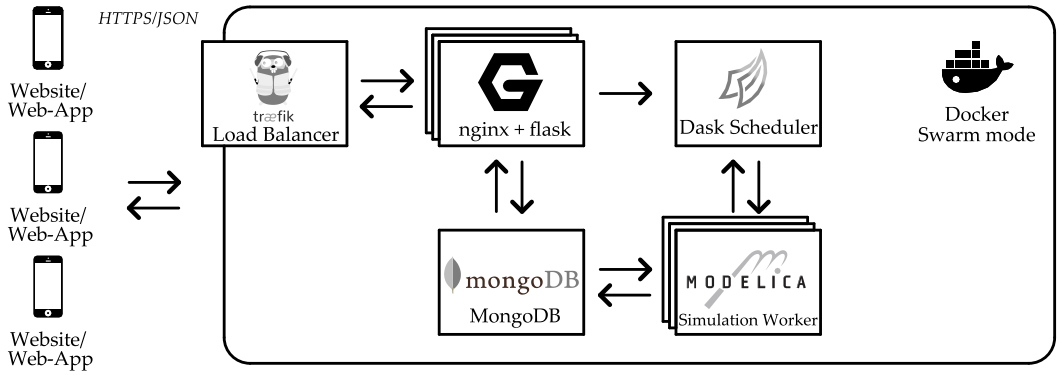


Abbildung 5.1: Übersicht über die Komponenten des Frameworks

Drittanbietersoftware abhängig ist [52]. Aus diesem Grund wird Docker für die Entwicklung und Bereitstellung des Optimierungsframeworks verwendet.

5.3 Komponenten und Interaktionen

5.3.1 Übersicht

Zur Erklärung der Struktur des Frameworks werden seine Bestandteile in drei Hauptkomponenten unterteilt: der Web API, der Datenspeicherung und der Aufgabenverteilung. Abbildung 5.1 stellt die Komponenten schematisch dar, welche im Folgenden näher beschrieben werden.

Web API

Die Schnittstelle zwischen einem Anwender und dem Optimierungsframework bildet eine Web API im REST-Paradigma. Dahinter beantwortet ein Webserver die eingehenden Anfragen und bedient sich der Datenbank zur dauerhaften Speicherung der Optimierungsaufträge. Zum Anstoßen einer in Auftrag gegebenen Parameteroptimierung wird ein Initialisierungsprogramm an den Dask-Scheduler geschickt. Dieses wird auf einem Worker

ausgeführt, liest zu verarbeitende Aufträge aus der Datenbank und führt die Optimierungen durch.

Eine Skalierung wird auf der Ebene der Web API dadurch ermöglicht, dass beliebig viele Instanzen des Webservers koexistieren und ihren Dienst gleichzeitig verrichten können. Um nach außen hin als einheitliches System aufzutreten und die Anfragen auf intelligente Art auf die verfügbaren Ressourcen verteilen zu können, wird eine Lastverteilung (*Load Balancing*) vorgeschalten. Die Lastverteilung muss über die Anzahl und die Erreichbarkeit der verfügbaren Webserverinstanzen Bescheid wissen.

Datenspeicherung

Eine zentrale Rolle im Framework nimmt die Datenbank ein. Beim Eintreffen eines neuen Auftrags werden ihr das Modell und die Startparameter von einer Webserverinstanz übergeben. Sobald Rechenkapazität zum Durchführen der Parameteroptimierung vorhanden ist, wird dieser Datensatz von einer JModelica-Umgebung zunächst abgerufen und später mit Ergebnissen befüllt. Ein Webserver kann sich auf Anfrage eines Anwenders stets nach den Ergebnissen eines Auftrags erkundigen.

Aufgabenverteilung

Der Dask-Scheduler hat, wie die Datenbank, eine zentrale Rolle im Framework. Laut Dokumentation [53] ist Dask in der Lage, als Verbund auf Hunderten von Maschinen und Tausenden von Prozessorkernen zu laufen. Diese Kapazität wird als ausreichend erachtet, weshalb auf eine zusätzlich redundante Auslegung des Schedulers verzichtet wird.

Die eigentliche Skalierung findet auf Ebene der Dask-Worker statt. Es wird eine Worker-Instanz je Rechner gestartet, welche wiederum je einen Prozess pro logischem Prozessorkern erstellt. Ein Worker verbindet sich zunächst mit dem Scheduler, um seine Verfügbarkeit anzumelden. Zur Durchführung einer Parameteroptimierung stellt der Worker zudem auch eine Verbindung mit der Datenbank her.

Webanwendung

Für eine erleichterte Bedienung der Web API während der Entwicklung, aber auch zur Demonstration der Möglichkeiten, wird eine rudimentäre Webanwendung auf den Webservern bereitgestellt. Sie ermöglicht das Erstellen eines Benutzerkontos, das Starten von Simulationen sowie das Anzeigen von Ergebnissen.

5.3.2 Entwicklungs- und Bereitstellungsumgebung

Eine in der Open-Source-Community aktive[2] Persönlichkeit, Sebastián Ramírez, hat häufig verwendete Softwarekombinationen in Entwicklungs- und Bereitstellungsumgebungen als Vorlage zum Verwenden in eigenen Projekten frei zur Verfügung gestellt. Das GitHub-Projekt *tiangolo/flask-frontend-docker*[3] bietet eine optimale Grundlage für den Aufbau des Optimierungsframeworks.

Die Projektvorlage bündelt das Python-basierte Webframework *Flask*[4] mit *nginx*[5] als vorgeschalteten Webserver sowie den Reverse-Proxy und Lastverteiler *Traefik*[6]. Es bietet zahlreiche Konfigurationsmöglichkeiten und bringt Skripte für eine einfache Bereitstellung auf einem lokalen Computer oder einem Rechnerverbund mit. Die zusätzlich benötigten Container für die Datenbank und die Aufgabenverteilung werden in einem späteren Schritt hinzugefügt.

Um ein Projekt von der Projektvorlage zu erstellen, wird das Programm *Cookiecutter* mit der Repository-URL aufgerufen. Es folgen Abfragen zur Anpassung der Konfiguration. Als Projektname wird *GameOpSys* angegeben, von den verbleibenden Einstellungen werden die vorgeschlagenen Standardwerte übernommen.

[2] https://tiangolo.netlify.com/projects
[3] https://github.com/tiangolo/flask-frontend-docker
[4] https://palletsprojects.com/p/flask/
[5] https://nginx.org/
[6] https://traefik.io/

```
$ cookiecutter https://github.com/tiangolo/flask-frontend-docker
project_name [Base Project]: GameOpSys
project_slug [gameopsys]:
domain_main [gameopsys.com]:
backend_cors_origins [http://localhost, http://localhost:4200, http://
    localhost:3000, http://localhost:8080, http://dev.gameopsys.
    com, https://gameopsys.com, http://local.dockertoolbox.
    tiangolo.com, http://localhost.tiangolo.com]:
docker_swarm_stack_name_main [gameopsys-com]: gameopsys
traefik_constraint_tag [gameopsys.com]:
traefik_public_network [traefik-public]:
traefik_public_constraint_tag [traefik-public]:
docker_image_prefix []:
docker_image_backend [backend]:
docker_image_frontend [frontend]:
```

Cookiecutter erstellt ein Verzeichnis mit dem Namen *gameopsys* und folgendem Inhalt:

- *Docker Compose*-Dateien zum Erzeugen und Starten des Container-Clusters
- Bereitstellungsskripte zum Starten des Clusters auf einem Docker Swarm im Produktivbetrieb
- eine Vorlage für eine Webanwendung mit *Vue.js* im Ordner *frontend*
- Konfigurationsdateien für eine Web API mit *Flask*
- eine umfangreiche Anleitung für die Nutzung dieser Projektvorlage

Das neu erstellte Projekt kann mit dem Befehl

```
$ docker compose up
```

gestartet werden. In den folgenden Abschnitten werden die Komponenten des Frameworks sowie die Anpassungen und Erweiterungen des vorgegebenen Projekts bis zum funktionsfähigen Framework erläutert.

5.3.3 Traefik — HTTP Reverse-Proxy und Lastverteiler

Andrew S. Tanenbaum und Maarten van Steen definieren die Einheitlichkeit *(Single coherent system)* als Charakteristik für verteilte Systeme. Endanwender sollen nicht bemerken, dass sie es mit auf Computernetzwerken verteilten Prozessen und Daten zu tun haben [54, S. 4].

