ulm university universität
uulm

Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften

Automatisches Differenzieren in Simulationen

Dissertation

in Mathematik

vorgelegt von
Ralf, Leidenberger
am 10. Dezember 2013

Gutachter

Prof. Dr. Karsten Urban
Prof. Dr. Franz Schweiggert

ulm university universität
uulm

Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften

Automatisches Differenzieren in Simulationen

Dissertation

in Mathematik

vorgelegt von
Ralf, Leidenberger
am 10. Dezember 2013

Gutachter

Prof. Dr. Karsten Urban
Prof. Dr. Franz Schweiggert

Dekan:	Prof. Dr. Dieter Rautenbach
Erstgutachter:	Prof. Dr. Kartsen Urban
Zweitgutachter:	Prof. Dr. Franz Schweiggert
Tag der Promotion:	7. Juli 2014

Danksagung

Ich möchte mich bei allen Menschen, die mich auf meinem Weg unterstützt haben bedanken. Da ich sicherlich den einen oder anderen bei einer namentlichen Erwähnung vergessen würde, möchte ich es mit der namentlichen Nennung meiner Betreuer Prof. Dr. Karsten Urban und Prof. Dr. Franz Schweiggert belassen.

Gewidmet ist diese Arbeit meiner Frau Natalie und meiner Tochter Leonie, sowie meinen Eltern.

Danke für Eure Unterstützung.

Meiner Frau Natalie gilt zusätzlich besonderer Dankfür die aktive Hilfe beim Korrekturlesen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Im Rahmen dieser Promotion sollte die Anwendung des Automatischen Differenzierens (AD) in Simulationen aus unterschiedlichen Bereichen untersucht werden. Grundsätzlich ist das Automatische Differenzieren ein Ansatz, der bei allen Simulationen angewandt werden kann, wenn das zugrundeliegende Problem differenzierbar und der Quellcode der Simulation vorhanden ist. Die Vorteile, des Automatischen Differenzierens können sehr unterschiedlich sein. Sie reichen vom klassischen Berechnen der Ableitungswerte für beispielsweise Optimierungsverfahren über die Bestimmung von Sensitivitäten, bis zur Reduktion des zu entwickelnden Codes. Desweiteren werden die Untersuchungen Erkenntnisse liefern, die es ermöglichen, in Zukunft Aufwand und Schwierigkeiten der Anwendung des Automatischen Differenzierens auf eine Simulation im Vorfeld abzuschätzen.
Die Anwendungsbereiche sind so gewählt, dass man Anwendungen aus sehr unterschiedlichen Bereichen - dem naturwissenschaftlichen und dem ökonomischen-, abdeckt, mit ihren spezifischen Eigenheiten bei der Simulation. Jeder Bereich hat seine unterschiedliche Herangehensweise an die Problemstellungen.
Der Aufbau dieser Arbeit gliedert sich in drei thematische Kapitel: *Automatisches Differenzieren*, *Strömungsmechanik* und *Finanzmathematische Simulation*. In einem abschließenden Kapitel fassen wir die Resultate am Schluss zusammen.

Die Tools, die im Rahmen dieser Arbeit entwickelt wurden, als auch die vorgestellten Beispielprogramme, sind auf der beigelegten CD enthalten. In der Arbeit werden wir auf die Programme verweisen.

Im Kapitel 2 führen wir zuerst das Automatische Differenzieren ein. In der Einführung der Methode befassen wir uns mit den unterschiedlichen Varianten des Automatischen Differenzierens. Einführungen zum Automatischen Differenzieren lassen sich beispielsweise auch in den Arbeiten *Automatic Differentiation: Applications, Theory, and Implementations* [MB06], *Evaluating Derivatives* [GW08] oder *Automatic differentiation in flow simulation* [Lei07] finden.
Anschließend widmen wir uns den Herausforderungen des Automatischen Differenzierens, die sich in die drei Kategorien - technische und numerische Herausforderungen und Effizienz-Anforderungen - einteilen lassen. Die Erkenntnisse sind bei den Untersuchungen im Rahmen dieser Arbeit entstanden, beziehungsweise übertragen bekannte Phänomene der Informatik und Numerik in den Kontext des Automatischen Differenzierens.
Im letzten Teil dieses Kapitels beschreiben und bewerten wir die im Zuge dieser Promotion neu entwickelten Werkzeuge. Die Werkzeuge sind ein Konzept zur Entwicklung eines programmiersprachen unabhängigen Source-Transformation-Tools. Dieses Konzept ermöglicht eine Trennung von den Bereichen der Analyse des Programmcodes zur Ermittelung einer Abhängigkeit innerhalb des Programms, was eine Tätigkeit mit sehr hohem Informatik-Anteil ist, von der mathematisch-numerischen Aufgabe möglichst effiziente Ableitungsstrukturen zu

ermitteln. Sowie eine C++-Bibliothek mit Überladenen Operatoren, die ein Debugging Werkzeug zur Identifikation von numerischen Instabilitäten zur Laufzeit besitzt. Die Besonderheit dieses Mechanismusses ist, dass er ohne Modifikation des Programms per Compileroption aktiviert und deaktiviert werden kann. Im deaktivierten Zustand entsteht kein zusätzlicher Aufwand zur Laufzeit. Tools, die zum Teil eine ähnliche Funktionalität, beziehungsweise Herangehensweise besitzen, sind in *OpenAD/F: User Manual* [UNL11] und *FADBAD, a flexible C++ package for automatic differentiation* [BS96] beschrieben. Da alle vorhandenen Softwarepakete zum Automatischen Differenzieren, dieselben Grund Funktionalitäten zur Verfügung stellen, gibt es auch zu diesen Berührungspunkte, ohne dass eine Beeinflussung dieser stattgefunden hat.

Im Kapitel 3 *Strömungsdynamik* leiten wir uns zuerst die physikalischen Grundlagen zur Beschreibung der Strömungsmechanik her. In der physikalischen Einführung gehen wir zunächst auf die Euler-Gleichungen ein. Diese erweitern wir dann zu den Navier-Stokes-Gleichungen. Den physikalischen Grundlagen folgt ein Abschnitt, in dem wir die benötigten numerischen Grundlagen erläutern. Dort gehen wir kurz auf die Strömungssimulation mit der Finiten-Differenzen-Methode und der Finiten-Elemente-Methode ein. Die Finite-Volumen-Methode betrachten wir detaillierter, da die später verwendeten Simulationsprogramme dieses Verfahren verwenden. Eine Auswahl an Herangehensweisen an die Strömungsmechanik und ihre numerische Betrachtung finden wir auch in den Büchern *Grundzüge der Strömungslehre* von Zierep und Bühler [ZB08], *Strömungslehre* von Spurk und Aksel [SA07], *Numerische Strömungsmechanik* von Ferziger und Peric [FP08] und *Numerische Simulation in der Strömungsmechanik, eine praxisorientierte Einführung* von Griebel, Dornseifer und Neunhoeffer [GDN95].
Anschließend diskutieren wir die sinnvolle Anwendung des Automatischen Differenzierens in diesem Kontext. Darauf aufbauend werden wir die Besonderheiten, die bei der Numerischen Strömungssimualtion für das Automatische Differenzieren entstehen, betrachten. Wir werden die Anwendung an zwei verschiedenen CFD-Programmen betrachten: An dem Programm Caffa und dem Programm Comet. Beide Programme verwenden das Konzept der Finiten-Volumen-Methode. Von der Funktionalität und dem internen Aufbau unterscheiden sich die Programme zum Teil massiv. Caffa ist ein 2D-Löser, bei dem die Bewegung der Gitter vollständig im Quellcode integriert ist. Comet ist ein 2D/3D-Löser, bei dem die Bewegung der Gitterstruktur durch ein externes Tool über eine Skriptsprache realisiert ist. Wir werden die Programme und die damit zusammenhängenden Probleme ausführlich analysieren.
Als wesentliche Neuerung haben wir hier die ausführliche Analyse der Anwendung des Automatischen Differenzierens im Bereich der Strömungsmechanik mit bewegten Gittern geleistet.

Im finanzmathematischen Teil dieser Arbeit, Kapitel 4, beschäftigen wir uns zuerst mit der finanzmathematischen Formulierung von Optionspreisen. Anschließend betrachten wir zwei Modelle zur Berechnung von Optionspreisen. Als Einführung in diesen Themenkomplex sei *Financial Mathematics I* [Kie04] und *Numerische Methoden der Finanzmathematik* [Grü09] genannt.
Da für die Berechnung der Optionspreise mit den Modellen die numerische Integration eine wichtige Rolle spielt, befassen wir uns anschließend näher mit der numerischen Integration. Nachdem wir die Grundlagen abgeschlossen haben, betrachten wir zuerst das Black-Scholes-Modell zur Optionspreis-Berechnung, erörtern anhand dieses Modells den Nutzen und die An-

wendbarkeit des Automatischen Differenzierens und stellen die Ergebnisse vor. Anschließend befassen wir uns mit dem Heston-Modell, das als Erweiterung des Black-Scholes-Modell zu verstehen ist und wenden auch hier das Automatische Differenzieren an. Beim Heston-Modell haben wir mit dem Automatischen Differenzieren eine Parameter-Kalibrierung durchgeführt. Zuletzt fassen wir die Kenntnisse aus diesem Bereich zu einer Richtlinie zusammen.
In diesem Kapitel haben wir die Anwendung des Automatischen Differenzierens auf das Heston-Modell in dieser Form und die Zusammenfassung der Erkenntnisse zu den *Richtlinien zur Sensitivitäten-Ermittlung in der Finanzmathematik* als neuen Beitrag geleistet.

Im letzten Kapitel dieser Arbeitet fassen wir die Ergebnisse zusammen und geben einen Überblick über die Anwendung des Automatischen Differenzierens. Die Ergebnisse lassen sich sowohl in die Bereiche der Weiterentwicklung des Automatischen Differenzierens, Erkenntnisse bei der Anwendung in der Strömungsmechanik und der Optionspreisberechnung als auch gemeinsam in einem gesamtheitlichen Kontext, eingliedern. Zum Abschluss geben wir noch einen kleinen Ausblick für mögliche weitere Betrachtungen.

2 Automatisches Differenzieren

In diesem Kapitel führen wir das zentrale Thema dieser Arbeit, das Automatische Differenzieren, ein. Wir betrachten zunächst die Grundidee des Automatischen Differenzierenes. Danach befassen wir uns mit den Herausforderungen, die beim Anwenden des Automatischen Differenzierens entstehen. Zuletzt werden die im Rahmen der Arbeit entstandenen Tools beschrieben.

Bevor wir näher in die Betrachtung des Automatischen Differenzierens einsteigen, geben wir einen Überblick über die verwendete Litertaur. Als Einführung in das Thema sind die Arbeiten *Automatic Differentiation: Applications, Theory, and Implementations* [MB06], *Evaluating Derivatives* von Griewank und Walther [GW08], *Algorithmisches Differenzieren* von Fischer [Fis05] und *Automatic differentiation in flow simulation* von Leidenberger [Lei07] geeignet. Als Dokumentationen von Tools zum Automatischen Differenzieren sind beispielsweise *OpenAD/F: User Manual* von Utke, Naumann und Lyons [UNL11] und *FADBAD, a flexible C++ package for automatic differentiation* von Bendtsen und Stauning [BS96] zu empfehlen. Als eine thematisch verwandte Anwendung sei *Automatic Differentiation for the Optimization of a Ship Propulsion and Steering System: A proof of concept* von Leidenberger und Urban [LU11] genannt. Auf weitere verwendete Litertur wird im Text verwiesen.

2.1 Das Prinzip des Automatischen Differenzierens

Bemerkung 2.1.1 (Bezeichnungen)
Das Automatische Differenzieren wird auch als Algorithmisches Differenzieren bezeichnet. Die gebräuchliche Abkürzung für das Automatische Differenzieren wie auch für die alternative Bezeichnung Algorithmisches Differenzieren ist **AD**.

Beim Automatischen Differenzieren betrachten wir in der Regel ein Problem der folgenden Art:

Definition 2.1.2 (Problemstellung - Automatisches Differenzieren)
Gegeben ist eine Funktion,

$$f \in \mathcal{C}^d(\Omega),\ d \in \mathbb{N}\ \Omega \subset \mathbb{R}^m\ und\ \Omega\ ist\ offen,$$

$$f : \Omega \to \mathbb{R}^n,$$

und ein Program $\mathtt{P}$*, das die Funktion* f *berechnet. Gesucht sind neben den Werten von* f *an der Stelle* $x \in \Omega$ *auch die Werte von* f' *an der Stelle* x*.*

Ausgehend von einer Problemstellung wie in Definition 2.1.2 haben wir zwei Möglichkeiten, die Werte von f' an der Stelle x zu berechnen: Die erste Variante ist ein analytischer Zugang. Hierbei berechnet man den Gradienten von f und schreibt ein Programm $\mathtt{P}'$, welches die Werte von f' berechnet. Dieses Vorgehen hat mehrere Nachteile. Zum einem kann das Aufstellen der Ableitung ziemlich aufwendig sein, oder es ist nicht möglich, da man keine geschlossene

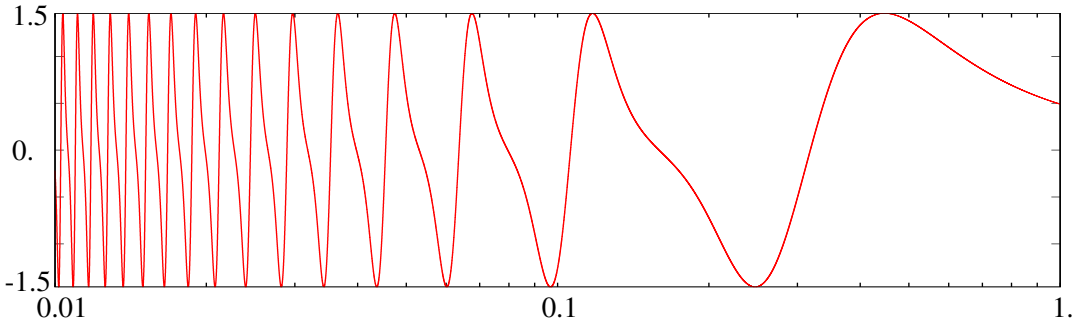

Abbildung 2.1: Darstellung der Funktion (2.1) auf dem Intervall $(0.01, 1)$.

Formel für f besitzt und das Programm P die Werte von f mittels einer Simulation berechnet. Der alternative Zugang berücksichtigt genau diese Probleme, indem er das direkte Aufstellen des Gradienten von f vermeidet. Stattdessen wird das Programm P sequenziell abgeleitet, dass heißt jede Zeile von P, die zur Berechnung von f mathematisch relevant ist, wird abgeleitet. Dass dieser Ansatz in der Theorie zum selben Ergebnis führt, sieht man leicht, wenn wir die Linearität des Differenzial Operators betrachten und berücksichtigen, dass ein Programm P nur elementar differenzierbare Operationen verwendet. Das sequenziell abgeleitete Programm P bezeichnen wir im Folgenden mit $\mathtt{P}_{\mathtt{AD}}$. Das sequenzielle Ableiten des Programmcodes kann von Hand erfolgen, was allerdings relativ aufwendig und sehr fehleranfällig ist. Die effizientere und Fehler unanfälligere Variante ist die des automatischen beziehungsweise algorithmischen Ableitens des Programmcodes. Probleme, die beim Ableiten des Programmcodes entstehen können, betrachten wir später im Abschnitt 2.2 ausführlich.
Diese abstrakte Erklärung des Automatischen Differenzierens veranschaulichen wir nun mit einem kleinen Beispiel.

Beispiel 2.1.3
Gegeben ist die Funktion

$$f(x) := \frac{\sin(1/x)}{\exp(\cos(1/x))}, \tag{2.1}$$

und $x \in (0, 1)$, hier sieht man direkt, dass der erste Ansatz, zuerst die Ableitung der Funktion $f(x)$ aufzustellen, relativ aufwendig ist da,

$$\begin{aligned} f'(x) &= \frac{-1/x^2 \cos(1/x)\exp(\cos(1/x)) - \sin(1/x)1/x^2 \sin(1/x)\exp(\cos(1/x))}{\exp(\cos(1/x))^2} \\ &= \frac{-1/x^2 \cos(1/x) - \sin(1/x)^2 1/x^2}{\exp(\cos(1/x))}, \end{aligned} \tag{2.2}$$

und man beim Aufstellen und Vereinfachen leicht einen Flüchtigkeitsfehler begeht. Alternativ kann man die Ableitung mit einem Computeralgebrasystem, (wie z.B. Maple[TM1] oder Mathematica[TM2]) berechnen lassen und anschließend implementieren. Dieser Ansatz ist allerdings auch etwas aufwendig.
Die Idee, den Ableitungswert mittels Finite-Differenzen zu approximieren, ist bei dieser Funktion nicht trivial, da sie im Bereich $x < 0, 1$ stark oszilliert, wie in Abbildung 2.1 veranschaulicht

[1] Ein Produkt der Waterloo Maple Inc.
[2] Ein Produkt der Wolfram Research, Inc.

ist. Die x-Achse ist logarithmisch skaliert.
Somit bietet es sich an, direkt den Programmcode zu modifizieren, so dass die Ableitung mitberechnet wird, ohne die Ableitung der Funktion aufzustellen. Diesen Ansatz kann man direkt mit dem Automatischen Differenzieren realisieren.

Listing 2.1: Implementierung von Gleichung (2.1)

```
1 double f(double x) {
2     double tmp = 1./x;
3     return sin(tmp)/exp(cos(tmp));
4 }
```

Listing 2.2: Implementierung von Gleichung (2.1) und (2.2) mit AD

```
1 ad f(ad x) {
2     ad tmp = 1./x;
3     return sin(tmp)/exp(cos(tmp));
4 }
```

Im Listing 2.1 ist die Implementierung der Funktion $f(x)$ aus Gleichung (2.1) in C++ abgebildet, in Listing 2.2 ist die Implementierung von $f(x)$ und $f'(x)$ gezeigt. Die Ableitung wird mit Hilfe von Überladenen Operatoren berechnet. Wir mussten nur drei Variablen vom Typ `double` *in den Type* `ad` *ändern.*

Beim Automatischen Differenzieren gibt es zwei Möglichkeiten der Umsetzung: Zum einen das Prinzip der Source-Transformation und zum anderen das der Überladenen Operatoren. Im Folgenden geben wir ein Entscheidungsschema zur Auswahl des geeigneteren Ansatzes.
Beim Automatischen Differenzieren gibt es zwei grundlegende Situationen. Die erste Möglichkeit ist, dass das Programm zur Berechnung der Zielgröße, die abgeleitet werden soll, schon existiert. In diesem Fall muss man unterscheiden, ob die Implementierung kurz und übersichtlich oder ob sie umfangreich und komplex ist. Für den umfangreichen, komplexen Fall sollte die Source-Transformations Variante gewählt werden. Im Falle, dass der Quellcode kurz und übersichtlich ist, muss die Programmiersprache berücksichtigt werden. Bei einer Sprache, die das Prinzip der Überladenen Operator unterstützt, ist dieser Ansatz für das Automatische Differenzieren zu bevorzugen. Ansonsten muss man mangels Alternativen grundsätzlich die Source-Transformation anwenden. Bei der zweiten Möglichkeit muss man das Programm zur Berechnung der Zielgröße erst implementieren. Hier bietet es sich an, den Ansatz der Überladenen Operatoren zu bevorzugen, vorausgesetzt man kann eine Programmiersprache wählen, die diese Funktionalität besitzt.
Die Entscheidung welche Automatische Differentiation's Variante man bevorzugen sollte, kann man nochmals der Entscheidungsgrafik in Abbildung 2.2 entnehmen.

Die Source-Transformation hat den Vorteil, dass es keine Voraussetzungen an die Programmiersprache stellt, solange man ein AD-Tool für die Programmiersprache hat. Zudem ist sie relativ einfach auf komplexe bestehende Programme anwendbar. Der Nachteil dieses Ansatzes ist, dass der neu erzeugte Code in der Regel deutlich umfangreicher und schwerer zu lesen ist. Damit geht einher, dass die Weiterentwicklung eines so differenzierten Codes sehr aufwendig ist.
Bei dem Ansatz der Überladenen Operatoren wird eine formale Anforderung an die Programmiersprache gestellt, nämlich dass sie das Prinzip der Überladenen Operatoren unterstützt, wie zum Beispiel C++. Ein Nachteil dieses Ansatzes ist die Anwendung auf große und komplexe bestehende Programme. Bei Neuentwicklungen ist dieser Ansatz leicht und gut nachvollziehbar anwendbar.

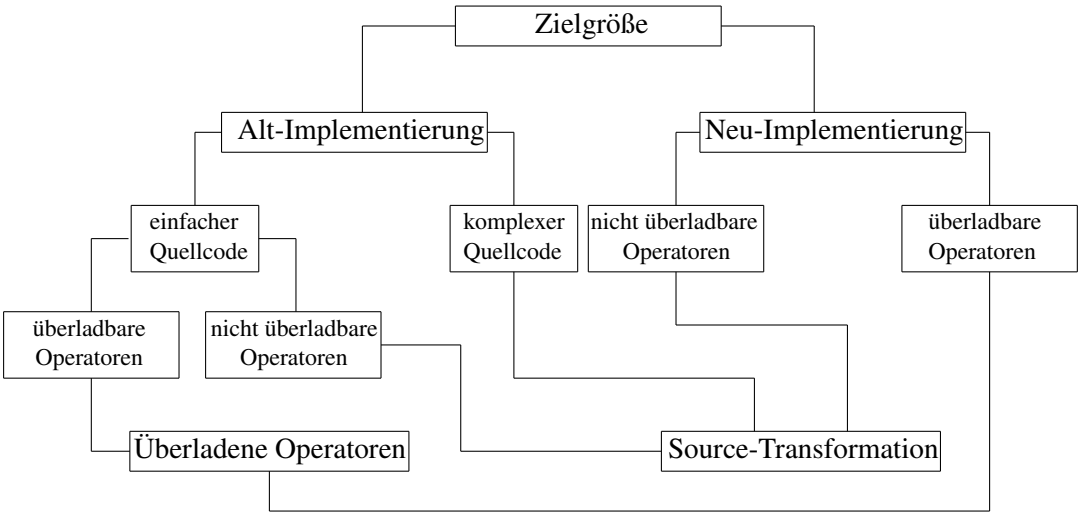

Abbildung 2.2: Entscheidungsgrafik AD-Variante.

Ein weiteres Unterscheidungsmerkmal beim Automatischen Differenzieren ist der gewählte Modus. Man unterscheidet zwischen dem Vorwärtsmodus (Forward Mode) und dem Rückwärtsmodus (Reverse Mode) .
Der Vorwärtsmodus arbeitet direkt, dies bedeutet, dass die Ableitungswerte simultan zu den Funktionswerten berechnet werden. Von temporären Objekten abgesehen, müssen, - bis auf die Ableitungswerte der einzelnen Variablen - keine Informationen gespeichert werden.
Der Rückwärtsmodus arbeitet in einem zweistufigen Vorgehen. Im ersten Schritt werden alle mathematischen Operationen und zugehörigen Funktionswerte in einer Datenstruktur gespeichert. Im zweiten Schritt wird ausgehend von dieser Datenstruktur rückwärts die Ableitung der Zielgröße ermittelt.
Eine detailierte Beschreibung des Vorwärts- und Rückwärtsmodus lässt sich in der Arbeit *Algorithmisches Differenzieren in Simulation und Optimalsteuerung von Differentialgleichungen* von Slawig [Sla05] finden.

2.2 Herausforderungen beim Automatischen Differenzieren

Dieser Abschnitt behandelt die Herausforderungen, die beim Anwenden des Automatischen Differenzierens auftreten können und stellt Methoden zum besseren Handhaben dieser Problematiken zur Verfügung.
Man kann die Problematiken in drei Teilbereiche aufteilen:

- Technische Herausforderungen,
- Numerische Herausforderungen und
- Effizienz-Anforderungen.

Im Folgenden betrachten wir die drei Problemklassen ausführlich.

2.2.1 Technische Herausforderungen

Mit *Technischen Herausforderungen* werden die Problematiken bezeichnet, die direkt aus dem programmiertechnischen Bereich heraus entstehen. Dies umfasst Probleme wie die Verwendung von externen Bibliotheken oder die Kombination von verschiedenen Programmiersprachen.

Externe Bibliotheken

Bei der Verwendung von externen Bibliotheken muss der Quellcode der Bibliothek zu Verfügung stehen. Zudem sollte der Quellcode in der gleichen Programmiersprache sein, ansonsten gibt es andere technische Schwierigkeiten, auf die wir im Abschnitt *Verschiedene Programmiersprachen* näher eingehen werden.
Weitere Probleme können bei der Umsetzung aufgrund der Größe, beziehungsweise der Komplexität oder des Programmierstils auftreten. Diese Probleme sind allerdings durch eine gute Vorarbeit lösbar. Wenn eines oder mehrere der genannten Probleme auftreten, muss man überprüfen ob die ganze Bibliothek oder nur Teile für das Automatische Differenzieren relevant sind.
Eine langfristige Möglichkeit diese Problematiken zu minimieren, wäre eine Entwicklung der Bibliotheken unter Berücksichtigung des Automatischen Differenzierens. Bei einer C++-Bibliothek wäre dies Beispielhaft relativ einfach durch Verwendung von Templates möglich. Eine andere interessante Anwendung wäre eine Portierung von BLAS[3] und LAPACK[4] zu einer *standardisierten* AD-Schnittstelle.

Verschiedene Programmiersprachen

Bei der Verwendung von verschiedenen Programmiersprachen innerhalb eines Projektes kann es zu gewissen technischen Problemen kommen. Speziell muss man hierbei zwischen dem Operator Überladen und der Source-Transformation unterscheiden.
Hinzu kommen Problematiken, die es generell bei der Verwendung von verschiedenen Programmiersprachen gibt. Hierzu zählt als Beispiel die unterschiedliche Speicherung von Arrays. Dies tritt beispielsweise bei der Verknüpfung von Fortran und C auf. Zuerst erläutern wir hier das Vorgehen und die Probleme beim Ansatz der Source-Transformations und anschließend mit den Überladenen Operatoren.

Source-Transformation
Die erste Schwierigkeit besteht in der Auswahl eines AD-Tools zur Source-Transformation, welches beide Programmiersprachen unterstützt. In folgenden Beispielen erläutern wir die auftretenden Effekte anhand eines Programmes in C und Fortran.

Beispiel 2.2.1 (C-Fortran)
Wir betrachten in diesem Beispiel ein C Programm, das eine Fortran Routine in einem kleinen übersichtlichen Programm aufruft. Der zugrundeliegende Code ist in den Listings 2.3 und 2.4 gegeben.

[3] **B**asic **L**inear **A**lgebra **S**ubprograms
[4] **L**inear **A**lgebra **PACK**age

Listing 2.3: C-Fortran: Aufruf der Fortran Routine

```
1  float quadratic_(float*);
2
3  float callQuadratic(float x, float* f)
4  {
5      *f = quadratic_(&x);
6      return *f;
7  }
```

In der Zeile 1 von Listing 2.3 ist die Fortran Routine deklariert. Zu beachten ist, dass der Unterstrich am Ende des Funktionsnamens notwendig ist, da der verwendete Compiler der `gcc`[5] *beim Compilieren der Fortran Routine diesen selbstständig anfügt.*
In Zeile 5 des Listings 2.3 wird dann die Fortran Routine aus dem C Programm heraus aufgerufen.

Listing 2.4: C-Fortran: Fortran Routine

```
1        real function quadratic(x)
2          real x
3          quadratic = x*x
4          return
5        end function
```

Das Automatische Differenzieren mit dem Source-Transformation Ansatz realisieren wir mit dem Tool Tapenade [6]. Dieses einfache Beispiel zeigt schon schön die Probleme, die Tapenade mit den Programmiersprachen hat, auf. Eine Einschränkung ist, dass der neu erzeugte Code in nur einer Programmiersprache vorliegt, das heißt es ist nicht möglich, dass die C Funktion `CallFortran` *und die Fortran Funktion* `quadratic` *in einem Durchlauf erzeugt werden. Desweiteren entdeckt Tapenade die vom Compiler erzwungene Namesanpassung mit dem Unterstrich. Dadurch erkennt Tapenade nicht, dass die Variable* `f` *von* `x` *abhängig ist und leitet diese Zeile folglich nicht ab. In diesem übersichtlichen Beispiel ist die notwendige manuelle Code-Änderung kein Problem. In einem großen Programm ist dies unter Umständen nicht so einfach und birgt eine Fehlerquelle.*

Listing 2.5: C-Fortran: Fortran AD Routine

```
1        REAL FUNCTION QUADRATIC_D(x, xd, quadratic)
2          IMPLICIT NONE
3          REAL x
4          REAL xd
5          REAL quadratic
6          quadratic_d = xd*x + x*xd
7          quadratic = x*x
8          RETURN
9        END
```

In Listing 2.5 ist der automatisch abgeleitet Code der Fortran Routine aufgezeigt. Man sieht direkt, dass der Code nicht minimal ist. Das folgende Listing 2.6 zeigt den von Hand modifizierten C Code.

Listing 2.6: C-Fortran: Aufruf der Fortran AD Routine

```
1  float quadratic_d_(float*, float*, float*);
2
3  float callQuadratic(float x, float xd, float* f, float* fd)
4  {
5      *fd = quadratic_d_(&x, &xd, f);
6      return *fd;
7  }
```

[5] gcc-mp-4.4 (GCC) 4.4.6
[6] Tapenade wurde von der INRIA in Frankreich entwickelt. Die verwendete Version ist 3.5 (r3931).

Der vollständige Programmcode aus dem Beispiel 2.2.1 ist auf der beiliegenden CD im Verzeichnis `Programme/ex:MixCF01` hinterlegt.
Im Verzeichnis `Programme/ex:MixCF02` ist noch ein Beispiel gegeben, in dem das Fortran Programm eine C Routine aufruft.

Bemerkung 2.2.2 (Fortran Routinen in C-Programmen)
Die Verwendung von Fortran Routinen in einem C oder C++ Programm ist kein rein akademisches Beispiel. Diese Kombination tritt bei der Verwendung von BLAS auf, da die Bibliothek in Fortran geschrieben ist.

Im Anschluss an die Betrachtung der verschiedenen Programmiersprachen mit dem Source-Transformations Ansatz, kommen wir zum Ansatz der Überladenen Operatoren imselben Kontext.

Überladene Operatoren

Bei der Verwendung von überladenen Operatoren mit verschiedenen Programmiersprachen sind die Schwierigkeiten zum einen die Schnittstelle zwischen den Programmiersprachen und zum anderen, dass beide Programmiersprachen das Konzept der überladenen Operatoren unterstützen.

Beispiel 2.2.3 (Java-C++)
In diesem Beispiel betrachten wir ein Java Programm, das eine C++ Routine mit einem Argument vom Typ AD aufruft. Der Rückgabewert ist ebenfalls vom Typ AD. Den Sourcecode der AD-Klassen in Java und C++ haben wir hier nicht wiedergegeben, da er für die Problematik, die bei der Verknüpfung auftritt, nicht relevant ist. Die in diesem Beispiel verwendeten Klassen beschränken sich der Einfachheit halber auf das Nötigste und stellen keine Implementierung dar, die für den produktiven Einsatz gedacht ist.
Die vollständige Implementierung des Beispieles ist auf der beiliegenden CD im Verzeichnis `Programme/ex:MixJCpp01` *zu finden.*
In Listing 2.7 ist das Java Programm, das die C++ Routine aufruft, abgebildet.

