FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Automatisierte Anbindung von Simulations- und Optimierungssoftware zur parallelen Lösung inverser Problemklassen

Dissertation
zur Erlangung des akademischen Grades
„Doktor-Ingenieur“ (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena

von Dipl.-Inf. Ralf Seidler

geboren am 27.11.1984 **in** Jena

Gutachter

1. Prof. Dr.-Ing. Hanns Martin Bücker

2. Prof. Dr. rer. nat. Michael Herty

Tag der Verteidigung: 9. November 2020

Zusammenfassung

In dieser Arbeit wird die in Python geschriebene Software “Environment for Combining Optimization and Simulation Software” (EFCOSS) zur Verbindung von Optimierungsalgorithmen mit Simulationsroutinen beschrieben, welche für verschiedene Problemstellungen aus der Geothermie und Materialwissenschaft angewendet wird. Zur Lösung werden inverse Probleme für Parameterbestimmung, Space-Mapping, Optimal Experimental Design (OED) und Modellidentifikation aufgestellt und diese mit Hilfe von EFCOSS gelöst. Dies wird nur möglich, da die interne Struktur von EFCOSS grundlegend umgearbeitet und dabei eine neue Softwarearchitektur unter Verwendung von Standard Python Paketen geschaffen wurde. Die Software ist gezielt darauf ausgelegt, mehrere Optimierungsprobleme, Zielfunktionen und Simulationsroutinen auszuwerten und damit ein breites Anwendungsspektrum effektiv zu lösen. Beispielhaft werden Simulationsmodelle für die geologischen Gegebenheiten in der Region um Perth in Australien, sowie in der Toskana in Italien, betrachtet, um neue Explorationsbohrlochpositionen mit Hilfe von OED zu finden, die niedrige Unsicherheiten in das Modell einbringen. Darüber hinaus werden Parameterschätzprobleme mit grob und fein aufgelösten Modellen der Regionen mit Hilfe des Space-Mapping Algorithmus optimiert und hierbei eine hohe Genauigkeit erzielt. Mit Hilfe der Modellidentifikation werden Modelle der Metallplastizität miteinander verglichen und Aussagen über deren Güte getroffen. Durch den automatisierten Einsatz von automatischem Differenzieren, sowohl im Vorwärts-, wie auch im Rückwärtsmodus, Parallelisierung und Wiederverwendung von bereits berechneten Ergebnissen, werden die seriellen Ausführungszeiten zur Lösung der gegebenen Probleme von mehreren Tagen beziehungsweise Wochen auf wenige Minuten gesenkt.

Abstract

This work introduces a novel extension of the Python software “Environment for Combining Optimization and Simulation Software” (EFCOSS). The extension addresses the solution of optimization problems of different types. Various problem instances that demonstrate the feasibility of this approach in new practical application scenarios include geothermal engineering and material science. In a more general context, EFCOSS enables to investigate the questions of which parameter values best fit a given computer model to measurements from real-world experiments, how should such experiments be designed with minimal uncertainty, which computer models should be used for a specific task, and how can the efficiency of such investigations be improved by using simpler models. These questions are addressed by employing techniques from parameter estimation, space mapping, optimal experimental design, and model identification that are implemented and brought together in EFCOSS. To this end, the internal structure of EFCOSS had to be redesigned completely to introduce a new software architecture based on standard Python packages. This new architecture allows for multiple optimization problems, objective functions, and simulation routines to be used within a single application. Simulation models of geothermal reservoirs in the regions of Perth in Australia and Tuscany in Italy are used as illustrating examples of optimal experimental design to find the location of new borehole sites that introduce low uncertainty in the parameter estimation. Furthermore, new parameter estimation problems are solved using space-mapping algorithms. Model identification is applied to metal-plasticity models to investigate different kinds of models. By combining automatic differentiation, parallelization, and reuse of intermediate results, the serial runtimes of the described problems are reduced from several weeks to minutes.

Danksagung

An dieser Stelle möchte ich allen danken, die mir in der langen Zeit der Bearbeitung dieser Arbeit zur Seite standen und mit konstruktiver Kritik und Anregungen zum Erfolg beigetragen haben. Ein besonderer Dank geht dabei an meinen Betreuer Prof. Martin Bücker für die lange ausdauernde Unterstützung, die vielen neuen Ideen und die Freiheiten, die ich als Mitarbeiter hatte. Weiterhin gilt mein Dank dem Team des Lehrstuhls Advanced Computing, Ali Rostami, David Neuhäuser, Adrian Knoth, Torsten Bosse, Markus Mieth, Frank Taubert, Daniel Walther und Johannes Schoder, die mich in meiner Zeit immer wieder aufgebaut und unterstützt haben. Die gemeinsamen Mittagessen und konstruktiven Gespräche mit vielen neuen Ideen waren großartig.

Mein weiterer Dank geht an die Kollegen der RWTH Aachen, mit denen ich gemeinsam am MeProRisk II Projekt gearbeitet habe: Kateryna Graf und mein Zweitgutachter Prof. Michael Herty vom Institut für angewandte Mathematik (IGPM), Johanna Bruckmann, Johannes Keller, Henrik Büsing, Jan Niederau, Anozie Ebigbo, Gabriele Marquardt und Prof. Christoph Clauser vom E.on Energy Reserach Center des Institutes für angewandte Geophysik und geothermische Energie. Außerdem möchte ich Andreas Wolf, dem Gruppenleiter Hochleitungsrechnen am Institut für Scientific Computing der TU Darmstadt, für die Informationen und Hinweise über den Shemat-Code danken. Die Forschung wurde teilweise finanziert durch die Verbundprojekte MeProRisk I und II der Bundesministerien für Bildung und Forschung (BMBF), Umwelt und Reaktorsicherheit (BMU) und Wirschaft (BMWi), Kennzeichen 0325389 A und F. Die Arbeiten wurden außerdem gfördert durch die Deutsche Forschungsgemeinschaft – INST 275/334–1 FUGG und INST 275/363–1 FUGG sowie durch den Freistaat Thüringen unter Gewährung eines Zuschusses aus Mitteln des Europäischen Fonds für regionale Entwicklung (EFRE-OP 2014-2020), Vorhabens-Nummer 2017 FGI 0031.

Abschließend gilt mein größter Dank meiner sehr verständnisvollen Frau Marion und meinen Kindern Johann und Leonore. Ohne euch hätte ich diese anstrengende und nervenaufreibende Zeit sicher nicht überstanden. Ihr seid die Besten, ich liebe euch!

Inhaltsverzeichnis

Kapitel 1

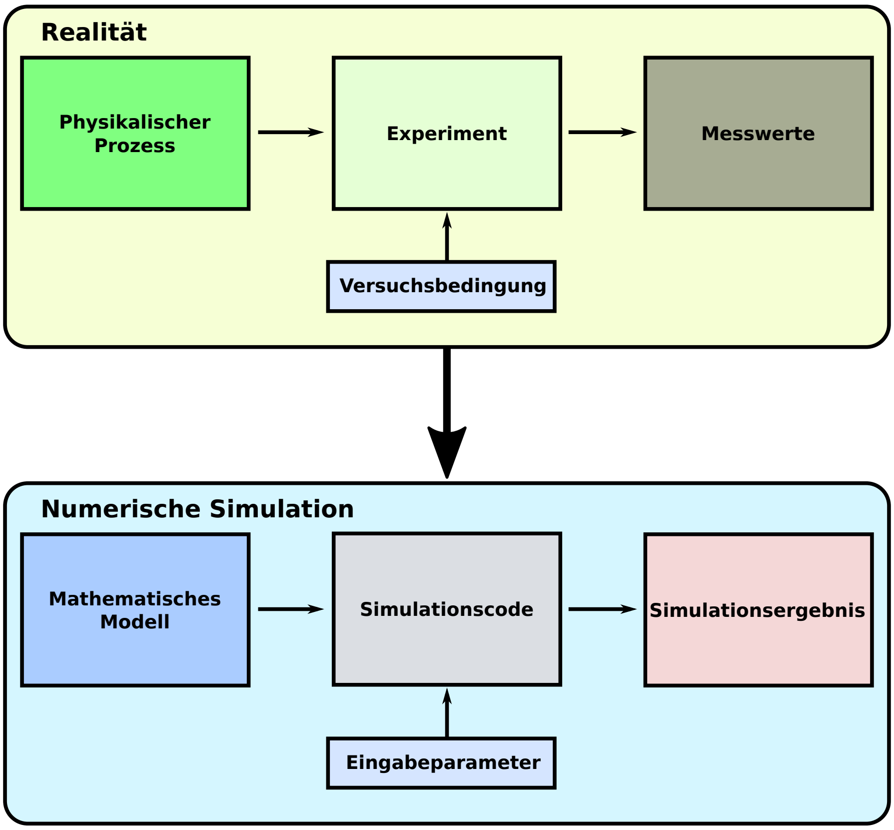
Einleitung

1.1 Ein Überblick

Industrie, Wissenschaft und Gesellschaft versuchen vermehrt die Probleme der realen Welt mittels Computersimulationen abzubilden. In der klassischen Physik werden Experimente definiert, welche unter bestimmten Versuchsbedingungen als Ergebnis Messwerte produzieren. Dies ist in Abb. 1.1 oben zu sehen. Ausgehend von der realen Welt kann eine numerische Simulation erstellt werden. Diese Simulationen helfen, komplexe Phänomene besser zu verstehen, indem an ihnen die untersuchten Geschehnisse analysiert und vorhergesagt werden können. Damit werden die sonst notwendigen zeitaufwändigen physikalischen Experimente auf ein Mindestmaß reduziert und der Fokus der Wissenschaftler kann auf der Auswertung der durch die Ergebnisse der Simulation gewonnenen Erkenntnisse liegen.

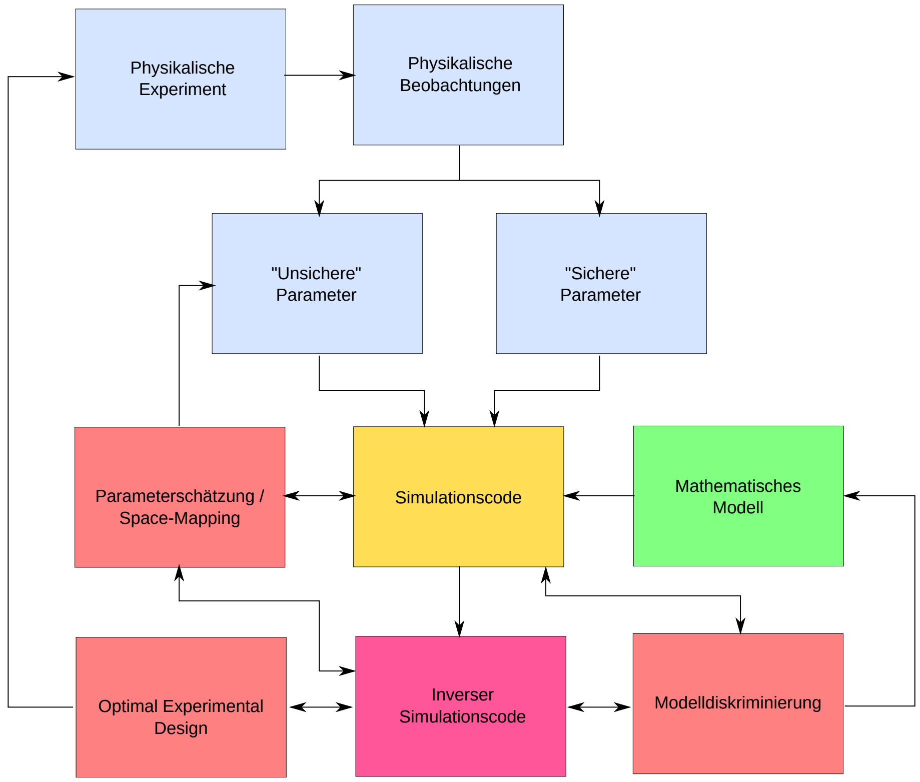
Um eine numerische Simulation zu erstellen, benötigt man eine mathematische Beschreibung der zu simulierenden Begebenheit, das mathematische Modell. Dieses Modell wird durch ein Computerprogramm berechnet, welches aufgrund von Eingabeparametern Simulationsergebnisse generiert. Um die Ergebnisse in annehmbarer Zeit zu erhalten, sollte bei der Implementierung auf parallele Algorithmen gesetzt werden. Hier bieten sich Techniken für gemeinsamen und verteilten Speicher an. Mehrkernprozessoren werden zum Beispiel mit OpenMP für gemeinsamen Speicher programmiert. Große Rechenanlagen mit mehreren Rechnern, sogenannte Cluster, müssen durch eine Netzwerkstruktur miteinander kommunizieren und nutzen folglich verteilten Speicher. Zur Programmierung bietet sich das Message Passing Interface (MPI) an.

Abbildung 1.1 Vom mathematischen Modell zum Simulationsergebnis.



Nachdem die Simulation entwickelt und getestet wurde, kann man damit beginnen, diese in vielerlei Richtungen weiter zu untersuchen. Abbildung 1.2 zeigt die in dieser Arbeit getätigten

Abbildung 1.2 Überblick über die in dieser Arbeit verwendeten Technologien und Modelle



Untersuchungen.

Zum Beispiel können geeignete Werte für die Eingabeparameter bestimmt werden, die a priori unbekannt sind. Dies geschieht mit Hilfe von Parameterschätzung. Dabei werden ausgehend von Messwerten eines physikalischen Experiments die Eingabeparameter einer Simulation mit Hilfe von Optimierungsalgorithmen angepasst, bis die Simulation in einer bestimmten Genauigkeit die gleichen Simulationsergebnisse liefert wie die Messwerte. Ein weiterer Punkt zur Untersuchung ist die optimale Versuchsplanung sowie die Modelldiskriminierung. Bei der optimalen Versuchsplanung werden die Versuchsbedingungen des physikalischen Experiments optimiert, um möglichst fehlertolerante Messwerte zu generieren. Bei der Modelldiskriminierung wird versucht, unter verschiedenen mathematischen Modellen ein optimales Modell auszuwählen.

Für alle diese weiterführenden Überlegungen benötigt man Optimierungsalgorithmen, welche wiederum Ableitungen benötigen. Wir werden in dieser Arbeit die Technik des automatischen Differenzierens einsetzen, um die Ableitungen unserer numerischen Simulationen zu erzeugen. Um die Geschwindigkeit der Ausführung zu gewährleisten, wird Parallelisierung eingesetzt. Dies gilt sowohl im Hinblick auf die Simulation als auch für die weiterführenden Überlegungen.

1.2 Motivation

Nach dem Erdbeben und der daraus resultierenden Tsunamikatastrophe im Jahre 2011 in Japan wurde wieder einmal klar, dass Kernenergie nicht mehr als sicher gelten darf. Zudem zeigte sich durch die Abschaltung der restlichen Atomkraftwerke, dass diese Form der Energieerzeugung nicht als zuverlässig gelten kann. Die alternativen zur Kernenergie sind vielfältig. Fossile Energieträger wie Kohle, Gas und Öl sind nicht in unendlichen Vorkommen auf unserem Planeten zu finden, das heißt, sie werden früher oder später zur Neige gehen und können uns nicht mehr zur Stromerzeugung dienen. Dieser Zeitpunkt ist noch lange nicht erreicht, Schätzungen gehen von einer Reichweite der Kohle von etwa 400 Jahren aus [1]. Der Nachschub von Öl und Gas wird jedoch schon deutlich früher versiegen, in etwa 50 Jahren sind die Ressourcen aufgebraucht. Auf der anderen Seite sind diese fossilen Energieträger unter den schlimmsten Klimakiller unserer Zeit, da sie die Erderwärmung antreiben und den sog. Treibhauseffekt begünstigen.

Die regenerative Energieerzeugung auf der anderen Seite ist klimafreundlich und sicher. Als Beispiele, die sowohl ökologisch als auch gesellschaftlich akzeptiert sind, gelten Wind-, Sonnen- und Wasserkraft. Wind und Wasser wurden schon seit Jahrhunderten für die Energieerzeugung genutzt, hier sind Windmühlen sowie Wassermühlen die ältesten Vertreter. Regenerative Energien tragen in starkem Maße dazu bei, unsere Gesellschaft, mit ihrem immer weiter steigenden Energiebedarf, nachhaltig mit „grüner“ Energie zu beliefern. Die genannten Energieformen, Wind, Sonne und Wasser, haben jedoch einen entscheidenden Nachteil. Aufgrund von Wettereinflüssen können sie nur begrenzt als Basisträger der Stromversorgung dienen. Eine andere Form der Energieerzeugung ist, geothermische Energie zu nutzen. Tief im Inneren der Erde findet sich ein riesiges Potential ungenutzter Energie. Diese kann zum einen als thermische Energie genutzt werden, um zum Beispiel Städte und Schwimmbäder zu heizen. Durch sogenannte Kraft-Wärmekopplung kann die Energie aber auch in elektrische Energie umgewandelt werden. Darüber hinaus können geothermische Kraftwerke als „baseload“ Energieversorger dienen.

Um die Risiken der Geothermie besser abzuschätzen, wurde das Verbundprojekt MeProRisk II, als Nachfolger von MeProRisk, initiiert. Dabei arbeiten die Universitäten Rheinisch-Westfälische Technische Hochschule Aachen, Friedrich-Schiller-Universität Jena, Universität Kiel, Technische Universität Freiberg, Freie Universität Berlin, University of Western Australia aus Perth in Australien, und die VIGOR Research Group aus Gent in Belgien, sowie die Industriepartner Geophysika GmbH aus Aachen und die ENEL Green Power S.p.A. aus Rom in Italien zusammen.

Bevor man geothermische Energie nutzen kann, muss eine lohnenswerte Position für ein Bohrloch gefunden werden. Diese Lokation muss verschiedene Aspekte des Untergrundes, wie zum Beispiel die thermische Speicherbarkeit und Fluid-Durchlässigkeit der geologischen Schichten, beachten. Des weiteren sollte beachtet werden, nicht in seismisch aktive Schichten zu bohren, da es dadurch zu Erdbeben und möglichen Schäden an der Erdoberfläche kommen kann.

Die Auswahl einer geeigneten Stelle für ein Bohrloch ist schwierig und zeitaufwändig. Es muss eine Vielzahl von Daten über den Untergrund gesammelt werden. Hierzu werden zuerst seismische Reflexions-Messungen des Untergrundes angefertigt, welche dann von Experten ausgewertet werden. Sollten diese Auswertungen verheißungsvoll sein, können anschließend mit Hilfe von Explorationsbohrungen weitere Erkenntnisse über die Zusammensetzung des Untergrundes gewonnen werden. Dazu werden an verschiedenen Stellen der untersuchten Region Bohrlöcher gebohrt, in denen diverse Gesteinsparameter gemessen werden. Außerdem können sog. Tracer in den Untergrund gebracht werden, welche dann an anderer Stelle gemessen werden. Damit lässt sich zum Beispiel der Wasserfluss im Untergrund messen. Ausgehend von diesen Daten kann ein Computermodell erstellt werden, welches die Vorgänge im Untergrund simuliert und wichtige Erkenntnisse über die endgültige Position des Arbeitsbohrlochs bieten kann.

Dabei sollte jedoch größte Vorsicht gelten. Zum einen kann ein falsch platziertes Bohrloch unter Umständen nicht wirtschaftlich sein, zum anderen können die Veränderungen im Erdreich auch katastrophale Folgen, wie Erdbeben oder Gasexplosionen, zur Folge haben. Außerdem zeigt eine Studie des BMU [2], dass es ungefähr 4 Mio. Euro kostet, ein einziges 5 km tiefes Bohrloch

Abbildung 1.3 2D Modell der Gesteinsschichten in der Region Perth

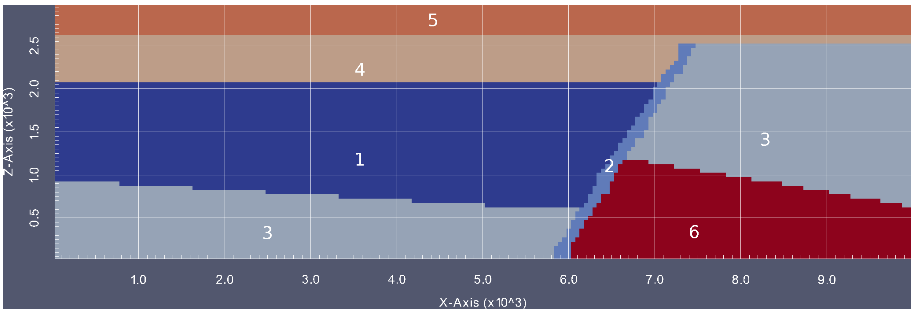
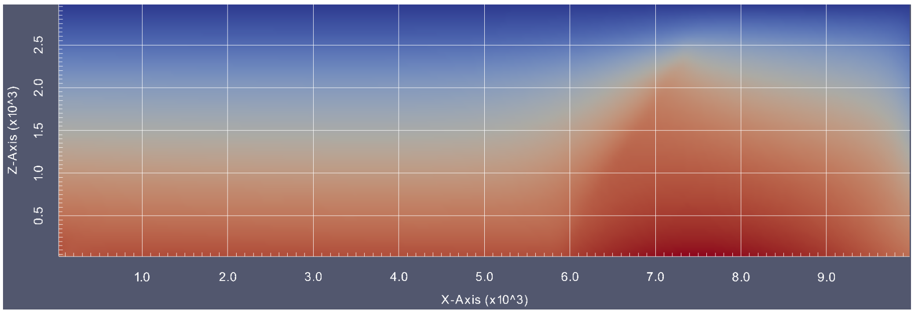


Abbildung 1.4 Simulation der Temperatur im 2D Modell der Region Perth.



niederzubringen.

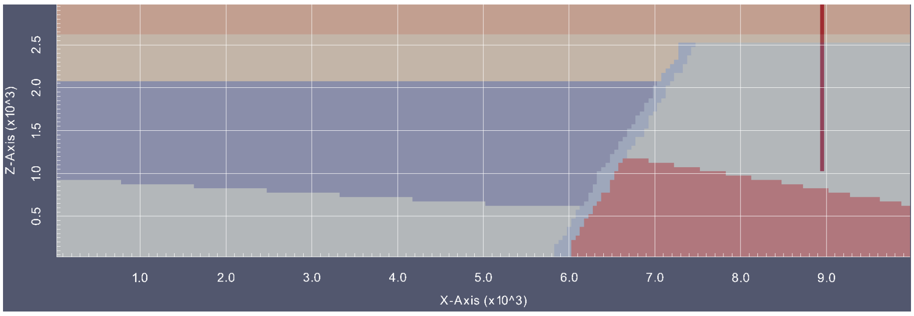
Als Illustration soll hier kurz ein einfaches Gesteins-Modell der Region um Perth, Australien dienen. Abbildung 1.3 zeigt die sechs Gesteinsschichten in der Region in einem 2D-Schnitt. Das Modell ist 10 km breit und betrachtet eine Tiefe von 3 km mit einer Auflösung von 50 Metern. Das Modell besteht aus den folgenden 6 Gesteinsschichten:

(1) Parmelia Formation
(2) Bruchzone (engl. Fault)
(3) Yarragadee-Formation – Der Hauptgrundwasserleiter (Aquifer) des Beckens
(4) South Perth Shale – Die niedrig permeable Deckschicht des Reservoirs
(5) Leederville Formation – Die oberste permeable Deckschicht der Formation
(6) Cattamarra Coal Measures – Permeable Schicht unterhalb der Yarragadee Formation

Ausgehend von diesem Modell mit entsprechenden Parametern ist es möglich, die Wärmeausbreitung im Untergrund zu simulieren. Diese ist in Abb. 1.4 beispielhaft zu sehen. Die Gesteinsschicht (6) ist ein natürliches Reservoir an heißem Gestein, welches die Umgebung (in Form von Wasser) wärmt. Man sieht, dass die Kluftzone (2) als Barriere für die Wärmeausbreitung dient. Dies liegt daran, dass die Schicht eine niedrige Permeabilität besitzt und damit Wasser nicht gut hindurch fließen kann, was zu einem verminderten Wärmetransport führt. Wenn man das Simulationsmodell mit hoher Permeabilität rechnet, würde ein größerer Wärmetransport stattfinden und die Wärmeverteilung wäre gleichmäßiger.

Die Daten für dieses Modell entstammen [3], welches ein Explorationsbohrloch beschreibt, dessen Position in Abb. 1.5 zu sehen ist.

Abbildung 1.5 Explorationsbohrloch im 2D Modell der Region Perth.



1.3 Problemstellung und Lösungsansätze

Ausgehend von diesem Modell ergeben sich weitere Fragestellungen, die in dieser Arbeit betrachtet werden sollen:

1. Welche Parameter müssen für gegebene Messwerte gelten? (Parameterbestimmung)
2. Wo müssen Messwerte aufgenommen werden, damit die gewonnenen Erkenntnisse über das Modell maximal werden? (Optimale Versuchsplanung)
3. Welches Modell liefert die beste Beschreibung des Problems? (Modelldiskriminierung)
4. Kann man die Problemlösung durch den Einsatz von einfacheren Modellen beschleunigen? (Space-Mapping)

Abbildung 1.2 zeigt die in dieser Arbeit untersuchten Fragestellungen und deren Einfluss auf den Arbeitsablauf in einem größeren Projekt.

Zur Lösung dieser Fragestellungen müssen sog. Enabling Technologies betrachtet werden. Zu diesen zählen

1. Optimierungsalgorithmen
2. Automatisches Differenzieren
3. Parallelisierung

1.4 Aufbau der Arbeit

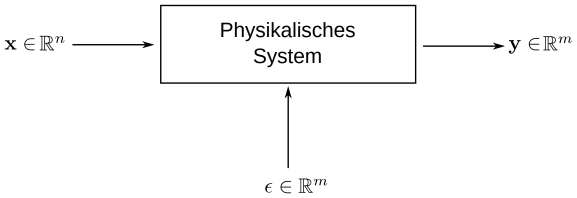
Die Arbeit gliedert sich in sechs Hauptkapitel. In Kapitel 2 werden die verwendeten Problemklassen beschrieben, die in dieser Arbeit untersucht werden. Dabei werden mit Hilfe von einfachen Testbeispielen die grundlegenden Theorien beschrieben, welche dann anhand zweier komplexerer realer Anwendungen aus der Geothermie untersucht werden. Dabei spielen sowohl die Definition des Vorwärtsproblems als auch die anschließenden inversen Problemstellungen eine entscheidende Rolle. Nachdem die wichtigsten Problemklassen definiert wurden, wird in Kapitel 3 genauer auf die sogenannten Enabling Technologies eingegangen. Ausgehend von diversen Optimierungsalgorithmen werden weitere Verbesserungen untersucht. Dabei spielen die automatische Erzeugung von Ableitungen sowie der geschickte Einsatz von Parallelisierungen eine wichtige Rolle zum Lösen der vorgestellten Probleme. Alle Problemklassen werden mit dem in Kapitel 4 vorgestellten Framework EFCOSS, dem Environment For Combining Optimization and Simulation Software, bearbeitet. Diese Software unterstützt den Entwickler durch eine einfache Kopplung zwischen Simulationswerkzeugen und Optimierungsalgorithmen. Ursprünglich an der RWTH Aachen von Arno Rasch und anderen entwickelt [4], wird diese Software in der Arbeit grundlegend überarbeitet, angepasst und um neue Funktionalitäten erweitert. Dabei wird besonderer Wert auf eine einfache Nutzbarkeit und automatische Integration von algorithmisch generierten Ableitungen gelegt. Darüber hinaus wird die Verwendung von Parallelisierung thematisiert. Kapitel 5 beschreibt die Anwendung der vorgestellten Technologien auf die definierten Problemklassen. Hierbei wird im speziellen die Kopplung eines umfangreichen Simulationswerkzeuges SHEMAT [5, 6] – Simulation of HEat and MaterialTransport – mittels EFCOSS an diverse Optimierer gezeigt und die verschiedenen gegebenen Problemstellungen gelöst, sowie numerische Ergebnisse vorgestellt und ausgewertet und deren Laufzeiten betrachtet. Darüber hinaus wird die Einsatzmöglichkeit der Modellidentifikation anhand eines Problems aus der Werkstofftechnik gezeigt. Die Arbeit endet mit einer Zusammenfassung und einem Ausblick auf weitergehende Fragestellungen.

Kapitel 2

Problemklassen

Bei der Modellierung von physikalischen Vorgängen ist es oft erforderlich, die dem Modell zugrunde liegenden Modellparameter experimentell zu bestimmen. Dabei ist der Aufbau eines solchen Experiments in Abb. 2.1 schematisch zu sehen. Ausgehend von einer Eingabe $\mathbf{x} \in \mathbb{R}^n$, den Experimentierbedingungen, auch Erklärungsvariablen (engl. explanatory variables) genannt, antwortet das System mit einer Ausgabe $\mathbf{y} \in \mathbb{R}^m$. Zufällige Messfehler $\epsilon \in \mathbb{R}^m$ verfälschen dabei die Ausgabe, können aber nicht gemessen werden.

Abbildung 2.1 Schematischer Aufbau eines Experiments mit Eingabe $\mathbf{x}$, Ausgabe $\mathbf{y}$ und Messfehler ϵ.



Im Allgemeinen geht man davon aus, dass diese Fehler sich nicht gegenseitig beeinflussen, sondern lediglich additiv den Messwert beeinflussen. Eine Annahme besagt, dass die Fehler ϵ normalverteilt sind,

$$\epsilon \propto \mathcal{N}(\mu, \sigma^2),$$

mit Mittelwert μ und Varianz σ^2 [7]. In diesem Kapitel sollen ausgehend von dieser Definition eines Experiments verschiedene Problemklassen definiert werden. Zuerst wird ein mathematisches Modell aufgestellt, das sogenannte Vorwärtsproblem. Ausgehend von diesem können inverse Fragestellungen betrachtet werden. Diese basieren auf der Ableitung des Vorwärtsproblems

2.1 Vorwärtsproblem

Als Vorwärtsproblem wird in dieser Arbeit ein mathematisches Modell bezeichnet, welches ein bestimmtes physikalisches, chemisches oder anderes wissenschaftliches Phänomen beschreibt. Dieses Modell soll dabei möglichst akkurat die Gegebenheiten errechnen und dabei parametrisierbar sein.

Ein solches parametrisierbares Modell ist definiert als eine Funktion

$$\mathbf{y} = \mathbf{f}(\mathbf{x}, \theta), f : \mathbb{R}^{n+\xi} \to \mathbb{R}^m \tag{2.1}$$

mit den Parametern $\mathbf{x} \in \mathbb{R}^n$ und den Versuchsbedingungen $\theta \in \mathbb{R}^\xi$ als Eingaben und den Ausgaben $\mathbf{y} \in \mathbb{R}^m$. Dabei gehen wir im Allgemeinen davon aus, dass die Funktion $\mathbf{f}$ nichtlinear in $\mathbf{x}$ und θ ist.

Das diskretisierte Modell ist meist sehr komplex und besteht aus einer Vielzahl linearer oder nicht-linearer Gleichungen und Lösern, welche eine Simulationssoftware mit entsprechender Domänen-Diskretisierung zur Verfügung stellt. Des Weiteren kann die Anzahl der Parameter $\mathbf{x}$ für das Modell sehr groß sein, etwa wenn jeder Diskretisierungspunkt seine eigenen Parameter erhält.

2.1.1 Minpack-2 Testsuite

Um die im Weiteren verwendete Software zu testen, bietet es sich an, auf einfache Testbeispiele aus der Literatur zurückzugreifen. Der Vorteil besteht darin, dass die Ergebnisse der Tests gut dokumentiert und damit auch nachvollziehbar sind. Sie dienen im wesentlichen dazu, die Funktion der Software an einfachen Beispielen zu erklären.

Problemdefinition

Zum Testen werden ausgewählte Beispiele aus der Minpack-Testsuite [8] verwendet. Sie wurde 1992 erstmals vorgestellt und enthält eine Auswahl von wissenschaftlichen Problemen, welche in drei Bereiche eingeteilt werden:

1. nichtlineare Gleichungssysteme,
2. kleinste-Quadrate Probleme und
3. allgemeine Minimierungsprobleme.

Zu jedem dieser Bereiche finden sich acht Beispiele aus realen Anwendungen und deren Lösung. Die Beispiele umfassen Anwendungen aus den Bereichen Fluid Dynamik, Medizin, Elastizität, Verbrennung, molekulare Bindung, nicht-zerstörende Tests, chemische Kinetik, Lubrikation und Supraleiter. Des Weiteren wird Fortran-Code angegeben, der die Vorwärtsprobleme zusammen mit geeigneten Startwerten, Beispieldaten und Nebenbedingungen löst.

2.1.2 Geothermie

Am Institut für angewandte Geophysik und geothermische Energie (GGE - Institute for Applied Geophysics and Geothermal Energy) des E.ON Energy Research Center an der RWTH Aachen wurde eine Software zur Simulation von geothermischen Prozessen entwickelt – der „Simulator for HEat and MAss Transport“ (SHEMAT) [5]. Mit dieser Software werden in drei Raumdimensionen gekoppelte transiente Gleichungen für Grundwasserfluss, Wärmetransport und Transport von reaktiven gelösten Stoffen in porösen Gesteinen für hohe Temperaturen berechnet.

Problemdefinition

Als Vorwärtsproblem der Geothermie wird in dieser Arbeit das mathematische Modell für Fluidfluss und Wärmetransport betrachtet. Für den Fluidfluss wird das sogenannte hydraulische Potential h [N kg^{-1}], auch *head* genannt, also die Wasserstandshöhe, die einem bestimmten Druck entspricht, mit der Formel

$$\nabla \cdot \left[\frac{\rho_f g \kappa}{\mu_f} \cdot (\nabla h + \rho_r \nabla z) \right] + W = S \frac{\partial h}{\partial t} \tag{2.2}$$

berechnet.

Dabei sind ρ_f und ρ_r [kg m^{-3}] die Dichten des Fluids und des Gesteins, der hydraulische Permeabilitäts-Tensor wird mit κ [m^2] bezeichnet. Die dynamische Viskosität des Fluids wird durch μ_f [Pa s] gekennzeichnet. Mit g wird die Gravitationskonstante bezeichnet und W [m^3 s^{-1}] korrespondiert mit einem Massenquellterm, der durch zu- und abfließendes Wasser entsteht. Der spezifische Speicherkoeffizient S [m^{-1}] auf der rechten Seite der Gleichung ist durch

$$S = \rho_f g(\alpha + \psi\beta),$$

definiert, mit α und β als Kompressibilität der Gesteins- und Fluid-Phase [m^{2}N^{-1}] und ψ [m^{-1}] der Porosität des Gesteins, definiert.

Die Temperatur T [K] wird durch die konduktiv-advektive Wärmetransportgleichung

$$\nabla \cdot (\lambda_e \nabla T) - (\rho c)_f \, \mathbf{a} \cdot \nabla T + H = (\rho c)_e \frac{\partial T}{\partial t} \tag{2.3}$$

beschrieben.

Hier bezeichnet $(\rho c)_e$ $[\mathrm{J\,kg^{-1}\,K^{-1}}]$ die effektive Wärmekapazität des gesättigten porösen Gesteins und des Fluids, λ_e $[\mathrm{W\,m^{-1}\,K^{-1}}]$ ist die effektive Wärmeleitfähigkeit. Darüber hinaus steht $(\rho c)_f$ $[\mathrm{J\,kg^{-1}\,K^{-1}}]$ für die volumetrische Wärmekapazität des Fluids und H $[\mathrm{W\,kg^{-1}\,K^{-1}}]$ führt einen möglichen Wärmequellterm ein.

Die Gleichungen (2.2) und (2.3) sind durch das Gesetz von Darcy miteinander verbunden, so dass die Geschwindigkeit

$$\mathbf{a} = -\frac{\rho_f g}{\mu_f} \kappa \cdot \nabla h \tag{2.4}$$

erfüllt. Damit gibt es für die Temperatur T eine indirekte Abhängigkeit (über $\mathbf{a}$) vom hydraulischen Potential h. Zusätzlich sind die in den Differentialgleichungen vorkommenden Koeffizienten abhängig von der Temperatur und dem Druck des Fluids, welcher wiederum von der Tiefe sowie der Verteilung des hydraulischen Potentials abhängt.

Die in der Problemstellung vorkommenden partiellen Differentialgleichungen können mithilfe eines zellbasierten Finite-Differenzen Verfahrens diskretisiert und gelöst werden. Für gültige Initial- und Randbedingungen gibt es folglich eine numerische Lösung der Gleichungen (2.2)–(2.4), welche jeweils durch ein lineares Gleichungssystem approximiert werden können. Diese Gleichungssysteme hängen dann wechselseitig von vorherigen Werten der entsprechenden Größe ab. So hängt das lineare Gleichungssystem

$$K(T^{(t)}, h^{(t)}) T^{(t+1)} = b(T^{(t)}, h^{(t)}) \tag{2.5}$$

für die Berechnung der Temperatur T im Zeitschritt $t+1$ von der vorhergehenden Lösung der Temperatur T und des hydraulischen Drucks h im Zeitschritt t ab. Durch Anwendung einer einfachen Picard-Iteration[1] können die nichtlinear gekoppelten Systeme von linearen Gleichungssystemen gelöst werden.

Nach Diskretisierung der Domäne wird die in der Gleichung

$$-(1-\omega) K \cdot h^{(t)} - R \cdot h^{(t)} - W = \omega K \cdot h^{(t+1)} - R \cdot h^{(t+1)} \tag{2.6}$$

gezeigte iterative Lösung der Strömungsgleichung gelöst. Hier bezeichnet ω den Zeitgewichtungsparameter, welcher die Implizitheit und Explizitheit der Lösung steuert.

Das Symbol R wird durch die Gleichung

$$R = \frac{\rho_f g (\alpha + \psi \beta)}{\Delta t},$$

beschrieben. Die Konduktivitätsmatrix K ergibt sich zu

$$K = A h_{l_z-1} + B h_{l_y-1} + C h_{l_x-1} + D h + E h_{l_x+1} + F h_{l_y+1} + G h_{l_z+1},$$

mit A, B, C, D, E, F und G als Koeffizienten, welche durch die Diskretisierung entstehen. Das Symbol h_{l_x, l_y, l_z} beschreibt die Variable des hydraulischen Potential h in den dreidimensionalen

---

[1] Nach Charles E. Picard benannte Fixpunktiteration zur approximativen Lösung von gewöhnlichen Differentialgleichungen.

Zellen (l_x, l_y, l_z). Um die Übersichtlichkeit zu erhöhen, wurden die Indizes vereinfacht. So steht h_{l_x+1} für die Variable h in X-Richtung um 1 erhöht, also h_{l_x+1,l_y,l_z}, und h bezeichnet den aktuell betrachteten Punkt h_{l_x,l_y,l_z}. Eine genauere Betrachtung findet sich in [5, 9].

Die Gleichung (2.6) löst die zeitinvariante Strömungsgleichung. Zur Lösung des zeitvarianten Problems muss eine Zeitschrittiteration gelöst werden, welche in Algorithmus 1 gezeigt wird.

Algorithmus 1 Ablauf der Fix-Punkt-Iteration für das nichtlineare Geothermieproblem zur Lösung der zeitvarianten Kopplung von hydraulischem Potential h und der Temperatur T nach [5].

```
for $t = 0, \dots, t_{\text{end}}$ do
    // Initiale Vermutung für $h$ und $T$ für den aktuellen Zeitschritt
    $h_0^{(t+1)} = h^{(t)}$ and $T_0^{(t+1)} = T^{(t)}$
    for $k = 1, 2, \dots, \max$ (Fixed-point iteration) do
        // Berechne das hydraulische Potential $h$
        Setup der Koeffizienz-Matrix $K(h_{k-1}^{(t+1)}, T_{k-1}^{(t+1)})$
        Berechne rechte Seite $b$
        Löse $Kh_k^{(t+1)} = b$ nach $h_k^{(t+1)}$
        Berechne die Residuen: $RE_k(h_k) = \|h_k^{(t+1)} - h_{k-1}^{(t+1)}\|$
        Adaptive Relaxation: $h_k^{(t+1)} = (1 - \delta_k)h_{k-1}^{(t+1)} + \delta_k h_k^{(t+1)}$
        // Berechne die Temperatur $T$
        Setup der Koeffizienz-Matrix $K(h_k^{t+1}, T_{k-1}^{(t+1)})$
        Berechne rechte Seite $b$
        Löse $KT_k^{(t+1)} = b$ nach $T_k^{(t+1)}$
        Berechne Residuum $RE_k(T_k)$
        Adaptive Relaxation: $T_k^{(t+1)} = (1 - \delta_k)T_{k-1}^{(t+1)} + \delta_k T_k^{(t+1)}$
        // Überprüfe auf Konvergenz
        if $\max \|RE_k\| < \epsilon$ then
            $h^{(t+1)} = h_k^{(t+1)}$
            $T^{(t+1)} = T_k^{(t+1)}$
            break
        end if
    end for
end for
```

2.1.3 Metallplastizität

In der Metallformungsindustrie werden diverse Tests für Werkstücke durchgeführt. Dabei werden die verwendeten Werkstücke meist über die Zeit einer definierten Kraft ausgesetzt und ihr Verhalten überwacht. Auch bei diesen Aufgaben helfen Simulationsmodelle, eine Vorhersage über das erwartbare Verhalten der Werkstücke zu erzeugen und die Anzahl kostenintensiver realer Tests zu minimieren.

Problemdefinition

Materialmodelle werden häufig für die Simulation von Deformationsprozessen zur Beschreibung von Härtungsprozessen und die Bestimmung der resultierenden Form eingesetzt, das sogenannte “finite strain plasticity with isotropic and kinematic hardening”. Hierbei werden mittels Finite-Elemente-Methoden plastische Verformungsprozesse untersucht, die mathematisch mit den folgenden Formeln beschrieben werden. Weiterführend sei auf [10] verwiesen.

Als Input dient eine externe Kraft $\mathbf{F}(t)$, der Output ist die Antwort des Materials $\sigma(t)$. Zur Beschreibung des Materials werden die in Tabelle 2.1 gegebenen Materialparameter genutzt.

Tabelle 2.1 Materialparameter aus [11]

μ	Shear Modulus [MPa]
Λ	Lamé Konstante [MPa]
σ_y	Initial yield stress [MPa]
b	Rate of change of kinematic hardening modulus
c	Initial kinematic hardening modulus
β	Rate of change of elastic range size
Q	Maximum change in size of elastic range [MPa]

- Der zweite Piola-Kirchhoff Last-Tensor $\mathbf{S}$ sowie der Gegen-Last-Tensor $\mathbf{X}$ sind definiert durch

$$\mathbf{S} = \mu(\mathbf{C}_p^{-1} - \mathbf{C}^{-1}) + \frac{\Lambda}{2}\left(\det \mathbf{C}(\det \mathbf{C}_p)^{-1} - 1\right)\mathbf{C}^{-1} \tag{2.7}$$

$$\mathbf{X} = c(\mathbf{C}_{p_i}^{-1} - \mathbf{C}_p^{-1}), \tag{2.8}$$

- Die effektiven Belastungsquantitäten $\mathbf{Y}$ und $\mathbf{Y}_{\text{kin}}$

$$\mathbf{Y} = \mathbf{CS} - \mathbf{C}_p\mathbf{X}, \mathbf{Y}_{\text{kin}} = \mathbf{C}_p\mathbf{X} \tag{2.9}$$

- Plastische Fluss-Regel

$$\dot{\mathbf{C}}_p = 2\dot{\lambda}\frac{\mathbf{Y}^D\mathbf{C}_p}{\sqrt{\mathbf{Y}^D \cdot (\mathbf{Y}^D)^T}} \tag{2.10}$$

- Evolutionsgleichung für kinematische Aushärtung

$$\dot{\mathbf{C}}_{p_i} = 2\dot{\lambda}\frac{b}{2}\mathbf{Y}_{\text{kin}}^D\mathbf{C}_{p_i} \tag{2.11}$$

- Evolutionsgleichung für isotropische Aushärtung

$$\dot{\kappa} = \sqrt{\frac{2}{3}}\dot{\lambda} \tag{2.12}$$

- Ertragsfunktion des von Mises Typs

$$\phi = \sqrt{\mathbf{Y}^D \cdot (\mathbf{Y}^D)^T} - \frac{2}{3}(\sigma_y + Q(1 - e^{-\beta\kappa})) \tag{2.13}$$

- Kuhn-Tucker Bedingungen

$$\dot{\lambda} \geq 0,\ \phi \leq 0,\ \dot{\lambda}\phi = 0. \tag{2.14}$$

Dabei ist $\mathbf{C} = \mathbf{F}^T\mathbf{F}$ der rechte Cauchy-Green Deformationstensor, $\mathbf{C}_p = \mathbf{F}_p^T\mathbf{F}_p$ der rechte Cauchy-Green Plastizitätstensor und $\mathbf{C}_{p_i} = \mathbf{F}_{p_i}^T\mathbf{F}_{p_i}$ die kinetische Aushärtungsvariable. Der Deformations-Gradient $\mathbf{F}$ setzt sich zusammen aus $\mathbf{F} = \mathbf{F}_e\mathbf{F}_p = \mathbf{F}_l\mathbf{F}_{p_e}\mathbf{F}_{p_i}$.

2.2 Inverse Problemstellungen

Um die in der Einleitung genannten Fragestellungen beantworten zu können, genügt es nicht, die Simulation einfach durchzuführen. Vielmehr muss für die Beantwortung der Frage die Simulationsfunktion abgeleitet werden. Man spricht dann von der sogenannten inversen Problemstellung. Der folgende Abschnitt geht auf diverse inverse Fragestellungen ein und gibt einige Beispiele.

2.2.1 Parameterbestimmung

Zur Beantwortung der Frage „Welche Parameter müssen für gegebene Messwerte gelten?“ wird die sogenannte Parameterbestimmung oder Parameterschätzung eingesetzt.

Problemdefinition

Nachdem ein stabiles Vorwärtsmodell etabliert wurde, können weitere interessante Fragestellungen untersucht werden. Die wohl bedeutendste Fragestellung ist die sogenannte Parameterbestimmung (Parameter Estimation). In einer Simulation werden die zugrunde liegenden Parameter meist aus Erfahrungswerten der Wissenschaftler gewählt, um die reale Welt möglichst exakt nachzubilden.

Sei **f** die Vorwärtsproblemdefinition aus Formel (2.1). Gesucht ist die Zielfunktion

$$\min_{\mathbf{x}} r(\mathbf{f}, \mathbf{x}, \theta), \tag{2.15}$$

mit

$$r(\mathbf{f}, \mathbf{x}, \theta) = \|\mathbf{d}(\mathbf{f}, \mathbf{x}, \theta)\| = \|\mathbf{f}(\mathbf{x}, \theta) - \Phi_{\text{mess}}(\theta)\|. \tag{2.16}$$

Der Abstand **d** zwischen dem Modell **f** und den Messwerten Φ_{mess} für gegebene Eingaben wird auch Residuum genannt. Hier bezeichnen Φ_{mess} die Messwerte für den Simulationswert **f** und $\|\cdot\|$ die 2-Norm. Diese Norm kann aber unter Umständen durch eine andere Norm ersetzt werden, je nach Einsatzzweck.

Um ein Minimum des Parameterfitproblems mit $\|\cdot\|$ als Euklidische-Norm zu bestimmen, wird die folgende Ableitung benötigt:

$$\nabla_{\mathbf{x}} r = \nabla_{\mathbf{x}} \|\mathbf{f}(\mathbf{x}, \theta) - \Phi_{\text{mess}}\| = \frac{\sum_i (f_i(\mathbf{x}, \theta) - \Phi_{\text{mess}_i}) \cdot \nabla_{\mathbf{x}} f_i(\mathbf{x}, \theta)}{\|\mathbf{f}(\mathbf{x}, \theta) - \Phi_{\text{mess}}\|} = \frac{J^T \mathbf{d}(\mathbf{f}, \mathbf{x}, \theta)}{\|\mathbf{d}(\mathbf{f}, \mathbf{x}, \theta)\|} \tag{2.17}$$

Dabei ist $J = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ die Jacobi-Matrix der Funktion nach den Parametern.

Stochastische und deterministische Lösungsmethoden

Zur Lösung des oben gegebenen Problems gibt es zwei verschiedene Ansätze. Zum einen die stochastischen Methoden, welche in der Literatur auch Monte-Carlo Methoden genannt werden. Es werden viele verschiedene Funktionsauswertungen des Vorwärtsproblems durchgeführt, jeweils mit unterschiedlichen Parametersätzen.

Im Bereich der Geothermie findet sich ein Buch von Tarantola [12], in welchem stochastische Methoden, wie die Gausssche Inversion und Ensemble-Kalman-Filter besprochen werden.

Auf der anderen Seite finden sich deterministische Verfahren. Diese benötigen neben der Evaluierung der Vorwärtsfunktion auch die Evaluation der abgeleiteten Vorwärtsfunktion. Dabei benötigen sie im Allgemeinen deutlich weniger Funktionsauswertungen als die stochastischen Verfahren. Das Konvergenzverhalten kann jedoch maßgeblich von der Güte der verwendeten Ableitungen bestimmt werden. Mehr Details finden sich in [7, 13, 14]. In dieser Arbeit wird im Weiteren nur auf die deterministischen Methoden eingegangen.

Minpack-2 Testbeispiele

Testbeispiel 1: Exponential Datafitting Gegeben seien die Messwerte aus Tabelle 2.2.

Tabelle 2.2 Messwerte des Exponential Datafit Problems aus der Minpack-2 Testsuite.

i	Φ_i	i	Φ_i	i	Φ_i	i	Φ_i
1	0.844	11	0.751	21	0.506	31	0.414
2	0.908	12	0.718	22	0.49	32	0.411
3	0.932	13	0.685	23	0.478	33	0.406
4	0.936	14	0.658	24	0.467		
5	0.925	15	0.628	25	0.457		
6	0.908	16	0.603	26	0.448		
7	0.881	17	0.58	27	0.438		
8	0.85	18	0.558	28	0.431		
9	0.818	19	0.538	29	0.424		
10	0.784	20	0.522	30	0.42		

Gesucht werden die Parameter $\mathbf{x} = (x_1, \ldots, x_5) \in \mathbb{R}^5$, für die $\sum_{i=1}^{33} \|f_i(\mathbf{x}) - \Phi_i\|$ minimal wird, wobei die Funktionen $f_i : \mathbb{R}^5 \to \mathbb{R}$ für $t_i = 10(i-1)$ wie folgt definiert ist:

$$f_i(\mathbf{x}) = x_1 + x_2 \exp(-t_i x_4) + x_3 \exp(-t_i x_5) \tag{2.18}$$

Als Zielfunktion ist hier der Abstand zwischen dem berechneten Wert f_i und den Messwerten Φ_i zu minimieren

$$\min_{\mathbf{x}}(\|f_i(\mathbf{x}) - \Phi_i\|.$$

Quellcode 2.1 zeigt den korrespondierenden Fortran-Code für die Evaluierung von $\mathbf{f}(\mathbf{x})$.

Quellcode 2.1: Fortrancode für das Testbeispiel des Exponential Datafitting.

```fortran
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
  integer m
  double precision x1,x2,x3,x4,x5
  double precision fvec(m)
  integer i
  double precision temp,temp1,temp2

  do i = 1, m
      temp = dble(10*(i-1))
      temp1 = exp(-x4*temp)
      temp2 = exp(-x5*temp)
      fvec(i)=(x1+x2*temp1+x3*temp2)
  end do
end
```

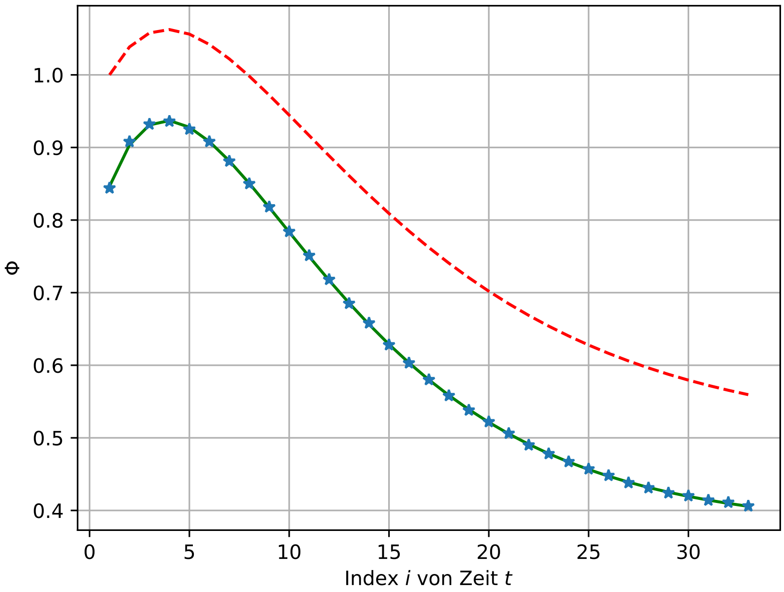
Zuerst wollen wir das unter Abschnitt 2.2.1 beschriebene Datenfitproblem angehen. Ein möglicher Startwert ist der Vektor

$$\mathbf{x}_0 = (0.5, 1.5, -1.0, 0.01, 0.02). \tag{2.19}$$

Als Lösung des Problems findet sich der Vektor

$$\mathbf{x}^* = (0.37541004, 1.93584515, -1.46468536, 0.01286753, 0.02212271) \tag{2.20}$$

Abbildung 2.2 Messwerte des Exponential Datafitting Problems als blaue Sterne, berechnete Werte für den Startwert $\mathbf{x}_0$ in roter gestrichelter Linie und Werte für das Optimum $\mathbf{x}^*$ in grüner Linie.



und als Zielfunktion liefert diese optimale Lösung

$$r(\mathbf{f}(\mathbf{x}^*))) = \|\mathbf{f}(\mathbf{x}^*) - \Phi_{\text{mess}}\| = 0.00739249260906. \tag{2.21}$$

Abbildung 2.2 zeigt die Messwerte als blaue Sterne (*). Die Werte für den Startwert $\mathbf{x}_0$ sind mit einer rot gestrichelten Linie markiert. Die optimale Lösung für $\mathbf{x}^*$ ist als grüne Linie angegeben. Die Linien wurden gewählt, um die Unterscheidbarkeit der errechneten Lösung von den Messwerten zu gewährleisten.

Testbeispiel 2: Enzymreaktion Als weiteres Beispiel aus der Testsuite wird das Problem 3.7 “Analysis of an Enzyme Reaction” gewählt. Das Problem stammt aus der Analyse der kinetischen Prozesse bei einer Enzymreaktion. Dabei ist es definiert als Funktion $\mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ mit $\mathbf{x} \in \mathbb{R}^n$ als

$$f_i(\mathbf{x}) = y_i - \frac{x_1(u_i^2 + u_i x_2)}{u_i^2 + u_i x_3 + x_4} \tag{2.22}$$

mit den Datenpunkten y_i und u_i. Die Standardproblemgröße ist $m = 11$ und $n = 4$.

Geothermie

Die Anwendung der Parameterschätzprobleme auf die Geothermie-Vorwärtsprobleme ist relativ einfach. Ausgehend von den unter Abschnitt 2.1.2 gezeigten Formeln (2.2)–(2.4) kann eine Parameterbestimmung realisiert werden. Sei Φ die physikalische Größe, die in einem Explorationsbohrloch mit den Koordinaten (c_x, c_y) gemessen wurde. Dabei werden die Messwerte in m verschiedenen Tiefen aufgenommen, d.h. es gibt $\mathbf{c_z} \in \mathbb{R}^m$ verschiedene Messwerte der physikalischen Größe, welche im Weiteren mit $\Phi_{\text{mess}}(c_x, c_y, \mathbf{c_z})$ bezeichnet werden. Des Weiteren werden einer oder mehrere

Tabelle 2.3 Datenpunkte für das Problem 3.7 “Analysis of an Enzyme Reaction” aus [8].

i	y_i	u_i
1	0.1957	4.0
2	0.1947	2.0
3	0.1735	1.0
4	0.16	0.5
5	0.0844	0.25
6	0.0627	0.167
7	0.0456	0.125
8	0.0342	0.1
9	0.0323	0.0833
10	0.0235	0.0714
11	0.0246	0.0625

unbekannte Parameter $\mathbf{x}$ einer oder mehrerer im Modell vorhandener Schichten gewählt, welche bestimmt werden sollen. Die Anzahl unbekannter Parameter wird mit n bezeichnet

Somit ergibt sich ein inverses Parameterschätzproblem folgender Form:

$$\min_{\mathbf{x}} \|\Phi_{\text{mess}}(c_x, c_y, \mathbf{c_z}) - \mathbf{f}(\mathbf{x}, (c_x, c_y, \mathbf{c_z}))\| \tag{2.23}$$

Wichtig dabei ist, dass die gemessene physikalische Größe auch von der Simulation berechnet werden kann.

2.2.2 Optimal Experimental Design

Zur Beantwortung der Frage „Wo müssen Messwerte aufgenommen werden, damit die gewonnenen Erkenntnisse über das Modell maximal werden?“ wird in dieser Arbeit das Optimal Experimental Design genutzt.

Problemdefinition

Die Idee des Optimal Experimental Designs (kurz OED) stammt von Kristine Smith, einer dänischen Statistikerin und Mathematikerin [15]. Dabei beschreibt Sie ein “optimales Design als eine Klasse von experimentellen Designs, die im Sinne eines statistischen Kriteriums optimal sind. Es werden Parameter mit minimaler statistischer Varianz bestimmt.”

Laut Atkinson und Donev [16] ermöglichen optimale Designs drei entscheidende Vorteile gegenüber nicht optimalen Designs:

1. Reduktion der Kosten von Experimenten
2. Anpassung an verschiedene Faktoren, wie Prozess, Mixtur oder diskrete Faktoren
3. Möglichkeit, den Design-Raum zu beschränken

Bei OED wird ein Optimierungsproblem der Form

$$\min_{\theta} \Psi(\mathbf{x}, \theta) \tag{2.24}$$

gelöst, wobei $\Psi(\mathbf{x}, \theta)$ die Unsicherheit bezüglich eines Parameters $\mathbf{x}$ für eine experimentelle Variable θ beschreibt. Für die Funktion $\Psi(\mathbf{x}, \theta)$ gibt es verschiedene Optimalitätskriterien.

- A-Optimalität: $\Psi_A(\mathbf{x}, \theta) = \log \operatorname{trace}(F^{-1}(\mathbf{x}, \theta))$

- D-Optimalität: $\Psi_D(\mathbf{x}, \theta) = -\log\det(F(\mathbf{x}, \theta))$
- E-Optimalität: $\Psi_E(\mathbf{x}, \theta) = \min \operatorname{eig}(F^{-1}(\mathbf{x}, \theta))$

Dabei wird in jedem Fall die sog. Fisher-Informations-Matrix F benötigt. Diese ist definiert als

$$F = F(\mathbf{x}, \theta) = \left(\frac{\partial f(\mathbf{x}, \theta)}{\partial \mathbf{x}}\right)^T W \left(\frac{\partial f(\mathbf{x}, \theta)}{\partial \mathbf{x}}\right) = \left(Q^T W Q\right). \tag{2.25}$$

mit den Ableitungen nach den Parametern und W als Gewichtsmatrix. Diese wird eingefügt, um bestimmte partielle Ableitungen mehr oder weniger stark in die Betrachtung der Optimalitätskriterien einfließen zu lassen.

Um das Minimierungsproblem (2.24) zu lösen, benötigt man zusätzlich zu den ersten Ableitungen der Fisher-Matrix noch die zweiten Ableitungen

$$\frac{\partial^2 f(\mathbf{x}, \theta)}{\partial \mathbf{x} \partial \theta} \tag{2.26}$$

Geothermie

Gerade die Reduktion der Kosten von Experimenten ist im betrachteten Geothermiebereich sehr sinnvoll. Außerdem lässt sich der Design-Raum zum Beispiel bei brüchigem Gestein beschränken, um spätere Probleme mit (Mikro-)Erdbeben zu minimieren.

Als Anwendung in der Geothermie können die folgenden Fragestellungen betrachtet werden:

- Wo sollte ein Explorationsbohrloch niedergebracht werden, um möglichst genaue Informationen über dem Untergrund zu erhalten?
- Wann sollten Messungen vorgenommen werden, um über die Konzentration eines Markers im Wasser den höchsten Informationsgehalt zu bekommen?

Die zweite Fragestellung wurde umfangreich in diversen Veröffentlichungen untersucht. Zumeist wurden die Ergebnisse jedoch nur an synthetischen Modellen erprobt. Die in dieser Arbeit untersuchte Fragestellung nach der Lokation eines Explorationsbohrlochs, ist in ihrer Art bisher einzigartig.

Sei $\theta = (c_x, c_y, \mathbf{c}_z)$ die Position des Explorationsbohrlochs mit $\mathbf{c}_z$ als Tiefe. Weiterhin sei $\mathbf{x}$ der untersuchte Parameter, z.B. Permeabilität κ. Dann ist

$$F(\mathbf{x}, \theta) = \left(\frac{\partial \mathbf{f}(\mathbf{x}, (c_x, c_y, \mathbf{c_z}))}{\partial \mathbf{x}}\right)^T \left(\frac{\partial \mathbf{f}(\mathbf{x}, (c_x, c_y, \mathbf{c_z}))}{\partial \mathbf{x}}\right) \tag{2.27}$$

die in diesem Fall resultierende Fisher-Matrix. Mit diesen Informationen kann die OED Optimalität untersucht werden.

2.3 Modelldiskriminierung

Die in diesem Abschnitt gezeigten Grundlagen basieren auf den Ausführungen von Schaber [17]. Ausgehend von den oben gemachten Überlegungen kommt man recht leicht zu dem Schluss, dass das verwendete Simulationsmodell als einzig wahres Modell gilt. Dies ist jedoch meist nicht der Fall. Der Statistiker George Box traf einst die Aussage "All models are wrong, but some are useful" – „Alle Modelle sind falsch, aber einige sind nützlich". Damit lassen sich ganz neue Aspekte der beschriebenen Modelle betrachten. Gesucht ist wieder ein Modell, welches das Residuum für gegebene Messwerte minimiert. Wir finden relativ leicht Modelle, welche mit sehr vielen freien Parametern beschrieben werden können. Diese Menge an freien Parametern ermöglicht eine enorme Flexibilität und erhöht damit unweigerlich die Wahrscheinlichkeit, an den richtigen Stellen den gleichen Funktionswert anzunehmen. Dabei muss dieses Modell nichts mit dem zugrundeliegenden Phänomen zu tun haben, es passt eben genau zu den Messwerten. Ein einfaches Beispiel ist ein Parameterfit mit einer Funktion beliebig hoher Ordnung. Diese Funktion kann sehr gut an die Messwerte angepasst werden, schwingt in den dazwischenliegenden Bereichen jedoch hin und her und erzeugt damit kein verlässliches Modell.

Meist hat der Wissenschaftler, welcher für die Modellerstellung zuständig ist, die Wahl zwischen verschiedenen Hypothesen, wie das untersuchte Phänomen zu beschreiben ist. Ausgehend von diesen unterschiedlichen Modellen kann nun die Frage gestellt werden, welches für einen gegebenen Prozess das beste Ergebnis liefert. Dabei kann man, wie oben erläutert, nicht einfach das Modell wählen, welches das geringste Residuum liefert. Vielmehr muss die Anzahl der Parameter ein entscheidender Faktor in der Betrachtung sein. Je weniger Parameter die Funktion benötigt, desto besser ist sie als Modell geeignet.

Damit ergeben sich drei Fälle:

1. Das Modell mit weniger Parametern passt besser als das mit mehr Parametern.
2. Bei zwei Modellen mit der gleichen Anzahl an Parametern passt eines besser.
3. Das Modell mit mehr Parametern passt besser.

Im ersten Fall haben wir nichts mehr zu tun. Da das einfachere Modell besser passt, nehmen wir immer dieses. Der zweite Fall ist nicht ganz so einfach zu beschreiben. Die Aussage, dass ein Modell besser passt, kann auch nur von den Fehlern der Messwerte herrühren. Im letzten Fall haben wir wieder die oben beschriebene Situation.

2.3.1 Verschachtelte Modelle

Wenn man Modelle miteinander vergleicht, sind diese relativ oft nah miteinander verwandt. Ein Modell ist eine einfachere Version des anderen. Dies ist dann der Fall, wenn im komplexeren Modell einige Parameter zu Null gesetzt werden können, um das einfachere Modell zu erhalten.

Seien $\mathbf{f}_1(\mathbf{x}_1, \theta)$ und $\mathbf{f}_2(\mathbf{x}_2, \theta)$ zwei Modellfunktionen, welche an Messwerte angefittet werden. Der Parametervektor $\mathbf{x}_1$ habe die Dimension r und der Parametervektor $\mathbf{x}_2$ die Dimension s mit $r < s$. Seien $\mathbf{p}(\mathbf{x}_1^*) = \mathbf{p}(\mathbf{f}_1(\mathbf{x}_1^*, \theta))$ und $\mathbf{p}(\mathbf{x}_2^*) = \mathbf{p}(\mathbf{f}_2(\mathbf{x}_2^*, \theta))$ die entsprechenden minimalen Residuen nach Formel (2.16).

Sei

$$K = \frac{(\mathbf{p}(\mathbf{x}_1^*) - \mathbf{p}(\mathbf{x}_2^*))(n - s - 1)}{\mathbf{p}(\mathbf{x}_2^*)(r - s)} \propto K_{r-s,n-s-1} \tag{2.28}$$

die Verteilungsfunktion unter der Null-Hypothese[2]. Mit K kann man den sogenannten P-Wert berechnen, die Wahrscheinlichkeit, nur durch Glück mindestens den Wert von K zu berechnen.

Das Problem dabei ist jedoch, dass der P-Wert aus einer Tabelle nachgeschlagen werden muss. Hierzu finden sich in der Literatur diverse Angaben [18].

---

[2] Die Daten Φ_{mess} werden vom einfacheren Modell $\mathbf{f}_1(\mathbf{x}_1, \theta)$ produziert.

2.3.2 Nicht-Verschachtelte Modelle

Nicht-Verschachtelte Modelle können in der Praxis deutlich einfacher berechnet werden. Hierzu werden Konzepte aus der Informationstheorie genutzt, um Modelle auszuwählen.

Dazu bieten sich zwei Kriterien an:

- das Akaike Informationskriterium (Akaike Information Criterion - AIC) [19]

  $$AIC(\mathbf{f}(\mathbf{x}^*, \theta)) = n \ln(\mathbf{p}(\mathbf{f}(\mathbf{x}^*, \theta))/n) + 2m \tag{2.29}$$

  und
- die Minimale Beschreibungslänge (Minimal Description Length - MDL)

  $$MDL(\mathbf{f}(\mathbf{x}^*, \theta)) = \frac{1}{2n} \mathbf{p}(\mathbf{f}(\mathbf{x}^*, \theta)) e^{\frac{\ln(n)(m-1)}{n-m-2}}. \tag{2.30}$$

Das Modell, welches den kleinsten Wert nach AIC oder MDL liefert, ist das bessere Modell und deshalb zu wählen.