Traefik ist ein Reverse-Proxy, der von einer Open-Source-Gemeinschaft entwickelt wird. Er verbirgt Dienste des internen Netzwerks hinter einem Gateway und leitet eingehenden Netzwerkverkehr gezielt auf lokale Server weiter. Durch dieses Prinzip wird eine Lastverteilung auf redundante Serverinstanzen möglich [55, S. 1].

Traefik bringt eine weitreichende Unterstützung für Drittanbietersoftware[7] mit, darunter auch Docker und Swarm mode. Es wird von den Entwicklern unter anderem bereits vorinstalliert als Docker-Image zum Download[8] angeboten. In der verwendeten Projektvorlage ist Traefik für die Verteilung der Last auf alle verfügbaren Webserverinstanzen vorkonfiguriert. Es sind daher keine weiteren Anpassungen notwendig.

5.3.4 Flask — Webapplikationsframework

Flask ist ein leichtgewichtiges Framework für Webanwendungen. Es ist darauf ausgelegt, Entwicklern einen schnellen und einfachen Entwicklungsstart zu ermöglichen und trotzdem für wachsende, komplexe Anwendungen geeignet zu sein[9].

In der Projektvorlage wird Flask in Kombination mit dem Webserver nginx betrieben. Die notwendigen Konfigurationen dafür sind bereits voreingestellt, anzupassen sind lediglich die Definitionen der Endpunkte mit dem auszuführenden Programmcode.

Listing 5.1 zeigt annäherungsweise wie Endpunkte zum Erstellen und Abrufen von Optimierungsaufträgen in Flask definiert werden können.

[7] `https://docs.traefik.io`
[8] `https://hub.docker.com/_/traefik`
[9] `https://palletsprojects.com/p/flask/`

Code zum Überprüfen der Anwendereingaben und zum Behandeln von Fehlern wurde außen vor gelassen.

```
@app.route('/api/users/<user>/optimizations', methods=['GET'])
def list_optimizations(user):
    db = DatabaseClient()
    return db.list_optimizations(user), 200

@app.route('/api/users/<user>/optimizations', methods=['POST'])
def schedule_optimization(user):
    # add optimization request to database
    timestamp = '%sZ' % datetime.utcnow()\
        .replace(microsecond=0).isoformat()
    parameters = request.get_json()
    optimization = {
        'parameters': parameters,
        'created_at': timestamp,
        'status': 'queued'
    }
    db.add_optimization(user, optimization)

    # trigger dask workers to perform optimizations
    from distributed import Client, fire_and_forget
    client = Client()
    future = client.submit(fetch_and_perform_optimizations)
    fire_and_forget(future)
    client.close()

    return optimization, 201
```

Listing 5.1: Definieren von Web API-Endpunkten in Flask

5.3.5 Webanwendung mit Vue.js und Vuetify

Im Projekt *GameOpSys* ist die Entwicklung einer rudimentären Webanwendung vorgesehen, jedoch liegt dies außerhalb des Rahmens dieser Arbeit. Zum Nachweis der Machbarkeit (*Proof of Concept*) wird dennoch eine minimale Webanwendung erstellt.

Abbildung 5.2 zeigt die Begrüßungsansicht der Webanwendung, welche eine rudimentäre Benutzerauswahl erlaubt. Eine Anwendung im produktiven Betrieb würde an dieser Stelle Zugangsdaten abfragen.

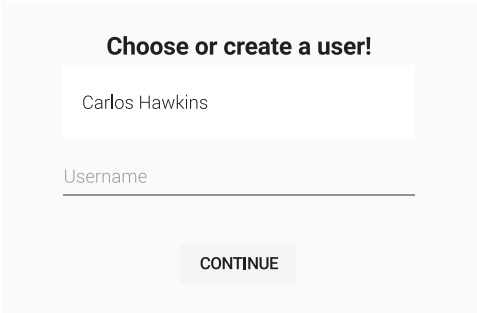
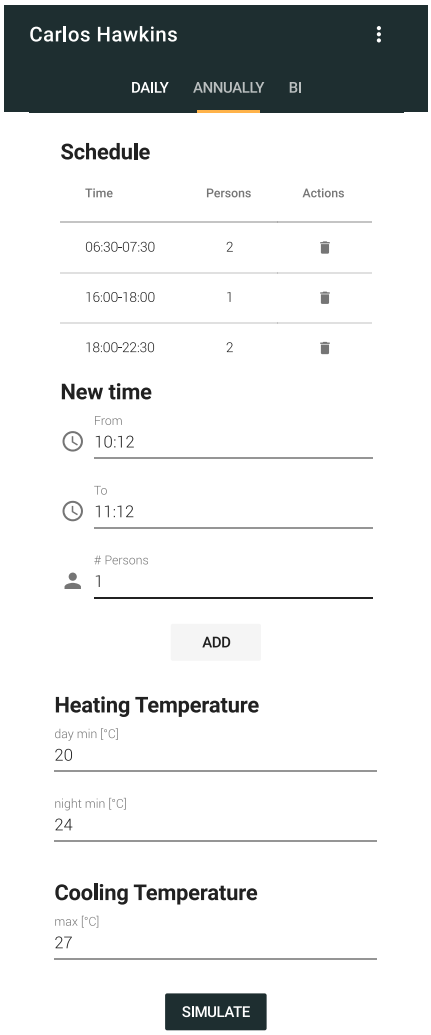


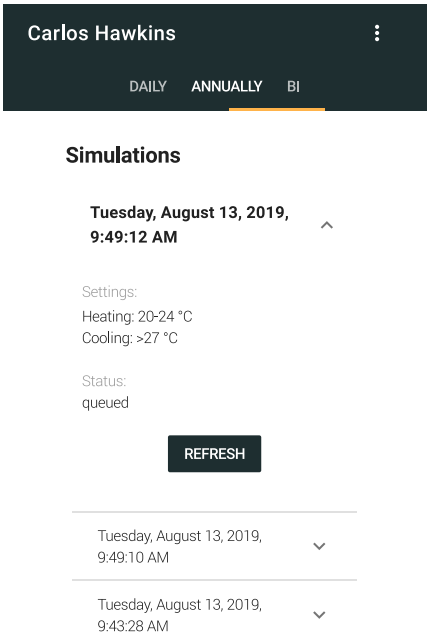
Abbildung 5.2: Screenshot der Anmeldungsseite der Webanwendung

In Abbildung 5.3 werden die Ansichten zum Starten und Einsehen von Optimierungsaufträgen dargestellt. Sie beinhalten die Eingabemöglichkeit von Anwesenheitszeiten der Bewohner sowie die gewünschten Mindest- und Maximalraumtemperaturen. Zur Abfrage von Ergebnissen kann eine Aktualisieren-Schaltfläche gedrückt werden.

Die Webanwendung wird, wie auch die Web API, von einer weiteren nginx-Instanz bereitgestellt. Der Lastverteiler Traefik entscheidet auf Basis der URL, ob eine Anfrage von einem *frontend*-Webserver oder einem *backend*- bzw. *api*-Webserver beantwortet werden muss. Für Anfragen über URLs, die mit `/api/` beginnen, ist zweiterer zuständig, alle anderen Anfragen werden an ersteren weitergeleitet.



(a) Eingabe der Anwesenheitszeiten und der gewünschten Raumtemperaturen



(b) Übersicht über die Optimierungsaufträge mit Aktualisierungsmöglichkeit

Abbildung 5.3: Screenshots der Ansichten zur Bedienung der Web API

5.3.6 MongoDB — Dokumentenorientierte NoSQL-Datenbank

Um eine persistente Datenspeicherung zu ermöglichen, wurde das Projekt um einen Container mit einer *MongoDB*-Instanz erweitert. MongoDB ist eine hoch skalierbare, dokumentenorientierte NoSQL-Datenbanklösung, die kein starres Datenbankschema benötigt. Die gespeicherten Daten werden untergliedert in *Collections*, welche *Tabellen* in relationalen Datenbanken entsprechen, sowie *Documents*, der Äquivalenz von *Zeilen* (*rows*) [56].