Listing 2.7: Java-C++: Aufruf der Cpp Routine

```
1  class javaCppAD {
2      public native AD CppADFct(AD x);
3      static {
4          System.loadLibrary("CppADlib");
5      }
6
7      public static void main(String args[]) {
8          javaCppAD fct = new javaCppAD();
9
10         AD x = new AD(1.2,1);
11         AD f = new AD(fct.CppADFct(x));
12
13         System.out.println("x =" + x.toString());
14         System.out.println("f =" + f.toString());
15     }
16 }
```

Zeile 2 deklariert die native Methode `CppADFct`*. Die Zeile 4 lädt die C++ Routine in Form einer Bibliothek. In Zeile 11 wird die C++ Routine aufgerufen und deren Rückgabewert an den Konstruktor der Java AD Klasse übergeben.*
Das Listing 2.8 zeigt, das vom Programm `javah` *automatisch erzeugte Headerfile für die C++ Routine.*

Listing 2.8: Java-C++: Header für C++ Routine

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class javaCppAD */
4
5  #ifndef _Included_javaCppAD
6  #define _Included_javaCppAD
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:     javaCppAD
12  * Method:    CppADFct
13  * Signature: (LAD;)LAD;
14  */
15 JNIEXPORT jobject JNICALL Java_javaCppAD_CppADFct
16   (JNIEnv *, jobject, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
```

Die Implementierung dieser Schnittstelle ist im Listing 2.9 wiedergegeben. In den Zeilen 5 bis 10 lesen wir den Java AD-Datentyp aus und in den Zeilen 15 bis 19 setzen wir den Rückgabewert. Wir sehen sofort, dass es sehr viel Aufwand ist, die Java Datentypen aus C++ heraus auszulesen beziehungsweise sie zu setzen. Bei nativen Datentypen, wie zum Beispiel `double`*, geht dies sehr viel einfacher, mit* `double x = (double)_x;` *wobei* `_x` *vom Type* `jdouble` *ist. Das setzen einer* `jdouble` *Variable geht einfach mit* `_x = (jdouble)x;`*.*

Listing 2.9: Java-C++: C++ Routine

```
1  AD quadratic(const AD &x) { return x+x; }
2
3  JNIEXPORT jobject JNICALL Java_javaCppAD_CppADFct
4            (JNIEnv *env, jobject tmp, jobject _x) {
5      jclass cls = (env)->GetObjectClass(_x);
6      jfieldID fieldID;
7      fieldID = (env)->GetFieldID(cls, "_value", "D");
8      double value = (env)->GetDoubleField( _x, fieldID);
9      fieldID = (env)->GetFieldID(cls, "_derivative","D");
10     double derivative = (env)->GetDoubleField(_x, fieldID);
11
12     AD x(value, derivative);
13     AD f = quadratic(x);
14
15     jclass _f(cls);
16     fieldID = (env)->GetFieldID(cls, "_value", "D");
17     env->SetDoubleField( _f, fieldID, (jdouble)f.getValue());
18     fieldID = (env)->GetFieldID(cls, "_derivative","D");
19     env->SetDoubleField( _f, fieldID, (jdouble)f.getDerivative());
20
21     return _f;
22 }
```

So erkennen wir sofort, dass das Automatische Differenzieren über die Grenzen von Programmiersprachen hinweg relativ viel Handarbeit benötigt.

Für vertiefende Informationen zur Verwendung von verschiedenen Programmiersprachen sind folgende Quellen von Interesse: *Using the GNU Compiler Collection* [SGDC10], *Using GNU Fortran* [tea11], Bjarne Stroustrup *The C++ Programming Language* [Str05] oder in Sheng Liang *The Java™ Native Interface* [Lia99].

Der betrachtete Abschnitt stellt keine vollständige Betrachtung des Themenkomplexes des Automatischen Differenzierens mit mehreren Programmiersprachen dar. Er ist viel mehr als Motivation zu verstehen, wenn wir später bei größeren Simulationen erklären, wo die Schwierigkeiten bei der Verknüpfung von verschiedenen Programmiersprachen und dem Automatischen

Differenzieren liegen.
Prinzipiell ist es kein Hindernis, wenn mehrere Programmiersprachen zusammen mit dem Automatischen Differenzieren verwendet werden. Es erhöht sich nur der Anteil der Handarbeit signifikant.

2.2.2 Numerische Herausforderungen

Die *Numerischen Herausforderungen*, die beim Automatischen Differenzieren auftreten können, sind vielseitig. Es kann sich um numerische Instabilitäten aufgrund von Ungenauigkeiten bei der Darstellung der Maschinenzahlen handeln, aber auch Probleme, die die Voraussetzungen eines Algorithmus' verletzten. Die dritte Klasse die hier auftreten kann, ist die, in der die Korrektheit der Problemstellung verletzt wird.

Maschinengenauigkeit

Numerische Probleme aufgrund von der Maschinengenauigkeit sind innerhalb der Numerik ein bekanntes Problem. Beim Automatischen Differenzieren können allerdings die Probleme erst beim differenzierten Code auftreten und sind somit nicht immer als solche zu erkennen.

Definition 2.2.4 (Maschinenzahl)
Wir bezeichnen im folgenden eine reelle Zahle $x \in \mathbb{R}$ in Maschinenzahldarstellung mit $x_{_M} \in \mathbb{M}$. Die Zahl $x_{_M}$ wird intern wie folgt dargestellt,

$$x_{_M} := s \cdot m \cdot b^e, \tag{2.3}$$

wobei in (2.3) *s das Vorzeichen, m die Mantisse, b die Basis und e den Exponenten repräsentiert.*

Eine ausführlichere Einführung zum Thema der Maschinenzahlen findet man beispielsweise in dem Buch *Numerische Mathematik 1* von Stoer [Sto02]. Eine exakte interne Festlegung von Zahlenformaten wird in der Norm IEEE 754 für binäre und in der Norm IEEE 854 für nicht binäre Gleitkommazahlen gegeben.
Im Folgenden betrachten wir ein kleines Beispiel, das den Sachverhalt erklärt, wie es beim ursprünglichen Algorithmus keine numerischen Probleme gibt, es beim abgeleiteten Algorithmus allerdings zu Problemen kommt.

Beispiel 2.2.5 (Maschinengenauigkeit Probleme)
Wir betrachten folgende Problemstellung:

$$f(x) := \frac{c_2 - \sqrt{x}}{\log(x) - c_1}, \tag{2.4}$$

und setzen

$$c_1 := \log 10^{-7},$$
$$c_2 := \sqrt{10^{-7}}.$$

Wir betrachten nun Gleichung (2.4) *im Bereich*

$$1 > x \to 10^{-7}.$$

Bei $x = 10^{-7}$ ist die Gleichung (2.4) *ein unbestimmter Ausdruck der Form $\frac{0}{0}$. Mit der Regel von L'Hospital, siehe [Kö00], können wir eine Grenzwertbetrachtung der Funktion durchführen.*

$$\begin{aligned}\lim_{x\to 10^{-7}} f(x) &= \lim_{x\to 10^{-7}} \frac{c_2 - \sqrt{x}}{\log(x) - c_1} = \lim_{x\to 10^{-7}} \frac{f_1(x)}{f_2(x)} \\ &= \lim_{x\to 10^{-7}} \frac{f_1'(x)}{f_2'(x)} = \lim_{x\to 10^{-7}} \frac{-1/(2\sqrt{x})}{1/x} = \lim_{x\to 10^{-7}} \frac{-\sqrt{x}}{2} = \frac{\sqrt{10^{-7}}}{2}\end{aligned}$$

Im nächsten Schritt betrachten wir die Ableitung von (2.4)*,*

$$f'(x) = \frac{-1/(2\sqrt{x})\,(\log x - c_1) - (c_2 - \sqrt{x})1/x}{(\log x - c_1)^2}. \tag{2.5}$$

Da die Gleichung (2.5) *wie die Gleichung* (2.4) *bei $x = 10^{-7}$ ein unbestimmter Ausdruck der Art $\frac{0}{0}$ ist, machen wir erneut eine Grenzwertbetrachtung mit der Regel von L'Hospital.*

$$\lim_{x\to 10^{-7}} f'(x) = \lim_{x\to 10^{-7}} \frac{\overbrace{-1/(2\sqrt{x})\,(\log x - c_1) - (c_2 - \sqrt{x})1/x}^{:=g(x)}}{\underbrace{(\log x - c_1)^2}_{:=h(x)}} \tag{2.6}$$

Die Ableitungen der in Gleichung (2.6) *definierten Funktionen $g(x)$ und $h(x)$ sind*

$$g'(x) = -\frac{1}{4}x^{-\frac{3}{2}}(\log x - c_1) - x^{-\frac{3}{2}} + c_2 x^{-2} \tag{2.7}$$

$$= x^{-\frac{3}{2}} \underbrace{\left(-\frac{1}{4}(\log x - c_1) - 1 + c_2 x^{-\frac{1}{2}}\right)}_{:=\tilde{g}(x)}, \tag{2.8}$$

$$h'(x) = 2x^{-1}(\log x - c_1). \tag{2.9}$$

Mit den Gleichungen (2.7) *und* (2.9) *ergibt sich für* (2.6)*,*

$$\lim_{x\to 10^{-7}} f'(x) = \lim_{x\to 10^{-7}} \frac{g(x)}{h'(x)} = \lim_{x\to 10^{-7}} \frac{g'(x)}{h'(x)} \tag{2.10}$$

erneut ein Ausdruck der Form $\frac{0}{0}$. Mit dem erneuten Anwenden der Regel von L'Hospital erhalten wir

$$g''(x) = -\frac{3}{2}x^{-\frac{5}{2}}\tilde{g}(x) + x^{-\frac{3}{2}}\left(-\frac{1}{4}x^{-1} - \frac{1}{2}c_2 x^{-\frac{3}{2}}\right)$$

und

$$h''(x) = -2x^{-2}(\log x - c_1 + 1)$$

Damit sehen wir direkt, dass der Grenzwert von $f'(x)$ an der Stelle $x = 10^{-7}$ erklärt ist. Es gilt:

$$\lim_{x\to 10^{-7}} f'(x) = \lim_{x\to 10^{-7}} \frac{g(x)}{h'(x)} = \lim_{x\to 10^{-7}} \frac{g'(x)}{h'(x)} = \lim_{x\to 10^{-7}} \frac{g''(x)}{h''(x)} = -395.53. \tag{2.11}$$

Nachdem wir das Problem theoretisch untersucht haben, betrachten wir nun ein Programm, das die Funktion aus Gleichung (2.4) *und deren Ableitung, im Bereich $x \in (9.9995 \cdot 10^{-8}, 1.00005 \cdot 10^{-7})$ implementiert.*

Listing 2.10: Maschinengenauigkeit

```
1  int main() {
2      const double x0 = 0.00000009999995;
3      const double x1 = 0.00000010000005;
4      const int    N  = 10000;
5      double delta = (x1-x0)/N;
6      double c1    =  log(0.0000001);
7      double c2    = sqrt(0.0000001);
8      double init[1];
9      init[0] = 1.;
10     ad f,g,x(1,init);
11     for (int i=0; i<N; ++i) {
12         x.setValue(x0+i*delta);
13         f = (c2-sqrt(x))/(log(x)-c1);
14         cout<< i*delta<< " "<< f.value()<< " "<<f.derivative(0)<< endl;
15     }
16     return 0;
17 }
```

Das Listing 2.10 zeigt den wesentlichen Teil der Implementierung von den Gleichungen (2.4) *und* (2.5)*. In der Grafik 2.3 sind die Funktions- und Ableitungswerte der Implementierung grafisch dargestellt. Man sieht direkt, dass die Funktionswerte nahe der Stelle* $x = 10^{-7}$ *einem geringen Rauschen unterliegen. Bei den Ableitungswerten ist das Rauschen aufgrund der Maschinengenauigkeit deutlich größer.*

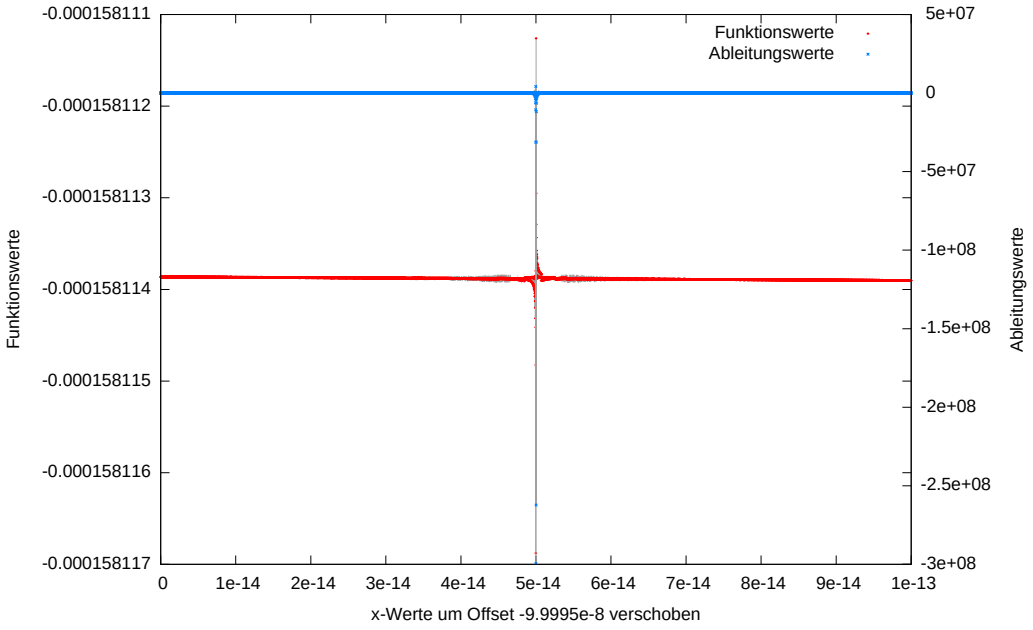

Abbildung 2.3: Grafische Darstellung der Funktions- und Ableitungswerte der Funktionen (2.4) und (2.5), berechnet mit dem Programm aus Listing 2.10.

Im Anschluss an dieses Beispiel, in dem das Problem in einer Umgebung um eine Stelle auftritt an der der Grenzwert des Ausdruckes existiert, allerdings der Ausdruck selbst nicht definiert ist, betrachten wir nun ein Beispiel, in dem die Ableitung beschränkt ist, aber signifikant größer als der Funktionswert wird.

Beispiel 2.2.6 (Differentiations invariater Quotient)
Nun betrachten wir eine Funktion bei der der Quotient bezüglich des Differentationsparamater invariant ist, der Zähler allerdings signifikant vom Differentationsparamater abhängt. Dazu betrachten wir folgende Funktion

$$\underbrace{f_n(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)}_{=:f_n(\alpha)} := f_{n-1}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)\frac{g(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)}{V(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)}, \tag{2.12}$$

wobei für $\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha$ *gilt:*

$$\cdot_\alpha := \cdot(\alpha) := \begin{pmatrix} \cdot_1 \cos\alpha - \cdot_2 \sin\alpha \\ \cdot_1 \sin\alpha + \cdot_2 \cos\alpha \end{pmatrix} \in \mathbb{R}^2. \tag{2.13}$$

$$\underbrace{V(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)}_{=:V(\alpha)} = 1/2\,|(u_{\alpha,2} - w_{\alpha,2})(x_{\alpha,1} - v_{\alpha,1}) + (v_{\alpha,2} - x_{\alpha,2})(u_{\alpha,1} - w_{\alpha,1})| \tag{2.14}$$

$$\underbrace{g(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)}_{=:g(\alpha)} := c\sqrt{x_{\alpha,2} - u_{\alpha,2}} \tag{2.15}$$

Durch elementares Nachrechnen erhält man, dass

$$\frac{dV}{d\alpha}(\alpha) := \frac{dV}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) = 0 \tag{2.16}$$

und

$$\frac{dg}{d\alpha}(\alpha) := \frac{dg}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) = c\frac{x_1\cos\alpha - x_2\sin\alpha - u_1\cos\alpha + u_2\sin\alpha}{2\sqrt{x_{\alpha,2} - u_{\alpha,2}}} \tag{2.17}$$

gilt. Die Ableitung des Ausgangsproblemes (2.12) *ergibt sich mit den Gleichungen* (2.16) *und* (2.17)*:*

$$\frac{df_n}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) = f_{n-1}(\alpha)\frac{\frac{dg}{d\alpha}(\alpha)}{V(\alpha)} + \frac{df_{n-1}}{d\alpha}(\alpha)\frac{g(\alpha)}{V(\alpha)}. \tag{2.18}$$

Da beim Automatischen Differenzieren das Wissen über die Invarianz des Quotienten nicht immer vorliegt, kann dies hier zu Problemen führen, da dann anstatt der Darstellung (2.18) *die folgende Form verwendet wird:*

$$\frac{df_n}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) := f_{n-1}(\alpha)\frac{\frac{dg}{d\alpha}(\alpha)V(\alpha) - g(\alpha)\frac{dV}{d\alpha}(\alpha)}{V(\alpha)^2} + \frac{df_{n-1}}{d\alpha}(\alpha)\frac{g(\alpha)}{V(\alpha)} \tag{2.19}$$

und es hier bei $V(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) << 1$ *leichter zu numerischen Problemen kommen kann. Weitere Gründe für numerische Probleme beim Ableiten können hier auch durch folgende Effekte hervorgerufen werden:*

- *Theoretisch gilt* $\frac{dV}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) = 0$ *allerdings ist der numerische Wert von* $\frac{dV}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) \neq 0$.
- *Für* $V(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha) << 1$ *und* $|\frac{dg}{d\alpha}(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)| >> |g(\mathbf{u}_\alpha, \mathbf{v}_\alpha, \mathbf{w}_\alpha, \mathbf{x}_\alpha)|$.

Wir berechnen nun die Ableitungen des Problems (2.12) *mittels Automatischen Differenzierens in den Formulierungen* (2.18) *und* (2.19). *In unserem Programm* [7] *waren bei ungünstiger Wahl der Parameter und single precision Abweichungen von bis zu* $\sim 30\%$ *zwischen den Berechnungsvarianten* (2.18) *und* (2.19) *zu verzeichnen.*

Bemerkung 2.2.7
Ein Hauptgrund für das Automatische Differenzieren ist, das man bei großen Simulationen nicht jede zu differenzierende Programmzeile von Hand ableiten und analysieren möchte, beziehungsweise es aufgrund des Programmumfangs gar nicht kann. Dies kann aber unter Umständen, wie wir sie im Beispiel 2.2.6 gesehen haben, zum Problem werden.

Auf ein Problem dieser Art kommen wir später im Abschnitt 3.3.2 nochmals zurück.

Wohlgestelltheit des Problems

Definition 2.2.8 (Wohlgestelltes Problem)
Man nennt ein Problem $f(x)$ *wohlgestellt, wenn das Problem eine eindeutige Lösung besitzt und stetig von den Daten* x *abhängt.*

In diesem Abschnitt betrachten wir ein Problem, das nach Definition 2.2.8 korrekt gestellt ist, die Ableitung allerdings nicht auf dem gesamten Definitionsbereich des Problems erklärt ist. Die dabei auftretenden Probleme erläutern wir beispielhaft an der Funktion

$$f(x) = \sqrt{x}, \tag{2.20}$$

die mittels dem Automatischen Differenzieren abgeleitet wird. Die Funktion (2.20) ist auf dem Bereich $x \in \mathbb{R}_0^+$ stetig und die Ableitung

$$f'(x) = \frac{1}{2\sqrt{x}}, \tag{2.21}$$

ist auf dem Bereich $x \in \mathbb{R}^+$ stetig differenzierbar. Eine Schwierigkeit, die beim Automatischen Differenzieren nun auftreten kann, ist, dass der Variablenwert x in der Gleichung (2.21) den Wert 0 annimmt, beziehungsweise gegen Null geht. Für $x = 0$ ist die Gleichung (2.21) nicht erklärt. Dies ist dadurch relativ leicht im Programmcode zu finden, da ab dieser Stelle der Variablenwert von `f` den Wert `inf`[8] annimmt. Deutlich schwieriger ist es allerdings, wenn theoretisch $x = 0$ gilt, aber aufgrund der Darstellung von Maschinenzahlen $x_{\mathcal{M}} \neq 0$ aber $x_{\mathcal{M}} > 0$ gilt, da hier (2.21) erklärt ist aber sehr groß ist. Probleme, die auftreten wenn $x < 0$ ist, werden in diesem Zusammenhang nicht betrachtet, da dies kein Problem beim Automatischen Differenzieren ist, sondern schon beim ursprünglichen Problem behandelt werden muss.
Generell kann man diese Art von Problemen wie folgt beschreiben:

[7] Auch dieses Programm findet sich auf der beiliegende CD, im Verzeichnis `Programme/ex:Maschinengenauigkeit02`.

[8] Der Wert `inf` wird bei C++ angenommen bei Fortran90 wird er mit `+Infinity` beziehungsweise mit `-Infinity` bezeichnet.

Bemerkung 2.2.9 (Unbekannter Wertebereich)
Wir betrachten eine Funktion

$$f(x) \in \mathcal{C}(I), \tag{2.22}$$

und $I \subset \mathbb{R}^n$. Weiter ist

$$f'(x) \in \mathcal{C}^1(\dot{I}), \tag{2.23}$$

und $\dot{I} \subset I$ aber $\dot{I} \neq I$. Der Wert, beziehungsweise die Werte, die x annimmt, sind vorher nicht sicher. Somit kann $x \in I$ aber $x \notin \dot{I}$ auftreten.

Probleme des Types wie in Definition 2.2.9 können ausser bei der Wurzelfunktion, wie wir es bei Gleichung (2.20) und (2.21) betrachten haben, beispielsweise noch beim Absolutbetrag oder bei Modulo und Rundungs-Operationen auftreten. Wenn man weitere, nicht elementare Operationen[9] betrachtet, kann die Liste beliebig erweitert werden.

Algorithmus Voraussetzungen

Dieser Abschnitt behandelt Probleme, die entstehen, wenn für den ursprünglichen Algorithmus die Voraussetzungen erfüllt sind, für die abgeleitete Version diese Voraussetzungen allerdings verletzt sind. Wir betrachten die Funktion

$$\mathbf{f}(t, \mathbf{x}) := \mathbf{x}^T \mathsf{A}(t)\mathbf{x} \tag{2.24}$$

Die Matrix $\mathsf{A} \in \mathbb{R}^{n\times n}$ und für die Koeffizienten der Matrix gilt:

$$a_{i,j}(t) := t^{c|i-j|},\ 1 \leq i, j \leq n, c \in \mathbb{N}. \tag{2.25}$$

Damit bildet die Funktion (2.24) vom Raum $\mathbb{R}^n$ in den Raum $\mathbb{R}$ ab.

Wir betrachten nun die Funktion (2.24) mit $t = 0.55$, $c = 5$, $n = 100$ und $\mathbf{x} := (\alpha, \alpha, \cdots)^T$. Zur Reduktion der Komplexität für die Auswertung von $f(t, x)$ von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$ benutzen wir nicht die volle Matrix A, sondern nur die Diagonale und ihre ersten beiden Nebendiagonalen. Das heißt wir ersetzen (2.25) durch

$$a_{i,j}(t) := \begin{cases} t^{c|i-j|} & |i-j| \leq 2 \\ 0 & |i-j| > 2 \end{cases} \tag{2.26}$$

Dies ist eine gute Näherung da die Einträge $a_{i,j}(t)$ für kleine t und mit wachsender Differenz von $|i - j|$ schnell sehr klein werden. Schauen wir uns parallel dazu noch die Ableitung von $\mathbf{f}(t, \mathbf{x})$ bezüglich t an

$$\frac{d\mathbf{f}}{dt}(t, \mathbf{x}) = \mathbf{x}^T \frac{d\mathsf{A}}{dt}(t)\mathbf{x}, \tag{2.27}$$

sehen wir direkt, dass die Einträge in der Matrix A bezüglich t abgeleitet nicht so schnell gegen Null gehen, da

$$\frac{da_{i,j}}{dt}(t) = (c|i-j|)t^{c|i-j|-1},\ 1 \leq i, j \leq n, \tag{2.28}$$

gilt.

[9] Elementar im Sinne von Programmiersprachen.

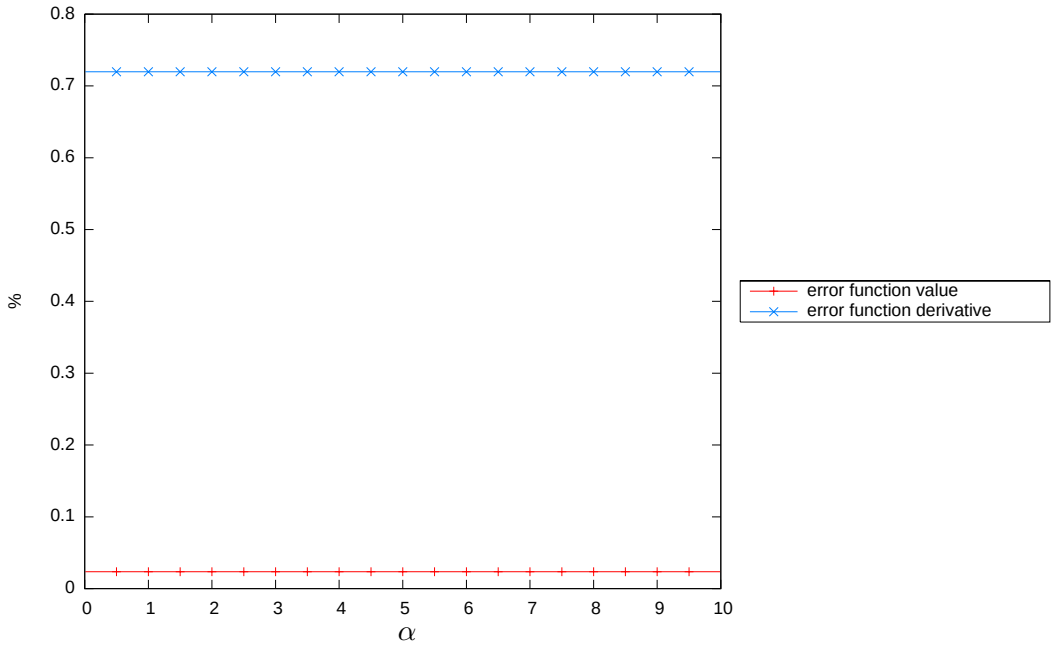

Abbildung 2.4: Prozentualer Fehler der Approximation für den Funktions-und Ableitungswert.

Das prozentuale Fehlerverhalten von (2.24) und (2.27) haben wir in Figur 2.4 dargestellt. Wir sehen, dass der Fehler des Funktionswertes bei der Variation von α näherungsweise konstant und sehr klein ist. Das Verhalten des Fehlers für die Ableitungswerte ist zwar ebenfalls bezüglich α näherungsweise konstant, aber auf einem deutlich höheren Niveau. Der Fehler der Ableitungswerte ist ungefähr 27 mal so groß wie der des Funktionswertes.
Ein weiteres Problem, das ebenfalls in diese Problemklasse gehört, ist in der Diplomarbeit *Automatic differentiation in flow simulation* von Leidenberger im Abschnitt *AD is not a silver bullet* [Lei07] beschrieben.
Probleme dieser Art innerhalb einer komplexen Simulation aufzufinden kann sich zu einem schwierigen und sehr zeitintensiven Unterfangen entwickeln, da man die ganze Simulation unter Umständen zeilenweise analysieren muss, oder man jede Approximation auf ihre Approximationsgüte in abgeleiteter Form untersuchen muss.

2.2.3 Effizienz-Anforderungn

Unter *Effizienz-Anforderungen* im Kontext des Automatischen Differenzierens versteht man die Problematik des gestiegenen Rechen- und Speicheraufwandes durch die Anwendung des Automatischen Differenzierens. Die Effizienz des zugrundeliegenden Algorithmus wird nicht betrachtet, es sei denn das Automatische Differenzieren wird gezielt eingesetzt um die Gesamtkomplexität des Problems zu reduzieren, zum Beispiel bei einer Optimierungsaufgabe um so ein effizienteres Optimierungsverfahren einsetzen zu können.

Speichereffizienz

Bei der Speichereffizienz muss zwischen dem Vorwärtsmodus und dem Rückwärtsmodus unterschieden werden. Bei dem Vorwärtsmodus ist die Abschätzung bezüglich des Speicherbedarfs relativ einfach.

Bemerkung 2.2.10 (Speicherbedarf Vorwärtsmodus)
Der Speicherbedarf eines Programmes, das man mit dem Vorwärtsmodus differenziert hat, lässt sich wie folgt abschätzen:

a) **Einfachste Abschätzung:**
Sei $\mathcal{M}$ der Speicherbedarf des ursprünglichen Programmes so gilt:

$$\mathcal{M}_{AD} \leq \mathcal{O}(\mathcal{M} \cdot (d+1)), \tag{2.29}$$

und d ist die Dimension des Gradients und $\mathcal{M}_{AD}$ der Speicherbedarf des differenzierten Programmes. Diese Abschätzung ist sehr einfach durchzuführen, allerdings überschätzt sie den Speicherbedarf des abgeleiteten Programmes in der Regel deutlich.

b) **Verbesserte Abschätzung:**
Sei $\mathcal{M}$ wieder der Speicherbedarf des ursprünglichen Programmes und $\mathcal{M}_{aktiv}$ der Speicherbedarf der aktiven Variablen. So gilt offensichtlich:

$$\mathcal{M}_{aktiv} < \mathcal{M}. \tag{2.30}$$

Damit ergibt sich für den Speicherbedarf der aktiven Variablen im abgeleiteten Zustand die Abschätzung:

$$\mathcal{M}_{aktivAD} = \mathcal{M}_{aktiv} \cdot (d+1), \tag{2.31}$$

wobei $\mathcal{M}_{aktivAD}$ der Speicherbedarf der aktiven Variablen nach dem Ableiten und d die Dimension des Gradienten ist. Aus Gleichung (2.31) folgt, dass der gesamte Speicheraufwand des abgeleiteten Programmes $\mathcal{M}_{AD}$ mit

$$\mathcal{M}_{AD} = \mathcal{O}(\mathcal{M} - \mathcal{M}_{aktiv} + \mathcal{M}_{aktivAD}) \tag{2.32}$$

angegeben werden kann. In der Praxis lässt sich der Speicherbedarf der aktiven Variablen über die Anzahl der Variablen, die Arraygröße und dem Gültigkeitsbereich abschätzen.

Wie man durch die beiden Abschätzungen in Bemerkung 2.2.10 sieht, ist der zusätzliche Speicherbedarf des Programmes beim Vorwärtsmodus relativ moderat, solange man keine übermäßig große Dimension beim Gradienten hat. Dafür gibt es so gut wie keine Möglichkeiten, diesen Bedarf noch zu senken.
Der Rückwärtsmodus ist bei der Abschätzung des Speicherbedarfes deutlich komplexer.

Bemerkung 2.2.11 (Speicherbedarf Rückwärtsmodus)
Der Speicherbedarf beim Rückwärtsmodus hängt von der Anzahl der ausgeführten aktiven Rechenoperationen $\mathcal{C}_{aktiv}$ bis zur Auswertung und einer Speicherbedarfskonstante C ab.

$$\mathcal{M}_{AD} = \mathcal{O}(\mathcal{C}_{aktiv} \cdot C + \mathcal{M}) \tag{2.33}$$

Die Schwierigkeit liegt hier A-Priori zu bestimmen wieviele aktive Rechenoperationen ausgeführt werden, da eine Vielzahl von Algorithmen Abbruchkriterien für die Konvergenz besitzen, die erst zur Laufzeit bekannt sind.

Die Bemerkung 2.2.11 hat uns deutlich gezeigt, dass man den zusätzlichen Speicherbedarf eines differenzierten Programmes mit dem Rückwärtsmodus unter Umständen A-Priori nicht oder nur sehr ungenau abschätzen kann. Diese Unsicherheit kann man durch Zwischenauswertungen teilweise umgehen, allerdings verliert man damit zum Teil die Vorteile des Rückwärtsmodus'.
Abschließend können wir zu der Speichereffizienz beim Automatischen Differenzieren festhalten, dass beim Vorwärtsmodus die Möglichkeiten zur Reduzierung des Speicheraufwandes relativ beschränkt sind, dafür sich der Speicheraufwand im Rahmen hält und gut abschätzbar ist.
Beim Rückwärtsmodus hat man mehr Möglichkeiten den Speicheraufwand zu beeinflussen, dafür ist er allerdings im Normalfall deutlich höher als beim Vorwärtsmodus. Die Reduktion des Speicheraufwandes beeinträchtigt dafür in der Regel die Recheneffizienz.

Recheneffizienz

Die Recheneffizienz beim Automatischen Differenzieren kann durch mehrere Faktoren stark beeinflusst werden. Die Wahl des Modus spielt dabei eine große Rolle, allerdings kann durch geschickte Umformulierungen der Algorithmus so angepaßt werden, dass die Rechenkomplexität deutlich reduziert wird.
Wie solche möglichen Umformulierungen aussehen können, zeigen wir anhand des Lösens von linearen Gleichungssystemen, was in vielen komplexen Simulationen eine wesentliche Rolle spielt und somit sehr relevant ist.
Wir betrachten folgendes Problem:

$$\mathsf{A}\mathbf{x} = \mathbf{b}. \tag{2.34}$$

Im nächsten Schritt möchten wir auch das zugehörige abgeleitete Problem betrachten,

$$(\mathsf{A}\mathbf{x})' = \mathbf{b}'. \tag{2.35}$$

Zum Lösen des Gleichungssystemes (2.34) gibt es grundsätzlich zwei Möglichkeiten. Die erste ist das explizite Invertieren der Matrix A. Dieses Vorgehen ist bei den meisten der in der Anwendung auftretenden Probleme relativ aufwendig, weshalb man besser ein geeignetes Verfahren zum Lösen des linearen Gleichungssystems wählt.
In beiden Fällen wird beim klassischen Automatischen Differenzieren der verwendete Algorithmus Operation für Operation abgeleitet, um so die Gleichung (2.35) mit zu lösen. Dieses Vorgehen ist für die Recheneffizienz nicht optimal. Aus diesem Grund bietet es sich hier an, gezielt den Algorithmus von Hand zu ändern. Wir betrachten nun die Gleichung (2.35) im Detail,

$$(\mathsf{A}\mathbf{x})' = \mathsf{A}'\mathbf{x} + \mathsf{A}\mathbf{x}' = \mathbf{b}'. \tag{2.36}$$

Wenn wir zuerst die Gleichung (2.34) mit einem geeigneten Verfahren lösen, sehen wir, dass wir die gesuchte Größe $\mathbf{x}'$ in Gleichung (2.36) sehr einfach durch erneutes Lösen des linearen Gleichungssystemes mit einer neuen rechten Seite erhalten,

$$\mathsf{A}x' = b' - \mathsf{A}'x. \tag{2.37}$$

Im Anschluss an diese mathematisch äquivalente Umformulierung betrachten wir die Recheneffizienz beider Vorgehen und gehen auch auf die numerischen Eigenschaften beider Varianten ein.

Bemerkung 2.2.12 (Lineares Gleichungssystem AD)
Wir betrachten das lineare Gleichungssystem (2.34) und das zugehörige abgeleitete Problem in seiner umgeformten Version (2.37).
Die Komplexität zum Lösen des Problems (2.34) bezeichnen wir mit $\mathcal{O}(\mathcal{C})$, wobei $\mathcal{C}$ abhängig von dem gewählten Lösungsverfahren ist. So gilt beispielsweise für eine vollbesetzte Matrix $\mathsf{A} \in \mathbb{R}^{N\times N}$ mit dem Gauß-Algorithmus als Lösungsmethode die Komplexität $\mathcal{C} = N^3$.

Für das Lösen des zugehörigen Problemes (2.37) haben wir die gleiche Komplexität zuzüglich der Bestimmung der rechten Seite $b' - \mathsf{A}'x$, das heißt es kommt eine Matrix-Vektor-Multiplikation und eine Vektor-Vektor-Addition dazu. Dies kann man im Worst-Case mit $\mathcal{O}(N^2)$ für $\mathsf{A}, \mathsf{A}' \in \mathbb{R}^{N\times N}$ abschätzen.

Diesem Ansatz stellen wir das operationsweise abgeleitete Vorgehen gegenüber. Als verwendeten Löser für das Lineare Gleichungssystem betrachten wir das CG-Verfahren und GMRES-Verfahren.

Bemerkung 2.2.13 (Lineares Gleichungssystem Operationsweise)
Wie in Bemerkung 2.2.12 betrachten wir das Problem (2.34). Als Löser betrachten wir zuerst das CG-Verfahren und anschließend das GMRES-Verfahren.

- **CG**
  Das CG-Verfahren ist ein Verfahren zum Lösen von linearen Gleichungssystemen bei denen die Matrix A Symmetrisch Positiv Definit (s.p.d.) ist. Eine detailierte Betrachtung und Analyse des CG-Verfahrens findet man zum Beispiel in dem Buch Numerik linearer Gleichungssysteme von C. Kanzow [Kan00] oder in Numerik linearer Gleichungssysteme von A. Meister [Mei05].

  In Algorithmus 2.2.14 sehen wir den Algorithmus des CG-Verfahrens. Denselben Algorithmus in abgeleiteter Version sehen wir im Algorithmus 2.2.15. Der Rechenaufwand pro Schleifendurchlauf in Algorithmus 2.2.14 setzt sich aus einer Matrix-Vektor-Multiplikation, zwei Skalarprodukten, drei Skalar-Vektor-Multiplikationen, drei Vektor-Vektor-Additionen und zwei Skalar-Operationen zusammen. Im Allgemeinen ist die Matrix-Vektor-Multiplikation der Hauptaufwand. Im Falle, dass die Matrix A dünnbesetzt (sparse) ist, spielen die Vektor-Operationen auch eine Rolle. Damit ergibt sich folgender Gesamtaufwand: $\mathcal{O}(N^2 + 10N)$.

  Der Rechenaufwand des abgeleiteten CG-Verfahrens ergibt sich aus Algorithmus 2.2.15 mit zwei Matrix-Vektor-Multiplikationen, drei Skalarprodukten, sieben Vektor-Vektor-Additionen, sechs Skalar-Matrix-Multiplikationen und 13 Skalar-Operationen. Damit ergibt sich ein zusätzlicher Gesamtaufwand von $\mathcal{O}(2N^2 + 19N)$.

  Daraus folgt, dass der Rechenaufwandzum Lösen des umformulierten Problems der Gleichungen (2.34) und (2.35) ungefähr 1,5-fach geringer ist, als das lösen der Gleichungen (2.34) und (2.35) in der operationsweisen Formulierung des CG-Verfahrens.

Algorithm 2.2.14 CG Verfahren

wähle $\mathbf{x}^0 \in \mathbb{R}^N$;
$\mathbf{r}^0 := \mathsf{A}\mathbf{x}^0 - \mathbf{b}$;
$\mathbf{d}^0 := -\mathbf{r}^0$;
for $i = 0, 1, 2, \ldots$ **do**
  $\alpha^i := \frac{\|\mathbf{r}^i\|^2}{(\mathbf{d}^i)^T \mathsf{A}\mathbf{d}^i}$;
  $\mathbf{x}^{i+1} := \mathbf{x}^i + \alpha^i \mathbf{d}^i$;
  $\mathbf{r}^{i+1} := \mathbf{r}^i + \alpha^i \mathsf{A}\mathbf{d}^i$;
  $\beta^i := \frac{\|\mathbf{r}^{i+1}\|^2}{\|\mathbf{r}^i\|^2}$;
  $\mathbf{d}^{i+1} := -\mathbf{r}^{i+1} + \beta^i \mathbf{d}^i$;
end for

- **GMRES**
  Das GMRES-Verfahren wurde 1986 von Saad und Schutz [SS86] vorgestellt. Das Verfahren löst für jede reguläre Matrix A *das dazugehörige Gleichungssystem* (2.34)*. Herleitung und Analyse zu dem Verfahren findet sich wie für das CG-Verfahren in den Büchern von C. Kanzow [Kan00] und A. Meister [Mei05]. Für die Rechenkomplexitätsanalyse vergleichen wir die Algorithmen 2.2.16 und 2.2.17 zeilenweise. Wenn man die wesentlichen Operationen zusammenfasst, kommt man wie beim CG-Verfahren, auf einen zusätzlichen Aufwand, der zweimal dem des ursprünglichen Verfahrens entspricht.*

Algorithm 2.2.15 CG-Verfahren Abgeleitet

$\mathbf{x}^{D\,0} := \mathbf{0}$;
$\mathbf{r}^{D\,0} := \mathsf{A}^D \mathbf{x}^0 - \mathbf{b}^D$;
$\mathbf{d}^{D\,0} := -\mathbf{r}^{D\,0}$;
for $i = 0, 1, 2, \ldots$ **do**
  $\alpha^{D\,i} := \frac{2<\mathbf{r}^{D\,i}, \mathbf{r}^i>(\mathbf{d}^i)^T \mathsf{A}\mathbf{d}^i) - \|\mathbf{r}^i\|^2 \left((\mathbf{d}^{D\,i})^T \mathsf{A}\mathbf{d}^i + (\mathbf{d}^i)^T (\mathsf{A}^D \mathbf{d}^i + \mathsf{A}\mathbf{d}^{D\,i})\right)}{((\mathbf{d}^i)^T \mathsf{A}\mathbf{d}^i)^2}$;
  $\mathbf{x}^{D\,i+1} := \mathbf{x}^{D\,i} + \alpha^{D\,i}\mathbf{d}^i + \alpha^i \mathbf{d}^{D\,i}$;
  $\mathbf{r}^{D\,i+1} := \mathbf{r}^{D\,i} + \alpha^{D\,i}\mathsf{A}\mathbf{d}^i + \alpha^i(\mathsf{A}^D \mathbf{d}^i + \mathsf{A}\mathbf{d}^{D\,i})$;
  $\beta^{D\,i} := 2\frac{<\mathbf{r}^{D\,i+1}, \mathbf{r}^{i+1}>\|\mathbf{r}^i\|^2 - \|\mathbf{r}^{i+1}\|^2 <\mathbf{r}^{D\,i}, \mathbf{r}^i>}{\|\mathbf{r}^i\|^4}$;
  $\mathbf{d}^{D\,i+1} := -\mathbf{r}^{D\,i+1} + \beta^{i\,D}\mathbf{d}^i + \beta^i \mathbf{d}^{D\,i}$;
end for

Zusammengefasst können wir festhalten, dass sich der Aufwand für das Umstellen der Gleichung lohnt. Ein weiterer Vorteil ist hierbei, dass wir die Eigenschaften der Matrix A kennen und gegebenenfalls den schon vom ursprünglichen Problem bekannten Vorkonditionierer wieder verwenden können.

Der nächste Abschnitt behandelt die im Rahmen dieser Dissertation entwickelte Sammlung von AD-Tools im Projekt GAuDi.

Algorithm 2.2.16 GMRES-Verfahren

wähle $\mathbf{x}^0 \in \mathbb{R}^N$;
wähle $\epsilon \geq 0$;
$\mathbf{r}^0 := \mathbf{b} - \mathsf{A}\mathbf{x}^0$;
$\beta := \|\mathbf{r}^0\|$;
$\mathbf{v}^1 := {}^{\mathbf{r}^0}\!/_{\beta}$;
$z_1 := \beta$;
for $i = 1, 2, \ldots$ **do**
  $\mathbf{w}^i := \mathsf{A}\mathbf{v}^i$;
  for $j = 1 : i$ **do**
    $h_{j,i} := (\mathbf{v}^j)^T \mathbf{w}^i$;
    $\mathbf{w}^i := \mathbf{w}^i - h_{j,i}\mathbf{v}^i$;
  end for
  $h_{i+1,i} := \|\mathbf{w}^i\|$;
  $\mathbf{v}^{i+1} := {}^{\mathbf{w}^i}\!/_{h_{i+1,i}}$;
  for $j = 1 : (i-1)$ **do**
    $\begin{pmatrix} h_{j,i} \\ h_{j+1,i} \end{pmatrix} := \begin{pmatrix} c_j & s_j \\ -s_j & c_j \end{pmatrix} \begin{pmatrix} h_{j,i} \\ h_{j+1,i} \end{pmatrix}$;
  end for
  $\tau := |h_{i,i}| + |h_{i+1,i}|$;
  $\nu := \tau\sqrt{(h_{i,i}/\tau)^2 + (h_{i+1,i}/\tau)^2}$;
  $c_i := {}^{h_{i,i}}\!/_{\nu}$;
  $s_i := {}^{h_{i+1,i}}\!/_{\nu}$;
  $h_{i,i} := \nu$;
  $h_{i+1,i} := 0$;
  $z^{i+1} := -s_i z^i$;
  $z^i := c_i z^i$;
  if ${}^{|z^{i+1}|}\!/_{\beta} \leq \epsilon$ **then**
    STOP;
  end if
end for
$y_i := {}^{z_i}\!/_{h_{i,i}}$;
for $j = (i-1) : 1$ **do**
  $y^j := \left(z^j - \sum_{k=j+1}^{i} h_{j,k} y^k\right) /_{h_{j,j}}$;
end for
$\mathbf{x}^i := \mathbf{x}^0 + \sum_{j=1}^{i} y^j \mathbf{v}^j$;

Algorithm 2.2.17 GMRES-Verfahren Abgeleitet

wähle $\mathbf{x}^{D\,0} = \mathbf{0}$;
$\mathbf{r}^{D\,0} := \mathbf{b}^D - \mathsf{A}^D\mathbf{x}^0$;
$\beta^D := 2 < \mathbf{r}^{D\,0}, \mathbf{r}^0 >$;
$\mathbf{v}^1 := (\mathbf{r}^{D\,0}\beta - \mathbf{r}^0\beta^D)/\beta^2$;
$z^D{}_1 := \beta^D$;
for $i = 1, 2, \ldots$ **do**
  $\mathbf{w}^{D\,i} := \mathsf{A}^D\mathbf{v}^i + \mathsf{A}\mathbf{v}^{D\,i}$;
  for $j = 1 : i$ **do**
    $h^D{}_{j,i} := (\mathbf{v}^{D\,j})^T\mathbf{w}^i - (\mathbf{v}^j)^T\mathbf{w}^{D\,j}$;
    $\mathbf{w}^{D\,i} := \mathbf{w}^{D\,i} - h^D{}_{j,i}\mathbf{v}^i - h_{j,i}\mathbf{v}^{D\,i}$;
  end for
  $h^D{}_{i+1,i} := < \mathbf{w}^{D\,i}, \mathbf{w}^i > /\|\mathbf{w}^i\|$;
  $\mathbf{v}^{D\,i+1} := \mathbf{w}^{D\,i}h_{i+1,i} - \mathbf{w}^i h^D{}_{i+1,i}/h^2_{i+1,i}$;
  for $j = 1 : (i-1)$ **do**
    $\begin{pmatrix} h^D{}_{j,i} \\ h^D{}_{j+1,i} \end{pmatrix} := \begin{pmatrix} c^D{}_j & s^D{}_j \\ -s^D{}_j & c^D{}_j \end{pmatrix} \begin{pmatrix} h_{j,i} \\ h_{j+1,i} \end{pmatrix} + \begin{pmatrix} c_j & s_j \\ -s_j & c_j \end{pmatrix} \begin{pmatrix} h^D{}_{j,i} \\ h^D{}_{j+1,i} \end{pmatrix}$;
  end for
  $\tau^D := |h_{i,i}|^D + |h_{i+1,i}|^D$;
  $\nu^D := \tau^D\sqrt{(h_{i,i}/\tau)^2 + (h_{i+1,i}/\tau)^2} + \frac{\tau((2h^D{}_{i,i}h_{i,i}\tau^2 - h^2_{i,i}2\tau^D\tau)/\tau^4 + 2h^D{}_{i+1,i}h_{i+1,i}\tau^2 - h^2_{i+1,i}2\tau^D\tau)/\tau^4)}{\sqrt{(h_{i,i}/\tau)^2 + (h_{i+1,i}/\tau)^2}}$;
  $c^D{}_i := (h^D{}_{i,i}\nu - h_{i,i}\nu^D)/\nu^2$;
  $s^D{}_i := (h^D{}_{i+1,i}\nu - h_{i+1,i}\nu^D)/\nu^2$;
  $h^D{}_{i,i} := \nu^D$;
  $h^D{}_{i+1,i} := 0$;
  $z^{D\,i+1} := -s^D{}_i z^i - s_i z^{D\,i}$;
  $z^{D\,i} := c^D{}_i z^i + c_i z^{D\,i}$;
  if $|z^{i+1}|/\beta \leq \epsilon$ **then**
    STOP;
  end if
end for
$y^D{}_i := {}^{z^D{}_i h_{i,i} - z_i h^D{}_{i,i}}/h^2_{i,i}$;
for $j = (i-1) : 1$ **do**
  $y^{D\,j} := \left(\left(z^{D\,j} - \sum_{k=j+1}^{i} h^D{}_{j,k}y^k + h_{j,k}y^{D\,k}\right)h_{j,j} - \left(z^j - \sum_{k=j+1}^{i} h_{j,k}y^k\right)h^D{}_{j,j}\right)/h^2_{j,j}$;
end for
$\mathbf{x}^{D\,i} := \mathbf{x}^{D\,0} + \sum_{j=1}^{i} y^{D\,j}\mathbf{v}^j + y^j\mathbf{v}^{D\,j}$;

2.3 GAuDi

Das GAuDi-Projekt steht für *GNU Automatic Differentiation* und ist eine Sammlung von AD-Tools, die im Rahmen dieser Dissertation entwickelt wurden. GAuDi umfasst zwei Hauptgruppen: Ein Element ist ein Source-Transformations-Tool, das einen möglichst programmiersprachenunabhängigen Ansatz verfolgt, das zweite Element ist eine C++-Bibliothek, die den Ansatz der Überladenen Operatoren verfolgt und eine Schnittstelle zum verbesserten Debugging bereitstellt.

2.3.1 Source-Transformation

Das AD-Tool, das wir in diesem Abschnitt vorstellen, wurde im Rahmen der Promotion entwickelt, um einen neuen Ansatz im Bereich der Source-Transformations-Tools zu erproben. Es gibt schon einige Tools zur Source-Transformation, wie zum Beispiel Tapenade [HP04] oder OpenAD [UNL11]. Zu dem Tool-Tapenade grenzt sich das hier entwickelte Tool durch seinen sprachunabhängigen Ansatz ab. OpenAD verfolgt zwar auch einen sprachunabhängigen Ansatz, allerdings muss hier das zur Quellcode-Analyse verwendete *Front End* ebenfalls selbst erstellt werden. Wir verwenden dafür den `gcc`.
Die vollständige Implementierung des GAuDi-Source-Transformations-Tools ist auf der beiliegenden CD im Verzeichnis `Programme/gaudi/adbin` zu finden. In diesem Verzeichnis finden sich neben einer Anleitung für die Verwendung auch einige Demo-Programme.

Vorgehensweise

Der Kerngedanke, der bei der Entwicklung des GAuDi-Source-Transformations-Tools (GAuDi-STT) zugrundeliegt ist das Prinzip, dass man bestehende und ausgereifte Strukturen verwendet, um die Gesamtkomplexität der Entwicklung zu reduzieren.
Man kann die Source-Transformation in vier Teilaufgaben zerlegen:

- Quellcode-Analyse,
- Abhängigkeitsanalyse,
- Differenzieren und
- Quellcode-Erzeugung.

Diese vier Teilaspekte betrachten wir nun etwas näher, bevor wir im Abschnitt *Realisierung* auf die Aspekte der technischen Umsetzung zu Sprechen kommen.

Quellcode-Analyse

Diese Aufgabe ist der zentrale sprachabhängige Teil der Source-Transformation. An diesem Punkt müssen wir ansetzen, um die größtmögliche Sprachunabhängigkeit zu erreichen. Hierzu setzen wir bei dem GAuDi-STT auf den `gcc` und verwenden ihn als eine Art Präprozessor, um eine sprachunabhängige Darstellung unseres zu differenzierenden Programmes zu erzeugen.
Weiter muss die sprachunabhängige Programm-Representation eine Darstellung aufweisen, die es uns im nächsten Schritt - der Abhängigkeitsanalyse - erlaubt, den Graphen, der die

Abhängigkeit zwischen der Zielvariablen und Eingangsvariablen bestimmt, leicht aufzustellen.

Abhängigkeitsanalyse

Die Analyse zur Abhängigkeit zwischen der Zielvariablen und der Eingangsvariablen ist der Grundbaustein für das anschließende Differenzieren. In diesem Schritt wird ausgehend von der Zielvariablen rückwärts durch das Programm gegangen und ein Graph aufgebaut, der die Abhängigkeit dieser Variablen von den anderen Variablen und den entsprechenden Elementar-Operationen nachzeichnet.
Dieser Graph wird noch von den aktiven Eingangsvariablen aus *eingefärbt* um darzustellen, welche Bereiche des Graphen im Sinne des Automatischen Differenzierens aktiv sind.
Eine Funktion, die das GAuDi-STT zur Verfügung stellt, ist eine Liste aller abhängigen Variablen. Diese Funktion ist bei großen Softwareprojekten sehr interessant, da man anhand der Liste mit einem moderaten Aufwand auch das Prinzip der Überladenen Operatoren anwenden kann, weil die Untersuchung, welche Variablen von der Eingangsvariablen abhängen und auch wieder Einfluss auf die Zielvariable haben, entfällt.

Differenzieren

Das Kernstück der Source-Transformation ist das Differenzieren des Abhängigkeitsgraphen bezüglich der angegebenen Variablen. Dieser Vorgang ist vollständig sprachunabhängig, da das Differenzieren auf der Ebene der toolinternen Darstellung geschieht.
Dafür müssen für alle elementaren Rechenoperationen die Ableitungsregeln implementiert sein. Zudem müssen Routinen zur Manipulation des Graphen existieren.

Quellcode-Erzeugung

Dies ist der zweite programmiersprachenabhängige Teil der Source-Transformation. Allerdings ist dieser Teil relativ leicht realisierbar, da man sich auf einen sehr kleinen Bereich der Programmiersprache beschränken kann. Folgende Aufgaben muss diese Einheit erledigen:

- Abgeleitete Zeilen erzeugen,
- neue Variablen anlegen und initialisieren,
- neue Funktionsköpfe anlegen und
- neue Funktionsaufrufe erzeugen.

Hierbei ist das Erzeugen der abgeleiteten Zeilen *relativ sprachunabhängig*, da bei den meisten gängigen Programmiersprachen die *reinen Rechenzeilen* einem sehr ähnlichen Aufbau folgen. Beim Anlegen und Initialisieren der neuen Variablen folgen die Sprachen zwar einem ähnlichen Schema, allerdings sind gerade bei den Namen der Datentypen mehr Unterschiede vorhanden. Die größten Unterschiede liegen beim Aufrufen der neuen Funktionen und dem Erzeugen der Funktionsköpfe.

Realisierung

Nun betrachten wir die Umsetzung des oben beschrieben Vorgehens. Das GAuDi-STT ist in der Programmiersprache Perl geschrieben. Die Wahl ist auf die Programmiersprache Perl ge-

fallen, da diese gut geeignet ist, um Text-Elemente mittels regulären Ausdrücken (engl. *regular expression*) siehe [Bir11], [McK10] und [Tru05], zu bearbeiten.
Den inneren Ablauf vom GAuDi-STT sehen wir in Figur 2.5.

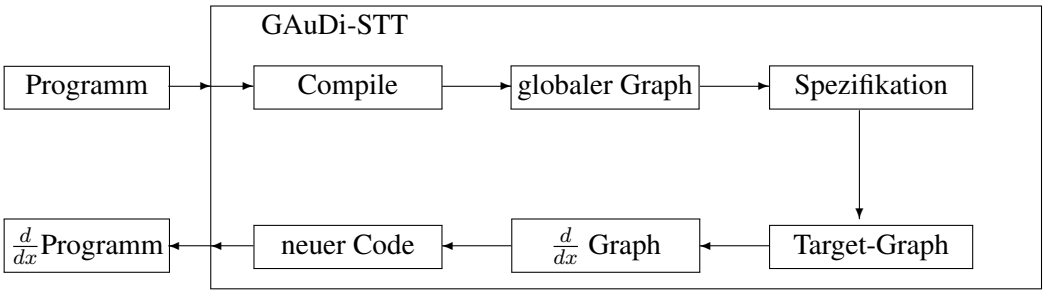

Abbildung 2.5: Schema des inneren Ablauf vom GAuDi-STT.

Wir gehen auf die einzelnen Zwischenschritte vom GAuDi-STT nun im Detail ein. Die *Quellcode-Analyse* des vorherigen Abschnittes korrespondiert mit dem Block *Compile*. Zur *Abhängigkeitsanalyse* gehören *globaler Graph*, *Spezifikation* und *Target-Graph*. Der Block $\frac{d}{dx}$ *Graph* entspricht dem *Differenzieren* und der letzte Schritt *Quellcode-Erzeugung* wird durch den Block *neuer Code* vertreten.

Compile

Den sprachabhängigen Quellcode müssen wir zuerst in ein Format übersetzen, das unabhängig von der Programmiersprache ist. Als sprachunabhängige Darstellung haben wir das CFG[10]-Format vom `gcc` gewählt.
Die Erzeugung des CFG-Code aus dem ursprünglichen Programmtext ist mit dem `gcc` realisiert. Das CFG-Format ist eine interne Representationsstruktur des `gcc`. Eine ausführliche Beschreibung ist zum Beispiel in *GNU Compiler Collection Internals* von Richard Stallman and the GCC Developer Community [SGDC08] zu finden. Der Vorteil dieses Formates ist, dass es mit dem `gcc` für die meisten gängigen Programmiersprachen erzeugt werden kann, zum Beispiel C, C++, Fortran und Java. Ein weiterer Vorteil ist, dass Ausdrücke in eine Tupel-Darstellung von maximal drei Operanden zerlegt werden. Wir werden nun das CFG Format anhand einiger kleiner Beispiele erläutern. Als zugrundeliegende Literatur haben wir uns neben dem bereits erwähnten Werk von Stallman [SGDC08] noch auf *Tree SSA A New Optimization Infrastructure for GCC* von Diego Novillo [Nov03] gestützt.
Mit der Option `-fdump-tree-cfg` erzeugt man mittels des `gcc` das CFG Format aus dem zugrundeliegenden Sourcecode. Damit das Erzeugen des neuen Quellcode am Ende des Differentiationsprozesses erleichtert wird, erweitern wir die Option zu `-fdump-tree-cfg-uid-lineno`. Dadurch erhalten wir zusätzliche Information zur ursprünglichen Datei und Zeilennummer.

Beispiel 2.3.1 (CFG)
Wir betrachten das Folgende übersichtliche C Programm, siehe Listing 2.11 und seine Darstellung im CFG Format, Listing 2.12.

[10] CFG ist die Abkürzung von Control Flow Graph.

Listing 2.11: CFG Beispielprogramm in C.