Aus diesen Kriterien können wir Algorithmus 2 ableiten, welcher das beste Modell auswählt.

Algorithmus 2 Algorithmus zum Finden des besten Modells in der Modelldiskriminierung.

```
function FINDBESTMODEL($\mathbf{f}$, $\mathbf{\Phi}_\mathrm{m}$)
    for all $\mathbf{f}_i \in \mathbf{f}$ do
        $\mathbf{r}_i(\mathbf{f}_i(\mathbf{x}_i^*, \theta_i)) = \min_{\mathbf{x}_i} \|\mathbf{f}_i(\mathbf{x}_i, \theta_i) - \mathbf{\Phi}_\mathrm{m}\|^2$
        Compute either $AIC_i(\mathbf{f}_i(\mathbf{x}_i^*, \theta_i)) = n_i \ln(\mathbf{p}_i(\mathbf{f}_i(\mathbf{x}_i^*, \theta_i))/n_i) + 2m_i$
        or $MDL_i(\mathbf{f}_i(\mathbf{x}_i^*, \theta_i)) = \frac{1}{2n_i} \mathbf{p}_i(\mathbf{f}_i(\mathbf{x}_i^*, \theta_i)) e^{\frac{\ln(n_i)(m_i-1)}{n_i-m_i-2}}$
    end for
    return $\mathbf{f}_i$ with minimal $AIC_i$ or $MDL_i$
end function
```

Wie man sieht, können die einzelnen $\mathbf{r}_i$ unabhängig voneinander berechnet werden. Hier bietet sich der Einsatz von Parallelverarbeitung an.

2.3.3 Modellidentifikation

Der im Folgenden beschriebene Modellidentifikationsalgorithmus wurde in [20] eingeführt und in [11] praktisch eingesetzt.

Die Modelle $\mathbf{f}_i$ werden als nicht-lineare Gleichungssysteme der Form

$$\mathbf{A}_{\mathbf{f}_i} \mathbf{y}_i = \mathbf{b}_k \tag{2.31}$$

betrachtet. Hierbei ist mit $\mathbf{A}_{\mathbf{f}_i} = \mathbf{A}_{\mathbf{f}_i}(\mathbf{x})$ die Systemmatrix des Modells $\mathbf{f}_i$ in Abhängigkeit von den Eingabeparametern $\mathbf{x} = \Theta$, mit $\mathbf{b}_k \in \mathbf{F}$ wird die Initialbedingung, im Speziellen die an das System angewandten k Kräfte aus $\mathbf{F}$, und mit $\mathbf{y}_i$ die Lösung des Systems beschrieben.

Der Modellidentifikationsalgorithmus in Algorithmus 3 beschreibt drei Schritte zur Lösung des Problems. Als Eingabe dienen die Funktionen $\mathbf{f}$, die Eingabekräfte $\mathbf{F}$ sowie die Eingabefehler δ^f und Messfehler δ^o.

Der erste Schritt, welcher in Algorithmus 4 beschrieben ist, benötigt für die Ausführung keine Messdaten und kann somit a priori durchgeführt werden. Er wird deshalb auch Offline-Schritt genannt. Der Algorithmus berechnet den maximalen minimalen Abstand zweier Modelle für die

Algorithmus 3 Modellidentifikationsschritt nach [20].

```
function ModelIdentification($\mathbf{f}$, $\mathbf{F}$, $\delta^f$,$\delta^o$)
    $\gamma, \mathbf{b}, \nu$=OfflineStep($\mathbf{f}$, $\mathbf{F}$)
    $\mu$=OnlineStep($\mathbf{f}$, $\mathbf{F}$, $\mathbf{b}$, $\delta^f$, $\delta^o$)
    BestModel=SelectionStep($\mathbf{f}$, $\gamma$, $\nu$, $\mu$, $\delta^f$,$\delta^o$)
    return BestModel
end function
```

Eingaben Θ in Abhängigkeit von den Eingaben der Kräfte $\mathbf{b}_k$ und schreibt diesen Wert in eine Matrix $\gamma_{i,j}$. Die zugrundeliegende Kraft wird in einer weiteren Matrix $\mathbf{b}_{i,j}$ gespeichert. Des Weiteren wird die maximale Sensitivität der Modelle bezüglich der Kräfte $\mathbf{b}$ berechnet und dies in $\nu_{i,j}$ gespeichert.

Algorithmus 4 Offline-Schritt des Algorithmus der Modellidentifikation nach [20].

```
function OfflineStep($\mathbf{f}$, $\mathbf{F}$)
    for all $\mathbf{f}_i \in \mathbf{f}$ do
        for all $\mathbf{f}_j \in \mathbf{f}, j > i$ do
            $\gamma_{i,j} = \gamma_{j,i} = \max_{\mathbf{b}\in\mathbf{F}} \min_{\Theta_i,\Theta_j} \|\mathbf{f}_i(\Theta_i, \mathbf{b}) - \mathbf{f}_j(\Theta_j, \mathbf{b})\|$
            $\mathbf{b}_{i,j} = \mathbf{b}_{j,i} = \arg\max_{\mathbf{b}\in\mathbf{F}} \min_{\Theta_i,\Theta_j} \|\mathbf{f}_i(\Theta_i, \mathbf{b}) - \mathbf{f}_j(\Theta_j, \mathbf{b})\|$
            $\nu_{i,j} = \nu_{j,i} = \max_{\mathbf{b}\in\mathbf{F}} \max_{\Theta_i,\Theta_j} \left\{ \|\frac{\partial \mathbf{f}_i(\Theta_i,\mathbf{b})}{\partial \mathbf{b}}\|, \|\frac{\partial \mathbf{f}_j(\Theta_j,\mathbf{b})}{\partial \mathbf{b}}\| \right\}$
        end for
    end for
    return $\gamma$, $\mathbf{b}$, $\nu$
end function
```

Ausgehend von diesen drei berechneten Matrizen wird der sog. Online-Schritt, zu sehen in Algorithmus 5, durchgeführt. Hierfür wird das Experiment $\mathbf{f}^*$ gewählt, welches das echte Modell (engl. True Model) darstellt. Damit werden Messwerte $\Phi_{i,j}$ mit der aus $\gamma_{i,j}$ benutzten Kraft $\mathbf{b}_{i,j}$ berechnet. Hierbei wird die Kraft um einen Fehler $\delta^f_{i,j}$, den sogenannten Eingabefehler gestört. Außerdem wird in diesem Zusammenhang noch ein Messwert-Fehler $\delta^o_{i,j}$ betrachtet, der zu $\Phi_{i,j}$ hinzu addiert wird.

Algorithmus 5 Online-Schritt des Algorithmus der Modellidentifikation nach [20].

```
function OnlineStep($\mathbf{f}$, $\mathbf{F}$, $\mathbf{b}$, $\delta^f$, $\delta^o$)
    for all $\mathbf{f}_i \in \mathbf{f}$ do
        for all $\mathbf{f}_j \in \mathbf{f}, j \neq i$ do
            $\Phi_{i,j} = \mathbf{f}^*(\Theta^*, \mathbf{b}_{i,j} + \delta^f_{i,j}) + \delta^o_{i,j}$
            $\mu_{i,j} = \min_\Theta \|\mathbf{f}_i(\Theta, \mathbf{b}_{i,j}) - \Phi_{i,j}\|$
        end for
    end for
    return $\mu$
end function
```

Der letzte Schritt des Algorithmus ist abschließend die auswahl der Modelle durchzuführen, welche in Algorithmus 6 beschrieben ist.

Algorithmus 6 Auswahl-Schritt des Algorithmus der Modellidentifikation nach [20].

```
function SelectionStep($\mathbf{f}$, $\gamma$, $\nu$, $\mu$, $\delta^f$, $\delta^o$)
    BestModel= $-1$
    for all $\mathbf{f}_i \in \mathbf{f}$ do
        Let $\tau_{i,j} \geq \nu_{i,j} \|\delta^f_{i,j}\| + \|\delta^o_{i,j}\|$
        if $\mu_{i,j} \leq \tau_{i,j}$ and $\mu_{i,j} \geq \gamma_{i,j} - \tau_{i,j} \quad \forall j \neq i$ then
            BestModel= $i$
        end if
    end for
    return BestModel
end function
```

2.4 Space-Mapping

Space-Mapping ist eine Methode, um sehr komplexe und rechenintensive Simulationsmodelle mit Hilfe von einfacheren Modellen zu optimieren. Die Grundlagen für Space-Mapping wurden 1994 von Bandler [21] gelegt. Bakr stieß 1997 zur Forschungsgruppe von Bandler [22] und veröffentlichte 2000 einen Überblick über das Space-Mapping [23]. In einigen wissenschaftlichen Ausarbeitungen wurden Unterarten von Space-Mapping eingesetzt, so zum Beispiel zur Optimierung von Flugzeugtragflächen [24], Strukturoptimierung für Fahrzeugsicherheit [25] und Optimale Kontrolle in PDEs [26].

In dieser Arbeit soll Space-Mapping als Parameter-Schätzproblem eingesetzt werden, bei dem ein einfaches Ersatz-Problem anstelle eines komplexen (feinen) Problems gelöst wird. Das Ersatz-Problem wird meist als grobes Modell bezeichnet. Es zeichnet sich durch eine im Vergleich zum feinen Problem schnelle Auswertbarkeit aus, sodass auch dessen Ableitungen einfach zu erzeugen sind.

Um die Funktion des Space-Mapping für Optimierungsprobleme zu beschreiben, müssen zuerst einige Definitionen getroffen werden. Sei

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} U(\mathbf{R}(\mathbf{x})) \tag{2.32}$$

mit $\mathbf{R} \in \mathbb{R}^m$ ein Vektor mit m Antworten eines Modells, $\mathbf{x} \in \mathbb{R}^n$ ist ein Vektor von n Designparametern und $U \in \mathbb{R}^m \to \mathbb{R}$ eine passende Zielfunktion. Dann ist $\mathbf{x}^*$ die Lösung dieses Optimierungsproblems. Es existiert eine Funktion $\mathbf{R}_f(\mathbf{x}_f)$, welche sehr komplex ist. Im Folgenden wird die Funktion $\mathbf{R}_f(\mathbf{x}_f)$ „feines“ Modell genannt. Des Weiteren existiert eine einfachere Funktion $\mathbf{R}_c(\mathbf{x}_c)$, welche wir im Folgenden „grobes“ Modell nennen. Des weiteren existieren Messwerte $\mathbf{y}$ der zugrundeliegenden Modelle.

Gesucht ist ein Mapping $\mathbf{P}$, welches die Modellparameter des feinen und groben Modells miteinander in Verbindung bringt

$$\mathbf{x}_c = \mathbf{P}(\mathbf{x}_f), \ \mathbf{P} : \mathbb{R}^n \to \mathbb{R}^n \tag{2.33}$$

so dass

$$\mathbf{R}_c(\mathbf{P}(\mathbf{x}_f)) \approx \mathbf{R}_f(\mathbf{x}_f) \tag{2.34}$$

in einer Region of Interest (ROI) gilt. Dann kann auf die direkte Lösung von (2.32), um $\mathbf{x}_f$ zu lösen, verzichtet werden, indem ein $\overline{\mathbf{x}}_f$ mit den Eigenschaften

$$\overline{\mathbf{x}}_f^* = \mathbf{P}^{-1}(\mathbf{x}_c^*) \approx \mathbf{x}_f^* \tag{2.35}$$

definiert wird. Dies ist eine gute Approximation von $\mathbf{x}_f^*$, wobei $\mathbf{x}_c^*$ das Resultat der Optimierung des groben Modells ist.

Um dieses Problem zu lösen, betrachten wir die Jacobi-Matrix $\mathbf{J}_P$ von $\mathbf{P}$. Diese ist definiert an der Stelle $\mathbf{x}_f$ als

$$\mathbf{J}_P(\mathbf{x}_f) = \left(\frac{\partial \mathbf{P}^T}{\partial \mathbf{x}_f}\right)^T = \left(\frac{\partial (\mathbf{x}_c^T)}{\partial \mathbf{x}_f}\right)^T. \tag{2.36}$$

Diese Ableitungsmatrix $\mathbf{J}_P$ wollen wir nun durch die Matrix $\mathbf{B}$ approximieren. Seien $\mathbf{J}_f = \frac{\partial \mathbf{R}_f(\mathbf{x}_f)}{\partial \mathbf{x}_f}$ und $\mathbf{J}_c = \frac{\partial \mathbf{R}_c(\mathbf{x}_c)}{\partial \mathbf{x}_c}$ die Ableitungsmatrizen des feinen und groben Modells. Dann gilt wegen Formel (2.34)

$$\mathbf{J}_f \approx \mathbf{J}_c \mathbf{B}. \tag{2.37}$$

Eine Herausforderung ist es nun, die Matrix $\mathbf{B}$ zu bestimmen.

Wenn die Jacobi-Matrizen für das feine und grobe Modell existieren, kann $\mathbf{B}$ wie folgt berechnet werden

$$\mathbf{B} = (\mathbf{J}_c^T \mathbf{J}_c)^{-1} \mathbf{J}_c^T \mathbf{J}_f. \tag{2.38}$$

Dabei ist zu beachten, dass $\mathbf{J}_c$ vollen Rang hat und es gilt $n \leq m$.

Weiterhin definieren wir den Residuumvektor $\mathbf{f}$, mit

$$\mathbf{f} = \mathbf{f}(\mathbf{x}_f, \mathbf{x}_c^*) = \mathbf{P}(\mathbf{x}_f) - \mathbf{x}_c^*. \tag{2.39}$$

In dieser Arbeit werden zwei der Ansätze von Bandler und Bakr vorgestellt. Das Aggressive-Space-Mapping und das Trust-Region Aggressive-Space-Mapping. Diese unterscheiden sich darin, wie der Residuumsvektor $\mathbf{f}$ aus (2.39) bestimmt werden kann.

2.4.1 Aggressive-Space-Mapping

Das Aggressive-Space-Mapping (ASM) versucht, das Gleichungssystem

$$\mathbf{f}(\mathbf{x}_f, \mathbf{x}_c^*) = 0 \tag{2.40}$$

iterativ zu lösen.

In der j-ten Iteration muss für den Residuumvektor $\mathbf{f}^{(j)}$ die Funktion $\mathbf{P}^{(j)}(\mathbf{x}_f^{(j)})$ berechnet werden. Der Quasi-Newton Schritt des feinen Modells wird definiert durch

$$\mathbf{B}^{(j)} \mathbf{h}^{(j)} = -\mathbf{f}^{(j)} \tag{2.41}$$

wobei $\mathbf{B}^{(j)}$ mittels Broyden-Rang-Eins-Update aktualisiert wird. Als Startwerte für die Iteration werden $\mathbf{B}^{(0)} = I$ und $\mathbf{f}^{(0)}(\mathbf{x}_f^{(0)}, \mathbf{x}_c^*)$ mit $\mathbf{x}_f^{(0)} = \mathbf{x}_0$ gesetzt. Dabei ist $\mathbf{x}_0$ ein vorgegebener Initialvektor.

Wenn Formel (2.41) nach $\mathbf{h}^{(j)}$ gelöst wird, kann man den nächsten iterativen Schritt für $\mathbf{x}_f^{(j+1)}$ durch

$$\mathbf{x}_f^{(j+1)} = \mathbf{x}_f^{(j)} + \mathbf{h}^{(j)} \tag{2.42}$$

bestimmen. Für den Vektor $\mathbf{P}^{(j+1)}$, aus dem wir $\mathbf{B}^{(j+1)}$ berechnen, gilt weiterhin

$$\mathbf{P}^{(j+1)} = \arg\min_{\mathbf{x}_c} \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(\mathbf{x}_f^{(j+1)})\|. \tag{2.43}$$

Es wird also der Abstand zwischen $\mathbf{R}_c$ und $\mathbf{R}_f$ an der Stelle $\mathbf{x}_f^{(j+1)}$ minimiert.

Der nächste Iterationsschritt der Matrix $\mathbf{B}^{(j+1)}$ ist gegeben durch

$$\mathbf{B}^{(j+1)} = \mathbf{B}^{(j)} + \frac{(\mathbf{P}^{(j+1)} - \mathbf{x}_c^*){\mathbf{h}^{(j)}}^T}{{\mathbf{h}^{(j)}}^T \mathbf{h}^{(j)}}. \tag{2.44}$$

Somit können wir nun den Algorithmus 7 beschreiben, welcher das Aggressive-Space-Mapping löst. Zuerst müssen die Startwerte für $\mathbf{x}_c^*$ und $\mathbf{P}^{(0)}$ berechnet werden. Dies geschieht mittels des Algorithmus 8. Nun kann in jeder Iteration (bis der Abstand $\|\mathbf{P}^{(k)} - \mathbf{x}_c^*\|$ klein genug ist) die Inverse der Matrix $\mathbf{B}$ für die Berechnung von $\mathbf{h}^{(j)}$ genutzt werden. Damit kann nun $\mathbf{x}_f^{(j+1)}$ nach (2.42) berechnet werden. Nun wird noch der Vektor $\mathbf{P}^{(j+1)}$ berechnet und dies für das neue $\mathbf{B}^{(j+1)}$ genutzt.

Algorithmus 7 Aggressive-Space-Mapping Algorithmus.

```
function ASM($\mathbf{x}_0$, $\mathbf{y}$, $maxIter$)
    $\mathbf{x}_c^*$, $\mathbf{P}^{(0)}$ = INIT($\mathbf{x}_0$,$\mathbf{y}$)
    $\mathbf{x}_f^{(0)} = \mathbf{x}_0$
    $\mathbf{B}^{(0)} = I$
    $j = 0$
    while $\|\mathbf{P}^{(j)} - \mathbf{x}_c^*\| > \epsilon \wedge j < maxIter$ do
        $\hat{h} = {\mathbf{B}^{(j)}}^{-1}$
        $\mathbf{h}^{(j)} = -\hat{h} \cdot (\mathbf{P}^{(j)} - \mathbf{x}_c^*)$
        $\mathbf{x}_f^{(j+1)} = \mathbf{x}_f^{(j)} + \mathbf{h}^{(j)}$
        Solve $\mathbf{P}^{(j+1)} = \arg\min_{\mathbf{x}_c} \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(\mathbf{x}_f^{(j+1)})\|$ iteratively, start with $\mathbf{x}_f^{(j+1)}$
        $\mathbf{B}^{(j+1)} = \mathbf{B}^{(j)} + \frac{(\mathbf{P}^{(j+1)} - \mathbf{x}_c^*){\mathbf{h}^{(j)}}^T}{{\mathbf{h}^{(j)}}^T \mathbf{h}^{(j)}}$
        $j = j + 1$
    end while
end function
```

Algorithmus 8 Initialisierungsfunktion der Space-Mapping Algorithmen.

```
function INIT($\mathbf{x}_0$, $\mathbf{y}$)
    Solve $\mathbf{x}_c^* = \arg\min_{\mathbf{x}_c} \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{y}\|$ iteratively, start with $\mathbf{x}_0$
    Solve $\mathbf{P}^{(0)} = \arg\min_{\mathbf{x}_c} \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(\mathbf{x}_0)\|$ iteratively, start with $\mathbf{x}_0$
    return $\mathbf{x}_c^*$, $\mathbf{P}^{(0)}$
end function
```

2.4.2 Trust-Region Aggressive-Space-Mapping

Die hier beschriebene Methode kombiniert einen Trust-Region Ansatz mit der ASM Methode. Der Trust-Region Ansatz, welcher in [27] im Detail beschrieben wird, soll die Lösung des Optimierungsproblems dergestalt verbessern, dass die Lösung bei einem beliebigen Startpunkt immer in einem lokalen Minimum konvergiert. Dabei soll der Schritt $\mathbf{h}^{(j)}$ aus Formel (2.42) so gewählt werden, dass die Zielfunktionsauswertung der nächsten Iteration nur innerhalb eines bestimmten Radius $\delta \in \mathbb{R}_+$, in welchem dem groben Modell vertraut wird, durchgeführt wird. Dieser Vertrauens-Radius wird Trust-Region genannt.

Der größte Unterschied zum klassischen Aggressive-Space-Mapping besteht in der Behandlung des Updates von $\mathbf{B}$. In der j-ten Iteration wird Formel (2.41) erweitert zu

$$\left(\mathbf{B}^{(j)^T}\mathbf{B}^{(j)} + \lambda I\right)\mathbf{h}^{(j)} = -\mathbf{B}^{(j)^T}\mathbf{f}^{(j)}. \tag{2.45}$$

Dabei ist der Parameter λ so zu wählen, dass $\|\mathbf{h}^{(j)}\| \leq \delta$ gilt. Der Parameter δ ist dabei die Größe der Trust-Region. Eine besondere Herausforderung ist dabei die geschickte Wahl des Parameters λ. Eine mögliche Herangehensweise ist in Algorithmus 9 gezeigt. Hierbei wird der maximale Eigenwert der Matrix $\mathbf{B}^T\mathbf{B}$, gedämpft um einen Faktor ι zum aktuellen λ hinzuaddiert.

Algorithmus 9 Updatefunktion für λ

```
function UPDATELAMBDA($\lambda$, $\mathbf{B}$, $\iota$)
    $\lambda_n = \lambda + \iota \max \operatorname{eig}(\mathbf{B}^T\mathbf{B})$
    return $\lambda_n$
end function
```

Davon ausgehend ergibt sich der Algorithmus 10, eine um Trust-Region erweiterte Version des ASM-Algorithmus.

Algorithmus 10 Trust-Region Aggressive-Space-Mapping Algorithmus.

```
function TRASM($\mathbf{x}_0$, $\mathbf{y}$, $maxIter$, $\lambda_0$, $\iota$)
    $\mathbf{x}_c^*$, $\mathbf{P}^{(0)}$ = INIT($\mathbf{x}_0$,$\mathbf{y}$)
    $\mathbf{x}^{(0)} = \mathbf{x}_0$
    $\mathbf{B}^{(0)} = I$
    $j = 0$
    $\lambda^{(j)} = \lambda_0$
    while $\|\mathbf{P}^{(j)} - \mathbf{x}_c^*\| > \epsilon \wedge j < maxIter$ do
        $\hat{h} = (\mathbf{B}^{(j)^T}\mathbf{B}^{(j)} + \lambda^{(j)} I)^{-1}\mathbf{B}^{(j)^T}$
        $\mathbf{h}^{(j)} = -\hat{h}(\mathbf{P}^{(j)} - \mathbf{x}_c^*)$
        $\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)} + \mathbf{h}^{(j)}$
        $\mathbf{P}^{(j+1)} = \arg\min_{\mathbf{x}_c} \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(x^{(j+1)})\|$
        $\mathbf{B}^{(j+1)} = \mathbf{B}^{(j)} + \frac{(\mathbf{P}^{(j+1)} - \mathbf{x}_c^*)\mathbf{h}^{(j)^T}}{\mathbf{h}^{(j)^T}\mathbf{h}^{(j)}}$
        $\lambda^{(j+1)}$ =UPDATELAMBDA($\lambda^{(j)}$, $\mathbf{B}$, $\iota$)
        $j = j + 1$
    end while
end function
```

Kapitel 3

Enabling Technologies

Um die im vorhergehenden Kapitel beschriebenen Optimierungsprobleme zu lösen, werden verschiedene Building-Blocks benötigt, welche in diesem Kapitel betrachtet werden sollen.

Zuerst wird das Vorwärtsmodell $\mathbf{f}(\mathbf{x}, \theta)$ mit Hilfe einer geeigneten Programmiersprache in ein Computerprogramm übersetzt. Dieser Prozess ist sehr zeitaufwändig und wird meist von – oder zumindest unter Mithilfe von – Wissenschaftlern aus dem entsprechenden Bereich durchgeführt. Dabei werden jedoch meist die Werkzeuge der Informatik, wie Softwaretechnik, Parallelverarbeitung oder ähnliches verwendet. Diese Modelle haben wir uns bereits in Kapitel 2 angesehen.

Die verwendeten Optimierungsalgorithmen definieren einen weiteren Building-Block. Ihre Fähigkeit beeinflusst sowohl die Konvergenzgeschwindigkeit des Optimierungsproblems, als auch dessen Ergebnis. Um die Arbeit mit den hier untersuchten Problemstellungen einfach zu gestalten, sie insbesondere für informatikfremde Gebiete beherrschbar zu machen, wird in dieser Arbeit Python mit den mathematischen Paketen Numpy und Scipy verwendet.

Ein weiterer Building-Block sind die benötigten Ableitungen der Simulation. Hierbei werden meist die ersten Ableitungen benötigt. Diese können mittels analytischer Verfahren, finiter Differenzen oder mit automatischem Differenzieren erzeugt werden.

Der letzte hier betrachtete Building-Block ist die Parallelverarbeitung von Daten. Mit zunehmender Größe der betrachteten Diskretisierungsfelder der physikalischen Probleme nimmt der Rechenaufwand und damit die benötigt Rechenzeit zu. Um dies zu kompensieren, finden zwei wesentliche Parallelisierungsparadigmen Anwendung, die sich in den jeweiligen Zugriffen auf den verwendeten Speicher unterscheiden: Shared-Memory Parallelverarbeitung und Distributed-Memory Parallelverarbeitung.

3.1 Optimierungsalgorithmen

Als zweiter Building-Block, welcher auf den Vorwärtssimulationen aufbaut, sind die Optimierungsalgorithmen zu nennen. Jede der im Kapitel 2 beschriebenen inversen Problemstellungen lässt sich auf ein Minimierungs- (oder Maximierungs-) Problem zurückführen. Dabei sind verschiedene Optimierungsalgorithmen in der Theorie beschrieben [7, 14, 13, 28], welche jeweils Vor- und Nachteile besitzen. Im weiteren Verlauf dieses Kapitels werden Grundlagen der Optimierungsalgorithmen basierend auf dem Buch von Nocedal und Wright [7] wiedergegeben.

3.1.1 Definitionen

Das Problem

$$\min_{\mathbf{x}} \mathbf{f}(\mathbf{x})$$

heißt Minimierungsproblem mit der Zielfunktion $\mathbf{f} : \mathbf{V} \to \mathbb{R}^m$. Für $m = 1$ spricht man von skalaren Optimierungsproblemen, sonst von Vektoroptimierungsproblemen.

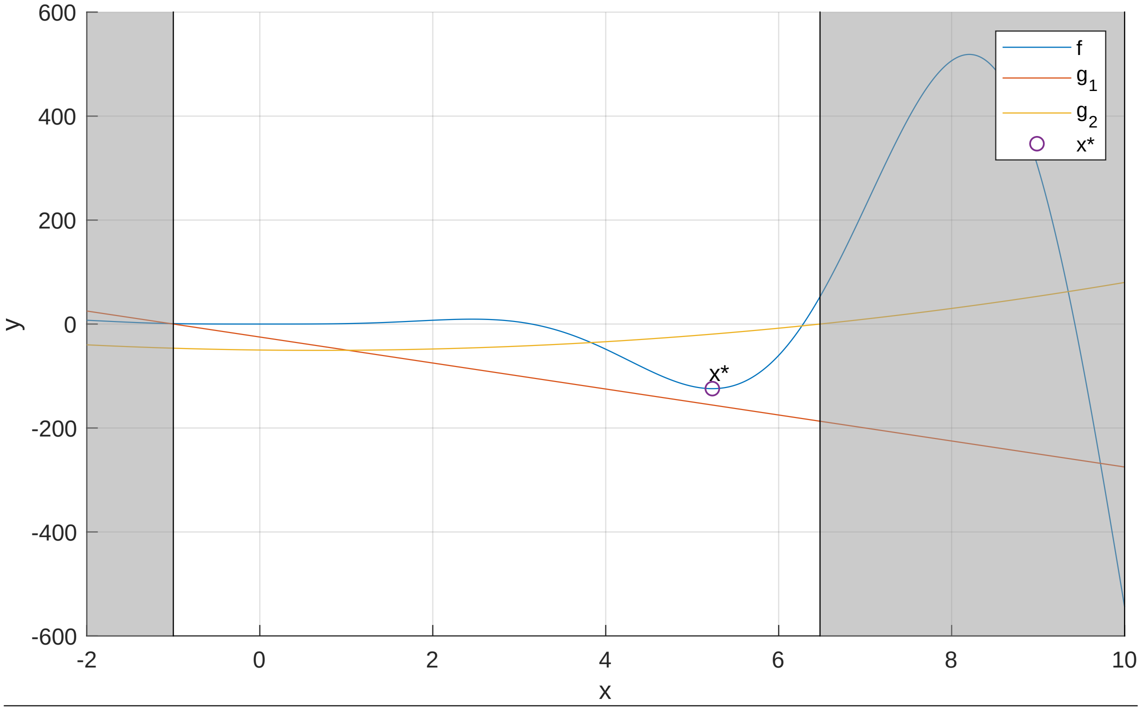
Des Weiteren kann $\mathbf{x}$ durch sogenannte Nebenbedingungen eingeschränkt sein. Das heißt $\mathbf{x} \in \mathbf{X}$, mit $\mathbf{X} \subseteq \mathbf{V}$, welches für gewöhnlich durch $\mathbf{X} = \{\mathbf{x} \in \mathbf{V} : \mathbf{g}(\mathbf{x}) \leq 0\}$ beschrieben wird.

Dabei sind die Nebenbedingungen weiter in lineare ($\mathbf{g}(\mathbf{x})$ ist linear) und nichtlineare ($\mathbf{g}(\mathbf{x})$ ist nichtlinear) Gleichungen zu unterscheiden.

In Abb. 3.1 ist ein einfaches Beispiel eines beschränkten Optimierungsproblems gegeben. Die zugrundeliegende Zielfunktion $\mathbf{f}$ ist gegeben durch

$$f(x) = \sin(x) \cdot x^3. \tag{3.1}$$

Abbildung 3.1 Beispiel eines beschränkten Optimierungsproblems $\min f(x)$ mit $g_1(x) \leq 0$ und $g_2(x) \leq 0$ (schwarze vertikale Linien).



Als lineare Nebenbedingung wurde

$$g_1(x) = -25x - 25 \tag{3.2}$$

gewählt, während als nichtlineare Nebenbedingung

$$g_2(x) = 1.5x^2 - 2x - 50 \tag{3.3}$$

gilt. Die Lösung dieses Optimierungsproblems ist bei $x^* = 5,2329$ und $f(x^*) = -124,3167$ gegeben.

Um eine Lösung des Minimierungsproblems zu finden, nutzen wir iterative Optimieralgorithmen, wobei die Lösung mittels der Iteration

$$x_{k+1} = x_k + \alpha_k p_k, \tag{3.4}$$

angenähert wird. Hierbei ist p_k eine Suchrichtung und α_k eine Schrittweite, welche mittels einer sogenannten Linien-Suche (engl. Line-Search) für den Schritt k auf $k+1$ bestimmt wird. Der Schritt basiert meist auf einer gedämpften Variante des Gradienten des Zielfunktionals

$$p_k = -B_k^{-1} \nabla f_k. \tag{3.5}$$

Je nach benutzter Optimierungs-Methode wird die Matrix B_k unterschiedlich gesetzt:

- Steilster Abstieg: Identität $B_k = I$
- Newton: Hessematrix $B_k = \nabla^2 f(x_k)$
- Quasi-Newton: approximierte Hessematrix

Die Schrittweite $\alpha_k \in \mathbb{R}_+$ wird so gewählt, dass eine substanzielle Reduktion der Zielfunktion eintritt. Die Berechnung soll dabei nicht zu zeitintensiv sein, also der Richtung möglichst weit folgen können, ohne die Zielfunktion und deren Ableitungen zu oft auszuwerten. Typischerweise nutzen Line-Search Algorithmen einen ausgeklügelten zweiteiligen Mechanismus, um diese zu bestimmen. Zuerst wird in der Bracketing-Phase ein Intervall bestimmt, in dem mögliche Kandidaten der Schrittweite enthalten sind. Dann folgt die Bisektions- oder Interpolations-Phase, um innerhalb des Intervalls einen guten Kandidaten auszuwählen.

Die meisten Linien-Such-Algorithmen verlangen, dass für die Schrittweite α_k die Richtung p_k eine Abstiegsrichtung ist, das heißt $p_k^T \nabla f_k < 0$. Dies wird im Fall des steilsten Abstiegs erfüllt. Andernfalls muss dafür Sorge getragen werden, dass die Matrix B_k positiv definit ist, zum Beispiel im Quasi-Newton Fall durch geeignete Niedrig-Rang-Approximationen [7].

3.1.2 Klassifizierung

Je nach Art der verwendeten Zielfunktion und der möglicherweise vorhandenen Nebenbedingungen lassen sich Optimierungsalgorithmen in verschiedene Klassen einteilen. Des Weiteren muss man zwischen lokaler und globaler Optimierung unterscheiden.

- Lineare Programmierung
- Nicht-Lineare Programmierung
- Quadratische Optimierung
- Kleinste-Quadrate
- Konjugierte Gradienten

Nach Nocedal und Wright [7] können die folgenden Eigenschaften für einen Optimierer bestimmt werden:

- Robustheit: Wie konvergiert der Algorithmus bezüglich der Eingaben des Startpunktes?
- Effizienz: Wie viele Evaluierungsschritte sind nötig, bis ein (optimales) Ergebnis vorliegt?
- Genauigkeit: Wie genau kann die Lösung gefunden werden?

Je nach ihrer Klasse eignen sich Optimierungsalgorithmen damit mehr oder weniger für bestimmte Probleme.

3.1.3 Lokale und Globale Optimierung

Ein Optimierungsalgorithmus ist ein iterativer Prozess, in welchem ausgehend von den bisherigen Ergebnissen eine neue Suchposition ermittelt wird. Zu diesen Ergebnissen gehören vorhergehende Funktionsauswertungen sowie die erste oder höhere Ableitungen. Dabei wird meist durch eine sogenannte Schrittweite die Suchposition verändert. Sollte der Optimierungsalgorithmus nahe bei einem Minimum sein, so wird dieser innerhalb einer ϵ-Umgebung keine bessere Lösung finden und das Ergebnis entsprechend ausgeben. Nun muss zwischen lokaler und globaler Optimierung entschieden werden. Bei lokaler Optimierung, welche meist deutlich schneller zu einem Ergebnis kommt, wird wie der Name schon sagt, nur eine lokale Lösung des Problems ermittelt. Diese lokale Lösung muss aber nicht zwangsweise die globale Optimierungslösung darstellen. Der Startpunkt der lokalen Optimierung ist hier von entscheidender Bedeutung. Ist dieser schlecht gewählt, kann der Algorithmus kein besseres Ergebnis finden.

3.1.4 Beispiele

In dieser Arbeit wurden die folgenden Optimierungsalgorithmen implementiert und untersucht.

- Nelder-Mead
- Konjugierte Gradienten (CG)
- Newton-CG
- BFGS
- L-BFGS-B
- PORT
- COBYLA
- ELSUNC
- ENLSIP

3.2 Numpy und Scipy

Da in dieser Arbeit die Programmiersprache Python vornehmlich Verwendung findet, soll an dieser Stelle ein kurzer Überblick über die in Scientific Python (Scipy) [29] gebotenen Funktionen gegeben werden. Scipy ist eng mit Numerical Python (Numpy) verknüpft, welches die Grundlage für mathematische Algorithmen in Python liefert. In Numpy werden Array-Datentypen für Vektoren und Matrizen und Operationen mit diesen definiert. Die Grundlage liefert das sogenannte `ndarray`. Es ist ein mehrdimensionales Array, welches einen bestimmten Datentypen haben kann. Dabei hat der Datentyp die folgenden Attribute

- `ndarray.ndim` – Anzahl der Dimensionen
- `ndarray.shape` – Form des Arrays
- `ndarray.size` – Anzahl der Elemente
- `ndarray.dtype` – Verwendeter Datentyp
- `ndarray.data` – Zugrunde liegender Speicherbereich

Für eine genauere Beschreibung sei hier auf die Dokumentation von Numpy [30] verwiesen.

Scipy bietet verschiedene Module für wissenschaftliches Arbeiten. Die Module, welche Scipy liefert sind

- scipy.optimize – Optimierung von Funktionen
- scipy.linalg – Lineare Algebra
- scipy.integrate – Integration
- scipy.fftpack – Schnelle Fourier Transformation
- scipy.signal – Signalverarbeitungsroutinen

In dieser Arbeit wird das `scipy.optimize` Modul verwendet, um eine komfortable Anbindung an viele verschiedene Optimierer in Python zu erhalten. Um eine skalare Funktion zu minimieren bietet das Modul die Methode `minimize(fun,x0[,args,method,jac,hess])` Dazu muss eine Funktion `fun`, sowie ein Startvektor `x0` angegeben werden. Für unsere Anwendung sind weiterhin die verwendete Optimierermethode `method` und die Jacobi-Matrix `jac` interessant. In Kapitel 4.4.5 wird auf die Verwendung der Scipy-Optimerer genauer eingegangen.

3.3 Ableitungen

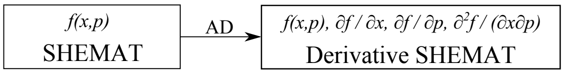
Ausgehend von den im vorhergehenden Abschnitt besprochenen Simulationen und Optimierern müssen wir nun die Frage nach den Ableitungen der Simulationen genauer betrachten. Diese Ableitungen beeinflussen maßgeblich die Güte und die Konvergenzgeschwindigkeit des Optimierungsalgorithmus. Das heißt mit guten Ableitungen kann ein Optimierer viel schneller oder auch noch genauer ein Optimum finden. Dabei haben sich drei grundverschiedene Ansätze etabliert.

Wenn die zugrundeliegende Funktion sehr einfach ist, bietet es sich an, diese analytisch per Hand abzuleiten. Dabei werden die verschiedenen Ableitungsregeln auf die Funktion angewandt, bis das Ergebnis die Ableitung repräsentiert.

Eine weitere Möglichkeit der Ableitungsberechnung ist, finite Differenzen zu nutzen. Dabei werden um den gesuchten Punkt der Funktion die Funktionswerte leicht um einen (meist konstanten) Wert h gestört und die Differenz zum ursprünglichen Funktionswert genommen und durch h dividiert. Dabei ist es entscheidend, welchen Wert h annimmt. Denn ein zu großer Wert verfälscht das Ergebnis und ein zu kleiner Wert bringt Probleme mit der Numerik, da die Maschinengenauigkeit unterschritten werden kann.

Neben analytischen und finiten Ableitungen findet sich das automatische, oder algorithmische, Differenzieren (AD). Hierbei wird ein gegebener Programmcode mithilfe von einfachen Regeln automatisch abgeleitet, ohne dabei Rundungsfehler zu generieren. Weiterführende Informationen finden sich in [31, 32]. Der grundlegende Transformationsschritt ist in Abb. 3.2 zu sehen.

Abbildung 3.2 Prinzip des Automatischen Differenzierens.



3.3.1 Analytische Ableitungen

Wenn die abzuleitende Funktion einfach genug ist, können analytische Ableitungen benutzt werden. Ein einfaches Beispiel ist hier die Funktion $f(x_1, x_2) = (x_1 - x_2)^2$. Die partiellen Ableitungen sind nach Anwendung der Kettenregel $\frac{\partial f}{\partial x_1} = 2(x_1 - x_2)$ sowie $\frac{\partial f}{\partial x_2} = -2(x_1 - x_2)$. Komplizierte Funktionen lassen sich mit entsprechendem Aufwand auch so händisch ableiten. Die Gefahr von Fehlern nimmt jedoch stark zu.

Da die zugrundeliegenden Regeln jedoch bekannt sind, können Computerprogramme Ableitungen automatisch erzeugen, sofern die entsprechende Funktion vorliegt. Als Technik hat sich das symbolische Ableiten etabliert, welches in vielen Mathematiksoftwaresystemen wie Mathematica oder Matlab implementiert ist.

Sollte die Funktion jedoch nicht so einfach zu beschreiben sein oder die Berechnungsvorschrift nur anhand eines Computerprogramms vorliegen, so kann das symbolische Differenzieren keine Anwendung finden. Hier haben sich zwei Techniken etabliert, die bei sehr komplexen Anwendungen genutzt werden, die finiten Differenzen und das automatische Differenzieren.

3.3.2 Finite Differenzen

Wie zu Beginn erwähnt, werden finite Differenzen (FD) häufig eingesetzt, da sie einfach zu bestimmen sind und meist als hinreichend genau anerkannt werden. Dabei werden die Inputparameter einer Funktion minimal gestört, um die Ableitung an dem Punkt der Inputparameter $\mathbf{x} = (x_1, \ldots, x_n)$ zu bestimmen und so die Richtungsableitung zu erhalten.

Dabei unterscheidet man drei verschiedene Varianten der verwendeten Differenz:

- Die Vorwärtsdifferenz
$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}. \tag{3.6}$$
- Die Rückwärtsdifferenz
$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x}) - f(\mathbf{x} - h\mathbf{e}_i)}{h}. \tag{3.7}$$
- Sowie die Zentraldifferenz
$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + \frac{h}{2}\mathbf{e}_i) - f(\mathbf{x} - \frac{h}{2}\mathbf{e}_i)}{h}. \tag{3.8}$$

Da der Parameter h nicht beliebig klein gewählt werden kann und somit als feste Zahl angenommen wird, kann im folgenden die Ableitung approximativ bestimmt werden durch (für die Vorwärtsdifferenz, andere analog)
$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}.$$
Dabei können die Fehler bei den einzelnen Differenzen aufgestellt werden. Bei der Vorwärts- bzw. Rückwärtsdifferenz ist der Fehler
$$\frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h} - \frac{\partial f(\mathbf{x})}{\partial x_i} = O(h),$$
falls die Funktion zweifach differenzierbar ist. Die Zentraldifferenz hat hingegen einen Fehler von $O(h^2)$, wenn die Funktion zusätzlich dreifach differenzierbar ist [33].

3.3.3 Automatisches Differenzieren

Eine weitere Technik zur Berechnung von Ableitungen ist das sogenannte automatische Differenzieren, kurz AD. In einigen Publikationen wird auch vom algorithmischen Differenzieren gesprochen, da die Ableitungsgenerierung auf einem Algorithmus basiert. Einer der Vorteile gegenüber finiten Differenzen ist dabei in der Genauigkeit der Berechnungen zu sehen, da die Berechnungen exakt (im Sinne der Maschinengenauigkeit) durchgeführt werden und keine Approximationen der Ableitungen erzeugt werden. Ein weiterer Vorteil ist, dass für die Berechnung der Ableitung ein konstanter Overhead c bei der Berechnung benötigt wird. Die Zeit ist also $O(c \cdot T_f(n))$, wobei $T_f(n)$ die Zeitkomplexität der Funktion f mit n Eingaben beschreibt. Anders bei finiten Differenzen, die $n + 1$ mal die Funktion f ausführen müssen um den Gradienten zu berechnen.

Gegeben sei eine Funktion $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{z})$ mit $\mathbf{f} : \mathbb{R}^{n+o} \to \mathbb{R}^m$ gegeben. Gesucht ist nur die Ableitung nach Parameter dem $\mathbf{x} \in \mathbb{R}^n$, $J := \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. In diesem Fall wird $\mathbf{x}$ als unabhängige Input und $\mathbf{y}$ als abhängige Output bezeichnet.

Beim automatischen Differenzieren wird zwischen dem Vorwärts- und dem Rückwärtsmodus (Forward and Reverse Mode) unterscheiden, welche jeweils unterschiedliche Operationen ausführen, um entweder das Jacobi-Vektorprodukt Js oder das adjungierte Produkt $J^T t$, mit entsprechenden Vektoren $s \in \mathbb{R}^n$ und $t \in \mathbb{R}^m$, zu erhalten.

AD generiert aus einem bestehenden Programm für die Funktion f ein neues Programm, welches die Ableitungen berechnen kann. Das differenzierte Programm hat als Eingaben die eigentlichen Ein- und Ausgaben der Ursprungsfunktion, sowie zusätzlich neue Ein- und Ausgaben für die Ableitungsobjekte. Im einfachsten Fall haben die Ableitungsobjekte die Dimensionalität ihrer zugeordneten Ursprungsobjekte. Dieser Modus heißt Skalarmodus. Zusätzlich können mehrere Ableitungsrichtungen mit einem Funktionsaufruf erzeugt werden. Dieser Modus wird je nach verwendetem Sprachgebrauch als Vektor- oder Multi-direktionaler-Modus bezeichnet.

Ein Funktionsinterface für f ist in Abb. 3.1 als Fortrancode zu sehen. Nachdem ein Tool das automatische Differenzieren auf den Programmcode angewendet hat, entsteht ein neues Programm, welches die gleiche Funktionalität wie das ursprüngliche Programm aufweist, aber zusätzlich die Ableitungen berechnen kann.

Quellcode 3.1: Signatur einer abzuleitenden Fortran-Funktion.

```
subroutine f(x,z,y)
```

In Abb. 3.2 ist das Interface der Ableitung mit automatischem Differenzieren wieder in Fortran gegeben. Ersichtlich ist, dass zusätzlich zu den ursprünglichen Parametern die neuen Parameter `g_x` und `g_y` auftauchen, diese sind die Ableitungsobjekte für `x` und `y`.

Quellcode 3.2: Signatur der abgeleiteten Fortran-Funktion aus 3.1 im Vorwärtsmodus.

```
subroutine g_f(x,g_x,z,y,g_y)
```

Im skalaren Modus haben die Ableitungsobjekte die Form `g_x`$\in \mathbb{R}^n$ und `g_y`$\in \mathbb{R}^m$. Diese müssen beim Aufruf der Funktion richtig gesetzt werden, um die Ableitung in einer Variablen zu erhalten. Um die komplette Jacobi-Matrix $J \in \mathbb{R}^{n\times m}$ zu erhalten, muss man den Code im skalaren Vorwärts-Modus n mal ausführen, im skalaren Rückwärts-Modus m mal.

Im Vektor-Modus haben die entsprechenden Objekte die Größe `g_x`$\in \mathbb{R}^{p\times n}$ und `g_y`$\in \mathbb{R}^{p\times m}$ mit $p \in \mathbb{N}^+$ und $p \leq n$ für den Vorwärts-Modus, also wie viele Ableitungen im Input-Raum untersucht werden sollen und $p \leq m$ für den Rückwärts-Modus, also die Anzahl der Ableitungen im Output-Raum.

3.3.3.1 Forward- und Reverse Mode

Der Vorwärtsmodus (engl. Forward-Mode) des automatischen Differenzierens leitet das Programm „von vorn nach hinten“ ab. Das heißt jede Anweisung wird nach der Kettenregel abgeleitet und direkt bei der eigentlichen Anweisung ausgeführt. Der Vorwärtsmodus berechnet die Jacobi-Matrix, multipliziert mit der sogenannten Seed-Matrix S der Größe $n \times p$. In Quellcode 3.2 ist dies entsprechend:

$$\mathrm{g\_y} = J \cdot S = J \cdot \mathrm{g\_x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \cdot \begin{pmatrix} s_{1,1} & \cdots & s_{1,p} \\ \vdots & \ddots & \vdots \\ s_{n,1} & \cdots & s_{n,p} \end{pmatrix} \tag{3.9}$$

an der Stelle x. Die Ausführungszeit hängt ganz entscheidend von der Anzahl der Richtungen p ab, $T_{\mathrm{FW}}(n,m,p) = O(p \cdot T_f(n,m))$. Ist $S = I$ die Einheitsmatrix der Dimension $n \times n$, so berechnet der Vorwärtsmodus die komplette Jacobi-Matrix J.

Dies kann auch mit Hilfe des Rückwärtsmodus und der Einheitsmatrix der Größe $m \times m$ geschehen. Im allgemeinen berechnet der vektorwärtige Rückwärtmodus die Transponierte der Jacobi-

Tabelle 3.1 Überblick der unterstützten Sprachen verschiedener AD-Sourcecodetransformationstools.

Tool	Fortran	C	Matlab
Tapenade	ja	ja	nein
TAF	ja	nein	nein
AdiFor2	ja	nein	nein
AdiFor3	ja	nein	nein
Adi-C	nein	ja	nein
AdiMat	nein	nein	ja

Matrix multipliziert mit einer anderen Seed-Matrix S' der Göße $m \times q$:

$$\text{g\_x} = J^T \cdot S' = J^T \cdot \text{g\_y} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \cdot \begin{pmatrix} t_{1,1} & \cdots & t_{1,q} \\ \vdots & \ddots & \vdots \\ t_{m,1} & \cdots & t_{m,q} \end{pmatrix} \tag{3.10}$$

Dabei wird der Programmcode einmal komplett vorwärts ausgeführt und an bestimmten Stellen Zwischenergebnisse gespeichert (sog. Snapshots). Die Ableitungsinformationen werden im Anschluss entsprechend der Code-Ausführung in umgekehrter Reihenfolge berechnet, wobei die erstellten Snapshots genutzt werden. Allen Methoden ist gemein, dass verschiedene Treiberroutinen für die Berechnung der Ableitungen nötig sind.

Für bestimmte Probleme kann die Größe der Seed-Matrix, also die Anzahl der Richtungsableitungen reduziert werden, womit die Ausführungszeit deutlich verringert werden kann. Insbesondere für dünnbesetzte Probleme ist es oft möglich, durch das Lösen eines Färbungsproblems geeignete Richtungen zu finden, um die komplette Jacobi-Matrix mit einer weitaus geringeren Anzahl von Richtungen auszuwerten. Weitere Information zu diesem Thema finden sich zum Beispiel in [34].

3.3.3.2 Sourcecodetransformation

Sourcecodetransformation ist die klassische Form der AD-Tools. Ein Programmcode wird mit Hilfe des AD-Tools in einen neuen Programmcode transformiert, welcher die geforderten Ableitungen berechnen kann. Je nach Programmiersprache finden sich verschiedene Transformations-Programme. An dieser Stelle soll eine kleine unvollständige Übersicht für verschiedene Programmiersprachen erstellt werden.

Dabei werden die verschiedenen Programmiersprachen jedoch nur zu einem Teil der Gesamtmächtigkeit unterstützt. Dies hat teilweise pragmatische Gründe. So kann ein Tool für C nur mit Basistypen umgehen, wohingegen der Umgang mit komplexen Pointerstrukturen sehr schwierig ist und viele Tools diese als Ableitungsdatentypen ausschließen.

Weiterhin zeigt sich, dass eine algorithmische Ableitung viel schneller zu erzeugen ist, als von Hand abgeleiteter Code, besonders wenn der zu Grunde liegende Code noch in Entwicklung ist. Dennoch können von Hand abgeleitete Teilroutinen die Ausführungszeit massiv verkürzen. So finden zum Beispiel Ableitungen von linearen Lösern (BLAS Routinen) häufig Anwendung. Dabei ist die automatisch generierte Ableitung meist sehr inperformant. Als Beispiel diene uns die Fortran-Funktion `solve(A,y,b,n)`, also $\mathbf{y} = A^{-1}\mathbf{b}$. Eine händische Ableitung ist in Quellcode 3.3 zu sehen. Es wird also nur zwei mal die Ursprungsfunktion `solve` aufgerufen. Für eine genauere Betrachtung sei auf [35, 36] verwiesen.

Quellcode 3.3: Händische Ableitung der Funktion `solve` aus BLAS, nach [36].

```
SUBROUTINE SOLVE_D(A,Ad,y,yd,b,bd,n)
       INTEGER n
       REAL A(n,n), Ad(n,n)
       REAL y(n), yd(n), b(n), bd(n)
       INTEGER i,j
       REAL RHSd(n)

       call SOLVE(A,y,b,n)
       DO i=1,n
         RHSd(i) = bd(i)
         DO j=1,n
           RHSd(i) = RHSd(i) - Ad(i,j)*y(j)
         ENDDO
       ENDDO
       call SOLVE(A,Yd,RHSd,n)
     END
```

3.3.3.3 Operator Overloading

Als Alternative bieten manche Programmiersprachen die Möglichkeit, per Operator-Überladung (engl. operator overloading), beliebige Operatoren wie Addition, Subtraktion usw. durch andere Funktionen zu ersetzen. Im Falle des automatischen Differenzierens kann man so die Operatoren entsprechend den Regeln für Ableitungen überladen.

Viele Programmiersprachen unterstützen Operator-Overloading. Zu diesen gehören die gängigen wie C++, C#, Java, Fortran ab Version 90, Matlab und Python. Damit lassen sich sowohl der Vorwärts- als auch der Rückwärtsmodus des automatischen Differenzierens realisieren. Für eine Übersicht der aktuell existierenden AD-Tools sei auf die Webseite `autodiff.org` verwiesen [37].

3.3.3.4 Beispiel

Zur Veranschaulichung des Aufwandes des Automatischen Differenzierens wurde die einfache Funktion `sim` aus Quellcode 2.1 mit Hilfe des AD-Tools Tapenade im Vektor-Vorwärts-Modus transformiert. Der resultierende Code findet sich in Quellcode 3.4.

Um das transformierte Programm die komplette Jacobi-Matrix berechnen zu lassen, muss es einmal ausgeführt werden. Dazu wird jede Ableitungsvariable `x1d,x2d,x4d,x5d` mit dem entsprechenden i-ten Einheitsvektor gesetzt. Die Vektoren mit den entsprechenden Richtungsableitungen stehen dann in der Matrix `fvecd`. Dazu muss Modul `DIFFSIZES` die Variable `nbdirsmax` enthalten, welche die Anzahl der maximal möglichen Ableitungsrichtungen p_{max} angibt, hier also `nbdirsmax=5`. Zur Ausführung kann durch den Übergabeparameter `nbdirs` diese Anzahl $p \leq p_{max}$ noch verringert werden (siehe die Betrachtung der Laufzeit in Abschnitt 3.3.3.1)

Quellcode 3.4: Automatisch mit dem AD-Tool Tapenade im vektoriellen Modus generierter Ableitungscode von Quellcode 2.1.

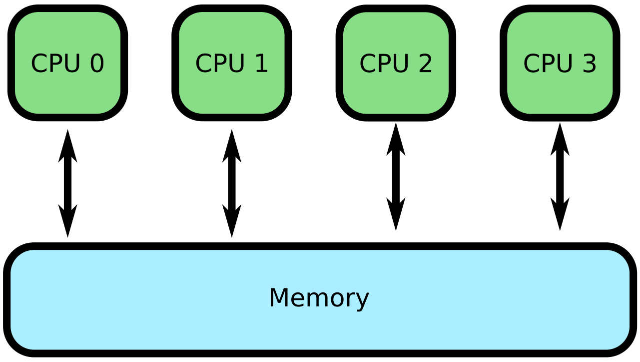
```
SUBROUTINE SIM_DV(x1, x1d, x2, x2d, x3, x3d, &
& x4, x4d, x5, x5d, fvec, fvecd, m, nbdirs)
  USE DIFFSIZES
  IMPLICIT NONE
  INTEGER :: m
```

```
  DOUBLE PRECISION :: x1, x2, x3, x4, x5
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: x1d, &
& x2d, x3d, x4d, x5d
  DOUBLE PRECISION :: fvec(m)
  DOUBLE PRECISION :: fvecd(nbdirsmax, m)
  INTEGER :: i
  DOUBLE PRECISION :: temp, temp1, temp2
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: &
& temp1d, temp2d
  INTRINSIC DBLE
  INTRINSIC EXP
  DOUBLE PRECISION :: arg1
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: arg1d
  INTEGER :: nd
  INTEGER :: nbdirs
  DO nd=1,nbdirs
    fvecd(nd, :) = 0.D0
  END DO
  DO i=1,m
    temp = DBLE(10*(i-1))
    arg1 = -(x4*temp)
    DO nd=1,nbdirs
      arg1d(nd) = -(temp*x4d(nd))
      temp1d(nd) = arg1d(nd)*EXP(arg1)
      arg1d(nd) = -(temp*x5d(nd))
    END DO
    temp1 = EXP(arg1)
    arg1 = -(x5*temp)
    temp2 = EXP(arg1)
    DO nd=1,nbdirs
      temp2d(nd) = arg1d(nd)*EXP(arg1)
      fvecd(nd,i) = x1d(nd) + x2d(nd)*temp1 + x2*&
&      temp1d(nd) + x3d(nd)*temp2 + x3*temp2d(nd)
    END DO
    fvec(i) = x1 + x2*temp1 + x3*temp2
  END DO
END SUBROUTINE SIM_DV
```

3.4 Parallelisierung

Wie Eingangs erwähnt, benötigen wir für die Lösung von großen Problemen eine Art der Parallelisierung. Dabei hat sich für Computer mit gemeinsamem Speicher (Shared-Memory Computer) das Programmiermodell OpenMP etabliert [38]. Die Anzahl der Recheneinheiten innerhalb eines Shared-Memory Systems ist limitiert auf aktuell 128 Rechenkerne. Für sehr große Probleme ist diese Anzahl von Rechenkernen ungenügend. Es werden daher Rechenanlagen mit vielen einzelnen Computern, die über ein Netzwerk miteinander verbunden sind, sogenannte Cluster-Rechner, aufgebaut. Diese Cluster haben dann keinen gemeinsamen Speicher mehr, sondern müssen verteilten Speicher (Distributed-Memory Computer) einsetzen. Zur Programmierung dieser hat sich das Message Passing Interface (MPI) etabliert [39].

Abbildung 3.3 Aufbau eines Mehrkern-Systems mit gemeinsamem Speicher.



3.4.1 Shared Memory Systeme

So gut wie jeder neue Prozessor ist heute ein Mehrkernprozessor. Das heißt, es existieren mehrere unabhängige Prozessorkerne mit ihren eigenen Ausführungseinheiten innerhalb eines Prozessors. Diese Kerne sind dabei über einen gemeinsamen Speicher miteinander verbunden, welcher von jedem Kern gleich schnell zugreifbar ist. Dies ist in Abb. 3.3 zu sehen.

OpenMP Eine einfache Möglichkeit solche parallelen Systeme zu programmieren bietet der Einsatz von OpenMP [40]. Dabei werden die in einem seriellen Programm existierenden Schleifen, welche sich parallelisieren lassen mit Hilfe von Pragmas, einfachen Präprozessordirektiven mit Hinweisen an den Compiler, parallelisiert.

Der Code aus Quellcode 2.1 soll uns als einfaches Beispiel dienen. Quellcode 3.5 zeigt den mit Hilfe von OpenMP parallelisiert Code, der die exponentielle Datafit-Funktion aus Abschnitt 2.2.1 parallel berechnet.

Quellcode 3.5: OpenMP-Parallelisierte Version des Fortrancodes der Funktion `sim` aus Quellcode 2.1.

```
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
...
!$OMP parallel
!$OMP do private(i,temp,temp1,temp2)
  do i = 1, m
      temp = dble(10*(i-1))
      temp1 = exp(-x4*temp)
      temp2 = exp(-x5*temp)
      fvec(i)=(x1+x2*temp1+x3*temp2)
  end do
!$OMP end do
!$OMP end parallel
end
```

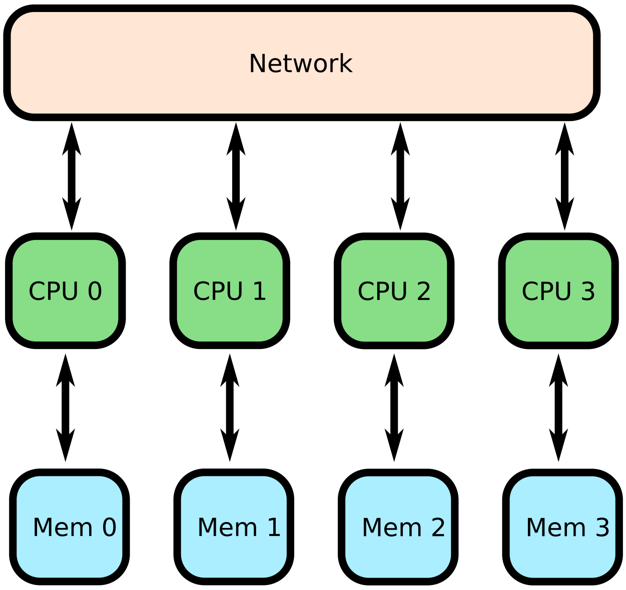
Der Code wurde um einige einfache OpenMP Compiler-Direktiven (kurz Pragmas), beginnend mit `!$OMP` versehen, welche die parallele Ausführung steuern. Abseits der Direktiven wurde das Programm nicht verändert, es ähnelt somit stark dem seriellen Code. Parallelisiert wird die Schleife über die Berechnung der `m` Elemente von `fvec`. Die Zählvariable `i`, sowie die temporären Variablen `temp,temp1,temp2` werden als private Variablen definiert und werden somit jedem Thread als private Kopien zur Verfügung gestellt. Die restlichen Variablen, insbesondere `fvec`, werden implizit als shared angenommen, können also von allen Threads gelesen und geschrieben werden. Da die einzelnen Schleifeniterationen unabhängig voneinander abgearbeitet werden können – jeder Thread greift auf einen anderen Eintrag von `fvec` zu – ist die Parallelisierung als korrekt anzusehen.

Quellcode 3.6 zeigt den mit Hilfe von Tapenade abgeleiteten Code von Quellcode 2.1. Dabei wurde der AD-Vorwärtsmodus mit der Option `-multi` genutzt, welcher den Vektormodus (Tangent Multidirectional Mode) bei Tapenade aktiviert. Dieser Code wurde mittels OpenMP parallelisiert. Es wurde ein konservativer Weg gewählt, der nur die äußere For-Schleife parallelisiert (so wie in der Parallelisierung der ursprünglichen Funktion Quellcode 3.5). Dabei wurde auf die Korrektheit der privat gesetzten Variablen geachtet. Da es zur Zeit kein AD-Tool gibt, welches zuverlässig OpenMP Paradigmen unterstützt, wurde die Parallelisierung im Nachhinein händisch durchgeführt.

Quellcode 3.6: Mit Tapenade automatisch generierte Ableitung der Funktion `sim` aus Quellcode 2.1 mit handgeschriebener Parallelisierung in OpenMP.

```fortran
  SUBROUTINE SIM_DV(x1, x1d, x2, x2d, x3, x3d, x4, &
& x4d, x5, x5d, fvec, fvecd, m, nbdirs)
  USE DIFFSIZES
  IMPLICIT NONE
  INTEGER :: m
  DOUBLE PRECISION :: x1, x2, x3, x4, x5, fvec(m)
  DOUBLE PRECISION :: x1d(nbdirsmax), x2d(nbdirsmax),&
& x3d(nbdirsmax), x4d(nbdirsmax), x5d(nbdirsmax),
& fvecd(nbdirsmax, m)
  INTEGER :: i
  DOUBLE PRECISION :: temp, temp1, temp2
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: temp1d, temp2d
  DOUBLE PRECISION :: arg1
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: arg1d
  INTEGER :: nd
  INTEGER :: nbdirs
  INTRINSIC EXP
  INTRINSIC DBLE
!$OMP PARALLEL DO private(i,temp,temp1,temp2,nd,arg1,
!   temp1d,arg1d,temp2d)
  DO i=1,m
    temp = DBLE(10∗(i-1))
    arg1 = -(x4∗temp)
    DO nd=1,nbdirs
      arg1d(nd) = -(temp∗x4d(nd))
      temp1d(nd) = arg1d(nd)∗EXP(arg1)
      arg1d(nd) = -(temp∗x5d(nd))
    END DO
    temp1 = EXP(arg1)
    arg1 = -(x5∗temp)
    temp2 = EXP(arg1)
    DO nd=1,nbdirs
      temp2d(nd) = arg1d(nd)∗EXP(arg1)
      fvecd(nd, i) = x1d(nd) + x2d(nd)∗temp1 + x2∗&
&      temp1d(nd)+ x3d(nd)∗ temp2 + x3∗temp2d(nd)
    END DO
    fvec(i) = x1 + x2∗temp1 + x3∗temp2
  END DO
!$OMP END PARALLEL DO
END SUBROUTINE SIM_DV
```

Abbildung 3.4 Aufbau eines Mehrkern-Systems mit verteiltem Speicher.



Python Multiprocessing Innerhalb Pythons gibt es ein einfach zu verwendendes Modul für die Arbeit mit Systemen mit gemeinsamem Speicher, das Modul Multiprocessing [41]. Kernelement ist ein Pool von Arbeitsprozessen, die über eine Map-Funktion ihre Arbeiten zugeteilt bekommen. Quellcode 3.7 zeigt hier das einfache Vorgehen um eine Funktion parallel auf eine große Anzahl von Elementen eines Arrays anzuwenden.

Quellcode 3.7: Einfaches Beispiel für die Verwendung eines Multiprocessing Pools zur Verteilung von Aufgaben auf mehrere Prozesse.

```
from multiprocessing import Pool
def fun(x):
    return x*x
arr=range(1E6)
p=Pool(5)
p.map(fun,arr)
```

3.4.2 Distributed Memory Systeme

Neben den Shared Memory Systemen haben sich, besonders bei Großrechnern verteilte Speichersysteme etabliert. Einen Überblick gibt Abb. 3.4. Jeder Prozessor (oder besser Knoten) eines verteilten Systems hat seinen eigenen lokalen Speicher. Wenn ein anderer Prozessor auf den entfernten Speicher zugreifen möchte, muss eine Verbindung über das Netzwerk erfolgen. Dabei werden zumeist Nachrichten erzeugt, die die geforderten Informationen beinhalten.

MPI Diese Systeme können nicht mit OpenMP programmiert werden, sondern benötigen ein anderes Programmiermodell. Das wohl bekannteste Paradigma ist das Message Passing Interface (MPI) [42, 43, 44]. Dabei benötigen die in MPI geschriebenen Programme im Vergleich zu OpenMP einen erhöhten Programmieraufwand, da mit dem verteilten Speicher die Datenstrukturen des Programms angepasst werden müssen. Im Speziellen müssen Arrays explizit aufgeteilt werden und die Daten entsprechend über Sende- und Empfangsbefehle verteilt werden. Dabei muss beachtet werden, dass jeder Prozess nur Zugriff auf seinen eigenen Teil des Gesamtarrays besitzt. Um auf andere Teile des Arrays zuzugreifen, ist ein Kommunikationsschritt notwendig. Somit wird eine Operation, für die das gesamte Array benötigt wird, zum Flaschenhals.