Neben der Skalierbarkeit von MongoDB war ein weiteres Kriterium ausschlaggebend für die Wahl einer dokumentenorientierten NoSQL-Datenspeicherlösung: die Daten können von der Webanwendung (in JavaScript), über die Web API und das Webapplikationsframework (in Python) bis hin zu ihrer Persistierung ohne Transformation durchgereicht werden. In allen Teilen dieser Kette werden Daten in Form von verschachtelten Datenkatalogen (*data dictionaries*) gehandhabt und können zum Austausch als *JavaScript Object Notation (JSON)*-Dokument dargestellt werden.

Listing 5.2 zeigt die Repräsentation der Daten eines beispielhaften Benutzers in JSON, welche zur Parameteroptimierung herangezogen werden.

```
{
    "name": "Carlos Hawkins",
    "preferences": {
        "temperatures": {
            "minPresent": 24, "minAbsent": 20, "maxAlways": 27
        },
        "availability": [{
                "persons": 2, "times": "06:30-07:30"
            }, {
                "persons": 1, "times": "16:00-18:00"
            }, {
                "persons": 2, "times" : "18:00-22:30"
            }
        ]
    }
}
```

Listing 5.2: Datenrepräsentation eines Benutzers

Für das Herstellen der Verbindung zur Datenbank wird die Programmbibliothek des Treibers *PyMongo*[10] verwendet. Listing 5.3 zeigt beispielhaft, wie die Interaktion mit der Datenbank funktioniert. Um die Datenbanklösung austauschbar zu halten, wird der Code in einer eigenen Klasse gekapselt.

```
1  from copy import deepcopy
2  import pymongo
3
4  DATABASE_URI = os.getenv("DATABASE_URI")
5  DATABASE_NAME = os.getenv("DATABASE_NAME")
6  USERS_COLLECTION='users'
7  OPTIMIZATIONS_COLLECTION='optimizations'
8
9  class Client:
10     database = None
11
12     def connect(self):
13         if not self.database:
14             connection = pymongo.MongoClient(DATABASE_URI)
15             self.database = connection[DATABASE_NAME]
16             self.database[USERS_COLLECTION] \
17                 .create_index([('name', pymongo.ASCENDING)], \
18                     unique=True)
19             self.database[OPTIMIZATIONS_COLLECTION] \
20                 .create_index([('user', pymongo.ASCENDING), \
21                     ('datetime', pymongo.DESCENDING)])
22
23     def list_users(self):
24         self.connect()
25         users = self.database[USERS_COLLECTION] \
26             .find({}, {'_id': False, 'name': True}) \
27             .sort('name', pymongo.ASCENDING)
28         return [user['name'] for user in users]
29
30     def add_user(self, user):
31         user = deepcopy(user)
32         self.connect()
33         self.database[USERS_COLLECTION].insert_one(user)
```

Listing 5.3: Klasse zum Abwickeln der Kommunikation mit MongoDB

[10] https://github.com/mongodb/mongo-python-driver

5.3.7 JModelica-Umgebung zum Erstellen von FMU

Als Basis für die JModelica-Umgebung dient das inoffizielle Docker-Image *mclab/jmodelica*, das entweder direkt von *Docker Hub* heruntergeladen[11] oder mithilfe des Git-Repositories[12] nachgebaut werden kann. Damit kann die benötigte JModelica-Umgebung in wenigen Schritten eingerichtet werden.

Zunächst werden die Pakete *Dask.distributed* und *PyMongo* installiert. Anschließend werden die Modelica-Bibliotheken *IDEAS* und *AixLib* heruntergeladen und extrahiert (siehe Listing 5.4).

```
1  FROM mclab/jmodelica
2
3  RUN apt-get update && \
4      apt-get install -y \
5      python3-pip \
6      curl && \
7      rm -rf /var/lib/apt/lists/*
8
9  RUN pip3 install dask distributed pymongo --upgrade
10
11 # download IDEAS library
12 RUN curl -L https://github.com/open-ideas/IDEAS/archive/
       v2.1.0.tar.gz | tar xzvf - -C /opt
13
14 # download AixLib library
15 RUN curl -L https://github.com/RWTH-EBC/AixLib/archive/
       v0.7.3.tar.gz | tar xzvf - -C /opt
16
17 COPY dask-worker.sh /root/work/
18
19 ENTRYPOINT ["./dask-worker.sh", "dask_scheduler:8786"]
```

Listing 5.4: Dockerfile zum Erstellen einer JModelica-/Dask-Umgebung

Gestartet wird die Umgebung durch Aufrufen des Dask Worker-Programms. Dies geschiet mit Hilfe eines Skripts, welches die zur Simulation benötigten Umgebungsvariablen setzt. Das Skript wurde von der JModelica-Installation entnommen und angepasst (siehe Listing 5.5).

[11] https://hub.docker.com/r/mclab/jmodelica
[12] https://bitbucket.org/mclab/jmodelica_dockerized

```
#!/bin/sh
set -ex

if test "${JAVA_HOME}" = ""; then
  export JAVA_HOME="$(java -XshowSettings:properties -version
     2>&1 | \
  sed '/^[[:space:]]*java\.home/!d;s/^[[:space:]]*java\.home[[:
     space:]]*=[[:space:]]*//')"
fi

JMODELICA_HOME=/opt/jmodelica \
IPOPT_HOME=/opt/ipopt \
SUNDIALS_HOME=${JMODELICA_HOME}/ThirdParty/Sundials \
PYTHONPATH=:${JMODELICA_HOME}/Python/::$PYTHONPATH \
LD_LIBRARY_PATH=:/opt/ipopt/lib/:${JMODELICA_HOME}/ThirdParty/
   Sundials/lib:${JMODELICA_HOME}/ThirdParty/CasADi/lib:
   $LD_LIBRARY_PATH \
SEPARATE_PROCESS_JVM=${JAVA_HOME} \
exec dask-worker --nthreads=1 --nprocs=$(nproc) "$@"
```

Listing 5.5: Skript zum Starten des Dask Workers

Eine Besonderheit der Simulationsumgebung ergibt sich dadurch, dass JModelica einen Python-Interpreter in der nicht länger entwickelten und unterstützten Version 2.x[13] benötigt, der Rest des Frameworks jedoch Python 3.x verwendet. Als Abhilfe werden im Container der Simulationsumgebung deshalb beide Versionen installiert. Das Kompilieren wird mit dem Python 2.x-Interpreter mittels Subprozess durchgeführt (siehe Listing 5.6).

```
#!/usr/bin/env python3
from subprocess import Popen, PIPE

cmd = 'python compile.py Model model.mo model.fmu'
popen = Popen(cmd.split(' '), stdout=PIPE, stderr=PIPE)
std_output, err_output = popen.communicate()

if popen.returncode != 0:
    # compilation failed, report content of variable err_output
else:
    # compilation succeeded, use model.fmu
```

Listing 5.6: Kompilieren eines Modelica-Modells mit Python 2

[13] https://pythonclock.org

5.4 Bereitstellung des Frameworks

Für die Bereitstellung des Frameworks wird Docker Swarm mode[14] genutzt. Ein Docker Swarm wird zunächst auf einem Rechner initialisiert. Dieser übernimmt die Rolle eines *Managers*.

```
$ docker swarm init
```

Dem Cluster können optional beliebig viele weitere Rechner als *Worker* beitreten. Die Ausgabe des *docker swarm init*-Befehls enthält einen Beitrittstoken und die IP-Adresse des Managers. Dieser hat die folgende Struktur und muss auf den Worker-Rechnern ausgeführt werden:

```
$ docker swarm join --token <token> <ip-address>:2377
```

Für die Kommunikation zwischen den Containern des Frameworks wird ein virtuelles Netzwerk erstellt. Dafür ist die Ausführung des folgenden Befehls erforderlich:

```
$ docker network create --driver=overlay traefik-public
```

Nun kann das Framework mit dem gewünschten Umfang erstellt werden. Die Projektvorlage enthält die benötigten Skripte zum Erstellen und Bereitstellen des Frameworks. Durch die entsprechende Angabe der Anzahl an Replikaten erstellt Docker Swarm mode je eine Instanz des JModelica-Containers auf jedem Manager- und Worker-Rechner.

```
JMODELICA_REPLICAS=‘docker node ls -q | wc -l‘ \
DOMAIN=gameopsys.com \
TRAEFIK_TAG=gameopsys.com \
STACK_NAME=gameopsys-com \
TAG=prod \
bash ./script-deploy.sh
```

Das Skript zum Starten eines Dask Workers (siehe Listing 5.5) sorgt dafür, dass auf jedem Rechner ein Prozess je Prozessorkern erstellt wird und damit eine optimale Rechenkapazität zur Durchführung von Parameteroptimierungen bereitsteht.