```
1  #include <cmath>
2
3  double fct1(double x1, double x2){
4      double tmp=x2;
5      for(int i=0; i<10; ++i){
6          tmp /= x1;
7      }
8      return tmp;
9  }
10
11 int main(){
12     double f1,f2, x1,x2;
13     x1 =1.;
14     x2=2.;
15     f1 = x1*x2+x1/sin(x2);
16     f2 = fct1(x1,x2);
17     return 0;
18 }
```

Im Listing 2.11 sehen wir in der `main`*-Routine zwei interessante Zeilen. In Zeile 15 wird die Variable* `x` *auf den Wert des ausgewerteten Ausdrucks gesetzt. In Zeile 16 wird der Variablen* `f2` *der Rückgabewert der Funktion* `fct1` *zugewiesen. Unser Augenmerk liegt aber auf der Darstellung in der CFG Form, die wir in Listing 2.12 sehen.*

Listing 2.12: CFG Beispielprogramm im CFG Format.

```
1
2  ;; Function int main() (main)
3
4  Removing basic block 3
5  Merging blocks 2 and 4
6  int main() ()
7  {
8    doubleD.35 x2D.4596;
9    doubleD.35 x1D.4595;
10   doubleD.35 f2D.4594;
11   doubleD.35 f1D.4593;
12   intD.2 D.4601;
13   doubleD.35 D.4600;
14   doubleD.35 D.4599;
15   doubleD.35 D.4598;
16   doubleD.35 D.4597;
17
18   # BLOCK 2, starting at line 13
19   # PRED: ENTRY (fallthru)
20   [cfg.c : 13] x1D.4595 = 1.0e+0;
21   [cfg.c : 14] x2D.4596 = 2.0e+0;
22   [cfg.c : 15] D.4597 = x1D.4595 * x2D.4596;
23   [cfg.c : 15] D.4598 = [cfg.c : 15] sin (x2D.4596);
24   [cfg.c : 15] D.4599 = x1D.4595 / D.4598;
25   [cfg.c : 15] f1D.4593 = D.4597 + D.4599;
26   [cfg.c : 16] D.4600 = [cfg.c : 16] fct1 (x1D.4595, x2D.4596);
27   [cfg.c : 16] f2D.4594 = D.4600;
28   [cfg.c : 17] D.4601 = 0;
29   return D.4601;
30   # SUCC: EXIT
31 }
32
33 ;; Function double fct1(double, double) (_Z4fct1dd)
34
35 Merging blocks 5 and 6
36 double fct1(double, double) (x1D.4580, x2D.4581)
37 {
38   intD.2 iD.4585;
39   doubleD.35 tmpD.4584;
40   doubleD.35 D.4590;
41
42   # BLOCK 2, starting at line 4
43   # PRED: ENTRY (fallthru)
44   [cfg.c : 4] tmpD.4584 = x2D.4581;
45   [cfg.c : 5] iD.4585 = 0;
46   [cfg.c : 5] goto <bb 4> (<L1>);
47   # SUCC: 4 (fallthru)
48
```

```
49    # BLOCK 3, starting at line 6
50    # PRED: 4 (true)
51  <L0>:;
52    [cfg.c : 6] tmpD.4584 = tmpD.4584 / x1D.4580;
53    [cfg.c : 5] iD.4585 = iD.4585 + 1;
54    # SUCC: 4 (fallthru)
55
56    # BLOCK 4, starting at line 5
57    # PRED: 2 (fallthru) 3 (fallthru)
58  <L1>:;
59    [cfg.c : 5] if (iD.4585 <= 9) [cfg.c : 5] goto <L0>; else [cfg.c : 5] goto <L2>;
60    # SUCC: 3 (true) 5 (false)
61
62    # BLOCK 5, starting at line 8
63    # PRED: 4 (false)
64  <L2>:;
65    [cfg.c : 8] D.4590 = tmpD.4584;
66    return D.4590;
67    # SUCC: EXIT
68  }
```

Vier Punkte fallen direkt bei der Betrachtung von Listing 2.12 auf: Der Programmfluss wird in Blöcke unterteilt, vor den Statements stehen in einem festen Format Informationen zum Dateinamen des Quellcodes zur ursprünglichen Zeilennummer, jeder Ausdruck ist in sogenannter SSA[11] Form und Schleifen werden durch if-Bedingungen und Sprungmarken repräsentiert.
Nun betrachten wir den Aufbau der CFG-Datei etwas näher und beginnen mit den Blöcken. Jeder Block hat drei Zeilen, die die Blockdaten enthalten. Diese Zeilen beginnen immer mit einem `#` *als erstes nicht whitespace-Zeichen, gefolgt von einem Schlüsselwort* `BLOCK`, `PRED:` *oder* `SUCC:`*. Die Blockstruktur ist wie folgt aufgebaut: Zuerst kommen die* `BLOCK` *und* `PRED:` *Zeilen, eventuell gefolgt von einer Sprungmarken-Zeile und den Block Statements. Abgeschlossen wird der Block mit der* `SUCC:` *Zeile. Die* `BLOCK` *Zeile nennt die Blocknummer und die Zeilennummer der ersten Blockzeile im Quellcode.* `PRED:` *steht für predecessor, also den Vorgänger Block und gibt an von welchem, beziehungsweise auch welchen, Blöcken der aktuelle Block aufgerufen wurde. Zuletzt gibt* `SUCC:` *für successor den Nachfolger an, also welcher Block im Anschluss aufgerufen wird. Dies kann von einem if-Statement abhängen.*
Ein Statement aus dem ursprünglichen Quellcode wird in sogenannte SSA-Anweisungen zerlegt. Diese Darstellung hat für uns den Vorteil, dass wir uns um die Priorität der einzelnen Operationen innerhalb eines Statements keine Gedanken machen müssen, da im CFG-Format ein Statement genau eine Operation besitzt. Zudem enthalten die einzelnen Statements Informationen zu welcher Zeile des Ausgangsquellcode sie gehören und aus welcher Datei sie stammen, falls der Quellcode aus mehreren Dateien besteht. Informationen zu SSA (Static Single Assignment) findet man in [SGDC08] und Translating Out of Static Single Assignment Form von Sreedhar [SJGS99].
Schleifen und Verzweigungen innerhalb des Programmes werden immer durch ein `if`*-Statement mit* `else`*-Fall und* `goto` *Anweisung im CFG Format dargestellt. Diese Form ist zur Verarbeitung in einem Programm sehr gut geeignet.*
Damit haben wir die meisten Eigenschaften des CFG Formats die wir benötigen auch schon beschrieben. Der vollständige Quellcode zum Beispiel findet sich auf der CD im Verzeichnis `Programme/ex:CFG`.

Mit dem Beispiel 2.3.1 haben wir einen guten Überblick über die Erzeugung und den Aufbau des CFG erhalten. Im nächsten Schritt beschäftigen wir uns mit der Weiterverarbeitung der CFG-Datei für unsere Zwecke, so dass wir dann unseren Abhängigkeitsgraphen aufbauen können.

[11] Static Single Assignment

Globaler Graph

Dieser Schritt ist bereits unabhängig von der ursprünglichen Programmiersprache. Zuerst wird das Programm auf Rekursion untersucht. Wenn eine Rekursion gefunden wird, kann man entscheiden, wie man weiter vorgehen möchte. Folgende Optionen stehen uns zur Verfügung: GAuDi-STT macht einen Vorschlag an welcher Stelle die Rekursion durchbrochen wird oder wir geben von Hand an, wo wir die Rekursion beenden wollen. Eine weitere Möglichkeit besteht darin dass wir die Rekursion vor dem Anwenden des Tools auflösen. Dies kann unter Umständen zu einem effizienteren Code führen.
Als nächstes werden die CFG-Dateien in die einzelnen Blöcke zerlegt. Dann werden die Abhängigkeitsverhältnisse der einzelnen Blöcken zueinander analysiert und die Abhängigkeit zwischen den Funktionen wird aufgestellt. Als letztes wird ein Graph aufgestellt, der die Abhängigkeit aller Variablen darstellt. Den Algorithmus für die Erstellung des globalen Graphen betrachten wir nun in Algorithmus 2.3.2.

Algorithm 2.3.2 Globaler Graph

for all $e \in E$ **do**
  for $i = 0; i < \#e_{RHS}$; ++i **do**
    if $e_{RHS(i)_{POS}} = activated$ **then**
      continue;
    end if
    $B := \{e_{block}\}$;
    while $B \neq \emptyset$ **do**
      $b \in B$;
      $B = B \setminus b$;
      for all $\tilde{e} \in E$ *and* $\tilde{e}_{block} = b$ **do**
        if $\tilde{e}_{LHS} = e_{RHS(i)}$ **then**
          $e_{RHS(i)_{POS}} = \tilde{e}_{POS}$;
          break;
        end if
      end for
      if $e_{RHS(i)_{POS}} \neq activated$ **then**
        for $j = 1; j \leq \#b_{previous}$; ++$j$ **do**
          $B = B \cup \{b_{previous(j)}\}$;
        end for
      end if
    end while
  end for
end for

Bemerkung 2.3.3 (Komplexität Globaler Graph)
Eine Worst-Case-Abschätzung des Algorithmus 2.3.2 zum Aufstellen des globalen Graphen können wir einfach mit

$$\mathcal{O}(|E|^2) \tag{2.38}$$

angeben. $|E|$ bezeichnet die Anzahl der Elemente in der Menge E.

Die Menge E enthält alle Expressions des Programms. Die Elemente e von der Menge E enthalten folgende Informationen: Eine Position e_{POS}, einen Block e_{block}, zu dem sie gehören, ein LHS-Argument e_{LHS}, eine Expression-Operation e_{expr}, die dazu gehörigen RHS-Argumente $e_{RHS(i)}$ und deren Positionen $e_{RHS(i)_{POS}}$. Zusätzlich gibt es noch Felder für die zugehörige Datei, die korrespondierende ursprüngliche Zeilennummer und ein Feld zum Speichern der Information ob der Expression relevant für die Ableitung ist.
Für jeden Block ist gespeichert von welchem Block beziehungsweise von welchen Blöcken, er aufgerufen werden kann. Im Falle, dass die Position des RHS-Arguments nicht im gleichen Block enthalten ist und der aktuelle Block mehrere Vorgänger besitzt, gehören zu dem RHS-Argument auch mehrere Positionen.

Beispiel 2.3.4 (Gloabler Graph)
In diesem Beispiel betrachten wir den globalen Graphen des in Listing 2.13 gegebenen Programms.

Listing 2.13: Beispielprogramm für Graphen.