Quellcode 3.8: MPI Parallelisierung des Codes aus Quellcode 2.1.

```
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
use mpi
integer m,local_m,prank,psize
double precision x1,x2,x3,x4,x5,fvec(m)
double precision, allocatable:: local(:)
integer i
double precision temp, temp1, temp2
real t1,t2
call mpi_comm_rank(mpi_comm_world,prank,ierr)
call mpi_comm_size(mpi_comm_world,psize,ierr)
local_m = m/psize
allocate(local(local_m))
do i = 1, local_m
  k=i+prank*local_m
  temp = dble(10*(k-1))
  temp1 = exp(-x4*temp)
  temp2 = exp(-x5*temp)
  local(i)=(x1+x2*temp1+x3*temp2)
end do
call mpi_gather(local,local_m,&
  & MPI_DOUBLE_PRECISION,fvec,local_m,&
  & MPI_DOUBLE_PRECISION,MPI_IN_PLACE,&
  & MPI_COMM_WORLD,ierr)
end
```

Der Code in Quellcode 3.8 zeigt den so mit MPI parallelisierten Datafit-Code. Dabei wird das Array `local` mit Größe `local_m` als Arbeitsarray definiert, welches jeder Prozess als privates Array nutzt, um seine Zwischenergebnisse abzulegen. Mittels `mpi_gather` werden im Anschluss die finalen Ergebnisse jedes Prozesses im Ergebnisarray `fvec` des Masterprozesses gesammelt. Die Ableitungsroutine `sim_dv()` wurde analog parallelisiert und ist in Anhang A.1 zu sehen. In nicht-synthetischem Simulationscode wird der entsprechende Programmierer Aufrufe von `mpi_init()` und `mpi_finalize()` benutzt haben, um für eine korrekte Arbeitsweise der MPI Parallelisierung zu sorgen. Mit der Benutzung von `mpi4py` in EFCOSS werden diese jedoch implizit gesetzt und müssen für eine fehlerfreie Ausführung der Simulation auskommentiert werden [45].

Kapitel 4

EFCOSS

Typischerweise werden Pakete für Optimierungsprobleme und Software zur numerischen Simulation von unterschiedlichen wissenschaftlichen Gruppen entwickelt. Optimierer werden meist von Experten aus dem Bereich numerische Analysis oder wissenschaftliches Rechnen entwickelt, Simulationswerkzeuge hingegen von Wissenschaftlern mit Expertise aus dem jeweiligen wissenschaftlichen Feld. Diese beiden Gruppen beschreiben ihren Code zumeist so, dass er allgemeingültig in dem jeweiligen Anwendungsgebiet ist. Das heißt aber nicht, dass der Code einfach zusammengesetzt werden kann, sondern es ist Aufgabe der Softwareentwicklung, ein geeignetes Interface zwischen diesen beiden Codes zu entwickeln. Zwei Szenarien sind dabei denkbar.

Aus Sicht der Simulationsentwickler wäre es wünschenswert, verschiedene Optimierungspakete mit dem eigenen Code im Hinblick auf numerische Stabilität, Konvergenzverhalten etc. zu testen. Dies ergibt sich aus der Tatsache, dass man erst einen geeigneten Optimierer finden muss, der den eigenen Ansprüchen genügt. Aus Sicht eines Numerikers, der ein neues Optimierungswerkzeug entwickelt, ist es sinnvoll, dieses mit verschiedenen Simulationscodes zu testen.

Arno Rasch hat das Software-Framework EFCOSS – Environment For Combinig Optimization and Simulation Software – entwickelt, welches die Belange der beiden Gruppen möglichst gut abdeckt. Einen guten Überblick über EFCOSS bietet die Veröffentlichung von Rasch und Bücker [4], sowie die Dissertation von Rasch [46], in der er EFCOSS anhand eines Beispiels für Optimal Experimental Design beschreibt.

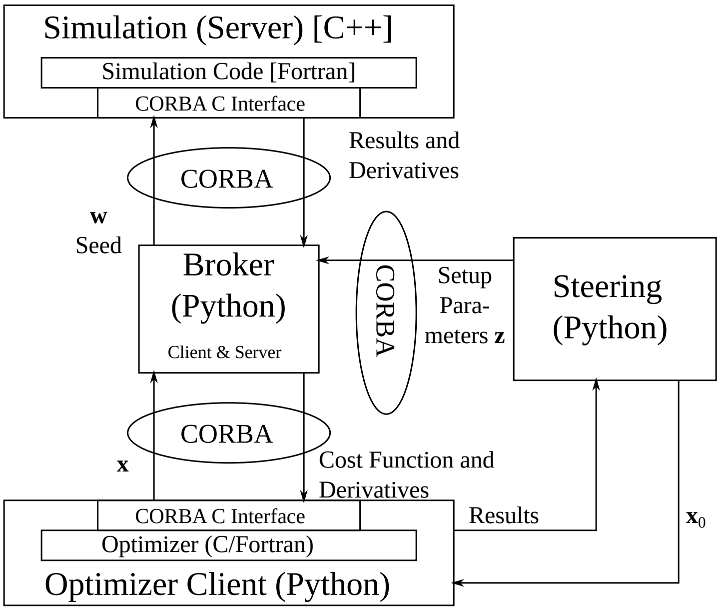
Ein alternatives Softwarepaket, welches Simulation und Optimierung zusammenbringen möchte, ist das Toolkit for Advanced Optimization (kurz TAO) [47, 48, 49]. Es ist eine komponentenbasierte Optimierungssoftware, welche für sehr große Optimierungsprobleme entworfen wurde. Die unterstützten Zielfunktionen kommen aus dem Bereich der nichtlinearen kleinsten Quadrate, der unbeschränkten und beschränkten Minimierung, sowie der generellen nichtlinearen Optimierung. Im Bereich der optimalen Versuchsplanung (OED) ist als weiteres das VPLAN Softwarepaket [50] zu nennen. Dieses wurde für die Lösung von Problemen aus dem Bereich der Verfahrenstechnik entwickelt. Weitere Softwarepakete zur Kopplung von Optimierung und Simulation werden in [4] vorgestellt.

4.1 Ursprüngliches Design

Der ursprüngliche Entwurf von EFCOSS sah ein komplett verteiltes Framework vor, welches auf der Common Object Request Broker Architecture (kurz CORBA) [51] aufbaut. Abbildung 4.1 zeigt dieses Design. Das Modul „Simulationsserver“ übernimmt die Berechnung des Vorwärtsmodells und eventueller Ableitungen von diesem. Der „Optimierer“ führt den entsprechenden Optimierungsalgorithmus aus. Die Kommunikation zwischen den einzelnen Komponenten wird über CORBA realisiert. Damit folgt EFCOSS ganz klar dem dort genutzten Prinzip eines Brokers, der als Vermittler zwischen der Simulation und dem Optimierer fungiert. Zur Steuerung des gesamten Systems wird ein in Python geschriebenes Script genutzt.

CORBA bietet eine große Anzahl von Services und kann verschiedene Programmiersprachen miteinander kombinieren, um so verteiltes Rechnen zu ermöglichen. Damit ist es möglich, das rechenintensive Simulationsmodell auf einem Supercomputer auszuführen, während die Optimierung auf einem Laptop laufen kann. Der sogenannte Object Request Broker (ORB), den jede CORBA Anwendung beinhaltet, definiert die plattformunabhängigen Protokolle und Services. Um die Anfragen der anderen Teilnehmer zu bearbeiten und anfallende Daten zu übertragen, benötigt jede Komponente ein CORBA Interface. Dieses wird zumindest für die Simulation automatisch in

Abbildung 4.1 Ursprüngliches EFCOSS Design nach [4].



EFCOSS generiert.

Der Broker, welcher ebenfalls in Python geschrieben ist, spielt als Vermittler zwischen den einzelnen Teilen eine entscheidende Rolle. Der Eingabevektor für die Simulation $\mathbf{w} \in \mathbb{R}^{n+o}$ besteht aus zwei verschiedenen Gruppen von Variablen: zu optimierende Variablen $\mathbf{x} \in \mathbb{R}^n$ und fixe Argumente $\mathbf{z} \in \mathbb{R}^o$. Mit Hilfe des Steuerungsscriptes werden Initialwerte für alle Parameter und Variablen der Simulation gesetzt, das heißt $\mathbf{w} = (\mathbf{x}_0, \mathbf{z})$.

Für den Optimierungsalgorithmus wird das $\mathbf{x}_0$ als Startwert benötigt. Wenn der Optimierer die Zielfunktion $\phi(\mathbf{x})$ evaluieren möchte, übermittelt er dem Broker (über CORBA) einen entsprechenden Wert für $\mathbf{x}$. Der Broker übernimmt daraufhin diesen Wert für $\mathbf{x}$ und kombiniert ihn mit den hinterlegten, unveränderlichen Parametern $\mathbf{z}$ für die Simulation, so dass ein neuer Eingabevektor für die Simulation $\mathbf{w} = (\mathbf{x}, \mathbf{z})$ erzeugt wird. Dieses Wertepaar wird, zusammen mit weiteren Werten wie der Saatmatrix (im Beispiel `Seed`) für die automatisch generierten Ableitungen, an den Simulationsserver übertragen. Als Rückgabe erhält der Broker das Ergebnis und eventuelle Ableitungen. Mit diesen Ergebnissen generiert er ein passendes Kostenfunktional und gegebenenfalls dessen Ableitungen. Zusätzlich können in diesem Konstrukt Nebenbedingungen ausgewertet werden, die jedoch in einem eigenen Simulationsserver angesiedelt sind. Diese Funktions- und Nebenbedingungswerte werden dann an den Optimierer weitergegeben, der dann seinem Algorithmus weiter folgen kann und entweder ein neues $\mathbf{x}$ generiert oder das Ergebnis an das Steuerungsscript zurückgibt.

Die Daten in EFCOSS werden in zwei Kategorien unterteilt, Integer und Double Datentypen. Wenn eine Variable in EFCOSS erzeugt wird, so entscheidet der Initialwert, welcher Datentyp genommen wird. Des Weiteren können bereits angelegte Integervariablen als Initialwert für Double Datentypen genutzt werden, um ein Array zu definieren, welches die Größe des in der Integervariablen gespeicherten Wertes besitzt.

Damit lässt sich ein einfacher Arbeitsablauf für die Benutzung von EFCOSS ableiten:

1. Generieren eines CORBA Interface für den Simulationscode in C++.
2. Schreiben eines Steuerungsscriptes für die Konfiguration des Brokers in Python.

3. Schreiben eines Scriptes für die Optimierung in Python.
4. Starten des CORBA Nameservices zur Identifikation der beteiligten Dienste.
5. Starten des Simulationsservers.
6. Starten des Brokers.
7. Ausführen des Steuerungsscriptes.
8. Starten des Optimiererscripts.

Dieser Arbeitsablauf wurde erfolgreich in diversen Veröffentlichungen genutzt, um Optimierungsprobleme aus unterschiedlichen Anwendungsgebieten zu lösen [52, 53, 54, 46].

CORBA fehlen jedoch entscheidende Eigenschaften, die wir in dieser Arbeit benötigen. Wenn CORBA in einem HPC-Umfeld eingesetzt wird, müssen zwangsläufig offene Ports in der Firewall geschaffen werden, oder der Anwender muss diverse SSH-Tunnelmethoden nutzen, um die Systeme zum Laufen zu bringen. Dies macht es schwierig, CORBA auf groß skalierten Rechenanlagen ordentlich zu warten und zu betreiben [55]. Ein weiterer wichtiger Punkt ist die fehlende Interoperabilität mit parallelen Programmiermodellen wie MPI, OpenMP oder CUDA. Man kann zwar den CORBA Server parallelisieren, jedoch ist dies manchmal nicht ausreichend. Für kleinere Cluster zeigt sich dieses Vorgehen als ausreichend, wenn die Anzahl der Knoten signifikant erhöht wird, tendieren Simulationscodes jedoch dazu, limitiert in ihrer Skalierbarkeit zu sein. Dann wäre es sinnvoll, die nicht benötigten Computerknoten mit anderen Dingen zu befüllen, um die volle Leistung aus dem System herauszubekommen.

Dieses kann zum Beispiel bei den in Kapitel 2.2.2 definierten OED-Problemen Anwendung finden. Wie bereits beschrieben, ist die Fisher-Matrix (2.27) für das Vorwärtsproblem $\mathbf{f}(\mathbf{x}, \theta)$ von θ abhängig. Somit hängt auch das zugrundeliegende Optimierungsproblem

$$\min_{\mathbf{x}} \Psi(\mathbf{x}) \equiv \Psi_\theta(\mathbf{x})$$

von θ ab. Aus diesem Grund ist man normalerweise nicht nur an der Lösung eines OED Problems interessiert, sondern an der Lösung verschiedener OED Probleme, deren Parameter θ verschieden sind. Diese OED Probleme können unabhängig voneinander gelöst werden und sind somit prädestiniert für eine parallele Ausführung. Somit muss die Software-Architektur die parallele Ausführung von Optimierungsalgorithmen unterstützen. Darüber hinaus benötigt jeder dieser Optimierer mit CORBA eine eigene Instanz eines Simulationsservers. Aus unserer Sicht ist daher der Einsatz von CORBA an dieser Stelle nicht mehr zeitgemäß.

Ein anderer entscheidender Fakt, speziell für Nicht-Informatiker, ist die schiere Komplexität, um ein solches verteiltes System aufzusetzen. Dabei stellen sowohl die Handhabung und Wartung der verschiedenen Softwarepakete mit ihren verschiedenen Abhängigkeiten, sowie möglicherweise komplett verschiedene Plattformen, eine nur schwierig zu überwindende Barriere für typische Anwender dar.

Für den bisherigen Anwendungsfall, bei dem EFCOSS auf einer einzelnen Maschine gestartet wird, ist ein verteilter Ansatz hinderlich und sollte durch ein einfaches Script oder interaktive Eingaben ersetzt werden, ohne dass man sich vorher mit den technischen Problemen von Nameservices, Client-Server-Modellen usw. auseinandersetzen muss.

Als Ziel dieser Arbeit können folgende Punkte als Anforderungen an das neue Software-Design definiert werden:

1. Vereinfachung des Setups und der Nutzung von EFCOSS.

2. Verteiltes Rechnen sollte eine Option darstellen, keine Notwendigkeit.
3. EFCOSS sollte die parallele Ausführung sowohl von mehreren Simulations-, als auch mehreren Optimierungsinstanzen unterstützen
4. Verwendung von Standard Python-Paketen und deren Datentypen als Grundlage des gesamten Systems.
5. Einfache Umsetzung der Probleme mit mehreren Ziel- und/oder Simulationsfunktionen für Space-Mapping und Modelldiskriminierung.
6. Erweiterung um den Rückwärtsmodus des automatischen Differenzierens.

4.2 Überarbeitung

Mit den oben genannten Aspekten können wir nun die Überarbeitung von EFCOSS beschreiben. Die wohl wichtigste Architekturentscheidung ist dabei, CORBA zu ersetzen und auf eine reine Python-basierte Implementierung zu setzen. Dies schließt insbesondere die Vorteile der Softwareinfrastruktur Pythons und deren umfangreiche Paketsammlung mit ein.

4.2.1 Neue Struktur

In Abb. 4.2 ist die neue Struktur von EFCOSS aufgeführt. In einem Steuerungsscript beschreibt ein Benutzer die Stuktur seiner Optimierungsaufgabe und des zugrundeliegenden Simulationscodes. Der Optimierer kommuniziert intern über die von EFCOSS generierten und genutzten Schnittstellen mit der Simulation.

Abbildung 4.2 Neues EFCOSS Design ohne CORBA

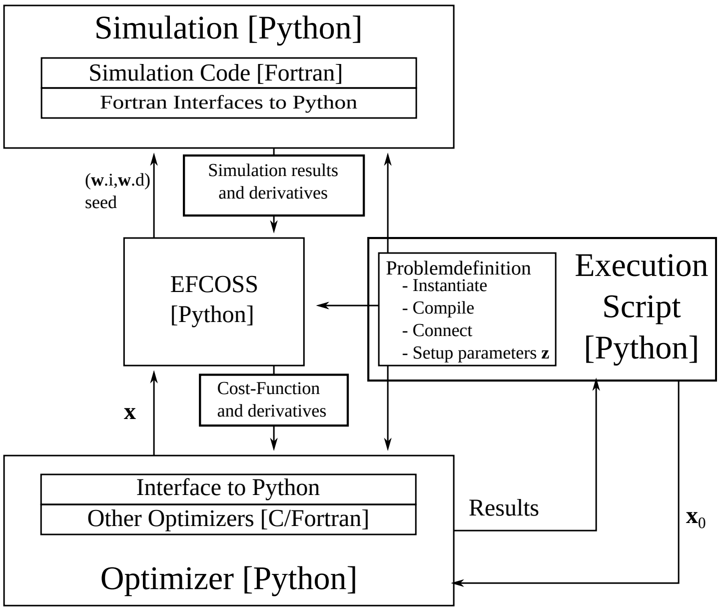
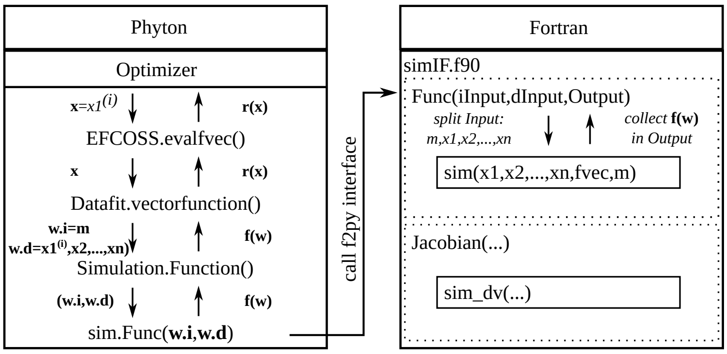


Abbildung 4.3 zeigt die angestrebte Struktur von EFCOSS. Von einem Optimierungs-Algorithmus wird die `evalfvec` Routine von EFCOSS aufgerufen, d.h. der Optimierer möchte den Ergebnisvektor einer Zielfunktionsauswertung. Als Eingabe erhält die Funktion ein neues $\mathbf{x}$ vom Optimierer und liefert als Ergebnis $\mathbf{r}(\mathbf{x})$. Dieses Interface ruft die vorher vom Benutzer festgelegte Zielfunktion auf, in diesem Beispiel die Datafit Vektorfunktion. In dieser Funktion wird der gegebene Eingabevektor um die weiteren Eingaben der Simulation ergänzt, welche im Optimierungsalgorithmus

Abbildung 4.3 Schematischer Überblick über den Datenfluss einer Iteration des Optimierungsprozesses.



nicht berücksichtigt werden sollen. Die zusammengesetzten Eingaben heißen auf der Abbildung **w.i**, welches im Simulations-Interface für die Integer-Eingabewerte (`iInput`), und **w.d**, welches für die Floating-Point-Eingabewerte (`dInput`) steht. In der Zielfunktion, hier $\mathbf{r}(\mathbf{x}) = \mathbf{f}(\mathbf{w}) - \mathbf{y}_{mess}$, wird die Simulation als primäres Ergebnis benötigt und entsprechend der vorher definierten Funktionen genutzt. Deshalb müssen innerhalb der Zielfunktionen die Simulations-Interfaces aufgerufen werden, welche letztendlich die Ergebnisse der Simulation $\mathbf{f}(\mathbf{w})$ liefern. In dieser Simulationsklasse werden dann die darunterliegenden Simulationscodes, mit einem entsprechenden Wrapper für die Programmiersprache der Simulation, aufgerufen. In diesem Beispiel ist dies die Anbindung an eine in Fortran geschriebene Simulation mittels des Tools Fortran to Python `f2py` [56]. Wie dies genau funktioniert, ist in Abschnitt 4.4.4 beschrieben. Nun berechnet die Simulation eine Funktionsauswertung und gibt das Ergebnis über die bereits besprochenen Komponenten an den Optimierer zurück.

Der Optimierer kann nun über eine ähnliche Struktur die Ableitungsobjekte beziehungsweise die Nebenbedingungen berechnen und anhand deren Ergebnissen die neue Suchrichtung festlegen. Für manche der beschriebenen Probleme aus Abschnitt 2 sind jedoch verschiedene Simulationsfunktionen oder Zielfunktionen notwendig. Gegebenenfalls müssen auch verschiedene Optimierungsalgorithmen eingesetzt werden.

Aus diesen Überlegungen lässt sich nun eine neue Struktur ableiten, die die neuen Gegebenheiten berücksichtigt und insbesondere im Hinblick auf Parallelität einfacher handhabbar ist. Um dem Rechnung zu tragen, bietet Python die Möglichkeit von Untermodulen, welche im Folgenden genauer untersucht werden soll. Die überarbeitete Struktur von EFCOSS besteht aus den folgenden **`(Unter-)Modulen`** (fett), sowie deren wichtigsten Klassen.

- **`EFCOSS`** - Das übergeordnete Modul
  - `EFCOSS` - Die EFCOSS-Interface Klasse
  - `ProblemDefinition` - Basisbeschreibung für die Arbeit mit EFCOSS
  - **`Simulation`** - Modul für die Anbindung von Simulationen
    - `Simulation` - Die Basisklasse für die Simulation
    - `Variables` - In der Simulation verwendete Variablen
    - `Buffer` - Speicherung von Zwischenergebnissen
    - **`Codegenerator`** - Das Codegenerator-Modul
      - `CodeGeneratorBase` - Die Basisklasse für die Codegeneratoren
      - `CodeGeneratorFortran` - Eine explizite Beschreibung der Sprachkonstrukte für

        Fortran
      - `SimulationIF` - Die abstrakte Beschreibung von Interfaces zur Simulation
      - `SimulationIF_tapenade` - Die abstrakte Beschreibung zur Ableitung der Simulation
  - **`Optimization`** - Das Optimierungs-Modul
    - **`Objectives`** - Das Modul für die Zielfunktionen
      - `ScalarObjective` - Eine Klasse für Skalare Zielfunktionen
      - `VectorObjective` - Eine Klasse Vektorielle Zielfunktionen
      - `DataFit` - Eine Klasse mit Zielfunktionen bzw. Residualvektor für die Parameterbestimmung
      - `OED` - Eine Klasse mit Zielfunktionen für die optimale Versuchsplanung
    - **`Constraints`** - Das Modul für die Nebenbedingungen (NB)
      - `LinearConstraints` - Die Klasse für Lineare NB
      - `NonLinearConstraints` - Die Klasse für nichtlineare NB
    - `opt_scipy` - Ein Interface zu Scipy Optimierern
    - `opt_elsunc` - Ein Interface zum ELSUNC Optimierer
  - **`Utilities`** - Das Modul für Hilfsfunktionen
  - **`Applications`** - Applikationsspezifischer Code

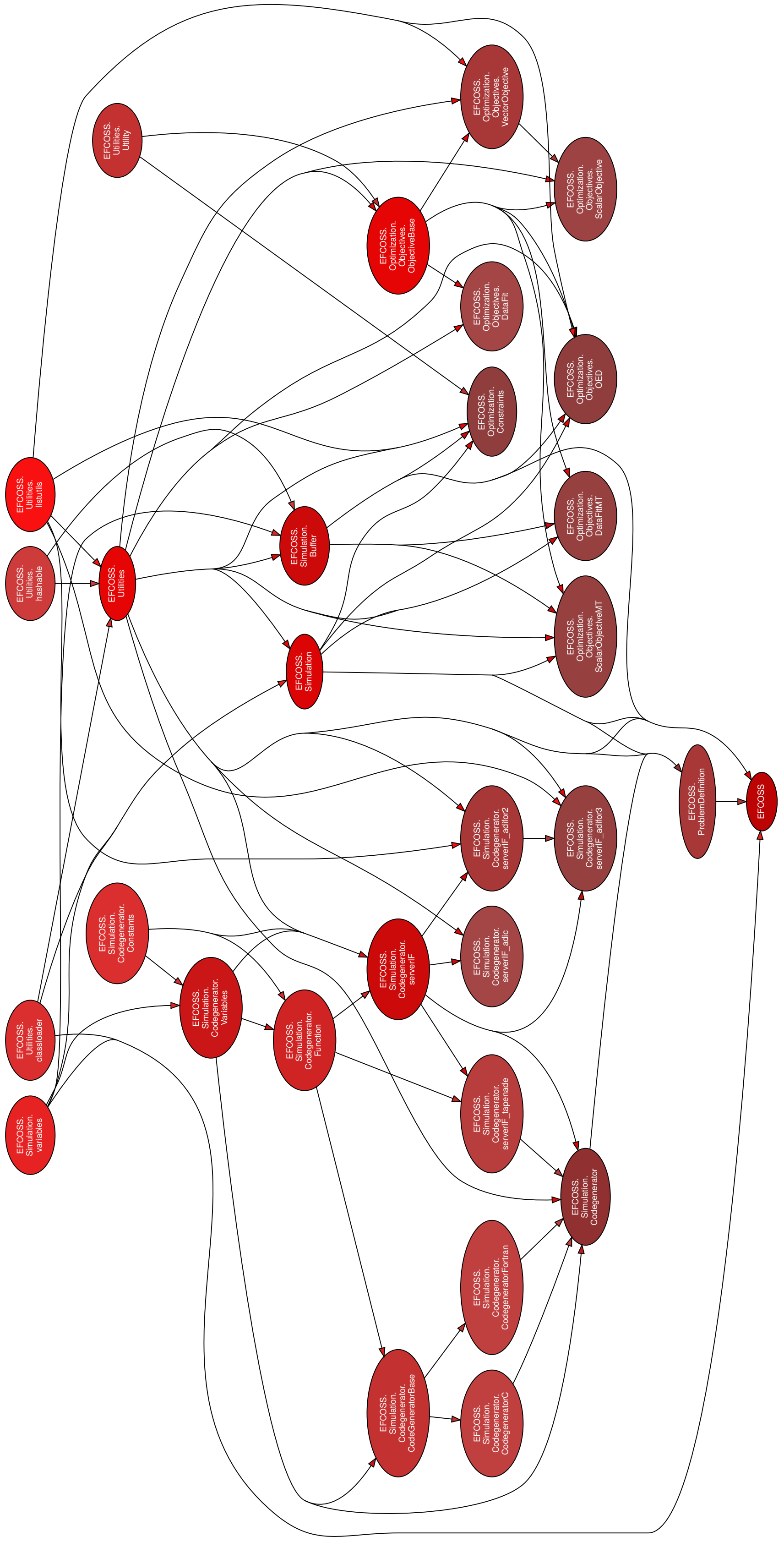
Die oberste Ebene unseres Modulbaums beinhaltet die `EFCOSS` Klasse sowie die Basisklasse einer einfachen `ProblemDefinition`. Als Module finden sich die Hauptmodule `Simulation` und `Optimization`, sowie `Utilities` und `Applications`.

Im `Simulation` Modul werden alle für die Anbindung von Simulationen an EFCOSS benötigten Grundmethoden und Klassen bereitgestellt. Zu diesen gehören die Basisklassen für Variablen und der Simulation, sowie die Methoden zum Speichern von Zwischenergebnissen (`Buffer`). Zum Erstellen von Interfaces findet sich das `Codegenerator` Modul, welches abstrakte Beschreibungen der Simulations-Interfaces bereitstellt. Diese können über die verschiedenen Codegenerator-Klassen ausführbaren Interface-Code für verschiedene Programmiersprachen generieren. Die in dieser Arbeit hauptsächlich benutzte Programmiersprache für Simulations-Codes ist Fortran, die Klasse ist jedoch darauf vorbereitet, um weitere Programmiersprachen erweitert zu werden. Das Modul `Optimization` beinhaltet die Interfaces für die zur Verfügung stehenden Optimierer. Zu diesen gehören die Optimierer aus dem Scipy Optimizer Paket, sowie Interfaces zu den in anderen Sprachen implementierten Optimierern. Für die Zielfunktionen und etwaige Constraints sind die Module `Objectives` und `Constraints` verantwortlich. In den `utilities` finden sich Abkürzungen zu häufig benötigten Funktionen, wie dem Erzeugen und Auslesen von Variablen oder zum formatierten Ausgeben von Ergebnissen. Des Weiteren finden sich hier Methoden zur Erzeugung von Hashes. In Abb. 4.4 findet sich eine Übersicht der in EFCOSS enthaltenen Klassen und Module.

4.2.2 Arbeiten mit der neuen Struktur von EFCOSS

Um die Arbeit mit EFCOSS so einfach wie möglich zu gestalten, wurde eine grobe Struktur einer Anwender-Klasse erstellt, die `ProblemDefinition`, die als Basis für jedes mit EFCOSS zu berechnende Problem dienen kann. Dabei wurde auf eine möglichst einfache Implementierbarkeit der fehlenden Teilmethoden in der Klasse geachtet. Dies ist in Quellcode 4.1 exemplarisch dargestellt.

Abbildung 4.4 Überblick über die EFCOSS Module.



Quellcode 4.1: Grobe Struktur einer vom Benutzer zu implementierenden Klassendefinition für die Arbeit mit EFCOSS.

```
import EFCOSS

class NAME(EFCOSS.ProblemDefinition):
    ...
   #Initialize EFCOSS environment
   # + set input/output variables
   # + set calling sequence
   # + set optimization parameters
   def initEFCOSS(self,x0):
      ...
   #Set an objective function for the optimization
   def setObjectiveFunction(self):
      ...
   #Do the optimization
   def runOptimizer(self):
      ...
```

Nehmen wir nun an, ein Benutzer möchte das unter Abschnitt 2.2.1 definierte Parameterschätzproblem umsetzen. Als Grundlage dient ihm der Fortran Code aus Quellcode 2.1 sowie der Ableitungscode Quellcode 3.4, welcher mittels Tapenade erzeugt wurde. Zusätzlich sollen die Messdaten in einer Datei `data.txt` in der Form [Index, Datum, Gewicht] vorliegen.

Es müssen zwei neue Dateien implementiert werden. Eine für die eigentliche Problemdefinition und eine zum Steuern dieser. Die Namensgebung der Dateien ist dabei dem Nutzer überlassen, wir nennen sie hier Exemplarisch `DatafitExample.py` für die Definition und `run.py` für das Ausführungsscript. Das Ausführungsscript ist in Quellcode 4.2 zu sehen und besteht im Wesentlichen aus dem Aufruf der einzelnen Methoden der Klasse `DatafitExample` aus der Datei `DatafitExample.py`. Wir starten mit der Definition für einen Startwert `x0`. Dann wird die Klasse instanziiert und anschließend die Methode `initEFCOSS` aufgerufen.

Danach kann das Interface mit Hilfe der Funktion `generateSimulationInterfaceFortran()` generiert werden, welches dann mit `compileSimulationInterfaceFortran` compiliert wird. Im Anschluss kann das eben gebaute Interface genutzt werden, indem die `initSimulationInterface()` aufgerufen wird. Wichtig dabei ist, dass diese Funktionen aus der Klasse `ProblemDefinition` stammen und vom Benutzer nicht definiert werden müssen. Die genaue Funktion wird in der Beschreibung der Klasse `ProblemDefinition` (Abschnitt 4.4.1) und weiter in der Beschreibung der Codegeneratoren Abschnitt 4.4.4 gezeigt. Nun muss die Zielfunktion gesetzt werden. Die Funktion `setObjectiveFunction` muss vom Benutzer eigenhändig implementiert werden. Zum Schluss kann die Optimierung mittels `runOptimizer` gestartet werden.

Quellcode 4.2: Beispiel für ein Ausführungsscript (hier `run.py`).

```
from DatafitExample import *

#Initial Values for x
x0=[0.5,1.5,-1.0,0.01,0.02]
#Declare Instance df
df=DatafitExample("Datafit")
#Initialize EFCOSS with x0
df.initEFCOSS(x0)
#Generate simulation Interface with Fortran and Tapenade Codes
```

```
df.generateSimulationInterface(adtool="tapenade",adlevel=1,adreverse=True)
#Compile the Interface
df.compileSimulationInterfaceFortran()
#Set the simulation interface
df.initSimulationInterface()
#Set the objective Function
df.setObjectiveFunction()
#Run the optimizers
res=df.runOptimizer()
```

Die vom Benutzer zu implementierenden Funktionen sind demnach nur noch `initEFCOSS`, `setObjectiveFunction` sowie `runOptimizer`. In `initEFCOSS` werden die Variablen der Simulation sowie deren Ausführungsreihenfolge gesetzt. Der entsprechende Code ist in Quellcode 4.3 gegeben. Um zu verstehen, was hier geschieht, sehen wir uns die Signatur der Funktion `datafit` noch einmal an:

```
subroutine datafit(x1,x2,x3,x4,x5,fvec,m)
```

Zu sehen sind die Eingaben in Form von `x1`, `x2`,...,`x5`, sowie `m`. Als Ausgabe dient der Vektor `fvec`. Dies findet sich ebenso in der Definition `initEFCOSS` wieder, indem für die Eingabevariablen die Funktion `newInputVariable` und für die Ausgabe `newOutputVariable` verwendet werden. Im Anschluss wird die Aufrufreihenfolge festgelegt, sowie die aktiven, zu optimierenden Parameter definiert.

Quellcode 4.3: Beispiel für eine Initialisierungsmethode.

```
class DatafitExample(EFCOSS.ProblemDefinition):
  def initEFCOSS(self,x0):
      #Input Definition
      self.m   = self.efcoss.newInputVariable("m",33)
      self.x1 = self.efcoss.newInputVariable("x1", x0[0])
      self.x2 = self.efcoss.newInputVariable("x2", x0[1])
      self.x3 = self.efcoss.newInputVariable("x3", x0[2])
      self.x4 = self.efcoss.newInputVariable("x4", x0[3])
      self.x5 = self.efcoss.newInputVariable("x5", x0[4])
      #Output Definition
      self.fvec = self.efcoss.newOutputVariable("fvec",self.m)

      #Set Calling Sequence
      self.efcoss.setSimulationCallingSequence([self.x1,self.x2,self.x3,self.x4,self.x5,self.fvec,self.m
          ])
      #Set Optimization parameters
      self.efcoss.setOptVars([self.x1,self.x2,self.x3,self.x4,self.x5])
```

Als Zielfunktion wird

$$\min_{\mathbf{x}}(\mathbf{r}(\mathbf{f}(\mathbf{x}))) = \min_{\mathbf{x}} \|\mathbf{f}(\mathbf{x}) - \Phi_{\text{data}}\|$$

genutzt. EFCOSS bietet eine Implementierung der Klasse `DataFit1d` in dem Modul `DataFit` an, welche sich für dieses Problem eignet. Der Nutzer muss nun mit `setObjectiveFunction` eine Methode implementieren, welche die Zielfunktion in EFCOSS entsprechend setzt und die Messdaten Φ_{data} mit Hilfe der Hilfsfunktion `readDataFile1d` aus der Datei `data.txt` einliest. Dabei werden die Inidizes der Messwerte `indices`, die eigentlichen Daten `data` und Gewichte `weights` eingelesen. Diese Werte werden mit der Funktion `addData1d` der Klasse `DataFit1d` zu den erwarteten Daten-werten für die Simulationsergebnisse `fvec` hinzugefügt. Quellcode 4.4 zeigt diesen Code für das exemplarische Beispiel.

Quellcode 4.4: Beispiel für das Setzen der Zielfunktion.

```
class DatafitExample(EFCOSS.ProblemDefinition):
  ...
  def setObjectiveFunction(self):
      obj = self.efcoss.setObjectiveFunction("DataFit", "DataFit1d")
      indices, data, weights = readDataFile1d("data.txt")  # load measured data
      obj.addData1d(self.fvec, indices, data, weights)
```

Nun starten wir die eigentliche Optimierung mit Hilfe des in Quellcode 4.5 gezeigten Codes. Es wird ein Standardoptimierer aus der Scipy.Optimize Optimierungssoftware benutzt, welcher mit der in EFCOSS definierten `opt_scipy` Klasse angesprochen wird. Man übergibt dem Optimierer nur die Referenz auf EFCOSS, definiert die Toleranz und die Grenzen für jeden Parameter und startet dann die Optimierung. Als Ergebnis wird das in Scipy.Optimize genutzte Standardergebnis `OptimizeResult` zurückgegeben. Wir verwenden hier ein beschränktes Optimierungsproblem, für welches die Schranken in den `bounds` zu sehen sind (hier $-10 \leq x_1, x_2, x_3 \leq 10$, $0 \leq x_4, x_5 \leq 1$). Als Toleranz des Optimieres wird $tol = 10^{-10}$ gewählt.

Quellcode 4.5: Beispiel für die Optimierung mit Scipy.

```
class DatafitExample(EFCOSS.ProblemDefinition):
  ...
  def runOptimizer(self):
      from EFCOSS.Optimizers.opt_scipy import *
      x0=self.efcoss.getOptVec()
      opt=opt_scipy(self.efcoss,tol=1e-10, bounds=[(-10, 10), (-10, 10), (-10, 10), (0, 1), (0, 1)])
      return opt.minimize(x0)
```

Klar zu sehen ist, dass die neue Struktur von EFCOSS für den Anwender eine sehr einfache Möglichkeit bietet, seinen Simulationscode mit einem Optimierer zu koppeln. Es sind nur sehr wenige Arbeitsschritte nötig.

4.3 Grundlegende Überlegungen zur Parallelisierung

Zur Lösung der Problemstellungen bietet sich an diversen Stellen eine Parallelisierung an. Zu diesen gehören die parallele Ausführung des Simulationscodes, gegebenenfalls mit weiter verschachtelten Parallelisierungsstufen. Dabei spielen auch die Parallelisierungsparadigmen OpenMP und MPI sowie der geschickte gekoppelte Einsatz der zwei Technologien eine entscheidende Rolle. Darüber hinaus bietet sich auch die Verwendung von verteiltem Rechnen an. Ein Laptop steuert ähnlich wie in den Ausführungen der ursprünglichen EFCOSS Implementierung einen großen Parallelrechner.

Als Alternative und/oder Erweiterung der oben genannten Möglichkeiten kann man auch mehrere Instanzen von EFCOSS parallel zueinander ausführen, um so Parameterstudien effizienter durchzuführen. Als Anwendungsszenario hierfür ist ein OED-Problem mit Parametersensitivitätsanalyse in dieser Arbeit betrachtet worden, welches mittels paralleler EFCOSS-Instanzen die OED-Probleme der Geothermie löst und dazu OpenMP-parallelen Simulationscode einsetzt. Details dazu werden in Kapitel 5 gezeigt.

4.3.1 Parallelisierte Simulationssoftware

Komplexe Simulationssoftware für Anwendungen aus Wissenschaft und Technik benötigt große Rechenzeiten und Speicherplatz. Für sehr große, sogenannte Large-Scale Probleme, werden Simulationen immer häufiger auf großen Parallelrechnern (Hochleistungsrechnern, HPC) durchgeführt. Diese können, sofern skalierbar implementiert, einen deutlichen Leistungssprung in den Anwendungen bieten und damit eine deutlich verkürzte Rechenzeit gegenüber einem einfachen Arbeitsplatzrechner bringen. Zudem können bestimmte Problemgrößen nicht mehr auf einem einzelnen Rechner gerechnet werden, da der Arbeitsspeicher nicht ausreicht. Hier ist der Einsatz von verteiltem Speicher zwingend erforderlich.

Aber auch für kleine und mittelgroße Probleme bietet es sich an, über Parallelisierungen der Simulationscodes nachzudenken. Denn fast jeder verkaufte Laptop und Arbeitsplatzrechner hat heutzutage mindestens zwei Kerne und ist damit in sich schon ein kleiner Parallelrechner, der zum Beispiel über das shared-memory Paradigma OpenMP programmiert werden kann. Alternativ bietet sich hier, wie auch im Rahmen der HPC-Geräte, der Einsatz von Beschleunigerkarten an. Diese dienen der Auslagerung von parallelisierbaren Berechnungen auf eine Erweiterungskarte, zum Beispiel eine Grafikkarte.

EFCOSS zeigt hier seine Stärke, da es sehr eng mit der parallelisierten Simulationssoftware verbunden ist. Das heißt, es wird die Lösung von Optimierungsproblemen unterstützt, wobei die Simulationssoftware eines oder beide der parallelen Programmierparadigmen – shared-memory und/oder distributed-memory – nutzt. In diesem Abschnitt wollen wir die Integration von EFCOSS mit OpenMP- und MPI-parallelisierten Simulationscodes zeigen.

4.3.1.1 OpenMP

Die beiden OpenMP-Codes aus Abschnitt 3.4, Quellcode 3.5 und Quellcode 3.6 werden in jeweils einer Datei gespeichert, `sim.f90` beinhaltet den Simulationscode, `sim_dv.f90` den abgeleiteten Code. Am Simulationsinterface `simIF.f90` ändert sich nichts. In EFCOSS wird dieser Code nun mittels der folgenden Kommandos eingebunden:

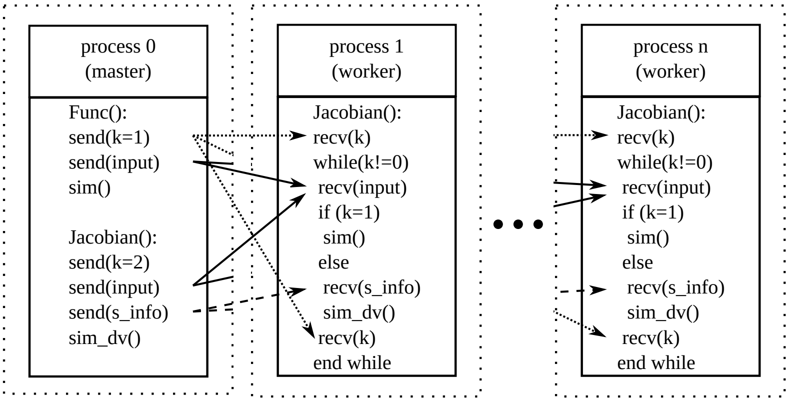
1. `gfortran -c -fopenmp -fPIC sim.f90 sim_dv.f90`
2. `ar -r libsim.a sim.o sim_dv.o`
3. `f2py -f90flags='-fopenmp' -c -m sim simIF.f90 -L. -lsim -lgomp`

Hierbei werden unter 1. die Simulationscodes, sowie deren Ableitung in Object-Files compiliert. Wichtig hierbei ist die Option `-fopenmp`, welche beim Compilieren die OpenMP Pragmas und Funktionsaufrufe berücksichtigt. Als zweites wird eine Bibliothek namens `libsim.a` erstellt. Im letzten Schritt werden mittels f2py die Interfaces compiliert und mit der eben erstellten Bibliothek und OpenMP gelinkt. Wichtig ist hierbei das `-f90flags='-fopenmp'`. Nun kann die Funktion wie gehabt mittels EFCOSS verwendet werden. Diese Teile werden exakt so in der Methode `compileSimulationInterfaceFortran` der Klasse `ProblemDefinition` umgesetzt, wenn als Übergabeparameter der Methode `generateSimulationInterfaceFortran` der Parameter `useOMP=True` gesetzt ist.

4.3.1.2 MPI

Da OpenMP beschränkt in den Einsatzmöglichkeiten für Simulationen ist, besonders die Skalierbarkeit ist hier ein Problem, bieten sich andere Parallelisierungsmöglichkeiten an. In dieser Arbeit

Abbildung 4.5 Benutzung eines Fortran Simulationsinterfaces in Verbindung mit MPI.



wird als wichtigste Alternative das Message Passing Interface (MPI) genutzt. Hierzu müssen wir uns zu erst über die grundlegende Frage Gedanken machen, wie im allgemeinen Simulationen, die mit MPI parallelisiert wurden, arbeiten. Das wohl bekannteste Prinzip ist ein Master-Worker Prinzip. Ein Prozess, der Master, verteilt eine zu bearbeitende Aufgabe an verschiedene Arbeitsprozesse, die Worker.

Das in Kapitel 2.2.1 beschriebene Beispiel des Parameterschätzproblems aus Quellcode 2.1 kann relativ einfach in MPI implementiert werden. Jeder MPI-Prozess kann einen lokalen Teil der Berechnungen unabhängig von den anderen Prozessen durchführen. Kommunikation ist hierbei nur am Anfang und Ende notwendig. Am Anfang muss der aktuelle Zustand allen teilnehmenden Prozessen mitgeteilt werden, am Ende müssen die einzelnen Teile des Arrays zusammengefügt und ausgegeben werden.

Um den so parallelisierten Simulationscode mit EFCOSS zu nutzen, müssen neue Interfaces erzeugt werden, die dann in einer anderen Simulations-Klasse `SimulationMPI` in der `initSim()` Methode aufgerufen werden. Zur Generierung der neuen Interfaces muss die `generateServerIF()` Methode um den Übergabeparameter `useMPI=1` ergänzt werden.

Um dies zu bewerkstelligen, muss der Optimierungsprozess auf einem ausgewählten Knoten, dem Master (MPI-Rank 0), ausgeführt werden. Die anderen MPI Prozesse werden als Worker eingesetzt. Diese erzeugen eine einfache EFCOSS Instanz und führen die Funktion `run_function_worker()` aus. Der korrespondierende Code findet sich in Quellcode 4.6. Der Master-Prozess muss die Methode `run_optimizer()` ausführen.

Quellcode 4.6: Ausführungscode der Simulation für einen Worker.

```
def run_function_worker(self):
  x=getInitialValues()
  res=self.efcoss.evalfjac(5,x,33)
```

Um die Wirkungsweise der Interfaces für Master und Worker Prozesse anschaulich darzustellen, bietet sich Abb. 4.5 an. Der Master Prozess, in welchem auch der Optimierungsalgorithmus läuft, bietet die Methoden `Func()` und `Jacobian()`, welche von den Optimierungsalgorithmen aufgerufen werden. Vergleiche hierzu Abb. 4.3. Der Optimierer berechnet hiermit den neuen Parameter-Vektor x. Zusätzlich ist der Master verantwortlich, die Worker entsprechend zu steuern. Hierfür wurde eine einfache Kommandostruktur eingeführt, welche die Worker über einfache Nachrichten mit der Variablen k steuert. Der gesendete Wert für k ist dabei der Tabelle 4.1 zu entnehmen.

Tabelle 4.1 Steuerparameter für die Simulation mit MPI

k	Operation
0	Beenden
1	Funktionsauswertung
2	Ableitungsauswertung

Der Ablauf eines Optimierer-Laufes aus Sicht des Masters sieht dann folgendermaßen aus:

1. Initialisiere EFCOSS
2. Initialisierung der Optimierung mit x_0 als Startwert
3. Wiederhole bis Optimum gefunden, starte mit $i = 0$:
    (a) Berechne Funktionswert:
        i. Sende `k=1` an Worker
        ii. Sende x_i an Worker
        iii. Starte Simulation
    (b) Berechne Ableitung:
        i. Sende `k=2` an Worker
        ii. Sende x_i an Worker
        iii. Sende Seed Matrix S an Worker
        iv. Starte Jacobi-Berechnung
    (c) Synchronisiere und Berechne neues x_{i+1}
4. Beenden
    (a) Sende $k = 0$ an Worker
    (b) Gib Ergebnis aus

Ein Worker Prozess startet mit der `evalfjac()` Methode der EFCOSS Klasse. Das bedeutet, es wird der Interfacewrapper `Jacobian()` gestartet. In dieser Interface-Methode verbleibt der Worker, bis er vom Master beendet wird:

1. Initialisiere EFCOSS
2. Starte `run_function_worker()`
3. Wiederhole
    (a) Empfange k
    (b) Falls $k == 1$
        i. Empfange x
        ii. Starte Simulation
    (c) Falls $k == 2$
        i. Empfange x
        ii. Empfange Seed-Matrix S
        iii. Starte Jacobi-Berechnung
    (d) Falls $k == 0$
        i. Beende Simulation

Dabei muss der Benutzer die Master- und Worker-Prozesse entsprechend starten. Ein exemplarisches Laufzeitscript ist in Quellcode 4.7 zu sehen. Zuerst wird die MPI-Umgebung aufgebaut und der entsprechende MPI-Rang des Prozesses abgerufen. Abhängig von dem Rang des Prozesses startet dieser entweder als Master ($rank == 0$) oder als Worker ($rank > 0$). Dann wird EFCOSS initialisiert und die Simulationsumgebung gesetzt. Der Master startet dann den Optimierungsprozess, die Worker starten die Funktion `run_function_worker()`. Nachdem die Optimierung abgeschlossen ist, muss zwangsläufig die `Finalize` Methode der `SimulationMPI` Klasse ausgeführt werden. Diese sendet $k = 0$ und beendet somit alle MPI Prozesse.

Quellcode 4.7: Laufzeitscript für die Lösung eines Optimierungsproblems mit MPI-paralleler Simulation.

```python
from OptimizeSim import OptimizeSim
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank=comm.Get_rank()

d=OptimizeSim()
d.initEFCOSS()
d.initSim()

if (rank==0):
   d.run_optimization()
   d.opt.Simulation.Finalize()
else:
   d.run_function_worker()
```

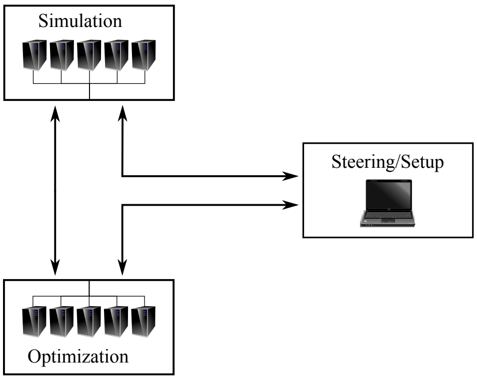
4.3.2 Remote Simulation

Eine Simulations- oder Optimierungssoftware wird meist für eine bestimmte High-Performance Plattform entwickelt, auf welcher sie besonders gut läuft. Im Allgemeinen sind diese hoch spezialisierten Optimierungs- und Simulationspakete nicht auf allen Plattformen verfügbar.

Aus diesem Grund ist die verteilte Abarbeitung der Aufgaben wünschenswert. Dabei sind die folgenden drei Komponenten herauszustellen:

1. Dedizierte Simulations-Workstation oder Cluster
2. Dedizierte Optimierungs-Workstation oder Cluster
3. Dedizierte Steuerungs-Workstation

Abbildung 4.6 Prinzip der Remote-Objekte zur Illustration der Möglichkeiten des verteilten Rechnens mit EFCOSS (Cliparts von openclipart.org.)



Dieses Konzept ist in Abb. 4.6 dargestellt. In dieser Ausarbeitung steht die parallele Simulation im Vordergrund. Dies ist im Wesentlichen damit zu begründen, dass die Gesamtlaufzeit der Optimierung meist durch die zeitintensive Berechnung der Simulationssoftware und deren Ableitungen dominiert wird. Die Optimierung kann zwar auch parallelisiert werden, zeigt in diesem Szenario jedoch keine signifikante Performancesteigerung. Die verteilte Anwendung sieht somit folgendermaßen aus. Auf einem Laptop oder einer Workstation wird mit Hilfe von Steuerungs- und Setupscripten eine Optimierungssoftware (lokal) gestartet. Auf einem entfernten Server wird ein mit MPI parallelisierter Simulationscode, ähnlich der in Abschnitt 4.3.1.2 ausgeführten Details, gestartet.

Um die Verteilung der Anwendung möglichst einfach zu gestalten, wurde darauf geachtet, eine reine Python Lösung zu erstellen. Zu diesem Zweck nutzen wir das Paket *Python Remote Object* (kurz PyRO) in der Version 4.62 [57].

Für den Einsatz von verteilten Systemen ist eine Versendung der Daten über das Netzwerk unumgänglich. Dabei müssen die Daten mittels eines Serialisierers bearbeitet werden. Der Standardserialisierer von PyRO seit Version 4.18, `serpent`, unterstützt keine komplexen Datentypen, wie sie zum Beispiel von Numpy bereitgestellt werden. Hier empfiehlt die Dokumentation von PyRO [58] eine der folgenden drei Vorgehensweisen:

1. Benutze keine Numpy Datentypen
2. Benutze keine Arrays als Rückgabeparameter
3. Nutze `pickle` oder `dill` als Serialisierer

Dabei sind die ersten zwei genannten Möglichkeiten für uns jedoch inakzeptabel. Die Rückgabe von Arrays ist essentiell für die Ausführung und die Benutzung von Numpy Datentypen wurde als wesentlicher Schritt zur Vereinfachung der EFCOSS Struktur eingeführt.

Der letzte Punkt, einen anderen Serialisierer zu verwenden, erscheint somit unumgänglich. Dabei ist jedoch zu beachten, dass die Nutzung von `pickle` oder der etwas verbesserten Implementierung `dill` ein gewisses Sicherheitsrisiko birgt [59]. Deshalb sollte man möglichst auf diese Serialisierer verzichten, da es die Sicherheit innerhalb des Netzwerks reduziert. Die verteilte Ausführung von EFCOSS mit PyRO sollte somit nur in Umgebungen mit verstärkter Sicherheit durchgeführt werden.

Die Einstellung, um PyRO mit `pickle` zu verwenden, ist in Quellcode 4.8 zu sehen. Weiterhin wird in diesem Codeabschnitt ein MPI basierter Simulationsserver gezeigt. Dazu muss ein PyRO Daemon auf dem Master-Prozess ($rank = 0$) gestartet werden, die restlichen MPI Prozesse initialisieren die `OptimizeSim` Klasse wie bisher und starten dann die Laufzeitschleife mit `run_function_worker()`.

Quellcode 4.8: Nutzung einer MPI Simulation mit PyRo.

```
from simulation import SimulationMPI
from OptimizeSim import OptimizeSim
import Pyro4
from mpi4py import MPI
Pyro4.config.SERIALIZER='pickle'
Pyro4.config.SERIALIZERS_ACCEPTED.add(
   'pickle')

def main():
   comm=MPI.COMM_WORLD
```

```
   rank=comm.Get_rank()
   size=comm.Get_size()
   if (rank==0):
     simulation=SimulationMPI(modname="sim")
     Pyro4.Daemon.serveSimple(
       {simulation:
         "efcoss.Simulation"},
       ns=False)
   else:
      d=OptimizeSim()
      d.initEFCOSS()
      d.initSim()
      d.run_function_worker()

if __name__=="__main__":
   main()
```

Auf der Client-Seite muss im Ausführungsscript nur die `initSim()` Methode, wie in Quellcode 4.9 zu sehen, verändert werden. Hier muss eine Verbindung zu dem Server aufgebaut werden. Dazu muss die vom Simulationsserver ausgegebene URI dem Clientscript übergeben werden. Danach kann das Objekt entsprechend genutzt werden.

Quellcode 4.9: Python Code, um ein entferntes Objekt vom Simulationsserver zu holen.

```
def initSimRemote(self):
    uri=input("Enter URI of Simulation Server: ").strip()
    sim=Pyro4.Proxy(uri)
    sim.setResultVec(self.opt.getResultVec())
    sim.setJacVec(self.opt.getJacVec())
    self.opt.setSimulationServer(sim)
```

4.4 Überblick über die Module

Nach diesem einfachen Beispiel sollten wir uns die hinter diesem simplen Benutzerinterface stehenden Mechaniken ansehen. Bevor das Toplevel Modul `EFCOSS` beschrieben wird, sollten wir jedoch kurz die `ProblemDefinition` besprechen, damit das im vorherigen Abschnitt besprochene Beispiel klarer wird. Im Anschluss werden wir uns die einzelnen Module im Detail ansehen.

4.4.1 Die Klasse ProblemDefinition

Die Klasse `ProblemDefintion` ist eine Basisklasse für verschiedene Probleme, die mit EFCOSS gelöst werden können. Sie ist das Interface, welches der Benutzer sieht und erweitert. Deshalb sollte es möglichst einfach sein, es zu verstehen und seine Mächtigkeit zu nutzen.

Die Initialisierungsroutine ist in Quellcode 4.10 zu sehen. Hier wird EFCOSS instanziiert und global unter Verwendung der Hilfsfunktion `getEFCOSSRef` verfügbar gemacht. Somit kann man diese EFCOSS Instanz zum Beispiel bei der Generierung der Interfaces direkt nutzen.

Quellcode 4.10: Die Intialisierungsmethode der `ProblemDefintion` Klasse.

```
class ProblemDefinition:
    def __init__(self,name,siminstances=1,objectives=1,constraints=1,matfree=False,logging=1):
```

```
        self.name=name
        self.efcoss=EFCOSS.EFCOSS(name,logging,siminstances,objectives,constraints,matfree)
        getEFCOSSRef(self.efcoss)
```

Um eine Simulation an EFCOSS zu koppeln, bietet die `ProblemDefinition` Klasse eine Funktion zum Generieren von Interfaces `generateSimulationInterface`, welche in Quellcode 4.11 abgebildet ist. Mit Hilfe der Übergabeparameter wie der Programmiersprache, des ADTools und des ADLevels wird dann eine Sprachbeschreibung `LanguageDescription` und eine Ableitungsbeschreibung `ADDescription` erstellt und mit diesen ein Simulationsinterface mit dem Dateinamen `ifname` und der Toplevelroutine `toplevel` erzeugt. Der Parameter `objectiveinstance` gibt bei mehreren Zielfunktionen diejenige an, für welche das Interface generiert werden soll.

Quellcode 4.11: Methode zum Generieren von Simulations-Interfaces.

```
class ProblemDefinition:
    ...
    def generateSimulationInterface(self,language="fortran",adtool="",adlevel=0,adreverse=False,useOMP=
     False,useMPI=False,toplevel=None,ifname=None,objectiveinstance=0):
        extensions={"fortran":"f90","c":"c","c++":"cpp"}
        if (toplevel==None):
            toplevel=self.name
        if (ifname==None):
            ifname = self.name+"IF"+extensions[language.lower()]
        self.toplevel=toplevel
        self.ifname=ifname
        self.languageDescription = LanguageDescription(language.lower(),useMPI)
        self.adDescription = ADDescription(adtool,adlevel,adreverse)
        generateSimulationIF(toplevel, ifname,self.languageDescription,self.adDescription,
             objectiveinstance)
```

Nachdem im letzten Schritt Programmcode für das Interface erzeugt wurde, kann dieser nun compiliert werden. Wir bieten in Quellcode 4.12 einen einfachen Wrapper für F2PY an, bei welchem die entsprechenden Bibliotheken für MPI und OpenMP, sowie für den AD Reversemode für Tapenade gesetzt werden. F2PY erzeugt ein Simulationsinterface als shared Object (Endung .so) mit dem Namen `toplevel`. Für die MPI Implementierung muss zusätzlich der MPI Compiler `mpif90` gesetzt werden.

Quellcode 4.12: Methode zum Compilieren der Fortran-Interfaces mittels F2PY.

```
class ProblemDefinition:
    ...
    def compileSimulationInterfaceFortran(self):
        extraArgs=" "
        if (self.adDescription.reversemode and self.adDescription.adtool=="tapenade"):
            extraArgs+=" -L$(EFCOSS_DIR)/lib -ladHelp "
        if (self.languageDescription.mpi):
            extraArgs+=" --f90exec= mpif90 "
            extraArgs+=" -lmpi "
        if (self.languageDescription.omp):
            extraArgs+=" -lgomp "
        f2py.compile(open(self.ifname).read(),self.toplevel,extraArgs=extraArgs+" -L. -l"+self.name,
             extension='.f90')
```

Nun muss das soeben gebaute Interface an EFCOSS angebunden werden. Dies geschieht in Quellcode 4.13. Es wird eine neue Simulationsinstanz mit dem Namen `toplevel` erzeugt und EFCOSS bereit gestellt.

Quellcode 4.13: Definieren der Simulation in EFCOSS.

```
def initSimulationInterface(self,instance=0):
    sim=EFCOSS.Simulation(self.efcoss,self.toplevel)
    self.efcoss.setSimulationServer(sim,instance)
```

4.4.2 Das Toplevel Modul EFCOSS

Die EFCOSS Klasse dient uns als Bindeglied zwischen den Optimierern und dem Simulationscode. Sie bietet Interfaces für verschiedene Evaluierungsfunktionen, die von Optimierern aufgerufen werden, sowie deren Ableitungen.

Weiterhin wird eine umfangreiche Initialisierungsroutine benötigt, welche die neuen Funktionen effizient umsetzt. Quellcode 4.14 zeigt die Struktur dieser Initialisierung. Im einfachsten Fall sollte nur ein Name für die Logfiles übergeben werden. Zusätzlich können in Abhängigkeit der Problembeschreibung die Anzahl der Simulationen, der Zielfunktionen, sowie der Nebenbedingungen definiert werden.

Quellcode 4.14: Initialisierungsroutine für eine EFCOSS Instanz.

```
class EFCOSS:
  def __init__(self, name, logs=0, simulationinstances=1, objectivefunctions=1, constraintfunctions=1,
       matfree=False):
      ...
```

Einige der Methoden, die EFCOSS bietet, haben wir bereits am einfachen Beispiel gesehen. Zu diesen gehören `newInputVariable`, `newOutputVariable`, `setSimulationCallingSequence`, `setOptVars`, sowie `setObjectiveFunction`.

Folgende Methoden werden dem Optimierer zur Verfügung gestellt:

- Funktionsauswertung
  - `evalf` Auswertung der skalaren Zielfunktion
  - `evalfvec` Auswertung der vektoriellen Zielfunktion
- Constraintsauswertung
  - `evalceq` Auswertung der Gleichheitsnebenbedingungen
  - `evalcineq` Auswertung der Ungleichheitsnebenbedingungen
- Erste Ableitung
  - `evalgf` Auswertung des Gradienten der Funktion
  - `evalfjac` Auswertung der Jacobi-Matrix
  - `evaljacvec` Auswertung der Jacobi-Vektor-Funktion mit Forward-Mode
  - `evaljactvec` Auswertung der Jacobi-Vektor-Funktion mit Reverse-Mode

Als Beispiele sollen uns die Methoden `evalfvec`, aus Quellcode 4.15, und `evalfjac`, aus Quellcode 4.16 dienen. Beiden Methoden ist die gleiche Signatur anzusehen. Die Eingaben sind `n`, die Anzahl der Eingabe-Elemente, der Eingabe-Vektor `x` sowie die erwartete Größe des Ausgabe-Vektors in `m`. Die optionalen Parameter `simulationInstance` und `objectiveFunctionInstance` geben die Instanzennummer der zu verwendenden Simulation, bzw. der Zielfunktion, an. Dies wird für die in Abschnitt 4.5 gezeigten Problemstellungen benötigt, bei denen sowohl mehrere Simulationsfunktionen, als auch mehrere Zielfunktionen eine Rolle spielen.

Quellcode 4.15: Methode der EFCOSS Klasse um den Funktionswert einer vektoriellen Zielfunktion auszuwerten.

```
def evalfvec(self, n, x, m, simulationInstance=0, objectiveFunctionInstance=0):
    try:
        solution_vector = self.CurrentObjectiveFunction[objectiveFunctionInstance].vectorfunction(x,
             m, simulationInstance)
    except:
        print "Error while evaluating function", sys.exc_info()[0]
        raise
    if len(solution_vector) != m:
        print "Wrong dimension of solution in evalfvec"
        raise
    self.CurrentObjectiveFunction[objectiveFunctionInstance].log_result(x, solution_vector)
    return solution_vector
```

Quellcode 4.16: Methode der EFCOSS Klasse um die Jacobi-Matrix einer Zielfunktion zu bestimmen.

```
def evalfjac(self, n, x, m, simulationInstance=0, objectiveFunctionInstance=0):
    try:
        jac, res = self.CurrentObjectiveFunction[objectiveFunctionInstance].jacobian(x, m,
            simulationInstance)
    except:
        print "Error while evaluating Jacobian", sys.exc_info()[0]
        raise
    if len(jac) != m:
        print "Wrong dimension of solution in evalfjac"
        raise
    return jac, res
```

4.4.3 Das Simulation Modul

Im `Simulation` Modul finden sich die Definitionen der Klassen für die Simulation, der Input- und Outputvariablen, der Pufferspeicher für bereits berechnete Simulationsergebnisse, sowie das `Codegenerator` Untermodul.

Variablen Klassen Als Definition für die Variablen wurden die unter dem alten Design verwendeten, mit CORBA definierten, Datentypen durch reine Python-Datentypen ersetzt. Unsere Wahl fiel dabei auf die im Numpy (numerical Python) Paket [60] definierten Datentypen, basierend auf einem n-dimensionalen Array `ndarray`. Zuerst wird eine Basisklasse `VariableClass` von `ndarray` abgeleitet, vgl. Quellcode 4.17. Diese muss grundlegende Funktionen von `ndarray` überschreiben, um für unsere Zwecke zu funktionieren. Dabei sind die Methoden `__array_finalize__` und `__array_wrap__` besonders zu nennen, da diese ein Arbeiten mit den so erstellten Variablen ohne vorhergehende Konversion erlaubt. Die in EFCOSS eingesetzten Datentypen für die Arbeit mit Simulationen sind `int` und `float64`, welche in `numpy` definiert sind. Andere Datentypen werden nicht unterstützt und gegebenenfalls in die erlaubten umgewandelt.

Als Erweiterung zum `ndarray` werden drei neue Attribute eingeführt:

- `efcoss`: Referenz zur EFCOSS Instanz
- `id`: Ein String zur Identifizierung der Variable
- `shapestring`: Ein String zur Speicherung der Struktur der Variable für die Interfacegeneratoren

Zum Setzen des `shapestring` wird über die eingebaute Funktion `hasattr()` überprüft, ob der übergebene `shape` einen Identifier `id` besitzt. Wenn dem so ist, wird im weiteren Verlauf das Attribut `shapestring` mit der `id` besetzt, also dem Namen der anderen Variablen. Ansonsten wird der `shapestring` mit dem eigentlichen `shape` des Objektes als String besetzt. Dieser String wird für die Interfacegeneratoren benötigt, welche im späteren Verlauf des Kapitels beschrieben werden.

Quellcode 4.17: Variablen Basisklasse für EFCOSS.

```
class VariableClass(ndarray):
   def __new__(cls,shape,dtype=float64,id="",e=None):
      obj = ndarray.__new__(cls,shape,dtype)
      obj._set_efcoss(efcoss)
      obj._set_id(id)
      obj._set_shapestring(shape)
      return obj

   def __array_finalize__(self,obj):
      if obj is None: return
      self.efcoss = getattr(obj,’efcoss’,None);
      self.id = getattr(obj,’id’,None);

   def __array_wrap__(self,arr,context=None):
      return ndarray.__array_wrap__(self,arr,context)

   def _get_id(self):
       return self.id

   def _set_id(self,id):
       self.id=id

   def _set_efcoss(self,efcoss):
       self.efcoss=efcoss

   def _set_shapestring(self,shape=None):
       tempShape=shape
       if (shape==None):
          tempShape = shape if self.shape!=() else []
       if not isinstance(tempShape, list):
           tempShape = [tempShape]
       print self.shape
       self.shapestring = map(lambda s: str(s.id) if hasattr(s, ’id’) else str(s), tempShape)
```

Die Inputvariablen `InputVariableClass` und Outputvariablen `OutputVariableClass` werden in Quellcode 4.18 beschrieben. Diese leiten sich beide von `VariableClass` ab. Bei den Inputvariablen wird ein übergebenes Array (oder ein skalarer Wert) mittels `asarray()` und `view()` als `InputVariableClass` festgelegt. Die Outputvariablen sind nur eine Dummy-Klasse und dienen der eindeutigen Unterscheidbarkeit der Variablen.