[14] https://docs.docker.com/engine/swarm

6 Evaluierung und Ergebnisse

Im ersten Teil dieses Kapitels werden die Ergebnisse der Leistungsvergleiche zwischen Parameteroptimierungen mit sequenzieller und paralleler Programmierung dargelegt. Der zweite Teil gibt Aufschluss über das Leistungsverhalten von verteilten Parameteroptimierungen, verglichen mit der Verwendung von Multi-Processing auf einem Einzelrechner. Im dritten und letzten Teil dieses Kapitels wird das entwickelte Framework auf die Erfüllung der Aufgabenstellung evaluiert.

Als Plattform für die Durchführung der Vergleichsmessungen dienten Instanzen des Web-Services *Amazon Elastic Compute Cloud (Amazon EC2)*[1] vom Typ *c5n.xlarge* und *c5n.2xlarge*[2] mit dem Linux-Image *Amazon Linux AMI 2018.03.0*[3]. Mit diesem Instanztyp stehen zwei bzw. vier physikalische Prozessorkerne mit *Hyper-Threading (HT)*, demzufolge vier bzw. acht logische Prozessorkerne *(vCPU)*, für die Berechnungen zur Verfügung.

Die Messungen wurden auf gemeinsam genutzter Hardware durchgeführt, weshalb sich leichte Schwankungen in den Ergebnissen abzeichnen. Die Korrelation zwischen den Größen der Aufgaben, den überschlagsmäßigen Ausführungszeiten und der Anzahl der verwendeten Rechenkerne ist dennoch gut erkennbar.

[1] `https://docs.aws.amazon.com/ec2/index.html`
[2] `https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking`
[3] `https://aws.amazon.com/de/amazon-linux-ami`

6.1 Sequenzielle vs. parallele Parameteroptimierung

Im Folgenden werden zunächst die Ausführungszeiten der Parameteroptimierung des Rastrigin-Modells, bezogen auf die Anzahl unabhängig voneinander durchgeführter Teiloptimierungen, dargestellt. Anschließend werden die Ergebnisse des mit dem Gebäudemodell wiederholten Versuchs wiedergegeben.

Zur Messung der Ausführungszeiten wurden die Programmaufrufe mit einem Skript automatisiert. Mit dem *Bash*-Befehl *time* wurde die tatsächlich vergangene Zeit vom Start bis zur Beendigung jedes Programmaufrufs protokolliert. Aufgrund der sehr kurzen Simulationszeit des Rastrigin-Modells wurden diese Messungen je drei Mal durchgeführt und der Durchschnittswert herangezogen. Für Zeitmessungen von einzelnen Ausführungsschritten innerhalb eines Programms wurde die Python-Funktion *time*[4] aus dem Modul *time* verwendet.

6.1.1 Parameteroptimierung des Rastrigin-Modells

In der sequenziellen Ausführung wird die Modelldefinition am Beginn zu einer FMU kompiliert. Dies nahm am Testsystem durchschnittlich 1,2 Sekunden in Anspruch. Das Laden der FMU in den Speicher benötigte etwa 11 Millisekunden. Ein Simulationslauf dieses einfachen Modells dauerte durchschnittlich eine halbe Millisekunde. Ein lokales Minimum konnte in den meisten Fällen innerhalb von 40 bis 60 Millisekunden gefunden werden, in Ausnahmefällen dauerte es annähernd 100 Millisekunden. Das kleinste gefundene Minimum wurde jeweils am Ende aller Durchläufe ermittelt.

Abbildung 6.1 veranschaulicht den annähernd linearen Anstieg der Ausführungszeiten der hunderten bis tausenden Teiloptimierungen bei sequenzieller Durchführung. Im Gegensatz dazu kann durch die Verwendung von Multi-Processing und der Auslastung mehrerer Prozessorkerne eine deutliche Reduktion der gesamten Ausführungszeit erzielt werden.

[4] https://docs.python.org/2/library/time.html#time.time

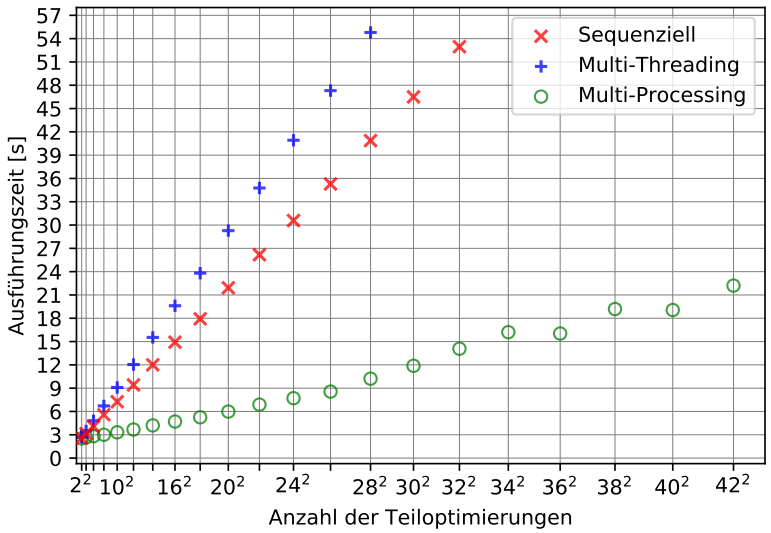


Abbildung 6.1: Ausführungszeiten der Rastrigin-Optimierung mit seqenzieller und paralleler Programmierung auf einer EC2-Instanz v. Typ *c5n.2xlarge* (8 vCPU)

Entgegen der Annahme, die Ausführungszeiten auch mit Multi-Threading reduzieren zu können, legen die Messergebnisse eine deutlich schlechtere Leistung gegenüber der sequenziellen Variante dar. Eine mögliche Erklärung dafür ist die gegenseitige Blockade der Threads durch das Akquirieren des *Global interpreter lock (GIL)*[5], der wechselseitigen Zugriffssperre für Threads auf Python-Objekte. Da auf eine tiefere Inspektion der Programmpakete von JModelica verzichtet wurde, lässt sich jedoch nicht ausschließen, dass dieses Verhalten noch weitere Ursachen hat.

6.1.2 Parameteroptimierung des Gebäudemodells

Im Vergleich zum Rastrigin-Modell ist das Gebäudemodell wesentlich komplexer. Das Kompilieren der Definition zu einer FMU beanspruchte bis zu 5 Sekunden. Geladen wurde das Modell durchschnittlich in ca. 30 Millisekunden. Ein Simulationsdurchlauf des Jahresenergieverbrauchs benötigte etwa 10 Sekunden. Für das Ermitteln der optimalen Dämmstärke waren

[5] `https://wiki.python.org/moin/GlobalInterpreterLock`

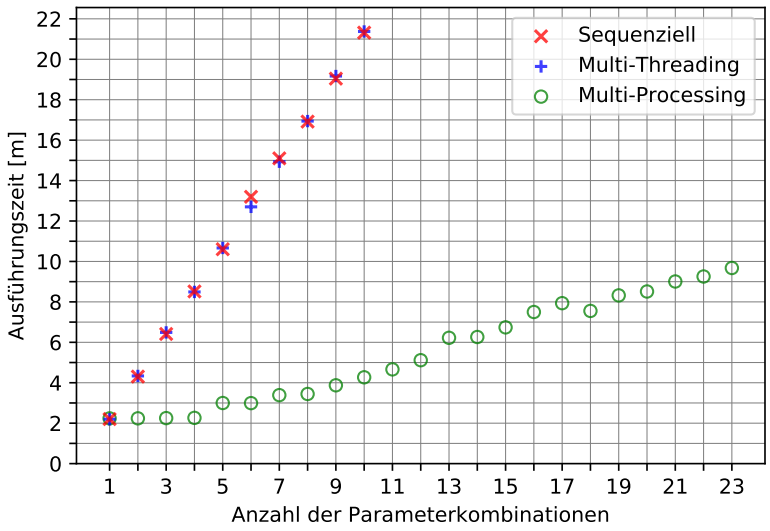


Abbildung 6.2: Ausführungszeiten der Energieoptimierung mit seqenzieller und paralleler Programmierung auf einer EC2-Instanz vom Typ *c5n.2xlarge* (8 vCPU)

jeweils zwischen 5 und 15 Jahressimulationen notwendig, wodurch sich eine gesamte Ausführungszeit je Parameterkombination von ungefähr 2,5 Minuten ergab.