```
1  int main()
2  {
3      double x,y,z;
4      double f,g,h;
5      x = 1.1;
6      y = 1.2;
7      z = 1.3;
8      f = x*x;
9      g = y*y;
10     h = x/z;
11     if (h>f) {
12         h *= g;
13     }else{
14         f /= g;
15     }
16     for(int i=1; i<10; ++i) {
17         g += x*z;
18         f -= y;
19         h += z;
20         z *= y;
21     }
22     return 0;
23 }
```

Das Programm aus Listing 2.13, ist kurz und übersichtlich. Die Darstellung des globalen Graphen ist allerdings schon an der Grenze der Möglichkeit des übersichtlichen Darstellens.
In der Figur 2.6 ist der globale Graph des Programms abgebildet.

Als nächstes betrachten wir die Spezifikation des Targets und des Differentiations-Parameters.

Spezifikation

Nun muss die zu differenzierende Zielgröße, das *Target* und der Differentiations-Parameter[12] ausgewählt werden. Hierzu erstellt das Tool eine Liste aller möglichen Targets und aller potentieller Differentiations-Parameter, aus der der Nutzer die Größen festlegt.
Potentielle Targets sind alle Variablen eines Gleitkomma-Type (float), denen ein Ausdruck zugewiesen wird. Als Differentiations-Parameter stehen alle Gleitkommazahlen, die als Argumente in einem Ausdruck vorkommen, zur Verfügung. Da in der CFG-Darstellung viele

[12]Der Differentiations-Parameter wird auch als Independent-Variable bezeichnet. Alle vom Differentiations-Parameter beeinflussten Variablen werden dann Dependent -Variablen genannt.

zusätzliche temporäre Variablen eingeführt werden, wird die Liste um diese bereinigt.

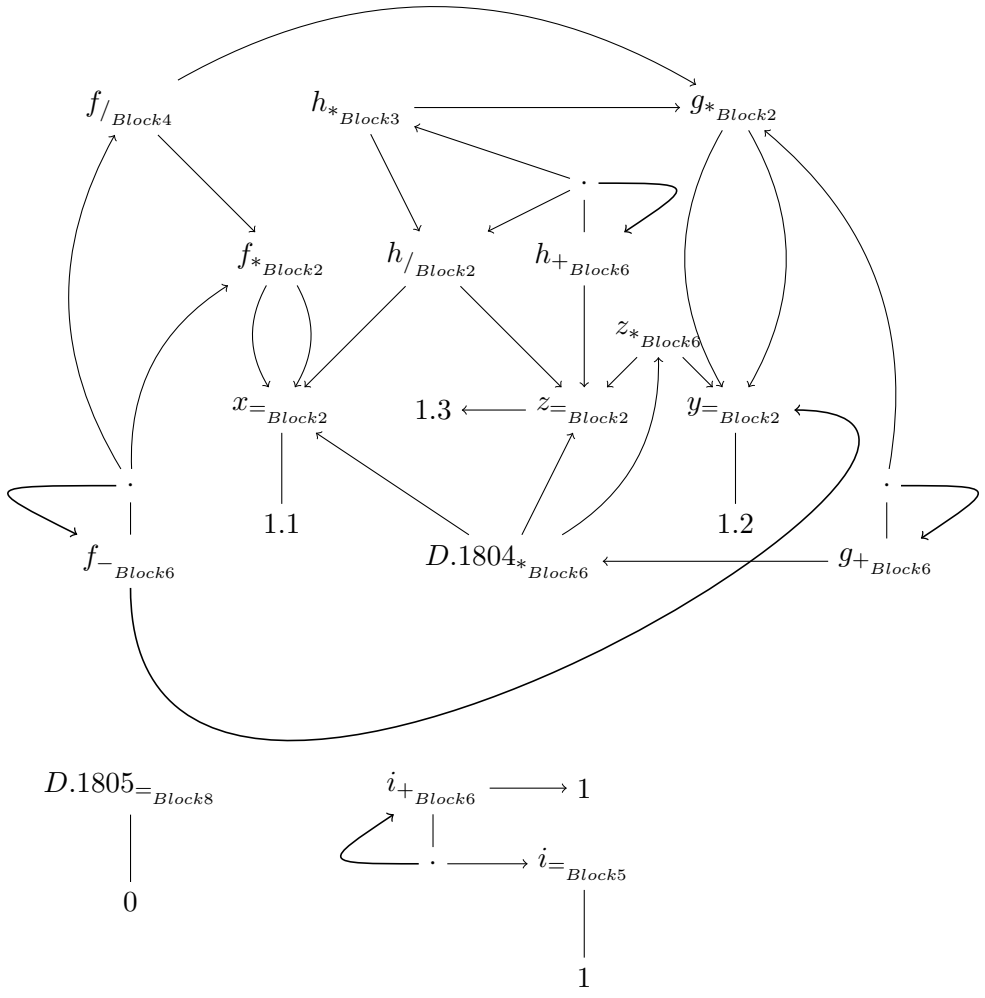

Abbildung 2.6: Graphendarstellung des Programms 2.13.

Target-Graph

Das Aufbauen des Target-Graphen passiert in zwei Schritten. Zuerst wird vom Target aus der relevante Bereich im Graphen ermittelt. Im zweiten Schritt wird dann von den Differentiations-Parameter aus der Graph markiert. So erhält man einen aktiven Bereich und einen oder mehrere passive Bereiche des Graphen. Unter dem aktiven Graphen versteht man den Teil des Graphen, der von den Differentiations-Parametern abhängig ist und auch einen Einfluss auf das Target hat. Die passiven Teile des Graphen sind aus Sicht der Differentiations-Parameter her konstant und spielen für die Differentiation nur eine untergeordnete Rolle.
Wir betrachten nun den Algorithmus 2.3.5 zum Aufbau des Target-Graphen und den Algorith-

mus 2.3.7 zur Aktivierung des Target-Graphen detailiert.
Im Algorithmus 2.3.5 bauen wir den Target-Graphen in einem zweistufigen Verfahren auf.

Algorithm 2.3.5 Target-Graph

```
T := ∅;
for all e ∈ G do
  if e_LHS = target then
    T = T ∪ {e};
  end if
end for
for all e ∈ T do
  for all ẽ ∈ e_RHS do
    if ẽ ≠ target then
      T = T ∪ G[ẽ_POS];
    end if
  end for
end for
```

Im ersten Schritt fügen wir zu dem, am Anfang noch leeren, Target-Graphen T alle Expressions des globalen Graphen G hinzu. Für dessen LHS-Argument gilt, dass es eine Target-Variable ist.
Im zweiten Schritt durchläuft der Algorithmus jedes Element e des vorläufigen Target-Graphen T. Von den Elementen e betrachten wir nun die RHS-Argumente und prüfen, ob sie ungleich der Target-Variablen sind, Ist dies der Fall, wird der zum RHS-Argument gehörende Expression-Eintrag des Graphen G dem Target-Graphen T zugefügt.

Bemerkung 2.3.6 (Aufwand Target-Graph)
Den Aufwand des Algorithmus 2.3.5 zum Aufstellen des Target-Graphen können wir mit

$$\mathcal{O}(|T_{Elemente}| \cdot |T_{RHS}|), \tag{2.39}$$

abschätzen. $|T_{Elemente}|$ *ist die Anzahl der Elemente des vollständigen Graphen* T *und* $|T_{RHS}|$ *gibt die maximale Anzahl an RHS-Argumente aller Elemente von* T*.*

Der Algorithmus 2.3.7 zur Aktivierung des Graphen arbeitet auch in zwei Stufen.
Die erste Stufe aktiviert alle Elemente e des Target-Graphen T, deren LHS-Argument ein Differentiations-Parameter ist.
Anschließend werden alle Elemente e des Graphen T_{new}, die von einem aktiven Element abhängen, aktiviert. Dieser Schritt wird so lange wiederholt, bis wir einen statischen Zustand erreicht haben. Es ist sichergestellt, dass dieser Zustand existiert, da jedes aktivierte Element aktiviert bleibt und wir einen endlichen Graphen haben. Es wäre auch möglich, das Einfärben des Graphen mit einer modifizierten Tiefensuche zu realisieren. Beim Auftreten von Zyklen wäre dies allerdings nicht ganz so einfach geworden, da das Markieren erst bei den Independent-Variablen beginnen soll. Zudem haben wir einen gerichteten Graphen, bei dem wir nicht von den Independent-Variablen aus losgehen konnten. Als mögliche Literatur zum Thema Tiefensuche und Graphen allgemein diente das Buch *Algoritmic* von U. Schöning [Sch01].

Algorithm 2.3.7 Graphen Aktivierung

```
T_old := T;
T_new := T;
for all e ∈ T_new do
  if e_LHS = independent then
    e_active = true;
  end if
end for
while T_new ≠ T_old do
  T_old = T_new;
  for all e ∈ T_new do
    for all ẽ ∈ e_RHS do
      if T_new[ẽ_POS]_active = true then
        e_active = true;
        break;
      end if
    end for
  end for
end while
T = T_new;
```

Bemerkung 2.3.8 (Aufwand Graphen-Aktivierung)
Da sich bei der Implementierung des GAuDi-STT um eine Methodenerprobung handelt, lag der Fokus nicht auf der internen Komplexität. Er lag stattdessen auf der Einfachheit der verwendeten Methoden. Die Komplexität des Aktivierens des Graphen kann für den Algorithmus 2.3.7, für den Worst-Case mit

$$\mathcal{O}(|T_{Elemete}| \cdot |T_{Tiefe}| \cdot |T_{RHS}|), \tag{2.40}$$

abgeschätzt werden, wobei $|T_{Elemente}|$ die Anzahl der Elemente im Graphen, $|T_{Tiefe}|$ die maximale Tiefe des Graphen und $|T_{RHS}|$ die maximale Anzahl an RHS-Argumente von allen Elementen des Graphen bezeichnet.

Als nächstes befassen wir uns mit dem Vorgehen beim Ableiten des Target-Graphen.

$\frac{d}{dx}$ Graph

Nachdem die ganze Vorarbeit erledigt ist, können wir uns nun dem Differenzieren widmen. Hierzu wird jeder Eintrag im Target-Graph betrachtet, und überprüft, ob er für die Differentiation relevant ist, das heißt ob der von einem der Differentiations-Parameter abhängig ist. Ist diese Eigenschaft verletzt, wird der Eintrag nicht weiter betrachtet. Ist er aber aktiviert, so wird abhängig vom Operator-Key differenziert. Gehört der Operator-Key zu einer der Elementar-Operationen, so ist das Vorgehen anhand einer internen Regel-Tabelle festgelegt. Diesen Fall werden wir gleich anhand des / Operator im Beispiel 2.3.9 näher erläutern. Anderenfalls gehört der Operator-Key zu einer selbst geschriebenen Funktion, beziehungsweise Routine. In diesem Fall muss man näher untersuchen wie der neue Funktionskopf aussehen muss, ein Beispiel dazu betrachten wir im Beispiel 2.3.10.

Beispiel 2.3.9 (Differentiation des Operators $/$)
Wir betrachten einen Ausdruck der Form

$$z = x/y. \tag{2.41}$$

Der Ausdruck (2.41) *in Form eines Graphen repräsentiert sehen wir in der Figur 2.7.*

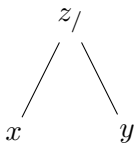

Abbildung 2.7: Graphendarstellung der Gleichung (2.41).

Für die Differentiation müssen wir die drei Fälle:

a) x *und* y *sind aktiv,*

b) nur x *ist aktiv oder*

c) nur y *ist aktiv,*

betrachten. Der Fall, dass x *und* y *nicht aktiv sind ist uninteressant, da dann auch* z *nicht als Expression aktiv wäre und wir den Expression auch nicht differenzieren würden.*

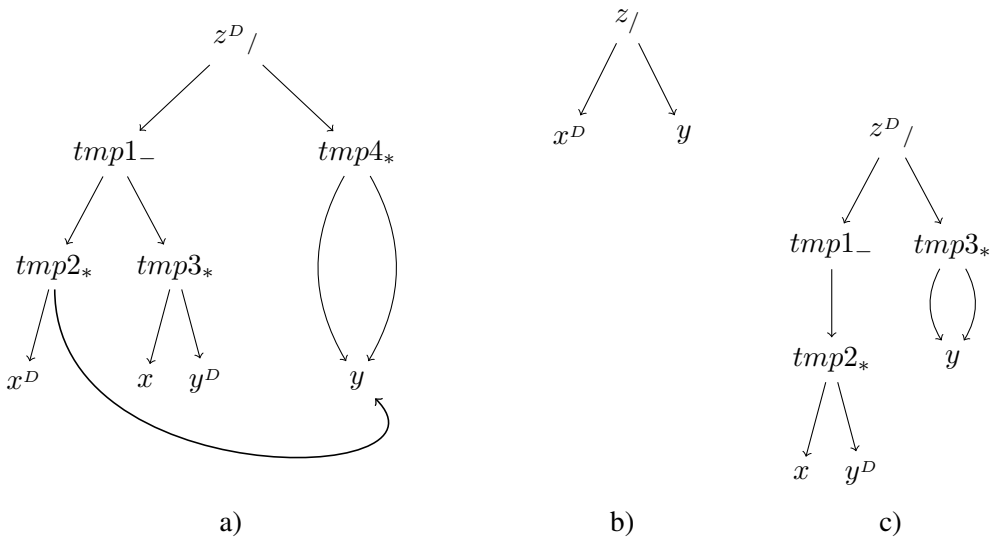

Abbildung 2.8: Darstellung des differenzierten Operators $/$.

In der Figur 2.8 sehen wir die Graphendarstellung der drei Varianten der differenzierten Gleichung (2.41)*. Daraus ergibt sich, dass der differenzierte Graph für ein* $/$*-Operator des Typs a) fünf neue Knoten,* z^D *und die temporären Knoten* $tmp1$*,* $tmp2$*,* $tmp3$ *und* $tmp4$ *erhält. Für*

den Fall b) kommt nur der Knoten z^D, und bei Fall c) kommen die vier Knoten: z^D, $tmp1$, $tmp2$ und $tmp3$ hinzu.
Die Graphen aus Figur 2.8 in einer Formel-Darstellung betrachtet sehen wir in Tabelle 2.3.9.

$tmp4 = y \cdot y$		
$tmp3 = x \cdot y^D$		$tmp3 = y \cdot y$
$tmp2 = x^D \cdot y$		$tmp2 = x \cdot y^D$
$tmp1 = tmp2 - tmp3$		$tmp1 = -tmp2$
$z^D = \frac{tmp1}{tmp2}$	$z^D = \frac{x^D}{y}$	$z^D = \frac{tmp1}{tmp2}$
a)	b)	c)

Tabelle 2.1: Formel-Darstellung des differenzierten $/$-Operators.

Eine Liste aller zu den Elementar-Operationen gehörender Operationen haben wir in Tabelle 2.2 zusammengestellt.
Nun betrachten wir im Beispiel 2.3.10 das Vorgehen bei einer selbst geschriebenen Routine.

Beispiel 2.3.10 (Differentiation einer selbstgeschriebenen Funktion)
Wir betrachten folgendes Ausgangsproblem,

$$z = f(a, x, y) \tag{2.42}$$

und

$$f(x, y) = a(x \cdot y). \tag{2.43}$$

Wir betrachten die Differentation der Funktion (2.42) *bezüglich x und y. Die Gleichungen* (2.42) *und* (2.43) *in der GAuDi-STT internen Graphendarstellung sind in der Figur 2.9 abgebildet. Wir sehen, dass es zwei Graphen gibt, nämlich für jede Funktion einen eigenen.*

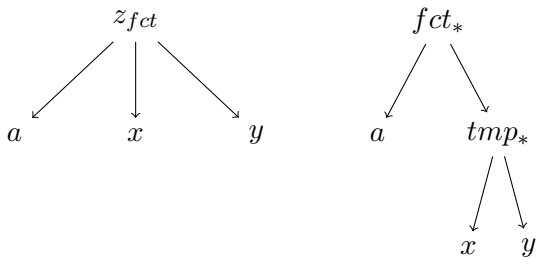

Abbildung 2.9: Graphendarstellung der Gleichungen (2.42) und (2.43).

Die differenzierte Form der Graphen in Figur 2.9 ist in Figur 2.10 dargestellt. Wie in der Figur 2.9 haben wir nun auch in der Abbildung 2.10 für jede Funktion des Programmes einen eigenen Graphen.

Operator	$y =$	$y' =$
$=$	x	${x}^{D}$
$+$	$x_1 + x_2$	${x_1}^{D} + {x_2}^{D}$
		${x_1}^{D}$
		${x_2}^{D}$
$+=$	$y + x$	${y}^{D} + {x}^{D}$
		${y}^{D}$
$-$	$x_1 - x_2$	${x_1}^{D} - {x_2}^{D}$
		${x_1}^{D}$
		$-{x_2}^{D}$
$-=$	$y - x$	${y}^{D} - {x}^{D}$
		${y}^{D}$
$*$	$x_1 * x_2$	${x_1}^{D} \cdot x_2 + x_1 \cdot {x_2}^{D}$
		${x_1}^{D} \cdot x_2$
		$x_1 \cdot {x_2}^{D}$
$*=$	$y \cdot x$	${y}^{D} \cdot x + x \cdot {y}^{D}$
		${y}^{D} * x$
$/$	x_1/x_2	$({x_1}^{D} \cdot x_2 - x_1 \cdot {x_2}^{D})/(x_2 \cdot x_2)$
		${x_1}^{D}/x_2$
		$(-x_1 \cdot {x_2}^{D})/(x_2 \cdot x_2)$
$/=$	y/x	${y}^{D} \cdot x - x \cdot {y}^{D}$
		${y}^{D}/x$
sin	$\sin(x)$	$\cos(x)$
cos	$\cos(x)$	$-\sin(x)$
tan	$\tan(x)$	${x}^{D} \cdot (1 + \tan^2 x)$
$asin$	$\arcsin(x)$	${x}^{D}/(\sqrt{1 - x \cdot x})$
$acos$	$\arccos(x)$	$-{x}^{D}/(\sqrt{1 - x \cdot x})$
$atan$	$\arctan(x)$	${x}^{D}/(1 + x \cdot x)$
$atan2$	$\arctan_2(x_1, x_2)$	$(({x_1}^{D} x_2 - x_1 {x_2}^{D})/(x_2 x_2))/(1 + (x_1/x_2)^2)$
		$({x_1}^{D}/x_2)/(1 + (x_1/x_2)^2)$
		$((-x_1 {x_2}^{D})/(x_2 x_2))/(1 + (x_1/x_2)^2)$
exp	$\exp(x)$	${x}^{D} \cdot \exp(x)$
log	$\log(x)$	${x}^{D}/x$
$log10$	$\log_{10}(x)$	${x}^{D}/(x \cdot \log(10))$
$sqrt$	$\sqrt{x}$	${x}^{D}/(2 \cdot \sqrt{x})$
pow	$x_1^{x_2}$	$(({x_1}^{D}/x_1) \cdot x_2 + x_1 {x_2}^{D}) \cdot x_1^{x_2}$
		$(({x_1}^{D}/x_1) \cdot x_2) \cdot x_1^{x_2}$
		$({x_1}{x_2}^{D}) \cdot x_1^{x_2}$

Tabelle 2.2: Liste aller elementarer Operationen in C++.

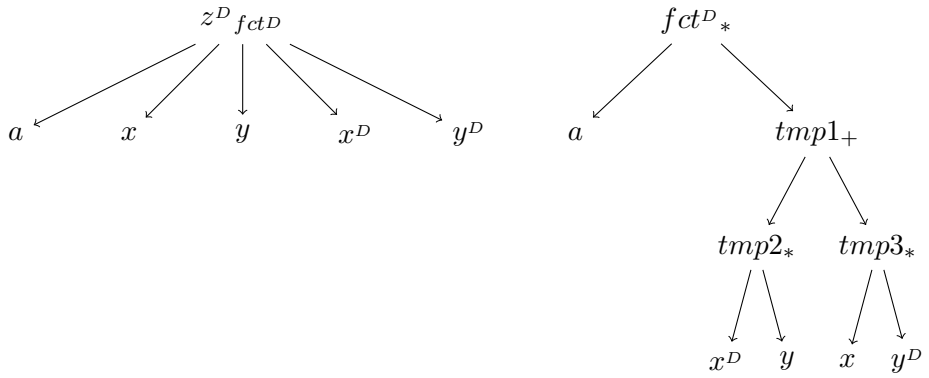

Abbildung 2.10: Graphendarstellung der Gleichungen (2.44) und (2.45).

Die Gleichungen (2.42) *und* (2.43) *haben den folgenden differenzierten Aufbau im Programm,*

$$\begin{aligned} fct1 &= f(a, x, y, x^D, y^D, fct1^D) \\ z &= fct1 \\ z^D &= fct1^D \end{aligned} \tag{2.44}$$

und

$$f(a, x, y, x^D, y^D, f^D) = \left\{ \begin{array}{l} a(x \cdot y) \\ f^D = a(x^D y + x y^D) \end{array} \right. . \tag{2.45}$$

Die Gleichung (2.45) *besagt, dass der Funktionswert f und die Ableitung f^D berechnet werden.*
Das Beispiel findet sich auf der CD im Verzeichnis `Programme/ex:GAuDiFct`.

Nun kommen wir zur letzten Programmeinheit von GAuDi-STT, dem Erzeugen der neuen Zeilen.

Neuer Code

Dieser Schritt ist selbstverständlich wieder sprachabhängig, denn hier wird aus den differenzierten Graphen wieder ein Quellcode erzeugt. Zuerst werden die Graphen in Kleingraphen zerlegt. Für jede Programmcodezeile wird ein eigener Graph erstellt. Aus Kleingraphen können wir nun die einzelnen Ausdrücke und die neuen Funktionsaufrufe in der entsprechenden Programmiersprache erzeugen. Anschließend werden die neuen Zeilen in das ursprüngliche Programm eingefügt. Die neuen Zeilen werden von der letzten Programmzeile ausgehend eingefügt, da dies so einfacher realisierbar ist.
GAuDi-STT finden wir auf der CD im Verzeichnis `Programme/gaudi/adbin`.

2.3.2 Überladene Operatoren

Die C++ Bibliothek GAuDi-OO[13] grenzt sich von anderen AD-Bibliotheken, wie zum Beispiel ADOL-C [GJU96], FAD [ADC99] oder FADBAD/TADIFF [BS96], durch seine Möglichkeit

[13] GAuDi-OO steht für GAuDi-Operator Overloading

zur Problem-Rückverfolgung ab.
Auch bei dieser Implementierung wurde das Prinzip verwendet, bewährte und getestete Strukturen zu verwenden. Zudem liegt hier ein weiterer Fokus auf der Laufzeitkomplexität der Bibliothek. Der folgende Abschnitt erklärt zuerst die Rückverfolgungs-Schnittstelle und geht dann auf die Realisierung der Überladenen Operatoren ein.

Problem-Rückverfolgung

Die im Abschnitt 2.2 beschriebene Problematik der Fehlersuche im AD-Code wird mit dem Ansatz der Problem-Rückverfolgung stark reduziert, da wir mit der Rückverfolgungsschnittstelle ein Werkzeug haben, mit dem wir zur Laufzeit relativ einfach kritische Bereiche überprüfen können, ohne jede eventuell kritische Stelle von Hand lokalisieren zu müssen.

Als erstes beschreiben wir wie der Problem-Rückverfolgungs-Mechanismus arbeitet. Anschließend gehen wir auf die technische Umsetzung ein und zuletzt zeigen wir eine Anwendung des Mechanismus.

Arbeitsweise

Die Problem-Rückverfolgung arbeitet in zwei Schritten. Der erste Schritt überprüft, ob eine Anforderung erfüllt oder verletzt ist. Der zweite Schritt ermittelt bei Bedarf den aktuellen *Backtrace* des Programms.
Standardmässig ist folgende Anforderung für die *assertion* implementiert. Es wird bei jeder Elementar-Operation überprüft, ob der Ableitungswert betragsmässig kleiner als eine vorgegebene Schranke ist. Allerdings können wir als Benutzer spezielle Anforderungen auf eine einfache Weise selbst ergänzen. Hierzu müssen dann zusätzliche Anforderungen für den entsprechenden Operator festgelegt werden. Eine Möglichkeit wäre, dass das Verhältnis zwischen dem Funktionswert und dem Ableitungswert eine bestimmte Grenze nicht über- oder unterschreiten darf, wenn ein Wert einer Variablen mit dem =-Operator zugewiesen wird.
Wenn die Bedingung erfüllt ist, arbeitet das Programm normal im Algorithmus weiter. Bei verletzter Bedingung wird der aktuelle *Backtrace* des Programmes bestimmt und ausgegeben. Abhängig von den Vorgaben des Anwenders arbeitet das Programm anschließend weiter oder es terminiert.

Technische Umsetzung

Die Realisierung der Problem-Rückverfolgung ist auch in zwei Stufen aufgeteilt. Die erste Stufe trifft die Entscheidung, ob ein Problem vorliegt. Dies ist in den `assertLogger` Routinen umgesetzt. Diese Funktionen rufen bei Bedarf die zweite Stufe auf, die die Rückverfolgung übernimmt. Zuerst beschreiben wir kurz die Umsetzung der `assertLogger` Routinen, bevor wir zur Realisierung der Rückverfolgung kommen.

Die Implementierung der `assertLogger` Funktionalität orientiert sich direkt an der ursprünglichen Idee zur Implementierung der `assert`-Routine aus dem Paket `assert.h`. So ist die Funktion als Makro realisiert und kann mittels eines Argumentes beim Compilieren aktiviert beziehungsweise deaktiviert werden. So ist sichergestellt, dass im deaktivierten Zustand kein zusätzlicher Aufwand zur Laufzeit entsteht.

Listing 2.14: Deklaration des assertionLogger Konstruktes.

```
1  #include<string>
2  #ifndef ASSERTLOGGER_H
3  #define ASSERTLOGGER_H 1
4  #ifdef NDEBUG
5  #define assertLogger(e1, e2)  ((void)0)
6  #else
7  #define LOGFILE "assertLogger.txt"
8  #ifdef NKILL
9  #define KILL ((void)0)
10 #else
11 #define KILL exit(1)
12 #endif
13   void assertLogger(const bool, const std::string&);
14 #endif
15 #endif
```

Im Listing 2.14 sehen wir die Deklaration des `assertLogger`'s. Wenn das Makro `NDEBUG` gesetzt ist, wird der Funktionsaufruf im Programmverlauf durch `((void)0)` ersetzt und erzeugt somit keinen zusätzlichen Aufwand zur Laufzeit des Programms. Ansonsten wird die Schnittstelle der Funktion deklariert und der Dateiname des Logfiles per Makro festgelegt. Das Makro `NKILL` entscheidet, ob das Programm bei einer nicht erfüllten Bedingung des `assertLogger` beendet wird oder ob es weiterarbeitet.

Listing 2.15: Implementierung der assertionLogger Funktion.

```
1  #include <assertLogger.h>
2  #include <logger.h>
3
4  #ifndef NDEBUG
5  void assertLogger(const bool assertion,const std::string &message)
6  {
7      if (!assertion) {
8          Logger logger(LOGFILE);
9          logger.logTrace(message);
10         KILL;
11     }
12 }
13 #endif
```

Das Listing 2.15 zeigt den Quellcode der `assertLogger` Funktion. Falls die Bedingung nicht erfüllt ist, wird eine Instanz der Klasse `Logger` angelegt und aufgerufen. Ist das Makro `NKILL` nicht gesetzt, wird das Programm anschließend beendet - bei gesetztem Makro läuft es weiter.

Bemerkung 2.3.11 (Makro NDEBUG)
Das Makro `NDEBUG` *muss nicht in einer Datei der Anwendung gesetzt werden. Mit dem* `gcc` *kann dies einfach mit der Option* `-D NDEBUG` *erledigt werden.*

Als nächstes erklären wir den Aufbau und die Funktionsweise der `Logger`-Klasse.

Die Rückverfolgung verwendet den GNU Debugger *gdb*, ohne dass wir uns als AD-Anwender mit dem Debugger von Hand durch das Programm arbeiten müssen.
Dies hat mehrere Vorteile: Da das Programm sehr lange laufen kann, bis irgendwo etwas passiert, das im Sinne der AD-Problematik interessant ist, müssen wir uns so nicht zeitaufwendig von Hand zu dieser Problemstelle vorarbeiten.
Zum Anderen suchen wir hier keine klassischen Probleme, wie man sie normalerweise mit einem Debugger sucht - wie zum Beispiel eine Speicherzugriffsverletzung - weshalb wir die Variablen einzeln anschauen müssen. Das händische Vorgehen, welches die Gefahr birgt, dass relevante Bereiche übersehen werden, entfällt.

Die `Logger`-Klasse startet zur Laufzeit den *gdb* und bereitet die *backtrace*-Daten auf und speichert sie ab. Die Filterung der *backtrace*-Daten wird mit den Tools *grep*, *sed* und *tail* realisiert.

Listing 2.16: Implementierung der Logger Klasse.

```
1  #include <logger.h>
2
3  //#include <execinfo.h>
4  #include <fstream>
5  #include <sstream>
6  #include <string>
7
8  #include<cstdlib>
9
10 Logger::Logger(const std::string& file)
11     :_file(file)
12 {
13     std::fstream fp(_file.c_str(), std::ios::out );
14     fp.close();
15 }
16
17 void
18 Logger::logTrace ()
19 {
20     std::stringstream systemStr;
21         systemStr << "gdb " << "$(ps " << getpid() << " -o command | tail -n1) " << getpid()
22                   << " --batch -x gdb.txt 2> .err.log |" << " grep ^# |"
23                   << " sed -e 's/#[0-9]*  0x[0-9a-zA-Z]* in//g'|"
24                   << " sed -e 's/) at .*$//g' |"
25                   << " tail -n"<<RECURSION_DEPTH<< " > .tmp.log"  <<";"
26                   << " tail -n$(echo $(cat .tmp.log | wc -l) - 5 | bc -l) .tmp.log>>"
27                   << _file << "; " <<" rm .err.log";
28         system(systemStr.str().c_str());
29 }
30
31 void
32 Logger::logTrace(const std::string& info){
33     logInfo(info);
34     logTrace();
35     logInfo("\n");
36 }
37
38 void
39 Logger::logInfo(const std::string& info)
40 {
41     std::fstream fp(_file.c_str(), std::ios::out | std::ios::app);
42     fp << info << std::endl;
43     fp.close();
44 }
```

Die zentrale Funktionalität der Klasse wird in der Methode `logTrace` zur Verfügung gestellt. In den Zeilen 23 bis 29 im Listing 2.16 wird der *gdb* mit den entsprechenden Optionen gestartet und die Ergebnisse aufbereitet. Im Folgenden erklären wir den Aufruf etwas näher. Mit der Funktion `getpid()` erhalten wir die Prozessid (PID) unseres Programms. Den Namen des ausgeführten Programmes erhalten wir mit dem Ausdruck `ps PID -o command | tail -n1`. Diesen Ausdruck übergeben wir ausgewertet an den *gdb*, die Auswertung ist durch den Ausdruck `$(BEFEHL)` realisiert. Das Argument `-o command` sorgt dafür, dass der Befehl *ps* nur den Namen des Prozesses liefert. Da wir nur an der letzten Zeile der Ausgabe vom Befehl *ps* interessiert sind, übergeben wir das Ergebnis mit einer Pipe `|` an den Befehl *tail* und `-n1` besagt dass wir nur die letzte Zeile möchten.
Der *gdb* erhält noch zwei Argumente: `--batch` dieses Argument startet den *gdb* im Batch-Modus. In diesem wird dann mit dem Argument `-x` eine Datei erwartet, in der die Kommandos für den *gdb* erwartet werden. Die Kommando-Datei enthält bei uns in der Default-Konfiguration nur zwei Anweisungen, nämlich `bt`, damit der Backtrace ausgegeben wird und anschließend den Befehl `quit` zum Terminieren des *gdb*'s. So haben wir Benutzer die Mög-

lichkeit, ohne großen Aufwand, die Funktionnalität auf eigene Wünsche, anzupassen. Als Letztes werden mögliche Fehlerausgaben in eine temporäre Datei umgeleitet, die am Ende mit *rm* automatisch gelöscht wird.
Anschließend wird die Ausgabe an das Programm *grep* übergeben, welches alle für uns relevanten Zeilen herausfiltert. Diese werden dann mit den folgenden beiden *sed* Aufrufen:

- `-e 's/#[0-9]*  0x[0-9a-zA-Z]* in//g '|`,
- `-e 's/) at .*$/)/g' |`

auf eine lesbare Form formatiert.
Zuletzt wird die Ausgabe auf die gewünschte Rekursionstiefe im Backtrace beschnitten. Dies erledigen wir mit einer Kombination aus den Befehlen *tail, echo, wc* und *bc*. Das endgültige Resultat wird in die dem Logger übergebene Datei geschrieben.
Eine ausführliche Erklärung zu den verwendeten Tools *bc, cat, echo, gdb, ps, rm, sed, tail* und *wc* sind auf den Manpages [Nel], [BSD04], [BSD03] [Fre02], [BSD05a], [BSD99], [Spi05], [BSD93] und [BSD05b] zu finden. Die Manpages können auf einer Kommandozeile über `man BEFEHL` aufgerufen werden.
Damit dieser Rückverfolgungs-Mechanismus zur Verfügung steht, müssen wie als Benutzer nur beim Compilieren des Programmes die Compiler-Option `-g` angeben[14]. Zudem entsteht kein zusätzlicher Programmieraufwand. Beim Weglassen der Debug option `-g` und dem Deaktivieren der Assertions gibt es auch keinen zusätzlichen Laufzeitaufwand.

Anwendung

Nun zeigen wir anhand eines kleinen Beispielprogramms wie die Rückverfolgung funktioniert.

Listing 2.17: Verwendung von *assertLogger*.