Quellcode 4.18: Klassen für Input- und Outputvariablen in EFCOSS.

```
class InputVariableClass(VariableClass):
   def __new__(cls,input_array,efcoss=None,id=""):
      obj = asarray(input_array).view(cls)
      obj._set_efcoss(efcoss)
      obj._set_id(id)
      obj._set_shapestring()
      return obj

   def _get_defaultvalue(self):
       return self

class OutputVariableClass(VariableClass):
   pass
```

Quellcode 4.19 zeigt die in EFCOSS verwendete Klasse `VariableFactory`, welche eine Referenz an EFCOSS hält und die beiden Methode `create_input_variable` und `create_output_variable` bereitstellt.

Quellcode 4.19: `VariableFactory` Klasse für EFCOSS.

```
class VariableFactory:
    def __init__(self,efcoss):
        self.efcoss = efcoss

    def create_input_variable(self,varname, input_array):
        vi = InputVariableClass(input_array,id=varname,efcoss=self.efcoss)
        return vi

    def create_output_variable(self, varname, vartype, shape):
        vi = OutputVariableClass(shape=shape,dtype=vartype,id=varname,efcoss=self.efcoss);
        return vi
```

Zum Aufruf der Simulation werden die Eingaben in einer einzelnen Variablen `x` gespeichert. Hierbei muss jedoch zwischen den Datentypen Integer (`int`) und Double (`float64`) unterschieden werden, da in unserem Kontext eine Optimierung nur auf Floating-Point Datentypen sinnvoll ist. Aus diesem Grund wurde eine einfache Klasse `InputValues` definiert (vgl. Quellcode 4.20), welche diese Unterscheidung durchführt und die Datentypen entsprechend setzt. Sie bekommt jeweils ein Array mit den in dem Simulationsaufruf definierten Eingabewerten, mit `i` als Integer- und `d` als Doublewerten. Diese Unterscheidung ist notwendig, da Fortran und insbesondere die Verbindung zwischen Fortran und Python keine gemischten Datentypen in Form von Strukturen oder ähnlichem erlaubt. Außerdem wurde für Logging und Debugging eine einfache Funktion `__str__()` implementiert, welche einen String mit den gespeicherten Werten ausgibt.

Quellcode 4.20: Klasse für die Variablenfabrik in EFCOSS.

```
class InputValues:
  def __init__(self,i,d):
     self.i=i.astype(int)
     self.d=d.astype(float64)

  def __str__(self):
      return "Inputvalues=[integer="+str(self.i)+" float="+str(self.d)+"]"
```

Simulations Klasse Als Interface zwischen EFCOSS und der Simulation wurde eine Basisklasse geschrieben, welche als Stumpf (engl. Stub) für die in nativem Python, Fortran oder C/C++ geschriebenen Simulationsmodelle dient. Diese ist in Auszügen in Quellcode 4.21 zu sehen. Sie bietet Methoden für die Funktionsauswertung der Simulation `Function`, sowie deren Ableitung. Für die erste Ableitung bietet die Routine die Berechnung der vollen Jacobi-Matrix `JacobianAD`, sowie Jacobi-mal-Vektor `JacobianVector` und Jacobi-Transponiert-mal-Vektor `JacobianTVector` an. Für die zweite Ableitung finden sich die Hessematrix und die projizierte Hessematrix. In den einzelnen Funktionen werden die Funktionsrümpfe `func`, `jac`, `jacvec` und `jactvec`, sowie `hes` und `phes` genutzt. Diese müssen vorher in der eigentlichen Simulations-Klasse gesetzt werden. Der Eingabe-Vektor `x` ist dabei vom Typ `InputValues` und wird entsprechend für die Übergabe an die Simulation in den Integer- und den Floating-Point-Teil aufgespalten. Das ist nötig, da üblicherweise keine komplexen Strukturen an Fortran übergeben werden können. Für den Einsatz der Simulations-Basisklasse direkt in einer Python Simulation wirkt dies nicht hinderlich, muss jedoch beim Implementieren von Interfaces beachtet werden.

Quellcode 4.21: Simulations-Basisklasse für EFCOSS.

```
class SimulationBase:

   def Function(self,x):
      self.func(x.i,x.d,self.result)
      return self.result

   def JacobianAD(self,x,S):
      self.jac(x.i,x.d,S,self.result,self.jacVec)
      return (self.result,self.jacVec)

   def JacobianVector(self,x,vec,S):
      ...

   def JacobianTVector(self, x, vec, S):
       outvec = zeros(S[0])
       if (self.useFullJacobi):
           jac, tmp = self.Jacobian(x, S)
           outvec = jac.T * vec
       else:
           self.jactvec(x.i, x.d, vec, self.result, outvec)
       return outvec, self.result
```

Die in Quellcode 4.22 gezeigte Klasse `Simulation` ist die Schnittstelle zwischen EFCOSS und den Simulationsinterfaces. Die Klasse ist so gestaltet, dass man diese auch ohne EFCOSS Instanz benutzen kann. Das heißt, man kann damit die Funktionen einer in Fortran, C oder C++ geschriebenen Simulation in Python nutzbar machen. Wir wollen hier aber die Anbindung an EFCOSS zeigen, somit werden zwei wichtige Übergaben benötigt:

1. Eine EFCOSS Instanz `efcoss`, sowie
2. der Modulname der extern zur Verfügung gestellten Simulation `modname`.

Die anderen Übergabeparameter sind wie folgt:

- `useDD` - Nutze finite Differenzen (FD)
- `compare` - Vergleich zwischen AD und FD

- `simulationInstance` - Instanznummer der Simulation
- `objectiveInstance` - Für welche Zielfunktion ist die Simulation gedacht

Die ersten beiden Parameter werden benutzt, falls entweder kein AD Code zur Verfügung steht, oder ein Vergleich zwischen AD und FD gewünscht ist. Die beiden Letzteren werden für Multi-EFCOSS genutzt und sollen an dieser Stelle nur der Vollständigkeit halber genannt sein.

Die Klasse bietet nach außen die Methoden

1. `Function`
2. `Jacobian`
3. `JacobianVector`
4. `JacobianTVector`
5. `Hessian`
6. `ProjectedHessian`

Diese werden in der Initialisierungsmethode gesetzt. Standardmäßig werden die Methoden für die AD abgeleiteten Simulationsinterfaces genutzt. Dabei wird jedoch in der Methode `loadModule()`, welche das Laden der Simulationsinterfaces aus dem mit F2PY generierten Shared-Object aus dem Modul mit dem Namen `modname` übernimmt, geprüft, ob die gegebenen Funktionen vorhanden sind. Dies ist nötig, da die (höheren) Ableitungen gegebenenfalls nicht mit AD generiert werden können, beziehungsweise der Benutzer diese nicht nutzen möchte.

Quellcode 4.22: Simulations-Klasse für EFCOSS.

```python
class Simulation(SimulationBase):
   def __init__(self,efcoss=None,modname=None,useDD=False,compare=False,simulationInstance=0,
        objectiveFunctionInstance=0):
      self.efcoss=efcoss
      self.useDD=useDD
      self.compare=compare
      self.Jacobian=self.JacobianAD
      self.Hessian=self.HessianAD
      self.ProjectedHessian=self.ProjectedHessianAD
      self.instance=simulationInstance
      self.objective=objectiveFunctionInstance
      if (efcoss!=None):
         self.setResultVec(self.efcoss.getResultVec())
         self.setJacVec(self.efcoss.getJacVec(self.instance,self.objective))
      else:
         self.result=None
         self.jacVec=None
      self.loadModule(modname)


   def setResultVec(self,res):
      self.result=res

   def setJacVec(self,jac):
      self.jacVec=jac

   def loadModule(self,modname):
      self.func=self.setSimulationObject(modname,"func")
      self.jac=self.setSimulationObject(modname,"jacobian")
      self.jacvec = self.setSimulationObject(modname, "jacobianvector")
```

```
        self.jactvec = self.setSimulationObject(modname, "jacobiantvector")
        self.hes=self.setSimulationObject(modname,"hessian")
        self.phes=self.setSimulationObject(modname,"projectedhessian")
        if self.jac==None or self.useDD:
            self.Jacobian=self.JacobianDD
        if self.hes==None or self.useDD:
            self.Hessian=self.HessianDD
        if self.phes==None or self.useDD:
            self.ProjectedHessian=self.ProjectedHessianDD
        if self.compare:
            self.Jacobian=self.compareAD_DD
```

In der `loadModule` Methode wird die Funktion `setSimulationObject` genutzt, welche in Quellcode 4.23 gezeigt ist. Diese nutzt die `classloader` Klasse aus `EFCOSS.Utilities`, um die Funktionsstubs `func`, `jac`, `jacvec`, ... zu setzen. Falls die Routine `None` zurück gibt, ist die gewünschte (AD-)Funktion nicht verfügbar und es sollten entsprechend finite Differenzen in der Optimierung genutzt werden.

Quellcode 4.23: Simulations-Klasse für EFCOSS.

```
    def setSimulationObject(self,modname,classname):
      if modname is None:
         return None
      aFullClassName=modname+'.'+classname
      print "Loading objective class:",aFullClassName
      try:
         function= classloader._get_func(aFullClassName)
      except:
         print "Could not load ",aFullClassName
         function=None
      return function
```

Die zur Verfügung stehenden finite Differenzen Funktionen sind in Quellcode 4.24 gezeigt. Es wird eine Funktion `perturb` geboten, welche einen Eintrag `i` des Vektors `x` um `h` stört. Dabei wird ein Vektor `S` genutzt, welcher die Indizes der abzuleitenden Variablen in `x.d` beinhaltet. Für jeden Eintrag $s_i > -1$ des Vektors `S` wird die Ableitungsfunktion

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}_i} = \frac{\mathbf{f}(\mathbf{x}_i + h\mathbf{e}_i) - \mathbf{f}(\mathbf{x})}{h}$$

berechnet und im Ausgabevektor `jacVec` gespeichert. Zusätzlich kann mit der Variablen `compare` der Initialisierungsroutine der Vergleich zwischen AD und FD gestartet werden. Dies führt `JacobianAD` und `JacobianDD` nacheinander aus, vergleicht die Ergebnisse und gibt diese aus.

Quellcode 4.24: Finite Differenzen Modul der Simulations-Basisklasse für EFCOSS.

```
def perturb(x, i, h=1.5e-8):
    x[i] = x[i] + h
    return x

class SimulationBase:
   ...
   def JacobianDD(self,x,S):
       import variables
       self.Function(x)
```

```
       fun=array([f for f in self.result])
       i=1
       self.jacVec=self.efcoss.getJacVec()
       h=1.5e-8
       k=0
       for i,s in enumerate(S[1:]):
           newx=x.d.copy()
           if (s!=-1):
              newx=perturb(newx,i,h)
              pfun=self.Function(variables.InputValues(x.i, newx))
              self.jacVec[k]=(pfun-fun)/h
           k=k+1
       return (self.result,self.jacVec)


   def compareAD_DD(self,x,S):
       f,ad=self.JacobianAD(x,S)
       f,dd=self.JacobianDD(x,S)
       print "comparison\n==========="
       print "ad=",ad
       print "dd=",dd
       print "ad-dd",ad-dd
       return (f,ad)
```

Buffer Klasse Die `Buffer` Klasse wird dazu verwendet, einmal berechnete Ergebnisse der Simulation zu speichern und später wieder darauf zugreifen zu können. Dies ist insbesondere dann sinnvoll, wenn die Ausführung des Simulationscodes lange dauert und der Optimierer mehrfach den gleichen Wert abfragt. Der zugrundeliegende Code findet sich in Quellcode 4.25.

Quellcode 4.25: Buffer-Klasse für EFCOSS.

```
class Buffer:
    def __init__(self,enabled=True):
        self.kv = {}
        self.enabled=enabled

    def set(self,key,value):
        if self.enabled:
            if isinstance(key,InputValues):
                self.kv[hashable(key.d)]=value
            else:
                self.kv[hashable(array(key))]=value

    def lookup(self,key):
        if self.enabled:
            if isinstance(key,InputValues):
                k=hashable(key.d)
            else:
                k=hashable(array(key))
            if (k in self.kv.keys()):
                return self.kv[k]
        return None
```

Die Klasse benötigt zwei Attribute, zum einen ein Key-Value-Paar `kv` und ob wir die `Buffer` Funktionalität verwenden möchten. Dies kann mittels der Initialisierungsvariablen `enabled` ein-gestellt werden. Weiterhin besitzt die Klasse zwei Methoden, das Setzen eines Key-Value-Paares `set()` und das Nachsehen, ob der gesuchte Key schon gespeichert wurde, `lookup`. Es wird hierbei extensiv die Funktionalität `hashable` verwendet, welche dem Blogspot Eintrag [61] entnommen wurde.

4.4.4 Das Codegenerator Untermodul

Die in der ursprünglichen Version von EFCOSS eingesetzten Funktionen zur Generierung von Interfaces für die Anbindung von Simulationssoftware können in ihrer ursprünglichen Version nicht mehr verwendet werden, da sie nur eine Kopplung an CORBA vorsehen. Somit muss ein neuer Interfacegenerator entwickelt werden, mit welchem Interfaces zur Kopplung der Simulationssoftware an EFCOSS möglich gemacht werden. Je nachdem in welcher Sprache die Simulationssoftware geschrieben ist, müssen unterschiedliche Interfaces generiert werden. Für Simulationen, die in Fortran oder C geschrieben sind, kann das Tool Fortran to Python (F2PY) [56] genutzt werden, um die von EFCOSS generierten Interfaces zu bauen und für Python verfügbar zu machen. Da F2PY mit Numpy und Scipy entwickelt wurde, hat es den Vorteil, sehr effektiven Code zur Arbeit mit Numpy-Arrays zu erzeugen.

Laut [62] gibt es für C/C++ einige alternative Interfacegeneratoren, wie zum Beispiel der Simplified Wrapper and Interface Generator (SWIG) [63] zur Anbindung von C/C++ Code an Scriptsprachen wie Python, TCL oder Ruby. Die weiter genannten CXX [64], Boost.Python [65] und SCXX [66], zielen darauf ab, die Arbeit mit der Python C API zu vereinfachen. Diese "generieren jedoch die Interfaces nicht automatisch und manuelles coding ist erforderlich" [62]. Weiter wird das Tool Pyfort [67] als eine Alternative zu F2PY zur Anbindung von Fortran-Code an Python genannt. Dieses unterstützt jedoch nur die Anbindung von Fortran 77 Code. Für unsere Anwendung ist dies jedoch nicht ausreichend, da wir Fortran 90/95 Code verwenden wollen.

Um nun eine Simulation mittels F2PY an EFCOSS anzubinden, müssen standardisierte Interfaces geschrieben werden, welche von EFCOSS aus aufgerufen werden. Die grobe Struktur der Interfaces für die Simulation ist wie folgt:

1. Allokiere Eingabe- und Ausgabevektoren
2. Setze Eingabeparameter entsprechend EFCOSS
3. Führe Simulation aus
4. Kopiere Ausgabevektoren
5. Cleanup

Für die Interfaces der Ableitungsberechnung sieht die Struktur folgendermaßen aus:

1. Allokiere Eingabe- und Ausgabevektoren
   (a) Allokiere Ableitungsobjekte
2. Setze Eingabeparameter entsprechend EFCOSS
   (a) Setze Ableitungsvektoren entsprechend EFCOSS
3. Führe Ableitungscode aus
4. Kopiere Ausgabevektoren
   (a) Kopiere Ableitungsergebnisse
5. Cleanup

Wie zu sehen ist, sind einige der Aufgaben in beiden Interfaces gleich. Deshalb können diese in allen Interfaces auch gleich definiert werden. Dabei ist außerdem zu beachten, dass die Interfaces für verschiedene Programmiersprachen eine ähnliche Struktur besitzen. Somit können wir eine sprachunabhängige Infrastruktur implementieren, welche die oben genannten Schritte abstrakt beschreibt und die eigentliche Implementierung einem spezialisierten Codegenerator für die entsprechende Sprache überlässt.

4.4.4.1 Abstrakte Beschreibung

Um eine für unsere Zwecke geeignete abstrakte Beschreibung erstellen zu können, müssen zuerst verschiedene grundlegende Teile definiert werden, welche für alle Arbeitsabläufe benötigt werden. Hierzu zählen Variablendefinitionen und Funktionsdefinitionen im resultierenden Programm. Des weiteren sollte die abstrakte Beschreibung auch die Möglichkeit bieten, MPI-parallele Simulationscodes anzubinden.

Variablendefinition Für die Variablendefinitionen muss gelten, dass sie entweder Hilfsvariablen oder die in Abschnitt 4.4.3 definierten Simulationsvariablen mit den Klassen `InputVariableClass` und `OutputVariableClass` repräsentieren. Eine für die Codeerstellung genutzte Variable benötigt folgende Eigenschaften

- Variablenname
- Datentyp, Integer oder Float
- Struktur (Shape)
- Ein-/Ausgabe (Intent)
- Tags
- Größe

Dabei sind die zu erwartenden Tags einer Variablen

- Allokierbares Array
- Ableitungsobjekt
- Aktives/inaktives Ableitungsobjekt

Deshalb wurde eine `Variables` Klasse implementiert, welche die Bedürfnisse erfüllt. Der Konstruktor der Funktion hat die in Quellcode 4.26 gezeigte Struktur. Anhand des Typs der Übergabevariable `var` wird entschieden, um welche Variable es sich handelt.

Quellcode 4.26: Variables Klasse für die Verwendung in den Codegeneratoren

```
class Variables:
  def __init__(self,var="",dtype="int",shape=[],intent=None,tags=None,size=1,shapestring=[]):
      if not isinstance(var,variables.VariableClass):
          self.name = var
          self.dtype = dtype
          self.intent = intent
          self.tags = tags
          self.size = size
          self.shape = shape
          ...
      else:
          self.name = var.id
```

```
        self.dtype = str(var.dtype)
        self.intent = CodeGeneratorBase.intentIn if isinstance(var, variables.InputVariableClass) else
              CodeGeneratorBase.intentOut
        self.tags = tags
        self.size = var.size
        self.shape = var.shape
        ...
```

Funktionsbeschreibung Eine abstrakte Beschreibung von Funktionswrappern sieht, zumindest für die Programmiersprachen C/C++ und Fortran, ähnlich aus. Um ein einheitliches Layout zu erstellen und damit letztlich alle in diesen Sprachen geschriebenen Funktionen an EFCOSS anbinden zu können, muss der Funktionswrapper die folgende Signatur aufweisen. Als erstes muss die Eingabe der Funktionen stehen. Hierzu wird zwischen Integer und Double Werten unterschieden. Auf der Python Seite werden die vorher definierten Eingabevariablen entsprechend ihres Datentypes sortiert und jeweils hintereinander in einen Vektor geschrieben. Dieser Vektor wird dem Interface-Wrapper übergeben und muss in diesem auf die entsprechenden Eingabewerte der zu rufenden Funktion kopiert werden.

Zu diesem Zweck wurde die Klasse `Function` in Quellcode 4.27 entworfen. Sie verwaltet die Funktionsbestandteile, welche für die Codegenerierung benötigt werden.

Quellcode 4.27: Abstrakte Funktionsbeschreibung.

```
class Function:
   def __init__(self,name,simName,paramsOrdered,dLevel=0):
      ...
```

4.4.4.2 Codegenerator

Für die Implementierung der neuen Interfacegeneratoren wurde eine Bachelorarbeit [68] vergeben. Deren Ergebnisse sollen hier detaillierter ausgeführt werden. Der Codegenerator sollte dabei für Simulationsroutinen in verschiedenen Programmiersprachen funktionieren und die abstrakte Beschreibung aus dem letzten Kapitel in ausführbaren Code verwandeln.

Die Basisklasse `CodeGeneratorBase`, vgl. dazu Auszüge in Quellcode 4.28, definiert Konstanten und Datentypen für die Verwendung im Codegenerator. Dabei ist der Aufbau generisch gehalten, um eine Vielzahl von Programmiersprachen zu unterstützen. So werden zum Beispiel bei den Datentypen die in der programmiersprachenspezifischen Implementierung definierten Funktionen wie `integert` oder `doublet` genutzt, um die Datentypennamen entsprechend zu setzen. So muss es bei Fortran `integert="integer"` respektive `doublet="double precision"` heißen, bei C/C++ hingegen `integert="int"` und `doublet="double"`. Fortran bietet ab Version 1990 darüber hinaus noch die Möglichkeit, die Intention der Ein-/Ausgabevariablen anzugeben. Diese sind `"intent in"`, `"intent out"` sowie `"intent inout"`, bei C/C++ existiert eine solche Unterscheidung nicht.

Quellcode 4.28: Basisklasse für den Codegenerator.

```
class CodeGeneratorBase:
    # Constants for intents to avoid hardcoding of values


    ## Constructor
```

```
    # @param include List of modules the function shall include
    # @param indent Size of the indent used to align the code
    def __init__(self, include=[], indent="   "):
        self.include = include
        self.indent = indent  # size of one indent
        self.currentindent = indent
        self.indentlevel = 1
        self.dtypes = {"int": self.integert(),
                       "double": self.doublet(),
                       "float": self.floatt(),
                       "float64": self.doublet(),
                       "int64": self.integert()
                       }
        self.intents = {Intents.intentIn: self.intentIn(),
                        Intents.intentOut: self.intentOut(),
                        Intents.intentInOut: self.intentOut()
                        }
        self.INCLUDE_MPI = ""  # constant to be filled for derivatives
```

Die Klasse beinhaltet einige grundlegende sprachunabhängige Funktionen zur Vereinfachung der Interfacegenerierung:

- `variableDeclarations` - Deklaration von Variablen
- Initialisierungsroutinen:
  - `initialization` - Initialisierung von Variablen für die Simulation
  - `initializationJacobian` - Variableninitialisierung für Jacobi-Matrixberechnung
  - `initializationJacobianVector` - Variableninitialisierung für Jacobi-Vektor-Produkte
  - `initializationJacobianTVector` - Variableninitialisierung für Jacobi-Transponiert-Vektor-Produkte
- Kopierfunktionen:
  - `copyResults` - Ergebnisse der Simulation kopieren
  - `copyResultsJacobian` - Ergebnisse der Jacobi-Matrixberechnung
  - `copyResultsJacobianVector` - Ergebnisse für Jacobi-Vektor-Produkte
  - `copyResultsJacobianTVector` - Ergebnisse der Jacobi-Transponiert-Vektor-Produkte
- `cleanup` - Deallokieren aller vorher allokierten Variablen

Die Umsetzung des Codegenerators für Fortran ist in Quellcode 4.29 zu sehen. Hier werden weitere Konstanten wie Zeilenbeendigungen `endl` oder auch MPI Spezifika definiert. Des weiteren müssen die Eigenheiten der Programmiersprache wie Vergleiche (z.B. Fortran `.eq.`, C `==`) gesetzt werden.

Quellcode 4.29: Codegeneratorklasse für Fortran-basierte Simulationen.

```
class CodegeneratorFortran(CodeGeneratorBase):

    ## Constructor
    # @param include List of modules the function shall inclued
    # @param indent Size of the indent used to align the code
    def __init__(self,include=[],indent="   "):
        CodeGeneratorBase.__init__(self,include,indent)
        ## Symbol used to end a line
        self.endl="\n"
```

```
        ## Module name for MPI
        self.INCLUDE_MPI = "mpi"

    ## Key word for integer type
    def integert(self):
        return "integer"
    ...
    ## Key word for Fortrans intent(in)
    def intentIn(self):
        return "in"
    ...
    ## Key word used to compare two objects in a condition for equality
    def equal(self):
        return ".eq."
```

Als wichtige weitere Funktionen dieser Klasse sind die Funktions- und Variablendeklarationen `functionDeclaration` und `declareOneVariable` zu nennen, welche in Quellcode 4.30 sowie Quellcode 4.31 zu sehen sind. Hier müssen die Fortran-spezifischen Definitionen für die `subroutine` und deren Übergabeparameter aus der übergebenen abstrakten Funktionsbeschreibung `func` extrahiert und entsprechend angewendet werden. Zusätzlich sollen an dieser Stelle noch die *Includes* gesetzt werden, bevor ein `implicit none` implizite Variablendefinitionen verbietet.

Quellcode 4.30: Fortran Codegenerator: Deklarationen für Funktionen und Variablen, sowie Aufruf einer Funktion.

```
    ## Declaration of function head and includes
    # @params all are supposed to be pointers
    def functionDeclaration(self,func):
        sstr = "subroutine "+func.name+"("+', '.join(map(lambda s: s.name,func.paramsOrdered.values()))+
            ")" + self.endl
        for i in self.include:
           sstr = sstr + self.currentindent + "use " + i + self.endl
        sstr = sstr + self.currentindent + "implicit none" + self.endl*2
        return sstr
```

Die Variablendefinitionen werden über die Funktion `declareOneVariable`, zu sehen in Quellcode 4.31, realisiert. Dabei werden die Variablen aus der abstrakten Beschreibung als Eingabe genommen und entsprechend ihres Datentyps und des Shapes in einem String `sstr` zusammengesetzt, welcher am Ende zurückgegeben wird. Für mehrdimensionale Variablen, welche zum Aufruf der Simulationsroutine allokierbar gesetzt sind, wird das entsprechende Key-Word `allocatable` an die Deklaration angehangen und die Dimension `dimensions` der Variablen entsprechend mit `:,:,...,:` gesetzt. Sollte die Variable nicht allokierbar sein, es handelt sich also um eine statische Variable, werden die Dimensionen entsprechend dem `shapestring` der Variablen gesetzt. Weiter sind die Variablen nach ihrer Funktion als Ein- und Ausgabe (`group="paramVar"`) oder Zwischenvariablen (`group="other"`) zu unterscheiden. Für die ersten beiden wird noch die Intention, also ob es sich um eine Eingabevariable `intent(in)` oder Ausgabevariable `intent(out)` handelt, gesetzt.

Quellcode 4.31: Fortran Codegenerator: Deklaration einer Variablen.

```
    ## Declare one variable
    # @param var: Variable (type Variables) to declare
    # @param group: Usage of variable, can be one of:
    #  - paramVar: Parameter passed to the function
```

```
#  - other (default)
def declareOneVariable(self,var,group="other"):

   # Indent and datatype
   sstr = self.currentindent + self.dtypes[var.dtype]

   # Array declaration
   if var.shapestring != []:
      if ( Tags.ALLOCATABLE in var.tags ):
        dimensions = ":"+',:'*(len(var.shapestring)-1)
      else:
        dimensions = ','.join(map(lambda s: str(s), var.shapestring))
      sstr += ", dimension(" + dimensions + ")"
      if (group != "paramVar" and ':' in dimensions):
        sstr += ", allocatable"

   # Define direction of function parameters
   if group == "paramVar":
      sstr += ", intent(" + self.intents[var.intent] + ")"


   sstr += " :: " + var.name + self.endl
   return sstr
```

Um einen Funktionsaufruf mit seinen Parametern zu erzeugen, wird die in Quellcode 4.32 gezeigte Routine benötigt. Diese zeigt für den Fortran-Generator den Aufbau der Funktionsaufrufe der Funktion `funcName` über eine mit “,” separierte Liste als String über die Parameter der Funktion `params`.

Quellcode 4.32: Fortran Codegenerator: Deklarationen für den Aufruf einer Funktion.

```
## Call a function
# @param funcName Name of the function
# @param params String list of parameters passed to the function
def callFunction(self, funcName, params):
    paramString = "(" + ','.join(map(lambda s: str(s), params)) + ")"
    sstr = self.currentindent + "call " + funcName + paramString + self.endl
    sstr = self.wrapLine(sstr)
    return sstr
```

Weiterhin bietet die Routine die folgenden sprachabhängigen Methoden:

- Arbeiten mit Arrays:
  - `getArrayElement` - Zugriff auf ein Array-Element
  - `setValueForAllElements` - Alle Elemente eines Arrays auf einen bestimmten Wert setzen
  - `getArrayElementRange` - Auswahl von zusammenhängenden Array-Elementen erhalten
- Verzweigungen:
  - `condition` - Bedingung erzeugen
  - `ifthen` - Verzweigung anlegen (If-Zweig)
  - `elsebranch` - Sonst-Zweig
  - `endif` - Verzweigung beenden

- Schleifen:
  - `forloop` - For-Schleife
  - `whileloop` - While-Schleife
  - `endloop` - Schleife beenden
- Arbeiten mit Arrays und Variablen:
  - `allocate` - Speicher allokieren
  - `free` - Allokierten Speicher freigeben
  - `incVar` - Inkrement einer Variablen
  - `sizeOf` - Einträge eines Arrays bestimmen
- Zusätzliche Funktionen:
  - `println` - Ausgabe eines Strings
  - `wrapLine` - Zeilentrennung für zu lange Zeilen

Mit diesen Definitionen ist es möglich, eine abstrakte Beschreibung der geforderten Interfaceroutinen anzugeben. Wie dies genau funktioniert, wird an einem Beispiel in Quellcode 4.39 aus Abschnitt 4.4.4.3 gezeigt.

Die Klasse `SimulationInterfaceBase`, zu sehen in Quellcode 4.33–4.35, ist die Basisklasse für die Interfaces an die Simulation. Zur Initialisierung wird eine Struktur namens `Specs` benötigt. Diese hat den folgenden Aufbau:

1. Eine EFCOSS-Instanz
2. Der Name der Simulation
3. Eine Sprachbeschreibung
4. Eine AD-Beschreibung
5. Die genutzte Simulationsinstanz

Quellcode 4.33: Basisklasse für den Codegenerator.

```
class SimulationInterfaceBase:

    ## ...
    def __init__(self,Specs):
        self.efcoss = Specs[0]
        self.simName = Specs[1]
        self.languageDescription=Specs[2]
        self.adDescrption=Specs[3]
        self.instance=Specs[4]
        #Specs = (efcoss, TopLevelRoutine, languageDescription, ADdescription, instance)
        self.inputVars = self.efcoss._get_InputVars(self.instance)
        self.outputVars = self.efcoss._get_OutputVars(self.instance)
        self.optVars = self.efcoss._get_OptVars(self.instance)

        self.SimulationCallingSequence = self.efcoss._get_SimulationCallingSequence(self.instance)


        self.mpi=self.languageDescription.mpi
        self.adtool=self.adDescrption.adtool
        self.adlevel=self.adDescrption.adlevel
        self.reversemode=self.adDescrption.reversemode
        self.codeGenerator=self.languageDescription.getGenerator()
```

```
        ## Maximum number of differentiation directions
        self.nbDirsMax = sum( getVariableSizes(self.efcoss.getNumericalInputVars()) )

        # enable mpi if specified
        if (self.mpi==True):
           self.codeGenerator.addInclude("mpi")

        ## Dictionary containing the parameters of the next function
        #self.paramsOrdered = {}

        self.function = self.declareFunction()
        self.jacobian = self.declareJacobian()
        self.jacvec = self.declareJacobianVector()
        self.jactvec = self.declareJacobianTVector()
        self.hessian = self.declareHessian()
        self.projected_hessian = self.declareProjectedHessian()


        if (self.mpi==True):
          self.addMPIVariables()
```

Die Methoden `declareFunction`, `declareJacobian`, und so weiter sollen exemplarisch in Quellcode 4.34 gezeigt werden. Unsere Interfaceroutine soll die Eingabevariablen `iInputVector` und `dInputVector` jeweils für Integer und Floating-Point Variablen besitzen. Als Ausgabevektor wird `dOutputVector` genutzt. Diese werden jeweils als `Variables` definiert und entsprechend ihrer Funktionsbeschreibung in das Dictionary `params` eingefügt. Dieses wird dann benutzt, um die `Function` mit dem Namen `"Func"` zu initialisieren. Diese benötigt den Namen der Toplevel-Routine des Simulationscodes. Die Reihenfolge und Namen der Übergabeparameter der Toplevel-Routine werden mittels der Methode `addSimVar` von `Function` hinzugefügt.

Quellcode 4.34: Basisklasse für den Codegenerator - Funktionsdeklaration.

```
    # Creates an ordered list of parameters to be used in function interface
    # @returns Ordered list of parameters
    def createParamsFunction(self):
        # Vars for Function
        iInputVector = Variables("iInputVector", "int", [":"], Intents.intentIn, [Tags.MPI_SEND])
        dInputVector = Variables("dInputVector", "double", [":"], Intents.intentIn, [Tags.MPI_SEND])
        dOutputVector = Variables("dOutputVector", "double", [":"], Intents.intentOut)
        params = {}

        # Prepare pameter dictionary for Function
        params[Function.iInputVector] = iInputVector
        params[Function.dInputVector] = dInputVector
        params[Function.dOutputVector] = dOutputVector

        return params

    ## Create self.function with all variables
    # @returns Object of type Function representing the function
    def declareFunction(self):
        paramsOrdered = self.createParamsFunction()

        # declare Function
        function = Function("Func", self.simName, paramsOrdered, 0)
```

```
    # add simulation variables to function
    # NOTE: Variables MUST be added in the order of SimulationCallingSequence.
    for var in self.SimulationCallingSequence:
        function.addSimVar(Variables(var))

    return function
```

Um die Interfaceroutine für die Jacobiberechnung zu generieren, benötigt man die Methode `declareJacobian` aus Quellcode 4.35, welche die Ein- und Ausgabevariablen für die Ableitungsberechnung enthält. Hier werden außerdem Ableitungsvariablen definiert, mit denen die automatisch generierte Ableitung aufgerufen wird.

Quellcode 4.35: Basisklasse für den Codegenerator - Jacobideklaration.

```
## Create self.jacobian with all variables
# @returns Object of type Function representing the jacobian
def declareJacobian(self):
    paramsOrdered = self.createParamsJacobian()

    # declare Jacobian
    jacobian = Function("Jacobian", self.simName + "_dv", paramsOrdered, 1)

    # Add variables to jacobian
    # NOTE: Variables MUST be added in the order of SimulationCallingSequence.
    idsOptVars = getVariableIds(self.optVars)
    idsInputVars = getVariableIds(self.inputVars)
    idsOutputVars = getVariableIds(self.outputVars)
    for var in self.SimulationCallingSequence:
        jacobian.addSimVar(Variables(var))  # normal variables

        # create and add derived variables of float/double variables in OptVars
        # if (var.id in idsOptVars or var.id in idsOutputVars):
        if (var.dtype in ["float", "float64", "double"] and (var.id in idsOptVars or var.id in
             idsOutputVars)):

            # NOTE: After introducing prefix and suffix here, the method
            # Function.getOriginalVar() must be adapted, too
            name = 'd_' + var.id
            dtype = str(var.dtype)
            size = var.size * self.nbDirsMax

            shape = (self.nbDirsMax,) + var.shape
            shapestring = [str(Function.AD_P_MAX)] + var.shapestring
            tags = [Tags.DERIVED]

            # if (var.id in idsOptVars):
            if (var.id in idsInputVars):
                intent = None
                if (not var.id in idsOptVars):
                    tags.append(Tags.SKIP_SEED_INPUT)
            else:
                intent = Intents.intentOut

            jacobian.addSimVar(Variables(name, dtype, shape, intent, tags, size, shapestring))
```

```
        jacobian.addSimVar(Variables(Function.AD_P))
        jacobian.addHelperVar(Variables(Function.AD_P_MAX))

        self.createDiffSizes()

        return jacobian
```

Der Python Code in Quellcode 4.36 zeigt die komplette Generatorroutine für das Jacobi-Interface. Es wird dabei ein String `r` mit der Funktionsdeklaration für `Jacobian` erzeugt, welcher dann im Laufe der Routine gefüllt und am Ende zurückgegeben wird. Dazu müssen die verwendeten Variablen deklariert und initialisiert werden (mittels `variableDeclarations, initialization` und `initializationJacobian`). Dann kann die Simulation mittels `callSimulation` aufgerufen werden. Am Ende müssen die Ergebnisse mittels `copyResults*` in die EFCOSS-verständlichen Arrays kopiert werden. Zusätzlich werden die allokierten Arrays wieder freigegeben, dies geschieht mit der Routine `cleanup`. Ein `endFunction` schließt die Routine ab. Der Einfachheit halber wurde an dieser Stelle der komplette MPI-spezifische Teil für die Beschreibung weggelassen.

Quellcode 4.36: Basisklasse für den Codegenerator.

```
class TapenadeInterface(SimulationInterfaceBase):

   ## contains the code for the jacobian matrix
   def jacobian_interface(self):
      r = self.codeGenerator.functionDeclaration(self.jacobian)
      r += self.codeGenerator.variableDeclarations(self.jacobian)

      r += self.codeGenerator.initialization(self.function)
      r += self.codeGenerator.initializationJacobian(self.jacobian)

      r += self.codeGenerator.callSimulation(self.jacobian)

      r += self.codeGenerator.copyResults(self.jacobian)
      r += self.codeGenerator.copyResultsJacobian(self.jacobian)
      r += self.codeGenerator.cleanup(self.function)
      r += self.codeGenerator.cleanupJacobian(self.jacobian)

      r += self.codeGenerator.endFunction()
      return r
```

4.4.4.3 Neues Interface für den Rückwärtsmodus des automatischen Differenzierens

Die in EFCOSS implementierten Interfaces für das Arbeiten mit AD-Code sind darauf ausgelegt, die komplette Jacobi-Matrix mit dem Vorwärtsmodus des automatischen Differenzierens zu berechnen und diese dann weiter zu verwenden. Dies hat, wie in Abschnitt 3.3 beschrieben, einen negativen Effekt auf die Ausführungsgeschwindigkeit, da die Berechnung der kompletten Jacobi-Matrix linear mit der Größe der Eingabedaten skaliert.

Eine weitere Möglichkeit besteht darin, die Jacobi-Matrix über den Rückwärtsmodus zu berechnen. Hierbei spielt jedoch die Größe der Ausgabedaten eine entscheidende Rolle für die Performance. Für die in dieser Arbeit betrachteten Funktionen macht der Einsatz des Rückwärtsmodus für die Berechnung der Jacobi-Matrix keinen Sinn. Es gibt jedoch Optimierungsalgorithmen, die intern das Produkt

$$\mathbf{y} = J^T J\mathbf{x} \tag{4.1}$$

auswerten. Dabei kann

$$J\mathbf{x} = \mathbf{z} \tag{4.2}$$

mit dem Vorwärtsmodus des automatischen Differenzierens und anschließend

$$J^T\mathbf{z} = \mathbf{y} \tag{4.3}$$

mit dem Rückwärtsmodus berechnet werden kann.

Außerdem haben wir in Abschnitt 2.2.1 festgestellt, dass die Formel

$$\frac{df}{d\mathbf{x}} = \frac{J^T\mathbf{f}(\mathbf{x})}{\|\mathbf{f}(\mathbf{x})\|}$$

zur Berechnung der Ableitungsfunktion der Zielfunktion bei Parameterschätzproblemen genutzt werden kann.

Dies verdeutlicht die Notwendigkeit der Implementierung von Interfaceroutinen, welche $J\mathbf{v} = \mathbf{t}$ beziehungsweise $J^T\mathbf{w} = \mathbf{u}$, mit $\mathbf{v}$ und $\mathbf{w}$ beliebige Vektoren richtiger Dimension möglichst effektiv, d.h. ohne Aufstellen der kompletten Jacobi-Matrix, berechnen können.

Es wird ein Interface benötigt, welches als Eingaben den Vektor $\mathbf{x}$ (also die Stelle der Ableitung), sowie den Vektor $\mathbf{v}$ bzw. $\mathbf{w}$ bekommt und dann das Matrix-Vektor-Produkt $J\mathbf{v} = \mathbf{t}$ mit dem Vorwärtsmodus des automatischen Differenzierens berechnet und ausgibt. Dieses Interface nennen wir `JacobianVector`. Ein weiteres Interface `JacobianTVector`, berechnet $J^T\mathbf{w} = \mathbf{u}$ mit Hilfe des Rückwärtsmodus des automatischen Differenzierens.

In Quellcode 4.37 ist eine Routinendefinition für das `JacobianTVector` Interface gegeben. Die Eingaben sind wie vorher `iInputVector` und `dInputVector`. Des weiteren wird `dInputSeed` als Double-Vektor $\mathbf{w}$ genutzt. Die Ausgaben sind wieder `dOutputVector`, also die Ausgabe der Funktion, sowie die Ausgabe der Multiplikation $J^T\mathbf{w}$ in `dJacobiTVector`. Das Interface für den Vorwärtsmodus hat die gleiche Struktur.

Wichtig für die Ausführung des Ableitungscodes ist das Setzen der entsprechenden Ableitungsvariablen. Für das Interface des Vorwärtsmodus müssen die Ableitungsvariablen der Eingaben entsprechend gesetzt werden, für das Interface des Rückwärtsmodus müssen die Ableitungsvariablen der Ausgabe gesetzt werden. Die Richtung der Variablen, nach denen nicht abgeleitet werden soll, müssen entsprechend auf 0 gesetzt werden.

Quellcode 4.37: Interface für $J^T\mathbf{w} = \mathbf{u}$.

```fortran
subroutine JacobianTVector(iInputVector, dInputVector, dInputSeed, dOutputVector, dJacobiVector)
   integer, dimension(:), intent(in) :: iInputVector
   double precision, dimension(:), intent(in) :: dInputVector
   double precision, dimension(:), intent(in) :: dInputSeed
   double precision, dimension(:), intent(inout) :: dOutputVector
   double precision, dimension(:), intent(inout) :: dJacobiVector
```

Somit ergibt sich der in Quellcode 4.38 gezeigte Pythoncode. Wieder wurde zum besseren Verständnis der MPI-Code weggelassen. Der Ausgabestring `r` muss neben der Funktionsdeklaration, welche von `functionDeclaration` mit dem Übergabeparameter `self.jactvec` erzeugt wird noch die Variablendeklarationen und Initialisierungen der Funktion beinhalten. Danach kann die abgeleitete Simulationsroutine ausgeführt werden. Im Anschluss werden die Ergebnisse der Simulation und des Matrix-Vektor-Produktes in die Ausgabevektoren kopiert, der angelegte Speicher wieder freigegeben und die Funktion beendet.

Quellcode 4.38: Codegenerator Code für $J^T \mathbf{w} = \mathbf{u}$.

```
 ## contains the code for the jacobian transposed vector
def jacobiantvector_interface(self):
    r = self.codeGenerator.functionDeclaration(self.jactvec)
    r += self.codeGenerator.variableDeclarations(self.jactvec)

    r += self.codeGenerator.initialization(self.function)
    r += self.codeGenerator.initializationJacobianTVector(self.jactvec)

    r += self.codeGenerator.callSimulation(self.jactvec)
    r += self.codeGenerator.copyResults(self.jactvec)
    r += self.codeGenerator.copyResultsJacobianVector(self.jactvec)
    r += self.codeGenerator.cleanup(self.function)
    r += self.codeGenerator.cleanupJacobian(self.jactvec)

    r += self.codeGenerator.endFunction()

    return r
```

Wir sehen uns exemplarisch in Quellcode 4.39 die Initialisierungsroutine für die Variablen des Matrix-Transponiert-Vektor-Produktes des Fortran Codegenerators an. Wie zu sehen ist, müssen zuerst die Variablen allokiert werden. Zuvor wird jedoch geprüft, ob die zu erzeugende Funktion eine MPI-Worker-Routine ist oder nicht. Dazu wird die Variable `seed` entsprechend auf den String `"dInputSeedWorker"` für die Worker und `dInputSeed` für den Master (oder wenn kein MPI genutzt werden soll) gesetzt. Als nächstes müssen alle abgeleiteten Variablen, die als allokierbar definiert sind, allokiert werden. Nun wird eine Zählvariable benötigt, um durch den Seed-Vektor zu iterieren. Diese wird im weiteren Verlauf mit 1 initialisiert. Davor wird jedoch der aktuell verwendete Zähler der Funktion gespeichert, um diesen später wieder zurück zu setzen.

Nun kann über den Seed-Vektor jede Input-Variable der Simulation, die den Tag `DERIVED` hat, also abgeleitet werden soll, iteriert und der entsprechende Wert des Seeds gesetzt werden. Falls die Variable ein Array ist, muss zusätzlich noch über dieses iteriert werden. Zum Schluss wird der aktuelle Zähler wieder auf die vorher definierte Zählvariable gesetzt und der Programm-String zurückgegeben.

Quellcode 4.39: Codegenerator Code für die Initialisierung der AD-Variablen für $J^T \mathbf{w} = \mathbf{u}$.

```
def initializationJacobianTVector(self, func, isWorker=False):
    sstr = ""

    if (isWorker == False):
        seed = func.paramsOrdered[Function.dInputSeed]
    else:
        seed = Function.INPUT_SEED_WORKER

    # Allocate derived variables
    for var in func.getVariablesWithTags([Tags.DERIVED, Tags.ALLOCATABLE]):
        if (var.shape != ()):
            sstr += self.allocate(var)
```

```
    startCtr = func.curctr  # save index of current counter
    seedCtr = func.getCounter()  # counter for seed matrix
    sstr += self.endl + self.assign(1, seedCtr) + self.endl

    # initialize all derivatives of the optimization variables with the seed matrix
    for var in func.getVariablesWithTags(Tags.DERIVED):
        sstr += self.setValueForAllElements(var, 0.0)
        if (var.intent == 1):
            print var.name, var.shape
            # scalar vars
            if var.shape == ():
                sstr += self.assign(seed, var, seedCtr.name)
                sstr += self.incVar(seedCtr)
            # array vars
            else:
                # assign non-zero value
                loopStr, indices = self.beginIterationOverList(var, func, False,
                                                               len(var.shapestring))  # iterate over
                                                                    variable
                print loopStr
                sstr += loopStr
                element = self.getArrayElement(seed, seedCtr.name)  # get current element of
                    iInputSeed
                sstr += self.assign(element, var, None, indices)
                sstr += self.incVar(seedCtr)
                sstr += self.endloop(len(indices))

                func.curctr = startCtr + 1  # for initialization above

    func.curctr = startCtr  # reset current counter

    return sstr + self.endl
```

Das fertige Interface für die Funktion `jactvec` zur Berechnung von $J^T\mathbf{w} = \mathbf{u}$ findet sich im Anhang Kapitel A.2 in Quellcode A.2

4.4.5 Das Optimization Modul

In diesem Modul finden sich die Klassen für Zielfunktionen `Objectives` und Nebenbedingungen `Constraints`, sowie Interfaces zu den unterstützten Optimierern.

4.4.5.1 Anbindung der Scipy Optimierer

Ein weiterer wichtiger Schritt ist, die in EFCOSS gegebenen Optimierer hinsichtlich ihrer Nutzbarkeit zu untersuchen. Einige der bekannten Optimierer, die EFCOSS in seiner ursprünglichen Form bereitgestellt hat, finden sich in den Paketen von Scipy.Optimizer. Zu den verfügbaren Methoden zählen:

- Minpacks Levenberg-Marquardt Verfahren (`method='lm'`)
- COBYLA (`method='COBYLA'`)
- L-BFGS-B (`method='L-BFGS-B'`)
- TNC (`method='TNC'`)
- SLSQP (`method='SLSQP'`)

Die Klasse `ScipyOptimizer`, zu sehen in Quellcode 4.40, ist das Interface für diese Optimierer. Diese bietet die Funktionen `run`, als Aufruf für den lokalen, vektoriellen Optimierer, `run_scalar` als Aufruf für den lokalen, skalaren Optimierer und `run_global` als Aufruf für die globale Optimierung. Des Weiteren bietet sie Set-Methoden für die Bounds und Optionen.

Quellcode 4.40: Python Interface für die Scipy Optimierer aus `scipy.optimize`.

```
class ScipyOptimizer:
    def __init__(self, method='L-BFGS-B', tol=1e-6, bounds=None, options={'disp': True}):
        self.method = method
        self.bounds = bounds
        self.options = options
        self.tol = tol

    def setBounds(self, bounds):
        if (bounds != None):
            self.bounds = bounds

    def setOptions(self, options):
        self.options = options

    def run(self, fun, x0, args=(), jac=None, bounds=None):
        self.setBounds(bounds)
        return scipy.optimize.minimize(fun, x0, args=args, method=self.method,
                                       bounds=self.bounds, tol=self.tol,
                                       options=self.options, jac=jac)

    def run_scalar(self, fun, x0, args=(), jac=None, bounds=None):
        self.setBounds(bounds)
        return scipy.optimize.minimize_scalar(fun, x0, args=args, method=self.method, bounds=self.bounds
            )

    def run_global(self,fun,x0,jac=None,bounds=None):
        return scipy.optimize.basinhopping(fun,x0,minimizer_kwargs={"method":self.method, "bounds":self.
            bounds,"tol":self.tol,"options":self.options, "jac":jac})
```

In der Klasse `opt_scipy`, vgl. Quellcode 4.41 ist das Interface für EFCOSS an die eben beschriebene Klasse geschrieben. Sie leitet sich von `ScipyOptimzer` ab und nutzt die EFCOSS-Routinen für den Aufruf von Funktionen und Ableitungen der Simulationsroutine. Dies geschieht abhängig von der Größe der Ausgaben entweder durch die skalare Version von Scipy.Optimize (`minimize_scalar` bei `m=1`) oder durch die vektorielle Version (`minimize` bei `m>1`). Dabei werden die Funktionen aus Quellcode 4.42 verwendet, welche die EFCOSS-Routinen `evalf` beziehungsweise `evalfvec` für die Simulation, sowie `evalgf` beziehungsweise `evalfjac` für die Ableitungen benutzt. Als Initialisierungsargumente muss eine EFCOSS-Instanz gegeben werden. Standardmäßig wird ein unbeschränktes (`bounds=[(None,None)]`) Optimierungsproblem mit der Methode `L-BFGS-B` und einer Toleranz von 10^{-10} gelöst. Dabei ist die Ausgabe (`disp`) des Optimierers aktiviert und die interne Ableitungsberechnung von Scipy deaktiviert (mittels `useInternalDD`).

Quellcode 4.41: Python Interface für die Initialisierung des ScipyOptimizer zur Verwendung in EFCOSS.

```
class opt_scipy(ScipyOptimizer.ScipyOptimizer):
    def __init__(self, efcoss, method='L-BFGS-B', tol=1e-10, bounds=[(None, None)], options={'disp':
        True},
                useInternalDD=False):
```

```
        ScipyOptimizer.ScipyOptimizer.__init__(self, method, tol, bounds, options)
        self.efcoss = efcoss
        self.n = len(self.efcoss.getOptVec())
        self.m = len(self.efcoss.getResultVec())
        if self.m>1:
            self.fun=self.fun_fvec
            self.jac=self.jac_fvec
        else:
            self.fun=self.fun_f
            self.jac=self.g_f
        if useInternalDD:
            self.jac=None
        if (len(self.bounds) < self.n):
            self.bounds = [bounds for i in range(self.n)]
```

Quellcode 4.42: Python Interface für die Funktionen zur Verwendung im ScipyOptimizer.

```
    def fun_f(self, x):
        return self.efcoss.evalf(self.n, x, self.m)

    def fun_fvec(self, x):
        return self.efcoss.evalfvec(self.n, x, self.m)

    def g_f(self, x):
        return self.efcoss.evalgf(self.n, x, self.m)

    def jac_fvec(self, x):
        jac, f = self.efcoss.evalfjac(self.n, x, self.m)
        return jac, f
```

Zum Starten der Optimierer sind die Funktionen `minimize` und `minimize_global` vorgesehen, siehe Quellcode 4.43. Diese rufen die Funktionen `run` bzw. `run_global` mit den gegebenen Einstellungen auf.

Quellcode 4.43: Methoden für die Minimierung in ScipyOptimizer.

```
    def minimize(self, x0):
        print "Starting Minimization for x0=", x0
        self.efcoss.writelog("Starting Minimization for x0="+str(x0)+"\n")
        return self.run(self.fun, x0, jac=self.jac)

    def minimize_global(self, x0):
        print "Starting Minimization for x0=", x0
        self.efcoss.writelog("Starting global Minimization for x0="+str(x0)+"\n")
        return self.run_global(self.fun, x0, jac=self.jac)
```

Im weiteren Verlauf dieser Arbeit werden weitere Interfaces exemplarisch für ausgewählte Optimierer außerhalb von `ScipyOptimizer` vorgestellt.

4.4.5.2 Fortran Interfaces

Als Beispiel für den Fortran Teil eines Optimierers soll ELSUNC dienen. Die Toplevel Routine in der Datei `dblelsunc.f` befindet sich im Verzeichnis `EFCOSS/OPT/ELSUNC` und hat die Signatur

```
 SUBROUTINE ELSUNC(X,N,MDC,MDW,M,FFUNC,BND,BL,BU,P,W,EXIT,F,C)
```

Dabei ist die Liste der Übergabeparameter wie folgt:

- X(N) - Startvektor und Ergebnis
- N - Größe der Unbekannten
- MDC - Leading Dimension von C
- MDW - Leading Dimension von W
- M - Größe des Arrays F
- FFUNC - Zielfunktion
- BND - Bounds Optionen
- BL(N) - Lower Bounds
- BU(N) - Upper Bounds
- P(11+2*N) - Arbeitsvektor mit Optionen für den Optimierer
- W(MDW) - Abbruchtoleranz einstellen
- EXIT - Exit-Code
- F(M) - Funktionswert am Ende
- C(MDC,N) - Speicher für die Kovarianz-Matrix

Dabei hat der Arbeitsvektor P die folgenden wichtigen Einträge

- Input:

P(1) - Outputschrittweite
P(2) - Outputstream
P(3) - Maximale Iterationsanzahl

- Output:

P(6) - Iterationsanzahl nach Bearbeitung
P(7) - Anzahl Funktionsauswertungen gesamt
P(8) - Anzahl Funktionsauswertungen für die Jacobi-Berechnung
P(11) - Pseudo-Rang der Matrix $\hat{J}$

Die wichtigen Einträge der Matrix W sind

- Input:

W(1) - TOL=SQRT(SRELPR) Pseudo-Rank Toleranz
W(2) - EPSREL=SQRT(SRELPR) Relative Konvergenz
W(3) - EPSABS=SRELPR Absolute Konvergenz
W(4) - EPSX=SQRT(SRELPR) Parameter Konvergenz

- Output:

W(5) - Wert der Zielfunktion

Die Einstellung der Grenzen geschieht über die Parameter `BND`, sowie `BL` und `BU`. Dabei gilt:

- `if BND==0:` Keine Beschränkung des Problems
- `if BND==1:` Jeder X-Wert hat die gleiche Schranke `BL[1]`$< x <$`BU[1]`
- `else:` Jeder X-Wert hat eine eigene Schranke `BL[`i`]`$< x_i <$`BU[`i`]` mit $i \in \{1 \ldots n\}$

Um diese Funktion nun in EFCOSS nutzen zu können, muss der Optimierer zunächst in eine Bibliothek compiliert werden, welche dann mittels F2PY an Python angeschlossen wird. Dazu werden die folgenden Befehle in einem Makefile bereit gehalten:

```
library:
   gfortran -fPIC -c dbluchelp.f
   ar -r libelsunc.a dbluchelp.o
   f2py -c -m elsunclib dblelsunc.f -L. -lelsunc --debug
```

Um die Bibliothek zu erstellen, muss im Verzeichnis `OPT/ELSUNC` nur

```
make library
```

eingegeben werden.

Das Interface wird daraufhin erzeugt und kann in Python mittels `import elsunclib` verwendet werden. Dies geschieht über die Klasse `opt_elsunc`, welche ähnlich zu den Interfaces von Scipy aufgebaut ist. Die internen Komponenten des Optimierers, also die verwendeten (Hilfs-)Matrizen und Vektoren werden vom Anwender versteckt. Die Größe des Optimierungsproblems wird entsprechend aus EFCOSS ausgelesen und für den Optimierer die Parameter gesetzt. Der Anwender kann aber weiterhin in das Geschehen eingreifen, indem er die entsprechenden Optionen in `options` setzt. So ermöglicht die Option `'disp':True` die Ausgabe der Zwischenergebnisse, indem in Vektor `p` die Werte `p[0]=1`[1] und `p[1]=6` gesetzt werden. Die übergebenen Iterationsanzahlen und Toleranzen werden entsprechend in `p` bzw. `w` eingetragen. Die Beschränkungen des Optimierungsproblems werden ebenso wie in Scipy als Tupel übergeben und entsprechend in die `bl` und `bu` überführt.

Quellcode 4.44: Python Interface für die Initialisierung der Optimierungsfunktion ELSUNC.

```
import elsunclib

class opt_elsunc:

    def __init__(self, efcoss,
                 options={'disp': True, 'tol': 1e-6, 'maxiter': 100, 'pranktol': 1e-6, 'rtol': 1e-6, '
                     atol': 1e-6,
                         'ctol': 1e-6, 'newton': 2},
                 bnd=0, bounds=((0, 0)),
                 objectiveFunctionInstance=0):
        self.efcoss = efcoss
        self.options = options
        self.instance = objectiveFunctionInstance
        self.m = len(self.efcoss.getResultVec(self.instance))
        self.n = len(self.efcoss.getOptVec(self.instance))

        self.mdc = self.m
        self.mdw = self.n * self.n + 5 * self.n + 3 * self.m + 6
        self.p = array([-1] * (11 + 2 * self.n), int32)
        self.w = array([-1.0] * self.mdw, float64)

        if self.options['disp']:
            self.p[0] = 1  # step between writing
            self.p[1] = 6  # output device (6=Terminal)
        self.p[2] = self.options['maxiter']  # max. number of iterations
        self.p[4] = self.options['newton'] # Newton-Method
        self.p[5] = 0  # No Internal Scaling

        self.w[0] = self.options['tol']  # pseudo rank tolerance constant
```

---

[1] Python/C Nummerierung der Arrays.

```
        self.w[1] = self.options['rtol']  # relative convergence constant
        self.w[2] = self.options['atol']  # absolute convergence constant
        self.w[3] = self.options['ctol']  # parameter convergence constant

        self.bnd = bnd
        if self.bnd == 0:
            pass
        elif self.bnd == 1:
            self.bl = array(bounds[0][0], float64)
            self.bu = array(bounds[0][0], float64)
        else:
            self.bl = zeros(self.n, float64)
            self.bu = zeros(self.n, float64)
            for i, bnds in enumerate(bounds):
                self.bl[i] = bnds[0]
                self.bu[i] = bnds[1]
```

Der Aufruf des Optimierers geschieht mittels der `minimize` Funktion aus Quellcode 4.45. Hier wird die Funktion `ffunc` benötigt, welche je nach Übergabeparameter `ctrl` entweder die Simulation (für $|ctrl| = 1$) oder die Ableitungsroutine (sonst) aus EFCOSS aufrufen.

Quellcode 4.45: Python Interface für die Ausführung und Zielfunktionswrapper der Optimierungsfunktion ELSUNC.

```
    def minimize(self, x0):
        t_exit, f, c = elsunclib.elsunc(x0, self.mdc, self.m, self.ffunc, self.bnd, self.bl, self.bu,
             self.p, self.w)
        return t_exit, f, c

    def ffunc(self, x, f_in, ctrl, c_in, n, m, mdc):
        f = f_in
        jac = c_in
        if (abs(ctrl) == 1):
            f = self.efcoss.evalfvec(n, x, m)
        else:
            jac, f = self.efcoss.evalfjac(n, x, m)
        return f, jac
```

4.4.5.3 Python Interfaces

Für einige der in Kapitel 3.1 beschriebenen Optimierer gibt es abseits von Scipy Bestrebungen, diese direkt in Python verfügbar zu machen. Ein einfaches Beispiel ist IPOPT [69], welches mit dem Python Paket `pyipopt` [70] angebunden wird, siehe Quellcode 4.46. Die Grenzen der Optimierung werden entsprechend der Variablen `bounds` gesetzt. Für die Grenzen der Nebenbedingungen ist die Variable `gbounds` entscheidend. Sind diese ungleich `none`, so werden die Grenzen betrachtet. Dabei sind `xl` und `xu` die untere beziehungsweise obere Schranke an `x`. Je nach der Länge der Eingabearrays werden entweder für alle Optimierungsvariablen die gleichen Grenzen gesetzt, oder es müssen genau so viele Wertepaare angegeben werden, wie es Optimierungsvariablen gibt. Für die Grenzen der Nebenbedingungen wird analog verfahren.

Als erste Ausführungsroutine findet sich `run`, welche die Hauptroutine des Paketes PyIpopt nutzt, um eine Instanz des Optimierers zu erstellen und diesen auszuführen. Die zweite Ausführungsroutine ist `fmin_unc`, welche ein unbeschränktes Optimierungsproblem löst. Beiden Routinen ist gemein, dass sie die Optimierer mit den Methoden zur Lösung der Funktion, `evalf` und deren Ableitung in `gradf` benutzen. Es sind jeweils kurze Wrapper um die EFCOSS Methoden `evalf` und `evalgf` an das geforderte Interface des Optimierers anzupassen. Für das beschränkte Problem finden sich darüber hinaus noch Methoden für die Berechnung der Nebenbedingungen `evalg` und `gradg`.

Quellcode 4.46: Interface für den Optimierer PyIpopt.

```
class ipopt:
    def __init__(self, efcoss, bounds=None, gbounds=None, objectiveFunctionInstance=0):
        ...
        self.n=len(self.efcoss.getOptVec(self.instance))
        self.xl=ones(self.n,dtype=float64)*-1e125
        self.xu=ones(self.n,dtype=float64)*1e125
        if bounds is not None:
            if len(self.bounds)==1:
                self.xl=ones(self.n,dtype=float64)*bounds[0][0]
                self.xu=ones(self.n,dtype=float64)*bounds[0][1]
            elif len(self.bound)==self.n:
                self.xl=zeros(self.n,dtype=float64)
                self.xu = zeros(self.n, dtype=float64)
                for i in range(self.n):
                    self.xl[i]=bounds[i][0]
                    self.xu[i]=bounds[i][1]
           else:
                print "Bounds should either be 1 dimensional (e.g. 1 for all Inputs) or have the size of
                      the input"
                raise(ValueError)
        ...

    def run(self, x0):
        nlp = pyipopt.create(self.n, self.xl, self.xu, self.lenconst, self.gl, self.gu, self.evalf, self
             .gradf,
                             self.evalg, self.gradg, None)
        return nlp.solve(x0)

    def fmin_unc():
        return pyipopt.fmin_unconstrained(self.evalf, self.x0, self.gradf)
```

4.4.5.4 Das Objective Modul

Da nun die Anbindungen für die Optimierung und Simulation besprochen wurden, müssen zusätzlich noch Zielfunktionen definiert werden. Diese dienen als Bindeglied zwischen der Simulationsfunktion und dem eigentlich zu optimierenden Problem.

Hierzu wurde eine Basisklasse geschrieben, welche die grundlegenden Funktionalitäten der Zielfunktionen übernimmt, vgl. `ObjectiveBase` aus Quellcode 4.47. Die gebotenen Funktionen sind Hilfsmethoden für Logging und finite Differenzen, sollte AD nicht eingesetzt werden. Für letztere wird die in Quellcode 4.48 gebotene Funktion genutzt, um einen Numpy-Vektor zu stören. Dabei wird der Vektor `x` an der Stelle `pos` um den Faktor `eps` gestört. Dieser Faktor wird dabei in `_set_dd_stepsize` gesetzt.

$$\mathtt{x}_i = \begin{cases} \mathtt{x}_i + \mathtt{eps} & , \; i = \mathtt{pos} \\ \mathtt{x}_i & , \; \text{sonst} \end{cases}$$

Quellcode 4.47: Basisklasse für Zielfunktionale

```
class ObjectiveBase(Utility):

  def _set_dd_stepsize(self, eps):
      self.dd_stepsize = eps

  def _get_dd_stepsize(self):
      return self.dd_stepsize

  ###### initialize logging (invoked by the objective class on instantiation)
  def init_logging(self, logmode=0):
      self.logmode = logmode  # 0 = no logging
      # 1 = use one log file for x and f(x)
      # 2 = one file for x , multiple files for f(x)
      self.logfile = "objective"
      self.counter = 0

  ###### write intermediate objective function values (invoked by the efcoss)
  def log_result(self, x, res):
      self.counter = self.counter + 1
      if self.logmode > 0:
          if self.counter == 1:
              fd = open(self.logfile, "w")
          else:
              fd = open(self.logfile, "a")
          if self.logmode == 1:
              fd.write(str(self.counter) + " " + str(x) + " " + str(res) + "\n")
          else:
              fd.write(str(self.counter) + " " + str(x) + "\n")
          fd.close()
      if self.logmode == 2:
          writeVector(self.logfile + "_" + str(self.counter), res)
```

Quellcode 4.48: Hilfsfunktion zum Stören eines Vektorelementes für die finite Differenzen Berechnung der Ableitungsobjekte.

```
def perturb(pos, eps, x):
   vector = copy(x)
   vector[pos] = vector[pos] + eps
   return vector
```

Jede der betrachteten Zielfunktionen soll, je nach Art, die folgenden Methoden implementieren.

- `scalarfunction` Funktion zur Berechnung eines skalaren Funktionswerts
- `vectorfunction` Funktion zur Berechnung eines Vektors von Funktionswerten
- `gradient` Funktion zur Berechnung des Gradienten
- `jacobian` Funktion zur Berechnung der Jacobi-Matrix J
- `jacobianvector` Funktion zur Berechnung von $J\mathbf{v}$
- `jacobiantvector` Funktion zur Berechnung von $J^T\mathbf{w}$

4.4.5.5 Skalare Zielfunktionen

Skalare Zielfunktionen zeichnen sich durch ihre einfache Form aus. Sie nehmen einen Vektor als Eingabe und erzeugen ein skalares Ergebnis. Deshalb nutzen wir hier nur die Methoden `scalarfunction` und `gradient`. Die Klasse `Min` aus Quellcode 4.49 ist ein Beispiel dafür. Der Konstruktor benötigt als Eingabe eine EFCOSS-Instanz `efcoss` und eine optionale Objective-Instanz-ID `objective`. Die in der Zielfunktion genutzte Variable `simulation_result` speichert die Ergebnisse der Simulation für die weitere Verwendung im Optimierer.

Quellcode 4.49: Initialisierung der skalaren Minimierungszielfunktion.

```
class Min(ObjectiveBase):

    def __init__(self, efcoss, objectiveFunctionInstance=0):
        self.efcoss = efcoss
        self.objective = objectiveFunctionInstance
        self.simulation_result = []
        for i in range(0, self.efcoss.simulationinstances):
            self.simulation_result.append(self.efcoss.OutputVars[i])
            print self.efcoss.OutputVars[i]
            print "By default all simulation output variables:", getVariableIds(self.efcoss.OutputVars[i
                ])
            print "are considered relevant for computing the objective function"
        self.init_logging()
```

Des Weiteren wollen wir die Skalarfunktion, sowie die Gradientenberechnung für dieses einfache Beispiel ansehen. Die Skalarfunktion aus Quellcode 4.50 zeigt den Funktionsaufruf für die Optimierung. Hierbei wird der Vektor $\mathbf{x}$ eingegeben, welcher den aktuell auszuwertenden Punkt widerspiegelt. Mithilfe der Funktion `completeParameterList` aus EFCOSS kann aus diesem und dem Parametervektor $\mathbf{z}$ die Eingabe der Simulation $\mathbf{w}$ erzeugt werden. Mit Hilfe dieser Eingabe wird das Ergebnis $fvec$ der entsprechenden Simulationsinstanz `instance` durch `Simulation.Function` berechnet und mittels `select` ausgewählt. Zum Schluss wird ein Skalar des Ergebnisses zurückgegeben, welches mit dem Index j ausgewählt werden kann. Aus Performancegründen kann die zeitaufwändige Berechnung der Simulation umgangen werden, indem im Funktionspuffer nach einem Eintrag für den Parametervektor $\mathbf{w}$ nachgeschaut wird, ob für diese Eingaben bereits ein Ergebnis berechnet wurde (vgl. Abschnitt 4.4.3).