Abbildung 6.2 veranschaulicht die Verläufe der Ausführungszeiten mehrerer Parameteroptimierungen in sequenzieller bzw. paralleler Durchführung. Es ist erkennbar, dass sich die Leistungsdaten der sequenziellen und der Multi-Threading-Variante nahezu decken. Eine Beschleunigung der Ausführung ist mit Multi-Threading demnach nicht möglich.

Mit Multi-Processing ist eine Beschleunigung sehr wohl möglich. Am Verlauf der Ausführungszeiten ist erkennbar, dass auf dieser EC2-Instanz mit vier physischen Recheneinheiten bis zu vier rechenintensive Simulationen überwiegend gleichzeitig durchgeführt werden können, ohne dass sich die gesamte Ausführungszeit signifikant ändert. Mit zunehmender Rechenlast verbessert sich die relative Zeitersparnis.

6.2 Verteilte Parameteroptimierung

In diesem Abschnitt werden die Ausführungszeiten der verteilten Parameteroptimierungen jenen mit Multi-Processing gegenübergestellt. Für die Durchführung des Versuchs wurden zunächst zwei Amazon EC2-Instanzen vom Typ *c5n.xlarge* mit je vier logischen Prozessorkernen über ein Netzwerk verbunden. Dieses Experiment vergleicht die Ausführungszeiten der Parameteroptimierungen mit acht logischen Prozessorkernen auf einem Rechner mit jenen von je vier logischen Prozessorkernen auf zwei Rechnern und veranschaulicht die Leistungseinbußen durch die Lastverteilung mittels Dask. Anschließend wurden drei Instanzen vom Typ *c5n.2xlarge* mit je acht logischen Prozessorkernen zu einem Rechnerverbund vereint. Damit wurde getestet, in welchem Ausmaß Parameteroptimierungen von einer horizontalen Skalierung der Rechenkapazität profitieren können.

Abbildung 6.3 verdeutlicht, dass die verteilte Optimierung von einfachen Modellen mittels Dask erst ab einer hohen Anzahl zu parallelisierender Teilprobleme Leistungssteigerungen bringt. Durch den vergleichsweise hohen Mehraufwand und Zeitverlust der Parallelisierung fällt die Beschleunigung jedoch gering aus.

Abbildung 6.4 stellt die Ergebnisse des gleichen Versuchs mit dem Gebäudemodell dar. Im Gegensatz zur Parameteroptimierung des einfachen Modells liegen die Ausführungszeiten der Optimierung des komplexeren Modells mit einmal acht und zweimal vier Recheneinheiten nahe aneinander. Auch beim Zusammenschluss von dreimal acht Recheneinheiten ist der Mehraufwand zum Verteilen der Last vernachlässigbar.

Die Schlussfolgerung aus den bisherigen Ergebnissen ist, dass eine Parallelisierung von Simulationen und Parameteroptimierungen von Modelica-Modellen grundsätzlich möglich ist. Eine Effizienzsteigerung ergibt sich jedoch erst ab einer bestimmten Komplexität der zu simulierenden Modelle. Das vereinfachte Gebäudemodell, und damit jegliches komplexere Modell, erfüllt diese Voraussetzung durchaus, weshalb die Sinnhaftigkeit der skalierbaren Architektur des Frameworks gegeben ist.

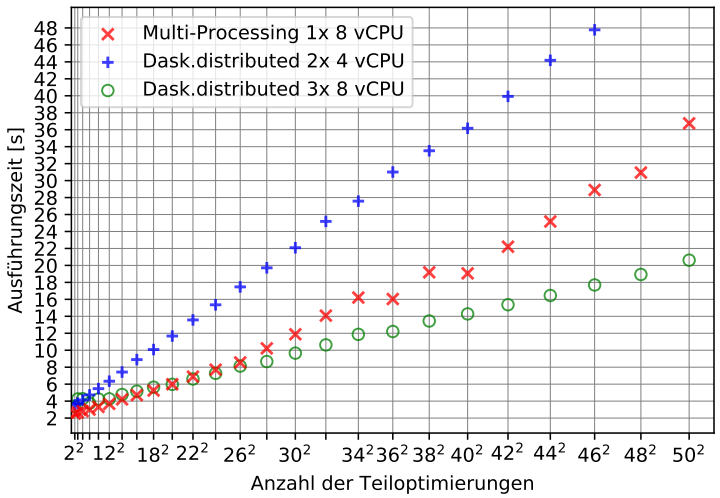


Abbildung 6.3: Ausführungszeiten der Rastrigin-Optimierung mit Multi-Processing und verteilter Ausführung

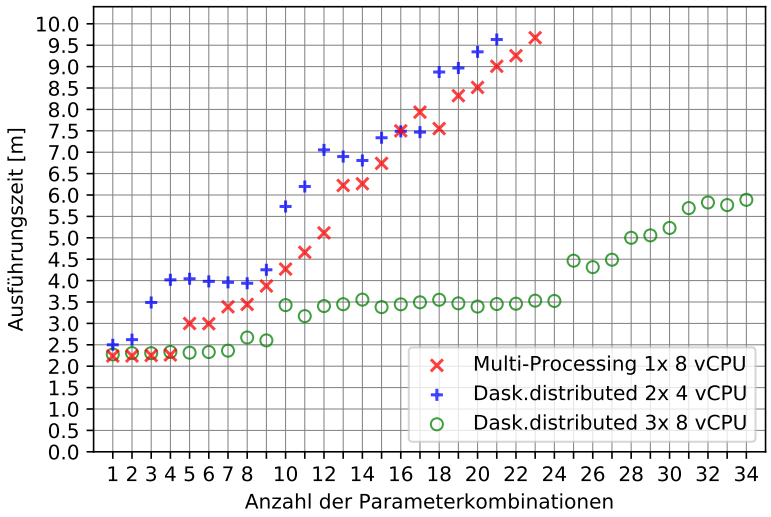


Abbildung 6.4: Ausführungszeiten der Energieoptimierung mit Multi-Processing und verteilter Ausführung

6.3 Evaluierung des Frameworks

In diesem Bereich wird das entwickelte Framework hinsichtlich verschiedener Gesichtspunkte evaluiert.

Korrektheit

Zur Validierung der korrekten Ergebnisermittlung werden Parameteroptimierungen des Rosenbrock- sowie des Rastrigin-Modells mittels API gestartet. Durch Senden eines *POST*-Requests an den Endpunkt `/api/rosenbrock` und der Angabe mehrerer Startparameter mittels JSON-Ausdruck wird das Framework zur Parameteroptimierung des Rosenbrock-Modells angestoßen.

```
$ curl -d '{"x0": [[-2, 4], [-1, 3], [-2, -2], [-5, 5]]}' \
    -H "Content-Type: application/json" \
    -X POST \
    http://localhost/api/rosenbrock
{
  "created_at": "2019-08-15T12:56:36Z",
  "parameters": {
    "x0": [[-2, 4], [-1, 3], [-2, -2], [-5, 5]]
  },
  "status": "queued"
}
```

Der Zeitstempel des Eingangs der Anfrage dient zur Identifizierung. Das Ergebnis lässt sich damit durch Senden eines *GET*-Requests abfragen.

```
$ curl http://localhost/api/rosenbrock/2019-08-15T12:56:36Z
{
  "created_at": "2019-08-15T12:56:36Z",
  "parameters": {
    "x0": [[-2, 4], [-1, 3], [-2, -2], [-5, 5]]
  },
  "result": {"f": 0.0, "x": [1.0, 1.0]},
  "status": "done"
}
```