```
1  #include <assertLogger.h>
2
3  void assertionFct(int i){ assertLogger(i<10, "i<10"); }
4
5  void fct1(int i){ assertionFct(1); }
6  void fct2(int i){ assertionFct(i); }
7  void fct3(int i){ fct2(i-1); }
8  void fct4(int i){ fct1(i-1); }
9  void fct5(int i){ fct4(i-1); fct3(i-1); }
10
11 int main() {
12     for(int i=0; i<10000; ++i) {
13         fct5(i);
14         fct4(i-2); ++i;
15         fct3(i-1); ++i;
16         fct4(i++);
17         fct1(i);
18         fct1(2);
19     }
20     return 0;
21 }
```

Im Listing 2.17 haben wir eine kleine Implementierung zur Demonstration des `assertLogger`'s. Die Funktionen `fct1` bis `fct5` haben die Aufgabe einen Backtrace aufzubauen. Sobald die Variable `i` in der Funktion `assertionFct` den Wert 10 überschreitet, wird die Bedingung des `assertLogger` verletzt und der Backtrace wird ermittelt. Der ermittelte Backtrace hat bei unserem Beispiel folgende Gestalt:

[14] Die Compiler Optionen beziehen sich immer auf den Compiler *gcc*.

```
i<10
 assertionFct ()
 fct2 ()
 fct3 ()
 fct5 ()
 main ()
```

Zu beachten ist, dass die Laufzeit der mit der Option `-g` übersetzten Debugging-Version deutlich größer als die Version, bei der man auf das Überprüfen der Bedingung, mittels `-D NDEBUG` verzichtet, ist.
Das vollständige Beispiel mit einem Skript, das die unterschiedlichen Versionen erzeugt, findet man auf der CD im Verzeichnis `Programme/ex:Logger`.

AD-Klasse

Jetzt befassen wir uns mit dem Aufbau der eigentlichen Bibliothek, die das Automatische Differenzieren zur Verfügung stellt. Wir betrachten die Bibliothek nur ausschnittsweise, folgende Bereiche betrachten wir exemplarischen: Die Konstruktoren, den Zuweisungsoperator, den Additionsoperator, einen Vergleichsoperator, und eine mathematische Funktion.

Listing 2.18: AD-Template-Klasse

```
1 template<int D, typename T>
2 class AD {
3     private:
4         T _value;
5         T _derivative[D];
6 ...
```

Den Kopf der AD-Template-Klasse sehen wir im Listing 2.18. Der erste Template Parameter `int D` legt die Dimension des Gradienten Vektors fest. Dadurch ist die Dimension zur compile Zeit bekannt. Somit brauchen wir keine Mechanismen zur Überprüfung der Dimension des Gradienten Vektors bei den Rechenoperationen, beziehungsweise bei den Zuweisungen, da dies durch den Template Parameter schon bei Compilieren erledigt wird. Dies wirkt sich sehr positiv auf die Laufzeit aus.
Der zweite Template Parameter legt den Datentyp fest. Somit können wir festlegen ob wir in `single`, `double` Precision oder einem anderen Datentyp rechnen möchten. Denkbar wäre auch ein Datentyp für die GPU. Somit könnte man die komplette Berechnung von der CPU auf die GPU verlagern, ohne das man dass Programm vollständig umarbeiten muss.

Konstruktoren

Wir betrachten nun die Konstruktoren und den Destruktor, die die AD-Klasse zur Verfügung stellt.

Listing 2.19: Deklaration der Konstruktoren

```
1 AD();
2 AD(const T& value);
3 AD(const T derivative[]);
4 AD(const T& value, const T derivative[]);
5 AD(const AD<D,T>& arg);
6 ~AD();
```

In Listing 2.19 sehen wir die Deklaration der fünf Konstruktoren und des Destruktors. Die fünf Konstruktoren sind der Default-Konstruktor, drei Initialisierungs-Konstruktoren und der Copy-Konstruktor.

Listing 2.20: Implementierung der Konstruktoren

```
1  template<int D, typename T>
2  AD<D,T>::AD ()
3      : _value(0)
4  {
5      for (int i=0; i<D; ++i) {
6          _derivative[i] = 0;
7      }
8  }
9
10 template<int D, typename T>
11 AD<D,T>::AD (const T& value)
12     :_value(value)
13 {
14     for (int i=0; i<D; ++i) {
15         _derivative[i] = 0;
16     }
17 }
18
19 template<int D, typename T>
20 AD<D,T>::AD(const T derivative[])
21     :_value(0)
22 {
23     for (int i=0; i<D; ++i) {
24         _derivative[i] = derivative[i];
25     }
26 }
27
28 template<int D, typename T>
29 AD<D,T>::AD(const T& value, const T derivative[])
30     : _value(value)
31 {
32     for (int i=0; i<D; ++i) {
33         _derivative[i] = derivative[i];
34     }
35 }
36
37 template<int D, typename T>
38 AD<D,T>::AD(const AD<D,T>& arg)
39     : _value(arg._value)
40 {
41     for (int i=0; i<D; ++i) {
42         _derivative[i] = arg._derivative[i];
43     }
44 }
45 //
46 template<int D, typename T>
47 AD<D,T>::~AD()
48 { }
```

Die Implementierung der Konstruktoren in dem Listing 2.20 haben alle einen analogen Aufbau. Als erstes wird die Variable `_value` auf einen definierten Wert gesetzt: 0 falls kein anderer Wert für die Variable `_value` übergeben wurde, ansonsten auf den übergebenen Wert. Die Initialisierung des Gradienten Vektors geschieht im Inneren der Methode mit einer Schleife. Da der Laufbereich schon beim Compilieren bekannt ist, können hier von Seiten des Compilers Optimierungen vorgenommen werden.
Der Destruktor in den Zeilen 46 bis 48 hat in dieser Version keine Aufgaben, da kein Speicher dynamisch allokiert wird.

Zuweisungsoperator

Für den Zuweisungsoperator benötigen wir zwei Varianten, die wir in den Listings 2.21 und 2.22 betrachten werden.

Listing 2.21: Deklaration des Zuweisungsoperators

```
1     AD<D,T> & operator=(const T&);
2     AD<D,T> & operator=(const AD<D,T>&);
```

In der Deklaration der Zuweisungsoperatoren sehen wir die zwei Fälle, die betrachtet werden müssen. Wir weisen einer AD-Variable eine andere AD-Variable oder eine nicht AD-Variablen zu, zum Beispiel eine `double` Variable.

Listing 2.22: Implementierung des Zuweisungsoperators

```
1  template<int D, typename T>
2  AD<D,T> & AD<D,T>::operator=(const T& rhs)
3  {
4      _value = rhs;
5      for (int i=0; i<D; ++i) {
6          _derivative[i] = 0;
7      }
8      return *this;
9  }
10
11 template<int D, typename T>
12 AD<D,T> & AD<D,T>::operator=(const AD<D,T>& rhs)
13 {
14     if (this != &rhs) {
15         _value = rhs._value;
16         for (int i=0; i<D; ++i) {
17             _derivative[i] = rhs._derivative[i];
18         }
19     }
20     return *this;
21 }
```

Wenn wir der AD-Variablen eine nicht AD-Variable vom Template Typ `T` zuweisen, ist das Vorgehen sehr einfach. Die Variable `_value` wird auf den Wert der rechten Seite (RHS) `rhs` gesetzt und der Gradienten-Vektor `_derivative` wird auf 0 gesetzt.
Wird dem AD-Typ ein anderer AD-Typ zugewiesen, so wird die Zuweisung nur gemacht, wenn die Speicheradresse der linken Seite (LHS) und der rechten Seite unterschiedlich sind. Ist dies der Fall, so werden die Einträge Element für Element zugewiesen.

Additionsoperator

Im Listing 2.23 sind die sechs unterschiedlichen Realisierungen des + Operators, stellvertretend für alle Rechenoperatoren, dargestellt.

Listing 2.23: Deklaration der + Operatoren

```
1  AD<D,T> operator+();
2  //
3  const AD<D,T> operator+(const T&) const;
4  const AD<D,T> operator+(const AD<D,T>&) const;
5  //
6  template<int E, typename U> friend const AD<E,U> operator+(const U&, const AD<E,U>&);
7  //
8  AD<D,T> & operator+=(const T&);
9  AD<D,T> & operator+=(const AD<D,T>&);
10 //
11 AD<D,T> & operator++();
```

In Zeile 1 ist der Vorzeichen-Operator deklariert, Zeile 3 deklariert den Fall, dass das Argument auf der rechten Seite (RHS) ein nicht AD-Typ ist, das heißt zum Beispiel vom Typ `double` ist. Die 4. Zeile behandelt den Fall, dass beide Argumente (LHS, RHS) vom AD-Typ sind. Zeile 6 betrachtet den Fall, dass das Argument auf der linken Seite des Operators kein AD-Typ ist. Somit muss der Operator über eine `friend` Funktion erklärt werden. Die Zeilen 8 und 9

deklarieren den += Operator für beide Fälle. In der letzten Zeile ist noch der Inkrementoperator deklariert.

Listing 2.24: Implementierung der + Operatoren

```
1  template<int D, typename T> AD<D,T> AD<D,T>::operator+() {
2      return AD(*this);
3  }
4  //
5  template<int D, typename T> const AD<D,T> AD<D,T>::operator+(const T& rhs) const {
6      return AD(*this) += rhs;
7  }
8
9  template<int D, typename T> const AD<D,T> AD<D,T>::operator+(const AD<D,T>& rhs) const {
10     return AD(*this) += rhs;
11 }
12
13 template<int D, typename T> const AD<D,T> operator+(const T& lhs, const AD<D,T>& rhs) {
14     return AD<D,T>(rhs) += lhs;
15 }
16 //
17 template<int D, typename T> AD<D,T> & AD<D,T>::operator+=(const T& rhs) {
18     _value += rhs;
19     return *this;
20 }
21
22 template<int D, typename T> AD<D,T> & AD<D,T>::operator+=(const AD<D,T>& rhs) {
23     _value += rhs._value;
24     for (int i=0; i<D; ++i) {
25         _derivative[i] += rhs._derivative[i];
26         assertLogger(fabs(_derivative[i])<AD_MAX, int2str(i));
27     }
28     return *this;
29 }
30 //
31 template<int D, typename T> AD<D,T> & AD<D,T>::operator++() {
32     ++_value;
33     return *this;
34 }
```

Im Anschluss an die Deklarationen im Listing 2.23 schauen wir uns im Listing 2.24 die zugehörigen Implementierungen an. Die Zeilen 1 bis 3 implementieren den Vorzeichen-Operator und geben nur den Typ zurück. Die Implementierung der einzelnen + Operatoren, in Zeile 5 bis 15, werden auf einen der beiden += Operatoren zurückgeführt. Der Vorteil dieses Ansatzes ist es, eine möglichst minimale Implementierung zu erhalten, um die Fehleranfälligkeit klein zu halten. Die Realisierung des += Operators mit einem nicht AD-Typ auf der rechten Seite reduziert sich auf einen += Operator für den Funktionswert. Dies ist in den Zeilen 17 bis 20 gezeigt. In den Zeilen 22 bis 29 betrachten wir den += Operator mit einem AD-Typ als rechte Seiten-Argument, in Zeile 23 wird der Funktionswert behandelt. In der Schleife in den Zeilen 24 bis 27 wird erst der Ableitungswert berechnet und anschließend mit der Debugging-Routine `assertLogger` getestet. Eine Erklärung der Funktionsweise des Debugging Mechanismus ist in 2.3.2 gegeben. In den Zeilen 31 bis 33 sehen wir noch den Inkrementoperator, der nur den Funktionswert erhöht, den Ableitungswert aber unverändert lässt und die Variable zurückgibt.

Vergleichsoperator

Bei der Umsetzung der Vergleichsoperatoren muss man zuerst eine grundlegende Entscheidung treffen, was man Vergleichen möchte. Es gibt zwei naheliegende Varianten. Die erste, die wir hier auch verfolgen ist, dass man nur den Eintrag der `_value` Variable vergleicht, da so derselbe Programmfluss gewährleistet ist, wie wenn man das Programm mit einem nativen Datentyp ohne das Automatische Differenzieren ausführen würde. Der Seiteneffekt dieser Vorgehensweise ist, dass es möglich ist, dass `x._value == y._value` wahr ist, aber

`x._derivative(i) == y._derivative(i)` und `i=0,...,D` für ein oder mehrere `i` falsch ist.
Die andere Variante verfolgt den Ansatz "*was gleich ist, ist auch gleich*". Damit ist gemeint, dass wenn ein Vergleich wahr ist, dies für alle Einträge des AD-Types gilt, dass heißt für `_value` und `_derivative[i]` für `i=0,...,D`. Der Nachteil ist, dass sich das Verhalten eines Programmes ändern kann, wenn wir es von nativen Datentypen auf den AD-Typ umstellen.
Wir haben die Entscheidung anhand des ==-Operators erklärt; das Verhalten für die anderen Vergleichsoperatoren !=, <, <=, > und >= ist analog dazu.

Listing 2.25: Deklaration der Vergleichsoperatoren

```
1     const bool operator==(const T&) const;
2     const bool operator==(const AD<D,T>&) const;
3
4     template<int E, typename U> friend
5         const bool operator==(const U&, const AD<E,U>&);
```

Listing 2.25 zeigt die Deklaration der drei Formen des Vergleichsoperator ==. Wir müssen die folgenden Unterscheidungen in der Signatur betrachten. Die Argumente der linken Seite und der rechten Seite sind vom selben AD-Typ. Zusätzlich müssen die beiden Variablen berücksichtigt werden, bei denen das Argument der linken Seite und das Argument der rechten Seite unterschiedliche Typen haben. Der Typ des einen Argument ist hierbei sicher ein AD-Typ. Für das zweite Argument gilt, dass es vom Datentyp des AD-Argument ist. Es ist hierbei möglich, dass beide Argumente auch von unterschiedlichem AD-Typ sind. Dies ist der Fall, wenn wir einen AD-Typ haben der einen AD-Typ als Datentyp hat.

Listing 2.26: Implementierung der Vergleichsoperatoren

```
1  template<int D, typename T>
2  const bool AD<D,T>::operator==(const T& rhs) const
3  {
4      return _value==rhs;
5  }
6
7  template<int D, typename T>
8  const bool AD<D,T>::operator==(const AD<D,T>&rhs) const
9  {
10     return _value==rhs._value;
11 }
12
13 template<int D, typename T>
14     const bool operator==(const T& lhs, const AD<D,T>& rhs)
15 {
16     return lhs._value==rhs;
17 }
```

Die Implementierungen des Vergleichsoperator ==, siehe Listing 2.26, vergleichen jeweils nur die Funktionswerte miteinander und geben dieses Ergebnis zurück.

Mathematische Funktion

Als Letztes betrachten wir exemplarisch folgende mathematische Funktionen, die wir mit dem Schlüsselwort `friend` markiert haben, da sie auf die privaten Einträge unserer Klasse zugreifen dürfen.

Das Listing 2.27 zeigt die drei mathematischen Funktionen, `log`, `sqrt` und `pow`, die wir hier exemplarisch betrachten.

Listing 2.27: Deklaration von mathematischen Funktionen

```
1     const AD<E,U> log(const AD<E,U>&);
2     template<int E, typename U> friend
3     const AD<E,U> sqrt(const AD<E,U>&);
4
5     template<int E, typename U> friend
6     const AD<E,U> pow(const AD<E,U>& arg1, const U& arg2);
7     template<int E, typename U> friend
8     const AD<E,U> pow(const U& arg1, const AD<E,U>& arg2);
9     template<int E, typename U> friend
10    const AD<E,U> pow(const AD<E,U>& arg1, const AD<E,U>& arg2);
```

Listing 2.28: Implementierung von mathematischen Funktionen

```
1  template<int D, typename T>
2  const AD<D,T> log(const AD<D,T>& arg)
3  {
4      AD<D,T> out(arg);
5      out._value = log(arg._value);
6      for (int i=0; i<D; ++i) {
7          out._derivative[i] /= arg._value;
8          assertLogger(fabs(out._derivative[i])<AD_MAX, int2str(i));
9      }
10     return out;
11 }
12 template<int D, typename T>
13 const AD<D,T> sqrt(const AD<D,T>& arg)
14 {
15     AD<D,T> out(arg);
16     out._value = sqrt(arg._value);
17     const T tmp = out._value*2;
18     for (int i=0; i<D; ++i) {
19         out._derivative[i] /= tmp;
20         assertLogger(fabs(out._derivative[i])<AD_MAX, int2str(i));
21     }
22     return out;
23 }
24 template<int D, typename T>
25 const AD<D,T> pow(const AD<D,T>& arg1, const T& arg2)
26 {
27     AD<D,T> out(arg1);
28     out._value = pow(arg1._value, arg2-1);
29     const T tmp = out._value*arg2;
30     out._value *= arg1._value;
31     for (int i=0; i<D; ++i) {
32         out._derivative[i] *= tmp;
33         assertLogger(fabs(out._derivative[i])<AD_MAX, int2str(i));
34     }
35     return out;
36 }
37 template<int D, typename T>
38 const AD<D,T> pow(const T& arg1, const AD<D,T>& arg2)
39 {
40     AD<D,T> out(arg2);
41     out._value = pow(arg1, arg2._value);
42     const T tmp = out._value*log(arg1);
43     for (int i=0; i<D; ++i) {
44         out._derivative[i] *= tmp;
45         assertLogger(fabs(out._derivative[i])<AD_MAX, int2str(i));
46     }
47     return out;
48 }
49 template<int D, typename T>
50 const AD<D,T> pow(const AD<D,T>& arg1, const AD<D,T>& arg2)
51 {
52     AD<D,T> out(arg1);
53     out._value = pow(arg1._value, arg2._value);
54     const T tmp1 = log(arg1._value);
55     const T tmp2 = arg2._value/arg1._value;
56     for (int i=0; i<D; ++i) {
57         out._derivative[i] = (arg2._derivative[i]*tmp1 +
58                               arg1._derivative[i]*tmp2 )*out._value;
59         assertLogger(fabs(out._derivative[i])<AD_MAX, int2str(i));
60     }
61     return out;
62 }
```

Listing 2.28 setzt die Implementierung der drei mathematischen Funktionen um. Zuerst sehen wir die Implementierung der `log`-Funktion. Hier muss man sich um den Defintionsbereich der Ableitung keine Gedanken machen, da dieser mit dem Definitionsbereich der Funktion selbst übereinstimmt. Die Ableitung `sqrt`-Funktion ist im Punkt $x = 0$ nicht mehr erklärt. Allerdings behandeln wir diese Stelle implizit mit dem assertLogger da für $x = 0$ die geforderte Bedingung, die Beschränktheit der Ableitung innerhalb eines Limits, nicht mehr erfüllt wird. Bei den Implementierungen der `pow`-Funktion sehen wir gleich, dass sich die Implementierung deutlich unterscheidet, abhängig davon, in welchem der drei Fälle *AD-Typ, Basis-Typ*, *Basis-Typ, AD-Typ* und *AD-Typ, AD-Typ*, wir sind.

Höhere Ableitungen

Eine einfache Möglichkeit, höherere Ableitungen zu realisieren, ist es, in der beschriebenen AD-Klasse den Templateparameter `T` erneut als AD-Typ zu setzen. Dies hätte dann beispielsweise folgende Gestalt `AD<2,AD<2,double> >` für die zweite Ableitung. Der Vorteil dieser Variante ist, dass man keine zweite Bibliothek mit Überladenen Operatoren benötigt, die zur ersten auch noch die zweite, dritte, $\cdots$ Ableitung mitberechnen. Den Preis, den man für diese einfachere Variante zahlt, ist, dass man die erste Ableitung doppelt berechnet.
Um die Funktionalität zu demonstrieren, betrachten wir das Beispiel 2.3.12.

Beispiel 2.3.12 (Zweite Ableitung)
Wir berechnen den Funktionswert der Funktion

$$f(x) := x^2, \tag{2.46}$$

in einem Programm. Nun möchten wir von der Funktion (2.46) *zusätzlich die erste und zweite Ableitung mitberechnen. Dazu verwenden wir das Automatische Differenzieren.*

Listing 2.29: Höhere Ableitungen