Für den Gradienten wird in ähnlicher Form vorgegangen, anstelle von `Simulation.Function` wird jedoch `Simulation.Jacobian` ausgewertet.

Quellcode 4.50: Berechnung der Skalarfunktion der skalaren Minimierungszielfunktion.

```
    def scalarfunction(self, x, j, instance=0):
        print "Evaluating Function"
        w = self.efcoss.completeParameterList(x, self.efcoss.OptVars[self.objective], instance)
        F = self.efcoss.functionBuffer[self.objective][instance].lookup(w)
        if F is None:
            self.efcoss.writelog("Evaluating Simulation at: " + str(w) + "\n")
            fvec = self.efcoss.Simulation[instance][self.objective].Function(w)
            F = self.select(self.simulation_result[instance], fvec)
            self.efcoss.functionBuffer[self.objective][instance].set(w, F)
            print "F=", F, fvec
        return F[j - 1]
```

4.4.5.6 Vektorielle Zielfunktionen

Als vektorielle Zielfunktion bietet EFCOSS die `Identity` Klasse. In Quellcode 4.51 ist die Klasse mit ihrer `vectorfunction` zu sehen. Als Eingaben werden der Vektor x, sowie die erwartete Größe des Ergebnisvektors benötigt. Wieder wird die `completeParameterList` Funktion benötigt und der Funktionspuffer ausgewertet.

Sollte im Puffer kein Wert gefunden werden, wird die Simulationsfunktion `Simulation.Function` aufgerufen. Das Ergebnis wird dann entsprechend der `simulation_result` Variablen ausgewählt und mittels `concatenate` und `append` an das Output-Array F angehängt.

Quellcode 4.51: Berechnung der Vektorfunktion einer vektoriellen Zielfunktion.

```
class Identity(ObjectiveBase):

    def vectorfunction(self, x, m, instance=0):
        w = self.efcoss.completeParameterList(x, self.efcoss.OptVars[self.objective], instance)
        self.efcoss.writelog("Evaluating Simulation " + str(instance) + " at: " + str(w) + "\n")
        fvec = self.efcoss.functionBuffer[self.objective][instance].lookup(w)
        if fvec is None:
            fvec = self.efcoss.Simulation[instance][self.objective].Function(w)
            self.efcoss.functionBuffer[self.objective][instance].set(w, fvec)
            self.efcoss.writelog("Inserted " + str(w) + " into functionBuffer[" + str(instance) + "][" +
                str(
                self.objective) + "]=" + str(fvec) + "\n")
        residual = array([])
        for var in self.simulation_result[instance]:
            vname = var._get_id()
            sim_out = array(concatenate(self.select([var], fvec)), float64)
            residual = append(residual, sim_out)
        F = residual
        if len(F) != m: self.SimulationResultWrongSizeError(m, len(F))
        return F
```

Zusätzlich bietet die Klasse die `jacobian` Methode zur Berechnung der Jacobi-Matrix, sowie Methoden zur Berechnung von $J\mathbf{v}$ bzw $J^T\mathbf{w}$. Exemplarisch sei in Quellcode 4.52 die `jacvec` Funktion für $J\mathbf{v}$ gegeben. Als Eingaben dienen wieder $\mathbf{z}$ und *instance* wie oben. Der Vektor $\mathbf{v}$ wird mit `fvec` übergeben. Es wird wieder die Funktion `completeParameterList` von EFCOSS genutzt. An dieser Stelle muss aus der Simulation jedoch die `JacobianVector` Methode aufgerufen werden. Es sei darauf verwiesen, dass hier keine Speicherung des Ergebnisses in einem Puffer für sinnvoll erachtet wurde. Dies hätte zum Einen eine Umstrukturierung der Buffer Klasse notwendig gemacht, zum Anderen wird die Funktion in der Optimierung mit unterschiedlichen Vektoren $\mathbf{v}$ aufgerufen und damit ist ein Speichern der Ergebnisse nicht sinnvoll.

Quellcode 4.52: Berechnung von $J\mathbf{v}$ in der vektoriellen Zielfunktion.

```
    def jacvec(self, x, fvec, instance=0):
        w = self.efcoss.completeParameterList(x, self.efcoss.OptVars[self.objective])
        matvec, res = self.efcoss.Simulation[instance][self.objective].JacobianVector(w, fvec)
        return matvec, res
```

4.4.5.7 Parameterbestimmung

Für das Parameterfit-Problem

$$\mathbf{r}(\mathbf{f},\mathbf{x},\theta) = \|\mathbf{d}(\mathbf{f},\mathbf{x},\theta)\|^2 = \|\mathbf{f}(\mathbf{x},\theta) - \Phi_{\mathrm{m}}(\mathbf{x})\|^2$$

aus Formel (2.16) wurde eine Basisklasse `DataFit` geschrieben. In Quellcode 4.53 ist die Initialisierungsroutine gegeben. Die Parameter sind hier die EFCOSS-Instanz `efcoss` sowie eine Objective-ID und die `useMatVec` Variable. Letztere wird dazu verwendet, den Gradienten entsprechend Formel 2.17 mit der Matrix-Vektor Operation für $J^T\mathbf{w}$ mittels dem Reverse-Mode des automatischen Differenzierens zu berechnen.

Quellcode 4.53: Initialisierung der Zielfunktion für das Datenfitproblem.

```
class DataFit(ObjectiveBase):
   def __init__(self, efcoss, objective=0, useMatVec=True):
       self.efcoss = efcoss
       self.useMatVec = useMatVec
       self.clearData()
       self._set_model_parameters(self.efcoss.OptVars[objective])
       self.objective = objective
       self.simulation_result = []
       self.init_logging()
```

Um die Berechnung des Residuumvektors durchzuführen, muss zuerst der Vektor der Messwerte Φ eingegeben werden. Dazu bieten sich die Funktionen `addData1d` aus Quellcode 4.54. an. Die Datenfit Klasse `DataFit1d` wird verwendet, um das eindimensionale Problem zu lösen. Das bedeutet, dass sowohl die Simulationsfunktion, als auch die Messwerte als eindimensionaler Vektor vorliegen. Andere Varianten dieses Problems werden wir hier nicht betrachten.

Quellcode 4.54: Eingabe der Messwerte in die Datenfit Klasse `DataFit1d`.

```
class DataFit1d(DataFit):
    # all dictionaries are addressed via the name of the output variable
    def clearData(self):
        self.DataIndices = {}
        self.DataValues = {}
        self.DataWeights = {}

    def addData1d(self, output_variable1d, index_vector, value_vector, weight_vector):
        vname = output_variable1d._get_id()
        if (not vname in getVariableIds(self.simulation_result)):
            self.simulation_result.append(output_variable1d)
        self.DataIndices[vname] = index_vector
        self.DataValues[vname] = value_vector
        self.DataWeights[vname] = weight_vector
```

In Quellcode 4.55 wird die Berechnung der Skalar- und Vektorfunktionen der Klasse gezeigt. Die Skalarfunktion ist dabei durch die Norm der Vektorfunktion implementiert. In der Vektorfunktion nutzen wir zum Einen die Pufferfunktionalität von EFCOSS, um bereits abgelegte Ergebnisse zu speichern. Sollte das Ergebnis nicht bereits im Puffer liegen, so muss aus dem Simulationsinterface die `Function` Subroutine aufgerufen werden. Das Ergebnis der Simulation (oder des Pufferspeichers) wird dann in der Methode `compute_residual` entsprechend Formel (2.16) umgesetzt.

Hierzu werden die in der Klasse gespeicherten Messwerte, sowie deren Indizes und Gewichte verarbeitet. Aus den Simulationsergebnissen werden mittels der Hilfsfunktion `take` diejenigen Werte ausgewählt, deren Index mit denen aus dem Indizes-Array übereinstimmt. Hier erfolgt also gegebenenfalls ein Umsortieren der Ergebnisse. Von diesen in `select` gespeicherten Simulationswerten wird dann der Messdatenvektor `exp_data` abgezogen und der resultierende Vektor mit der Wurzel der Gewichte multipliziert und zurückgegeben.

Quellcode 4.55: Methoden zur Vektor- und Skalarfunktionsberechnung der Datenfit Zielfunktion.

```
def compute_residual(self, fvec, (Indices, Values, Weights), instance=0):
    residual = []
    for var in self.simulation_result:
        vname = var._get_id()
        sim_out = array(concatenate(self.select([var], fvec, instance)), float64)
        indices = array(Indices[vname]) - 1
        exp_dat = array(Values[vname], float64)
        weights = array(Weights[vname], float64)
        select = take(sim_out, indices)
        resid1 = sqrt(weights) * (select - exp_dat)
        residual = residual + (resid1.tolist())  # list concatenation
    return array(residual)

def vectorfunction(self, x, m, instance):
    w = self.efcoss.completeParameterList(x, self.model_parameters, instance)
    self.efcoss.writelog("Evaluating Simulation at: " + str(w) + "\n")
    fvec = self.efcoss.functionBuffer[self.objective][instance].lookup(w)
    if fvec is None:
        fvec = self.efcoss.Simulation[self.objective][instance].Function(w)
        self.efcoss.functionBuffer[self.objective][instance].set(w, fvec)
        self.efcoss.writelog("Inserted " + str(w) + " into functionBuffer[" + str(instance) + "]=" +
             str(fvec) + "\n")
    result = self.compute_residual(fvec, (self.DataIndices, self.DataValues, self.DataWeights),
         instance)
    if len(result) != m: self.SimulationResultWrongSizeError(m, len(result))
    return result

def scalarfunction(self, x, m, instance):
    vec = self.vectorfunction(x, m, instance)
    return array(linalg.norm(vec))
```

Zur Berechnung der Ableitung findet sich die Funktion `gradient` entsprechend der Formel (2.17), gezeigt in Quellcode 4.56. Falls `useMatVec=True` gesetzt ist, wird der Vektor `fvec` über die `vectorfunction` berechnet. Dann nutzen wir `jactvec` entsprechend mit `fvec` aufgerufen, d.h. es wird $jtv = J^T \texttt{fvec}$ berechnet. Sollte der Reversemode des automatischen Differenzierens nicht zur Verfügung stehen (`useMatVec=False`), berechnet diese Funktion die Werte für J und `fvec` über die `jacobian` Funktion. Dabei wird die komplette Jacobi-Matrix aufgestellt und im Anschluss die Matrix-Vektor-Multiplikation $jtv = J^T \texttt{fvec}$ über das Scipy `dot`-Produkt durchgeführt. Die eigentliche Ableitung der Datenfitfunktion findet dann entsprechend der Überlegungen mit $ret = \frac{jtv}{2\|\texttt{fvec}\|}$ statt.

Quellcode 4.56: Berechnung des Gradienten für das Datenfit Problem.

```
def gradient(self, x, m, instance):
    if self.useMatVec:
```

```
            fvec = self.vectorfunction(x, m, instance)
            jtv, foo = self.jactvec(x, fvec, instance)
        else:
            jac, fvec = self.jacobian(x, m, instance)
            jtv = dot(jac.T, fvec)
        ret = jtv / (2 * linalg.norm(fvec))
        return array(ret)

    def jactvec(self, x, fvec, instance):
        w = self.efcoss.completeParameterList(x, self.model_parameters)
        return self.efcoss.Simulation[self.objective][instance].JacobianTVector(w, fvec)
```

4.4.5.8 Optimale Versuchsplanung

Wir wollen nun die optimale Versuchsplanung, Optimal Experimental Design (OED), in EFCOSS beschreiben. Als Basisklasse für die Optimalitätskriterien dient die `OED` Klasse, deren Initialisierung in Quellcode 4.57 zu sehen ist. Die Klasse benötigt nur eine EFCOSS-Instanz und eine optionale Objective-Instanz-ID als Eingabe.

Quellcode 4.57: Basisklasse für die Zielfunktionen des optimal Experimental Designs.

```
class OED(ObjectiveBase):

    def __init__(self, efcoss, objective=0):
        self.efcoss = efcoss
        self.simulation_result = self.efcoss.OutputVars[objective]
        self.clearBuffers()
        self.model_parameters = []
        self.design_parameters = []
        self.clearDesign()
        self.init_logging()
        self.variance = {}
        self.objective = objective
        for v in self.simulation_result:
            self.variance[v._get_id()] = [1.0] * v.size
```

Ein Designkandidat wird mit der Funktion `addDesign` aus Quellcode 4.58 festgelegt. Es werden die zu betrachtenden Eingabevariablen `input_variables`, deren Initialwerte `values` und die Gewichte `weight` als Eingaben benötigt. Nach einer Fehlerabfrage und dem Setzen der VariablenIDs für die Eingabevariablen wird ein `SupportPoint`, dessen Beschreibung in Quellcode 4.59 zu sehen ist, angelegt und die im OED zu untersuchenden Designs aus `self.experimental_design` um diesen Punkt erweitert.

Quellcode 4.58: Basisklasse für die Zielfunktionen der optimalen Versuchsplanung.

```
    def addDesign(self, input_variables, values, weight):
        if sum(getVariableSizes(input_variables)) != len(values):
            raise "Error: the number of scalar design variables does not match the size of the given
                vector"
        ivar_names = getVariableIds(input_variables)
        design_parameter_names = getVariableIds(self.design_parameters)
        if self.design_parameters == []:
            print "Setting design parameters = ", ivar_names
            self.design_parameters = input_variables
```

```
        else:
            if ivar_names != design_parameter_names:
                self.design_parameters = input_variables
        support_point = SupportPoint(input_variables, values, weight)
        self.experimental_design.append(support_point)
```

Quellcode 4.59: Klasse für einen Support-Punkt in der optimalen Versuchsplanung.

```
class SupportPoint:
    def __init__(self,input_variables,values,weight):
        self.input_variables = input_variables
        self.values = values
        self.weight = weight
```

Wie in Abschnitt 2.2.2 beschrieben, unterscheiden sich die unterschiedlichen Optimalitätskriterien zur Berechnung von Formel (2.24),

$$\min_{\mathbf{x}} \Psi(\mathbf{x}, \theta),$$

lediglich in der Weiterverarbeitung der Fisher-Informations-Matrix.

Unsere erste Aufgabe ist es also, diese Matrix zu berechnen. In Formel (2.25) ist die Fisher-Informations-Matrix beschrieben durch

$$F = F(\mathbf{x}, \theta) = \left(\frac{\partial f(\mathbf{x}, \theta)}{\partial \theta} \right)^T W \left(\frac{\partial f(\mathbf{x}, \theta)}{\partial \theta} \right) = \left(Q^T W Q \right).$$

Wir benötigen also die Matrizen $Q = \frac{\partial f(\mathbf{x},\theta)}{\partial \theta}$ und W. Letztere ist dabei nur eine Gewichtsmatrix für die Gewichte w_i der im Design gegebenen Support-Punkte. Die Berechnung ist in Quellcode 4.60 zu sehen. In der `fisher` Methode wird in den Puffern für Q und WQ nach bereits berechneten Werten gesucht. Falls diese nicht vorhanden sind, wird die Methode `compute_Q_and_WQ` ausgeführt. Nachdem die Anzahl der Support-Punkte, Designparameter, Simulationsergebnisse und Modellparameter bestimmt wurde, wird der Eingabevektor `X` in eine passende Matrix `design` umgewandelt. Für jedes der untersuchten Designs `dvector` wird dann das Simulationsinterface für die Berechnung der Ableitungsmatrix `Jacobian` mit der Parameterliste $\mathbf{w} = ($`dvector`$, \theta)$ des aktuellen Designs aufgerufen, welche mit der `completeParameterList` Methode erzeugt wird. Im Anschluss werden die Ergebnisse mittels der `select` Methode entsprechend ausgewählt und in das Array `jac_array` geschrieben. Dieses dient dann als Grundlage für die beiden Ausgabeparameter Q und WQ.

Quellcode 4.60: Basisklasse für die Zielfunktionen der optimalen Versuchsplanung.

```
    def compute_Q_and_WQ(self, X, instance=0):
        N = len(self.experimental_design)  # number of support points
        M = sum(getVariableSizes(self.design_parameters))  # number of designpar.
        L = sum(getVariableSizes(self.simulation_result))  # number of functions
        P = sum(getVariableSizes(self.model_parameters))  # number of model par.
        design = reshape(array(X, float64), (N, M))
        weights = [support_point.weight for support_point in self.experimental_design]
        variance = concat([self.variance[vname]
                           for vname in getVariableIds(self.simulation_result)])
        Q, WQ = zeros((N, L, P), float64), zeros((N, L, P), float64)
        i = 0
        for dvector in design:
```

```
            w = self.efcoss.completeParameterList(list(dvector), self.design_parameters)
            seedmatrix = self.efcoss.seedMatrix(self.model_parameters)
            F, J = self.efcoss.Simulation[instance][self.objective].Jacobian(w, seedmatrix)
            jac = array(self.select(self.simulation_result, J), float64)
            k = self.efcoss.getVectorIDs(self.efcoss._get_OptVars(), self.model_parameters)
            select = jac[0, :, k]
            jac_array = array(select, float64)
            Q[i, :, :] = jac_array.T[:, :]
            WQ[i, :, :] = weights[i].T * jac_array[:, :]
            i = i + 1
        for j in range(L):
            WQ[:, j, :] = WQ[:, j, :] / variance[j]
        Q, WQ = reshape(Q, (N * L, P)), reshape(WQ, (N * L, P))
        self.Q_Buffer.set(X, Q)
        self.WQ_Buffer.set(X, WQ)
        return (Q, WQ)

    def fisher(self, X):
        Q, WQ = self.Q_Buffer.lookup(X), self.WQ_Buffer.lookup(X)
        if ((Q is None) or (WQ is None)):
            (Q, WQ) = self.compute_Q_and_WQ(X)
        QtWQ = dot(Q.T, WQ)
        return QtWQ
```

Ausgehend von der Berechnung der Fisher-Matrix kann für jedes in Abschnitt 2.2.2 vorgestellte Optimalitätskriterium eine Klasse erzeugt werden. Die entsprechenden Skalarfunktionen der Klassen sind in Quellcode 4.61 4.62 und 4.63 zu sehen. Hierbei muss Sorge getragen werden, dass die verwendeten Funktionen wie die Inverse oder die Determinante zulässige Werte liefern. Dies wurde in dieser Ausarbeitung der Übersichtlichkeit halber weggelassen.

Quellcode 4.61: Klasse für das A-Optimalitätskriterium.

```
class A_opt(OED):

    def scalarfunction(self, X, j, instance=0):
        FIM = self.fisher(X)
        inv_F = inv(FIM)
        fcn = trace(inv_F)
        return fcn
```

Quellcode 4.62: Klasse für das D-Optimalitätskriterium.

```
class D_opt(OED):

    def scalarfunction(self, X, j, instance=0):
        FIM = self.fisher(X)
        det_F = det(FIM)
        fcn = -log(det_F)
        return fcn
```

Quellcode 4.63: Klasse für das E-Optimalitätskriterium.

```
class E_opt(OED):
```

```
    def scalarfunction(self, X, j, instance=0):
        FIM = self.fisher(X)
        eigen = [eigvals(FIM)]
        return -1.0 * min(eigen)
```

4.4.5.9 Das Constraints Modul

An dieser Stelle sollen die EFCOSS-Module für die Nebenbedingungen der Optimierungsalgorithmen beschrieben werden. Dabei werden lineare und nichtlineare Nebenbedingungen unterschieden. Für die Implementierung wurde eine Basisklasse geschrieben, welche den Typ der Nebenbedingungen abbildet, siehe hierzu Quellcode 4.64.

Quellcode 4.64: Basisklasse für die Nebenbedingunen.

```
class GenericConstraints:
    class Linear:
        pass

    class Nonlinear:
        pass

class ConstraintsBase(Utility):
    def _get_constraint_type(self):
        return self.constraint_type
```

Sowohl die linearen, als auch die nichtlinearen Nebenbedingungen, welche von der Basisklasse `ConstraintBase` abgeleitet werden, müssen die folgenden Funktionen bieten:

- Gleichheitsbedingung `Eq`
- Ungleichheitsbedingung `Ineq`
- Gradient einer Gleichheitsbedingung `GradEq1`
- Gradient einer Ungleichheitsbedingung `GradIneq1`
- Ableitungsmatrix der Gleichheitsbedingung `JacEq`
- Ableitungsmatrix der Ungleichheitsbedingung `JacIneq`

Quellcode 4.65: Klasse für die linearen Nebenbedingungen.

```
class LinearConstraints(ConstraintsBase):
    def __init__(self, efcoss, A, b, Aeq, beq):
        self.efcoss = efcoss
        self.constraint_type = GenericConstraints.Linear
        self.NumberIneqConstraints = 0
        self.NumberEqConstraints = 0
        if A != []:
            self.IneqConstraintMatrix = array(A, float64)
            self.NumberIneqConstraints, self.NumberVariables = self.IneqConstraintMatrix.shape
            if b == []: b = [0] * self.NumberIneqConstraints
            self.IneqConstraintRHS = array(b, float64)
        if Aeq != []:
            self.EqConstraintMatrix = array(Aeq, float64)
            self.NumberEqConstraints, self.NumberVariables = self.EqConstraintMatrix.shape
            if beq == []: beq = [0] * self.NumberEqConstraints
            self.EqConstraintRHS = array(beq, float64)
```

```
        self.clearBuffers()
```

Quellcode 4.66: Klasse für die nichtlinearen Nebenbedingunen.

```
class NonlinearConstraints(ConstraintsBase):
    def __init__(self, efcoss, IneqConstraintVars, EqConstraintVars):
        self.efcoss = efcoss
        self.constraint_type = GenericConstraints.Nonlinear
        self.IneqConstraintVars = IneqConstraintVars
        self.EqConstraintVars = EqConstraintVars
        self.NumberIneqConstraints = sum(getVariableSizes(IneqConstraintVars))
        self.NumberEqConstraints = sum(getVariableSizes(EqConstraintVars))
        self.clearBuffers()
```

4.4.6 Parallele EFCOSS Instanzen

Sollte die Parallelisierung der Simulation nicht vorhanden oder limitiert sein, bietet es sich an, verschiedene EFCOSS Instanzen parallel zu betreiben, um die gegebenen parallelen Ressourcen effektiver zu nutzen. Dies ist sinnvoll, wenn die gegebenen Probleme für einen großen Parameterraum untersucht werden sollen. Solche Probleme sind mit der neuen Struktur von EFCOSS ohne Probleme umsetzbar, jedoch müssen einige Vorkehrungen getroffen werden, damit diese ordentlich durchlaufen.

In vielen Simulations-Codes werden Zwischenergebnisse in Dateien geschrieben, die später unter Umständen wieder gelesen werden. Hier muss der zuständige (Simulations-)Programmierer die Ausgaben entsprechend der parallelen Ränge in unterschiedliche Dateien umleiten und von diesen wieder lesen.

Um darüber hinaus die Ausgaben auf der Kommandozeile unterschiedlicher Simulationsläufe einzeln abzuspeichern und damit einen Überblick über den Ausführungsverlauf zu erhalten, kann man den Python-Code Quellcode 4.67 im Konstruktor der Klasse verwenden.

Quellcode 4.67: Setzen des Standardoutputs in eine Datei.

```
def __init__(self,...,id):
    ...
    self.new_fds = [os.open(’proc_%(x)s.out’ % {’x’: self.id}, os.O_RDWR) for x in xrange(2)]
    self.save = os.dup(1), os.dup(2)
    os.dup2(self.new_fds[0], 1)
    os.dup2(self.new_fds[1], 2)
```

Es werden die zwei Ausgabestreams für Std-Out (`os.dup(1)`) und Std-Err (`os.dup(2)`) gespeichert und durch neu erstellte Dateideskriptoren (`new_fds`) ersetzt. Diese sind Dateien, welche durch eine eindeutige Identifikationsnummer (`id`) gekennzeichnet werden. Hier bietet sich zum Beispiel die Verwendung des Ranges des Prozesses an.

Nachdem die Ausführung beendet wurde, sollte demnach aufgeräumt und die gespeicherten Ausgabestreams zurückgesetzt werden. Dies sollte im Destruktor der Klasse passieren. Dies ist in Quellcode 4.68 zu sehen.

Quellcode 4.68: Zurücksetzen der Standardausgabe beim Beenden des Programms.

```
def __del__(self):
```

```
    os.dup2(self.save[0], 1)
    os.dup2(self.save[1], 2)
    os.close(self.new_fds[0])
    os.close(self.new_fds[1])
```

4.5 Multi-EFCOSS

Unter Multi-EFCOSS werden alle Programmteile für EFCOSS subsummiert, die mindestens zwei verschiedene Simulationsfunktionen oder die mindestens zwei verschiedene Zielfunktionen benutzen. Zu diesen zählen Space-Mapping, Modelldiskriminierung und Modellidentifikation.

4.5.1 Space-Mapping

Wie in Abschnitt 2.4 geschrieben, müssen wir Funktionen für ein feines und ein grobes Modell bereitstellen. Dazu werden zwei Simulationsinstanzen benötigt, d.h. bei der Initialisierung von EFCOSS muss `instances=2` gesetzt werden.

Es wurde eine Basisklasse für die Space-Mapping Methoden geschrieben, `SpaceMappingBasic`, deren Initialisierungsroutine in Quellcode 4.69 zu sehen ist. Die Eingaben der Initialisierungsfunktion sind:

- `efcoss` EFCOSS Instanz
- `x0` Startwert
- `epsilon` Genauigkeit
- `maxK` Maximale Iterationen
- `boundsMin` Lower Bounds
- `boundsMax` Upper Bounds
- `name` Name der Space-Mapping-Instanz
- `useAD` Soll Jacobi mit AD berechnet werden?
- `optimizer` Der im Space-Mapping benutzte Optimierer

Die Standardwerte sind entsprechend gegeben, so dass man im einfachsten Fall nur eine EFCOSS Instanz, den Startwert und eine Genauigkeit angeben muss.

Quellcode 4.69: Initialisierungsroutine für die Basisklasse des Space-Mappings.

```
class SpaceMappingBasic:
    def __init__(self, efcoss, x0, epsilon, maxK=100, boundsMin=None,
            boundsMax=None, name='', useAD=[False, False],
            optimizer=ScipyOptimizer(tol=1e-13, method='L-BFGS-B',
                options={'disp': True, 'ftol': 1e-10, 'eps': 1e-18,
                'gtol': 1e-22, 'maxls': 50}), useJacTVec=False):
```

EFCOSS muss dabei so konfiguriert werden, dass es zwei Simulationsinstanzen, sowie zwei Zielfunktionen benutzt. Die Simulationsinstanz mit Index 0 soll dabei die grobe Simulation berechnen, Index 1 die feine. Als Zielfunktionen setzen wir die Klasse `Datafit1D` für `objectivefunction=0`, im Folgenden die Funktion $\mathbf{f}_0$ und Klasse `Identity` für `objectivefunction=1`, im Folgenden die Funktion $\mathbf{f}_1$.

Um nun den Algorithmus 8 zu implementieren, benötigen wir die folgenden Methoden:

1. `distCoarse` – Evaluierung von $\mathbf{f}_0 = \mathbf{R}_c(\mathbf{x}_c) - \mathbf{y}$ mit `objectiveFunction=0`
2. `distFineCoarse` – Evaluieren von $\mathbf{f}_1 = \mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(\mathbf{x}_f)$ mit `objectiveFunction=1`
3. `evalGCoarse` – Zur Berechnung der Ableitungen nach $\mathbf{x}_c$.

In Quellcode 4.70 sind die entsprechenden Funktionen gelistet. So wird `evalCoarse` zur Auswertung des groben Modells genutzt, `evalFine` für das feine Modell. Für die benötigten Ableitungen findet sich die `evalGCoarse` Methode. Abhängig von der verwendeten `objectiveFunction` ist der Funktionswert $\mathbf{f}$ entweder $\mathbf{f}_c$ oder $\mathbf{f}_f$. Die Ableitungen sind somit

$$\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_c} = \frac{\frac{\partial \mathbf{R}_c}{\partial \mathbf{x}_c}^T \mathbf{f}_0(\mathbf{x}_c)}{\|\mathbf{f}_0(\mathbf{x}_c)\|} = \frac{J^T \mathbf{f}_0(\mathbf{x}_c)}{\|\mathbf{f}_0(\mathbf{x}_c)\|} \tag{4.4}$$

für `objectiveFunction=0`, beziehungsweise

$$\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_c} = \frac{\frac{\partial \mathbf{R}_c}{\partial \mathbf{x}_c}^T \mathbf{f}_1(\mathbf{x}_c, \mathbf{x}_f)}{\|\mathbf{f}_1(\mathbf{x}_c, \mathbf{x}_f)\|} \tag{4.5}$$

für `objectiveFunction=1`.

Dabei können wir uns abhängig von der Bool-Variablen `useJacTVec` entscheiden, ob die Ableitungen des Reverse-Modus für die Berechnung genutzt werden sollen, oder ob die komplette Jacobi-Matrix für die Berechnung genutzt werden soll. Hierbei werden die Ableitungen mit dem Vorwärts-Modus in `evalfjac` berechnet. Danach muss das Matrix-Vektorprodukt noch mit der Numpy Funktion `dot` berechnet werden. Sonst wird die Funktion `evaljactvec` verwendet. In beiden Fällen wird das Ergebnis in der Variablen `matvec` gespeichert, mit der Norm des vorher berechneten Funktionswertes dividiert und als Rückgabewert verwendet.

Quellcode 4.70: Methoden für die Evaluierung von grobem und feinem Modell, sowie deren Ableitung.

```python
def evalCoarse(self, x, objectiveFunction=0):
    res = self.efcoss.evalfvec(self.n, x, self.m, simulationInstance=0, objectiveFunction)
    return res

def evalFine(self, x, objectiveFunction=0):
    res = self.efcoss.evalfvec(self.n, x, self.m, simulationInstance=1, objectiveFunction)
    return res

def evalGCoarse(self, x, x_f=0, objectiveFunction=0):
    if (objectiveFunction == 0):
        f = self.evalCoarse(x, 0)
    else:
        f = self.evalCoarse(x, 1) - self.evalFine(x_f, 1)
    if self.useJacTVec:
        jac = self.efcoss.evalfjac(self.n, x, self.m, simulationInstance=0, objectiveFunction)
        matvec = numpy.dot(f, jac)
    else:
        matvec=self.efcoss.evaljactvec(self.n,x,self.m,f,objectiveFunction)
    ret = numpy.array(matvec / 2*numpy.linalg.norm(f))
    return ret
```

Mit diesen Funktionen können nun alle im Algorithmus vorkommenden Minimierungsfunktionen durchgeführt werden. Dazu wurden weiter die Methoden in Quellcode 4.71 implementiert. So haben wir für die erste Funktion zur Berechnung von $\mathbf{f}_0$ die Methode `optimizeCoarse`. Diese nutzt die Funktion `distCoarse`, welche die Norm von `evalCoarse` bildet, als Zielfunktion. Die Methode `optimizeFineCoarse` nutzt die Methode `distFineCoarse` als Zielfunktion, welche die Funktion $\mathbf{f}_1$ implementiert. Wichtig dabei ist, dass die `objectiveFunction` entsprechend in den `dist`-Funktionen gesetzt wird.

Quellcode 4.71: Optimierungsfunktionen für Space-Mapping.

```
def distCoarse(self, x):
    return numpy.linalg.norm(self.evalCoarse(x))

def distFine(self, x):
    return numpy.linalg.norm(self.evalFine(x))

def distFineCoarse(self, z, x, args=0):
    res = self.evalCoarse(z, 1) - self.evalFine(x, 1)
    return numpy.linalg.norm(res)

def optimizeCoarse(self):
    return self.optimizer.run(self.distCoarse, self.x[0], jac=self.useAD[0])

def optimizeFineCoarse(self, x, k=0):
    return self.optimizer.run(self.distFineCoarse, self.x[k + 1], args=(x, 1,), jac=self.useAD[1])
```

Die Klasse `ASM`, aus Quellcode 4.72, welche von `SpaceMappingBasic` abgeleitet ist, implementiert nun das in Abschnitt 2.4 unter Algorithmus 7 gezeigte Programm. Dazu bieten sich drei Methoden, `run`, `step` und `initRun`. Die Methode `run` beinhaltet dabei den Aufruf von außen, die anderen Methoden werden von ihr aufgerufen. Für die Initialisierung mittels des Algorithmus 8 (ebenfalls aus Abschnitt 2.4) wird die Methode `initRun` genutzt. Das Innere der While-Schleife wurde aber in der Methode `step` ausgelagert, da dieser Teil auch in der Trust-Region Variante genutzt werden kann.

Quellcode 4.72: Die Klasse ASM zur Lösung des Aggressive-Space-Mapping Algorithmus.

```
class ASM(SpaceMappingBasic):
    def step(self, hFac, k):
        h = -hFac.dot(self.p[k] - self.z_st)
        self.x[k + 1] = self.x[k] + h
        self.ffun[k + 1] = self.distFine(self.x[k + 1])
        newP = self.optimizeFineCoarse(self.x[k + 1], k)
        self.p[k + 1] = newP.x
        self.B = self.B + numpy.outer((self.p[k + 1] - self.z_st), h) / (h * h.getT())

    def run(self):
        k = 0
        self.initRun()
        while (numpy.linalg.norm(self.p[k] - self.z_st) > self.epsilon and k < self.maxK - 1):
            hfac = numpy.linalg.pinv(self.B)
            self.step(hfac, k)
            k = k + 1
        return [self.x, self.p, k, self.z_st, self.ffun]

    def initRun(self):
        self.z_st = self.optimizeCoarse().x
        self.ffun[0] = self.distFine(self.x[0])
        finecoarse = self.optimizeFineCoarse(self.x[0])
```

```
        self.p[0] = finecoarse.x
```

4.5.2 Modelldiskriminierung

Um die Modelldiskriminierung in EFCOSS zu implementieren, werden wir zuerst ein allgemeines Modell definieren. Dieses Modell dient als Grundlage für die weiteren Überlegungen zur Modelldiskriminierung, sowie der Modellidentifikation mit zum Beispiel dem Akaike Informationskriterium (AIC). In Quellcode 4.73 ist eine solche Klasse zu sehen. Dabei fällt die Verwendung der `optimize` Routine auf, welche die Zielfunktion für das gegebene Modell mit der Instanz-Nummer `instance` optimiert. Der verwendete Optimierer hier ist eine einfache Instanz von `ScipyOptimizer`. Hierbei wird die Funktion `ffun`, sowie deren Ableitungen `fjac` verwendet.

Quellcode 4.73: Definition einer Klasse für die Modelle zur Verwendung in der Modellidentifikation.

```
class Model:
    def __init__(self, efcoss, instance, optimizer=ScipyOptimizer()):
        self.efcoss = efcoss
        self.instance = instance
        self.m = self.efcoss.getResultVec().size
        self.x0 = self.efcoss.getOptVec(self.instance)
        self.n = self.x0.size()
        self.optimizer = optimizer

    def ffun(self, x0, objective=0):
        return self.efcoss.evalfvec(self.n, x0, self.m, self.instance, objective)

    def fjac(self, x0, objective=0):
        return self.efcoss.evalfjac(self.n, x0, self.m * len(self.x0), self.instance, objective)

    def optimize(self):
        res = self.optimizer.run(self.ffun, self.x0,jac=self.fjac)
        return res
```

Um nun die Modelldiskriminierung durchzuführen, wurde die in Quellcode 4.74 gezeigte Basisklasse `ModelDiscrimination` implementiert. Hierbei muss eine EFCOSS-Instanz, sowie die Anzahl der Modelle eingegeben werden. Sie speichert dann für jedes Modell eine eigene Instanz von `Model` in `self.models`. Für jedes dieser Modelle wird dann in `findResiduals` eine Optimierungsfunktion durchgeführt. Um das beste Modell zu finden, muss der Benutzer dann nur noch die Methode `findBestModel` aufrufen.

Quellcode 4.74: Die Klasse `ModelDiscrimination` zur Bestimmung des besten Modells.

```
 class ModelDiscrimination(ProblemDefinition):
    def __init__(self, efcoss, models=2):
        self.efcoss = efcoss
        for i in range(0, models):
            self.models.append(Model(self.efcoss, i))

    def findResiduals(self):
        self.residuals = [];
        for m in self.models:
            self.residuals.append(m.optimize())
```

```
    def findBestModel(self):
        r = []
        bestValue = float(inf)
        bestModel = -1
        for i, res in enumerate(self.residuals):
            val = self.fun(res, i)
            r.append(val)
            if bestValue > val:
                bestModel = i
                bestValue = val
        return [bestModel, bestValue, r, self.residuals]
```

Die Modelldiskriminierung wird weiter in zwei verschiedene Methoden unterschieden, Akaike (AIC) und Minimale Beschreibungslänge (MDL). Diese unterscheiden sich jeweils in der ausgeführten Funktion `fun`, welche für AIC die Formel (2.29), sowie für MDL die Formel (2.30) implementiert. Beide sind in Quellcode 4.75 zu sehen.

Quellcode 4.75: Modelldiskriminierung mit AIC und MDL.

```
class AIC(ModelDiscrimination):
    def fun(self, res, i):
        n = self.models[i].getN()
        m = self.models[i].getM()
        return n ∗ log(res.fun / n) + 2 ∗ m

class MDL(ModelDiscrimination):
    def fun(self, res, i):
        n = self.models[i].getN()
        m = self.models[i].getM()
        return 1 / (2 ∗ n) ∗ res.fun ∗ exp((log(n) ∗ (m - 1)) / (n - m - 2))
```

Kapitel 5

Anwendungsszenarien

In diesem Kapitel sollen die im Rahmen der Arbeit untersuchten inversen Problemstellungen an ausgewählten Beispielen numerisch ausgewertet und auf ihre Performance hin untersucht werden. Hierbei spielen sowohl die Genauigkeit, als auch die benötigte Rechenzeit auf unterschiedlichen Architekturen eine wichtige Rolle. Wir beginnen mit einem Überblick über die verwendeten Rechenanlagen, dann wird ein weiteres Beispiel aus der Minpack-2 Testsuite betrachtet. Danach werden ausgewählte Probleme der Geothermie an zwei Beispielen im Sinne der Parameter-Bestimmung, Space-Mapping und optimale Versuchsplanung untersucht. Im Anschluss wird die Modellidentifikation an einem Beispiel aus der Metallplastizität besprochen.

5.1 Verwendete Rechenanlagen

Die in dieser Arbeit gelisteten Ergebnisse wurden unter Verwendung der folgenden Rechenanlagen erzielt.

IBM Thinkpad T480s Das im Rahmen meiner Anstellung mir zur Verfügung gestellte Notebook, mit den folgenden Spezifikationen:

- Intel Core i7-8550U 1.8 GHz Quad-Core Prozessor mit Hyperthreading
- 16 GB RAM
- 1000 GB SSD
- Ubuntu 19.10

Dieses Gerät wurde maßgeblich für die Entwicklung und kurze Testläufe verwendet.

AMD Threadripper Im Rahmen der Lehre und Forschung wurden Grafikkarten Rechner angeschafft. Einer davon basiert auf der neueren Ryzen Architektur von AMD.

- AMD Ryzen Threadripper 1950X 16-Core Prozessor mit Hyperthreading
- 64 GB RAM
- 500 GB HDD
- Ubuntu 19.10

Lofar Cluster Der Lofar Cluster ist ein Rechnerverbund, welcher zur Erfüllung des Projektes Lofar im Jahr 2009 angeschafft wurde. Er besteht aus 17 Rechenknoten, einem Kopfknoten (Headnode), sowie einem Serviceknoten. Die Rechenknoten sind über ein 40 GBit/s Infiniband Netzwerk verbunden. Ein Gigabit Ethernet Netzwerk ist für die Serviceaufgaben vorhanden.

Die Rechenknoten haben die folgende Ausstattung:

- 2 x AMD Opteron 2378 2.4 GHz Quad-Core Prozessor
- 16 GB RAM
- Debian Sid

Der Headnode beinhaltet ebenso die clusterweit verfügbaren Home-Verzeichnisse mit 5.5 TB Bruttokapazität.

ScaleMP Cluster ScaleMP ist ein Rechnerverbund im Rechenzentrum der RWTH Aachen, welcher für den Benutzer wie ein einziger großer Rechner mit gemeinsamem Speicher sichtbar ist. Dennoch besteht der Rechner aus insgesamt 16 Rechenknoten, welche über QDR Infiniband miteinander verbunden sind. Jeder dieser Rechenknoten besteht aus

- 4 x Intel Nehalem EX Xeon X7550 2.0 GHz 8-Core Prozessor
- 256 GB RAM
- CentOS

Das heißt, der virtuelle Rechner enthält 512 Kerne mit 4 TB RAM. Wichtig bei diesem System ist jedoch, auf die sogenannte NUMA (Non-Uniform-Memory-Architecture) zu achten. Diese bezeichnet unterschiedliche Speicherzugriffszeiten von verschiedenen Kernen auf den gleichen Speicherbereich.

ARA Cluster Im Rahmen einer Ausschreibung wurde an der Friedrich-Schiller-Universität Jena eine Hochleistungsrechenanlage angeschafft, welche einige Zeit später erweitert wurde. Der Cluster besteht aus den folgenden Komponenten:

- 136 Broadwell-Rechenknoten mit je
  - 2x Intel Xeon E5-2650 12-Core
  - 128 GB DDR4-RAM
  - CentOS
- 166 Skylake-Rechenknoten mit je
  - 2x Intel Xeon Gold 6140 CPU 18-Core
  - 192 GB DDR4-RAM
  - CentOS
- 4 Skylake-Knoten mit großem Arbeitsspeicher mit je
  - 4x Intel Xeon Gold 6130 CPU 16-Core
  - 1.5 TB RAM
  - CentOS
- 2 Login-Knoten
- 80 TB Home-Directory-Storage mit 2 GB/s Datendurchsatz
- 524 TB Parallel-Storage mit 8 GB/s Datendurchsatz
- Intel Omni-Path 100 GBit/s Hochgeschwindigkeitsnetzwerk
- 10 GBit/s Ethernet

5.2 Minpack-2 Testsuite

Verschiedene Probleme der Minpack-2 Testsuite wurden mithilfe der neuen EFCOSS Version an diversen Optimierern getestet. Das Exponential Data-Fitting 1 Problem aus Abschnitt 2.2.1 wurde bereits im Kapitel 4 als Anwendungsbeispiel vorgestellt. Hier soll nun beispielhaft die Anbindung an das Problem der Enzymreaktion aus Abschnitt 2.2.1 gezeigt werden. In Quellcode 5.1 ist der zugrunde liegende Fortran-Code gegeben.

Quellcode 5.1: Fortran Implementierung des Enzymreaktionsproblems aus [8].

```
subroutine enzyme(x,fvec,n,m)
   double precision, intent(in):: x(n)
   double precision, intent(out):: fvec(m)

   integer:: i
   double precision:: temp1, temp2
   double precision:: v(11), y(11)

   data v/4.0d0, 2.0d0, 1.0d0, 5.0d-1, 2.5d-1, 1.67d-1, 1.25d-1, &
        1.0d-1, 8.33d-2, 7.14d-2, 6.25d-2/
   data y/1.957d-1, 1.947d-1, 1.735d-1, 1.6d-1, 8.44d-2, 6.27d-2,&
        4.56d-2, 3.42d-2, 3.23d-2, 2.35d-2, 2.46d-2/

   do 10 i = 1, m
     temp1 = v(i)*(v(i)+x(2))
     temp2 = v(i)*(v(i)+x(3)) + x(4)
     fvec(i) = y(i) - x(1)*temp1/temp2
   10 continue

end
```

Um diesen Code mittels Tapenade abzuleiten, verwenden wir die folgenden Befehle für die verschiedenen Ableitungsfunktionen in einem Makefile:

```
AD_TOP=enzyme
tapenade: $(AD_TOP).f90
    tapenade -d -head $(AD_TOP) $(AD_TOP).f90
    tapenade -d -multi -head $(AD_TOP) $(AD_TOP).f90
    tapenade -b -head $(AD_TOP) $(AD_TOP).f90
```

Die damit generierten Dateien sind `enzyme_d.f90` (Forward-Modus), `enzyme_dv.f90` (Forward-Modus mehrdimensional) sowie `enzyme_b.f90` (Reverse-Mode). Man beachte, dass Tapenade hier die Bestimmung der aktiven und passiven Variablen überlassen wurde. Um den so generierten Code zu nutzen, wird mit dem Make-Target

```
library:
    gfortran -c -fPIC $(AD_TOP).f90
    gfortran -c -fPIC $(AD_TOP)_d.f90
    gfortran -c -fPIC $(AD_TOP)_dv.f90
    gfortran -c -fPIC $(AD_TOP)_b.f90
    ar -r lib$(AD_TOP) $(AD_TOP).o $(AD_TOP)_d.o $(AD_TOP)_dv.o $(AD_TOP)_b.o
```

eine Bibliothek libenzyme.a generiert, welche im Folgenden Verwendung findet.

5.2.1 Anbindung an EFCOSS

Zur Anbindung an EFCOSS wurde die in Quellcode 5.2 gezeigte Klasse `Enzyme` geschrieben. Sie umfasst die `initEFCOSS()`, `setObjectiveFunction()`, sowie `runOptimizer` Routine. Alle sind entsprechend den Überlegungen in Abschnitt 4.2.2 ausgeführt.

Quellcode 5.2: Problemdefinition für die Enzymreaktion aus [8].

```
from EFCOSS import ProblemDefinition

class Enzyme(ProblemDefinition):
    def initEFCOSS(self,x0=[2.5E-1,3.9E-1,4.15E-1,3.9E-1]):
        self.n=self.efcoss.newInputVariable("n",4)
        self.m=self.efcoss.newInputVariable("m",11)
        self.x=self.efcoss.newInputVariable("x",x0)
        self.fvec=self.efcoss.newOutputVariable("fvec",self.m)

        self.efcoss.setSimulationCallingSequence([self.x,self.fvec,self.n,self.m])

        self.efcoss.setOptVars([self.x])

    def setObjectiveFunction(self):
        obj=self.efcoss.setObjectiveFunction("ScalarObjective","Norm")

    def runOptimizer(self,x0=None):
        from EFCOSS.Optimization.opt_scipy import *
        if x0 is None:
            x0=self.efcoss.getOptVec()
        optimizer=opt_scipy(self.efcoss,tol=1e-10,bounds=[(None,None),(None,None),(0,None),(0,None)])
        ret=optimizer.minimize(x0)
        print ret
```

Das Runtime-Script für die Generierung und Ausführung ist in Quellcode 5.3 zu sehen. Zuerst wird eine Instanz der Enzyme Klasse angelegt, dann EFCOSS initialisiert und das Simulations-Interface generiert und compiliert. Hierbei ist zu beachten, dass der AD-Reverse Mode aktiviert ist (`adreverse=True`). Dieser Schritt muss natürlich nicht in jedem Aufruf des Scripts passieren, sondern nur, wenn sich der zugrunde liegende Simulations-Code ändert. Danach kann die Simulation initialisiert und die Zielfunktion gesetzt werden. Zum Schluss kann der der Optimierer ausgeführt werden.

Quellcode 5.3: Ausführungsscript für das Enzymreaktionsproblem zur Lösung mit EFCOSS.

```
import numpy
from Enzyme import Enzyme

problem=Enzyme("enzyme")
problem.initEFCOSS()
problem.generateSimulationInterface(adtool="tapenade",adlevel=1,adreverse=True)
problem.compileSimulationInterface()
#testing the interfaces
import enzyme
problem.initSimulationInterface()
problem.setObjectiveFunction()
problem.runOptimizer()
```

5.2.2 Resultate

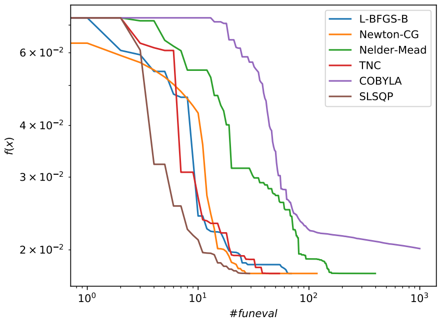
In Tabelle 5.1 sind die Ergebnisse einiger in `scipy.optimize` enthaltener Optimierer zu sehen. Dabei wurde die Toleranz auf 10^{-8} gesetzt. Wie zu sehen ist, terminieren die Löser L-BFGS-B, Nelder-Mead, TNC und SLSQP. Die Löser Newton-CG und COBYLA berichten einen nicht-

Tabelle 5.1 Resultate ausgewählter Optimierer aus `scipy.optimize.minimize()` für das Enzymreaktionsproblem aus [8].

Optimierer	Erfolg	$\|x^*\|$	$\|f^*(x^*)\|$	$\|\nabla f^*\|$	# f	# ∇f	# Iter	Begründung
L-BFGS-B	0	0.3277	0.01753	7.1166e-05	70	N/A	43	CONVERGENCE: REL REDUCTION OF F<= FACTR*EPSMCH
Newton-CG	2	0.3277	0.0175	3.1208e-08	119	318	27	Warning: Desired error not necessarily achieved due to precision loss.
Nelder-Mead	0	0.3277	0.0175	N/A	398	N/A	234	Optimization terminated successfully.
TNC	1	0.3277	0.0175	2.394e-05	54	N/A	13	Converged ($\|f_n - f_{n-1}\| \approx 0$)
COBYLA	2	0.6764	0.0201	N/A	1000	N/A	N/A	Maximum number of function evaluations has been exceeded.
SLSQP	0	0.3278	0.0175	0.0001	30	21	21	Optimization terminated successfully.
Basin	N/A	0.3277	0.0175	N/A	4994	N/A	100	Requested number of basinhopping iterations completed successfully

erfolgreichen Lauf. Dabei ist zu beachten, dass Newton-CG hier nur eine Warnung ausspricht, dass die Fehlertoleranz nicht erreicht werden kann. Die Konvergenz von COBYLA verläuft hingegen sehr langsam, sodass mit den eingestellten 1000 Funktionsauswertungen kein Minimum gefunden werden kann. In Abb. 5.1 ist dieses Verhalten zu beobachten. Es fällt außerdem auf, dass die am schnellsten konvergierenden Algorithmen SLSQP und Newton-CG sind, dicht gefolgt von TNC und L-BFGS-B. Der Newton-CG Algorithmus ist sich aber wie bereits erwähnt nicht sicher und bricht entsprechend ohne Erfolgsmeldung ab. Die Ergebnisse des Basinhopping Algorithmus wurden aus Gründen der Übersichtlichkeit weggelassen.

Abbildung 5.1 Minimaler Funktionswert in Abhängigkeit der Funktionsauswertungen des Enzymreaktionsproblems für verschiedene Optimierungsalgorithmen.



5.3 SHEMAT

In diesem Abschnitt werden wir die Anbindung von SHEMAT, der *Simulation for Heat and Mass Transport*, an EFCOSS zeigen. Es bietet sich hier ein gemeinsames Interface an, da diese in vielen Anwendungen des Geothermie-Projektes benutzt werden sollen.

5.3.1 Überblick über SHEMAT

SHEMAT ist eine umfangreiche Simulationsumgebung für den Einsatz in geologischen Problemstellungen wie Grundwasserströmung oder Wärme- und Massentransport, sowie Mehrphasenmodelle.

SHEMAT ist vollständig in der Programmiersprache Fortran geschrieben. Es umfasst die Vorwärtssimulation sowie verschiedene inverse Problemstellungen. Für diese kann der Code mit den AD-Tools Tapenade oder TAF abgeleitet werden. Als Parallelisierungsmöglichkeiten werden OpenMP [9] und MPI [71] genutzt. Die OpenMP Implementierung ist dabei die weitaus ausgereiftere und kann auch für inverse Problemstellungen genutzt werden. Die im Projekt MeProRisk II implementierte MPI-Parallelisierung ist primär für den Einsatz mit Monte-Carlo und Ensemble-Kalmann-Filter-Problemstellungen der Parameterschätzung gedacht. Es existiert eine Vorwärtssimulation mit MPI, jedoch ist diese für den Einsatz in inversen Problemen nur unzureichend getestet und instabil, weshalb wir diese Version für unsere Implementierung, besonders im Hinblick auf AD nicht nutzen werden.

Der in SHEMAT implementierte Algorithmus wurde in Abschnitt 2.1.2 gezeigt. Zusätzlich sind verschiedene Methoden vorhanden, welche in Abb. 5.2 gezeigt sind. So benötigt eine Simulation zwangsläufig eine Einleseroutine für die Modellbeschreibung, welche hier durch eine einfache Textdatei dargestellt ist. In dieser können alle Einstellungen vorgenommen werden, um eine Simulation durchzuführen. Es ist jedoch zu beachten, dass sehr große Simulationsmodelle nicht komplett in dieser Datei liegen sollten, da die Einleseroutine sehr langsam ist. Aus diesem Grund bietet SHEMAT die Möglichkeit HDF5 Dateien [72] für große Datenfelder einzubinden.

Zusätzlich zu den Eingaben kann die Simulation über verschiedene Module gesteuert werden. Diese Module werden unterschieden in die sogenannten USER Module, in denen Brunnen und Pumpvorgänge beschrieben werden können, und die sogenannten PROPS Module, in denen verschiedene physikalische Kopplungen und Vorgänge beschrieben werden.

Die in dieser Arbeit genutzten USER Module sind

- none - Einfaches Dummy-Modul
- wells3D - Standard Benutzer-Modul für die AD Ableitungsgenerierung

Die genutzten PROPS Module sind

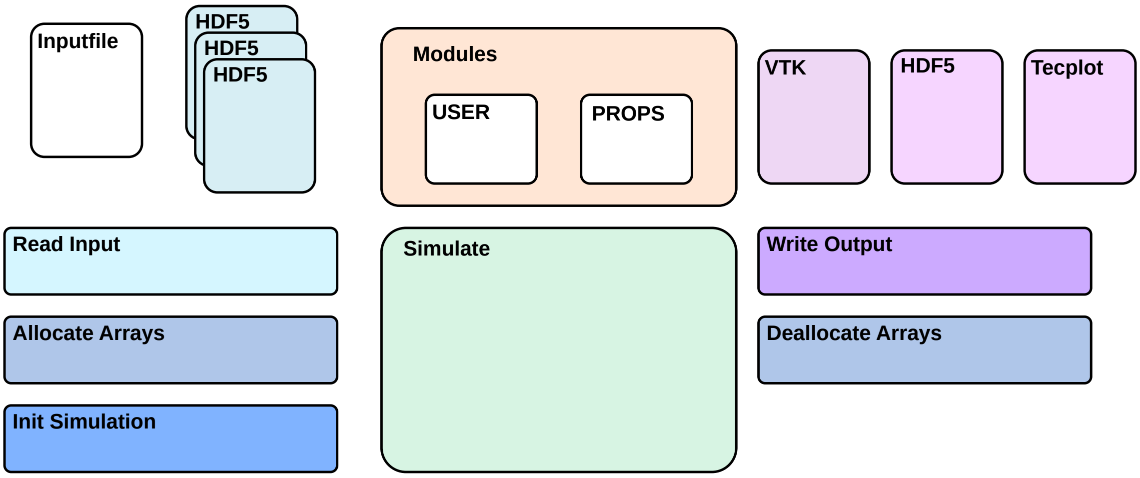
- const - Alle Modellparameter werden über die Simulationslaufzeit konstant gehalten
- bas - Standard PROPS Modul für die AD Ableitungsgenerierung
- toscana[1] - Spezielles Modul für das Toskana-Modell. Der Modellparameter der Permeabilität wird in diesem speziellen Modell in Abhängigkeit der Tiefe logarithmisch skaliert.

Nach dem Einlesen der Aufgaben müssen die entsprechenden Datenfelder angelegt und mit Werten gefüllt werden. Anschließend wird die Simulation durchgeführt. In jedem Zeitschritt werden die nichtlinearen gekoppelten Gleichungen iterativ gelöst. Dabei werden verschiedene lineare Gleichungen mit Hilfe von dünnbesetzten Matrix-Operationen gelöst. Durch die eingelesenen Informationen

---

[1] Name geändert Aufgrund der Geheimhaltungsklausel.

Abbildung 5.2 Überblick über die Komponenten von SHEMAT.



der Modellbeschreibung wird die Ausgabe in verschiedene Formate sowie die Ausgabehäufigkeit, also Anzahl der Zeitschritte zwischen den Ausgaben, gesteuert.

Im Code-Verzeichnis finden sich die folgenden wichtigen Verzeichnisse:

- `forward` - Quelldateien für die Vorwärtsrechnung
- `inverse` - Quelldateien für die Inverse-Rechnung
- `nonlinear` - Nichtlineare Löser
- `solve` - Lineare Löser
- `props` - Module für Gesteinseigenschaften
- `user` - Module für Brunnendefinitionen

Für die Berechnungen werden verschiedene Arrays benötigt, welche in `forward/arrays.f90` definiert sind. Eingaben:

- `nunits` – Anzahl der Gesteinsschichten
- `kx` – Permeabilität in x-Richtung

Die in dieser Arbeit betrachteten Ausgaben sind

- `head`/`pres` – hydraulisches Potential / Druck
- `temp` – Temperatur
- `conc` – Salzkonzentration

Die Datei `oed_inv`, welche eine Problemdefinition für das Perth Modell Abschnitt 5.3.5.1 beinhaltet, wird hier zu Demonstrationszwecken gezeigt. In Quellcode 5.4–5.10 ist die Datei, in verschiedene Bereiche zerlegt, gezeigt. Die meisten der Attribute haben zwei oder mehr Zeilen. In der ersten Zeile des Attributes steht das Keyword, beginnend mit `#`. Dann kommen die entsprechenden Einstellungen. Der Anfang der Datei ist in Quellcode 5.4 zu sehen. Dabei werden der Titel des Problems (in diesem Fall “Perth”) und der Informations-Output (`linfo`) gesetzt. Der Laufzeitmodus `runmode` wird auf `2` gesetzt, d.h. SHEMAT soll neben der Funktion auch die Ableitungen berechnen. Dabei wird eine Ableitung berechnet (`# sample 1`) mit den Properties `const` und Benutzerdefinitionen `none`. Als aktive Simulationsergebnisse werden die Strömung (*head*) und die Temperatur gewählt. Als Outputdateien werden die Formate HDF5, VTK und Tec-Plot genutzt.

Quellcode 5.4: Beispiel für eine Modellbeschreibung in SHEMAT, Header

```
# title
Perth
# linfo
4*1
#runmode
2
# samples
1
# PROPS =const
# USER = none
# active head temp
# file output: hdf vtk tec
```

Als nächstes (Quellcode 5.5) wird das zugrundeliegende Diskretisierungsgitter definiert. Es wird ein Gitter mit 200x1x60 Punkten definiert, die jeweils einen Abstand von 50 Metern besitzen, zu sehen an der `delx, dely, delz` Definition.

Quellcode 5.5: Beispiel für eine Modellbeschreibung in SHEMAT, Grid

```
# grid
200 1 60
# delx
200*50.
# dely
50.
# delz
60*50.
```

Der nächste Abschnitt in Quellcode 5.6 zeigt die Einstellungen der in SHEMAT genutzten Löser. Dabei wird zuerst der nichtlineare Löser konfiguriert, hier also 10 Iterationen, ohne adaptive Relaxation (`0`). Nun werden die Löser für Fluss und Temperatur konfiguriert. Die Konfiguration der linearen Löser hat die Form `<Genauigkeit> <Kontrollparameter> <Iterationen>`. Hier also eine Genauigkeit von 10^{-10} und 500 Iterationen. Als Kontrollparameter wird hier die 67 gewählt. Dieser Wert setzt sich wie folgt zusammen:

$$ctrl = solvername + 16 * criteria + 256 * precondition$$

Mit `solvername`

0. BiCGStab
1. NAG
2. CG
3. PLU

und `criterium`

0. Relatives Abbruchkriterium
1. Absolutes Abbruchkrit. : $||res|| < depsilon$
2. Maximum Abbruchkrit. : $\max(abs(res)) < depsilon$
3. Abs. und Rel. Abbruchkrit. : $(||res|| < depsilon) \land (||res|| < 0.99d0 * ||res_0||\ )$
4. Wie '3.', aber mit automatischer Bestimmung von `depsilon`

sowie `precondition`

0. ILU
1. SSOR
2. Diagonal
3. None

Die hier gegebene 67 besagt also `solvername=3` (PLU), `criterium=4` (`depsilon` automatisch) und `precondition=0` (ILU). Nun wird die nichtlineare Iteration konfiguriert, mit einer Toleranz von 10^{-8} und einem Relaxationsfaktor von 0.5. Den Abschluss dieses Abschnitts bildet die Beschreibung der Randbedingungen mit `bcunit`, welches 3 Einträge enthält. Der erste Wert (3005) wird als linker Rand des Fluss-Feldes gewählt (`head bcd. simple=left, bcindex = 1`), der rechten Rand wird auf den Initialwert an dieser Stelle gesetzt. Für die Temperatur wird der Rand oben und unten entsprechend der Indizes in `bcunits` gesetzt, die linken und rechten Ränder behalten den Initialwert.

Quellcode 5.6: Beispiel für eine Modellbeschreibung in SHEMAT, Löser und Boundary-Conditions.

```
!==========>>>>> NONLINEAR SOLVER
# nlsolve
10 0

!==========>>>>> FLOW
# lsolvef (linear solver control)
1.d-10 67 500
# nliterf (nonlinear iteration control)
1.0d-8 0.5
# head bcd. simple=right, value=init
# head bcd. simple=left, bcindex = 1

!==========>>>>> TEMPERATURE
# lsolvet (linear solver control)
1.d-10 67 500
# nlitert (nonlinear iteration control)
1.0d-8 0.5
# temp bcd. simple=top, bcindex = 1
# temp bcn. simple=base, bcindex = 2
# temp bcd. simple=left, value=init
# temp bcd. simple=right, value=init

!==========>>>>>   define boundary properties
# bcunits. records=3
1   3005  head
1   19.0  temp
2   0.075  temp
```

Die Initialwerte der Felder für Head und Temperatur werden aus der HDF5 Datei `oed_open_final.h5` gelesen (vgl. Quellcode 5.7).

Quellcode 5.7: Beispiel für eine Modellbeschreibung in SHEMAT, Initialisierung.

```
!==========>>>>> INITIAL VALUES
# head init. HDF5=oed_open_final.h5
# temp init. HDF5=oed_open_final.h5
```

In Quellcode 5.8 werden die Parameter der Gesteinsschichten definiert. Dabei entspricht eine Zeile einer Gesteinsschicht und jeder Eintrag der Zeile entspricht einem Gesteinsparameter, deren Reihenfolge entsprechend folgender Liste definiert ist:

1. Porosität
2. Permeabilität in x-Richtung
3. Permeabilität in y-Richtung
4. Permeabilität in z-Richtung
5. Kompressibilität
6. Wärmeleitfähigkeit in x-Richtung
7. Wärmeleitfähigkeit in y-Richtung
8. Wärmeleitfähigkeit in z-Richtung
9. Wärmeproduktionsrate
10. Wärmekapazität
11. Diffusivität
12. Elektrische Konduktivität
13. Kopplungskoeffizient

Im Anschluss daran findet sich die geometrische Beschreibung des Modells (`# uindex`). Dies wird hier beschrieben durch eine lineare Liste mit Einträgen $k \cdot n$ mit k als Anzahl der Zellen, die den Gesteinsschichtindex n besitzen.

Quellcode 5.8: Beispiel für eine Modellbeschreibung in SHEMAT, Gesteinsschichtparameter.

```
!==========>>>>> UNIT DESCRIPTION
# units
0.15   1.d0  1.d0 5.e-14 1.e-10  1.d0  1.d0  2.58  0.0e-06  2300000.  10d0  0.d0  0.d0
0.001  1.d0  1.d0 1.e-13 1.e-10  1.d0  1.d0  2.58  0.0e-06  2300000.  10d0  0.d0  0.d0
0.205  1.d0  1.d0 7.e-13 1.e-10  1.d0  1.d0  3.54  0.0e-06  2300000.  10d0  0.d0  0.d0
0.042  1.d0  1.d0 2.e-17 1.e-10  1.d0  1.d0  1.71  0.0e-06  2300000.  10d0  0.d0  0.d0
0.23   1.d0  1.d0 1.e-12 1.e-10  1.d0  1.d0  2.90  0.0e-06  2300000.  10d0  0.d0  0.d0
0.114  1.d0  1.d0 8.e-15 1.e-10  1.d0  1.d0  3.73  0.0e-06  2300000.  10d0  0.d0  0.d0

# uindex
117*3 4*2 79*6 117*3 4*2 79*6 117*3 4*2 79*6 118*3 3*2 79*6 119*3 3*2 .....
```

Nun kommt der für uns interessanteste Teil, die Definition der Ableitungsobjekte in Quellcode 5.9. Die Einstellungen `enable unit` und `enable property` definieren die aktiven Gesteinsschichten und Gesteinsparameter entsprechend, 0 bedeutet nicht aktiv, > 0 aktiv. In unserem Beispiel bedeutet `0 1 0 0 0 0`, dass die Gesteinsschicht mit dem Index 2, in welcher der Gesteinsparameter 4 aktiv ist, untersucht werden soll. Alle anderen Werte sind nicht aktiv. Da SHEMAT eine eigene Inversenberechnung für die Bestimmung von Parametern mitbringt, wird im weiteren Verlauf diese Berechnung konfiguriert (`inverse`).

Quellcode 5.9: Beispiel für eine Modellbeschreibung in SHEMAT, Ableitunsdefinition.

```
!==========>>>>> INVERSE DESCRIPTION
# enable unit
0 1 0 0 0 0

# enable property
0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
# inverse
1
20 0.5 1.0
1 1
-1
```

In Quellcode 5.10 werden Messdaten für dieses Problem definiert. Die Datenpunkte für eine Zelle, die zeilenweise gespeichert werden, haben die Form

1. Messwert
2. Fehler
3. x-Zelle
4. y-Zelle
5. z-Zelle
6. Physikalischer Wert (1=Druck, 2=Temperatur, 3=Konzentration)
7. Bohrlochnummer

Hier wird ein Bohrloch definiert, welches an Position $c_x = 180$ eine Tiefe $\mathbf{c}_z$ von 39 Zellen besitzt. Jeder Messwert liefert einen Temperaturwert (Physischer Wert 2).