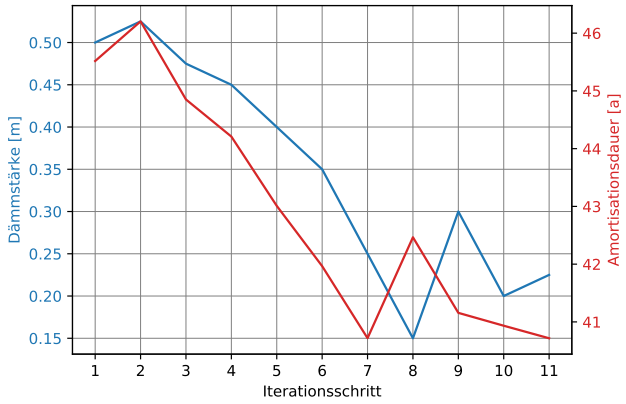


Abbildung 6.5: Iterationsschritte zur Ermittlung der optimalen Dämmstärke

Das berechnete Minimum ergibt gerundet wie zu erwarten *1,1*. Der gleiche Versuch, jedoch mit angepassten Parametern, wird mit dem Rastrigin-Modell durchgeführt. Das Ergebnis sieht wie folgt aus:

```
$ curl http://localhost/api/rastrigin/2019-08-15T12:57:10Z
{
  "created_at": "2019-08-15T12:57:10Z",
  "parameters": {
    "x0": [[-2, 4], [-1, 3], [-2, -2], [-5, 5]]
  },
  "result": {"f": 0.0, "x": [-0.0, 0.0]},
  "status": "done"
}
```

Auch hier stimmt das retournierte Ergebnis mit dem bekannten überein.

Zur Validierung der Berechnung der Dämmstärke werden die Zwischenergebnisse der Optimierung protokolliert und stichprobenartig manuell überprüft. Abbildung 6.5 stellt die Iterationsschritte des Minimierungsalgorithmus zur Ermittlung der optimalen Dämmstärke dar.

Flexibilität

Das Argument der Flexibilität des Frameworks gegenüber verschiedenen Modellen konnte bereits unter dem vorigen Punkt *Korrektheit* begutachtet werden. Es ist sowohl möglich, verschiedene Modelle mit generischem Code zu kompilieren und optimieren, als auch je Modell eine darauf abgestimmte Menge an Funktionen zur Durchführung dieser Aufgaben mitzuliefern.

Im Framework wurde dafür eine Konvention eingeführt:

- Die Funktion zum Kompilieren muss *compile_fmu* genannt werden und eine Modelldefinition unter dem Dateinamen *model.mo* laden.
- Die erstellte FMU muss als *model.fmu* abgelegt werden.
- Geladen werden muss das Modell mit Hilfe einer Funktion *load_fmu*.
- Der Name der Funktion, welche die Parameteroptimierung durchführt, muss *optimize* lauten. Ihr wird der Startwert, die Modell-Instanz sowie weitere Parameter als *Data dictionary* übergeben.

Solange diese Konvention eingehalten werden kann, ist das Framework ohne großen Aufwand auf die Unterstützung beliebiger Modelle erweiterbar.

Arten der Eingabeparameter

Während sich die Eingaben für das Rosenbrock- und das Rastrigin-Modell auf einen Vektor aus zwei Fließkommawerten beschränken, unterstützt das Framework zwei weitere Eingabearten, welche bei Gebäudemodellen zur Anwendung kommen. Zum einen kann eine Datei mit Wetterdaten beigefügt werden, welche im Zuge der Simulation geladen bzw. gelesen wird. Dies kann in weiterer Ausbaustufe mit geringem Aufwand insofern erweitert werden, dass durch die Angabe von Positionskoordinaten ein Abruf von ortsspezifischen Wetterdaten erfolgt.

Des Weiteren wird für die Heizregelung im Gebäudemodell die Angabe von An- und Abwesenheitszeiten von Personen unterstützt. Dies erfolgt durch die Erstellung einer MATLAB®[6]-Datei basierend auf den Parametern

[6] https://docs.scipy.org/doc/scipy/reference/io.html#matlab-files

aus der API, welche ebenfalls von der Simulationsumgebung geladen und verwendet wird.

Die Unterstützung dieser beiden zusätzlichen Eingabearten bildet einen wichtigen Baustein für die Verwendung des Frameworks für Optimierungen von realistischen Gebäudeenergiesystemen.

Bereitstellen von Frontend-Anwendungen

Durch die Verwendung der Vorlage für skalierbare Webanwendungen als Ausgangspunkt für den Aufbau des Frameworks ergibt sich der Vorteil, dass alle erforderlichen Komponenten für das Bereitstellen von grafischen, webbasierten Benutzerschnittstellen in JavaScript vorhanden sind. Mit nur minimalen Konfigurationsänderungen können zudem mehrere und auch aufwendigere Anwendungen bereitgestellt werden. Sollte der Webserver zum Engpass werden, kann mühelos eine weitere Instanz hinzugefügt werden. Der Load Balancer *Traefik* kommuniziert mit Docker, um stets über laufende Instanzen Bescheid zu wissen.

Lauffähigkeit auf beliebig leistungsstarker Hardware

Die einzelnen Komponenten dieses Frameworks sind bewusst klein und abgegrenzt gehalten. Für den Anwendungsfall der Parameteroptimierung des Gebäudemodells übersteigt der Rechenaufwand der Modellsimulationen dem des Webservers oder der Datenbank um ein Vielfaches. Sollte das Framework für andere Arten von Modellen zum Einsatz kommen, kann diese Verteilung ein sehr unterschiedliches Verhältnis aufweisen.

Durch die Nutzung von Anwendungscontainern mit je einer Teilaufgabe erhält das Framework ein hohes Maß an Flexibilität. Je nach Auslastung kann jeder Container beliebig oft repliziert werden und so gezielt für eine optimale Ausnutzung der Hardwareressourcen sorgen.

Ein weiterer positiver Punkt der Nutzung von Containervirtualisierung ist die Möglichkeit, das Framework unverändert sowohl auf einem Ein-

zelplatzrechner in Betrieb zu nehmen, als auch auf einem Cluster von leistungsstarken Rechnern.

7 Fazit und Ausblick

Den Ausgangspunkt für diese Arbeit legt die Notwendigkeit, den CO_2-Ausstoß trotz des fortwährend steigenden Energieverbrauchs zu reduzieren. Ein Ansatz zur Herangehensweise an diese Herausforderung ist das Erkennen und Nutzbarmachen des Energieeinsparungspotenzials aufseiten der EndverbraucherInnen. Die aktive Partizipation der KonsumentInnen ist unumgänglich, weshalb Bestrebungen dahingehend verfolgt werden, Bewusstsein zu wecken, Möglichkeiten zu bieten und Anreize zu schaffen, ihr Verhalten in Richtung eines schonenderen Umgangs mit Ressourcen anzupassen.

Das größte Energieeinsparungspotenzial in Haushalten findet sich im angemessenen Betrieb von Heizungs-, Lüftungs- und Klimatechniksystemen. Um einen optimalen Betrieb zu ermöglichen und die Energieeffizienz zu verbessern, wird im professionellen Umfeld Computersimulation genutzt. Aufseiten der Konsumenten findet diese Technologie jedoch kaum Verwendung.

Im Rahmen eines Forschungsprojekts wird evaluiert, wie eine mobile Anwendung unter Einbeziehung von Computersimulation die EndnutzerInnen dabei unterstützen kann, den eigenen Energieverbrauch zu senken. Aus diesem Anwendungskonzept ergibt sich die Erfordernis, eine *Back-End*-Umgebung zur Ausführung von Computersimulationen bereitzustellen. Das Ziel dieser Arbeit liegt in deren konzeptionellen Entwicklung und prototpyischen Umsetzung sowie deren Validierung anhand vereinfachter Anwendungsfälle.

Das Ergebnis der Arbeit zeigt, dass alle erforderlichen Komponenten verfügbar sind und sich zu einer funktionierenden Gesamtlösung kombinieren lassen. Ihre Anwendbarkeit und Erweiterbarkeit wird durch die Argumente belegt, dass die Plattform vollautomatisiert ausgeführt werden kann, korrekte

Resultate berechnet, verschiedene Arten von Modellen und Parametern unterstützt, sich auf Hardware unterschiedlicher Leistung betreiben lässt, skalierbar ist und von beliebig vielen AnwenderInnen gleichzeitig genutzt werden kann.