```
1  #include <iostream>
2  #include <ad.h>
3
4  AD<1, AD<1, double>> sqr(AD<1, AD<1, double>> x) {
5      return x*x;
6  }
7
8  int main() {
9      double init[] = {0};
10     AD<1, double> tmp1(1., init);
11     AD<1, double> tmp2[] ={tmp1};
12     init[0] = 1;
13     tmp1.setDerivative(init);
14     tmp1.setValue(2);
15     AD<1, AD<1, double>> f, x(tmp1, tmp2);
16     f = sqr(x);
17     AD<1, double> f1 = f.value();
18     AD<1, double> f2 = f.derivative(0);
19     std::cout<< "f(" << tmp1.value()<< ")   = "<< f1.value()      << std::endl;
20     std::cout<< "f'(" << tmp1.value()<< ")  = " << f1.derivative(0)<< std::endl;
21     std::cout<< "f'(" << tmp1.value()<< ")  = " << f2.value()      << std::endl;
22     std::cout<< "f''("<< tmp1.value()<< ") = "  << f2.derivative(0)<< std::endl;
23     return 0;
24 }
```

In Listing 2.29 sehen wir die Implementierung unseres Ausgangsproblems. Die Berechnung der Funktion (2.46) *ist in den Zeilen 4 bis 6 zu sehen. Hier erkennt man schön die geschachtelte Gestalt unserer AD-Template-Klasse. Der zweite Template-Parameter ist erneut vom Typ AD mit gleichem Dimensions-Template-Parameter, allerdings vom Datentyp* `double`*. Die eigentliche Berechnung von* (2.46)*, in Zeile 5, unterscheidet sich nicht.*

Im Hauptprogramm ist in den Zeilen 9 bis 15 die Initialisierung der aktiven Variable `x` *implementiert. Für das initialisierte* `x` *muss Folgendes gelten* $x' = 1$ *und* $x'' = 0$.
In der Zeile 16 rufen wir dann die Funktion `xSqr` *auf und weisen den Rückgabewert der Variablen* `f` *zu.*
Die Zeilen 17 bis 22 dienen anschließend nur noch der Ausgabe der Funktion (2.46) *und deren erste und zweite Ableitung. In den Zeilen 20 und 21 sieht man, dass die erste Ableitung doppelt berechnet wurde und wir somit auf unterschiedliche Weise auf den Ableitungswert zugreifen können.*
Das vollständige Beispiel findet sich auf der CD im Verzeichnis `Programme/ex:HoehereAbleitungen`.

Eine weitere Möglichkeit, die doppelte Berechnung zu vermeiden, wäre, wenn man eine AD-Klasse schreibt, die für den Funktionswert `_value` und den Gradienten-Vektor `_derivative` getrennte Template-Parameter besitzt. Allerdings muss man bei dieser Form darauf achten, dass sicher gestellt ist, dass die Datentypen zueinander verträglich sind.

3 Strömungsdynamik

In diesem Kapitel werden die notwendigen Grundlagen der Strömungsmeachnik zur Modellierung und numerischen Simulation der Navier-Stokes-Gleichungen[1] hergeleitet, damit wir anschließend die Anwendung vom Automatischen Differenzieren auf die Strömungssimulation betrachten und analysieren können.

3.1 Physikalische Grundlagen

Ziel der Strömungsdynamik ist es, das physikalische Verhalten von Fluiden zu beschreiben. Wir beginnen die Herleitung der physikalischen Eigenschaften von Strömungen mit der näheren Betrachtung des beobachteten Mediums, dem Fluid.

Bemerkung 3.1.1 (Fluid)
Das Wort Fluid kommt vom lateinischen Wort "fluidus" und bedeutet fließend. In der Physik beschreibt ein Fluid ein Medium, das äußeren Scherkräften[2] keinen Widerstand leistet.
Ein Fluid kann sowohl für ein Gas, wie auch für eine Flüssigkeit stehen, da unabhängig von den Unterschieden der beiden Medien dieselben Bewegungsgleichungen gelten.

Die Definition 3.1.1 orientiert sich an der Beschreiung eines Fluids im Buch *Numerische Strömungsmechanik* von Ferziger und Perić [FP08], eine detalierte Betrachtung der Fluideigenschaften findet man in *Grundzüge der Strömungslehre* von Zierep und Bühler [ZB08].

Definition 3.1.2 (Newtonsches Fluid)
Ein Fluid das folgende Eigenschaft

$$\tau = \eta \frac{du}{d\nu} \tag{3.1}$$

erfüllt, nennt man **newtonsches Fluid**. *Wobei τ die Scherspannung, η die dynamische Viskosität und $\frac{du}{d\nu}$ die Schergeschwindigkeit beschreibt. In Worten formuliert besagt die Gleichung* (3.1) *damit folgende Aussage: Ist die Scherspannung eines Fluides proportional zur Schergeschwindigkeit, so ist es ein newtonsches Fluid.*
In der Figur 3.1 ist der Zusammenhang der Scherspannung ν und der Schergeschwindigkeit $\mathbf{u}$ grafisch veranschaulicht. Die Größe der Schergeschwindigkeit ist durch die Länge der Vektoren in horizontaler Richtung dargestellt. In vertikaler Lage ist der Anstieg der Scherspannung abgebildet.

Im Folgenden werden wir nur newtonsche Fluide betrachten und meinen implizit newtonsche Fluide wenn wir von einem Fluid sprechen.
Für die folgenden Herleitungen benutzen wir das Kontinuitätsprinzip und das newtonsche Gesetz.

[1] Benannt nach Claude Louis Marie Henri Navier (1785-1836) und George Gabriel Stokes (1819-1903).
[2] Scherkräfte sind Kräfte die parallel zu den Flächen des betrachteten Körpers wirken.

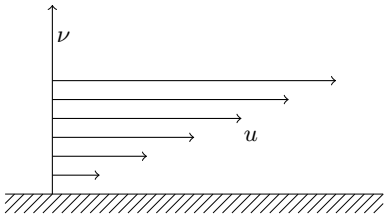

Abbildung 3.1: Schematische Darstellung der Schergrößen u und ν.

Definition 3.1.3 (Kontinuitätsprinzip)
Das Kontinuitätsprinzip besagt, dass bei der klassischen Physik nichts einfach verschwinden, beziehungsweise, dass nichts aus dem Nichts entstehen kann.

Definition 3.1.4 (Impulserhaltungssatz)
Mit dem zweiten und dritten Newton'sche Axiom ergibt sich direkt der Impulserhaltungssatz:

$$\sum \mathbf{f} = \frac{d}{dt} \sum \mathsf{p} \tag{3.2}$$

wobei $\sum \mathbf{f}$ die Summe aller Kräfte auf einen Körper und $\sum \mathsf{p}$ den Gesamtimpuls auf einen Körper bezeichnen.

Als Nächstes leiten wir die Euler-Gleichungen[3] über das Prinzip der Massenerhaltung und der Impulserhaltung her. Anschließend erweitern wir diese dann um die innere Reibungskräfte zu den Navier-Stokes-Gleichungen.
Das Vorgehen im nächsten Abschnitt orientiert sich an der Herangehensweise im Buch *Numerische Strömungsmechanik* von Ferziger und Perić [FP08].

3.1.1 Euler-Gleichungen

Zur Herleitung der Euler-Gleichungen betrachten wir zunächst eine generische Form der Erhaltungsgleichung (3.3) mit der allgemeinen Größe ϕ. Die Erhaltungsgleichung formalisiert das Kontinuitätsprinzip. Die generische Größe ϕ werden wir für die Massenerhaltung und die Impulserhaltung spezialisieren. Die generische Erhaltungsgleichung für ein festes Massenvolumen V_M lautet:

$$\Phi = \int_{V_M} \rho \phi dV, \tag{3.3}$$

ρ beschreibt hierbei die Dichte des betrachteten Mediums. Im Rahmen dieser Arbeit wird hiermit immer die Massendichte des Fluids gemeint, soweit nicht explizit etwas anderes gesagt wird. Die Erhaltungsgleichung gilt in dieser Form allerdings allgemein, zum Beispiel auch in der Elektrodynamik, wobei hier die Dichte dann eine Ladungsdichte ist.
Nun betrachten wir zusätzlich zum Massenvolumen V_M ein Kontrollvolumen V_V mit festem Volumen und der zugehörigen Oberfläche S_V. Damit ergibt sich:

$$\frac{\partial}{\partial t} \int_{V_M} \rho \phi dV = \frac{\partial}{\partial t} \int_{V_V} \rho \phi dV + \int_{S_V} \rho \phi (\mathbf{v} - \mathbf{v_s}) \nu dS. \tag{3.4}$$

[3] Benannt nach Leonhard Euler (1707-1783).

v beschreibt den Geschwindigkeitsfluss des Fluides und v_s die Oberflächengeschwindigkeit des Kontrollvolumen V_V. Bei einem ruhenden Kontrollvolumen entfällt die Komponente der Oberflächengeschwindigkeit. Mit ν wird die normierte äußere Normale der Oberfläche S_V bezeichnet.
Die Gleichung (3.4) besagt, dass die zeitliche Änderung des Integral über ein Massenvolumen gleich der Summe aus zeitlicher Änderung eines Integrals über ein Kontrollvolumen mit festem Volumen und der Menge des Zu- beziehungsweise des Abflusses über die Oberfläche des Kontrollvolumen ist.

Massenerhaltung

Für ein abgeschlossenen System folgt nach dem Kontinuitätsprinzip, dass das Prinzip der Massenerhaltung gelten muss. Da in einem abgeschlossenen System kein Massefluss vorhanden ist, ergibt sich für die Massenerhaltung

$$\frac{\partial m}{\partial t} \stackrel{!}{=} 0. \tag{3.5}$$

Nun betrachten wir die generische Erhaltungsgleichung (3.4) in einem abgeschlossenen System mit dem festen Wert $\phi = 1$. Die Differenz zwischen den Geschwindigkeiten des Fluides und der Oberfläche des Kontrollvolumens fassen wir zu $\mathbf{v}$ zusammen. Unter Berücksichtigung der Massenerhaltung (3.5) folgt damit

$$\frac{\partial}{\partial t} \int_{V_V} \rho dV + \int_{S_V} \rho \mathbf{v} \nu dS = 0. \tag{3.6}$$

In Worten besagt die Gleichung (3.6), dass sich die Masse eines festen Kontrollvolumen nur ändern kann, wenn etwas in das Volumen, beziehungsweise aus dem Volumen, fließt.
Die Integralform der Massenerhaltung (3.6) können wir auch in eine Differentialform umschreiben. Hierzu wenden wir die Leibnizregel für Parameterintegrale[4] und den Gaußschen Integralsatz[5] auf die Gleichung (3.6) an und differenzieren sie anschließend bezüglich der Ortsvariablen. Damit ergibt sich die Differentialform:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = \frac{\partial \rho}{\partial t} + \mathbf{div}\, \rho v = 0. \tag{3.7}$$

Gelegentlich wird die formale Umformung der Integralform (3.6) in die Differentialform (3.7) durch einen heuristischen Ansatz ersetzt. Bei dem heuristischen Vorgehen betrachtet man Kontrollvolumen, deren Volumina man gegen Null gehen lässt.

Impulserhaltung

Im Impulserhaltungssatz 3.1.4 können wir den Gesamtimpuls $\sum \mathsf{p}$ auch als das Produkt von Masse m und Geschwindigkeit $\mathbf{v}$, $\sum \mathsf{p} = m\mathbf{v}$ darstellen. Damit ergibt sich die Impulserhaltung (3.2) zu:

$$\frac{\partial (m\mathbf{v})}{\partial t} = \sum \mathbf{f}. \tag{3.8}$$

[4] Siehe *Analysis 2* §10 Satz 2 von Forster [For05].
[5] Siehe *Analysis 2* Kapitel 12.4 von Königsberger [Kö04].

Wir ersetzen nun in der generischen Form des Erhaltungssatzes (3.4) den generischen Parameter ϕ durch die Geschwindigkeit $\mathbf{v}$. Damit ergibt sich eine Formulierung der Impulserhaltung (3.8) für Kontrollvolumen V_V mit festem Volumen.

$$\frac{\partial}{\partial t}\int_{V_V}\rho\mathbf{v}dV + \int_{S_V}\rho\mathbf{v}\mathbf{v}\nu dS = \sum\mathbf{f}. \tag{3.9}$$

Näher betrachtet können wir die rechte Seite der Gleichung (3.8) in folgende Komponenten aufschlüsseln

$$\sum\mathbf{f} = \underbrace{\int_{V_V}\rho\mathbf{a}dV}_{(3.10a)} + \underbrace{\int_{S_V}\mathsf{T}\nu dS}_{(3.10b)}. \tag{3.10}$$

Die Kraftkomponenten aus der Gleichung (3.10) können wir in zwei Kategorien, (3.10a) Volumenkräfte [6] und (3.10b) Oberflächenkräfte , aufteilen.
Die Volumenkraft (3.10a) setzt sich beispielsweise aus Gravitations-, Zentrifugal- und Corioliskraft zusammen. Die Volumenkräfte wirken, über das Produkt der Dichte ρ und der Beschleunigung $\mathbf{a}$ eine Kraft pro Volumen Einheit, auf das Fluid aus. Die auf das Kontrollvolumen V_V resultierende Kraft lässt sich damit als Integral über das Kontrollvolumen schreiben, wie wir es in dem Term der Gleichung (3.10a) sehen.
Die Oberflächenkräfte (3.10b) setzen sich zum Beispiel aus Druck- und Reibungskräften zusammen. Bei den Euler-Gleichungen betrachten wir nur die Druckkräfte, da die inneren Reibungskräfte des Fluids nicht berücksichtigt werden. Die Berücksichtigung der Reibungskräfte erweitert die Euler-Gleichungen zu den Navier-Stokes-Gleichungen. Diese Erweiterung werden wir anschließend im Abschnitt 3.1.2 *Navier-Stokes-Gleichungen* durchführen. Die Druckkraft wirkt in Richtung der äußeren Normalen ν auf die Oberfläche S_V des Kontrollvolumens. Erzeugt wird sie durch den Druck p. Da der Druck eine skalare Größe ist, wird er durch die äußere Normale ν in eine vektorielle Größe transformiert. Für die Druckkraft ergibt sich der Term (3.10b) zu:

$$\mathbf{f_p} = \int_{S_V} p\nu dS. \tag{3.11}$$

Ergänzende Ausführungen zu Volumenkräften und Oberflächenkräften findet man in den Büchern *Strömungsmechanik* von Oertel, Bohle und Dohrmann [OjBD06], *Grenzschicht-Theorie* von Schlichting und Gersten [SG05], *Strömungslehre* von Spurk und Aksel [SA07], *Fluidmechanik* Band 1 und Band 2 von Truckenbrodt [Tru08a], [Tru08b] und *Grundzüge der Strömungslehre* von Zierep und Bühler [ZB08]. Damit ergibt sich eine Differentialform des Impulssatzes durch Anwendung der Leibnizregel für Parameterintegrale und des Gaußschen Integralsatzes analog zur Bestimmung der Differentialform der Massenerhaltung.

$$\frac{\partial(\rho\mathbf{v})}{\partial t} + \mathbf{div}\ \mathbf{v}(\rho\mathbf{v}) + \mathbf{v}\cdot\nabla(\rho\mathbf{v}) + \nabla p = \rho\mathbf{a} \tag{3.12}$$

Wie bei der Massenerhaltung gilt die Darstellung (3.12) für kartesische Koordinaten.

Energieerhaltung

Damit wir den vollständigen Satz an Euler-Gleichungen formulieren können, fehlt uns noch

[6] Die Volumenkräfte werden in der Literatur auch gerne Massenkräfte genannt.

die Berücksichtigung der in der Temperatur und Druck enthaltenen Energie. Diese Gleichung folgt im wesentlichen aus dem *ersten Hauptsatz der Thermodynamik*. Wir erhalten die Gleichung durch erneutes Betrachten der generischen Erhaltungsgleichung (3.4) in dem wir ϕ auf die spezifische Enthalpie h setzen. Die Geschwindigkeitskomponente $\mathbf{v}$ beschreibt die Differenz der Fluid- und der Oberflächen-Geschwindigkeit.
Die durch die Geschwindigkeit im System befindliche Energie muss nicht durch das Aufstellen der entsprechenden Gleichungen beschrieben werden, da diese nicht von den Impuls-Gleichungen unabhängig ist. Durch Lösen der Massenerhaltung, Impulserhaltung und der (Thermischen-) Energieerhaltung, wird die durch die Geschwindigkeit beschriebene Energieerhaltung bereits erfüllt. Hierzu sei auf [Ins01] verwiesen.

$$\frac{\partial}{\partial t}\int_{V_M}\rho h dV = \frac{\partial}{\partial t}\int_{V_V}\rho h dV + \int_{S_V}\rho h \mathbf{v}\nu dS \tag{3.13}$$

Wir betrachten die Gleichung (3.13) nun ausschließlich für ein Kontrollvolumen V_V mit festem Volumen. Das *Fouriersche Gesetz* beschreibt den diffusiven Transport der Wärmeenergie und ergibt somit die rechte Seite der Gleichung (3.13). Weitere Terme, die bei der Wärmeenergie eine Rollenspielen können, sind die durch Viskosität und Druck geleistete Arbeit. Da wir in diesem Abschnitt die Euler-Gleichungen herleiten, entfällt hier der Term für die durch die Viskosität verrichtete Arbeit. Damit ergibt sich folgender Ausdruck

$$\frac{\partial}{\partial t}\int_{V_V}\rho h dV + \int_{S_V}\rho h \mathbf{v}\nu dS = \frac{\partial}{\partial t}\int_{V_V} p dV + \int_{S_V}\kappa\nabla T \nu dS \tag{3.14}$$

wobei κ die Wärmeleitfähigkeit und T die Temperatur beschreibt. Ersetzt man nun in Gleichung (3.14) die Enthalpie durch die Temperatur und führt zusätzlich die Prandtl-Zahl[7] Pr und die spezifische Wärmekapazität c_p ein, ergibt sich damit

$$\frac{\partial}{\partial t}\int_{V_V}\rho T dV + \int_{S_V}\rho T \mathbf{v}\nu dS = \frac{\partial}{\partial t}\int_{V_V}\frac{p}{c_p}dV + \int_{V_V} < \mathbf{v}, \nabla p > dV + \int_{S_V}\frac{c_p}{\mathsf{Pr}}\nabla T \nu dS, \tag{3.15}$$

der thermodynamische Beitrag zur Euler-Gleichung in Integral Form. Die Differential Form lautet

$$\frac{\partial \rho T}{\partial t} + \mathbf{div}\,(\rho T \mathbf{v}) = \frac{1}{c_p}\frac{\partial p}{\partial t} + < \mathbf{v}, \nabla p > + \mathbf{div}\,(\frac{c_p}{\mathsf{Pr}}\nabla T). \tag{3.16}$$

Die Gleichung (3.16) könnten wir noch um Terme zur Beschreibung von Wärmequellen[8] erweitern, die zusätzliche Energie von außen in dem System zuführen. Es sei noch angemerkt, dass die Terme, die der Druck zur Energie zuführt, bei kompressiblen Fluiden im Allgemeinen vernachlässigt werden kann, siehe [FP08].

Definition 3.1.5 (Dimensionslose Kennzahl)
Dimensionslose Kennzahlen ergeben sich als ein Produkt oder Quotient von physikalischen Größen, sodass sich die Einheiten kürzen und eine Kennzahl entsteht, die nicht dimensionsabhängig ist.

[7] Benannt nach dem Physiker Ludwig Prandl (1875-1953),
[8] Eine Wärmequelle kann auch dem System Energie entziehen, in diesem Fall hat der Term ein negatives Vorzeichen.

Bemerkung 3.1.6 (Dimensionslose Kennzahl)
Die dimensionslosen Kennzahlen beschreiben Charakteristiken von physikalischen Vorgängen, die unabhängig von der betrachteten Größe sind. Vorgänge mit dimensionslosen Kennzahlen gleicher beziehungsweise ähnlicher Größen haben vergleichbares "ähnliches" physikalisches Verhalten. Dieses Verhalten wird mit der Ähnlichkeitstheorie beschrieben.
Bekannte dimensionslose Kennzahlen sind zum Beispiel die Prandtl-Zahl Pr*, die Reynolds-Zahl* Re *und die Mach-Zahl* Ma.

Ein Zugang zum Thema der dimensionslosen Kennzahlen ergibt sich über die Entdimensionalisierung von mathematischen Modellen [Wec05] oder das Buckinghamsche Π-Theorem, siehe [Buc14] und [Buc15].

Die Gleichungen (3.7), (3.12) und (3.16) ergeben zusammen die Euler-Gleichungen:

$$\frac{\partial \rho}{\partial t} + \mathbf{div}\ \rho v = 0, \tag{3.17a}$$

$$\frac{\partial(\rho \mathbf{v})}{\partial t} + \mathbf{div}\ \mathbf{v}(\rho \mathbf{v}) + \mathbf{v} \cdot \nabla(\rho \mathbf{v}) + \nabla p = \rho \mathbf{a}, \tag{3.17b}$$

$$\frac{\partial \rho T}{\partial t} + \mathbf{div}\ (\rho T \mathbf{v}) = \frac{1}{c_p}\frac{\partial p}{\partial t} + < \mathbf{v}, \nabla p > + \mathbf{div}\ (\frac{c_p}{\mathsf{Pr}} \nabla T). \tag{3.17c}$$

Wenn wir in den Euler-Gleichungen (3.17) die Inkompressibilität des Fluides voraussetzen, vereinfachen sich die Euler-Gleichungen zu:

$$\mathbf{div}\ v = 0, \tag{3.18a}$$

$$\rho \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla(\rho \mathbf{v}) + \nabla p = \rho \mathbf{a}, \tag{3.18b}$$

$$\frac{\partial \rho T}{\partial t} + \mathbf{div}\ (\rho T \mathbf{v}) = \mathbf{div}\ (\frac{c_p}{\mathsf{Pr}} \nabla T). \tag{3.18c}$$

Die Euler-Gleichungen sind ein Modell zur Beschreibung der Strömung von reibungsfreien Fluiden. Durch die Annahme der Nicht-Viskosität des Fluides können an festen Wänden keine Haft-Randbedingungen (No-Slip-Condition) betrachtet werden. Da sich bei Gleit-Randbedingungen (Slip-Condition) keine Grenzschicht ausbildet, muss man die Randbereiche nicht so fein auflösen, wie bei den Navier-Stokes-Gleichungen. Dies reduziert den Simulationsaufwand deutlich.
Das Euler-Modell eignet sich gut zur Simulation von kompressiblen Fluiden bei hohen Mach-Zahlen und zur Vorhersage von Auftriebskräften von umströmten Körpern. Zur Berechnung des Strömungswiderstandes ist das Modell aufgrund der vernachlässigten inneren Reibung nicht so gut geeignet.
Die Euler-Gleichungen (3.18) bestehen aus vier gekoppelten partiellen Differentialgleichungen im zwei-dimensionalen Fall. Im drei-dimensionalen Fall sind es dann fünf gekoppelte partielle Differentialgleichungen. Der Term der Massenerhaltung (3.18a) ändert sich bei der Erweiterung des Modells zu den Navier-Stokes-Gleichungen nicht. Bei den Impulsgleichungen (3.18b) und dem Temperatur-Term (3.18c) müssen wir noch zusätzlich die Viskosität berücksichtigen.

Bemerkung 3.1.7 (Reibungsfreie Strömungen)
Abgesehen von den Euler-Gleichungen kann man reibungsfreie (nichtviskose) Strömungen mit der Potentialströmung beschreiben. Die Potentialströmung gehört zu einem der einfachsten

Modellen zur Modellierung von Strömungen. Im Rahmen dieser Arbeit möchten wir auf eine Herleitung der Potentialströmung verzichten und auf die Bücher von Herwig [Her08], Truckenbrodt [Tru08b], Zierep und Bühler [ZB08] verweisen. Nachfolgende Potentialströmungen setzen immer eine Inkompressibilität des Fluides voraus.
Das Vektorfeld der Geschwindigkeit $\mathbf{v}$ *muss die Voraussetzung der Rotationsfreiheit erfüllen, das heißt:*

$$\nabla \times \mathbf{v} = 0. \tag{3.19}$$

Wenn die Bedingung (3.19) *erfüllt ist, existiert ein Geschwindigkeitspotential* Φ*. Der Vektor der Geschwindigkeit* $\mathbf{v}$ *ergibt sich dann zu*

$$\mathbf{v} = -\nabla\Phi \tag{3.20}$$

Die Kontinuitätsgleichung $\mathbf{div}\ \mathbf{v} = 0$ *wird zur Laplace-Gleichung und durch Integration des Impulses ergibt sich dann die algebraische Bernoulli-Gleichung.*
Die Potentialströmung ist ein Spezialfall der Euler-Gleichungen (3.18) *für ideale Fluide*[9]*. Für nicht ideale Fluide ist die Potentialströmung nicht allzu realistisch, da es gewisse nichtphysikalische Phänomene gibt, siehe das D'Alembert-Paradoxon [GPF08].*

3.1.2 Navier-Stokes-Gleichungen

Nachdem wir nun die Euler-Gleichungen (3.17), beziehungsweise (3.18) detailliert hergeleitet haben, erweitern wir sie nun zu den Navier-Stokes-Gleichungen. Wir beschränken uns im Folgenden bei der Herleitung immer auf den Fall eines inkompressiblen Fluid. Somit können wir uns auf die Form (3.18) der Euler-Gleichungen beschränken.
Das Navier-Stokes-Modell entsteht, wenn wir zusätzlich noch innere Reibung mitmodellieren. Die innere Reibung wird durch die Viskosität hervorgerufen.

Definition 3.1.8 (Viskosität)
Der Begriff der Viskosität beschreibt die Zähflüssigkeit eines Fluides. Bei den von uns betrachteten newtonschen Fluiden ist die Viskosität bezüglich der Geschwindigkeit unabhängig.
Zur formelle Definition der **dynamischen Viskosität** η *betrachten wir den in Figur 3.2 skizzierten Versuchsaufbau.*

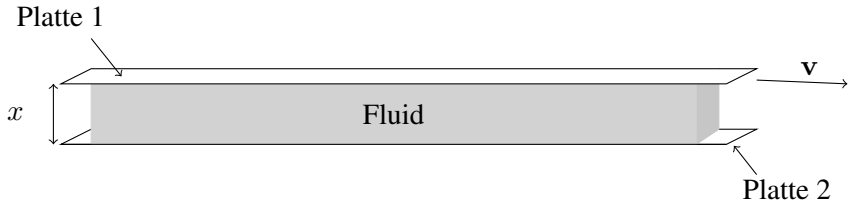

Abbildung 3.2: Versuchsaufbau zur Viskositätsbestimmung.

Die Platten haben die Kontaktfläche A *zum Fluid. Die Platte 1 ziehen wir mit der Geschwindigkeit* $\mathbf{v}$ *parallel zur Platte 2 nach rechts weg, die Platte 2 verbleibt an ihrer Position. Der*

[9] Ein ideales Fluid besitzt folgende Eigenschaften: keine innere Reibung, Inkompressibilität, keine Oberflächenspannung und Schwerelosigkeit.

vertikale Abstand der Platten hat den Abstand x.
Durch Experimente ergeben sich folgende Zusammenhänge für die Kraft F:

$$\begin{aligned} F &\approx A, \\ F &\approx \mathbf{v}, \\ F &\approx \tfrac{1}{x}. \end{aligned}$$

Damit ergibt sich als Gleichung für dynamische Viskosität η:

$$\eta := \frac{F \cdot x}{A \cdot \mathbf{v}}. \tag{3.21}$$

Die Einheit der dynamischen Viskosität ergibt sich damit zu $M \cdot L^{-1} \cdot T^{-1}$.
Damit ist die Viskosität eine Größe, die die Zähflüssigkeit eines Fluides angibt, wobei eine kleine Viskosität ein dünnflüssiges und eine große ein zähflüssiges Fluid charakterisieren.
Die **kinematische Viskosität** *ν entsteht durch Normierung der* **dynamischen Viskosität** *η mit der Dichte ρ. Damit gilt*

$$\nu := \frac{\eta}{\rho}. \tag{3.22}$$

Die Einheit der kinematischen Viskosität ergibt sich damit zu $L^2 \cdot T^{-1}$.

Analog zur Euler-Gleichung betrachten wir zunächst: Die Massenerhaltung, die Impulserhaltung und die Energieerhaltung.

Massenerhaltung

Bei der Herleitung der Massenerhaltung für das Modell der Euler-Gleichungen haben wir keine vereinfachten Annahmen bezüglich der Viskosität gemacht. Somit können wir die Kontinuitätsgleichung (3.18a) unverändert in das Modell der Navier-Stokes-Gleichungen übernehmen.

Impulserhaltung

Da der Impuls die wirkende Kraft pro Zeit ist, ergibt sich direkt, dass wir die Gleichungen der Impulserhaltung (3.18b) der Euler-Gleichungen für Viskose-Fluide erweitern müssen. Dazu betrachten wir die Gleichung (3.10) erneut. Der erste Term (3.10a) repräsentiert die wirkenden Volumenkräfte des Fluides. Bei den Oberflächenkräften (3.10b) haben wir bei den Euler-Gleichungen die durch Scherung entstehenden Reibungskräfte aufgrund der Nichtviskosität nicht berücksichtigt. Dies holen wir nun nach. Dazu betrachten wir als erstes die auf die Oberfläche des Kontrollvolumen wirkenden Kräfte anhand der Grafik 3.3.
Die Figur 3.3 ist an die Grafik *Bild 3.105* im Buch *Grundzüge der Strömungslehre* von Zierep und Bühler [ZB08] angelehnt.
Die, in Figur 3.3, in Normalen-Richtung auf die Oberfläche des Kontrollvolumen wirkenden Kräfte ν werden durch den Druck p auf die Oberfläche induziert. Da wir diese Kräfte schon für die Euler-Gleichungen in dem Term (3.11) analysiert haben, können wir uns direkt den Scherkräften σ zuwenden.
Der Tensor der Oberflächenkräfte hat dann die Form:

$$\mathsf{T} = -p\mathbb{I} - (2/3\ \eta \mathbf{div}\ \mathbf{v})\mathbb{I} + 2\eta\mathsf{D}, \tag{3.23}$$

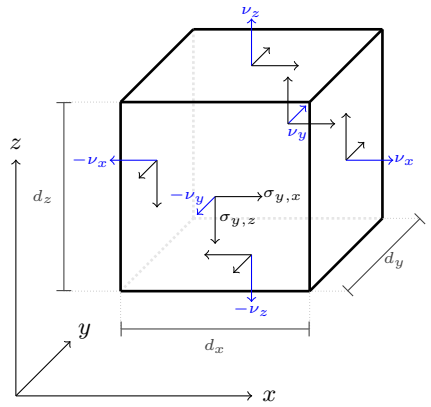

Abbildung 3.3: Auf ein Kontrollvolumen wirkende Scher- und Druckkräfte.

$\mathbb{I}$ ist dabei die Einheitsmatrix, die Gestalt der Matrix D ist in der Gleichung (3.24) gegeben.

$$\mathsf{D} = {}^1\!/_2 \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^T\right] \tag{3.24}$$

Die Tensoren (3.23) und (3.24) haben in Komponenten-Schreibweise folgende Darstellung:

$$\mathsf{T}_{\mathrm{i,j}} = -(p + {}^2\!/_3\, \eta \mathbf{div}\ \mathbf{v})\delta_{ij} + 2\eta \mathsf{D}_{\mathrm{i,j}} \tag{3.25}$$

$$\mathsf{D}_{i,j} = {}^1\!/_2 \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{x_i}\right). \tag{3.26}$$

Der zweite Term in der Gleichung (3.23) entfällt bei inkompressiblen Fluiden, da dort Divergenzfreiheit herrscht, siehe Gleichung (3.18a).
Der Faktor ${}^{-2}\!/_3$ im kompressiblen Viskositätsterm der Gleichung (3.23) ist ein Skalierungsparameter, der zur Berechnung der Kraft über die dynamische Viskosität η benötigt wird. Eine Analyse zur Ermittlung des Faktors ${}^{-2}\!/_3$ ist zum Beispiel im Werk von Stokes [Sto45] zu finden.
Für die Komponenten-Darstellung des Tensors (3.25) gilt obige Betrachtung für den kompressiblen Viskositätsterm analog.
Die Impulserhaltungsgleichung für die Navier-Stokes-Gleichungen ist damit:

$$\frac{\partial(\rho \mathbf{v})}{\partial t} + \mathbf{div}\ \mathbf{v}(\rho \mathbf{v}) + \mathbf{v} \cdot \nabla(\rho \mathbf{v}) + \nabla p + \eta \mathbf{div}\ ({}^2\!/_3\ (\mathbf{div}\ \mathbf{v})\mathbb{I} - {}^1\!/_2\ [\nabla \mathbf{v} + (\nabla \mathbf{v})^T]) = \rho \mathbf{a} \tag{3.27}$$

Die Gleichung (3.27) repräsentiert die Impulserhaltung in kartesischen Koordinaten.

Energieerhaltung

Zuletzt müssen wir noch die Energieerhaltung für ein viskoses Fluid formulieren. Dazu erweitern wir die Energieerhaltungsgleichung (3.15) für nichtviskose Strömungen um einen Term, der die in der Viskosität enthaltene Energie repräsentiert. Für diesen Term gilt:

$$\int_{V_V} \mathsf{S} : \nabla \mathbf{v} dV \tag{3.28}$$

Der Term (3.28) [10] beschreibt den kompressiblen Anteil, der in der Viskosität enthaltenen Energie. Für den Tensor S gilt: $S = T + p\mathbb{I}$. Er repräsentiert den Viskosen-Anteil des Spannungstensors T, siehe Gleichung (3.23).
Die Gleichung (3.15) um den Term (3.28) erweitert, ergibt die vollständige Energieerhaltung für die Navier-Stokes-Gleichungen in der Integralform:

$$\begin{aligned}\frac{\partial}{\partial t}\int_{V_V}\rho T dV + \int_{S_V}\rho T\mathbf{v}\nu dS =& \frac{\partial}{\partial t}\int_{V_V}\frac{p}{c_p}dV + \\ & \int_{V_V} < \mathbf{v}, \nabla p > dV + \int_{V_V}\mathsf{S} : \nabla\mathbf{v}dV + \int_{S_V}\frac{c_p}{\mathsf{Pr}}\nabla T\nu dS, .\end{aligned} \tag{3.29}$$

Die Differentialform von Gleichung (3.29) für kartesische Koordinaten ist:

$$\begin{aligned}\frac{\partial \rho T}{\partial t} + \mathbf{div}\,(\rho T\mathbf{v}) =& \frac{1}{c_p}\frac{\partial p}{\partial t} + \\ & < \mathbf{v}, \nabla p > + S : \nabla v + \mathbf{div}\,(\frac{c_p}{\mathsf{Pr}}\nabla T).\end{aligned} \tag{3.30}$$

Den viskosen Energie-Anteil kann man bei nicht kompressiblen Strömungen nach dem Buch von Ferziger und Perić [FP08] vernachlässigen.

Nun können wir die Navier-Stokes-Gleichungen als System von partiellen Differentialgleichungen angeben. Zuerst formulieren wir in der Gleichung (3.31) das Strömungsmodell von Navier und Stokes für ein beliebiges kompressibles newtonsches Fluid.

$$\frac{\partial \rho}{\partial t} + \mathbf{div}\,\rho v = 0 \tag{3.31a}$$

$$\begin{aligned}&\frac{\partial(\rho\mathbf{v})}{\partial t} + \mathbf{div}\,\mathbf{v}(\rho\mathbf{v}) + \mathbf{v}\cdot\nabla(\rho\mathbf{v}) + \nabla p + \\ &\eta\mathbf{div}\,(2/3\,(\mathbf{div}\,\mathbf{v})\mathbb{I} - 1/2\,[\nabla\mathbf{v} + (\nabla\mathbf{v})^T]) = \rho\mathbf{a}\end{aligned} \tag{3.31b}$$

$$\begin{aligned}\frac{\partial \rho T}{\partial t} + \mathbf{div}\,(\rho T\mathbf{v}) =& \frac{1}{c_p}\frac{\partial p}{\partial t} + \mathbf{div}\,(\frac{c_p}{\mathsf{Pr}}\nabla T) + \\ & < \mathbf{v}, \nabla p > + S : \nabla v\end{aligned} \tag{3.31c}$$

Im Nachfolgenden vereinfachen wir die Gleichung (3.31) durch die Einschränkung auf ein inkompressibles newtonsches Fluid.

$$\mathbf{div}\,v = 0 \tag{3.32a}$$

$$\frac{\partial(\rho\mathbf{v})}{\partial t} + \mathbf{v}\cdot\nabla(\rho\mathbf{v}) + \nabla p - \eta\Delta\mathbf{v} = \rho\mathbf{a} \tag{3.32b}$$

$$\frac{\partial \rho T}{\partial t} + \mathbf{div}\,(\rho T\mathbf{v}) = \frac{1}{c_p}\frac{\partial p}{\partial t} + \mathbf{div}\,(\frac{c_p}{\mathsf{Pr}}\nabla T). \tag{3.32c}$$

[10] Der : Operator steht für das Tensor Skalarprodukten. Das Tensor-Skalarprodukt ist wie folgt definiert: $\mathsf{A} : \mathsf{B} = \sum_{i=1}^{n}\sum_{j=1}^{m}\mathsf{A}_{i,j}\mathsf{B}_{i,j}$ und $\mathsf{A}, \mathsf{B} \in \mathbb{R}^{n\times m}$.

3.2 Numerische Grundlagen

Dieser Abschnitt befasst sich mit den numerischen Grundlagen zum Lösen der im vorherigen Abschnitt 3.1.2 aufgestellten Navier-Stokes-Gleichungen (3.31), beziehungsweise (3.32). Im Folgenden bezeichnen wir mit den Navier-Stokes-Gleichungen die inkompressiblen Navier-Stokes-Gleichungen, es sei denn wir reden explizit von den kompressiblen Navier-Stokes-Gleichungen.
Es gibt zwei unterschiedliche Vorgehensweisen um partielle Differentialgleichungen zu lösen. Man betrachtet die Variationsformulierung [11] oder die Differentialformulierung. Das Lösen in der Differentialformulierung realisiert man mit der Finiten-Differenzen-Methode (FDM). Für das Lösen des Problems in der Variationsformulierung gibt es eine ganze Anzahl an Verfahren: die Finite-Elemente-Methoden (FEM), Finite-Volumen-Methoden (FVM) und die Waveletmethoden. In dieser Arbeit gehen wir kurz auf die Finite-Differenzen-Methode, die Finite-Elemente-Methode und die Finite-Volumen-Methode ein.

3.2.1 Finite-Differenzen-Methode

Zuerst definieren wir die Finiten-Differenzen formal, anschließend befassen wir uns mit der Anwendung von Finiten-Differenzen zum Lösen von (partiellen) Differentialgleichungen . Die Nachteile und Probleme von Finiten-Differenzen werden wir auch näher betrachten.
Eine Finite-Differenz soll den Wert der Ableitung einer Funktion an der Stelle x_0 approximieren. Damit ist folgende Definition 3.2.1 naheliegend.

Definition 3.2.1 (Differenzenquotient)
Wir betrachten eine Funktion $f(x) \in \mathcal{C}^\ell(\Omega)$ und es gilt $\Omega \subset \mathbb{R}^n$ und $\ell, n \in \mathbb{N}$. Nun sei $x_0 \in \Omega$. Damit ergibt sich durch

$$f^{FD+}(x_0) := \frac{f(x_0+h) - f(x_0)}{h}, \tag{3.33}$$

der Vorwärtsdifferenzenquotient. Analog dazu ist durch

$$f^{FD-}(x_0) := \frac{f(x_0) - f(x_0-h)}{h}, \tag{3.34}$$

der Rückwärtsdifferenzenquotient definiert. Als dritte übliche Variante folgt der Zentraledifferenzenquotient

$$f^{FD}(x_0) := \frac{f(x_0+h) - f(x_0-h)}{2h}. \tag{3.35}$$

In der Abbildung 3.4 sehen wir die Differenzenquotienten grafisch dargestellt. Der Vorwärtsdifferenzenquotient ist an der Stelle x_0, der Rückwärtsdifferenzenquotient bei x_1 und der Zentraledifferenzenquotient ist bei x_3 abgebildet.
Nun befassen wir uns mit der Analyse der Approximationsgüte der drei Differenzenquotienten (3.33), (3.34) und (3.35).

Satz 3.2.2 (Approximationsordnung Finite-Differenzen)
Sei $x \in \mathbb{R}$, $h > 0$ und $\Omega := (x-h, x+h)$. Dann gilt, für eine Funktion $f(x) \in \mathcal{C}^2(\bar{\Omega})$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + hR, \tag{3.36}$$

[11] Die Variationsformulierung erhält man durch Integration der Differentialformulierung über das betrachtete Gebiet und entspricht auch der Integralformulierung.

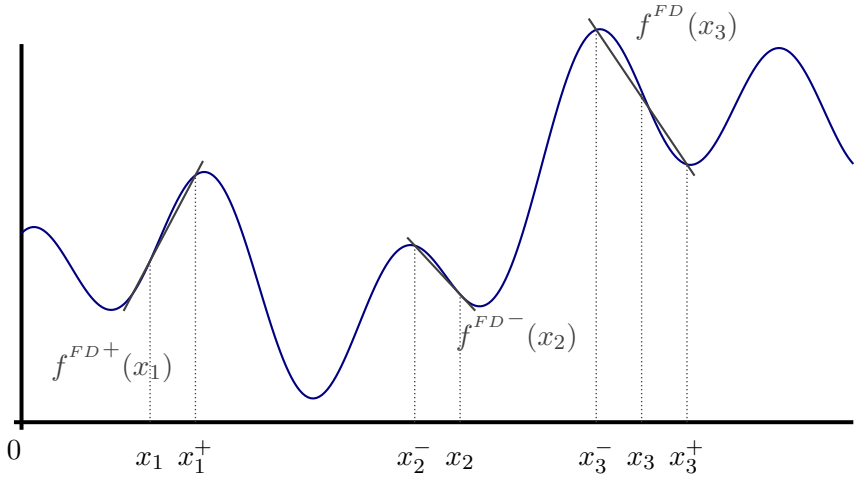

Abbildung 3.4: Grafische Darstellung der Differenzenquotienten.

beziehungsweise

$$f'(x) = \frac{f(x) - f(x-h)}{h} + hR, \tag{3.37}$$

für R gilt $R \leq \frac{1}{2}\|f''\|_{\infty,\Omega}$.
Die Gleichung (3.36) *ist die Fehlerabschätzung des Vorwärtsdifferenzenquotient und Gleichung* (3.37) *die des Rückwärtsdifferenzenquotienten.*
Nun sei $f \in \mathcal{C}^3(\bar{\Omega})$ dann gilt:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + h^2R. \tag{3.38}$$

Die Gleichung (3.38) *ist die Fehlerabschätzung zu den zentralen Differenzenquotienten. Für das Restglied gilt $R \leq \frac{1}{6}\|f'''\|_{\infty,\Omega:}$.*

Beweis. Der Beweis für alle drei Aussagen verwendet die Taylor-Entwicklung. Wir zeigen nun die Aussage für den Vorwärtsdifferenzenquotienten.

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots \tag{3.39}$$

Wir stellen nun (3.39) nach $f'(x)$ um und wenden darauf den Mittelwertsatz an. Damit ergibt sich die Aussage. Für die anderen beiden Aussagen ist das Vorgehen analog dazu. □

Das Theorem 3.2.2 findet man in ähnlicher Form zum Beispiel im Buch *Numerik partieller Differentialgleichungen* von P. Knabner und L. Angermann [KA00].

Bemerkung 3.2.3 (Differenzenquotienten höhrer Ordnung)
Analog zur Definition 3.2.1 können wir auch Differenzenquotienten höherer Ordnung erklären. Damit ergibt sich für eine Funktion $f \in \mathcal{C}^4(\Omega)$ der Differenzenquotient zweiter Ordnung zu:

$$f^{FD''}(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{3.40}$$

Analog zu Satz 3.2.2 können wir zeigen, dass

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + h^2 R, \tag{3.41}$$

für $R \leq \frac{1}{12}\|f^{(4)}\|_{\infty,\Omega}$ *gilt.*

Die Voraussetzungen, dass der Satz 3.2.2 gilt, sind sehr stark, da wir eine um bis zu zwei Stufen höhere Glattheit benötigen als wir später verwenden. Diese hohen Anforderungen sind mit ein Grund, dass wir die Finiten-Elemente- beziehungsweise Finiten-Volumen-Methoden zur Simulation von partiellen Differentialgleichungen im Anschluss an die Finiten-Differenzen betrachten.

Beispiel 3.2.4 (FD)
Wir betrachten folgendes Problem:

$$\frac{\partial}{\partial t} f(\mathbf{x}, t) - a\Delta f(\mathbf{x}) = f(\mathbf{x}), \tag{3.42}$$

die inhomogene Wärmeleitungsgleichung[12]*. Wir betrachten die eindimensionale Wärmeleitungsgleichung* (3.42)*,* $\mathbf{x} = x \in \Omega := (0,1)$ *und* $t \in [0,T]$.

Wir lösen das Problem (3.42) *in Zeit-Richtung mit dem expliziten Euler-Verfahren und die Ortsvariable mit Finiten-Differenzen auf. Für die Zeitdiskretisierung verwenden wir* $u^\ell := u(t_\ell)$*, wobei* $u_\ell = \ell \cdot \tau$ *linksseitige Differenzenquotienten sind. Damit folgt:*

$$u^{\ell+1} \approx \tau f + u^\ell + a\tau\Delta u^\ell(x). \tag{3.43}$$

Für die Diskretisierung der Ortsvariablen schreiben wir $u_i = u(x_i)$ *und* $x_i = x_0 + i \cdot h$*. Den diskreten Laplace-Operator können wir dann als:*

$$\Delta u_i = \frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1}), \tag{3.44}$$

schreiben. Damit ergibt sich aus den Gleichungen (3.43) *und* (3.44)*:*

$$u_i^{\ell+1} = (1 - {}^{a\tau}\!/_{h^2})u_i^\ell + {}^{a\tau}\!/_{h^2}\,(u_{i-1}^\ell u_{i+1}^\ell) + \tau f_i. \tag{3.45}$$

In der Abbildung 3.5 sehen wir das Ergebnis der Finiten-Differenzen-Simulation des Problems (3.42) *für die Parameter* $x \in (0,1)$*,* $t \in (0,1)$*,* $a = 1$*,* $f(x) = 0$*,* $u(0,x) = 0$ *für* $x \in (0,1)$*,* $u(t,0) = 0$ *und* $u(t,1) = 1$.

Betrachten wir nun das Vorgehen, wenn wir die Navier-Stokes-Gleichungen mit Finiten-Differenzen approximieren. Analog zu unserem Beispiel 3.2.4 müssen wir eine Ortsdiskretisierung und eine Zeitdiskretisierung durchführen. Bei der Betrachtung für die Navier-Stokes-Gleichungen mit Finiten-Differenzen beschränken wir uns im Folgenden auf den zwei-dimensionalen Fall eines inkompressiblen Fluides ohne die Energiegleichung (3.32c).

[12] Die Wärmeleitungsgleichung ist eine lineare parabolische Partielle Differentialgleichung zweiter Ordnung.

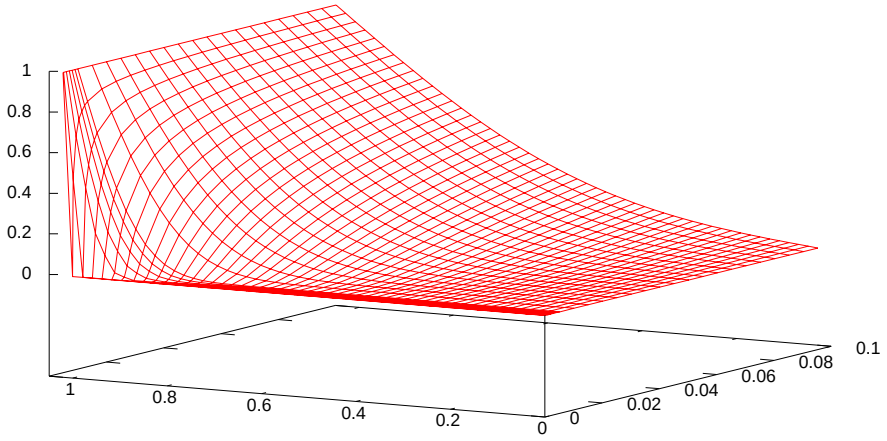

Abbildung 3.5: Numerische FD-Lösung des Problems (3.42).

Für die Ortsdiskretisierung der Navier-Stokes-Gleichung (3.32) müssen wir den Geschwindigkeitsvektor $\mathbf{v} := (v_1, v_2)^T$ und die skalare Größe des Drucks p diskretisieren. Um ein Oszillieren zu vermeiden verwenden wir ein versetztes Gitter (engl. staggered grid). Ein Beispiel zu dem Oszillationsverhalten bei nicht versetzten Gittern finden wir in *Numerische Simulation in der Strömungsmechanik, eine praxisorientierte Einführung* [GDN95] und *Automatic differentiation in flow simulation* [Lei07]. Auch bei der Simulation der Navier-Stokes-Gleichungen mit Finiten-Differenzen werden versetzte Gitter eingesetzt, siehe dazu zum Beispiel *A Compact Finite Difference Method on Staggered Grid for Navier-Stokes Flows* von Zhang, et. all [ZSMM06] oder *A Finite Difference Calculation Procedure for the Navier-Stokes Equations on a Staggered Curvilinear Grid* von R. M. Barron und B. Zogheib [RMB10].

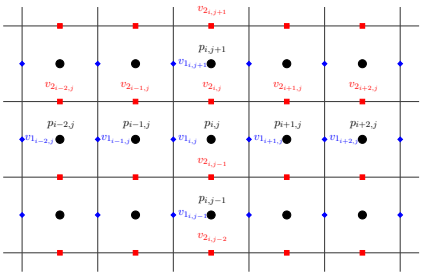

Abbildung 3.6: Versetztes Gitter.

Die Abbildung 3.6 zeigt uns ein mögliches Schemata für ein versetztes Gitter zur Simulation der Navier-Stokes-Gleichungen wie es auch bei *Numerische Simulation in der Strömungs-*

mechanik, eine praxisorientierte Einführung [GDN95] verwendet wird.
Für die Zeitdiskretisierung gibt es zwei naheliegende Vorgehensweisen, entweder das explizite oder das implizite Euler-Verfahren[13]. Das explizite Euler-Verfahren ist ein Einschrittverfahren mit
Konsistenz- und Konvergenzordnung eins. Weitere Informationen zu dem Verfahren findet man in den Büchern *Einführung in die Numerische Mathematik II* von Stoer und Bullrisch [SB00] und *Numerische Mathematik II* von Deufelhard und Bornemann [DB02]. Die Größe des Zeitschrittes müssen wir so wählen, dass für die CFL-Zahl[14]

$$c := \frac{u\Delta t}{\Delta x}, \tag{3.46}$$

die Bedingung $c < 1$ erfüllt ist. Die CFL-Zahl ist ein Maß um wieviele Zellen sich ein beobachteter Partikel pro Zeitschritt bewegt.
Damit ergibt sich folgende Formulierung zum Lösen der Navier-Stokes-Gleichungen

$$\mathsf{A}_p \mathbf{p}^{\ell+1} = \mathbf{F}^{\ell}. \tag{3.47}$$

Die Matrix A_p besitzt eine Band-Struktur, die das Finite-Differenzen Schema für den Druck abbildet. Die Randbedingungen des Drucks müssen in der Struktur der Matrix berücksichtigt werden. Der Druck Vektor $\mathbf{p}^{\ell+1}$ enthält für jeden Knoten des Gitters einen Eintrag. Nachdem wir $\mathbf{p}^{\ell+1}$ berechnet haben, können wir damit $\mathbf{v}^{\ell+1}$ berechnen.

Algorithm 3.2.5 Finite Differenzen Löser

1: **for** $iter = 1 \rightarrow maxIter$ **do**
2: stelle RHS für Druckgleichung auf
3: löse Druckgleichung
4: löse Geschwindigkeitsgleichung
5: **end for**

Die Randbedingungen haben wir in diesem Abschnitt nicht betrachtet. Möchten wir den Algorithmus 3.2.5 zu einer vollständigen Simulation erweitern, müssten dazu noch die Randbedingungen berücksichtigt werden.
Eine ausführliche Betrachtung zur Simulation der Navier-Stokes-Gleichungen mit Finiten-Differenzen finden wir im Buch *Numerische Simulation in der Strömungsmechanik* von M. Griebel, T. Dornseifer und T. Neunhoeffer [GDN95].

3.2.2 Finite-Elemente-Methode

In diesem Abschnitt betrachten wir, wie man mittels der Finiten-Elemente-Methode (FEM) eine numerische Lösung einer partiellen Differentialgleichung findet. Damit ist die Finite-Elementen-Methode eine weitere Möglichkeit, wie wir die Navier-Stokes-Gleichungen numerisch lösen können.
Für die Finite-Element-Methode betrachten wir zunächst die schwache Formulierung einer

[13] Das explizite Euler-Verfahren wird auch Eulersches Polygonzugverfahren genannt. Und ist nach Leonhard Euler, der es 1768 in dem Buch *Institutiones Calculi Integralis* [Eul68] vorgestellt hat, benannt.

[14] Die Bezeichnung CFL-Zahl kommt von der Ausgeschriebenen Form Courant-Friedrichs-Lewy-Zahl, manchmal wird sie auch nur Courant-Zahl genannt. Und ist nach Richard Courant (1888-1972), Kurt Friedrichs (1901-1982) und Hans Lewy (1904-1988) benannt, die sie 1928 definiert haben.

partiellen Differentialgleichung. Die schwache Formulierung ist über die schwache Ableitung erklärt.

Definition 3.2.6 (Schwache Ableitung)
Sei $\phi(x) \in \mathcal{C}_0^\infty(\Omega)$[15], $f, g \in L^2(\Omega)$ *und* $\Omega := (a, b),\ a, b \in \mathbb{R}$, *dann gilt:*

$$\int_\Omega g(t)\phi(t)dt = -\int_\Omega f(t)\phi'(t)dt \tag{3.48}$$

und wir nennen g *die* **schwache Ableitung** *(engl. weak derivative) von* f.
Gilt $f \in \mathcal{C}^1\Omega$, *dann ist* $g = f'$. *Dies folgt direkt aus Gleichung* (3.48) *mit der partiellen Integration und den Eigenschaften der Testfunktion* ϕ.

Mit der schwachen Ableitung gehen wir nun über zur schwache Formulierung.