Quellcode 5.10: Beispiel für eine Modellbeschreibung in SHEMAT, Datenbeschreibung.

```
# data. records=39
2.28357E+1 0.5 180 1 59 2 1
2.58450E+1 0.5 180 1 58 2 1
2.94219E+1 0.5 180 1 57 2 1
3.32524E+1 0.5 180 1 56 2 1
....
```

5.3.1.1 Compilierung

Um SHEMAT zu compilieren, wird ein Makefile benutzt, welches im Rahmen dieser Arbeit vereinfacht und für die Benutzer von SHEMAT besser dokumentiert wurde. Dazu wurden die verschiedenen Komponenten in eigene Makefiles aufgeteilt. Diese sind

- Makefile.ext - Definition von Dateierweiterungen
- Makefile.dirs - Definition von Verzeichnissen
- Makefile.goals - Ziele und Switches für die Generierung von Code
- Makefile.defaults - Standardwerte definieren
- Makefile.libs - Bibliotheksbehandlung für z.B. HDF5 oder BLAS
- Makefile.arch - Architekturspezifische Einstellungen
- Makefile.osrc - Definition von Variablen für die Generierung von Code
- Makefile.help - Hilfe

und befinden sich in dem Unterverzeichnis `mach`.

Für die Generierung von Ableitungsobjekten werden die Makefiles im Unterverzeichnis `mkAD` genutzt. Für die Arbeit standen die zwei AD Tools TAF und Tapenade zur Verfügung, für welche in SHEMAT Compilierziele gegeben waren.

- forward (fw) - Generierung von Programmen für die Vorwärtsausführung der Simulation (klassische Simulation ohne inverse Problemstellungen)

- inverse (ad) - Generierung von Programmen für die deterministische Inversion
- simulate (sm) - Generierung von Programmen für die stochastische Inversion
- utility (ut) - Generierung diverser Hilfswerkzeuge für die Arbeit mit SHEMAT
- doc - Generiert die Hilfe

Um die Anbindung von SHEMAT an EFCOSS zu gewährleisten, wurden die folgenden Makefile-Ziele erarbeitet.

- library-param - Packt SHEMAT in eine von externen Programmen (z.B. EFCOSS) aufrufbare Bibliothek für Parameterstudien
- library-oed - Packt SHEMAT in eine von externen Programmen (z.B. EFCOSS) aufrufbare Bibliothek für OED

Für den Vorwärts- und Rückwärtsmodus des automatischen Differenzierens finden sich verschiedene Makefiles. Die Generierung von Ableitungscodes für den sehr komplexen und umfangreichen Quellcode von SHEMAT ist eine aufwändige und schwierige Angelegenheit. Dies äußert sich besonders im Einsatz von OpenMP für die Parallelisierung der Simulation. Da die AD-Tools keine komplexen OpenMP Konstrukte unterstützen (zumindest zur Zeit der Erstellung von SHEMAT), muss hier ein Trick angewendet werden. Alle OpenMP-Pragmas werden mit Hilfe von Shell-Scripten durch Dummy-Routinenaufrufe ersetzt, welche die OpenMP-Konstrukte als String-Eingabe erhalten. Dann wird sichergestellt, dass die AD-Tools diese Routinen nicht wegoptimieren. Nach dem Transformationslauf werden die Dummy-Aufrufe wieder durch OpenMP Pragmas ersetzt. Diese Vorgehensweise ist jedoch nur für den Vorwärtsmodus voll einsetzbar. Im Rückwärtsmodus kann damit nur der Vorwärts-Sweep beschleunigt werden.

Eine weitere Schwierigkeit zeigt sich in der Benutzung der unterschiedlichen Module. Sei k die Anzahl der User-Module und l die Anzahl an Property-Modulen, dann ist $k \times l$ die Anzahl an Ausführungen des Ableitungstools, die für alle Module benötigt wird. Diesen Aufwand kann man vermeiden, indem standardisierte Module zur Generierung genutzt werden. Diese sind `wells3d` bei den User-Modulen und `bas` bei den Property-Modulen. Hierdurch wird die Laufzeit der Transformation drastisch gesenkt, da mit Hilfe von speziellen Shell-Scripten die Codes entsprechend angepasst und an die entsprechenden Stellen des Quellcode-Verzeichnisses kopiert werden.

Als Make-Befehl wird

```
make tap_tlm
```

für den Vorwärtsmodus mit dem Tool Tapenade genutzt. Für den Rückwärtsmodus wird

```
make tap_adm
```

genutzt.

Diese Befehle erzeugen die Ableitungscodes in einem temporären Verzeichnis und kopieren die Dateien anschließend in die entsprechenden Arbeitsverzeichnisse. Für den Vorwärtsmodus mit dem AD-Tool Tapenade sind dies:

- `g_tap` - Allgemeines Ableitungsverzeichnis
- `props/*/g_tap` - Properties Ableitungsverzeichnisse
- `user/*/g_tap` - User Ableitungsverzeichnisse

In die Ableitungsgenerierung ist im Rahmen des ersten MeProRisk Projektes viel Arbeit geflossen, weshalb es unter den Verzeichnissen

- `inverse/g_tap`
- `blas/g_tap`

spezielle Ableitungsroutinen gibt, die per Hand abgeleitet wurden. Im Speziellen bieten die Ableitungsroutinen für BLAS einen großen Vorteil in der Ausführungszeit gegenüber einem Black-Box Ansatz. Des Weiteren wurden die linearen Löser per Hand abgeleitet, da ein Durchdifferenzieren von diesen zu Problemen führen kann.

Das gleiche Schema gilt für den Reverse-Modus mit `ad_tap`. Wenn das AD-Tool TAF genutzt wird, muss lediglich `tap` durch `taf` ersetzt werden.

5.3.2 Parameterstudien

Für die Parameterstudien wurde ein Interface geschrieben, welches die benötigten Funktionen, also die Vorwärtsberechnung der Simulation, sowie die in den Optimierern benötigten Funktionen zur Berechnung der Lösung eines inversen Problems mit AD bereitstellt. Es wurden Interfaces für die Berechnung der von EFCOSS benötigten Werte direkt in Fortran implementiert. Dies ist nötig, da SHEMAT ein sehr komplexes Initialisierungsverhalten aufweist und dies in Python abzubilden nicht praktikabel ist. Des Weiteren sind die genutzten Datenfelder unter Umständen sehr groß, so dass die Zeit für die Konversion von Python in Fortran Arrays sehr lange dauern würde.

Diese Interfaces wurden im SHEMAT Verzeichnis in der Datei `shem_parameter.f90` implementiert. Für die Berechnung des Funktionswertes wird die folgende Funktionssignatur benutzt.

```
subroutine shemat(pp0,pn0,pparm0,pddata,infile,fdim,omp_inner,omp_outer)
```

Dabei geben die Werte `pp0` die Anzahl der Input-Parameter im Array `pparm0` und `pn0` die Anzahl der Outputparameter in `pddata` an. Der optionale Input-Parameter `infile` mit Größe `fdim` gibt die zu lesende Modell-Datei an. Wird es nicht genutzt, so werden die Modelldateinamen aus einer Datei namens `shemade.job` ausgelesen. Zusätzlich wurde eine Möglichkeit geschaffen, die in SHEMAT genutzten inneren und äußeren Parallelisierungsstufen von EFCOSS aus zu setzen, welche mit optionalen Parametern `omp_inner` und `omp_outer` gesetzt werden.

Die Berechnung der Ableitungen wird in der gleichen Subroutine, jedoch mit unterschiedlichen Einsprungpunkten, definiert. Diese sind

```
entry g_shemat_proc(g_p_,pp0,pn0,pparm0,g_pparm0,pddata,g_pddata,
                    infile,fdim,omp_inner,omp_outer)
```

für die Berechnung der vollen Jacobi-Matrix mit dem Vorwärtsmodus des automatischen Differenzierens. Die Übergabewerte sind wie oben. Zusätzlich finden sich die neuen Parameter `g_p_` für die Anzahl der Ableitungen, `g_pparm0` und `g_pddata` für die Ableitungsobjekte. Die Variable `omp_outer` setzt die äußere Parallelisierungsstufe von SHEMAT, also in diesem Fall die Anzahl an Threads, die gleichzeitig die Spalten der Jacobi-Matrix berechnen.

In der Berechnung von Parameterstudien in SHEMAT wurden Vorwärts-AD sowie Rückwärts-AD implementiert, um die Ableitungsinformationen $J\mathbf{v}$ und $J^T\mathbf{v}$, entsprechend der Überlegungen aus Abschnitt 2.2.1 und Abschnitt 4.4.4.3, zu bestimmen. Also definieren wir die Einsprungpunkte

```
entry g_shemat_matvec(pp0,pn0,pparm0,xvec,pddata,yvec,infilename,fdim,
                      omp_inner,omp_outer)
```

für die Berechnung von $J\mathbf{v} = \mathbf{w}$ und

```
entry g_shemat_mattvec(pp0,pn0,pparm0,xvec_ad,pddata,yvec_ad,infilename,
                       fdim,omp_inner,omp_outer)
```

für die Berechnung von $J^T\mathbf{v} = \mathbf{w}$ entsprechend. Mit `xvec` bzw `xvec_ad` als Eingabevektor $\mathbf{v}$ und `yvec` bzw `yvec_ad` als Ausgabevektor $\mathbf{w}$.

Je nach Einsprungpunkt werden die folgenden boolschen Parameter gesetzt

- `lshem` – Berechnung des Vorwärtsmodells
- `g_lshem` – AD-Berechnung der Jacobi-Matrix
- `g_matvec` – AD-Berechnung für Matrix-Vektor Operation
- `ad_matvec` – Rückwärts-AD-Berechnung für Matrix-Transponiert-Vektor Operation
- `g_pmax_` – Maximale Dimension der Ableitungsobjekte

Die zwei Beispiele Quellcode 5.11 und 5.12 zeigen die Einsprungpunkte und die gesetzten Werte zur Berechnung der kompletten Jacobi-Matrix J und dem Matrix-Vektorprodukt $J^T\mathbf{v}$.

Quellcode 5.11: SHEMAT Einsprungpunkt für die Berechnung der vollen Jacobi-Matrix.

```
ENTRY g_shemat_proc(g_p_,pp0,pn0,pparm0,g_pparm0, &
    pddata,g_pddata,infilename,fdim,omp_inner,omp_outer)
lshem = .TRUE.
g_lshem = .TRUE.
g_matvec=.FALSE.
ad_matvec= .FALSE.
g_pmax_ = g_p_
GO TO 200
```

Quellcode 5.12: SHEMAT Einsprungpunkt für die Berechnung des Jacobi-Matrix-Transponiert Vektor mit AD Reverse Mode $J^T\mathbf{v} = \mathbf{w}$.

```
ENTRY g_shemat_mattvec(pp0,pn0,pparm0,xvec_ad,pddata,yvec_ad,infilename,fdim,omp_inner,omp_outer)
lshem = .TRUE.
g_lshem = .FALSE.
g_matvec = .FALSE.
ad_matvec = .TRUE.
g_pmax_=1
GO TO 200
```

Das Label `200` markiert den eigentliche Beginn des Programms. Dieser besteht aus den folgenden Bestandteilen:

1. Einlesen der Modellbeschreibung aus der Datei
2. Setzen der Optimierungs-Parameter nach den Vorgaben aus EFCOSS
3. Initialisierung des Vorwärtsmodells `init_forward()`

4a. Ausführen des Vorwärtsmodells `forward_iter()`

4b. Ausführen des Ableitungscodes für die volle Jacobian-Matrix:

   (a) Setzen des Ableitungsseeds
   (b) Initialisierung des Ableitungscodes `prepare_jacobian`
   (c) Ausführen des Ableitungscodes `jacobi_compute`

4c. Ausführen des Matrix-Vektor Codes

   (a) Initialisierung der Realisation `single_init`

   (b) Initialisierung des Ableitungscodes `prepare_jacobian`
   (c) Ableitungsberechnung `jac(t)_mv`

5. Kopieren der Ergebnisse in die von EFCOSS verwalteten Speicherbereiche
6. Speicher freigeben

In Bestandteil 2, dem Setzen der Optimierungs-Parameter, wurde auf eine Besonderheit für die Permeabilität geachtet. Diese hat einen Wertebereich für die bei uns untersuchten Probleme von 10^{-18} bis 10^{-10}, was für einen Optimierer schwierig abzubilden ist. Deshalb wurde die Eingabe um den Faktor 10^{-13} skaliert. Das heißt, eine Eingabe von 15 bedeutet in dem Modell einen Parameter von $1.5 \cdot 10^{-12}$.

Wir definieren nun eine Anwenderklasse mit den Informationen über die Simulation von SHEMAT. Die Klasse `ShematDatafit` aus Quellcode 5.13, abgeleitet von `ProblemDefinition` definiert vier Methoden. Bei der Initialisierung wird die Initialisierung der Klasse `ProblemDefinition` mit einem Dateinamen, welcher die SHEMAT-Problembeschreibung enthält, genutzt. Weiterhin finden sich Platzhalter für die Positionen der Messpunkte `positions`, deren Messwerte `values` und etwaige Gewichte `weights`.

Mit der Routine `initEFCOSS` werden die Eingaben für den Aufruf der SHEMAT-Simulation definiert. Dabei wird als Eingabe ein Startwert des Optimierers `x_st`, sowie optional Werte für die innere und äußere Parallelisierung (`inner` und `outer`) benötigt. Nun können die Übergabeparameter für die Simulation als EFCOSS-Variablen angelegt und befüllt werden. Weiterhin wird die Aufrufsequenz der Simulation, sowie die zu optimierenden Parameter, in unserem Fall nur `pparm0`, gesetzt. Die `initSimulationInterface` Methode setzt die Simulation und eröffnet die wechselseitige Verarbeitung zwischen EFCOSS und den SHEMAT-Interfaces aus Quellcode 5.14.

Als Zielfunktion wird die `DataFit1d` Klasse verwendet. Vorher werden jedoch noch wichtige Informationen aus der Input-Definition von SHEMAT mittels `readGrid` und `readInput` eingelesen. Erstere Funktion liest Informationen über den Aufbau des zugrundeliegenden Diskretisierungsgitters ein, letztere die in der Eingabedatei beschriebenen Messwerte aus dem Abschnitt `#data`. Als Generator für die Interfaces wird die Standardroutine aus `EFCOSS.ProblemDefinition` genutzt.

Die Routine `runOptimizer` ist ebenfalls zu sehen. Diese bietet ein einfaches Interface zur Benutzung der Scipy-Optimierer. Ausgehend von einem Startwert `x0` wird die Optimierung mit der übergebenen Optimierungsmethode `method` gestartet. Weiterhin können die Grenzen entsprechend übergeben werden. Die Übergabeparameter `bound` dienen als Beschränkungen der Eingaben für den Optimierer und `tol` für die Toleranz des Ergebnisses.

Quellcode 5.13: Problemdefinition für ShematDatafit.

```
class ShematDatafit(ProblemDefinition):
  def __init__(self,fileName)
      ProblemDefinition(self,"EFCOSS+"fileName)
      self.fileName=fileName
      self.positions=[]
      self.values=[]
      self.weights=[]

  def initEFCOSS(self,x_st,inner=1,outer=1)
      self.numvars=len(x_st)
      self.pP0=self.efcoss.newInputVariable("pP0",self.numvars)
      self.pN0=self.efcoss.newInputVariable("pN0",len(self.values))
      self.pparm0=self.efcoss.newInputVariable("pparm0",x_st)
```

```
    self.pddata=self.efcoss.newOutputVariable("pddata",self.pN0)
    self.inner=self.efcoss.newInputVariable("inner",self.inner)
    self.outer=self.efcoss.newInputVariable("outer",self.outer)
    self.efcoss._set_SimulationCallingSequence([self.pP0,self.pN0,self.pparm0,self.pddata,self.inner,
        self.outer])
    self.efcoss._set_OptVars([self.pparm0])

  def initSimulationInterface(self):
    self.simulation = shematDatafitSimulation.shematDatafitSimulation(self.fileName,[self.fileName])
    self.efcoss.setSimulationServer(self.simulation)
    self.simulation.setEFCOSS(self.efcoss)

  def initObjectiveFunction(self):
    self.readGrid()
    self.readInput()
    self.dataFit = self.setObjectiveFunction("DataFit","DataFit1d")
    self.dataFit.addData1d(self.pddata,array(range(self.pN0))+1,self.values,ones(self.pN0))
    self.dataFit._set_model_parameters([self.pparm0])

  def runOptimizer(self,x0,method='L-BFGS-B',bound=[(1E-5,1E3)],tol=1E-8):
    optimizer=opt_scipy(self.efcoss,tol=tol,bounds=bound,method=method,useInternalDD=False,options={'
        disp':True})
    return optimizer.minimize(x0)
```

Quellcode 5.14: Python-Klasse für die Simulationsinterfaces des Datenfitproblems für SHEMAT.

```
class shematDatafitSimulation(Simulation):
  def __init__(self,name,fileName):
    Simulation(self,name)
    self.name=name
    self.fileName=fileName

  def setEFCOSS(self.efcoss):
    self.efcoss=efcoss

  def Function(self,x):
    result=shemat.func(x.i,x.d,x.i[1],self.filename)
    return result

  def Jacobian(self,x,seed):
    (result,jac)=shemat.jacobian(x.i,x.d,seed,x.i[1],self.filename)
    return (result,jac)

  def JacobianVector(self,x,vec):
    jacvec,res=shemat.jacobianvector(x.i,x.d,vec,self.filename)
    return jacvec,res

  def JacobianTVector(self, x, vec):
    jactvec,res=shemat.jacobiantvector(x.i,x.d,vec,self.filename)
    return jactvec,res
```

Die Interfaces werden dann mittels

```
make shemat-param
```

im Verzeichnis `src/EFCOSS/Applications/shemat` compiliert. Dazu muss die entsprechende SHEMAT Bibliothek vorhanden sein. Wie diese erzeugt wird, wird im Abschnitt 5.3.5 zu den einzelnen

Modellen beschrieben, da sich diese in den Property- und User-Modulen unterscheiden.

5.3.3 Space-Mapping

Für den Datafit kann weiterhin die Space-Mapping Methodik genutzt werden, um die benötigte Laufzeit drastisch zu verringern. Hierfür wird das unter Abschnitt 5.3.2 erarbeitete Interface weiter verwendet, jedoch um die Möglichkeit erweitert, andere Simulationsbeschreibungen zu laden. Dies ist wichtig, damit die verschiedenen Modelle für die grobe und feine Auflösung genutzt werden können. Hierzu müssen in EFCOSS zwei verschiedene Simulationsinstanzen (fein und grob) und zwei verschiedene Zielfunktionen definiert werden.

Quellcode 5.15 zeigt die Problemdefinition der Klasse für den Datenfit von SHEMAT mit Space-Mapping. Für die Simulationen wird im Ausführungsverzeichnis je ein Verzeichnis für das feine Modell (`fine/`) und das grobe Modell (`coarse/`) benötigt. In diesen Dateien finden sich jeweils die Dateien für die Berechnung des Modells. Hierbei wird angenommen, dass beide den gleichen Dateinamen besitzen, welcher im Konstruktor gesetzt wird. Weiter wird im Konstruktor auch die Größe der Blöcke gesetzt. Als Simulationsklasse wird sowohl für das feine als auch für das grobe Modell die `shematDatafitSimulation` Klasse genutzt. Das grobe Modell erhält in EFCOSS die ID 0, das Feine die ID 1. Die benutzten Zielfunktionen sind entsprechend den Überlegungen in Abschnitt 4.5.1 die Klasse `Datafit.Datafit1d` für die ID 0 und die Klasse `VectorObjective.Identity` für die ID 1. Die Datafit-Zielfunktion wird dabei für die Initialisierung des Space-Mapping-Ansatzes beim Berechnen des Startwertes $\mathbf{z}^*$ genutzt. Die Identität wird für die Distanzberechnung zwischen feinem und grobem Modell in jeder Iteration benötigt.

Nun kann mittels der Methode `run_spacemap_asm()` die Aggressive-Space-Mapping Operation durchgeführt werden. Dazu werden ein Name, sowie boolsche Variablen für die Jacobi-Berechnung mittels AD je Funktion eingegeben. Die Funktion gibt das Ergebnis dann in der Konsole aus. Wir verwenden an Stelle der im Space-Mapping vorgegebenen Optimierer den SLSQP, weil dieser für das Datenfit-Problem die besten Ergebnisse liefert, vergleiche hierzu Abschnitt 5.3.5.1.

Quellcode 5.15: Python-Klasse für das Lösen von Datenfitproblemen mit Space-Mapping für SHEMAT.

```
class ShematDatafitSpacemap(ShematDatafit):
    def __init__(self, _job_id, fileNameCoarse,fileNameFine):
        ProblemDefinition(self,"EFCOSSShematSpace-Mapping",2,2,matFree=True)
        self.job_id = _job_id
        self.fileName = fileNameCoarse
        self.fileNameFine=fileNameFine
        self.numvars = 1
        self.positions = []
        self.values = []
        self.weights = []

    def initSimulationInterface(self):
        self.simulation_fine = shematDatafitSimulation.shematDatafitSimulation(self.fileNameFine, [self.
            fileNameFine])
        self.simulation_coarse = shematDatafitSimulation.shematDatafitSimulation(self.fileName, [self.
            fileName])
        self.efcoss.setSimulationServer(self.simulation_coarse, 0)
        self.efcoss.setSimulationServer(self.simulation_fine, 1)
        self.simulation_fine.setEFCOSS(self.efcoss)
        self.simulation_coarse.setEFCOSS(self.efcoss)
```

```
    def initObjectiveFunction(self):
        ShematDatafit.initObjectiveFunction(self)
        self.scalarObjective = self.setObjectiveFunction("VectorObjective", "Identity", 1)

    def run_spacemap_asm(self, name=’’, jac=[False, False]):
        asm = SpaceMapping.ASM(self.efcoss, self.x_st, 1e-4, 100, 1e-5, 1e3, name, jac,optimizer=
             ScipyOptimizer.ScipyOptimizer(tol=1e-5, method=’SLSQP’, options={’disp’: True}))
        [x, p, k, z_st, ffun] = asm.run()
        print "ASM RESULTS:"
        print "================="
        print "Result=", p[k]
        print "Used steps=", k + 1
        print "z_st=", z_st
        print "x=", x[k]
        print "ffun=", ffun[k]
        print " "
        print " "
```

5.3.4 Optimal Experimental Design

Um die Ergebnisse des Optimal Experimental Designs zu berechnen, wird die komplette Jacobi-Matrix benötigt, da die Fischer Matrix als $F = J^T J$ benötigt wird. Für die Berechnung bietet sich der Vorwärtsmodus des automatischen Differenzierens an, da für unsere Anwendungen die Anzahl der Eingabeparameter beschränkt ist. Des Weiteren bietet die OpenMP parallelisierte Version von SHEMAT nur die Kombination mit dem Vorwärtsmodus des automatischen Differenzierens an.

Als anzubindende Routine an EFCOSS dient uns der Code aus `shem_oed.f90`. Die zur Verfügung gestellten Routinennamen sind

- `shemat` - Ausführen der Simulation
- `g_shemat_proc` - Erste Ableitung berechnen, volle Jacobi-Matrix
- `g_shemat_proc_get_fullgrid` - Erste Ableitung berechnen und zusätzlich alle Arrays an EFCOSS übergeben
- `g_shemat_proc_set_fullgrid` - Setzen der vorher übergebenen Arrays und Rückgabe der ersten Ableitung ohne neue Berechnung.

Die Routinen zur Ableitungsberechnung sind hierbei wieder als `ENTRY` definiert, also anderweitige Einsprungpunkte in die gleiche Routine `shemat`. Die Übergabeparameter der Simulation sind

```
subroutine shemat(pp0,pk0,pn0,ptmax,pparm0,pdhead,pdtemp,pdconc, &
         pxcoord,zmin,zmax,omp_inner,omp_outer)
```

mit `pp0`, `pk0` und `pn0` wie in Abschnitt 5.3.2 die Größen der in der Parameterliste auftretenden Felder. Der Parameter `ptmax` gibt den Zeitpunkt der Messpunkte an. Dies ist bei Nicht-Steady-State Problemen nötig. Wir betrachten hier jedoch zuerst nur Steady-State Modelle, somit ist dieser Parameter für uns unwichtig. Weitere Eingaben sind die Variable `pxcoord`, die eine Liste von c_x, c_y Wertepaaren für die Positionen der Bohrlöcher, sowie deren Tiefe in `zmin` bzw. `zmax` vorgibt. Die `omp_inner` bzw. `omp_outer` Parameter dienen dem Einstellen der inneren und äußeren Parallelität von SHEMAT. Als Ausgabearrays werden die Werte `pdhead`, `pdtemp` und `pdconc` für die Head, Temperatur und Konzentration benutzt. Um Berechnungszeit zu sparen, wurden die beiden Entry-Punkte `g_shemat_proc_get_fullgrid` und `g_shemat_proc_set_fullgrid` eingefügt. Ersteres wird verwendet, um die Berechnung der Jacobi-Matrix in jedem Gitterpunkt zu speichern. Letzteres sorgt dafür, dass die vorher gespeicherte Jacobi-Matrix wieder an SHEMAT zurückgegeben wird. Dies ist jedoch nur möglich, da die Ableitungen im vorliegenden OED-Fall sich nicht ändern, wenn die Position des Bohrlochs geändert wird.

Für die Anbindung an EFCOSS sorgt die Klasse `ShematOED`, vgl. Quellcode 5.16, welche ebenfalls in `src/EFCOSS/Applications/shemat` zu finden ist. Die Klasse ist abgeleitet von der Klasse `EFCOSS.Problemdefinition` und erwartet in der Initialisierung die folgenden Parameter: das verwendete Optimalitätskriterium `opt_krit` dient sowohl zum Setzen der richtigen Zielfunktion, sowie für einen Teil der Ergebnis-Ausgabedateinamen. Der Parameter `_job_id` als Übergabe des Jobschedulers im Cluster, ist eine eindeutige ID für die Ausführung und ebenfalls Teil der Ausgabedateinamen. Zusammen mit dem optionalen Parameter `_id` können unabhängig von anderen Ausführungsinstanzen OED Probleme parallel gelöst werden. Diese Aufteilung ist nötig, da sonst Ergebnisse für verschiedene Eingabeparameter in die gleiche Datei geschrieben werden würden. Die Ergebnisse werden weiterhin in die Verzeichnisse `results/<_opt_krit>` sowie `results/<_opt_krit>-proc` geschrieben. Ersteres ist für die OED-Ergebnisse, letzteres für die Zwischenergebnisse und Ausgaben der SHEMAT-Simulation bestimmt. Die Konsolen-Ausgaben werden über den optionalen Parameter `_noout=True` über die Python Funktion des Betriebssystems `os.dup2` in eine Datei umgeleitet.

Die Parameterliste beinhaltet außerdem die Beschränkung der Optimierungsprobleme. Diese werden durch das SHEMAT-Eingabefile `inputfile` definiert. Die verwendeten Parameter müssen die folgenden Bedingungen erfüllen,

- `_minx`, `_maxx` und `_spanx` für X-Richtung,
- `_miny`, `_maxy` und `_spany` für Y-Richtung, sowie
- `_minz`, `_maxz` und `_spanz` für Z-Richtung.

Der Wert `_minx` ist der minimale Zellenindex, `_maxx` der maximale und `_spanx` gibt die Größe einer Zelle an. Dies muss nicht zwangsläufig gleich den Modellparametern aus dem Eingabefile sein. Die Werte der Z-Richtung stehen hier für die betrachtete Tiefe des Explorationsbohrlochs.

Um dem Anwender einer OED-Optimierung weitere Möglichkeiten zu bieten, finden sich in der Initialisierung noch die Parameter zum Setzen weiterer Bohrlochpositionen (`_original_positions`), sowie deren minimale und maximale Tiefe (`original_minz` bzw. `original_maxz`). Dies dient dazu, die im Explorationsgebiet gegebenen Bohrlöcher in das OED-Problem zu übertragen.

Weiterhin finden sich die Variablen `_inner` und `_outer` für innere und äußere Parallelisierung in SHEMAT. Der folgende Parameter `saveJacobian` bietet die Möglichkeit, die berechnete Ableitung des SHEMAT-Modells in EFCOSS zu transferieren und dann wieder zu verwenden. Dazu finden sich die weiteren Parameter `gridX`, `gridY` und `gridZ`, die die Größe des zu speichernden Ableitungsobjekts angeben. Über den Parameter `inputfile` kann die zu verwendende SHEMAT-Problembeschreibung definiert werden.

Quellcode 5.16: Initialisierungsroutine für OED Probleme in SHEMAT.

```
class ShematOED(EFCOSS.ProblemDefinition):
    def __init__(self, _opt_krit, _job_id, _minx, _maxx, _spanx, _miny, _maxy, _spany, _minz, _maxz,
         _spanz,_original_positions,original_minz, original_maxz, _inner=1, _outer=1, _id="", _noout=
         True,saveJacobian=False, gridX=0, gridY=0, gridZ=0, inputfile="oed_inv"):
```

Neben der Minimierung des Optimierungsproblems war weiter ein Überblick über die zugrundeliegende Zielfunktion wünschenswert. Deshalb bietet die Klasse neben einer Optimierungsfunktion `run_solve_shemat` noch die Funktionen `run_func_shemat`, sowie `run_func_shemat2d`. Erstere berechnet für jede X-Position in einem 2D-Beispiel (ein X-Z-Schnitt durch das Reservoir, zum Beispiel für Perth, vgl. Kapitel 5.3.5.1) einen Zielfunktionswert und speichert diesen. Die zweite Funktion erzeugt eine 2D-Karte der Zielfunktionale (in der X-Y-Ebene) für das volle 3D-Beispiel (zum Beispiel für Toskana, vgl Kapitel 5.3.5.2).

In der `setup_efcoss` Methode, aus Quellcode 5.17, werden die entsprechenden EFCOSS-Input- und Output-Variablen für die Simulationsroutine `shemat` gesetzt. Dabei nimmt `pp0` die Anzahl der Modellparameter ein. Das entspricht genau der Länge des Eingabevektors `input_data`, der in `pparm0` Anwendung findet. Die Anzahl der Bohrlöcher wird in `pk0` und in `pn0` die Anzahl der Resultate pro Bohrloch angegeben. Die Ergebnisse für Head, Temperatur und Konzentration haben somit die Ausgabegröße von `pn0` x `pk0`. Das Bohrloch mit dem Index 0 wird dabei das neu hinzuzufügende und für OED vorgesehenen Bohrloch. Die weiteren potentiellen Bohrlöcher kommen aus `_original_positions`. Für jedes Bohrloch wird außerdem die Tiefe und die Anfangshöhe der ersten Messung angegeben. Zu beachten ist hier, dass für alle Bohrlöcher die gleiche Anzahl an Messwerten angenommen wird. Weiterhin werden die innere und äußere Parallelisierung, sowie die Eingabemodelldatei für SHEMAT gesetzt. Zusätzlich initialisiert die Methode die Simulationsklasse `shematSimulation` und setzt die Zielfunktion auf `OED` mit dem im Konstruktor übergebenen Optimalitätskriterium `opt_krit`. Die `OED` Klasse benötigt zur Initialisierung noch die Designparameter `design_parameter` und Modellparameter `model_parameter`. Der Modellparameter ist dabei der Optimialitätsparameter `parm0`, als Designparameter werden die Koordinaten der Bohrlöcher gewählt. Als Simulationsergebnis wird die Temperatur gewählt.

Quellcode 5.17: Setup Routine für die Verwendung von SHEMAT mit EFCOSS.

```python
def setup_efcoss(self, input_data, i, mode=0):
    positions = []
    depth = []
    height = []
    self.mode=mode

    # Extend a dummy value, that is updated every time
    self.positions.extend([0.0, 0.0])
    self.depth.extend([float(self.minz)])
    self.height.extend([self.maxz])
    # Extend original positions
    if (self.numdes == 1):
        self.positions.extend(self.original_positions)
        self.depth.extend(self.original_minz)
        self.height.extend(self.original_maxz)

    # set the number of active model parameters to length of input_data
    p = len(input_data)  # constant value ( equals number of active model parameters )

    # Define Input and Output variables of the simulation
    P0 = self.efcoss.newInputVariable("pp0", p)  # number of model parameters (fixed)
    K0 = self.efcoss.newInputVariable("pk0", len(positions) / 2)  # number of bore holes
    N0 = self.efcoss.newInputVariable("pn0", (int)(self.maxz - self.minz) / self.spanz)  # number of
          results
    Tmax = self.efcoss.newInputVariable("ptmax", 45.00) # dummy value for time
```

```
        self.parm0 = self.efcoss.newInputVariable("pparm0", input_data)  # model parameters
        dhead = self.efcoss.newOutputVariable("pdhead", [N0 * K0])  # head
        dtemp = self.efcoss.newOutputVariable("pdtemp", [N0 * K0])  # temperature
        dconc = self.efcoss.newOutputVariable("pdconc", [N0 * K0])  # concentration
        self.xcoord = self.efcoss.newInputVariable("pxcoord", positions)  # position of borehole
        self.zmin = self.efcoss.newInputVariable("zmin", depth)
        zmax = self.efcoss.newInputVariable("zmax", height)
        inner_threads = self.efcoss.newInputVariable("omp_inner", int(self.inner))
        outer_threads = self.efcoss.newInputVariable("omp_outer", int(self.outer))

        self.efcoss._set_SimulationCallingSequence(
            [P0, K0, N0, Tmax, self.parm0, dhead, dtemp, dconc, self.xcoord, self.zmin, zmax,
                inner_threads, outer_threads])

        self.efcoss._set_OptVars([self.parm0])

    def initSimulationInterface(self):
        self.simulation = shematSimulation.shematSimulation("shemat", self.saveJacobian, self.gridX,
                                    self.gridY, self.gridZ, self.spanx, self.spany, self.spanz,
                                    self.inputfile)

        self.efcoss.setSimulationServer(self.simulation)
        self.simulation.setOptimizer(self.efcoss)

    def initObjectiveFunction(self):
        if (self.mode == 0):
            design_parameters = [self.xcoord]
        else:
            design_parameters = [self.xcoord, self.zmin]
            self.mode = 1

        model_parameters = [self.parm0]

        oed = self.efcoss.setObjectiveFunction("OED", self.opt_krit)
        oed._set_design_parameters(design_parameters)
        oed._set_model_parameters(model_parameters)

        weight = 1.0

        for i in range(0, self.numdes):
            if (mode == 0):
                oed.addDesign(design_parameters, self.positions, self.weight)
            else:
                oed.addDesign(design_parameters, self.positions + self.depth, self.weight)

        oed._set_simulation_result([dtemp])
        oed.setVariance1d(dtemp,[1.0])
```

Diese Simulationsklasse `shematSimulation`, zu sehen in Quellcode 5.18, ist von der Klasse `Simulation` abgeleitet, bietet aber weitergehende Möglichkeiten. Die erste Anpassung ist die Unterstützung von übergebenen Dateinamen. EFCOSS speichert in seiner Datengrundlage nur Numpy-Arrays oder Integerwerte. Strings werden hier nicht berücksichtigt, müssen also anderweitig der Simulation übergeben werden. Dies geschieht beim Initialisieren der Simulation, indem zusätzlich der Dateiname übergeben wird. Deshalb benötigt auch die Python-Klasse den Dateinamen `filename`. Mit dem zusätzlichen Parameter `saveJacobi` wird der einmal mit der Simulation berechnete Ableitungswert abgespeichert und beim nächsten Aufruf an der entsprechenden Stelle in SHEMAT eingesetzt. Dies kann die Ausführungszeit massiv senken, wenn sich mit der Eingabe des Explorationsbohrloches keine Änderungen bezüglich der (intern in SHEMAT berechneten) Ableitungen ergeben. Zu diesem Zweck benötigen wir noch die Werte `gridX`, `gridY` und `gridZ` für die Größe des Gitters und `delX`, `delY` und `delZ` für die Größe der einzelnen Zellen.

Für die Funktionsauswertung mittels der `Function` Methode ändert sich nichts gegenüber dem Standardinterfaceaufruf, außer dass der Dateiname der Inputdefinition angegeben wird. Für die Berechnung der Ableitung mit der `Jacobian` Methode werden zwei Fälle unterschieden:

1. Die Jacobi-Matrix wird gespeichert (`saveJacobi=True`)
    - Für die erste Ausführung muss die Jacobi-Matrix erzeugt werden (`getfullgridjacobian`)
    - In jeder darauf folgenden Iteration wird der gespeicherte Wert verwendet (`setfullgridjacobian`)
2. Die Jacobi-Matrix wird nicht gespeichert
    - Berechne die Jacobi-Matrix in jeder Iteration neu

Quellcode 5.18: Genutzte Simulationsklasse für die optimale Versuchsplanung mit SHEMAT.

```
class shematOEDSimulation(Simulation):
    def __init__(self, name, saveJacobian=False, gridX=0, gridY=0, gridZ=0, delX=0, delY=0, delZ=0,
         filename=""):
        ...
        self.run=0

    def Function(self, x):
        result = shemat.func(x.i, x.d,x.i[0],x.i[1],x.i[2], self.filename)
        return result

    def Jacobian(self, x, seed):
        if (self.saveJacobian):
            if (self.run == 0):
                self.fullGridJacobi = zeros((3, self.gridX, self.gridY, self.gridZ, int(seed[0])))
                (result, jac, self.res, self.fullGridJacobi) = shemat.getfullgridjacobian(x.i, x.d, seed
                     , x.i[0], x.i[2], self.gridX, self.gridY, self.gridZ,self.filename)
                self.run += 1
            else:
                (result,jac)=shemat.setfullgridjacobian(x.i,x.d,seed,x.i[0],x.i[2],self.gridX,self.gridY
                     ,self.gridZ,self.res,self.fullGridJacobi,x.i[1])
                self.run += 1
        else:
            (result, jac) = shemat.jacobian(x.i, x.d, seed, x.i[0], x.i[2], self.filename)
        return (result, jac)
```

Das zugrundeliegenden Interface aus `shemat_oed_if.f90` wurde mit der Standardroutine erzeugt, aber um die oben beschriebenen Änderungen ergänzt. So bietet es die Einsprungpunkte `getfullgridjacobian` sowie `setfullgridjacobian` für die eben beschriebenen Anwendungen, wie in Quellcode 5.19 zu sehen.

Quellcode 5.19: Einsprungpunkte in die Shemat-Routine für das Speichern und Einsetzen der Jacobi-Matrix, ohne den AD Codes ausführen zu müssen.

```
entry getFullGridJacobian(iInputVector,dInputVector,seed_matrix,&
            dOutputVector,dJacobianOut,iP0,iK0,iN0,gridX,gridY,gridZ,&
```

```
            resultOut,JacobianGrid,infile,fdim)
entry setFullGridJacobian(iInputVector,dInputVector,seed_matrix,&
            dOutputVector,dJacobianOut,iP0,iK0,iN0,gridX,gridY,gridZ,&
            resultIn,JacobianGridIn,infile,fdim)
```

Diese Vorgehensweise ermöglicht es uns, große OED-Parameterstudien in relativ kurzer Zeit zu verwirklichen.

Die in dieser Arbeit betrachteten Parameterstudien für die Permeabilität sollten einen Wertebereich von $1 \cdot 10^{-10}$ bis $1 \cdot 10^{-18}$ m^2 haben. Je Zehnerpotenz sollten 2 bis 5 Werte untersucht werden, also zum Beispiel $1 \cdot 10^{-10}, 5 \cdot 10^{-10}, 1 \cdot 10^{-11}, 5 \cdot 10^{-11}, \ldots$. Um diese nicht-einheitliche Schrittweitenberechnung in ein Python-Programm einzugeben, wurde die in Quellcode 5.20 gezeigte Routine implementiert. Für jeden Wert eines Arrays `i0`, die Zehnerpotenz, und für jeden Wert eines Weiteren Arrays `k0`, die Schrittweite, wird der Rückgabewert entsprechend der unteren `lb` und oberen `ub` Grenze an die Rückgabeliste `f` angehängt.

Quellcode 5.20: Nicht-Reguläre Schrittweitenberechnung `nonregstep`.

```
def nonregstep(lb,ub,i0,k0):
   f=[]
   for i in i0:
     for k in k0:
        val=k*10**(-i)
        if lb<=val<=ub:
          f.append(val)
   return f
```

5.3.5 Beschreibung der Simulationsmodelle und Numerische Resultate

Wir wollen uns den am Anfang der Arbeit aufgestellten Fragestellungen widmen:

1. Welche Parameter müssen für gegebene Messwerte gelten? (Parameterbestimmung)
2. Wo müssen Messwerte aufgenommen werden, damit die gewonnenen Erkenntnisse über das Modell maximal werden? (Optimale Versuchsplanung)
3. Welches Modell liefert die beste Beschreibung des Problems? (Modelldiskriminierung)
4. Kann man die Problemlösung durch den Einsatz von einfacheren Modellen beschleunigen? (Space-Mapping)

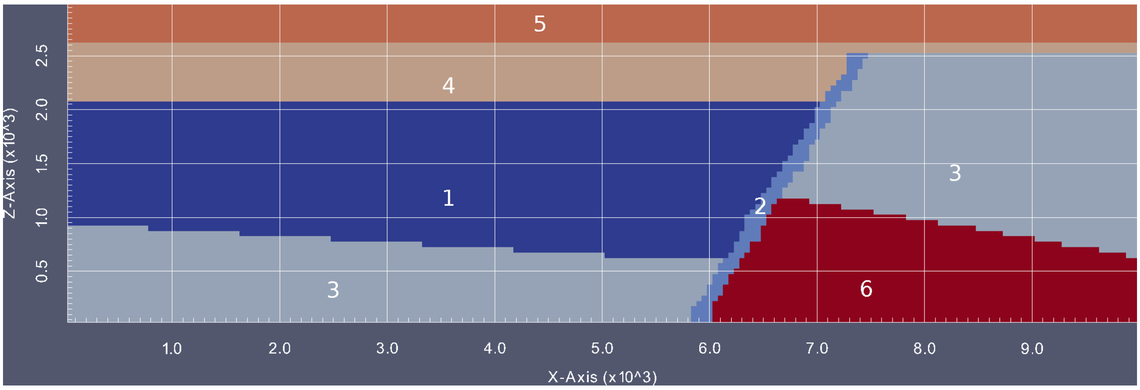
Dazu finden sich zwei unterschiedlich komplexe Untergrundmodelle für den Untergrund in der Region Perth, Australien und eine Region in der Toskana, Italien.

5.3.5.1 Perth

Wie Eingangs bereits kurz umrissen, dient dieser Arbeit als einfaches Beispiel aus dem Bereich der Geothermie ein Untergrundmodell aus der Region um Perth, Australien. Es handelt sich um ein einfaches 2D-Testmodell, welches für das MeProRisk Projekt entworfen wurde, um die untersuchten Fragestellungen möglichst schnell zu beantworten und ein Gefühl für die verwendeten Problemstellungen zu erhalten. Einen genauen Überblick über die zugrundeliegende geologische Struktur des Perth-Beckens bieten Niederau et al. [3].

Das Modell liefert einen repräsentativen Schnitt durch das Perth-Becken in Ost-West Richtung. Dabei werden die Gegebenheiten beim Übergang von On-Shore zu Off-Shore betrachtet. Die folgende Liste gibt einen Überblick über die in dem Modell vorhandenen Gesteinsschichten, welche in Abb. 5.3 dargestellt werden.

Abbildung 5.3 2D-Modell der Gesteinsschichten in der Region Perth aus [73].



(1) Parmelia Formation
(2) Bruchzone (engl. Fault)
(3) Yarragadee-Formation – Der Hauptgrundwasserleiter (Aquifer) des Beckens
(4) South Perth Shale – Die niedrig permeable Deckschicht des Reservoirs
(5) Leederville Formation – Die oberste permeable Deckschicht der Formation
(6) Cattamarra Coal Measures – Permable Schicht unterhalb der Yarragadee Formation

Wichtig dabei ist, dass die Gesteinsschicht (2), die Bruchzone, die in dem Modell interessanteste Schicht darstellt, da über sie nur wenige Informationen vorliegen und damit auch die höchsten

Tabelle 5.2 Verwendete Schichtparameter für die Simulation des Perth-Modells (aus [3]).

Schicht Nummer	Lithologische Schicht	Wärmeleitfähigkeit $[W/(mK)]$	Porosität	Permeabilität $[10^{-13}m^2]$
1	Parmelia	2.58	0.15	0.5
2	Fault	2.58	0.001	Unbekannt
3	Yarragadee	3.54	0.205	7
4	South-Perth	1.71	0.042	0.0002
5	Leederville	2.9	0.23	10
6	Cattamarra Coal	3.73	0.114	0.08

Unsicherheiten zu erwarten sind. Die Parameter der Gesteinsschichten sind in Tabelle 5.2 zu finden. Gesucht ist der unbekannte Permeabilitätsparameter der Bruchzone. Dieser kann einen Wert aus dem Wertebereich von 10^{-18} bis 10^{-10} m^2 annehmen. Das Modell hat eine Ausdehnung von 200×60 Zellen mit einer Größe von jeweils 50 Metern.

Um für das Perth-Modell eine Parameterstudie in EFCOSS berechnen zu können, wurde SHEMAT wie folgt compiliert:

```
make library-param use_matvec PROPS=const USER=none noomp hdf5
```

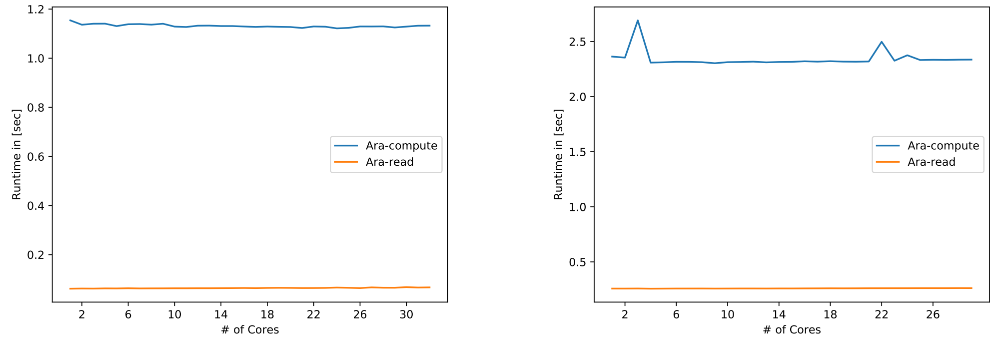
Für das OED-Problem war der Aufruf hingegen:

```
make library_oed PROPS=const USER=none omp hdf5
```

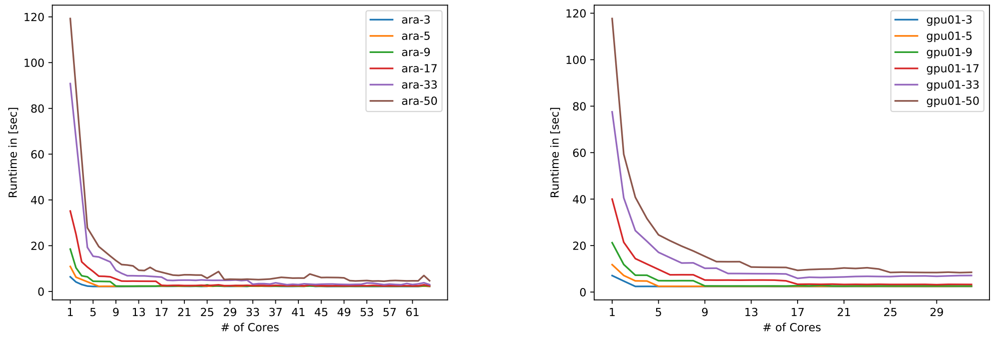
In Abb. 5.4 ist die Zeit zur Berechnung des Modells auf unterschiedlichen Architekturen zu sehen. Die Zeiten für den Ara-Cluster (ara) wurden auf einer Maschine mit 36 Kernen der Skylake-Standard Partition gemessen, für die Zeiten der AMD-Maschine (gpu01) wurde ein AMD Ryzen 1950X mit 16 Kernen und Hyperthreading verwendet. Zu sehen ist in (a) die Laufzeit der Berechnung des Vorwärtsmodells (compute) und die Zeit für das Einlesen der Simulationsbeschreibung (read), in (b) die Zeit zur Berechnung einer Ableitungsrichtung und deren Einlesen. Da wir hier keinen signifikanten Geschwindigkeitsgewinn durch die auf Verwendung der inneren Parallelität von SHEMAT sehen, wurde in einem weiteren Test die Anzahl der aktiven Ableitungsparameter für das Beispiel auf bis zu 50 erhöht, indem die Schicht zwei weiter verfeinert und somit neue Schichten eingefügt wurden. Dies ist ein rein synthetischer Test und spiegelt in keiner Weise die physikalischen Gegebenheiten der Region wider. Zu beobachten ist die sehr gute parallele Verarbeitung dieser Aufgabe mit den Zeiten in (c) und (d). In (e) und (f) wurde ein Speed-Up für den Ara-Knoten von bis zu 31 gemessen im Vergleich zur Ausführung mit einem Kern, für den AMD Ryzen ergibt sich ein Speed-Up von bis zu 14.

Weiterführend wollen wir uns nun dem Vergleich zwischen den unterschiedlichen AD Modi widmen, wenn diese für unterschiedliche Anzahl von Parametern eingesetzt werden. Abbildung 5.5 zeigt dabei die unterschiedlichen Ausführungszeiten T_{AD} im Vergleich zum Ursprungscode T_f für 1, 6 und 42 aktive Parameter. Dabei werden die Zeiten zur Berechnung der kompletten Jacobi-Matrix (blau) mit denen des Vorwärtsmodus $J\mathbf{v}$ (grün) und des Reversemodus $J^T\mathbf{v}$ (rot) verglichen. Es ist ersichtlich, dass die Zeiten für $J\mathbf{v}$ und J für einen Parameter gleich sind. Die Zeit zur Berechnung der vollen Matrix J ist dabei ein Vielfaches der Berechnung von $J\mathbf{v}$, da dies in SHEMAT mit mehrfachem Aufruf der gleichen Routine realisiert wird. Die wohl wichtigste Erkenntnis dieses Tests ist aber, dass die Ausführungszeit der Jacobi-Matrix-Vektor-Produkte für den Forward-,

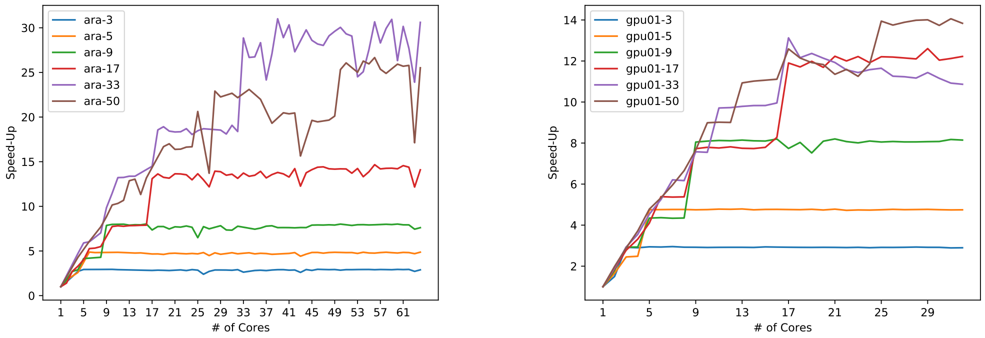
Abbildung 5.4 Zeitmessungen und Speed-Up für die Ausführung des Perth-Modells mit SHEMAT. In (a) ist die Zeit für die Vorwärtsrechnung zu sehen, in (b) die Zeit für die Berechnung einer Ableitungsrichtung. Zusätzlich ist die Zeit zum Einlesen des Modells angegeben. Die Abbildungen (c) und (d) zeigen die Zeit für die Berechnung von mehreren Ableitungsrichtungen, (e) und (f) den zugehörigen Speed-Up, links jeweils mit einem Ara-Knoten (ara), rechts mit dem AMD-Ryzen (gpu01).



(a) Simulations Zeit ARA (b) Skalare Ableitungs Zeit ARA



(c) Vektor-AD Zeit ARA (d) Vektor-AD Zeit AMD



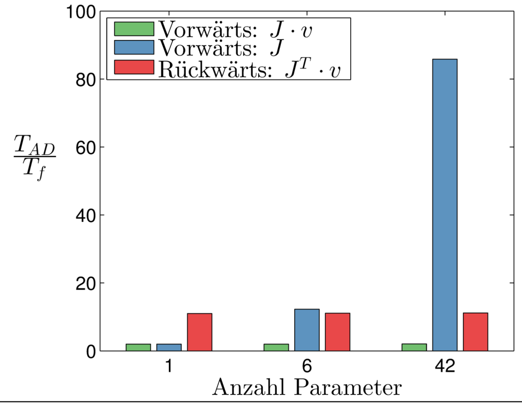
(e) Vektor-AD Speed-Up ARA (f) Vektor-AD Speed-Up AMD

Tabelle 5.3 Übersicht über die (simulierten) Temperaturmesswerte T in °C der Explorationsbohrung bei Perth, Australien. Die Spalte z gibt den vertikalen Index der Zelle der Temperaturmessung an. Die Zählung der z-Werte beginnt mit kleinen Werten am Boden des Reservoirs, nicht an der Oberfläche der Region.

z	T	z	T	z	T	z	T
59	22,8357	49	56,3876	39	79,7811	29	97,3088
58	25,8450	48	59,2712	38	81,4521	28	99,2318
57	29,4219	47	62,3695	37	83,7654	27	100,509
56	33,2524	46	64,1111	36	85,5313	26	101,402
55	36,7379	45	66,7228	35	87,3562	25	103,344
54	40,0120	44	68,6309	34	89,3601	24	103,437
53	43,2776	43	71,4233	33	91,1415	23	104,954
52	46,5661	42	73,9777	32	92,6562	22	106,915
51	50,5165	41	75,2263	31	94,4862	21	107,939
50	54,6430	40	77,9673	30	95,5531		

wie den Reverse-Mode, über alle drei Parameteranzahlen konstant bleibt. So kann bei vielen zu bestimmenden Parametern ein entscheidender Vorteil gegenüber der Berechnung der kompletten Jacobi-Matrix erzielt werden. Anzumerken sei, dass hierbei bewusst auf die Ausführung mittels Parallelisierung verzichtet wurde, um die Vergleichbarkeit zu gewährleisten.

Abbildung 5.5 Ausführungszeiten von AD Forward- und Reverse-Mode für unterschiedliche Anzahl Parameter am Beispiel Perth im Vergleich zum Ursprungscode. [74]



Parameterfit Perth Für die Region um Perth aus Abschnitt 5.3.5.1 findet sich an Position $x = 9000\ m$ ein Explorationsbohrloch. Dieses ist $z = 2000\ m$ tief und besitzt alle 50 m einen Temperaturmesspunkt. Abbildung 5.6 zeigt die Position des Explorationsbohrlochs im Schichtenmodell. Dieses ist als dicke rote senkrechte Linie gezeichnet. Die hier verwendeten synthetischen Messwerte stammen aus einem Simulationslauf und wurden mit Hilfe von White-Noise gestört. Sie sind in Tabelle 5.3 aufgeführt. Dabei ist zu beachten, dass die Bezeichnung der z-Werte von unten her erfolgt, d.h. ein kleiner Wert bedeutet eine tiefere Position. Gesucht wird die Permeabilität κ der Schicht 2. In Abb. 5.7 ist der Verlauf der Zielfunktion für verschiedene Eingaben κ_2 zu sehen.

Dieses Zielfunktional gilt es nun zu optimieren, indem in einem Test die in Quellcode 5.21 gezeigte Routine ausgeführt wurde. Es wurden die in Scipy implementierten Optimierer L-BFGS-B, Newton-CG, Nelder-Mead, TNC und SLSQP verwendet. Als Startwert für die Optimierung wurde immer der Wert $1 \cdot 10^{-13}$ gewählt.

Abbildung 5.6 Explorationsbohrloch im 2D-Modell der Region Perth.

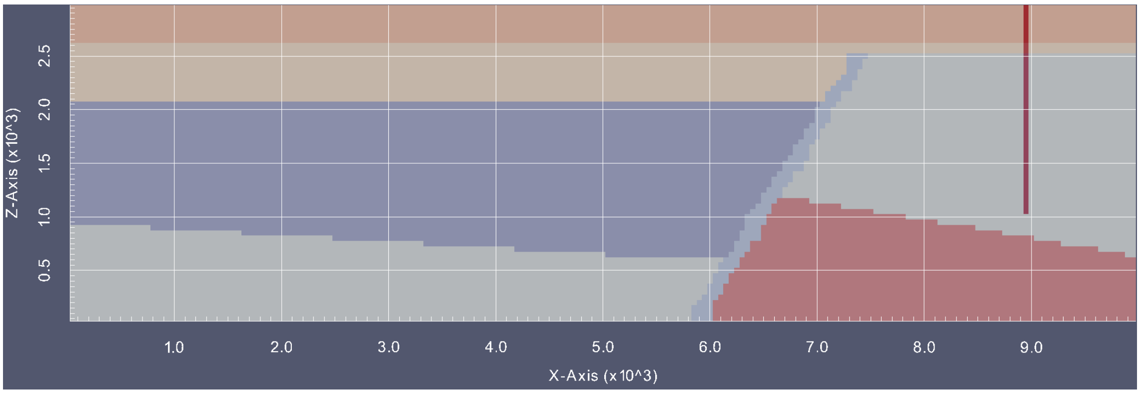
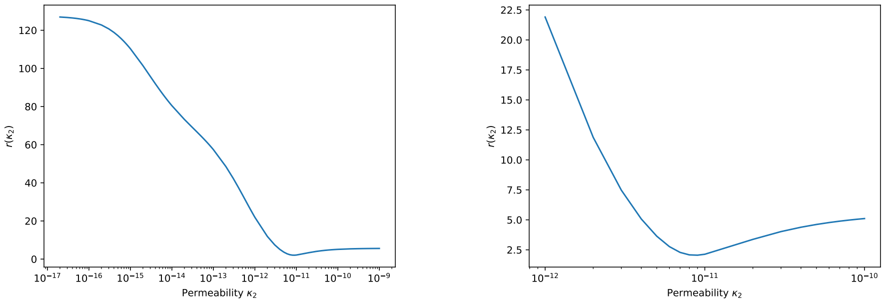


Abbildung 5.7 Verlauf der Zielfunktion der Parameterstudie für das Untergrundmodell Perth mit dem unbekannten Parameter κ_2.



Quellcode 5.21: Routine zum Aufrufen von verschiedenen Optimierern.

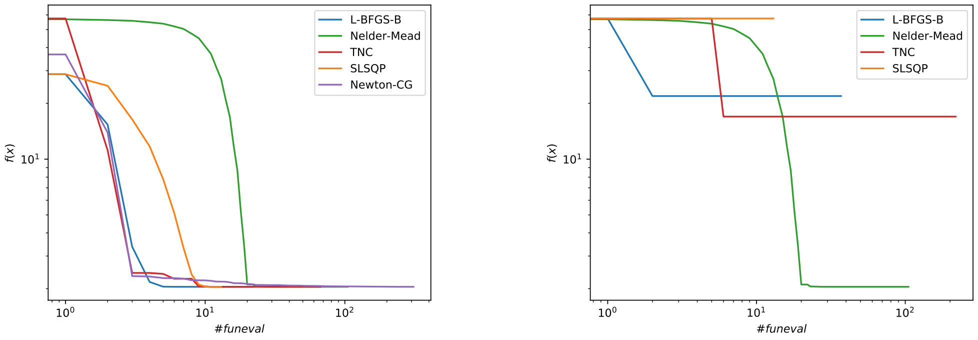
```
from numpy import *
from EFCOSS.Applications.shemat import ShematDatafit
from EFCOSS import *
import matplotlib.pyplot as plt
methods=[’L-BFGS-B’,’Newton-CG’,’Nelder-Mead’,’TNC’,’SLSQP’]
keys = [’status’,’fun’,’nfev’,’njev’,’nit’,’message’]

problem=ShematDatafit.ShematDatafit("shemat_datafit","oed_inv")
problem.setupEFCOSS()
problem.initSimulationInterface()
problem.initObjectivefunction()
solutions=[]
stats=[]
funevals=[]

for method in methods:
    problem.resetStats()
    solutions.append(problem.runOptimizer([1.0],method=method))
    stats.append(problem.getStats().copy())
```

Die Ergebnisse der untersuchten Optimierer sind in Tabelle 5.4 zu sehen. Alle Optimierer erreichen nach wenigen Iterationen oder Ausführungen der Zielfunktion eine vernünftige Genauigkeit. Der mit Abstand beste Optimierer ist hier SLSQP mit 14 Funktionsauswertungen und 14 Ableitungsausführungen, dicht gefolgt von L-BFGS-B mit 23 und TNC mit 92 Funktionsausführungen. Dabei muss jedoch beachtet werden, dass die beiden letzteren immer die Methode zur Berechnung des Gradienten der Funktion aufrufen, wohingegen die anderen Methoden dies getrennt ausführen. Eine Ausnahme bildet das Nedler-Mead Verfahren, welches ohne Ableitungen auskommt, jedoch deutlich häufiger die Funktion auswerten muss, 153 mal. Die schlechteste Performance zeigt das Newton-CG Verfahren, welches nicht in die gewünschte Genauigkeit von 10^{-8} kommt und dazu auch noch 311 Funktionsausfrufe und 378 Ableitungsberechnungen durchführt.

Tabelle 5.4 Ausführungsergebnisse verschiedener Optimierungsalgorithmen für das Datenfitproblem in der Region Perth mit AD-Reverse-Mode generierten Ableitungen.

Optimierer	Erfolg	κ_2^*	$r^*(\kappa_2^*)$	∇r	# r	# ∇r	# Iter	Begründung
L-BFGS-B	0	86.651	2.047	-6.904e-06	23	N/A	9	'CONVERGENCE REL REDUCTION OF F <= FACTR*EPSMCH'
Nelder-Mead	0	86.650	2.047	N/A	153	N/A	64	Optimization terminated successfully.
TNC	2	86.651	2.047	9.036e-06	92	N/A	8	'Converged ($\lvert x_n - x_{n-1}\rvert \approx 0$'
SLSQP	0	86.650	2.047	-7.730e-7	14	14	14	Optimization terminated successfully.
Newton-CG	2	86.0717	2.047	-0.031	311	378	26	'Warning: Desired error not necessarily achieved due to precision loss.'

Abbildung 5.8 Überblick über das Konvergenzverhalten der unterschiedlichen Optimierer für das Datenfit-Problem in der Region Perth.



(a) Automatisches Differenzieren (b) Finite Differenzen

In Abb. 5.8 (a) findet sich das Konvergenzverhalten der untersuchten Optimierer bei Verwendung des AD-Reverse-Modes zur Berechnung des Gradienten. Die Kurven von SLSQP und L-BFGS-B sind nahezu identisch, einzig die Anzahl der Funktionsauswertungen ist unterschiedlich. Zu sehen ist, dass die Optimierer L-BFGS-B, TNC und SLSQP schon nach rund zehn Funktionsauswertungen nahe an der entsprechenden Lösung sind. Nelder-Mead erreicht nach rund 20 Funktionsauswertungen das Optimum.

Nun nutzen wir obigen Code, um zu untersuchen, wie sich das Konvergenzverhalten entwickelt, wenn keine exakten Ableitungen zur Verfügung gestellt werden, der Optimierer also finite Differenzen einsetzen muss. Dies geschieht in den Scipy-Optimierern, indem die Übergabe `jacobian=False` gesetzt wird. Tabelle 5.5 stellt dabei die für diesen Lauf genutzten Optimierer und deren Ergebnisse dar. Auffällig ist, dass die meisten der Optimierer ohne die Ableitungen viel häufiger die Funktionen aufrufen, was auch erwartbar ist. So nutzt zum Beispiel der L-BFGS-B 2-mal mehr Funktionsauswertungen im Vergleich zu dem Lauf mit AD. Zusätzlich zeigt sich, dass die Optimierer viel häufiger in ein lokales Minimum laufen und den Zielfunktionswert von 2.047 (wie wir ihn in der obigen Tabelle gesehen haben) nicht mehr erreichen. Der Optimierungsalgorithmus Newton-CG benötigt zwangsläufig eine Ableitungsroutine, es erfolgte ein Abbruch bei Übergabe von `jacobian=False` an den Konstruktor. Da er auch keine gute Performance mit den AD generierten Ableitungen zeigt, wurde auf eine Ausführung mit den in EFCOSS gegebenen finiten Differenzen verzichtet.

Tabelle 5.5 Ausführungsergebnisse verschiedener Optimierungsalgorithmen für das Datenfitproblem in der Region Perth mit finiten Differenzen.

Optimierer	Erfolg	κ_2^*	$r^*(\kappa_2^*)$	∇r	# r	# ∇r	# Iter	Begründung
L-BFGS-B	0	20.402	11.6348	18.314	80	N/A	3	CONVERGENCE: REL REDUCTION OF F <= FACTR*EPSMCH
Nelder-Mead	0	86.650	2.047	N/A	153	N/A	64	Optimization terminated successfully.
TNC	2	1.027	57.096	5.748	7	N/A	2	Converged ($\|x_n - x_{n-1}\| \approx 0$)
SLSQP	0	21.615	10.944	0.819	49	6	6	Optimization terminated successfully.

Abbildung 5.8 (b) zeigt die Konvergenz der Optimierer, wenn interne finite Differenzen verwendet werden. Zu sehen ist, dass alle Optimierer bis auf TNC den Wert der Zielfunktion verbessern können. Im Vergleich zur AD-Version zeigen sich aber deutliche Unterschiede. So ist bei den Optimierern L-BFGS-B und SLSQP nach zehn Funktionsauswertungen ein Minimum gefunden, dieses liegt aber weit weg von dem zu erwartenden Minimum.

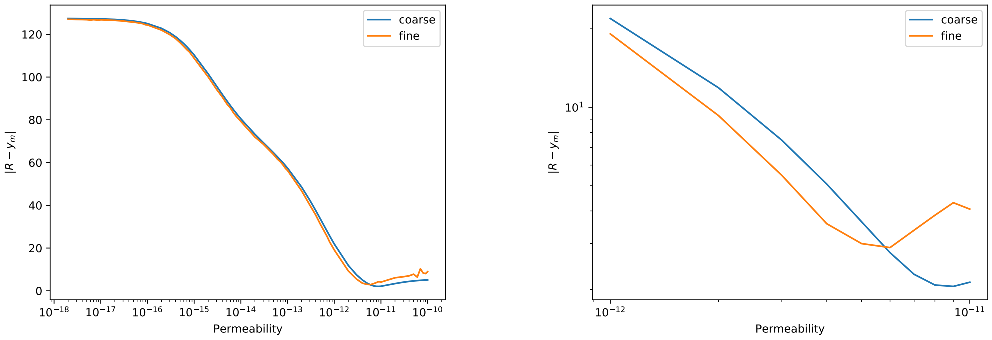
Space-Mapping Perth Aufbauend auf den Ergebnissen und Implementierungen für den Parameterfit wurde zusätzlich das Space-Mapping für das Perth Beispiel implementiert. Hierbei wurde das ursprüngliche Modell als grobes Modell eingeführt und ein weiteres, feines Modell über eine Verfeinerungsroutine 10-fach in X-Richtung und 4-fach in Z-Richtung erweitert.