Im Zuge der Ausarbeitung wurden Bereiche identifiziert, die Anknüpfungspunkte für zukünftige Arbeiten darstellen. Es wurde demonstriert, dass die Laufzeit durch gleichzeitiges Ausführen mehrerer Simulationen bzw. Optimierungen erheblich reduziert werden kann, jedoch wurde stets lediglich der frei verfügbare Solver von JModelica verwendet. Eine mögliche Weiterentwicklung des Frameworks könnte darin bestehen, andere freie oder kommerzielle Simulationsumgebungen einzugliedern und zu untersuchen, ob dadurch Effizienzsteigerungen erzielt werden können. Auch könnte eine Brücke zu Studien hergestellt werden, welche untersucht haben, wie Simulationscode von Modelica nachträglich parallelisiert werden kann (siehe [45]).

Ein weiterer Anknüpfungspunkt betrifft die Flexibilität des Frameworks gegenüber nicht vordefinierter Modelle und Optimierungsalgorithmen. Ein Ausbau der Plattform zu einem Dienst ist naheliegend, der jegliche Art von Modellen entgegennimmt und eine verteilte Ausführung von Parameteroptimierungen erlaubt. Neben der Konzeption einer geeigneten Schnittstelle müssten vor allem Sicherheitsaspekte berücksichtigt werden, die außerhalb des Rahmens dieser Arbeit lagen. Beispielsweise wäre es zu untersuchen, wie die Plattform das Ausführen von schadhaftem Code verhindern oder zumindest die Reichweite dessen eindämmen könnte.

Appendix

Quellenangaben

[1] Kaile Zhou und Shanlin Yang. »Understanding household energy consumption behavior: The contribution of energy big data analytics«. In: *Renewable and Sustainable Energy Reviews* 56 (2016), S. 810–819. ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2015.12.001. URL: http://www.sciencedirect.com/science/article/pii/S1364032115013817 (siehe S. 1).

[2] International Energy Agency. *Global Energy Economics and Climate Protection Report 2009*. Techn. Ber. International Energy Agency, 2010 (siehe S. 1).

[3] Gerald Schweiger u. a. »District energy systems: Modelling paradigms and general-purpose tools«. In: *Energy* 164 (2018), S. 1326–1340. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2018.08.193. URL: http://www.sciencedirect.com/science/article/pii/S0360544218317274 (siehe S. 1, 14).

[4] Diercks Joachim und Kristof Kupka. »Recrutainment – Bedeutung, Einflussfaktoren und Begriffsbestimmung«. In: *Recrutainment: Spielerische Ansätze in Personalmarketing und -auswahl*. Hrsg. von Joachim Diercks und Kristof Kupka. Wiesbaden: Springer Fachmedien Wiesbaden, 2013, S. 1–18. ISBN: 978-3-658-01570-1. DOI: 10.1007/978-3-658-01570-1_1. URL: https://doi.org/10.1007/978-3-658-01570-1_1 (siehe S. 2).

[5] Luca Morganti u. a. »Gaming for Earth: Serious games and gamification to engage consumers in pro-environmental behaviours for energy efficiency«. In: *Energy Research & Social Science* 29 (2017), S. 95–102. ISSN: 2214-6296. DOI: https://doi.org/10.1016/j.erss.2017.05.001. URL: http://www.sciencedirect.com/science/article/pii/S2214629617301093 (siehe S. 2).

[6] A.S. Tanenbaum und M. Van Steen. *Distributed Systems: Principles and Paradigms*. Maarten van Steen, 2017. ISBN: 9789081540629 (siehe S. 3, 45).

[7] Eric Winsberg. »Computer Simulations in Science«. In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019 (siehe S. 5).

[8] H. Bossel. *Modellbildung und Simulation: Konzepte, Verfahren und Modelle zum Verhalten dynamischer Systeme*. Vieweg+Teubner Verlag, 2013. ISBN: 9783322836588 (siehe S. 5–7, 9–12).

[9] D.W. Heermann. *Computer Simulation Methods in Theoretical Physics*. Springer Berlin Heidelberg, 2012. ISBN: 9783642969713 (siehe S. 5, 7).

[10] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley, 2015. ISBN: 9781118859162 (siehe S. 6–8, 11, 16–18, 20, 22–24).

[11] Abdul Afram und Farrokh Janabi-Sharifi. »Black-box modeling of residential HVAC system and comparison of gray-box and black-box modeling methods«. In: *Energy and Buildings* 94 (2015), S. 121–149. ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2015.02.045. URL: http://www.sciencedirect.com/science/article/pii/S0378778815001504 (siehe S. 12).

[12] Peter Rockett und Elizabeth Abigail Hathway. »Model-predictive control for non-domestic buildings: a critical review and prospects«. In: *Building Research & Information* 45.5 (2017), S. 556–571. DOI: 10.1080/09613218.2016.1139885. eprint: https://doi.org/10.1080/09613218.2016.1139885. URL: https://doi.org/10.1080/09613218.2016.1139885 (siehe S. 13).

[13] Gerald Schweiger. »Framework for dynamic simulation and dynamic optimization of district energy systems«. Diss. Universität Innsbruck, Dissertation, 2019, 2019 (siehe S. 13, 15).

[14] B. Zupancic u. a. »Continuous Systems Modelling Education – Causal or Acausal Approach?« In: *ITI 2008 - 30th International Conference on Information Technology Interfaces*. Juni 2008, S. 803–808. DOI: 10.1109/ITI.2008.4588514 (siehe S. 14).

[15] J Kofránek u. a. »Causal or acausal modeling: labour for humans or labour for machines«. In: *Technical computing prague* (2008), S. 1–16 (siehe S. 14).

[16] Hilding Elmqvist und Sven Erik Mattsson. »An introduction to the physical modeling language modelica«. In: *Proceedings of the 9th European simulation symposium, ESS*. Bd. 97. Citeseer. 1997, S. 19–23 (siehe S. 15, 16, 18).

[17] Peter Fritzson und Vadim Engelson. »Modelica — A unified object-oriented language for system modeling and simulation«. In: *ECOOP'98 — Object-Oriented Programming*. Hrsg. von Eric Jul. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, S. 67–90. ISBN: 978-3-540-69064-1 (siehe S. 15, 17, 18).

[18] Modelica Association. *Modelica Association*. URL: https://modelica.org/association (besucht am 23. 07. 2019) (siehe S. 16).

[19] Modelica Association. *Modelica Language Documents - Version 3.4 - April 2017*. URL: https://modelica.org/documents (besucht am 23. 07. 2019) (siehe S. 16).

[20] Modelica Association. *Modelica® - A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.4*. Apr. 2017. URL: https://www.modelica.org/documents/ModelicaSpec34.pdf (besucht am 23. 07. 2019) (siehe S. 16, 17).

[21] C. Kral. *Modelica - Objektorientierte Modellbildung von Drehfeldmaschinen: Theorie und Praxis für Elektrotechniker mit Tutorial für GitHub*. Carl Hanser Verlag GmbH & Company KG, 2018. ISBN: 9783446457331 (siehe S. 16, 17, 20, 21).

[22] Peter Fritzson. *The Modelica Language and Technology for Model-Based Development - Overview Talk*. März 2012. URL: https://openmodelica.org/images/docs/Modelica-and-OpenModelica-overview-Peter-Fritzson-120328.pdf (besucht am 23. 07. 2019) (siehe S. 17).

[23] Modelica Association. *Modelica Language*. URL: https://modelica.org/modelicalanguage (besucht am 23. 07. 2019) (siehe S. 18).

[24] Modelica Association. *Overview of Modelica Libraries*. URL: https://modelica.org/ModelicaLibrariesOverview (besucht am 23. 07. 2019) (siehe S. 18).

[25] Martin Sjölund, Peter Fritzson und Adrian Pop. »Bootstrapping a Modelica Compiler aiming at Modelica 4«. In: *: Linköping Electronic Conference Proceedings*. Linköping University Electronic Press, 2011, S. 510–521. ISBN: 978-91-7393-096-3 (siehe S. 19).

[26] Modelica Association. *Modelica Tools*. URL: https://modelica.org/tools (besucht am 23. 07. 2019) (siehe S. 20, 21).

[27] K. Nishimiya, T. Saito und S. Shimada. »Evaluation of Functional Mock-up Interface for vehicle power network modeling«. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. Juni 2015, S. 1–6. DOI: 10.1145/2744769.2747926 (siehe S. 23).