Beispiel 3.2.7 (Schwache Formulierung Poisson-Problem)
Wir betrachten nun das Poisson-Problem:

$$\begin{aligned} -\Delta u(\mathbf{x}) &= f(\mathbf{x}), \mathbf{x} \in \Omega \\ \nu \cdot \nabla u(\mathbf{x}) &= g, \mathbf{x} \in \delta\Omega. \end{aligned} \tag{3.49}$$

Die schwache Formulierung von Problem (3.49) *erhalten wir dann durch Multiplikation mit der Testfunktion* ϕ *und Integration über* Ω,

$$\int_\Omega \Delta f(\mathbf{x})\phi(\mathbf{x})dx = \int_\Omega f(\mathbf{x})\phi(\mathbf{x})dx. \tag{3.50}$$

Die Gleichung (3.50) *gilt für alle Testfunktionen* ϕ. *Partielle Integration von Gleichung* (3.50) *liefert die schwache Formulierung von* (3.49)*:*

$$\int_\Omega \nabla u(\mathbf{x})\nabla\phi(\mathbf{x})dx = \int_\Omega f(\mathbf{x})\phi(\mathbf{x})dx - \int_{\delta\Omega} g(\mathbf{x})\phi(\mathbf{x})dx, \tag{3.51}$$

für alle Testfunktionen $\phi \in \mathcal{C}_0^\infty(\Omega)$.

Bei der Finiten-Elementen-Methode betrachten wir die schwache Formulierung und verwenden spezielle Testfunktionen, die wir im Weiteren Ansatzfunktionen nennen. In der Abbildung 3.7 betrachten wir zwei übliche Ansatzfunktionen.

a) Lineares Finites Element b) Quadratisches Finites Element

Abbildung 3.7: Beispiel von Finiten-Elementen.

In Abbildung 3.7.a sehen wir eine eindimensionale lineare Ansatz Funktion. Abbildung 3.7.b zeigt uns eine eindimensionale quadratische Ansatz Funktion.
Wir betrachten nun das Poisson-Problem, das wir schon im Beispiel 3.2.7 betrachtet haben und lösen es mit der Finiten-Elemente-Methode.

[15] Eine Funktion $\phi(x) \in \mathcal{C}_0^\infty(\Omega)$ nennt man auch eine Testfunktion.

Beispiel 3.2.8 (FEM)
Die rechte Seite $f(x)$ der Gleichung setzen wir konstant auf den Wert 1 und die Randfunktion $g(x)$ auf den Wert 0. Für das Gebiet Ω nehmen wir das Einheitsquadrat. Damit erhalten wir folgendes Problem:

$$\begin{aligned} -\Delta u(\mathbf{x}) &= 2 \; \mathbf{x} \in \Omega \\ \nu \cdot \nabla u(\mathbf{x}) &= 1 \; \mathbf{x} \in \delta\Omega. \end{aligned} \tag{3.52}$$

Das Problem ist nur bis auf eine additive Konstante eindeutig bestimmt. Damit wir eine eindeutige Lösung erhalten, setzen wir $u(\mathbf{x}) = 0$, für alle $\mathbf{x} \in \delta\Omega$. Wir lösen das Problem (3.52) *mit den linearen Ansatz-Funktionen*

$$\phi_i(\mathbf{x}_j) = \phi_i(x_{j,1}, x_{j,2}) := \delta_{i,j}, \tag{3.53}$$

auf einem Courant-Dreiecksgitter wie wir es in Abbildung 3.8 sehen.

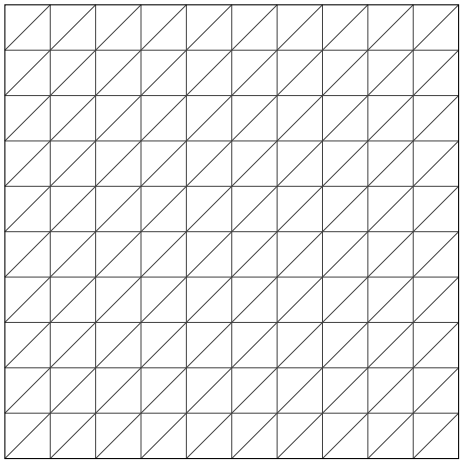

Abbildung 3.8: FEM-Dreicksgitter auf dem Intervall $[0, 1] \times [0, 1]$.

Damit erhalten wir folgendes lineares Gleichungssystem:

$$\mathsf{A}\mathbf{u} = \mathbf{f} \tag{3.54}$$

und es gilt $\mathsf{A} \in \mathbb{R}^{(n\cdot m)\times(n\cdot m)}$ und $\mathbf{u}, \mathbf{f} \in \mathbb{R}^{n\cdot m}$. Die Vektoren $\mathbf{u}$ und $\mathbf{f}$ sind folgendermaßen aufgebaut: $\mathbf{u}_{i+j\cdot n} = u(x_{i,j})$ und $\mathbf{f}_{i+j\cdot n} =< f, \phi_i >_0$. Mit der Ansatzfunktion (3.53) *und den Randbedingungen ergibt sich für die Matrix A folgende Struktur:*

$$\mathsf{A} := \begin{pmatrix} 0 & -1 & 0 & 0 & \cdots \\ -1 & 4 & -1 & 0 & \cdots \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots \end{pmatrix} \tag{3.55}$$

Da wir die Werte von $u(\mathbf{x})$ auf dem Rand auf den Wert Null gesetzt haben, müssen wir die Matrix nicht anpassen, da wir sogenannte "Geisterknoten" haben, die ein Anpassen der Matrix überflüssig machen. Die Einträge des Vektors f für die rechte Seite des Gleichungssystems (3.54) *sind durch das Integral $f_i = \int_\Omega f(x)\phi_i(x)dx$ gegeben. Wir lösen das Gleichungssystem* (3.54) *mit dem Gauß-Seidel-Verfahren. Die Lösung sehen wir in Abbildung 3.9.*

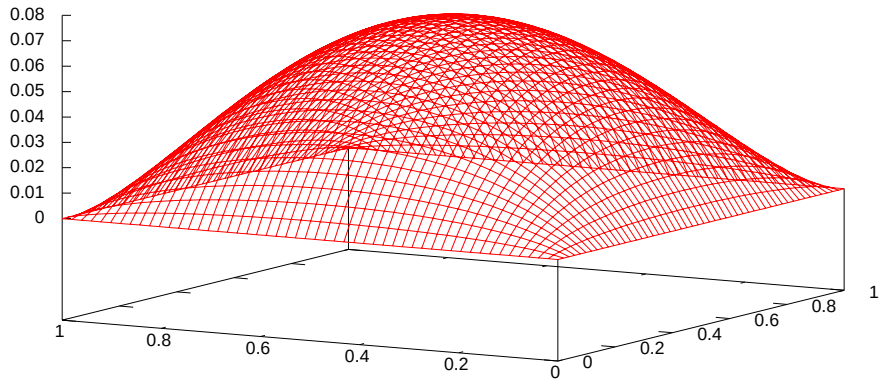

Abbildung 3.9: Lösung des Poisson-Problems (3.52).

Dieser Abschnitt stellt nur eine sehr kurze Einführung in die Finite-Elemente-Methode dar. Ausführlichere Betrachtungen zu dem Thema findet man zum Beispiel in den Büchern *Finite Elemente* von D. Braess [Bra00] oder in *Numerik partieller Differentialgleichungen* von P. Knabner und L. Angermann [KA00].
Die Behandlung der Navier-Stokes-Gleichungen mit der Finiten-Elementen-Methode findt man in folgenden Quellen: *Finite Element Methods for the Incompressible Navier-Stokes Equations* von Rannacher [Ran99] oder *Finite element methods for the incompressible Navier-Stokes equations* von Segal [Seg11].

3.2.3 Finite-Volumen-Methoden

Wir betrachten nun die Finite-Volumen-Methoden mit dem Fokus, dieses Verfahren zur numerischen Lösung der Navier-Stokes-Gleichungen zu verwenden. In diesem Abschnitt gehen wir zusätzlich noch auf die Programme Caffa und Comet ein. Beide Programme lösen die Navier-Stokes-Gleichung numerisch mit einer Finite-Volumen-Methode. Im Anschluss an diese Betrachtung gehen wir, im Abschnitt 3.3 auf die Anwendung des Automatischen Differenzierens im Bereich der Strömungssimulation ein.

Einführung

Die Finiten-Volumen-Methoden sind eine Klasse von numerischen Verfahren, basierend auf dem physikalischen Prinzip des Erhaltungssatzes zum numerischen Lösen von Erhaltungsgleichungen. Die folgende Einführung in die Finiten-Volumen-Methoden orientiert sich an der folgenden Literatur: *Numerische Strömungsmechanik* von Ferziger und Perić [FP08] und *Finite volume methods: foundation and analysis* von Barth und Ohlberger [BO04].

Wir betrachten dazu die generische Form der Erhaltungsgleichung (3.4), aus Abschnitt 3.1.1. Den Integral-Term über das Massenvolumen V_M können wir auch als den Transport unserer betrachteten Erhaltungsgröße ϕ, über den Rand unseres betrachteten Volumens, ohne die Konvektion auffassen. Wir berücksichtigen zusätzlich noch eine Quelle und Senke q_ϕ der beobachteten Größe ϕ. Damit ergibt sich aus Gleichung (3.4) folgender Ausdruck

$$\frac{\partial}{\partial t}\int_V \rho\phi dV + \int_S \rho\phi \mathbf{v}\nu\nu dS = \int_S \Gamma\nabla\phi\cdot\nu dS + \int_V q_\phi dV. \tag{3.56}$$

Nun vereinfachen wir die Gleichung (3.56), indem wir annehmen, der Term, der die zeitliche Änderung beschreibt, verschwindet. Somit erhalten wir die folgende Form der generischen Erhaltungsgleichung

$$\int_S \rho\phi\mathbf{v}\nu dS = \int_S \Gamma\nabla\phi\cdot\nu dS + \int_V q_\phi dV. \tag{3.57}$$

Definition 3.2.9 (Direkte Zerlegung)
Sei V nun ein beliebiges betrachtetes Volumen. Dann nennen wir eine Zerlegung V_I von V eine **direkte Zerlegung***, wenn die Bedingung* (3.58) *gilt.*

$$V_I := \left\{ V_i : i\in I := \{1,\dots,N\}, \bigcup_{i=1}^{N} V_i = V \; und \, V_i\cap V_j = \emptyset, \forall i\neq j \right\}. \tag{3.58}$$

Dann gilt für ein beliebiges betrachtetes Volumen V und der zugehörigen direkten Zerlegung V_I aufgrund der Linearität der Integrale folgender Zusammenhang:

$$\begin{aligned}&\int_S \rho\phi\mathbf{v}\nu dS = \\ &\sum_{i\in I}\int_{S_i}\rho\phi\mathbf{v}\nu dS = \sum_{i\in I}\int_{S_i}\Gamma\nabla\phi\cdot\nu dS + \sum_{i\in I}\int_{V_i} q_\phi dV \\ &\qquad\qquad = \int_S \Gamma\nabla\phi\cdot\nu dS + \int_V q_\phi dV.\end{aligned} \tag{3.59}$$

Da Gleichung (3.57) für beliebige Volumen gilt, folgt aus der Gleichung (3.59):

$$\int_{S_i}\rho\phi\mathbf{v}\nu\nu dS = \int_{S_i}\Gamma\nabla\phi\cdot\nu dS + \int_{V_i} q_\phi dV, \tag{3.60}$$

für alle $i\in I$. Damit können wir unser beobachtetes Gebiet mit dem Volumen V in eine direkte Zerlegung von Kontrollvolumen V_i zerlegen und und für jedes Kontrollvolumen gilt die Erhaltungsgleichung (3.60). Für die Kontrollvolumen V_i können wir die Integrale numerisch approximieren, da wir voraussetzen können, dass die Volumina der Kontrollvolumen V_i hinreichend klein sind. So können wir den durch die numerische Integration gemachten Fehler in einem vertretbaren Rahmen halten.
Wir zerlegen unser Gebiet mit einem Gitter in die bereits erwähnten Kontrollvolumen V_i. Es gibt zwei Möglichkeiten zur Definition der Kontrollvolumina V_i. Wir besprechen die beiden Varianten der Kontrollvolumina in der folgenden Definition 3.2.10.

Definition 3.2.10 (Kontrollvolumen.)
Ein **Kontrollvolumen** *V_i ist ein festgelegtes Volumen im betrachteten Gebiet $\mathcal{G}$ mit einem festen*

Bezugspunkt $\mathcal{P}_i$. Für die Kontrollvolumen eines Gebietes muss weiterhin gelten, dass sie eine direkte Zerlegung, im Sinne der Definition 3.2.9 des Gebietes sind.
Bei der Zerlegung eines Gebietes $\mathcal{G}$ in Kontrollvolumen gibt es zwei Vorgehensweisen. Die erste Variante ist die Folgende: Man unterteilt das Gebiet $\mathcal{G}$ in Kontrollvolumen V_i und legt den Bezugspunkt $\mathcal{P}_i$ in das Zentrum des Kontrollvolumen V_i. Alternativ kann man auch zuerst die Bezugspunkte $\mathcal{P}_i$ im Gebiet $\mathcal{G}$ festlegen und die Kontrollvolumen über die Bezugspunkte definieren. Dazu legt man die Ränder der Kontrollvolumen jeweils in die Mitte zwischen zwei Bezugspunkte.
Man unterscheidet die beiden Typen der Kontrollvolumen in zellen-zentrierte Elemente (engl. cell centered control volumes) und in kanten-zentrierte Elemente (engl. vertex centered control volumes).
Ein Kontrollvolumen V_i wird auch als **Finites-Volumen-Element** *bezeichnet.*

Beispiel 3.2.11 (Konstruktion Kontrollvolumen)
Wir betrachten hier anhand zweier Skizzen, siehe Abbildung 3.10 und 3.11, die Konstruktion beider Varianten von Kontrollvolumen.

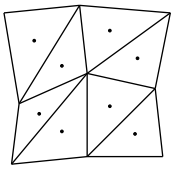

Abbildung 3.10: Ein zellen-zentriertes Kontrollvolumen.

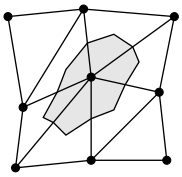

Abbildung 3.11: Ein kanten-zentriertes Kontrollvolumen.

Die kanten-zentrierten Kontrollvolumen haben eine bessere Approximationsgüte bei zentralen Differenzen. Bei der Repräsentationsgüte des Mittelwertes des Kontrollvolumen haben die zellen-zentrierten Kontrollvolumen die höhere Genauigkeit.
Als nächstes müssen wir uns mit der Approximation der Flächen- und Volumenintegrale der Kontrollvolumen befassen. Die Quadraturformeln zur Auswertung der Flächen- und Volumenintegrale müssen folgende Kriterien erfüllen: Eine möglichst hohe Genauigkeit, sie müssen schnell auswertbar und auf die bestehenden Variablen beschränkt sein.
Bei der drei-dimensionalen Finite-Volumen-Methode sind die Volumenintegrale drei-dimensional und die Flächenintegrale zwei-dimensional. Im Falle einer zwei-dimensionalen Finiten-Volumen-Methode reduziert sich die Dimension der Integrale jeweils um eine Dimension. Wir sprechen dann aber weiterhin bei den Integralen über das betrachtete Gebiet von Volumenintegralen und bei den Integralen über den Rand des Gebietes von Flächenintegralen, auch wenn

sie dann nur zwei- beziehungsweise ein-dimensional sind.

Flächenintegrale

In diesem Abschnitt befassen wir uns mit den üblichen Varianten zur numerischen Approximation von Flächenintegralen für die Finite-Volumen-Methode. Der Einfachheit halber beschränken wir unsere Betrachtungen auf die kartesischen Viereck-Gitter mit zellen-zentrierten Kontrollvolumen.

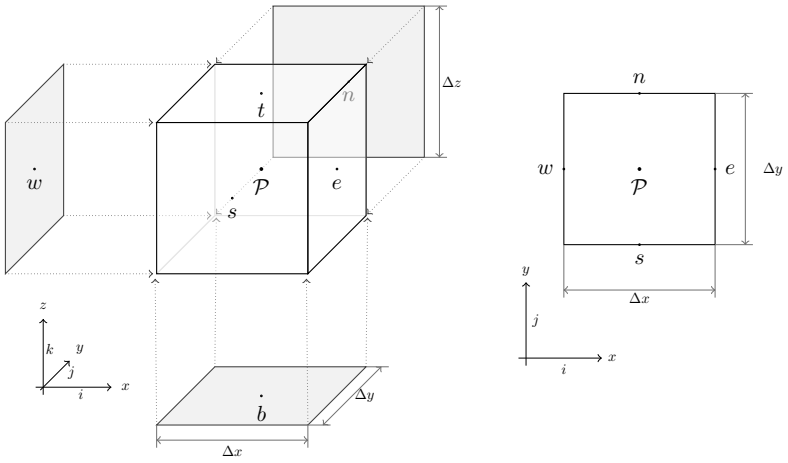

Abbildung 3.12: Kartesisches 2D und 3D Kontrollvolumen.

Die Abbildung 3.12 veranschaulicht links ein Kontrollvolumen im drei-dimensionalen Fall und rechts im zwei-dimensionalen Fall. Im Folgenden Werden wir die dort eingeführten Bezeichnungen für die einzelnen Punkte und Flächen verwenden.
Sei $S_i := \delta V_i$ die Oberfläche des Kontrollvolumens V_i. Weiter gilt für die Oberfläche S_i im kartesischen Gitter, dass wir sie als Vereinigung von F Planen-Flächen darstellen können, das heißt $S_i = \bigcup_{\ell \in F} S_{i_\ell}$. Dann gilt für das Flächenintegral über das Kontrollvolumen V_i aufgrund der Linearität folgende Gleichung:

$$\int_{S_i} f dS = \sum_{\ell \in F} \int_{S_{i\ell}} f dS \tag{3.61}$$

Im zwei-dimensionalen Fall sind es vier Flächen in Gleichung (3.61) und im drei-dimensionalen Fall sind es dann sechs Flächen, wie in der Abbildung 3.12 dargestellt. Wir betrachten nun mehrere Ansätze zur numerischen Quadratur über die Flächen S_ℓ. Wir setzen in den folgenden Betrachtungen immer voraus, dass wir die Funktionswerte an den notwendigen Punkten kennen. Im Falle, dass dies nicht gilt, interpolieren wir die Funktionswerte. Auf die Interpolation gehen wir später näher ein.

Mittelpunktsregel
Die einfachste numerische Quadratur zur Berechnung des Flächenintegrals I_{S_s} ist die Mittel-

punktsregel[16].

$$I_{S_s} = \int_{S_s} f dS = \bar{f} dS \approx f_s \cdot S_s \tag{3.62}$$

Die Mittelpunkt-Regel in Gleichung (3.61) unterscheidet sich nur bei der Berechnung der Fläche S_s vom zwei- zum drei-dimensionalen Fall.

Trapez-Regel
Ein weiteres Verfahren ist die Trapez-Regel. Hier müssen wir zwischen dem zwei- und drei-dimensionalen Fall unterscheiden.

$$I_{S_s} = \int_{S_s} f dS \approx \frac{S_s}{2}(f_{se} + f_{sw}) \tag{3.63}$$

Die Gleichung (3.63) zeigt die Trapez-Regel für den zwei-dimensionalen Fall und die Gleichung (3.64) die Variante für den drei-dimensionalen Fall.

$$I_{S_s} = \int_{S_s} f dS \approx \frac{S_s}{4}(f_{st} + f_{se} + f_{sw} + f_{sb}) \tag{3.64}$$

Simpson-Regel
Die Simpson-Regel[17] ist ein Verfahren höherer Ordnung zur Approximation des Flächenintegrals.

$$I_{S_s} = \int_{S_s} f dS \approx \frac{S_s}{6}(f_{se} + 4f_s + f_{sw}) \tag{3.65}$$

Die Gleichung (3.65) ist die Umsetzung der Simpson-Regel für den zwei-dimensionalen Fall mit einer Approximationsgüte 4. Ordnung. Für den drei-dimensionalen Fall lautet die Simpson Regel für eine rechteckige Oberfläche:

$$I_{S_s} = \int_{S_s} f dS \approx \frac{S_s}{36}(f_{swt} + 4f_{st} + f_{set} + 4f_{sw} + 16f_s + 4f_{se} + f_{swb} + 4f_{sb} + f_{seb}) \tag{3.66}$$

Detaillierte Betrachtungen der Mittelpunkt-Regel, der Trapez-Regel und der Simpson-Regel findet man zum Beispiel im Buch von Stoer *Numerische Mathematik I* [Sto02].

Volumenintegrale

Im Anschluss an die Flächenintegrale wenden wir uns der Approximation der Volumenintegrale zu. Der größte Unterschied zu den Flächenintegralen ist, dass wir das Integral nicht in eine Summe von einzelnen Integralen zerlegen müssen.

Mittelpunktsregel
Die Mittelpunktsregel für Volumenintegrale unterscheidet sich nur durch die Berechnung der Größe des Integrationsgebietes vom Flächenintegral mit der Mittelpunktsregel.

$$I_{V_\mathcal{P}} = \int_{V_\mathcal{P}} f dV = \bar{f} \underbrace{\int_{V_\mathcal{P}} dV}_{=:\Delta V_\mathcal{P}} \approx f_\mathcal{P} \Delta V \tag{3.67}$$

[16] Die Mittelpunktsregel wird auch Rechteckregel genannt.
[17] Die Simpson-Regel ist nach dem englischen Mathematiker Thomas Simpson (1710-1761) benannt. Sie wird auch Kepler'sche Fassregel nach Johannes Kepler (1571-1630) genannt.

Analog zum Flächenintegral ist der Unterschied vom zwei-dimensionalen zum drei-dimensionalen Fall der Gleichung (3.67) ebenfalls nur durch die Berechnung von $\Delta V_{\mathcal{P}}$ gegeben.

Simpson-Regel
Die Umsetzung der Simpson-Regel auf Volumenintegrale ist aufwendiger als die Simpson-Regel bei Flächenintegralen. Im zwei-dimensionalen Fall entsprechen die Oberflächenintegrale den Flächenintegralen des drei-dimensionalen Falls. Damit haben wir die Simpson-Regel im zwei-dimensionalen Fall schon mit der Gleichung (3.65) betrachtet. Für den 3D Fall gehen wir analog vor, somit gilt für die Simpson-Regel:

$$\begin{aligned} I_{V_{\mathcal{P}}} = \int_{V_{\mathcal{P}}} f dV \approx & \frac{\Delta V}{216} \left(f_{swb} + 4f_{sb} + f_{seb} + 4f_{wb} + 16f_b + 4f_{eb} + f_{nwb} + 4f_{nb} + f_{neb} + \right. \\ & 4f_{sw} + 16f_s + 4f_{se} + 16f_w + 64f_{\mathcal{P}} + 16f_e + 4f_{nw} + 16f_n + 4f_{ne} + \\ & \left. f_{swt} + 4f_{st} + f_{set} + 4f_{wt} + 16f_t + 4f_{et} + f_{nwt} + 4f_{nt} + f_{net} \right) \end{aligned} \tag{3.68}$$

Aus der Gleichung (3.68) folgt direkt, dass bei Verfahren mit höherer Ordnung, als die Simpson-Regel, der Aufwand unverhältnismässig ansteigt, da wir bei einer Finiten-Volumen-Simulation in der Regel sehr viele Kontrollvolumen betrachten.

Interpolation

In diesem Abschnitt befassen wir uns mit der Berechnung der interpolierten Funktionswerte. Da wir die Zustandsgrößen in jedem Kontrollvolumen nur am Referenzpunkt $\mathcal{P}$ kennen, müssen wir zur Berechnung der Flächen- und Volumenintegrale, sowie der benötigten Gradienten auf interpolierte Werte zurückgreifen.
Analog zu der Finiten-Differenzen-Methode, siehe Abschnitt 3.2.1, muss darauf geachtet werden, dass wir kein oszillierendes Verhalten der Zustandsgrößen bekommen. Eine Möglichkeit, dies sicherzustellen, ist die Verwendung der Aufwind-Interpolation, auf die wir zuerst eingehen werden.

Aufwind-Interpolation
Die Aufwind-Interpolation (engl. Upwind-Interpolation) ist analog zu den Vorwärts- beziehungsweise Rückwärtsdifferenzen in der Finiten-Differenzen-Methode erklärt. Damit erfüllt die Aufwind-Interpolation folgende Eigenschaft:

$$f_{s_{i,j}} := \begin{cases} f_{\mathcal{P}_{i,j}} & falls \quad (\mathbf{v} \cdot \nu)_s > 0; \\ f_{\mathcal{P}_{i,j-1}} & falls \quad (\mathbf{v} \cdot \nu)_s < 0; \end{cases} \tag{3.69}$$

Bemerkung 3.2.12 (Numerische Diffusion)
Man sagt eine Approximation sei numerisch diffusiv, wenn durch sie ein Term ein diffusives Verhalten bekommt. Ein Verhalten wird als diffusiv bezeichnet, wenn es sich ähnlich zur Diffusionsgleichung verhält. Dazu zählen die Aufwind-Methoden, die sich zur Berechnung auf Werte des Flusses stützen, die flussaufwärts liegen.
Eine detaillierte Betrachtung des diffusiven Verhaltens von einseitigen Differenzen und dem impliziten Euler-Verfahren findet man zum Beispiel in "Numerische Lösung partieller Differentialgleichungen" von Peter Bastian [Bas08] im Abschnitt 12.4.
Ein Vorteil von numerisch-diffusiven Approximationen ist ein stabilisierender Effekt, der dem Auftreten von Oszillationen entgegenwirkt.

Die Aufwind-Interpolation (3.69) ist in diesem Sinne eine numerisch-diffusive Interpolations-Methode. Sie ist ein Verfahren 1. Ordnung, und sichert die Oszillationsfreiheit. Die Ordnung des Verfahrens folgt direkt aus der Taylorentwicklung.

Lineare-Interpolation
Vergleichbar zu den zentralen Differenzen ist die lineare Interpolation. Bei einem kartesischem Gitter gilt für die Interpolations-Formel:

$$f_{s_{i,j,k}} = f_{s_{i,j-1,k}} \lambda_{s_{i,j,k}} + f_{s_{i,j,k}} (1 - \lambda_{s_{i,j,k}}) \tag{3.70}$$

$$\lambda_{s_{i,j,k}} = \frac{x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j,k}}}{x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j,k}}} \tag{3.71}$$

Zu beachten ist, dass die Gleichung (3.70) ein Verfahren 2. Ordnung ist. Dies folgt erneut direkt durch die Taylorentwicklung. Die lineare Interpolation kann allerdings, wie jedes Verfahren, für dessen Ordnung gilt, dass sie größer eins ist, zu einem Oszillationsverhalten führen.

Quadratische-Aufwind-Interpolation
Für ein Verfahren dessen Ordnung größer als zwei ist, benötigen wir weitere Punkte zur Interpolation. In diesem Fall möchten wir uns ein Verfahren der Ordnung drei herleiten und benötigen dazu einen dritten Punkt. Analog zur Aufwind-Interpolation (3.69) wählen wir den dritten Punkt flussaufwärts. Sei nun $\mathbf{u}_{i,j,k} := (u_{1_{i,j,k}}, u_{2_{i,j,k}}, u_{3_{i,j,k}})^T$ der Fluss und der betrachtete Punkt ist $\mathcal{P}_{i,j,k}$ un $d \in \{e, w, n, s, b, t\}$ dann gilt:

$$(i_{up_d}, j_{up_d}, k_{up_d}) := \begin{cases} I_{up}(1,i,j,k) & falls \quad d = e; \\ I_{up}(-1,i,j,k) & falls \quad d = w; \\ I_{up}(2,i,j,k) & falls \quad d = n; \\ I_{up}(-2,i,j,k) & falls \quad d = s; \\ I_{up}(3,i,j,k) & falls \quad d = t; \\ I_{up}(-3,i,j,k) & falls \quad d = b; \end{cases}, \tag{3.72}$$

$$I(\ell,i,j,k) := \begin{cases} (i + \delta_{1,|\ell|}, j + \delta_{2,|\ell|}, k + \delta_{3,|\ell|}) & falls \quad sign(\ell) \cdot u_{|\ell|_{i,j,k}} < 0 \\ (i - \delta_{1,|\ell|}, j - \delta_{2,|\ell|}, k - \delta_{3,|\ell|}) & falls \quad sign(\ell) \cdot u_{|\ell|_{i,j,k}} > 0 \end{cases} \tag{3.73}$$

Analog zum Aufwind-Index $(i_{up_d}, j_{up_d}, k_{up_d})$ ist der Abwind-Index $(i_{down_d}, j_{down_d}, k_{down_d})$ erklärt. Für die Bestimmung des Auf-Aufwind-Index, wird ein weiterer Index-Schritt in Richtung des Aufwindes gegangen. Mit den Ab- und Aufwind-Index können wir die Abwind- beziehungsweise Aufwind-Größen wie folgt definieren:

$$f_{down} := f_{\mathcal{P}_{i_{down_d}, j_{down_d}, k_{down_d}}},$$

$$f_{up} := f_{\mathcal{P}_{i_{up_d}, j_{up_d}, k_{up_d}}} \; und$$

$$f_{up-up} := f_{\mathcal{P}_{i_{up-up_d}, j_{up-up_d}, k_{up-up_d}}}.$$

Damit ergibt sich nun folgende Darstellung für die quadratische Aufwind-Interpolation:

$$f_{s_{i,j,k}} = f_{up} + g_1(f_{down} - f_{up}) + g_2(f_{up} - f_{up-up}), \tag{3.74}$$

wobei die Größen g_1 und g_2 über

$$g_1 := \frac{(x_{s_{i,j}} - x_u)(x_{s_{i,j}} - x_{uu})}{(x_d - x_u)(x_d - x_{uu})}, \tag{3.75a}$$

$$g_2 := \frac{(x_{s_{i,j}} - x_u)(x_d - x_{s_{i,j}})}{(x_u - x_{uu})(x_d - x_{uu})} \tag{3.75b}$$

definiert sind. Dieses Verfahren stellt eine naheliegende Verbesserung der linearen Interpolation dar. Bei äquidistanten Gittern vereinfacht sich die Gleichung (3.82) zu

$$f_{s_{i,j}} = \frac{3}{8} f_{down} + \frac{6}{8} f_{up} - \frac{1}{8} f_{up-up}. \tag{3.76}$$

Das Verfahren ist auch unter der Bezeichnung QUICK[18] bekannt.

Verfahren höherer Ordnung

Die Interpolationsverfahren müssen auf die verwendeten Integrations-Methoden, beziehungsweise auf die Gradienten-Berechnung abgestimmt sein, da sich eine aufwendigere Interpolation mit einer Ordnung, die größer gleich vier ist, nur lohnt wenn man ein Integrations Verfahren mit mindestens der Ordnung vier verwendet, weil immer die niedrigste Approximationsordnung die Gesamt-Ordnung festlegt.

In diesem Abschnitt betrachten wir eine Interpolation 4. Ordnung. Es handelt sich hier somit um eine Kubische-Interpolation. Höhere Interpolationsordnungen und auch Spline-Interpolationen sind auch möglich. Allerdings erhöhen sie den Aufwand nochmal deutlich, so dass gut abgewägt werden sollte, ob sich dieser erhöhte Aufwand lohnt.

Wir möchten erneut den Punkt f_s interpolieren. Es bietet sich eine symmetrische Aufteilung der vier Interpolations Punkte an. Damit ergibt sich über die Lagrange-Interpolation:

$$f_{s_{i,j,k}} = g_1 \cdot f_{\mathcal{P}_{i,j,k}} + g_2 \cdot f_{\mathcal{P}_{i,j-1,k}} + g_3 \cdot f_{\mathcal{P}_{i,j+1,k}} + g_4 \cdot f_{\mathcal{P}_{i,j-2,k}}, \tag{3.77}$$

$$\begin{aligned}
g_1 =& \frac{(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_2 =& \frac{(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_3 =& \frac{(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_4 =& \frac{(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j,k}})(x_{s_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})}{(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j+1,k}})}.
\end{aligned} \tag{3.78}$$

Für ein äquidistantes Gitter vereinfacht sich der Ausdruck (3.77) zu:

$$f_{s_{i,j,k}} = . \frac{9 f_{\mathcal{P}_{i,j,k}} + 9 f_{\mathcal{P}_{i,j-1,k}} - f_{\mathcal{P}_{i,j+1,k}} - f_{\mathcal{P}_{i,j-2,k}}}{16} \tag{3.79}$$

Anstatt der Lagrange-Interpolation bietet sich für die Anwendung eher das Neville-Aitken-Schema[19] an.

[18] QUICK steht für Quadratic Upwind Interpolation for Convective Kinematics und wurde 1979 von B.P. Leonard [Leo79] verbreitet.

[19] Das Neville-Aitken-Schema ist nach Eric Harold Neville (1889-1961) und Alexander Craig Aitken (1895-1967) benannt.

Wir haben uns bisher auf die Interpolation von Punkten beschränkt, die in den Mittelpunkten der Oberflächen liegen. Die Interpolation der Eckpunkte eines Kontrollvolumens erfolgt über die mehrfache Anwendung der Interpolations-Methoden. Im zwei-dimensionalen Fall liegen die Oberflächen-Mittelpunkte auf den Kanten unserer Kontrollvolumen. Beim erneuten Anwenden der Interpolation erhalten wir damit die Werte an den Ecken der Zellen. In drei Dimensionen benötigen wir einen Interpolations-Schritt mehr, das heißt wir gehen über die Oberflächen-Mittelpunkte, die in den Flächen liegen, zu den Kanten-Mittelpunkten und dann in die Eckpunkte über. Für die lineare Interpolation gilt damit für eine Größe f im Eckpunkt x_{net}:

$$\begin{aligned} f_{net} =& \lambda_{t_{i,j,k}}(\lambda_{n_{i,j,k+1}}(\lambda_{e_{i,j+1,k+1}} f_{\mathcal{P}_{i+1,j+1,k+1}} + (1-\lambda_{e_{i,j+1,k+1}}) f_{\mathcal{P}_{i,j+1,k+1}}) + \\ & (1-\lambda_{n_{i,j,k+1}})(\lambda_{e_{i,j,k+1}} f_{\mathcal{P}_{i+1,j,k+1}} + (1-\lambda_{e_{i,j,k+1}}) f_{\mathcal{P}_{i,j,k+1}})) + \\ & (1-\lambda_{t_{i,j,k}})(\lambda_{n_{i,j,k}}(\lambda_{e_{i,j+1,k}} f_{\mathcal{P}_{i+1,j+1,k}} + (1-\lambda_{e_{i,j+1,k}}) f_{\mathcal{P}_{i,j+1,k}}) + \\ & (1-\lambda_{n_{i,j,k}})(\lambda_{e_{i,j,k}} f_{\mathcal{P}_{i+1,j,k}} + (1-\lambda_{e_{i,j,k}}) f_{\mathcal{P}_{i,j,k}})). \end{aligned} \tag{3.80}$$

Die betrachteten Interpolations-Verfahren geben einen Überblick über die gebräuchlichsten Verfahren, die bei den Finiten-Volumen-Verfahren verwendet werden. Weitere gebräuchliche Verfahren sind zum Beispiel:

- LUD: Lineare Aufwind-Approximation (engl. Linear Upwind Differencing)
- SUS: Schräg-Aufwind-Verfahren (engl. skew Upwind schemes)
- MINMOD: Minimum-Modell (engl. Minimum model)

Ausführlichere Betrachtungen zu den Interpolations Methoden finden sich im Buch von Ferziger und Perić [FP08] oder im *User Manual Comet* [Ins01], an denen sich der obige Abschnitt orientiert. Die vorgestellten Methoden funktionieren auch bei einem nicht-kartesischen Gitter. Allerdings ist die Anwendung hier komplizierter.

Gradienten-Berechnung

Für die Berechnung der, in den Erhaltungsgleichungen auftretenden, Gradienten haben wir erneut mehrere Möglichkeiten. Wir gehen nun auf folgende Verfahren etwas näher ein:

- Lineare-Gradienten-Approximation
- Quadratische-Aufwind-Gradienten-Approximation
- Kubische-Gradienten-Approximation
- Satz von Gauß
- Kleinste-Fehlerquadrat-Methode

In der Abbindung 3.13 haben wir symbolisch ein beliebiges Kontrollvolumen eingeführt, wie es in einem nicht-kartesischen Gitter vorkommen könnte. Anhand der Abbildung sind die im folgenden verwendeten Begriffe und Bezeichnungen eingeführt.

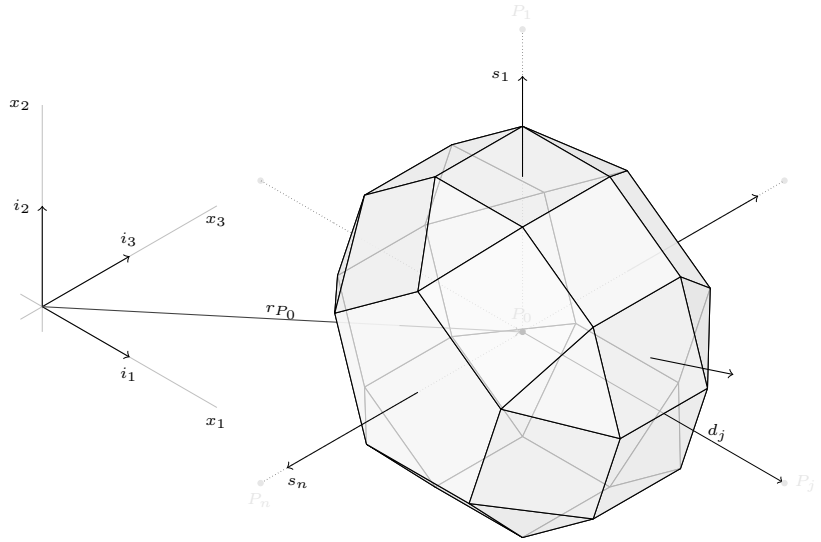

Abbildung 3.13: Ein beliebiges symbolisches Kontrollvolumen.

Lineare-Gradienten-Approximation
Legen wir das lineare Interpolation-Schema (3.70) zugrunde, ergibt sich für die lineare Gradienten-Approximation in Richtung *Norden*:

$$\left(\frac{\partial f}{\partial y}\right)_n = \frac{f_{\mathcal{P}_{i,j+1,k}} - f_{\mathcal{P}_{i,j,k}}}{x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+,k}}}. \tag{3.81}$$

Die Ableitungen in die anderen Richtungen ergeben sich in analoger Weise.

Quadratische-Aufwind-Gradienten-Approximation
Für die Quadratische-Aufwind-Gradienten-Approximation gilt:

$$\left(\partial \frac{f_{s_{i,j,k}}}{\partial y}\right)_s = \left(\frac{\partial g_1}{\partial y}\right)_s (f_{down} - f_{up}) + \left(\frac{\partial g_2}{\partial y}\right)_s (f_{up} - f_{up-up}), \tag{3.82}$$

wobei die Größen g_1 und g_2 über

$$\left(\frac{\partial g_1}{\partial y}\right)_s = \frac{2x_{s_{i,j}} - x_u - x_{uu}}{(x_d - x_u)(x_d - x_{uu})}, \tag{3.83a}$$

$$\left(\frac{\partial g_1}{\partial y}\right)_s = \frac{-2x_{s_{i,j}} + x_u - x_d}{(x_u - x_{uu})(x_d - x_{uu})} \tag{3.83b}$$

definiert sind.

Gradienten-Approximation 4. Ordnung
Für die Approximation des Gradienten mit der Ordnung 4 differenzieren wir die Darstellung der kubische Interpolation (3.77):

$$\left(\frac{\partial f}{\partial y}\right)_s = g_1' \cdot f_{\mathcal{P}_{i,j,k}} + g_2' \cdot f_{\mathcal{P}_{i,j-1,k}} + g_3' \cdot f_{\mathcal{P}_{i,j+1,k}} + g_4' \cdot f_{\mathcal{P}_{i,j-2,k}} \tag{3.84}$$

wobei die g_i' und $i = 1, \cdots 4$ wie folgt erklärt sind

$$\begin{aligned}
g_1' =& \frac{3x_{s_{i,j,k}}^2 - 2x_{s_{i,j,k}}(x_{\mathcal{P}_{i,j-1,k}} + x_{\mathcal{P}_{i,j+1,k}} + x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
&+ \frac{x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j+1,k}} + x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j+2,k}} + x_{\mathcal{P}_{i,j+1,k}}x_{\mathcal{P}_{i,j+2,k}}}{(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_2' =& \frac{3x_{s_{i,j,k}}^2 - 2x_{s_{i,j,k}}(x_{\mathcal{P}_{i,j,k}} + x_{\mathcal{P}_{i,j+1,k}} + x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
&+ \frac{x_{\mathcal{P}_{i,j,k}}x_{\mathcal{P}_{i,j+1,k}} + x_{\mathcal{P}_{i,j,k}}x_{\mathcal{P}_{i,j+2,k}} + x_{\mathcal{P}_{i,j+1,k}}x_{\mathcal{P}_{i,j+2,k}}}{(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+1,k}})(x_{\mathcal{P}_{i,j-1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_3' =& \frac{3x_{s_{i,j,k}}^2 - 2x_{\int_{i,j,k}}(x_{\mathcal{P}_{i,j-1,k}} + x_{\mathcal{P}_{i,j,k}} + x_{\mathcal{P}_{i,j+2,k}})}{(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
&+ \frac{x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j,k}} + x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j+2,k}} + x_{\mathcal{P}_{i,j,k}}x_{\mathcal{P}_{i,j+2,k}}}{(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+1,k}} - x_{\mathcal{P}_{i,j+2,k}})} \\
g_4' =& \frac{3x_{s_{i,j,k}}^2 - 2x_{\int_{i,j,k}}(x_{\mathcal{P}_{i,j-1,k}} + x_{\mathcal{P}_{i,j,k}} + x_{\mathcal{P}_{i,j+1,k}})}{(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j+1,k}})} \\
&+ \frac{x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j,k}} + x_{\mathcal{P}_{i,j-1,k}}x_{\mathcal{P}_{i,j+1,k}} + x_{\mathcal{P}_{i,j,k}}x_{\mathcal{P}_{i,j+1,k}}}{(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j-1,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j,k}})(x_{\mathcal{P}_{i,j+2,k}} - x_{\mathcal{P}_{i,j+1,k}})}.
\end{aligned} \tag{3.85}$$

Für die Auswertung der Gleichungen (3.84) und (3.85) kann man ein paar Vereinfachungen ausnutzen. Bei einen äquidistanten kartesischen Gitter vereinfacht sich die Gleichungen (3.84) und (3.85) beispielsweise zu:

$$\left(\frac{\partial f}{\partial y}\right)_s = \frac{27f_{\mathcal{P}_{i,j}} - 27f_{\mathcal{P}_{i,j}} + f_{\mathcal{P}_{i,j}} - f_{\mathcal{P}_{i,j}}}{24\Delta y}. \tag{3.86}$$

Satz von Gauß
Ein anderer Zugang zur Approximation des Gradienten, wie es beispielsweise auch im *Comet User Manual* [Ins01] beschrieben ist, basiert auf der Idee des Satzes von Gauß. Hier überführt man das Volumen-Integral eines differenzierten Terms in ein Oberflächen-Integral. Dazu betrachten wir den folgenden Ausdruck:

$$\int_{I_\alpha} D_\alpha f(y)dy = f(x + h\alpha) - f(x), \tag{3.87}$$

wobei $D_\alpha f$ die Richtungsableitung von f in Richtung α ist und für das Intervall I_α gilt $I_\alpha := [x, x + h\alpha]$. Desweiteren gilt für $V := I_\alpha \times S_\alpha$, damit folgt aus der Gleichung (3.87).

$$\int_V D_\alpha f dV = \int_{S_\alpha} f d\mathbf{s}. \tag{3.88}$$

Für den Gradienten von f folgt damit aus Gleichung (3.88):

$$\int_V \mathbf{grad} f dV = \sum_{j=1}^{n_f} \int_{S_j} f\nu_j dS \tag{3.89}$$

wobei ν_j die äussere Normale der Oberfläche S_j mit Länge Eins und n_f die Anzahl der Oberflächen des Kontrollvolumen V ist. Die Integrale approximieren wir nun mit der Mittelpunktsregel. Damit folgt für das Konrollvolumen im Punkt $\mathcal{P}_i$ aus der Gleichung (3.89):

$$(\,\mathbf{grad} f)_{\mathcal{P}_i} \approx \frac{1}{V_{\mathcal{P}_i}} \sum_{\ell=1}^{n_f} f_\ell \mathbf{s}_{\mathcal{P}_i,\ell}. \tag{3.90}$$

Der Vektor $\mathbf{s}_{\mathcal{P}_i,\ell}$ zeigt hierbei in Richtung der äußeren Normalen auf der Fläche ℓ, siehe Abbildung 3.13, die Länge des Vektors entspricht dem Flächeninhalt der Fläche ℓ und f_ℓ ist der Funktionswert von f im Mittelpunkt der Fläche ℓ. Wir können $\mathbf{s}_\ell$ mittels folgender Gleichung berechnen:

$$\mathbf{s}_\ell = \frac{1}{2} \sum_{i=3}^{n_\ell^{vertex}} \left[(\mathbf{r}_{i-1} - \mathbf{r}_1) \times (\mathbf{r}_i - \mathbf{r}_1)\right]. \tag{3.91}$$

Die Größen r_i sind die Ecken der Fläche ℓ und n_ℓ^{vertex} ist die Anzahl der Ecken.
Betrachten wir als Kontrollvolumen V einen Würfel mit Kanten Länge $2h$ entspricht die Gleichung (3.90) der Approximation des Gradienten mit dem zentralen Finiten-Differenzenquotienten siehe (3.35), mit auf den Flächen-Mittelpunkt basierenden Funktionswerten.

Kleinste-Fehlerquadrat-Methode
Als letzten Ansatz zur Approximation des Gradienten betrachten wir die Methode der kleinsten Fehlerquadrate (engl. least square method). Diese Methode wird auch im *Comet User Manual* [Ins01] betrachtet. Diese Methode gleicht eine lineare Funktion zwischen den nächsten Nachbar-Punkten an. Sei n_i^{cell} die Anzahl der benachbarten Zellen dann gilt:

$$\mathbf{d}_{\mathcal{P}_{i,\ell}} \cdot (\,\mathbf{grad} f)_{P_{\mathcal{P}_i}} = f_{\mathcal{P}_{i,\ell}} - f_{\mathcal{P}_i}. \tag{3.92}$$

Der Distanz-Vektor $\mathbf{d}_{\mathcal{P}_{i,\ell}} \in \mathbb{R}^3$ berechnet sich über den Abstand der Zell Mittelpunkte:

$$\mathbf{d}_{\mathcal{P}_{i,\ell}} = \mathbf{r}_{\mathcal{P}_{i,\ell}} - \mathbf{r}_{\mathcal{P}_i}. \tag{3.93}$$

Aus der Gleichung (3.92) ergibt sich das Gleichungssystem:

$$\mathsf{D}(\,\mathbf{grad} f)_{\mathcal{P}_i} = \mathbf{f}. \tag{3.94}$$

Für die Matrix D und den Vektor $\mathbf{f}$ der rechten Seite gilt:

$$\mathsf{D} := \sum_{j=1}^{n_i^{cell}} \mathbf{d}_{\mathcal{P}_{i,j}} \mathbf{d}_{\mathcal{P}_{i,j}}^T, \tag{3.95}$$

$$\mathbf{f} := \sum_{j=1}^{n_f} \mathbf{d}_{\mathcal{P}_{i,j}} (f_{\mathcal{P}_{i,j}} - f_{\mathcal{P}_i}). \tag{3.96}$$

Somit gilt für den Gradienten der Funktion f im Punkt $\mathcal{P}_i$

$$(\,\mathbf{grad} f)_{\mathcal{P}_i} = \mathsf{D}^{-1}\mathbf{f}. \tag{3.97}$$

Wir können die Matrix D explizit invertieren, da sie nur von der Größe 3×3 ist. Für ein gegebenes festes Gitter ist die Matrix D während der gesamten Simulation fix. Bei auftretenden

Gitter-Deformationen, beziehungsweise bei bewegten Gittern, muss die Matrix und die zugehörige Inverse unter Umständen für die betroffenen Zellen neu ermittelt werden.

Bei einem Gitter mit äquidistanten Würfeln als Kontrollvolumen ergibt sich wie beim vorherigen Ansatz wieder die Vereinfachung, dass die Approximation des Gradienten dem zentralen Differenzenquotient entspricht, allerdings werden die Funktionswerte der Zell-Mittelpunkte anstatt der Flächen-Mittelpunkte für die Approximation verwendet.

Randbedingungen

Die Behandlung der Randbedingungen beim Finiten-Volumen-Verfahren erfolgt über die Anpassung des Upwind-Schemas. Da keine Werte außerhalb des Rechengitters vorhanden sind, müssen wir einseitige Differenzen und Extrapolationen verwenden, ohne dass wir uns mit Sicherheit auf Aufwind-Verfahren beschränken können. Wie die einzelnen Randbedingungen behandelt werden, betrachten wir nun im Folgenden detailliert.

Einströmbedingung
Die Einströmbedingung wird mittels des Festsetzens einer festen Flussrate realisiert. Im Detail bedeutet dies, dass wir die Geschwindigkeit am Einfluss-Rand fest vorgeben und die verbleibenden Größen - zum Beispiel der Druck - werden aus den Werten im inneren approximiert.
Analog kann man auch eine Ausströmbedingung festlegen, indem man eine Strömung, die aus dem Gebiet herausfließt, vorgibt. Allerdings ist die nachfolgende Variante als Ausströmbedingung verbreiteter.

Ausströmbedingung
Bei der Ausströmbedingung wird in der Regel folgendes Schema vorausgesetzt: Die Ableitungen aller Größen werden in Richtung der Strömung auf Null gesetzt. Diese Wahl können wir so interpretieren, dass die betrachten Größen erst einmal keiner Änderung in Flussrichtung unterliegen, das heißt, dass das Verhalten außerhalb des betrachteten Strömungsgebiets keinen Einfluss auf das Verhalten im Strömungsgebiet hat.
Damit ergibt sich, dass wir den Ausström-Rand über die oben betrachteten Aufwind-Interpolation behandeln können, wie wir es in der Gleichung (3.69) betrachtet haben. Die Berechnung der Gradienten vereinfacht sich durch die Annahme, dass die Ableitungen in Flussrichtung Null sind. Dadurch müssen keine Gradienten in Richtung des Randes berechnet werden.

Haftrandbedingung
Haftrandbedingungen werden oft mit ihrer englischen Bezeichnung No-Slip Conditions bezeichnet. Wie der Name es schon ausdrückt, nehmen wir an, dass das Fluid am Rand keine Fließgeschwindigkeit parallel zum Rand besitzt. Um dieses Verhalten physikalisch korrekt zu simulieren benötigt man in Randrichtung eine feine Diskretisierung des Gitters. Nähere Betrachtungen zum Verhalten der Strömungen an Grenzschichten findet man in folgenden Literaturangaben: *Large-Eddy simulation of compressible flows using the finite-volume Method* von Roland von Kaenel [Kae03], *Numerische Simulation des Grenzschichtverhalten in Turbinengittern unter periodisch-instationären Strömungsbedingungen* von Jens Thurso [Thu02] oder

Grenzschicht-Theorie von Schlichting und Gersten [SG05].
Ein Ansatz, die Größen am Rand zu betrachten ist, die Strömungsrichtung parallel und senkrecht zum Rand auf Null zu setzen. Die Ableitungswerte werden mit einseitigen Differenzenquotienten und die anderen Größen mittels Extrapolation approximiert. Für die Betrachtung von Extrapolationsverfahren sei exemplarisch auf das Buch *Numerische Mathematik 2* von [Sto02] verwiesen.

Gleitrandbedingung
Die Gleitrandbedingung entspricht der Randbedingung der Euler-Gleichungen. Hier wird angenommen, dass die Geschwindigkeit parallel zum Rand nicht vom Rand beeinflusst wird. Nur der Anteil, der senkrecht zum Rand fließt, wird auf Null gesetzt. Für die Ableitung der Geschwindigkeiten gilt, dass die Randbedingung parallel zur Wand keinen Einfluss auf den Gradienten hat. Für die anderen Ableitungen müssen analog zu den anderen Randbedingungen einseitige Differenzenquotienten verwendet werden.
Gleitrandbedingungen werden gerne zur Begrenzung des simulierten Strömungsgebiets verwendet, wenn die dort auftretende Strömung näherungsweise parallel zum Rand verläuft. Als Beispiel können wir hier auf die Umströmung eines Schiffes verweisen, wenn wir eine gerade Fahrt annehmen und die Ränder parallel zum Schiffsrumpf in ausreichendem Abstand wählen.

Weitere Randbedingung
Es gibt natürlich noch eine Vielzahl an weiteren möglichen Randbedingungen. Die oben betrachteten Randbedingungen stellen die klassischen und gebräuchlichsten Varianten dar. Man kann noch Werte für die Temperatur am Rand fordern, beziehungsweise Änderungsraten. Man kann die Varianten teilweise kombinieren. Zudem gibt es dann noch *nicht physikalische* Randbedingungen. Damit sind zum Beispiel Randbedingungen gemeint, die zur Koppelung von unterschiedlichen Simulationsmethoden benötigt werden oder zur Umsetzung von bewegten Gittern. Hierzu sei exemplarisch das gleitende Interface (engl. sliding interface) erwähnt.

Das Gleichungssystem

In den letzten Abschnitten haben wir die einzelnen Schritte, die zum numerischen Lösen der Navier-Stokes-Gleichungen mit der Finiten-Volumen-Methode nötig sind, näher betrachtet. Im Folgenden müssen wir diese einzelnen Bausteine so zusammensetzen, dass wir das System simulieren können.
Durch die Zerlegung des gesamten betrachteten Strömungsgebiets, in kleine Konrollvolumen und deren numerischen Betrachtungen bezüglich der Approximation der Integrale, beziehungsweise der Ableitungswerte, kennen wir die Interaktion der einzelnen Zellen zueinander.
Analog zum Vorgehen beim Finiten-Differenzen-Verfahren ergibt sich damit, dass wir zuerst ein Gleichungssystem für den Druck und anschließend mit den neuen Druckwerten das System für die Geschwindigkeit lösen. Damit ergibt sich für den Druckvektor $\mathbf{p}$:

$$\mathsf{A}_p\mathbf{p} = \mathbf{f}_p \tag{3.98}$$

und für die Geschwindigkeit $\mathbf{v}$

$$\mathsf{A}_v\mathbf{v} = \mathbf{f}_v. \tag{3.99}$$

Als letztes müssen nur noch die entstandenen Gleichungssysteme (3.98) und (3.99) mit einem geeigneten Verfahren lösen. Hierzu sei auf die bereits erwähnte Literatur [Mei05] und [Kan00]

verwiesen.

In den nächsten beiden Abschnitten befassen wir uns mit der Umsetzung der Finiten-Volumen-Methode in zwei Implementierungen zum numerischen Lösen der Navier-Stokes-Gleichungen.

Caffa

Der Name Caffa steht für Computer-Aided Fluid Flow Analysis. Das Programm ist ein Quell offenes Programm zur zwei-dimensionalen Simulation der Navier-Stokes-Gleichungen, mit integrierter Gittererzeugung. Die Software wurde massgeblich von Perić entwickelt und von Hadzic an der TU Hamburg-Harburg weiterentwickelt. Die verwendete Programmiersprache ist Fortran.
Wir beschreiben im Folgenden den Aufbau und den Algorithmus der Software Caffa im Algorithmus 3.2.13.

Algorithm 3.2.13 Caffa Hauptalgorithmus