Wir sehen uns zuerst die Ergebnisse der Zielfunktionale im Vergleich der feinen und groben Modelle an, vgl. Abb. 5.9 (a). Zu sehen ist, dass beide Modelle für die Auswertung des Simulationsmodells geeignet sind. Das Optimum beider ist wie zu erwarten nicht an der gleichen Stelle zu sehen. Ein genauerer Blick auf die Abb. 5.9 (b) zeigt diesen Umstand nochmals im Bereich zwischen $1 \cdot 10^{-12}$ und $1 \cdot 10^{-11}$. Weiterhin ist zu sehen, dass sich beide Funktionen in der Nähe des Minimums der feinen Funktion schneiden. Dies sind also beste Voraussetzungen, um Space-Mapping anzuwenden.

Nun sehen wir uns einen Plot für den Abstand zwischen grobem und feinem Modell an. Dieser ist in Abb. 5.10 zu sehen. Auffällig ist das Tal in der Mitte, also dort wo die beiden Funktionen ähnliche Parameter haben, sind auch deren Antworten ungefähr gleich.

In Quellcode 5.22 ist der Code eines Laufzeitscriptes zum Ausführen der Space-Mapping-Methode gezeigt. Zuerst wird eine `ShametDafitSpacemap` Instanz mit dem Namen `shemat_datafit` und dem Dateinamen `oed_inv` für die Modelle erzeugt. Dann werden die intern genutzten Parame-

Abbildung 5.9 Vergleich der Zielfunktionen für ein grobes und verfeinertes Perth-Modell zum Einsatz in Space-Mapping.



(a) Ganze Funktion

(b) Ausschnitt zwischen 10^{-12} und 10^{-11}

Abbildung 5.10 3D-Plot für den Abstand zwischen den Simulationsergebnissen des feinen und groben Modells für das Perth-Modell.

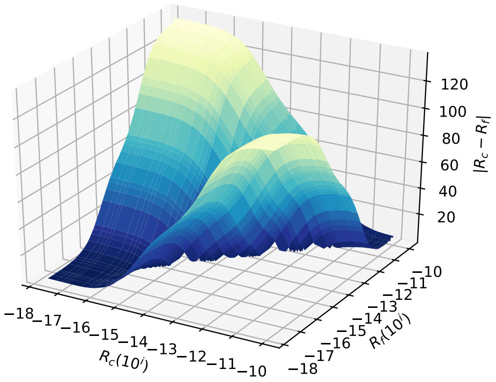
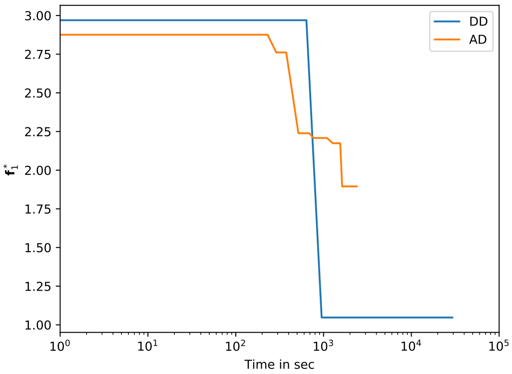


Abbildung 5.11 Vergleich des minimalen Funktionswertes $\mathbf{f}_1^*$ zu einer bestimmten Zeit des Aggressive-Space-Mapping Algorithmus bei Verwendung von finiten Differenzen (DD) und automatischem Differenzieren (AD) für die Parameterbestimmung des Perth Reservoirs.



ter mittels `setupEFCOSS` gesetzt. Im Anschluss kann das SpaceMapping durchgeführt werden. Hier ist ein Beispiel gegeben, bei dem die Ableitungen für die Optimierung mittels finiten Differenzieren erzeugt werden sollen.

Quellcode 5.22: Ausführungsscript für das SpaceMapping in SHEMAT. Dabei sollen für die in der Initialisierung verwendete Opimierung AD, für die Schritte des SpaceMapping-Algorithmus jedoch finite Differenzen verwendet werden (`jac=[False,False]`).

```
from EFCOSS.Applications.shemat import ShematDatafitSpacemap

shematDatafit=ShematDatafitSpacemap.ShematDatafitSpacemap("shemat_datafit","oed_inv")
shematDatafit.setupEFCOSS()
shematDatafit.run_spacemap_asm('_j00',jac=[False,False])
```

In Abb. 5.11 ist der zeitliche Verlauf der minimalen Zielfunktion $\mathbf{f}_1 = \|\mathbf{R}_c(\mathbf{x}_c) - \mathbf{R}_f(\mathbf{x}_f)\|$ der Optimierung mit dem Aggressive-Space-Mapping in logarithmischer Skalierung, als Vergleich zwischen finiten Differenzen (DD) und automatischem Differenzieren (AD) zu sehen. Es fällt auf, dass das Ergebnis der beiden Optimierungen unterschiedlich ist. Allerdings wird durch die Verwendung von AD deutlich weniger Zeit verwendet, als es mit FD benötigt wird. Die Anzahl der benötigten Iterationen des Algorithmus sind 24 bei finiten Differenzen und 28 bei Verwendung von AD. Bis zum ersten Knick der Kurve ist die Zeit aufgetragen, die die Initialisierungsroutine benötigt. Danach startet der eigentliche Schritt des Algorithmus.

OED Perth Die Ergebnisse dieses Abschnitts basieren auf den Erkenntnissen aus [73]. Es wird das unter Abschnitt 5.3.5.1 gegebene Modell benutzt, um ein weiteres Bohrloch einzusetzen. Dabei wird mit der Funktion f die Temperatur berechnet. Da es sich hier um ein 2D-Modell handelt, ist die Position des Bohrlochs nur vom Parameter für die X-Koordinate abhängig. Als Tiefe wählen wir die gleiche Tiefe wie beim Explorationsbohrloch aus Abschnitt 5.3.5.1 von zwei Kilometern mit jeweils 50 Metern Abstand zwischen den Messpunkten. Der gesuchte Parameter p ist die Permeabilität der zweiten Schicht, d.h. $p = \kappa_2$. Um nun OED zu verwenden wurde die Klasse `ShematOED` aus Kapitel 5.3.4 eingesetzt.

Das verwendete Startscript ist in Quellcode 5.23 zu sehen. Dabei sind die Eingaben für die Feldgröße (`gridX`,`spanX`, usw.) die gleichen wie in der Problembeschreibung. Entscheidend ist die

Angabe für `minx` und `maxx`, da hier die zu untersuchenden Grenzen der X-Koordinate für das Bohrloch definiert sind. Die Tiefe der Bohrlöcher wird mit `minz` und `maxz` definiert. Der Parameter `saveJacobi` wird verwendet, um die einmal mit SHEMAT generierten Ableitungsfelder in EFCOSS abzuspeichern und bei dem Sweep entsprechend zu nutzen. Dies beschleunigt die Ausführung massiv, da pro Parameter aus `k2` nur noch ein Aufruf an SHEMAT erfolgen muss.

Danach werden die Übergabeparamter eingelesen. Diese werden bei der Ausführung des Scriptes wie folgt erwartet:

```
run.py <OED-Kriterium> <#Threads> <#EFCOSS-Threads> \
       <#Outer-Threads> <#Inner-Threads> <JobID> <Mode>
```

Für das OED-Kriterium muss ein String mit dem Optimalitätskriterium übergeben werden, also etwa `D_opt` oder `A_opt`. Die Anzahl der `Threads` sind die maximal gestarteten Threads des Programms, also alle nachfolgenden Threadnummern multipliziert. Mit den `EFCSOSS-Threads` wird die Anzahl der parallel arbeitenden EFCOSS Instanzen definiert, also etwa ein Thread pro Parameter aus `k2` oder ein Thread für die gesammte Parametersuche. Die beiden Übergabeparameter `Outer-` und `Inner-Threads` geben den internen Parallelitätsgrad von SHEMAT an. `Outer-Threads` gibt dabei die Anzahl der parallel berechneten Ableitungsrichtungen in der Jacobi-Matrix an (in diesem speziellen Fall also `1`) und `Inner-Threads` gibt die Anzahl der Threads für die innerhalb der Berechnung genutzten parallelen Ausführungseinheiten an. Die `JobID` wird verwendet, wenn mehrere Ausführungen des Laufzeitscriptes mit unterschiedlichen Parametern getestet werden sollen. Hierbei werden unterschiedliche Ergebnisdateien erzeugt. Dies ist auch wichtig, damit sich gleichzeitig laufende Shemat-Instanzen nicht gegenseitig beeinflussen.

Der Parameter `Mode` gibt die auszuführende Funktion an,

- `mode=0` Keine Optimierung, sondern berechne den Zielfunktionswert an verschiedenen Bohrlochpositionen,
- `mode=1` Optimierung.

Modus 0 ist dabei besonders interessant, da er einen Überblick über die Zielfunktion erzeugt, welches für die Geophysik einen entscheidenden Vorteil bietet. Anstelle eines einzigen Wertes, den die Optimierung erzeugt, wird, zusammen mit unterschiedlichen Parametersätzen κ_2, eine Entscheidungshilfe gegeben, die potentiell interessante Areale der Region aufzeigt. Dafür benötigt die Ausführung normalerweise deutlich länger. Mittels `saveJacobi` kann die Ausführungszeit signifikant verkürzt werden.

Danach wird eine Ausgabedatei für die Zeitmessung angelegt und die Position und Tiefe des vorgegebenen Bohrlochs hinzugefügt. Es werden jetzt drei Methoden definiert, die wir für die Ausführung benötigen. Die Methode `init` initialisiert die `ShematOED` Klasse mit den Eingabeparametern von weiter oben. Die beiden anderen Methoden werden in der Ausführung genutzt, `run_func` für den Modus 0, `run_solve` für Modus 1.

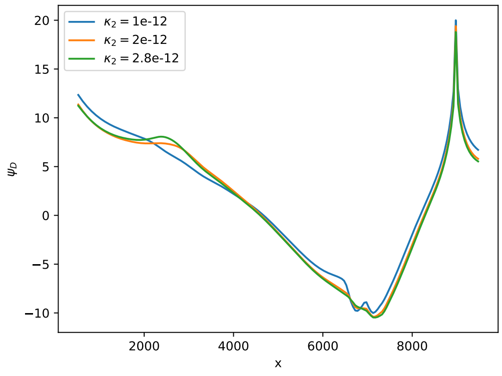
Es wird ein Multiprocessing-Pool von EFCOSS-Threads angelegt, vergleiche das einfache Beispiel in Abschnitt 3.4.1. Dabei werden die erzeugten Permeabilitätswerte aus `k2` auf die einzelnen Threads verteilt und deren Ausführung beginnt mit der `map`-Funktion.

Quellcode 5.23: Ausführungsscript für das OED Problem in der Region Perth.

```
k2=nonregstep(1.0E-18,1E-10,range(18,9,-1),range(1,10))
```

```
def initData(d):
    for j in k2:
        d.append([j])

d=[]
initData(d)

gridX=200
gridY=1
gridZ=60

spanx=50
spany=50
spanz=50

minx=525
maxx=9525
miny=25
maxy=26
minz=1025
maxz=2925
savejacobi=True


opt=sys.argv[1]

threads=int(sys.argv[2])

efcoss_threads=min(int(sys.argv[3]),len(d))
outer=int(sys.argv[4])
inner=int(sys.argv[5])

jobid=int(sys.argv[6])
mode=int(sys.argv[7])

output = open('time-%(x)i.out'%{'x':threads},'a')

original_positions=[]
original_minz=[1025]
original_maxz=[2925]
original_positions=[8975.0,25.0]
shemat=None

def run_func(i):
   global d,shemat
   shemat.run_func_shemat(d[i],i,0)

def run_solve(i):
   global d,shemat
   shemat.run_solve_shemat(d[i],i)

def init():
   global shemat,opt,jobid,minx,maxx,spanx,miny,maxy,spany,minz,maxz,spanz,original_positions,inner,
        outer,efcoss_threads,savejacobi
   shemat=ShematOED.ShematOED(opt, jobid, minx, maxx, spanx, miny, maxy, spany, minz, maxz, spanz,
        original_positions, original_minz, original_maxz, inner, outer,str(efcoss_threads) +"-" + str(
```

Abbildung 5.12 Verlauf der Zielfunktion des Optimal Experimental Designs für verschiedene Werte für κ_2 mit dem D-Optimalitätskriterium entlang der X-Achse im Perth-Reservoir.

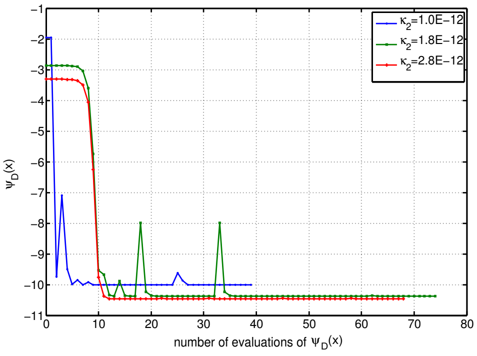


```
        outer) +"-" + str(inner),False, savejacobi, gridX, gridY, gridZ)

init()
p = Pool(processes=efcoss_threads,initializer=init)
start=time.time()
if mode==0:
    p.map(run_func,range(len(d)))
else:
    p.map(run_solve,range(len(d)))
end = time.time()
output.write('%(threads)i %(o)i %(j)i %(i)i %(time)f\n'%{'o':efcoss_threads,'i':inner,'j':outer,'time':(
     end-start),'threads':threads})
output.flush()
output.close()
```

Zunächst wollen wir uns die Zielfunktion ψ_D für die D-Optimalität mit unterschiedlichen Parametern κ_2 ansehen. Der Verlauf dieser ist in Abb. 5.12 zu sehen. Zu sehen ist, dass jede der Kurven ein globales Minimum bei einer X-Position von 7150 Meter besitzt.

In Abb. 5.13 ist der Verlauf der Optimierung mit dem L-BFGS-B Optimierer für verschiedene Parameter κ_2 zu sehen. Diese Abbildung wurde bereits in [73] besprochen.

Abbildung 5.13 Werte der Zielfunktion der D-Optimalität in der Optimierung mit dem L-BFGS-B Optimierer für verschiedene Parameter κ_2, aus [73].



Im Folgenden wollen wir eine kurze Laufzeitbetrachtung für dieses Problem anstellen. Auf dem

Lofar-Cluster wurden folgende Beobachtungen durchgeführt. Für die Auswertung eines Vorwärtslaufes von SHEMAT benötigen wir in dieser Simulation ungefähr eine Sekunde Laufzeit auf einer Maschine. Die Ausführungszeit zur Berechnung der Fisher-Matrix beträgt rund drei Sekunden.

Wenn wir die Ausführung ohne Parallelität und ohne den Einsatz der Speicherung der Jacobi-Matrix durchführen würden, würde diese für den kompletten Parameterraum, mit insgesamt 81 Parametern und jeweils 180 Bohrlochpositionen, seriell in rund zwölf Stunden beendet sein. Da es sich hierbei um ein einfach zu parallelisierendes Problem handelt, können die Ausführungszeiten durch Verteilung der Parameter auf einen eigenen Prozess je Parameter mittels MPI verteilt werden. So sinkt die Ausführungszeit maßgeblich auf rund zehn Minuten.

Durch das Speichern der Jacobi-Matrix kann darüber hinaus eine deutliche Steigerung der Geschwindigkeit, selbst auf dem seriellen Rechner erwartet werden. Da die Auswertung dann nur ein mal pro Parameter durchgeführt werden muss, sinkt die Ausführungszeit auf rund vier Minuten. Parallelisiert ist die Aufgabe dann in unter zehn Sekunden abgeschlossen.

Wir wollen in einem hypothetischen Szenario untersuchen, was bei der Verwendung von mehreren Parametern für eine Laufzeit zu erwarten ist. Wir nehmen an, dass der Permeabilitäts-Parameter aller sechs Schichten in diesem Beispiel geschätzt werden soll. Wieder soll eine niedrigere Auflösung pro Parameter von nur fünf Stützpunkten gewählt, und mit je 180 Bohrlochpositionen die Ergebnisse bestimmt werden. Zur Berechnung der Fisher-Matrix wird nun aber mehr Zeit benötigt, rund zehn Sekunden. Somit würde sich

$$T_{seriell} = 5^6 \cdot 180 \cdot 10 \text{ sec} \approx 7800 \text{ Stunden} \approx 325 \text{ Tage}$$

als serielle Laufzeit ergeben. Mit dem Lofar-Cluster ist eine Laufzeit von rund drei Tagen gemessen worden, ein Speed-Up von ungeführ 100, was bei den 136 Rechenkernen auch zu erwarten war. Weiter können wir die Laufzeit senken, wenn die Ableitungen gespeichert werden. Hierbei ist eine Laufzeit von rund 30 Minuten auf der gleichen Maschine ermittelt worden.

5.3.5.2 Toskana

In der Zusammenarbeit mit der Firma Enel Green Power, Italien, wurde ein Untergrundmodell für eine Region in der Toskana in Italien entwickelt. Es umfasst ein Gebiet von 22 km × 15 km und wurde von Enel Green Power für die Exploration für den späteren Gebrauch als möglichen Standort eines Geothermiekraftwerks ausgewählt. Das untersuchte Areal ist Teil des Orogengürtels der nördlichen Apenninen.

In dieser Region in der Toskana finden sich, bedingt durch seismische Aktivität, sogenannte thermische Anomalien, welche einen sehr hohen lokalen Temperaturgradienten von mehr als $100\,°\mathrm{C}\,\mathrm{km}^{-1}$ besitzen [75, 76]. Dies wird wahrscheinlich durch einen relativ jungen Granitpluton hervorgerufen. Dieser befindet sich in 4.5 km bis 6 km Tiefe. Durch Variationen der Tiefe dieses Gesteins werden hohe Differenzen im thermischen Gradienten beobachtet. Wasserfluss durch die heterogenen aufgebrochenen Sedimente, sowie metamorphe Felsschichten verhindern eine gleichmäßige Ausbreitung der Wärme und machen es schwierig, eine geeignete Position für ein Geothermiekraftwerk zu finden, welches auch auf lange Sicht produktiv arbeiten kann.

Die im Folgenden verwendete Untergrundstruktur des Gebiets wurde mit Hilfe von reflektionsseismischen Daten erstellt, bei denen fünf sich gegenseitig überlappende seismische Profile genutzt wurden. Eine genaue Erklärung der verwendeten Daten findet sich in Ebigbo et al. [77]; die geologische Betrachtung wurde bereits vorher durch Batini et al. [78] und Brogi [79, 80, 81, 82]

Abbildung 5.14 Das verwendete Schichtenmodell für die Simulation des Areals in der Toskana (aus [83]).

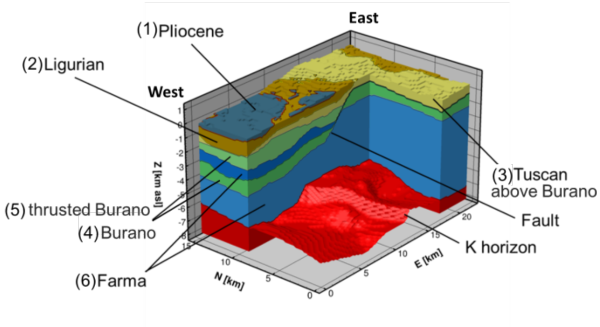


Tabelle 5.6 Verwendete Schichtparameter für die Simulation des Toskana-Areals (aus [83]).

Schicht Nummer	Lithologische Schicht	Wärmeleitfähigkeit $[W/(mK)]$	Wärmeproduktionsrate $[\mu W/m^3]$	Porosität	Permeabilität $[10^{-15}m^2]$
1	Pliocene	1.30	0.14	0.10	0.1
2	Ligurian	2.54	0.14	0.10	0.1
3	Tuscan above Burano	4.01	0.32	0.10	1.0
4	Burano	4.73	0.16	0.02	10
5	Thrusted Burano	4.73	0.16	0.02	Unbekannt
6	Farma	4.10	1.42	0.03	0.6

durchgeführt.

Abbildung 5.14 zeigt das Untergrundmodell der Region mit verschiedenen lithologischen Gesteinsschichten, auf welche im Folgenden kurz eingegangen wird. Die folgenden Schichten wurden in das geologische Untergrundmodell eingefügt, um eine möglichst realistische Darstellung des Untergrundes zu erhalten:

(1) Pliozän mit quartärer und neogener Schicht: Sedimente, welche sich in jungen Grabensystemen gesammelt haben und die darunterliegenden vor-neogenen Schichten überdecken

(2) Ligurische und subligurische Schichten, welche Teile der ozeanischen Kruste beinhalten

(3)–(5) Toskanische Überschiebungsdecke Gesteinsschichten: werden von Kontinentalrandablagerungen (Triassisch bis Miozän) des Adria Paleorandes repräsentiert.

  (3) karbonatische und ozeanisch-turbiditische Folge (“Tuscan units above Burano” abgekürzt mit TUaB)

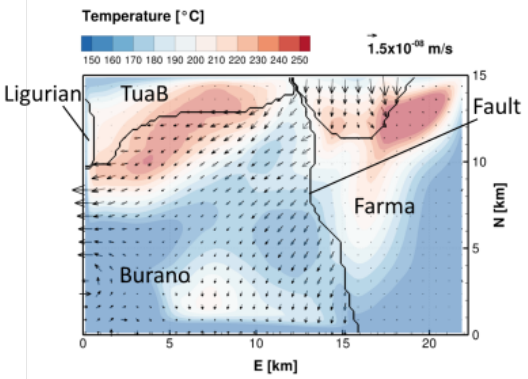
  (4) und (5) Triassische Burano Formation, welche die Basisschicht der Toskanischen Decke bildet. Sie besteht hauptsächlich aus Kalkstein und Anhydrit.

(6) Unterer toskanischer metamorpher Komplex: wird auch Farma Formation genannt.

Hinzu kommt noch der K-Horizont, welcher eine primäre seismische Reflektionsschicht darstellt, die von Geologen als rheologische Grenzschicht interpretiert wird [84, 85]. Die genaue Natur des K-Horizontes ist bis jetzt nicht klar, es zeigt sich jedoch an bereits funktionsfähigen Geothermalkraftwerken, dass der K-Horizont eine Temperatur von 400 °C bis 450 °C besitzt und als primäre Wärmequelle des Reservoirs dient.

Das in Abb. 5.14 gezeigte strukturelle Modell hat eine 3-Dimensionale Größe von 22 km ×

Abbildung 5.15 Schnitt durch die Region in der Toskana. In einer Tiefe von 1,25 km werden die Temperatur (farbige Skala), sowie der Fluss des Wassers dargestellt. Die schwarzen Linien zeigen die Grenzen der untersuchten Gesteinsschichten (aus [83])



15 km × 9 km in E (Ost), N (Nord) und Z (Tiefe). Für die Simulation wurde es in etwa eine halbe Million Blöcke der Größe 250 m × 250 m × 100 m zerlegt. Tabelle 5.6 zeigt die in dem Modell verwendeten Werte für Wärmeleitfähigkeit, Wärmeproduktionsrate, Porosität, sowie Permeabilität. Wasserdichte, Viskosität, spezifische Enthalpie und Wärmeleitfähigkeit sind in dem Modell Funktionen des Druckes und der Temperatur, wobei die Korrelationen der IAPWS (International Association for the Properties of Water and Steam) verwendet werden [76].

Um das Modell abschließend zu beschreiben, benötigt man noch die Randbedingungen bzw. Anfangswerte. Als untere Randbedingung wird der K-Horizont genutzt. Wie bereits beschrieben, besitzt er eine Temperatur von bis zu 450 °C. Dabei wird angenommen, dass es keinen Fluidfluss durch den K-Horizont gibt. Als obere Grenze dient die in der Region vorliegende Topographie, die mit einem Temperaturgradienten von 1 K je 184 m Höhe angenommen wird (vgl. [86]). Zusammen mit den in der Region gemessenen Daten einiger flacher Bohrungen und dem Lufttemperaturgradienten können damit die Temperaturen in der oberen Grenzregion bestimmt werden. Zur Bestimmung der seitlichen Randbedingungen wurde eine Steady-State Simulation des Wärmetransports durchgeführt, wobei in den seitlichen Rändern kein Wärmefluss zugelassen wurde. Die Simulationsergebnisse dienen für das Modell als seitliche Temperaturrandbedingungen. Der hydraulische Head aller Randbedingungen wird gleich der Topographie gesetzt.

Ausgehend von diesen Definitionen kann die Simulation gestartet werden. Abbildung 5.15 zeigt einen Schnitt durch die Region in 1.25 km Tiefe (unter N.N.). Darin sind die Temperatur sowie die Strömungsvektoren der Burano-Schicht und anderer Schichten gezeigt. Zu sehen ist eine generelle Nord-Süd Strömungsrichtung des Reservoirfluids, welche der Topographie folgt. Auffällig ist der Fluss durch die (permeable) Bruchzone (Fault), welche die beiden Burano-Schichten im Osten und Westen verbindet. Die genaue Betrachtung des Modells ist Ebigbo et. al. [77] zu entnehmen.

Zusätzlich zu dem großen Modell wurde ein 2-D Schnitt durch die Region als Modell angefertigt. Dieses ist ähnlich zu dem Modell von Perth angelegt, besitzt aber zwei Bruchzonen. Dieses einfachere Modell ist in Abb. 5.16 zu sehen.

Nun wollen wir die Ausführungszeiten dieses Modells auf verschiedenen Architekturen untersuchen. Da das Programm mittels OpenMP parallelisiert wurde, lohnt sich der Einsatz von Maschinen mit geteiltem Speicher für die Berechnung der Simulation.

Abb. 5.17 zeigt die erzielten Laufzeiten bei der Verwendung von einem Laptop mit vier Kernen,

Abbildung 5.16 Schnitt durch die Region in Nord-Süd Richtung als einfaches 2D-Modell.

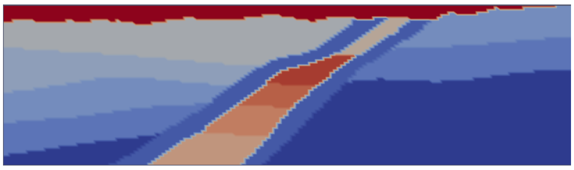
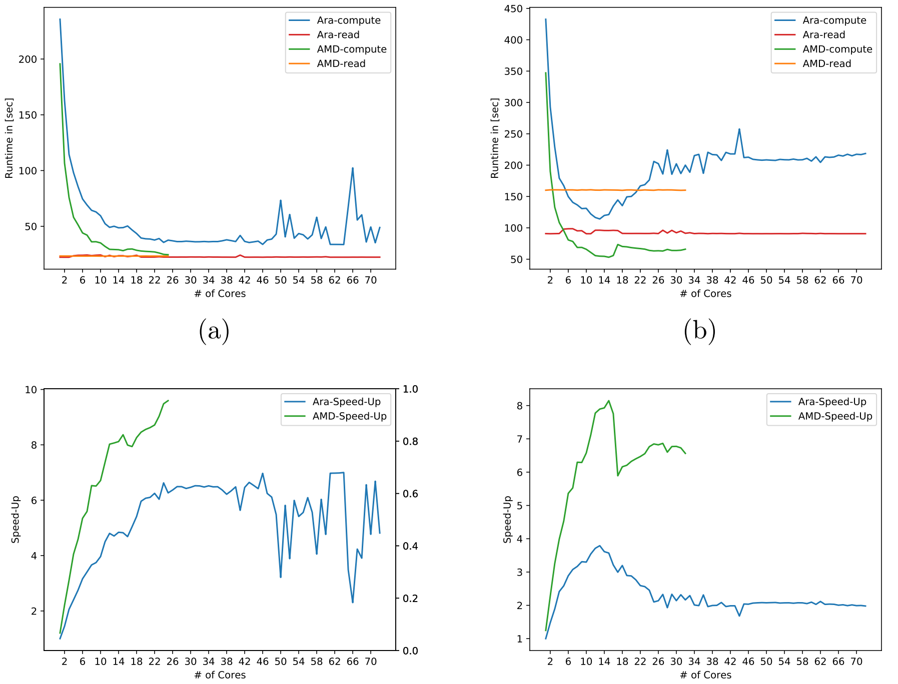


Abbildung 5.17 Zeitmessungen und Speed-Up für die Ausführung des 3D-Toskana-Modells mit SHEMAT. In (a) ist die Zeit für die Vorwärtsrechnung zu sehen, in (b) die Zeit für die Berechnung der kompletten Jacobi-Matrix.



(a) (b)

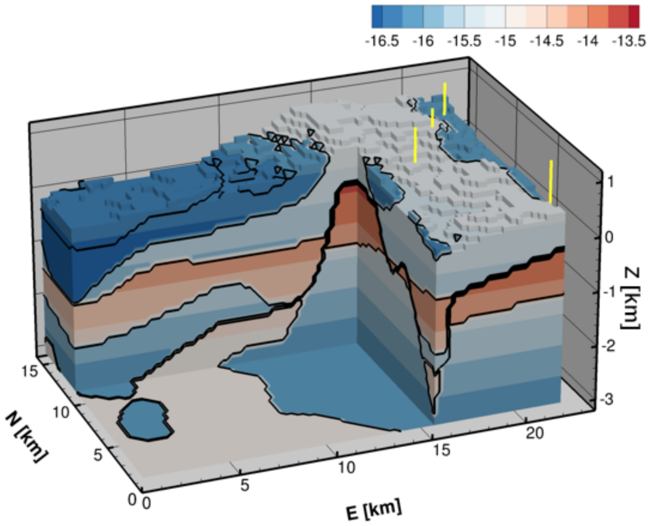
einem AMD-Rechner mit 16 Kernen, sowie einem Skylake-Knoten des Ara-Clusters mit 36 Kernen. Alle diese Architekturen unterstützen Hyper-Threading, welches zu diesem Test aktiviert wurde. Deshalb verdoppeln sich die untersuchten Kernanzahlen.

Für die Vorwärtsrechnung zeigt sich ein erwartetes Verhalten bei allen untersuchten Architekturen. Bis zur Anzahl der vorhandenen (echten) Kerne in den Architekturen sinkt die Ausführungszeit, wird dann aber durch die Verwendung von Hyper-Threading teils deutlich verlangsamt. Der maximale Speed-Up für den Skylake-Knoten ist 6,64 (bei Verwendung von 23 Threads).

Bei der Ausführung der Berechnung der Jacobi-Matrix auf der Ara-Cluster-Maschine mit der Skylake CPU von Intel ist eine Verbesserung in der Laufzeit bis zu einer Kernanzahl von 13 Kernen vorhanden. Danach sorgt der sog. Ellbogeneffekt für ein Absinken der Performance und die Laufzeiten werden wieder schlechter. Ebenso zeigt die Verwendung von Hyperthreading eine Verschlechterung in der Laufzeit, was zu erwarten war. Der maximale Speed-Up ist hier 6,67.

Für die AMD Ryzen Threadripper Architektur zeigt sich hingegen ein ähnliches Verhalten wie bei der Vorwärtsrechnung. Bis zu 16 Threads (genau der Anzahl von echten Kernen) zeigt sich eine gute Performance mit einem Speed-Up von bis zu 8,15 bei Verwendung von 15 Threads.

Abbildung 5.18 Untergrundmodell des Reservoirs in der Toskana. Die Farbe verdeutlicht die Permeabilitätsstartwerte für die einzelnen Schichten in logarithmischer Skalierung. Es existieren vier Probebohrungen (Gelbe Linien) (aus [83]).



Parameterfit Toskana Wir wollen nun für die Toskana-Region die gleichen Untersuchungen durchführen, die wir schon für das Perth Reservoir durchgeführt haben. Als erstes werden wir die Parameterbestimmung für den gesuchten Permeabilitätsparameter der fünften Schicht durchführen. Hierzu gibt es sieben existierende Explorationsbohrungen mit Wärmemesswerten, vier davon sind in Abbildung 5.18 zu sehen. Die Abbildung zeigt darüber hinaus einen Schnitt durch die Region, inklusive der vorhandenen Schichten, mit deren angenommenen Permeabilitätwerten in verschiedenen Farben (logarithmische Skala).

Für die Parameterstudien des Toskana-Modells wurde der SHEMAT-Code wie folgt übersetzt:

```
make library-param PROPS=hotprops USER=none omp hdf5
```

In der Tabelle 5.7 werden Messwerte für die Temperatur an diesen sieben verschiedenen Explorationsstellen gezeigt. Die Angaben der Explorationspositionen mit x, y- und z-Werten sind in Metern von dem gedachten Null-Punkt des Reservoirs zu sehen. Für das 2D-Modell werden nur die mit * gekennzeichneten Messwerte verwendet, die y-Werte werden entsprechend ignoriert.

Für das 2D-Modell ergibt sich der in Abb. 5.19 gezeigte Verlauf der Zielfunktion in den Grenzen $10^{-17}m^2 < \kappa < 10^{-11}m^2$. Zu sehen ist, dass die Zielfunktion im Bereich $6{\cdot}10^{-13}m^2$ einen minimalen Wert annimmt.

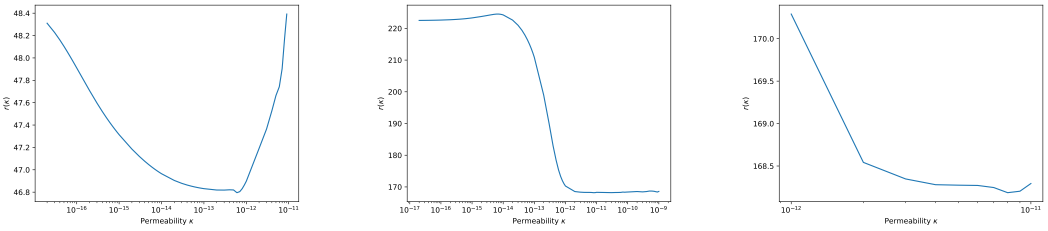
Nun werden vier verschiedene Optimierer zur Lösung der Minimierungsfunktion des 2D-Modells untersucht. Es zeigt sich, dass alle Optimierer sowohl bei Verwendung von AD, also auch bei Verwendung von FD, eine Lösung erzielen. Die Ergebnisse der einzelnen Läufe sind in den Tabellen 5.8 und 5.9 zu sehen. Die Optimierer L-BFGS-B und SLSQP zeigen hier sehr wenige Iterationen und auch insgesamt sehr wenige Auswertungen der Zielfunktion. Wieder sei darauf hingewiesen, dass bei L-BFGS-B die Anzahl der gemeldeten Funktionsauswertungen immer der Anzahl der ausgewerteten Ableitungen entspricht. Der einzige Optimierer, der aus dem lokalen Minimum bei 46,8179 herauskommt ist der TNC bei Verwendung von FD. Er erreicht den Wert 46,7952, benötigt dafür aber die meisten Ausführungen der Zielfunktion. In Abb. 5.20 ist der Verlauf der Optimie-

Tabelle 5.7 Übersicht über die Temperaturmesswerte T in °C von Explorationsbohrungen in der untersuchten Region in der Toksana, Italien. Die Angabe der Position für x, y und z ist in Metern gegeben. Die Messwerte mit * hinter der Nummer werden für das 2D-Modell verwendet.

Lokation	T	x	y	z
1	300,0	21170,0	13260,0	1665,0
2*	87,0	19260,0	10370,0	3336,0
2*	204,0	19260,0	10370,0	1805,0
2*	267,0	19260,0	10370,0	1360,0
2*	300,0	19260,0	10370,0	1045,0
2	334,0	19260,0	10370,0	125,0
2	116,0	19260,0	10370,0	2950,0
2	137,7	19260,0	10370,0	2450,0
2	184,2	19260,0	10370,0	1950,0
2	256,9	19260,0	10370,0	1450,0
2	280,2	19260,0	10370,0	1250,0
3*	102,0	17520,0	8430,0	3125,0
3*	102,0	17520,0	8430,0	3025,0
3*	100,0	17520,0	8430,0	2925,0
3	102,0	17520,0	8430,0	2825,0
3	100,0	17520,0	8430,0	2727,0
4	22,0	12466,0	1387,0	3183,0
5	33,0	7403,0	12267,0	3048,0
6	26,0	12296,0	8558,0	3290,0
7	41,0	7853,0	4533,0	3180,0

rer anhand der Anzahl der Funktionsauswertungen gezeigt. In (a) werden die Ergebnisse für AD und in (b) die Ergebnisse für FD des 2D-Modells gezeigt. Zu sehen ist, dass die Optimierer bei Verwendung von AD sehr schnell einen guten Zielfunktionswert erzielen, während sie bei FD mehr Funktionsauswertungen benötigen.

Abbildung 5.19 Überblick über die Zielfunktionen der Parameterschätzung für das 2D-Modell (a) und das 3D-Modell (b) der Toskana Region. In (c) ist der interessante Bereich des 3D-Modells zwischen $\kappa = 10^{-12}$ und $\kappa = 10^{-11}$ zu sehen.



(a) (b) (c)

Ausgehend von den hier erzielten Ergebnissen, wird jetzt das 3D Modell mit der gleichen Herangehensweise untersucht. Zuerst werfen wir in Abb. 5.19 einen Blick auf den Verlauf der Zielfunktion. Es ist zu sehen, dass es zwei lokale Minima in dieser Funktion gibt. Das Minimum um 10^{-16} ist dabei nicht zu bevorzugen, da es ein deutlich besseres Minimum zwischen 10^{-11} und 10^{-12} gibt. Dieses wurde in Abbildung (b) noch einmal genauer untersucht.

Die oben eingesetzten Optimierer sollen auch hier Anwendung finden, vgl. Abb. 5.20 (c) für AD und (d) für FD. Es zeigt sich, dass die Verwendung von FD die Optimierer L-BFGS-B und SLSQP oberhalb des Optimums, welches der Nelder-Mead-Algorithmus findet, bleiben. Bei Verwendung von AD erreicht zumindest der L-BFGS-B einen ebenbürtigen Wert. Die anderen Optimierer bre-

Tabelle 5.8 Ausführungsergebnisse verschiedener Optimierungsalgorithmen für das 2D Datenfitproblem in der Region Toskana mit AD.

Optimierer	Erfolg	$\|\kappa^*\|$	$r^*(\kappa^*)$	∇r^*	$\# r$	$\# \nabla r$	# Iter	Begründung
L-BFGS-B	0	2.552	46.818	2.9490e-09	6	N/A	5	'CONVERGENCE NORM OF PROJECTED GRADIENT <= PGTOL'
Nelder-Mead	0	2.552	46.818	N/A	76	N/A	33	'Optimization terminated successfully.'
TNC	1	2.552	46.818	-3.6653e-08	13	N/A	4	'Converged ($\|f_n - f_{n-1}\| \approx 0$)'
SLSQP	0	2.552	46.818	6.5676e-08	7	7	7	'Optimization terminated successfully.'

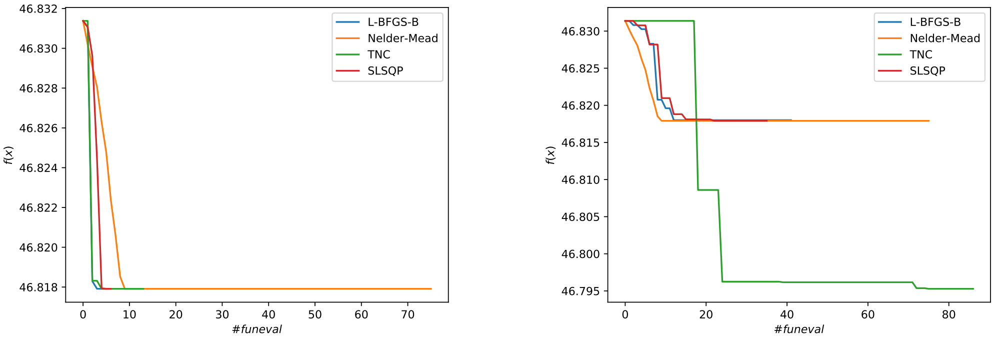
Tabelle 5.9 Ausführungsergebnisse verschiedener Optimierungsalgorithmen für das 2D Datenfitproblem in der Region Toskana mit finiten Differenzen.

Optimierer	Erfolg	$\|\kappa^*\|$	$r^*(\kappa^*)$	∇r^*	$\# r$	$\# \nabla r$	# Iter	Begründung
L-BFGS-B	0	2.717	46.818	0.0005	42	N/A	6	'CONVERGENCE REL REDUCTION OF F <= FACTR*EPSMCH'
Nelder-Mead	0	2.552	46.818	N/A	76	N/A	33	'Optimization terminated successfully.'
TNC	1	6.046	46.795	0.0022	28	N/A	4	'Converged ($\|f_n - f_{n-1}\| \approx 0$)'
SLSQP	0	2.560	46.818	0.0007	36	8	8	'Optimization terminated successfully.'

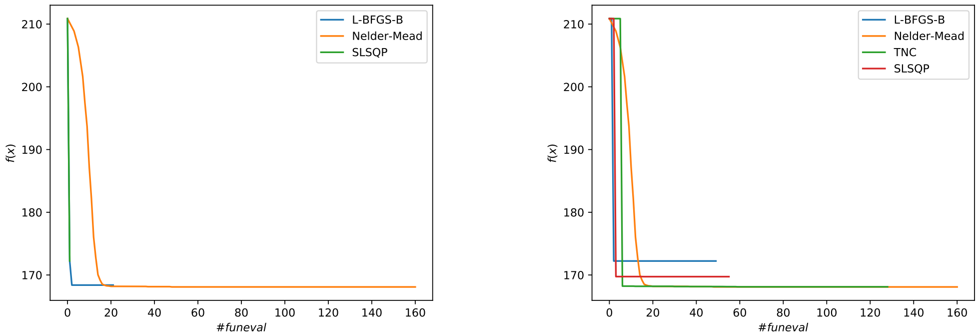
chen mit Fehlermeldungen ab.

Space-Mapping Toskana In einem weiteren Test wurde das 2D-Modell als grobes und das 3D-Modell als feines Modell für den Space-Mapping Algorithmus bestimmt. Alles andere wurde analog zu der in Abschnitt 5.3.5.1 beschriebenen Vorgehensweise durchgeführt. Es wurden für das feine Modell jedoch nur die mit * gekennzeichneten Temperaturmesswerte aus Tabelle 5.7 verwendet. In Abbildung Abb. 5.21 ist der zeitliche Verlauf des Algorithmus bei Verwendung des Optimierers L-BFGS-B für die Funktionswerte $\mathbf{f}_1$ zu sehen. Es fällt auf, dass der Algorithmus bei Verwendung von finiten Differenzen (DD) sehr schnell zu einem Ergebnis kommt. Im Gegensatz dazu benötigt der Algorithmus bei Verwendung von automatischem Differenzieren (AD) deutlich länger, kommt dafür aber auf einen besseren Funktionswert. Die Anzahl der Iterationsschritte sind bei DD zwei und bei AD 50.

Abbildung 5.20 Überblick über das Konvergenzverhalten der unterschiedlichen Optimierer für das Datenfit-Problem in der Toskana Region. (a) und (b) zeigen das Verhalten bei Verwendung des 2D-Modells, (c) und (d) das 3D-Modell.



(a) 2D Automatisches Differenzieren (b) 2D Finite Differenzen



(c) 3D Automatisches Differenzieren (d) 3D Finite Differenzen

OED Toskana Des Weiteren wurde das unter Abschnitt 5.3.5.2 gezeigte Modell analysiert. Dabei wurde ein etwas verkleinertes Modell, analog zu den Parameterstudien untersucht. Dieses ist in Abb. 5.18 gezeigt.

Um dieses Problem zu untersuchen, wurden die Funktionswerte der D-Optimalitäten in der Region für verschiedene Parameter bestimmt. Abb. 5.22 (a)–(e) zeigt die (normierten) D-Optimalitäten für κ in den Grenzen $10^{-13}m^2$ bis $10^{-17}m^2$. Da die einzelnen Funktionsauswertungen noch nicht aussagekräftig erscheinen, wurde im Weiteren die in Abbildung (f) gezeigte Auswertung durchgeführt. Hierzu wird der maximale Wert des minimalen D-Optimalitätskriteriums angegeben. Zu sehen ist, dass zwei Minima existieren (Dunkelblau). Da sich eines davon sehr stark am Rand des untersuchten Bereiches befindet, wollen wir uns auf das andere mögliche Explorationsbohrloch konzentrieren. Im überlagerten Darcy-Fluss (Pfeile) und den Bruchzonen (Linien) kann die Optimalität gut abgelesen werden. Die hohen Funktionswerte um das Minimumstal zeigen hier eine starke Tendenz, dass eine neue Probebohrung in diesen Bereichen keine signifikate Verbesserung des Modells erbringt. Ebensowenig ist es sinnvoll im nord-östlichen Bereich der Region neue Probebohrungen durchzuführen. Dies ist auch offensichtlich, da hier bereits Probebohrungen existieren.

Diese Ergebnisse wurden auf dem Ara-Cluster erzielt. Hierbei zeigten sich deutlich andere Schwerpunkte in der Ausführungszeit, als bei den Messwerten des Perth Areals. Das verwendete Modell basiert auf einer einzigen großen ASCII-Beschreibung, welche für jede von EFCOSS angeforderte Funktionsauswertung einmal eingelesen werden muss. Dies dauert auf dem Cluster 1

Abbildung 5.21 Vergleich des minimalen Funktionswertes $\mathbf{f}_1^*$ zu einer bestimmten Zeit des Aggressive-Space-Mapping Algorithmus bei Verwendung von finiten Differenzen (DD) und automatischem Differenzieren (AD) für die Parameterbestimmung Toskana Reservoirs.

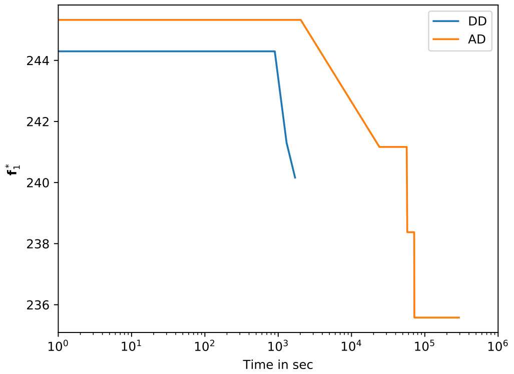
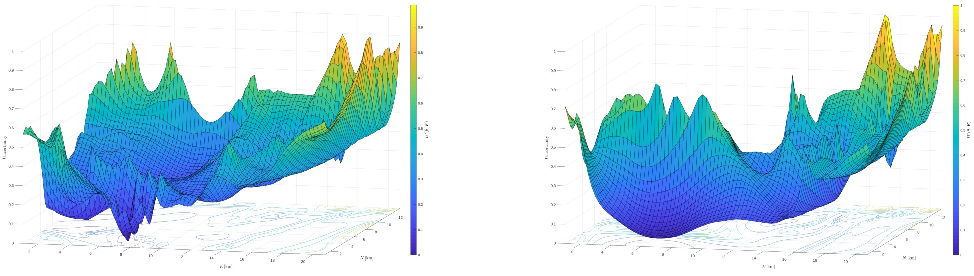
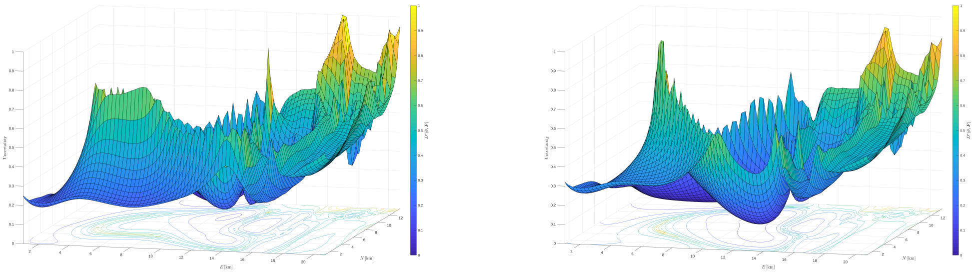


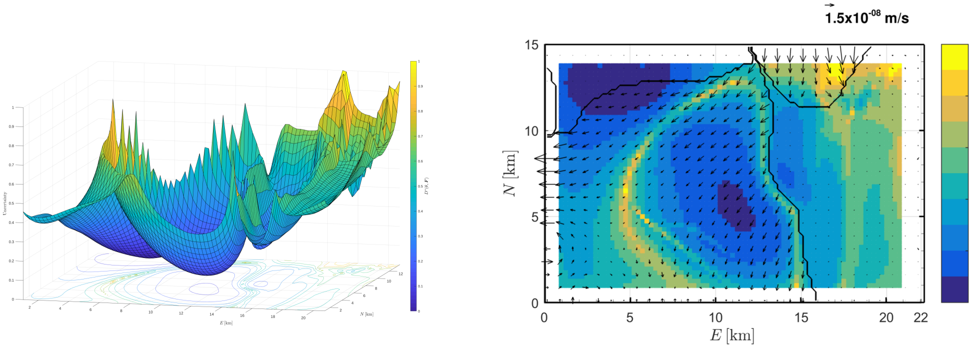
Abbildung 5.22 (a)–(e) Überblick über die D-Optimalität für den Parameter κ_5 für das OED-Problem in der Toskana-Region. (f) Ergebnisse des Optimal Experimental Designs mit dem maximalen Wert des minimalen D-Optimalitätskriteriums für das Toskana-Reservoir. Die überlagerten Pfeile geben den Darcy Fluss an, die Linien die Bruchzonen. Aus [83]



(a) 10^{-13} (b) 10^{-14}



(c) 10^{-15} (d) 10^{-16}



(e) 10^{-17} (f)

Tabelle 5.10 Überblick über die minimalen Funktionswerte der D-Optimalität für unterschiedliche κ für das flache und tiefe Bohrloch, sowie deren Abstand an der minimalen Position.

κ	$\arg\min(\psi_{D_{\text{shallow}}})$	$\arg\min(\psi_{D_{\text{deep}}})$	$\lVert \arg\min(\psi_{D_{\text{shallow}}}) - \arg\min(\psi_{D_{\text{deep}}}) \rVert$
$5 \cdot 10^{-12}$	6392.0	4484.0	1908.0
$1 \cdot 10^{-12}$	8936.0	8300.0	636.0
$5 \cdot 10^{-13}$	8512.0	8194.0	318.0
$1 \cdot 10^{-13}$	8300.0	7664.0	636.0
$5 \cdot 10^{-14}$	8194.0	7558.0	636.0
$1 \cdot 10^{-14}$	7558.0	7240.0	318.0
$5 \cdot 10^{-15}$	7558.0	7346.0	212.0
$1 \cdot 10^{-15}$	7876.0	7664.0	212.0
$5 \cdot 10^{-16}$	11904.0	11480.0	424.0
$1 \cdot 10^{-16}$	10844.0	10420.0	424.0

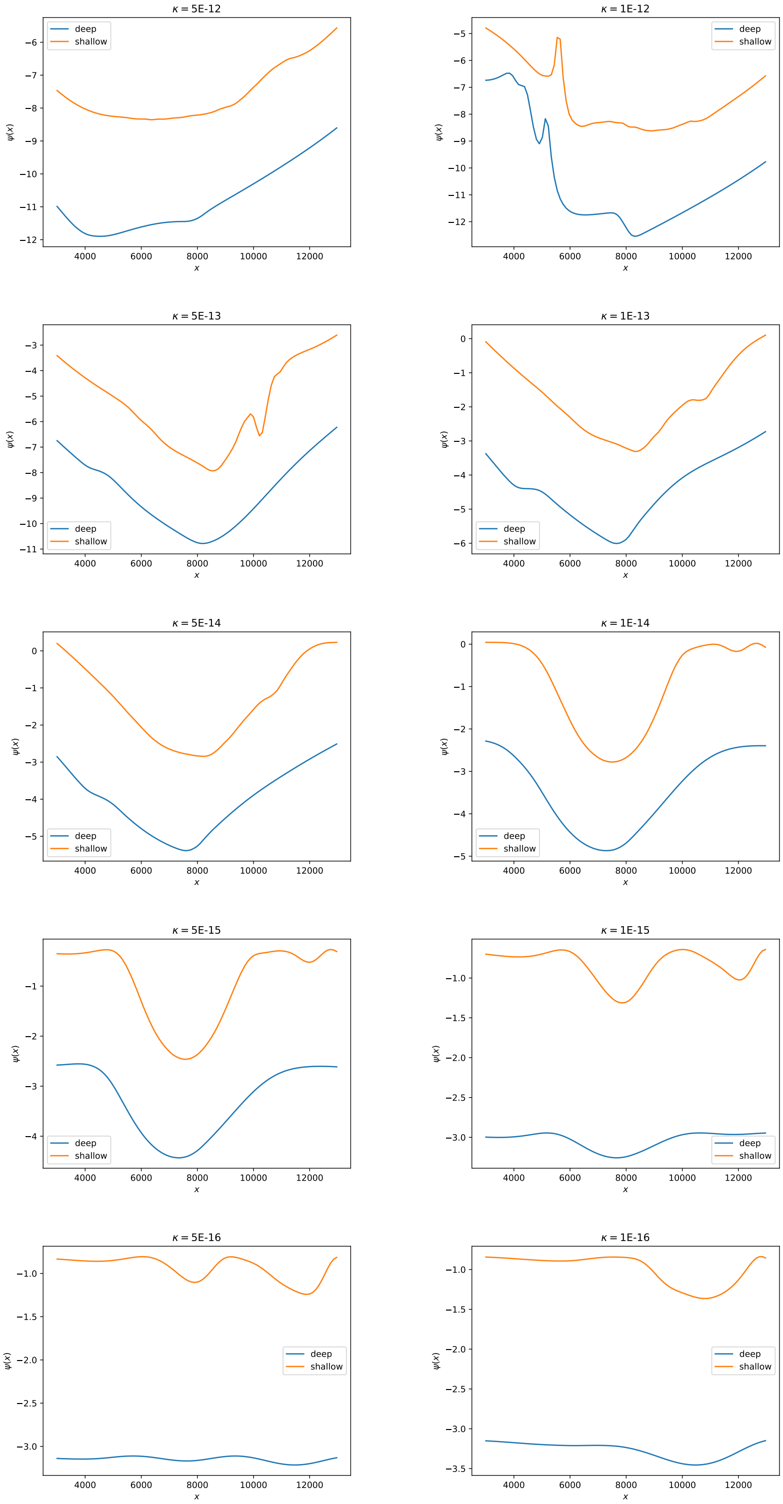
Minute und 50 Sekunden. Die eigentliche Berechnung nimmt noch einmal fünf Minuten in Anspruch. Es ergeben sich für unser Problem also die folgenden (theoretischen) seriellen Laufzeiten: Für jeden der fünf Parameterwert κ_5 muss die Zielfunktion in einem Areal von $80 \cdot 40$ Punkten ausgewertet werden.

$$T_{seriell} = 5 \cdot 80 \cdot 40 \cdot 400 \text{ Sekunden} = 1777,78 \text{ Stunden} \approx 74 \text{ Tage}$$

Eine einfache Parallelisierung, wie wir sie mit Perth durchgeführt haben, bringt hier keinen entscheidenden Vorteil. Die Zeit würde nur um den Faktor 5 sinken. Einzig die Verwendung des Speicherns der Jacobi-Matrix nach einmaliger Ausführung und Verwendung der gespeicherten Matrizen (ohne diese wieder in der SHEMAT-Simulation zu verwenden) verheißt hier minimale Ausführungszeiten von unter 10 Minuten. Dabei ist es weiter sinnvoll, die Anzahl der untersuchten Parameterwerte zu erhöhen, also nicht nur 5 zu untersuchen, sondern diese grobe Unterteilung noch einmal in 10 Werte pro 10er Potenz aufzuspalten. Dies ergibt dann 50 Parameterwerte und mit einer auf mindestens 50 erhöhten Anzahl von parallelen Prozessen, ein großes paralleles Potential.

Vergleich zwischen tiefen und flachen Bohrlöchern Eine weitere Frage, die sich bei der Betrachtung des OED-Problems stellt, ist, wie tief muss man das Probebohrloch treiben, damit die darin gemessenen Daten signifikanten Einfluss auf die Ergebnisse haben? Es wurden zwei Testreihen mit dem 2D-Modell der Toskana unternommen. Zum einen wurde ein Bohrloch mit einer Tiefe von 750 Metern untersucht, in einem weiteren Lauf wurde eine maximale Tiefe von 2250 Metern gewählt. Die Abbildung 5.23 zeigt die resultierenden Ergebnisse der Läufe. Ab einer Eingabe von $\kappa = 10^{-12}$ bis zu $\kappa = 10^{-15}$ findet sich ein Minimum der Funktion für das tiefe Loch im Bereich 7240 Meter bis 8300 Meter. Für das flachere Bohrloch finden sich die Minima im Bereich 7558 Meter bis 8936 Meter. Der minimale Abstand der optimalen Positionen findet sich bei $\kappa = 10^{-15}$ und $\kappa = 5 \cdot 10^{-15}$ mit je 212 Metern Abstand, gefolgt von 318 Metern bei den Parametern $\kappa = 5 \cdot 10^{-13}$ und $\kappa = 10^{-14}$. Sofern diese Fehlertoleranz akzeptabel erscheint, macht es durchaus Sinn, ein flacheres Bohrloch zu untersuchen, da damit hohe Kosten eingespart werden können und dennoch eine ausreichende Aussagefähigkeit über die Güte der Parameter erreicht wird.

Abbildung 5.23 Vergleich der D-Optimalitäts-Zielfunktionen für flache (shallow) und tiefe (deep) Bohrlöcher für verschiedene Parameterwerte κ.



5.4 Modellidentifikation

In Zusammenarbeit mit dem Institut für Werkstoffwissenschaften der RWTH Aachen wurden die Überlegungen der theoretischen Modelle in einem Exzellenzcluster mit Produktionscode verbunden. Dabei entstand sowohl ein Fortran-Code, welcher mit Hilfe von Adifor abgeleitet wurde, als auch Matlab-Code welcher die Implementierungen der Modellidentifikation beinhaltet und über ein Mex-Interface an den Fortran-Code gekoppelt ist [11]. An dieser Stelle wollen wir nun diese Arbeiten ebenfalls in EFCOSS nachvollziehen und erweitern. Zunächst testen wir die in der Arbeit genutzten synthetischen Modelle und in einem weiteren Schritt den Fortran-Code, der mit Hilfe von Tapenade abgeleitet wurde und mittels F2PY an EFCOSS angebunden wird.

5.4.1 Synthetische Modelle

Die hier betrachteten Modelle haben alle eine Struktur der Form

$$A(\Theta)\mathbf{y} = \mathbf{F}. \tag{5.1}$$

Für die unterschiedlichen Modelle wird im Folgenden zwischen verschiedenen Matrizen $A_{model}(\Theta)$ unterschieden. Dabei wird die Matrix A_{model} durch den Einheitsvektor $\mathbf{e}_{model}$ erzeugt, das heißt an der Stelle *model* wird der Wert in $\mathbf{e}$ zu 1, sonst ist er 0. Somit ist die Matrix A gegeben durch

$$A_{model}(\Theta) = I + \Theta \cdot \mathbf{e}_{model}\mathbf{e}_{model}^T \tag{5.2}$$

Lineare Modelle Für lineare Modelle wird die Funktion

$$\mathbf{y} = A_{model}^{-1}(\Theta)\mathbf{F} \tag{5.3}$$

gelöst. Deren Ableitung nach Θ ist gegeben durch

$$\frac{d\mathbf{y}}{d\Theta} = -A_{model}^{-1}(\Theta)\mathbf{e}_{model}\mathbf{e}_{model}^T\mathbf{y}. \tag{5.4}$$

Die Ableitung nach $\mathbf{F}$ ist

$$\frac{d\mathbf{y}}{d\mathbf{F}} = A_{model}^{-1}. \tag{5.5}$$

Wir können nun einfache Modelle definieren, die als Simulationsklassen in unserem Sinne genutzt werden. Den Anfang macht die Klasse `linearModel_theta` aus Quellcode 5.24, welche das lineare Modell mit der Ableitung nach Θ beschreibt. Die Klasse `linearModel_force` ist eine abgeleitete Klasse von `linearModel_theta`, vgl. Quellcode 5.25. Sie überlädt die Methode `Jac` für die Ableitung nach $\mathbf{F}$ ihrer Basisklasse. Beide Klassen müssen für die Verwendung in EFCOSS in jeweils einer eigenen Datei mit dem Namen `linearModel_theta.py` und `linearModel_force.py` gespeichert werden.

Quellcode 5.24: Simulationsklasse für das lineare Modell mit der Ableitung nach θ.

```
from numpy import *
from numpy.linalg import *

class linearModel_theta:
    def getInput(self,xi,xd):
```

```
        modelno=int(xi[0])
        numforces=int(xi[1])
        theta=xd[0]
        force=xd[1:]
        n=4
        e=zeros((n,1))
        e[modelno]=1
        A=diag(ones((n)))+theta*e*e.T
        return [modelno,numforces,theta,force,e,A]

    def Func(self,xi,xd,dOutputVector):
        [modelno,numforces,theta,force,e,A]=self.getInput(xi,xd)
        y=solve(A,force)
        dOutputVector[0:numforces]=y

    def Jac(self,xi,xd,seed,dOutput,dJacobian):
        [modelno,numforces,theta,force,e,A]=self.getInput(xi,xd)
        y=solve(A,force)
        g_y=solve(A,-e*e.T*y)
        dOutput[0:numforces]=y
        dJacobian[0:numforces][0:len(theta)]=g_y

func=linearModel_theta().Func
jacobian=linearModel_theta().Jac
```

Quellcode 5.25: Ableitung des linearen Modells nach der Kraft F

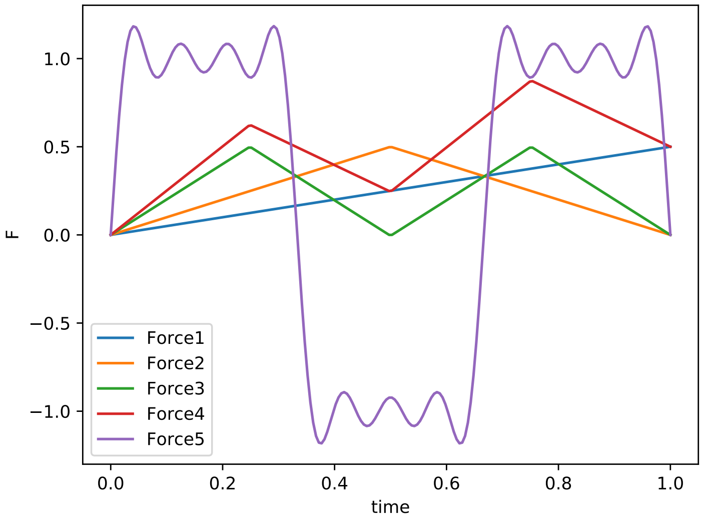
```
#from numpy import *
from linearModel_theta import *
from numpy.linalg import *

class linearModel_force(linearModel_theta):
    def Jac(self,xi,xd,dSeed,dOutput,dJacobian):
       [modelno,numforces,theta,force,e,A]=self.getInput(xi,xd)
       y=numpy.linalg.solve(A,force)
       g_y=numpy.linalg.inv(A)
       dOutput[0:numforces]=y
       dJacobian[0:numforces][0:len(force)]=g_y

func=linearModel_force().Func
jacobian=linearModel_force().Jac
```

5.4.2 Metallplastizität

Anders als die synthetischen Modelle wurde für die Metallplastizität ein auf Fortran basierender Code genutzt. Als Grundlage diente der Code des Institutes für angewandte Mechanik der RWTH Aachen. Dieser gliedert sich in zwei Fortran-Dateien, `materialInputDataLoop.f` für die Ausführung und Steuerung und `plasfinkin.f` für die eigentliche Simulation. Dieser Code wurde mit Hilfe von Tapenade abgeleitet, mit gfortran übersetzt und anschließend mit F2PY in ein von Python aus einbindbares Shared-Object umgewandelt.

Der Programmcode hat als Haupt-Funktion die Subroutine `materialInputDataLoop` mit den in Quellcode 5.26 gezeigten Eingabewerten, welche in einer For-Schleife die Subroutine `plasfinkin` aufruft. Die Anzahl der Aufrufe hängt dabei von dem Übergabeparameter `rawNumberTmp` ab, der

Abbildung 5.24 Überblick über die in der Metallplastizität genutzten Kräfte.



Anzahl der zu simulierenden Zeitschritte. Als Eingabe wurden fünf verschiedene Kräfte festgelegt, welche in Abb. 5.24 zu sehen sind.

Quellcode 5.26: Einsprungpunkt in die Subroutine für die Berechnung der Metallplastizität.

```
subroutine materialInputDataLoop(schalterTmp,matParam,inputDataTmp,force,rawNumberTmp,sigVZeit,epsVZeit)
```

Ableitung und Compilierung

Um den gegebenen Fortran-Code in Python nutzbar zu machen, wurde ein Makefile geschrieben, welches in Quellcode 5.27 zu sehen ist. Dabei ist zu beachten, dass für die zwei Parameter `matParam` und `force` jeweils eigene Ableitungen erzeugt werden müssen, was sich im Code mit dem Parameter `INDEP1=matParam` sowie `INDEP2=force` widerspiegelt. Die Ableitungen werden jeweils in ein eigenes Verzeichnis, `theta/` und `force/` geschrieben. Zum Compilieren der Interfaces nutzen wir die vorher beschriebene Methode zur Erstellung von gepackten Bibliotheken mit den Namen `libmaterialInputDataLoop_theta.a` und `libmaterialInputDataLoop_force.a`, welche dann in je einem weiteren Make Target mit F2PY in die Shared-Objekte `materialInputDataLoop_theta.so` und `materialInputDataLoop_force.so` umgewandelt werden und so von EFCOSS verwendet werden können.