[28] Wuzhu Chen, Michaela Huhn und Peter Fritzson. »A generic FMU interface for modelica«. In: *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools; Zurich; Switzerland; September 5; 2011*. 056. Linköping University Electronic Press. 2011, S. 19–24 (siehe S. 24).

[29] International Energy Agency. *Energy Efficiency: Buildings*. URL: https://www.iea.org/topics/energyefficiency/buildings/ (besucht am 23. 07. 2019) (siehe S. 24).

[30] V.S.K.V. Harish und Arun Kumar. »A review on modeling and simulation of building energy systems«. In: *Renewable and Sustainable Energy Reviews* 56 (2016), S. 1272–1292. ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2015.12.040. URL: http://www.sciencedirect.com/science/article/pii/S1364032115014239 (siehe S. 24–26).

[31] Anh-Tuan Nguyen, Sigrid Reiter und Philippe Rigo. »A review on simulation-based optimization methods applied to building performance analysis«. In: *Applied Energy* 113 (2014), S. 1043–1058. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2013.08.061. URL: http://www.sciencedirect.com/science/article/pii/S0306261913007058 (siehe S. 24, 27, 28).

[32] JA Clarke. *Energy Simulation in Building Design (Second Edition)*. Hrsg. von JA Clarke. Second Edition. Oxford: Butterworth-Heinemann, 2001. ISBN: 978-0-7506-5082-3 (siehe S. 25).

[33] Gerald Schweiger und Michael Wetter. »Dynamische Optimierung von Modelica-basierten Modellen«. In: Feb. 2018. ISBN: 978-3-85125-586-7. DOI: 10.3217/978-3-85125-586-7 (siehe S. 26, 29).

[34] Cladera Bohigas u. a. »Simulation of Thermal Building Behaviour in Modelica«. In: März 2002 (siehe S. 26).

[35] Michael Wetter, Christoph van Treeck und Jan Hensen. »New generation computational tools for building and community energy systems«. In: *Energy in Buildings and Communities Programme. IEA EBC Annex* 60 (2013) (siehe S. 26, 27).

[36] et al. E. Widl. »Eine neue Generation von Simulationswerkzeugen für Gebäude und kommunale Energiesysteme, basierend auf den Standards von Modelica und Functional Mockup Interface«. In: *Energy in Buildings and Communities Programme. IEA EBC Annex* 60 (2018) (siehe S. 27).

[37] Dirk Müller u. a. »AixLib - An Open-Source Modelica Library within the IEA-EBC Annex60 Framework«. In: *Proceedings of the CESBP Central European Symposium on Building Physics and BauSIM 2016 : September 14-16, 2016, Dresden, Germany / editor: Technische Universität Dresden, J. Grunewald ; international scientific committee (CESBP): John Grunewald (Dresden, Germany, chair) [und 21 weitere] ; scientific committe (BauSIM): John Grunewald (Dresden, Germany), Clemens Felsmann (Dresden, Germany) [und 18 weitere]*. Weitere Konferenz: BauSIM 2016, 2016-09-14 - 2016-09-16, Dresden, Germany. Central European Symposium on Building Physics, Dresden (Germany), 14 Sep 2016 - 16 Sep 2016. Stuttgart: Fraunhofer IRB Verlag, 14. Sep. 2016, S. 3–9. URL: http://publications.rwth-aachen.de/record/681852 (siehe S. 27).

[38] International Building Performance Simulaton Association. *IBPSA Project 1*. URL: https://ibpsa.github.io/project1/ (besucht am 23. 07. 2019) (siehe S. 27).

[39] Sudhanshu K Mishra. »Some new test functions for global optimization and performance of repulsive particle swarm method«. In: *Available at SSRN 926132* (2006) (siehe S. 30).

[40] H. H. Rosenbrock. »An Automatic Method for Finding the Greatest or Least Value of a Function«. In: *The Computer Journal* 3.3 (Jan. 1960), S. 175–184. ISSN: 0010-4620. DOI: 10.1093/comjnl/3.3.175. eprint: http://oup.prod.sis.lan/comjnl/article-pdf/3/3/175/988633/030175.pdf. URL: https://doi.org/10.1093/comjnl/3.3.175 (siehe S. 30).

[41] Majid Jaberipour und Esmaile Khorram. »Two improved harmony search algorithms for solving engineering optimization problems«. In: *Communications in Nonlinear Science and Numerical Simulation* 15.11 (2010), S. 3316–3331. ISSN: 1007-5704. DOI: https://doi.org/10.1016/j.cnsns.2010.01.009. URL: http://www.sciencedirect.com/science/article/pii/S1007570410000390 (siehe S. 30).

[42] K. Deb u. a. *Genetic and Evolutionary Computation — GECCO 2004: Genetic and Evolutionary Computation Conference Seattle, WA, USA, June 26–30, 2004, Proceedings*. Lecture Notes in Computer Science Teil 1. Springer Berlin Heidelberg, 2004. ISBN: 9783540223443 (siehe S. 32).

[43] R. Buyya, C. Vecchiola und S.T. Selvi. *Mastering Cloud Computing*. McGraw Hill, 2013. ISBN: 9781259029950. URL: https://books.google.at/books?id=VSDZAgAAQBAJ (siehe S. 36).

[44] Martin Sjölund u. a. »Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling«. In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools; Oslo; Norway; October 3*. 47. Linköping University Electronic Press; Linköpings universitet, 2010, S. 71–80 (siehe S. 36).

[45] Peter Aronsson und Peter Fritzson. »Multiprocessor scheduling of simulation code from modelica models«. In: *Proceedings Modelica*. 2002 (siehe S. 36, 79).

[46] Vadim Engelson und Peter Fritzson. »Distributed Simulation Environment for Heterogeneous Computer Clusters«. In: *Scandinavian Simulation Conference (SIMS), September 2002, Oulu, Finland*. Scandinavian Simulation Society. 2002 (siehe S. 37).

[47] Johan Åkesson, Magnus Gäfvert und Hubertus Tummescheit. »Jmodelica—an open source platform for optimization of modelica models«. In: *Proceedings of MATHMOD*. 2009 (siehe S. 37).

[48] Ami Marowka. »On parallel software engineering education using python«. In: *Education and Information Technologies* 23.1 (Jan. 2018), S. 357–372. ISSN: 1573-7608. DOI: 10.1007/s10639-017-9607-0. URL: https://doi.org/10.1007/s10639-017-9607-0 (siehe S. 39, 43).

[49] Philipp Moritz u. a. »Ray: A Distributed Framework for Emerging AI Applications«. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, S. 561–577. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/moritz (siehe S. 46).

[50] Matthew Rocklin. »Dask: Parallel Computation with Blocked algorithms and Task Scheduling«. In: Jan. 2015, S. 126–132. DOI: 10.25080/Majora-7b98e3ed-013 (siehe S. 47).

[51] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org (siehe S. 47).

[52] N. Nguyen und D. Bein. »Distributed MPI cluster with Docker Swarm mode«. In: *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*. Jan. 2017, S. 1–7. DOI: 10.1109/CCWC.2017.7868429 (siehe S. 52–54).

[53] Dask Development Team. *Dask: Library for dynamic task scheduling, Distributed Scheduling*. 2016. URL: https://docs.dask.org/en/latest/distributed.html (besucht am 25. 07. 2019) (siehe S. 55).

[54] T. C. Chieu u. a. »Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment«. In: *2009 IEEE International Conference on e-Business Engineering*. Okt. 2009, S. 281–286. DOI: 10.1109/ICEBE.2009.45 (siehe S. 58).

[55] W. Hyun, M. Y. Huh und J. Park. »Implementation of docker-based smart greenhouse data analysis platform«. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*.

Okt. 2018, S. 1103–1106. DOI: 10.1109/ICTC.2018.8539551 (siehe S. 58).

[56] Zachary Parker, Scott Poe und Susan V. Vrbsky. »Comparing NoSQL MongoDB to an SQL DB«. In: *Proceedings of the 51st ACM Southeast Conference*. ACMSE '13. Savannah, Georgia: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-1901-0. DOI: 10.1145/2498328.2500047. URL: http://doi.acm.org/10.1145/2498328.2500047 (siehe S. 62).