Quellcode 5.27: Makefile für die Erzeugung von Ableitungsobjekten und die Compilierung der Metallplastizitätsmodelle für die Modellidentifikation.

```
TOPLEVEL=materialInputDataLoop
SRC=$(TOPLEVEL).f plasfinkin.f
INDEP1=matParam
INDEP2=force
DEP1=sigVZeit
DEP2=sigVZeit epsVZeit
OPT=-O3
```

```
tap1:
   @-mkdir theta
   tapenade -d -multi -head "$(TOPLEVEL)($(INDEP1))\($(DEP1))" $(SRC) -outputlanguage fortran95 -O theta
         -output $(TOPLEVEL)

tap2:
   @-mkdir force
   tapenade -d -multi -head "$(TOPLEVEL)($(INDEP2))\($(DEP2))" $(SRC) -outputlanguage fortran95 -O force
         -output $(TOPLEVEL)

modelLib1: $(SRC) theta/diffsizes.f theta/$(TOPLEVEL)_dv.f95
   gfortran $(OPT) -c -fPIC $(SRC) theta/diffsizes.f
   gfortran $(OPT) -c -fPIC theta/$(TOPLEVEL)_dv.f95 -o $(TOPLEVEL)_theta_dv.o
   ar -r lib$(TOPLEVEL)_theta.a *.o

modelLib2: $(SRC) force/diffsizes.f force/$(TOPLEVEL)_dv.f95
   gfortran $(OPT) -c -fPIC $(SRC) force/diffsizes.f
   gfortran $(OPT) -c -fPIC force/$(TOPLEVEL)_dv.f95 -o $(TOPLEVEL)_force_dv.o
   ar -r lib$(TOPLEVEL)_force.a *.o

lib$(TOPLEVEL)_theta.so: modelLib1
   f2py -c -m $(TOPLEVEL)_theta $(TOPLEVEL)_thetaIF.f90 -L. -l$(TOPLEVEL)_theta -lgomp

lib$(TOPLEVEL)_force.so: modelLib2
   f2py -c -m $(TOPLEVEL)_force $(TOPLEVEL)_forceIF.f90 -L. -l$(TOPLEVEL)_force -lgomp
```

5.4.3 Anbindung der Modelle an EFCOSS

Um den beschriebenen Code mit EFCOSS zu benutzen, wurde die Klasse `PlasticityModel`, abgeleitet von `Model` entworfen. Der Konstruktor nimmt eine EFCOSS Instanz, sowie die Modellinstanz und einen Pointer auf den Parametervektor `theta` und die Kräftevektoren `forces`. Weitere optionale Parameter sind die Anzahl der aktiven Parameter in `numPar`, sowie untere und obere Grenzen in `lb` und `ub`. Als Methoden bietet es `getFun` zur Berechnung der Funktionswerte und `getJac` zur Berechnung der Ableitung.

Quellcode 5.28: Klasse für ein Modell in der Metallplastizität.

```
class PlasticityModel(Model):
    # @efcoss reference to efcoss
    # @modelInstance the used model
    # @theta a pointer to the efcoss variable of theta
    # @forces a pointer to the efcoss variable for forces
    # @numPar the number of active parameters
    def __init__(self, efcoss, modelInstance, theta, forces, numPar=1, lb=None, ub=None):
        print "Creating new plasticity model ", modelInstance, " with ", numPar, " parameters"
        self.numPar = numPar
        self.efcoss = efcoss
        self.forces = forces
        self.theta = theta
        self.instance = modelInstance
        self.m = self.efcoss.getResultVec().size
        self.x0 = zeros(len(self.efcoss.getOptVec()))
        self.n = self.numPar
        self.lb = lb
```

```
        self.ub = ub

    def getFun(self, x0, force, objective=0):
        self.forces[0:len(force)] = force
        self.x0[0:self.n] = x0
        res = self.ffun(self.x0)
        return res

    def getJac(self, x0, force, objective=0):
        if (objective == 0):
            self.forces[0:len(force)] = force
            self.x0[0:self.n] = x0
            return self.fjac(self.x0)
        else:
            self.theta[0:len(x0)] = x0
            n = len(force)
            m = self.m * n
            # print m,n
            return self.efcoss.evalfjac(n, force, m, self.instance, objective)
```

Dieses PlasticityModel wird weiter in der `ModelIdentification` Klasse für die Modelle verwendet, welche in Quellcode 5.29 zu sehen sind.

Quellcode 5.29: Initialisierungsroutine für die Modellidentifikationsklasse.

```
class ModelIdentification(ProblemDefinition):
    def __init__(self, efcoss, lb, ub, theta, active, forces, inputData, numData=1, errFactor=0.0,
                 optimizer=ScipyOptimizer(method=’L-BFGS-B’, options={’disp’: True}), trueModel=None):
        self.efcoss = efcoss
        self.numModels = self.efcoss.simulationinstances
        self.model = []
        print "Model identification initialization for ", self.numModels, " models (", theta, forces,
             active, lb, ub, ")"
        for i in range(0, self.numModels):
            self.model.append(PlasticityModel(efcoss, i, theta[i], forces, active[i], lb[i], ub[i]))
        self.numData = numData
        self.errorFactor = errFactor
        self.forces = inputData
        self.trueModel = trueModel
        self.lb = lb
        self.ub = ub
        self.optimizer = optimizer
```

Um nun die Modelle entsprechend auszuführen, haben wir die Klasse `MaterialModels` entworfen, welche in Quellcode 5.30 zu sehen ist. Sie dient als Einsprungpunkt für die Modellidentifikation, der mit den synthetischen, sowie mit dem Metallplastizitätscode Modellen umgehen kann. In der Initialisierungsroutine wird ein Name des Problems (welcher ebenfalls für den Namen der zu erzeugenden Interfaceroutinen gilt), die Anzahl der Modelle `nummodels` und die Anzahl der Kräfte `numforces` benötigt.

Die Klasse bietet weiterhin die Methode `initEFCOSS` zum Bekanntmachen der Ein- und Ausgabe des Simulationscodes in EFCOSS und zum Setzen der entsprechenden Ableitungsvariablen Θ und $\mathbf{F}$, sowie des “wahren Modells”. Weiterhin sind die Methoden `generateSimulationInterface`, zum Erzeugen der Fortran-Simulationsinterfaces `*_thetaIF.f90` und `*_force.f90`, `initSim`, zum Initialisieren der Simulationsroutinen, und `initOpt`, zum Initialisieren der Optimierer, wichtig. Für letztere wurde sich für die Verwendung der Vektorzielfunktion `Identity` für beide Optimierungsprobleme entschieden.

Quellcode 5.30: Klasse zur Verarbeitung der Materialmodelle.

```
class MaterialModels(ProblemDefinition):
    def __init__(self, toplevel="simple", nummodels=4, numforces=10, numtimesteps=4,useBuffer=False,
         matFree=False):
        ProblemDefinition.__init__(self, toplevel, nummodels, 2, 0, useBuffer=useBuffer,matfree=matFree)
        self.toplevel = self.name
        self.nummodels = nummodels
        self.numforces = numforces
        self.numtimesteps = numtimesteps
        self.initInput()

    def initEFCOSS(self, theta, trueModel=2):
        self.model = [self.efcoss.newInputVariable("model" + str(i), i) for i in range(0, self.nummodels
             )]
        self.timesteps = self.efcoss.newInputVariable("timesteps", self.numtimesteps)
        self.thetaInput = [self.efcoss.newInputVariable("theta"+str(i), theta[i]) for i in range(0, self
             .nummodels)]
        self.forceInput = self.efcoss.newInputVariable("force", self.forces[0].tolist())
        self.y = self.efcoss.newOutputVariable("y", self.timesteps)
        for i in range(0, self.nummodels):
            self.efcoss._set_SimulationCallingSequence(
                [self.model[i], self.thetaInput[i], self.forceInput, self.timesteps, self.y], i)

        self.efcoss._set_OptVars([self.thetaInput[2]], 0)
        self.efcoss._set_OptVars([self.forceInput], 1)
        self.setTrueModel(trueModel, theta)

    def setTrueModel(self, trueModel, theta):
        self.trueModel = TrueModel(trueModel, theta[trueModel])

    def generateSimulationInterface(self):
        ProblemDefinition.generateSimulationInterfaceFortran(toplevel=self.toplevel,
                                                             ifname=self.toplevel + "_thetaIF.f90",
                                                                  adtool="tapenade",
                                                             adlevel=1, adreverse=True, instance=0)
        ProblemDefinition.generateSimulationInterfaceFortran(toplevel=self.toplevel,
                                                             ifname=self.toplevel + "_forceIF.f90",
                                                                  adtool="tapenade",
                                                             adlevel=1, adreverse=True, instance=1)

    def initOpt(self):
        obj_identity = self.efcoss.setObjectiveFunction("VectorObjective", "Identity",0)
        obj_parameter= self.efcoss.setObjectiveFunction("VectorObjective","Identity",1)

    def initSim(self):
        sim_theta = Simulation(self.efcoss, self.toplevel + "_theta", instance=0, objective=0)
        sim_forces = Simulation(self.efcoss, self.toplevel + "_force", instance=1, objective=1)
        self.efcoss.setSimulationServer(sim_theta, objective=0)
        self.efcoss.setSimulationServer(sim_forces, objective=1)

    def run(self, lb, ub):
        model = ModelIdentification(self.efcoss, lb, ub, self.thetaInput, self.active, self.forceInput,
```

```
             self.forces,
                                    len(self.forces), trueModel=self.trueModel)
        model.run()

    def initInput(self):
        ...
```

5.4.4 Resultate

Lineare Modelle Um die linearen Modelle auszuführen, nutzen wir das in Quellcode 5.31 gezeigte Laufzeitscript. Zu sehen ist, dass zuerst die Startwerte für Θ definiert und untere und obere Grenzen gesetzt werden. Dann wird eine Instanz der Klasse `MaterialModels` erzeugt, welche den Namen `"linearModel"` übergeben bekommt. Dann muss die Verwendung von EFCOSS vorbereitet und die Simulation und Optimierer initialisiert werden. Für die Simulation werden die in Quellcode 5.24 und 5.25 gezeigten Klassen verwendet. Die Ausführung der Modellidentifikation geschieht dann über die `run` Routine, welche als Eingaben die unteren und oberen Grenzen erhält.

Quellcode 5.31: Ausführungsscript für die Modellidentifikation mit linearen Modellen.

```
import MaterialModels

theta=[[0.1],[0.1],[0.5],[0.1],[0.1]]
lb=[[0.1] for i in range(0,len(theta))]
ub=[[0.9] for i in range(0,len(theta))]
model=MaterialModels.MaterialModels("linearModel")
model.initEFCOSS(theta)
model.initSim()
model.initOpt()
model.run(lb,ub)
```

Die Ergebnisse für das lineare Modell sind in Quellcode 5.32 zu sehen. Für die Indizes 0 und 2 werden erfüllte Regeln gemeldet, d.h. die Modelle 0 und 2 sind den anderen Modellen zum Lösen des Problems vorzuziehen.

Quellcode 5.32: Ergebnis des Modellidentifikation für das Lineare Test-Modell.

```
Main Step 3
==================
muij[ 0 , [1, 2, 3] ]= [1.97498166e-12 1.97498166e-12 9.87489003e-13]
muij[ [1, 2, 3] , 0 ]= [0.00000000e+00 0.00000000e+00 9.87489003e-13]
Index  0  satisfies rule
muij[ 1 , [0, 2, 3] ]= [0.00000000e+00 5.55555556e-02 9.87489003e-13]
muij[ [0, 2, 3] , 1 ]= [1.97498166e-12 0.00000000e+00 9.87489003e-13]
Index  1  does not satisfy rule
muij[ 2 , [0, 1, 3] ]= [0. 0. 0.]
muij[ [0, 1, 3] , 2 ]= [1.97498166e-12 5.55555556e-02 5.55555556e-02]
Index  2  satisfies rule
muij[ 3 , [0, 1, 2] ]= [9.87489003e-13 9.87489003e-13 5.55555556e-02]
muij[ [0, 1, 2] , 3 ]= [9.87489003e-13 9.87489003e-13 0.00000000e+00]
Index  3  does not satisfy rule
```

Metallplastizität Die Klasse `MaterialModels` wurde für die Verwendung des Fortran-Codes `materialInputDataLoop.f` in der Klasse `MetalPlasticity` aus Quellcode 5.33 überladen. Damit wird in `initEFCOSS` die entsprechende Signatur der Routine `materialInputDataLoop` umgesetzt.

Quellcode 5.33: Die Klasse MetallPlasticity zur Verwendung des Metallplastizitätscodes mit der Modellidentifikation.

```
class MetalPlasticity(MaterialModels):
    def initEFCOSS(self,theta,trueModel=2):
        self.model=[self.efcoss.newInputVariable("model"+str(i),i+1) for i in range(0,self.nummodels)]
        self.thetaInput=[self.efcoss.newInputVariable("matparam",theta[i]) for i in range(0,self.
             nummodels)]
        self.inputData=self.efcoss.newInputVariable("inputData",self.input)
        self.forceInput=self.efcoss.newInputVariable("force",self.forces[0])
        self.timesteps=self.efcoss.newInputVariable("timesteps",self.numtimesteps)
        self.sigvzeit=self.efcoss.newOutputVariable("sigVZeit",self.timesteps)
        self.epsvzeit=self.efcoss.newOutputVariable("epsVZeit",self.timesteps)
        self.setTrueModel(trueModel,theta)

        for i in range(0,self.nummodels):
            self.efcoss._set_SimulationCallingSequence([self.model[i],self.thetaInput[i],self.inputData,
                 self.forceInput,self.timesteps,self.sigvzeit,self.epsvzeit],i)

        self.efcoss._set_OptVars([self.thetaInput[2]],0)
        self.efcoss._set_OptVars([self.forceInput],1)

    def initInput(self):
        ...
```

Wir nutzen das in Quellcode 5.34 gezeigte Ausführungsscript für die Modellidentifikation mit den Metallplastizitätsmodellen. Die Startwerte der Optimierungsprobleme der einzelnen Modelle sind in `theta` gespeichert, das heißt, sie starten jeweils bei den unteren Grenzen des Optimierungsproblems. Diese sowie die oberen Grenzen der Optimierung sind in `lb` und `ub` gespeichert. Danach kann die `MetallPlasticity` Instanz erstellt, EFCOSS initialisiert (`initEFCOSS`) und, falls noch nicht geschehen, die Interfaces mittels `generateInterfaces` erstellt werden. Diese müssen vor der weiteren Verwendung mit dem in Quellcode 5.27 gezeigten Makefile noch compiliert werden. Danach können die Interfaces entsprechend in `initSim` verwendet werden.

Quellcode 5.34: Ausführungsscript für die Metallplastizität.

```
import metalPlasticity

theta=[[1.0,100.0,0.0,0.0],[1.0,2500.0,0.0,0.0],[1.0,100.0,1.0,2500.0]]
lb=[[1.0,100.0],[1.0,2500.0],[1.0,100.0,1.0,2500.0]]
ub=[[30.0,300.],[30.0,2900.0],[30.0, 300.0, 30.0, 3000.0]]

mat=metalPlasticity.MetalPlasticity("materialInputDataLoop",len(theta),5,200)
mat.initEFCOSS(theta)
#mat.generateInterfaces()
mat.initSim()
mat.initOpt()
mat.run(lb,ub)
```

Für die Metallplastizität wurde die Berechnung entsprechend durchgeführt. Dabei wurden die in Quellcode 5.35 gezeigten Ergebnisse erzielt, welche im Folgenden weiter untersucht werden sollen. Zu sehen ist, dass das Ergebnis den Erwartungen entspricht. Die Modelle 1 und 2 (Index 0 und 1) sind dem Modell 3 in jeder Hinsicht unterlegen.

Quellcode 5.35: Ergebnis des Modellidentifikation für die Metallplastizität.

```
Main Step 3
==================
muij[ 0 , [1, 2] ]= [1278140.79187871 1278140.79187871]
muij[ [1, 2] , 0 ]= [2.06487548e+06 4.77939796e-10]
Index  0  does not satisfy rule
muij[ 1 , [0, 2] ]= [2064875.48454938 2064875.48454938]
muij[ [0, 2] , 1 ]= [1.27814079e+06 4.77939796e-10]
Index  1  does not satisfy rule
muij[ 2 , [0, 1] ]= [4.77939796e-10 4.77939796e-10]
muij[ [0, 1] , 2 ]= [1278140.79187871 2064875.48454938]
Index  2  satisfies rule
```

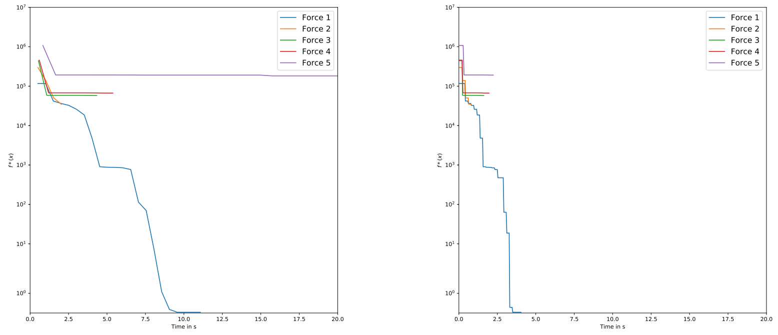
Schauen wir uns nun den Verlauf der Zielfunktionale für den Schritt eins an. Es müssen die folgenden beiden Optimierungsprobleme gelöst werden:

1. $\min_{\Theta_i,\Theta_j}(\|\mathbf{y}_i(\Theta_i,\mathbf{f}_l)-\mathbf{y}_j(\Theta_j,\mathbf{f}_l)\|)$
2. $\max_{\Theta_i,\Theta_j}(\max(\|\frac{\partial\mathbf{y}_i(\Theta_i,\mathbf{f}_l)}{\partial\mathbf{f}_l}\|,\|\frac{\partial\mathbf{y}_j(\theta_j,\mathbf{f}_l)}{\partial\mathbf{f}_l}\|))$

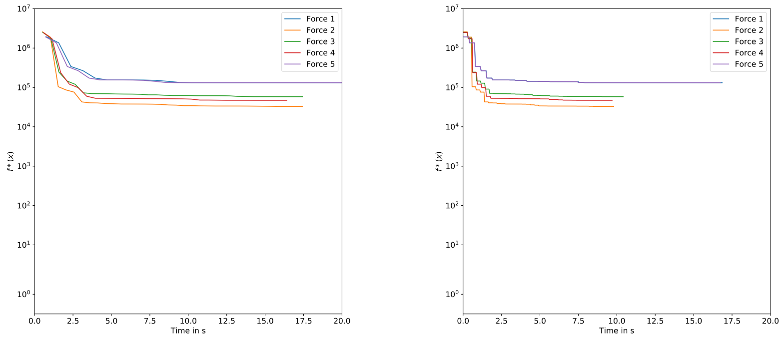
Die Ergebnisse sind in den Abbildungen 5.25 bei Verwendung von automatischem Differenzieren (links) bzw. bei Verwendung von finiten Differenzen (rechts) zu sehen. Als Optimierer wurde jeweils der L-BFGS-B verwendet. Weitere Tests mit anderen geeigneten Optimierern sind im Anhang A.1 bis A.4 zu sehen. Allen Optimierern ist gemein, dass die Anzahl der ausgewerteten Zielfunktionen bei Verwendung von AD deutlich sinkt, jedoch die Zeit insgesamt steigt. Hierfür ist der verwendete Black-Box Einsatz von Tapenade verantwortlich. Dies könnte durch eine geschickte Verwendung von Hand abgeleiteter Routinen verbessert werden.

In Abb. 5.26 und 5.27 sind die Laufzeiten der einzelnen Optimierungsprobleme im Vergleich verschiedener Optimierer zu sehen. Es fällt auf, dass der Optimierer TNC die insgesamt schlechteste Leistung bei Berechnung der ersten Zielfunktion (die linke Spalte der Abbildungen, verdeutlicht mit Θ) zeigt, L-BFGS-B befindet sich von der Leistung nahe an SLSQP, welcher die beste Leistung zeigt. Ein anderes Bild zeigt die Optimierung der zweiten Zielfunktion (rechte Spalte, $\mathbf{F}$). Hier ist der TNC zumeist den anderen getesteten Optimierern vorzuziehen, wenn finite Differenzen verwendet werden. Bei Verwendung des automatischen Differenzierens ist für die Kräfte zwei bis vier der Optimierer L-BFGS-B schneller, für die anderen Kräfte (eins und fünf) kann kein einheitlicher Sieger gefunden werden.

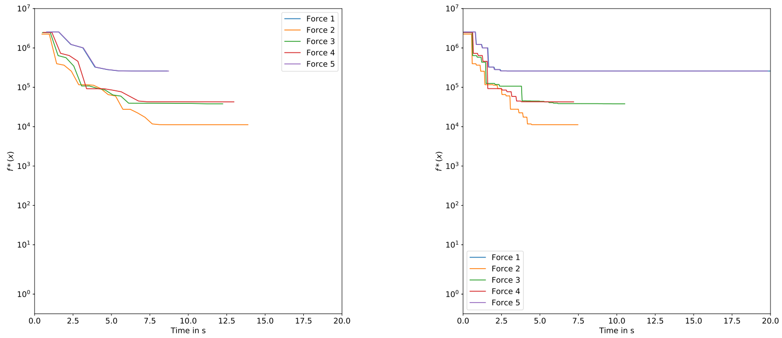
Abbildung 5.25 Zeitlicher Verlauf der Optimierung der Zielfunktion $\min_{\Theta_i,\Theta_j}(\|\mathbf{y}_i(\Theta_i,\mathbf{f}_l) - \mathbf{y}_j(\Theta_j,\mathbf{f}_l)\|)$ im ersten Schritt der Modellidentifikation bei Verwendung von automatischem Differenzieren (AD) und finiten Differenzen (FD) mit dem L-BFGS-B Optimierer.



(a) Modell 1,2 AD (b) Modell 1,2 FD



(c) Modell 1,3 AD (d) Modell 1,3 FD



(e) Modell 2,3 AD (f) Modell 1,3 FD

Abbildung 5.26 Laufzeit der Minimierungsprobleme der Modellidentifikation im ersten Schritt für verschiedene Optimierer bei Verwendung von AD.

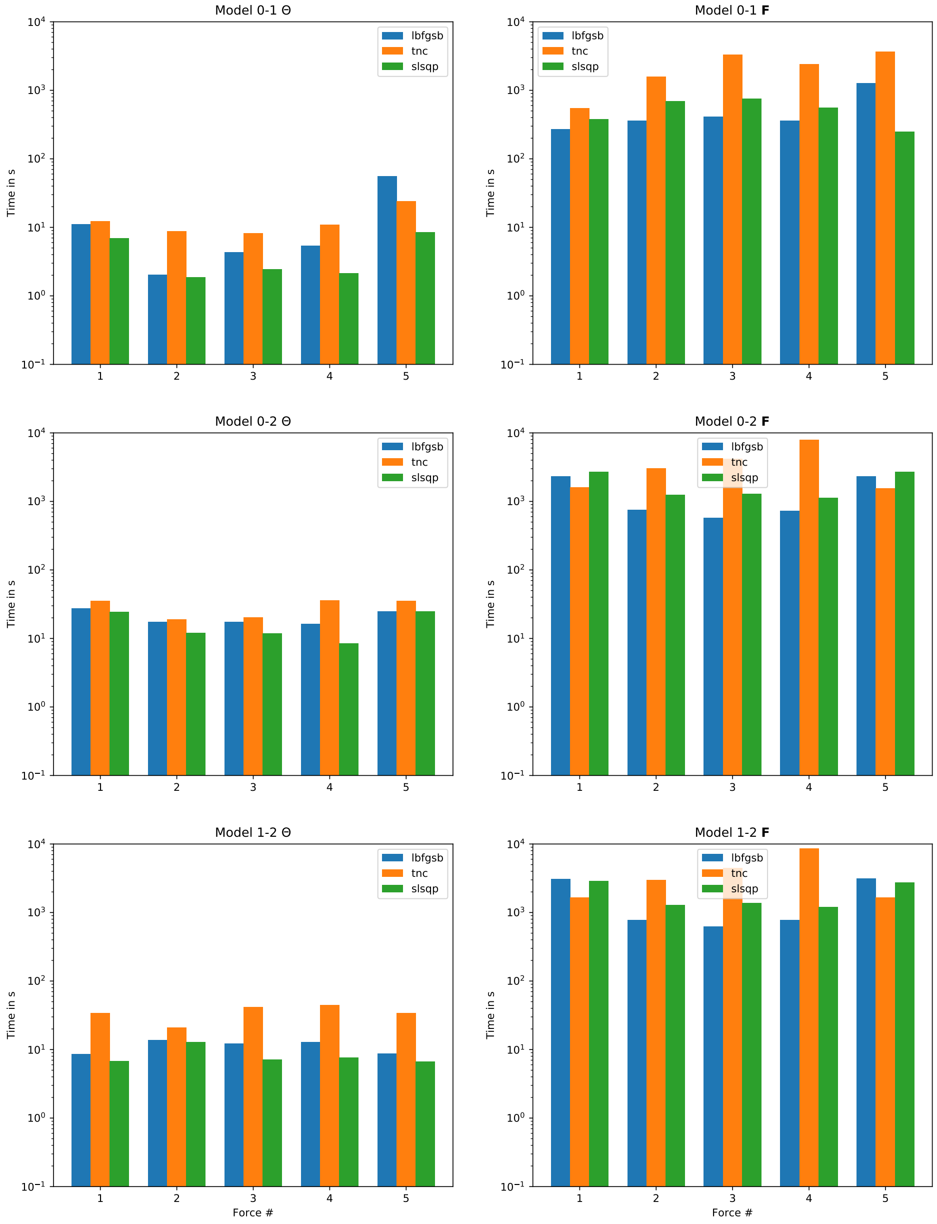
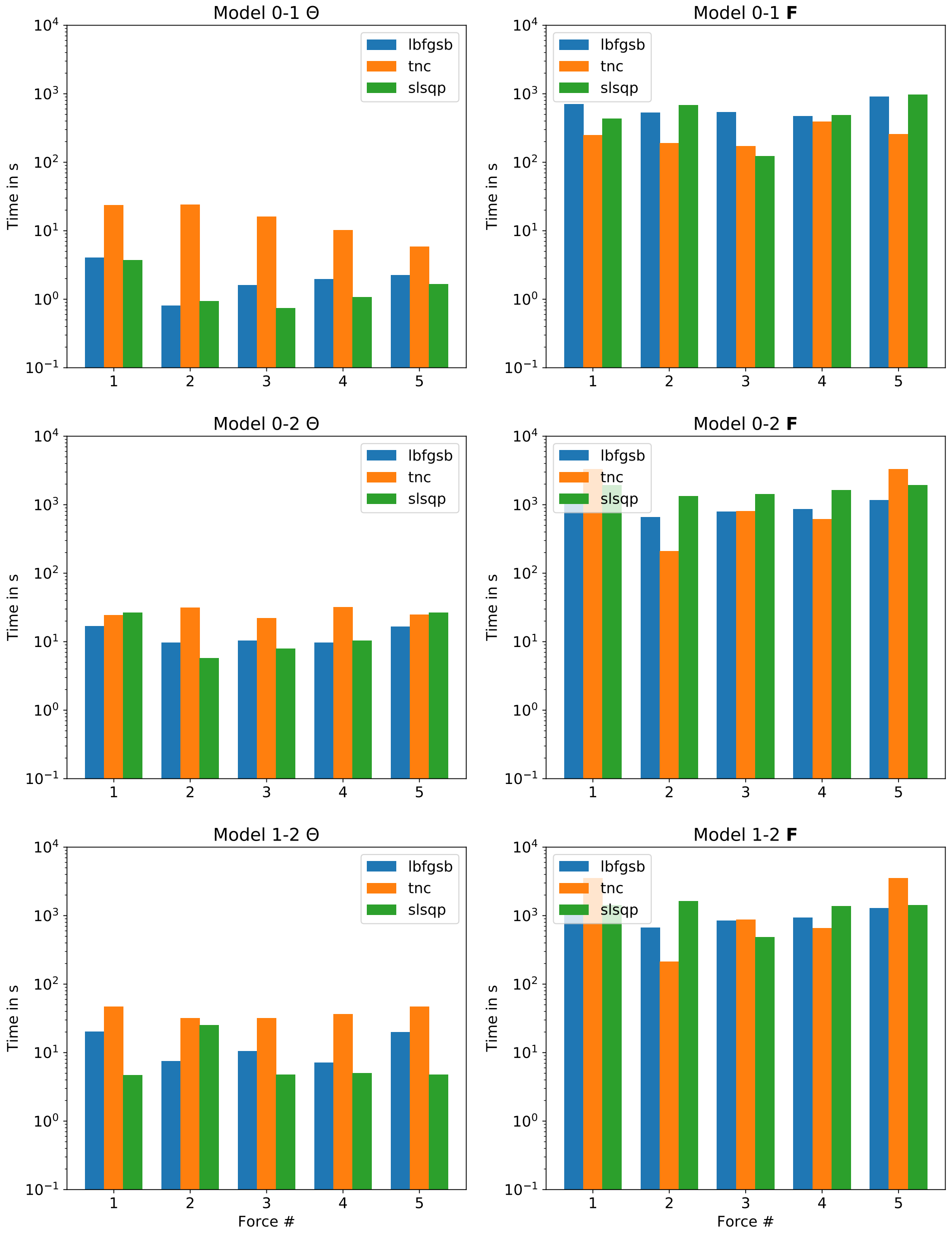


Abbildung 5.27 Laufzeit der Minimierungsprobleme der Modellidentifikation im ersten Schritt für verschiedene Optimierer bei Verwendung von finiten Differenzen.



Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Kopplung von Simulationswerkzeugen mit Optimierungspaketen mit Hilfe des EFCOSS Frameworks gezeigt. Dieses Framework wurde zum Zweck der besseren Nutzbarkeit und der Erweiterung um parallele Funktionen mit OpenMP und MPI, sowie dem Einsatz des Rückwärtsmodus des Automatischen Differenzierens grundlegend überarbeitet. Damit konnten neue Problemstellungen aus den Bereichen Parameterschätzung, Space-Mapping, optimale Versuchsplanung (OED) und Modellidentifikation gelöst werden.

Es wurden neue programmiersprachenunabhängige Interfacegenerierungsroutinen erstellt und mit diesen die Interfaces zu den hier getesteten Simulationen generiert. Mit diesen ist es möglich, neben den Interfaces zu den Routinen zur Berechnung der vollen Jacobi-Matrix, auch Interfaces für die Berechnung von (transponierten) Jacobi-Matrix-Vektor-Operationen durchzuführen. Damit ist es möglich, die Struktur der inversen Probleme deutlich besser auszunutzen und somit die Ausführungszeiten zu reduzieren.

Als Beispiel dient die geothermische Nutzung und Exploration von Reservoiren mit Untergrundmodellen aus der Toskana. Hier wurde durch den Einsatz von Parameterschätzung und Optimaler Versuchsplanung eine Risikoabschätzung durchgeführt und neue Explorationsbohrlochpositionen gefunden. Durch den geschickten Einsatz von automatischem Differenzieren, sowohl im Vorwärts-, wie auch im Rückwärtsmodus, Parallelisierung und Wiederverwendung von bereits berechneten Ergebnissen, konnten die seriellen Ausführungszeiten zur Lösung der gegebenen Probleme von mehreren Tagen beziehungsweise Wochen auf wenige Minuten gesenkt werden.

Durch die Veränderung der internen Struktur und die Verwendung von Standard Python Paketen ist es nun möglich, mehrere Optimierungsprobleme, Zielfunktionen und Simulationsroutinen innerhalb einer Aufgabenstellung zu lösen. Damit kann zum Beispiel das Space-Mapping in EFCOSS implementiert werden, welches durch den Einsatz von gröberen Modellen die Suche nach optimalen Parametern für ein gegebenes feines Problem beschleunigen kann. Dies wurde anhand des Modells eines Reservoirs in der Region um Perth in Australien angewendet. Das grobe Modell ist dabei um den Faktor 40 kleiner als das feine Modell. Es zeigt sich, dass durch den Einsatz von Space-Mapping hierbei eine annähernd gute Genauigkeit erreicht wird, das grobe Modell jedoch deutlich weniger oft ausgeführt werden muss als bei der direkten Optimierung. Außerdem benötigt man keine Ableitungsinformationen des feinen Modells.

Weiterhin ist durch den Einsatz von mehreren Simulationsmodellen und Optimierungsroutinen die Lösung von Modellidentifikationsproblemen mit EFCOSS möglich. Dies wurde an einem Beispiel aus der Metallplastizität angewendet, bei dem drei verschiedene Modelle miteinander verglichen wurden. Dazu musste die Antwort jedes der Modelle auf unterschiedliche Eingabekräfte ausgewertet und der Abstand der Antworten untereinander betrachtet werden. Es zeigt sich, dass durch den Einsatz von automatischem Differenzieren die Anzahl der Optimierungsschritte gesenkt werden konnte, die Gesamtausführungszeit jedoch teilweise höher ist. Dies ist dem Umstand geschuldet, dass der verwendete Fortran-Code mit einem Black-Box-Ansatz abgeleitet wurde. Hier wäre in einer weitergehenden Arbeit der Einsatz von Hand abgeleiteten Routinen vorteilhaft.

Die in dieser Arbeit gezeigten Anwendungen können als Grundlage für weiterführende Untersuchungen dienen, zum Beispiel für nicht in dieser Arbeit angesprochene Modelle. Ein weiterer interessanter Aspekt ist der Einsatz der Modellidentifikation im Geothermieumfeld. So können die gezeigten Ansätze für die Metallplastizität relativ einfach auf die Geothermie angewendet werden, indem zum Beispiel die Permeabilitäten der Schichten als ein Parameter und die Randwerte als andere Parameter untersucht werden.

Mit der neuen Struktur und den hinzugewonnenen Möglichkeiten können einige neue Techniken eingesetzt werden. Ausgehend von den gezeigten Umstrukturierungen zur Verwendung von Jacobi-Matrix-Vektor Produkten für die Optimierung kann durch die Verwendung von Färbungsalgorithmen die Anzahl der Spalten, für den Vorwärtsmodus des automatischen Differenzierens bzw. die Zeilen für den Rückwärtsmodus einer dünnbesetzten Jacobi-Matrix, die Laufzeit entsprechend gesenkt werden. Dies könnte weiter sogar automatisiert werden, was ein großes Potential für EFCOSS und dessen Einsatz mit automatisch generierten Ableitungen bedeuten würde.

Kapitel 7

Literaturverzeichnis

[1] Agenda21. Kohle, Braunkohle, Steinkohle. http://www.agenda21-treffpunkt.de/lexikon/Kohle.htm.

[2] Bundesumweltministerium. Erneuerbare Energien — Innovationen für die Zukunft. Brochure of the Federal Ministry for the Environment, Nature Conservation and Nuclear Safety, Germany, 2004. In German.

[3] J. Niederau, A. Ebigbo, G. Marquart, J. Arnold, and C. Clauser. On the impact of spatially heterogenous permeability on free convection in the Perth Basin, Australia. *Geothermics*, 66:119 – 133, 2017.

[4] A. Rasch and H. M. Bücker. EFCOSS: An interactive environment facilitating optimal experimental design. *ACM Transactions on Mathematical Software*, 37(2):13:1–13:37, 2010.

[5] J. Bartels, M. Kühn, and C. Clauser. Numerical Simulation of Reactive Flow using SHEMAT. In C. Clauser, editor, *Numerical Simulation of Reactive Flow in Hot Aquifers*, pages 5–74. Springer Berlin Heidelberg, 2003.

[6] J. Keller, V. Rath, J. Bruckmann, D. Mottaghy, C. Clauser, A. Wolf, R. Seidler, H. M. Bücker, and N. Klitzsch. SHEMAT-Suite: an open-source code for simulating flow, heat and species transport in porous media. *Accepted for publication at SoftwareX*, 12:100533, 2020.

[7] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer, New York, 2nd edition, 2006.

[8] B. M. Averick, R. G. Carter, J. J. Moré, and G.-L. Xue. The MINPACK-2 Test Problem Collection. Technical Report MCS–P153–0692, Argonne National Laboratory, June 1992.

[9] A. Wolf. *Ein Softwarekonzept zur hierarchischen Parallelisierung von stochastischen und deterministischen Inversionsproblemen auf modernen ccNUMA-Plattformen unter Nutzung automatischer Programmtransformation.* Dissertation, Department of Computer Science, RWTH Aachen University, 2011.

[10] I. N. Vladimirov, M. P. Pietryga, and S. Reese. On the modelling of non-linear kinematic hardening at finite strains with application to springback—Comparison of time integration algorithms. *International Journal for Numerical Methods in Engineering*, 75(1):1–28, 2008.

[11] M. Bambach, H. M. Bücker, S. Heppner, M. Herty, and I. N. Vladimirov. Characteristics of testing conditions for constitutive models in metal plasticity. *Journal of Engineering Mathematics*, 88(1):99–119, 2014.

[12] A. Tarantola. *Inverse Problem Theory and Methods for Model Parameter Estimation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

[13] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, New York, 2nd edition, 1987.

[14] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, 1983.

[15] K. Smith. On the standard deviations of adjusted and interpolated values of an observed polynomial function and its constants and the guidance they give towards a proper choice of the distribution of observations. *Biometrika*, 12:1–85, 1918.

[16] A. C. Atkinson and A. N. Donev. *Optimum Experimental Designs.* Oxford science publications. Clarendon Press, 1992.

[17] Jörg Schaber. Parameter estimation and model discrimination. 2008.

[18] D. Zwilling, editor. *CRC Standard Mathematical Tables and Formulas.* Chapman and Hall/-CRC, 33rd edition, 2018.

[19] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

[20] M. Bambach, M. Heinkenschloss, and M. Herty. A method for model identification and parameter estimation. *Inverse Problems*, 29(2):025009, 2013.

[21] J. W. Bandler, R. M. Biernacki, S. H. Chen, P. A. Grobelny, and R. H. Hemmers. Space mapping technique for electromagnetic optimization. *IEEE Transactions on Microwave Theory and Techniques*, 42(12):2536–2544, 1994.

[22] M. H. Bakr, J. W. Bandler, R. M. Biernacki, and S. H. Chen. Design of a three-section 3: 1 microstrip transformer using aggressive space mapping. *Simulation Optimization Syst. Res. Lab., Dept. Elect. Comput. Eng., McMaster Univ., Hamilton, ON, Canada, Rep. SOS-97-1-R*, 1997.

[23] M. H. Bakr, J. W. Bandler, K. Madsen, and J. Søndergaard. Review of the Space Mapping Approach to Engineering Optimization and Modeling. *Optimization and Engineering*, 1(3):241–276, 2000.

[24] T. D. Robinson, M. S. Eldred, K. E. Willcox, and R. Haimes. Surrogate-Based Optimization Using Multifidelity Models with Variable Parameterization and Corrected Space Mapping. *AIAA Journal*, 46(11):2814–2822, 2008.

[25] T. Jansson, L. Nilsson, and M. Redhe. Using surrogate models and response surfaces in structural optimization - With application to crashworthiness design and sheet metal forming. *Structural and Multidisciplinary Optimization*, 25:129–140, 07 2003.

[26] A. Borzi and V. Schulz. *Computational Optimization of Systems Governed by Partial Differential Equations.* Computational Science and Engineering. Society for Industrial and Applied Mathematics, 2011.

[27] A. Conn, N. Gould, and P. Toint. *Trust Region Methods.* Society for Industrial and Applied Mathematics, 2000.

[28] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, New York, 1981.

[29] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2013. http://www.scipy.org.

[30] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[31] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer, Berlin, 1981.

[32] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[33] H. Heuser. *Lehrbuch der Analysis Teil 2.* Lehrbuch der Analysis. Viehweg+Teubner Verlag, 2008.

[34] H. M. Bücker, M. A. Rostami, and M. Lülfesmann. An interactive educational module illustrating sparse matrix compression via graph coloring. In *2013 International Conference on Interactive Collaborative Learning (ICL)*, pages 330–335, 2013.

[35] H. Bastani and L. Guerrieri. On the Application of Automatic Differentiation to the Likelihood Function for Dynamic General Equilibrium Models. In C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 303–313, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[36] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

[37] H. M. Bücker, P. Hovland, C. Wente, and H. Bach. Community Portal for Automatic Differentiation. http://www.autodiff.org.

[38] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* Number 10 in Scientific Computation Series. MIT Press, 2008.

[39] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface.* Number 1 in Scientific and engineering computation. MIT Press, 1999.

[40] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0, 2013. http://www.openmp.org.

[41] Python Software Foundation. Python Multiprocessing - Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html, 2020.

[42] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference.* MIT Press, Cambridge, MA, USA, 1995.

[43] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core.* MIT Press, Cambridge, MA, USA, 2nd edition, 1998.

[44] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI–The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, USA, 1998.

[45] L. Dalcín, R. Paz, and M. Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[46] A. Rasch. *Efficient Computation of Derivatives for Optimal Experimental Design.* Dissertation, Department of Computer Science, RWTH Aachen University, 2012.

[47] T. Munson, J. Sarich, S. Wild, S. Benson, and L. Curfman McInnes. TAO 2.0 Users Manual. Technical Report ANL/MCS–TM–322, Mathematics and Computer Science Division, Argonne National Laboratory, 2012. http://www.mcs.anl.gov/tao.

[48] S. J. Benson, L. Curfman McInnes, and J. J. Moré. A Case Study in the Performance and Scalability of Optimization Algorithms. *ACM Transactions on Mathematical Software*, 27(3):361–376, 2001.

[49] J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. Curfman McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. L. Windus. Component-Based Integration of Chemistry and Optimization Software. *Journal of Computational Chemistry*, 25(14):1717–1725, 2004.

[50] S. Körkel. *Numerische Methoden für Optimale Versuchsplanungsprobleme bei nichtlinearen DAE-Modellen.* PhD thesis, University of Heidelberg, Germany, 2002.

[51] Object Management Group. Common Object Request Broker Architecture (CORBA): Specification, Version 3.3, November 2012. http://www.omg.org/spec/CORBA/3.3.

[52] C. H. Bischof, H. M. Bücker, B. Lang, and A. Rasch. Solving large-scale optimization problems with EFCOSS. *Advances in Engineering Software*, 34(10):633–639, 2003.

[53] A. Rasch, H. M. Bücker, and A. Bardow. Software supporting optimal experimental design: A case study of binary diffusion experiments using EFCOSS. *Computers & Chemical Engineering*, 33(4):838–849, 2009.

[54] H. M. Bücker, O. Fortmeier, and M. Petera. Solving a parameter estimation problem in a three-dimensional conical tube on a parallel and distributed software infrastructure. *Journal of Computational Science*, 2(2):95–104, 2011.

[55] M. Henning. The rise and fall of CORBA. *Communications of the ACM*, 51(8):52–57, 2008.

[56] P. Peterson. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, 2009.

[57] I. de Jong. Pyro – Python Remote Objects, 2013. http://pythonhosted.org/Pyro4.

[58] I. de Jong. Pyro Documentation – Configuring Pyro, 2013. https://pythonhosted.org/Pyro4/config.html.

[59] I. de Jong. Pyro Documentation – Security, 2013. http://pythonhosted.org/Pyro4/security.html.

[60] T. E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[61] Making Numpy ndarrays hashable. http://machineawakening.blogspot.de/2011/03/making-numpy-ndarrays-hashable.html.

[62] H. P. Langtangen. *Python Scripting for Computational Science*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[63] D. M. Beazley. SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org/.

[64] B. Scott. CXX. http://cxx.sourceforge.net/.

[65] David A. and Stefan S. Boost.Python. https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html.

[66] G. McMillan. Simplified CXX. https://github.com/scipy/weave/tree/master/weave/scxx.

[67] C. Doutriaux, P. F. Dubous, and M. de Hoon. Python-Fortran. https://sourceforge.net/projects/pyfortran/.

[68] E. Hauck. Implementierung eines Interfacegenerators für die Verbindung von Optimierungs- und Simulationssoftware. Bachelorarbeit, Friedrich-Schiller-Universität Jena, Germany, 2017.

[69] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 5 2006.

[70] E. Xu. PyIpopt auf Github. https://github.com/xuy/pyipopt, 2018.

[71] H. M. Bücker, M. A. Rostami, and R. Seidler. Turning a Serial Forward Code into a Parallel Inverse Code: A Case Study from Geothermal Engineering. In K. Psarris, P. Borne, I. Rudas, and Y. S. Shmaliy, editors, *Computers, Automatic Control, Signal Processing and Systems Science, Proceedings of the 2014 International Conference on System, Control, Signal Processing and Informatics II (SCSI'14), Prague, Czech Republic, April 2–4, 2014*, volume 33 of *Recent Advances in Electrical Engineering Series*, pages 22–25. EUROPMENT, 2014.

[72] The hdf5 library and file format. https://www.hdfgroup.org/solutions/hdf5/.

[73] R. Seidler, H. M. Bücker, K. Padalkina, M. Herty, J. Niederau, G. Marquart, and A. Rasch. Redesigning the EFCOSS Framework Toward Finding Optimally Located Boreholes in Geothermal Engineering. In I. Horvát and Z. Rusák, editors, *Proceedings of the tenth international symposium on tools and methods of competitive engineering (TMCE 2014), May 19–23, 2014, Budapest, Hungary*, pages 831–842, 2014.

[74] H. M. Bücker. Verbundprojekt MeProRisk II: Optimierungsstrategien und Risikoanalyse für tiefe geothermische Reservoire - eine Machbarkeitsstudie : Teilprojekt F: Paralleles Rechnen und optimale Versuchsplanung : Abschlussbericht : Laufzeit: 1. September 2012 - 31. August 2015. Technical report, Friedrich-Schiller-Universität Jena, Lehrstuhl für Advanced Computing, Jena, 2016.

[75] B. Della Vedova, C. Vecellio, S. Bellani, and U. Tinivella. Thermal modelling of the Larderello geothermal field (Tuscany, Italy). *International Journal of Earth Sciences*, 97(2):317–332, 2008.

[76] W. Wagner, J. R. Cooper, A. Dittmann, J. Kijima, H.-J. Kretzschmar, A. Kruse, R. Mares, K. Oguchi, H. Sato, I. Stocker, et al. The IAPWS industrial formulation 1997 for the thermodynamic properties of water and steam. *Journal of Engineering for Gas Turbines and Power*, 122(1):150–184, 2000.

[77] A. Ebigbo, J. Niederau, G. Marquart, I. Dini, M. Thorwart, W. Rabbel, R. Pechnig, R. Bertani, and C. Clauser. Influence of depth, temperature, and structure of a crustal heat source on the geothermal reservoirs of Tuscany: numerical modelling and sensitivity study. *Geothermal Energy*, 4(1):1, 2016.

[78] F. Batini, A. Brogi, A. Lazzarotto, D. Liotta, and E. Pandeli. Geological features of Larderello-Travale and Mt. Amiata geothermal areas (southern Tuscany, Italy). *Episodes*, 26(3):239–244, 2003.

[79] A. Brogi. Neogene extension in the Northern Apennines (Italy): insights from the southern part of the Mt. Amiata geothermal area. *Geodinamica Acta*, 19(1):33–50, 2006.

[80] A. Brogi. The Triassic and Palaeozoic successions drilled in the Bagnor geothermal field and Poggio Nibbio area (Monte Amiata, Northern Apennines, Italy). *Italian Journal of Geoscience*, 127(3):599–613, 2008.

[81] A. Brogi. Kinematics and geometry of Miocene low-angle detachments and exhumation of the metamorphic units in the hinterland of the Northern Apennines (Italy). *Journal of Structural Geology*, 30:2–20, January 2008.

[82] A. Brogi. The structure of the Monte Amiata volcano-geothermal area (Northern Apennines, Italy): Neogene-Quaternary compression versus extension. *International Journal of Earth Sciences*, 97:677–703, July 2008.

[83] R. Seidler, K. Padalkina, H. M. Bücker, A. Ebigbo, M. Herty, G. Marquart, and J. Niederau. Optimal experimental design for reservoir property estimates in geothermal exploration. *Computational Geosciences*, 2(2):357–383, 2016.

[84] R. Bertani, G. Bertini, G. Cappetti, A. Fiordelisi, and B. M. Marocco. An update of the Larderello-Travale/Radicondoli deep geothermal system. In *Proceedings*, pages 24–29. Citeseer, 2005.

[85] G. Bertini, M. Casini, G. Gianelli, and E. Pandeli. Geological structure of a long-living geothermal system, Larderello, Italy. *Terra Nova*, 18(3):163–169, 2006.

[86] P. Claps, P. Giordano, and G. Laguardia. Spatial distribution of the average air temperatures in Italy: quantitative analysis. *Journal of hydrologic engineering*, 13(4):242–249, 2008.

Anhang A

Anhang

A.1 Parallelisierte Ableitungen

Quellcode A.1: MPI Parallelisierung des AD-Codes aus Quellcode 3.4.

```
 SUBROUTINE MPISIM_DV(x1, x1d, x2, x2d, x3, x3d, x4, x4d, x5, x5d, fvec, &
& fvecd, m, nbdirs)
  USE MPI
  USE DIFFSIZES
!  Hint: nbdirsmax should be the maximum number of differentiation directions
  IMPLICIT NONE
  INTEGER :: m, local_m, prank, psize
  DOUBLE PRECISION :: x1, x2, x3, x4, x5, fvec(m)
  DOUBLE PRECISION :: x1d(nbdirsmax), x2d(nbdirsmax), x3d(nbdirsmax), &
& x4d(nbdirsmax), x5d(nbdirsmax), fvecd(nbdirsmax, m)
  DOUBLE PRECISION, ALLOCATABLE :: local(:)
  DOUBLE PRECISION, ALLOCATABLE :: locald(:, :)
  INTEGER :: i, j
  DOUBLE PRECISION :: temp, temp1, temp2
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: temp1d, temp2d
  REAL :: t1, t2
  INTEGER :: ierr
  INTEGER :: k
  INTRINSIC DBLE
  INTRINSIC EXP
  DOUBLE PRECISION :: arg1
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: arg1d
  INTEGER :: nd
  INTEGER :: nbdirs
  CALL MPI_COMM_RANK(mpi_comm_world, prank, ierr)
  CALL MPI_COMM_SIZE(mpi_comm_world, psize, ierr)
  local_m = m/psize
  ALLOCATE(locald(nbdirs, local_m))
  ALLOCATE(local(local_m))
  DO i=1,local_m
    k = i + prank*local_m
    temp = DBLE(10*(k-1))
    arg1 = -(x4*temp)
    DO nd=1,nbdirs
      arg1d(nd) = -(temp*x4d(nd))
      temp1d(nd) = arg1d(nd)*EXP(arg1)
      arg1d(nd) = -(temp*x5d(nd))
    END DO
    temp1 = EXP(arg1)
    arg1 = -(x5*temp)
    temp2 = EXP(arg1)
    DO nd=1,nbdirs
      temp2d(nd) = arg1d(nd)*EXP(arg1)
      locald(nd, i) = x1d(nd) + x2d(nd)*temp1 + x2*temp1d(nd) + x3d(nd)*&
&       temp2 + x3*temp2d(nd)
    END DO
    local(i) = x1 + x2*temp1 + x3*temp2
  END DO
  CALL MPI_GATHER(local, local_m, mpi_double_precision, fvec, local_m, &
&           mpi_double_precision, mpi_in_place, mpi_comm_world, ierr)
  do i=1,nbdirs
  CALL MPI_GATHER(locald(i,:), local_m, mpi_double_precision, fvecd(i,:), local_m, &
```

```
&           mpi_double_precision, mpi_in_place, mpi_comm_world, ierr)
  end do

END SUBROUTINE MPISIM_DV
```

A.2 EFCOSS Codegenerator

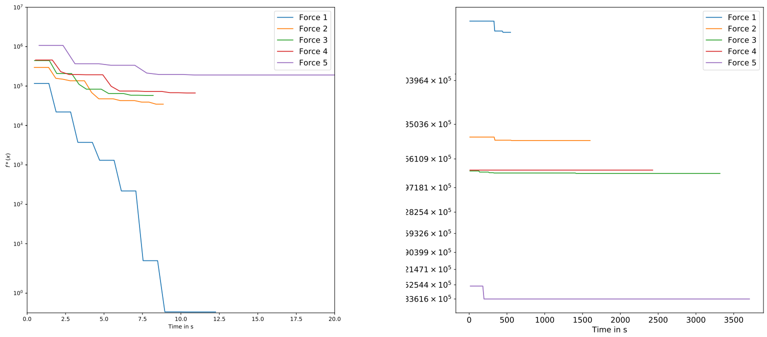
Quellcode A.2: Das generierte Interface für die Funktion zur Berechnung von $J^T v' = u$ für das Datafit Beispiel.

```
 subroutine JacobianTVector(iInputVector, dInputVector, dInputSeed, dOutputVector, dJacobiVector)
   implicit none

   integer, dimension(:), intent(in) :: iInputVector
   double precision, dimension(:), intent(in) :: dInputVector
   double precision, dimension(:), intent(in) :: dInputSeed
   double precision, dimension(:), intent(inout) :: dOutputVector
   double precision, dimension(:), intent(inout) :: dJacobiVector
   double precision :: x1
   double precision :: b_x1
   double precision :: x2
   double precision :: b_x2
   double precision :: x3
   double precision :: b_x3
   double precision :: x4
   double precision :: b_x4
   double precision :: x5
   double precision :: b_x5
   double precision, dimension(:), allocatable :: fvec
   double precision, dimension(:), allocatable :: b_fvec
   integer :: nf
   integer :: ctr_0
   integer :: ctr_1

   nf = iInputVector(1)
   allocate(fvec(nf))
   x1 = dInputVector(1)
   x2 = dInputVector(2)
   x3 = dInputVector(3)
   x4 = dInputVector(4)
   x5 = dInputVector(5)

   allocate(b_fvec(nf))

   ctr_0 = 1

   b_x1 = 0.0
   b_x2 = 0.0
   b_x3 = 0.0
   b_x4 = 0.0
   b_x5 = 0.0
   b_fvec(:) = 0.0
   do ctr_1 = 1,nf
      b_fvec(ctr_1) = dInputSeed(ctr_0)
```

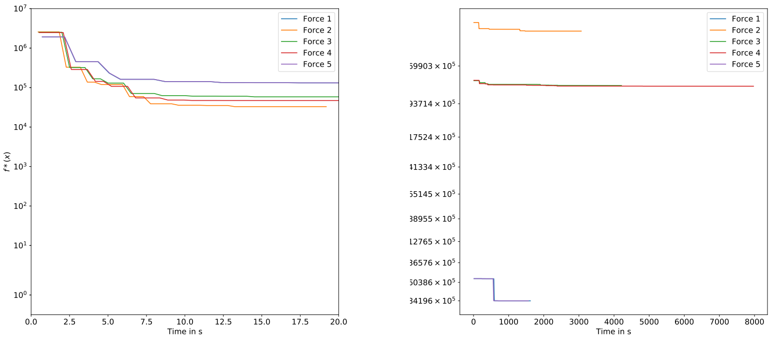
```
      ctr_0 = ctr_0 + 1
   end do

   call datafit_b(x1,b_x1,x2,b_x2,x3,b_x3,x4,b_x4,x5,b_x5,fvec,b_fvec,nf)

   ctr_0 = 1

   do ctr_1 = 1,nf
      dOutputVector(ctr_0) = fvec(ctr_1)
      ctr_0 = ctr_0 + 1
   end do

   ctr_0 = 1

   dJacobiVector(:) = 0.0
   dJacobiVector(ctr_0) = b_x1
   ctr_0 = ctr_0 + 1
   dJacobiVector(ctr_0) = b_x2
   ctr_0 = ctr_0 + 1
   dJacobiVector(ctr_0) = b_x3
   ctr_0 = ctr_0 + 1
   dJacobiVector(ctr_0) = b_x4
   ctr_0 = ctr_0 + 1
   dJacobiVector(ctr_0) = b_x5
   ctr_0 = ctr_0 + 1

   deallocate(fvec)

   deallocate(b_fvec)

end subroutine
```

A.3 Modellidentifikation

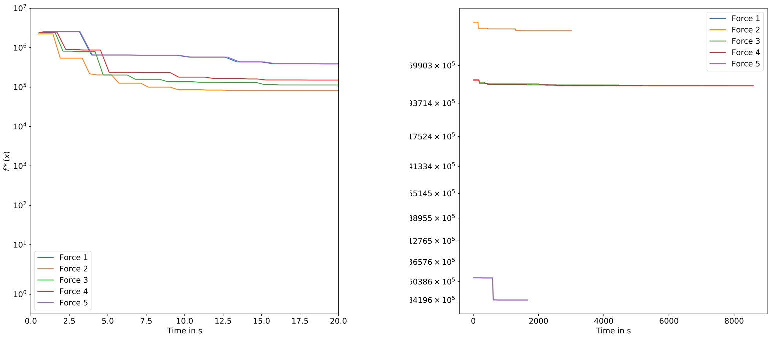
Abbildung A.1 Verlauf der Zielfunktionale der Optimierungsalgorithmen im ersten Schritt der Modellidentifikation bei Verwendung von AD mit dem Optimierer TNC.



(a) Modell 1,2 Θ (b) Modell 1,2 Kraft

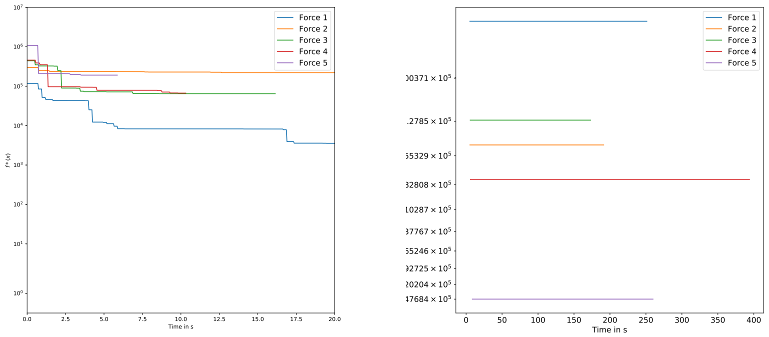


(c) Modell 1,3 Θ (d) Modell 1,3 Kraft

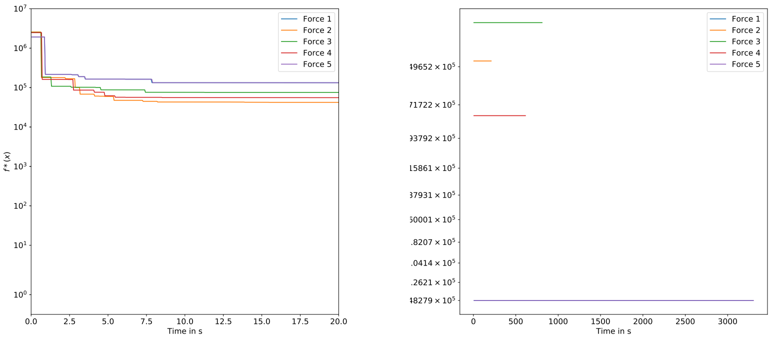


(e) Modell 2,3 Θ (f) Modell 1,3 Kraft

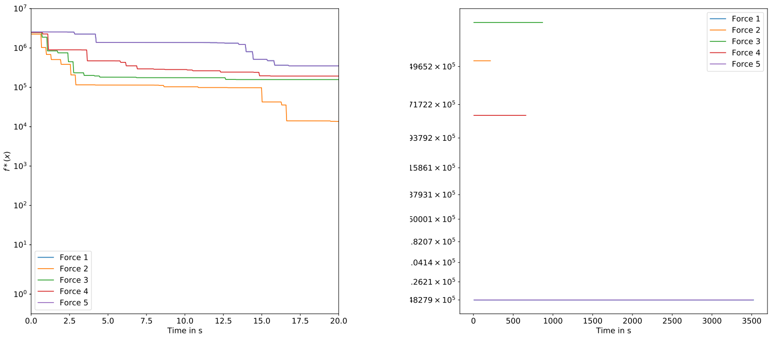
Abbildung A.2 Verlauf der Zielfunktionale der Optimierungsalgorithmen im ersten Schritt der Modellidentifikation bei Verwendung von finiten Differenzen mit dem Optimierer TNC.



(a) Modell 1,2 Θ (b) Modell 1,2 Kraft

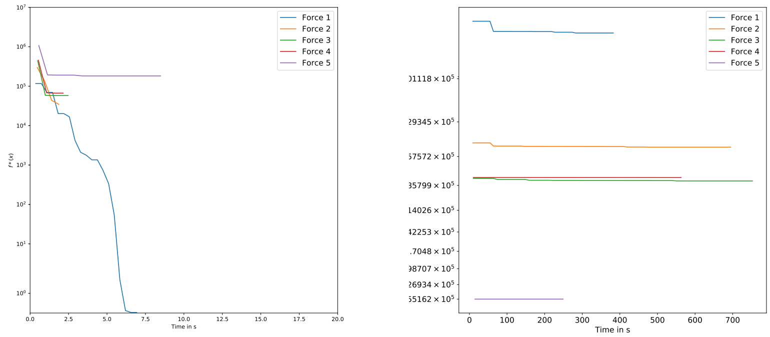


(c) Modell 1,3 Θ (d) Modell 1,3 Kraft

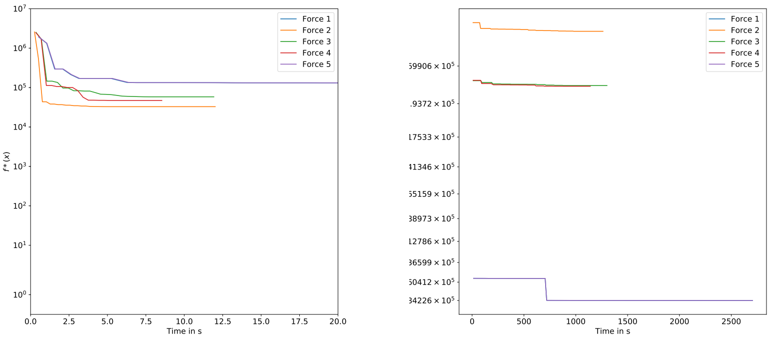


(e) Modell 2,3 Θ (f) Modell 1,3 Kraft

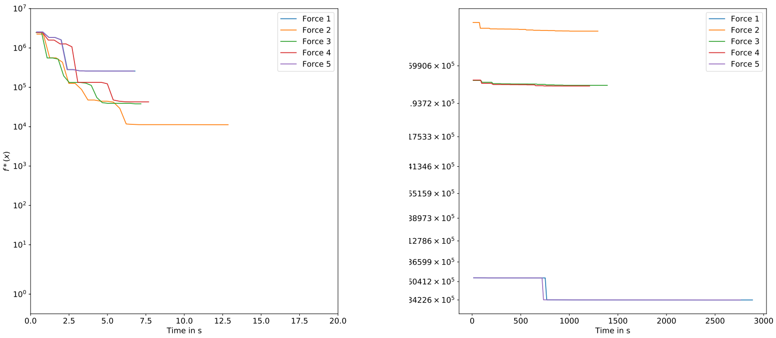
Abbildung A.3 Verlauf der Zielfunktionale der Optimierungsalgorithmen im ersten Schritt der Modellidentifikation bei Verwendung von AD mit dem Optimierer SLSQP.



(a) Modell 1,2 Θ (b) Modell 1,2 Kraft

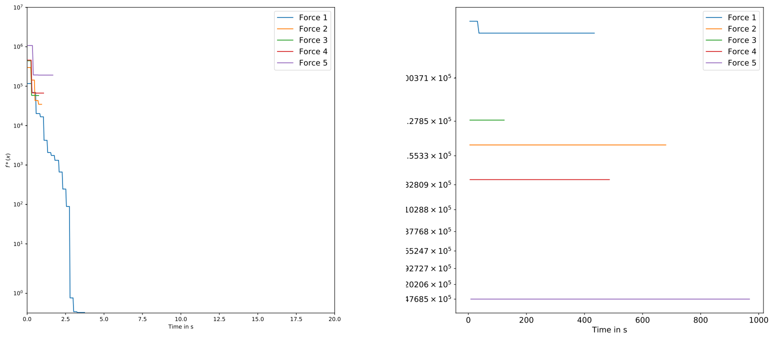


(c) Modell 1,3 Θ (d) Modell 1,3 Kraft



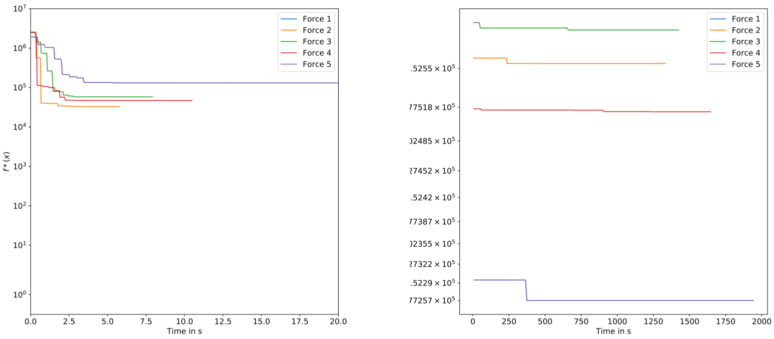
(e) Modell 2,3 Θ (f) Modell 1,3 Kraft

Abbildung A.4 Verlauf der Zielfunktionale der Optimierungsalgorithmen im ersten Schritt der Modellidentifikation bei Verwendung von finiten Differenzen mit dem Optimierer SLSQP.



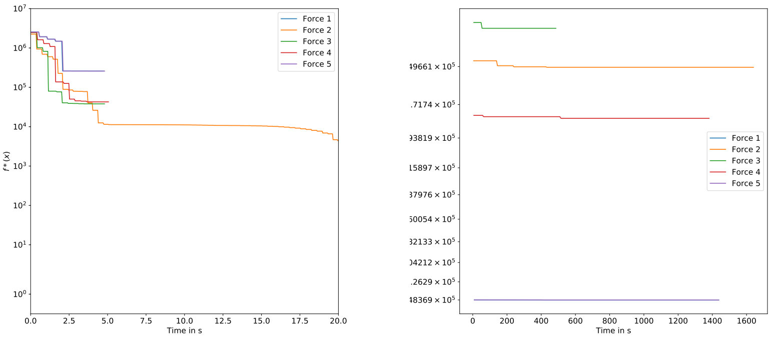
(a) Modell 1,2 Θ

(b) Modell 1,2 Kraft



(c) Modell 1,3 Θ

(d) Modell 1,3 Kraft



(e) Modell 2,3 Θ

(f) Modell 1,3 Kraft

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Ehrenwörtliche Erklärung

Hiermit erkläre ich,

- dass mir die Promotionsordnung der Fakultät bekannt ist,
- dass ich die Dissertation selbst angefertigt habe, keine Textabschnitte und Ergebnisse eines Dritten oder eigenen Prüfungsarbeiten ohne Kennzeichnung übernommen und alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben habe,
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und dass Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- dass ich die Dissertation nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe.

Ich habe die gleiche, eine in wesentlichen Teilen ähnliche bzw. eine andere Abhandlung* bereits bei einer anderen Hochschule als Dissertation eingereicht: Ja / [✓ Nein]*

(*Zutreffendes bitte unterstreichen)

Jena, den 24. September 2020

Unterschrift