```
1:  loadData()
2:  initProgram()
3:  if restart then
4:     readOldSimulationData()
5:  end if
6:  for i_grid = i_grid^start → i_grid^end do
7:     if i_grid > i_grid^start then
8:        extrapolateDataCoarse2FineGrid()
9:     end if
10:    for iter = 1 → maxIter do
11:       time+ = Δ_time
12:       varold()
13:       setBoundaryCondition()
14:       if moveGrid then
15:          moveGridGeometry()
16:          buildNewGrid()
17:       end if
18:       //Iteration per time step
19:       for k = 1 → K do
20:          solve: A_velocity v = rhs        //calculate Velocity
21:          solve: A_pressure p = rhs        //calculate Mass
22:          solve: A_temperature T = rhs     //calculate Temperature
23:       end for
24:       postProcess()
25:    end for
26: end for
```

Nachdem wir nun den Hauptalgorithmus von Caffa kennen, gehen wir im Folgenden etwas näher auf die einzelnen Punkte des Algorithmus ein.

Gittererzeugung

Die Gitter-Erzeugung bei Caffa basiert auf einem äußerst einfach beschriebenen Format zur Beschreibung des Gitters. Die Grundform der Geometrie ist ein Viereck. Die Zellen-Geometrie basiert dazu passend ebenfalls auf viereckigen Kontrollvolumen. Die Feinheit des Gitters beschreiben wir, indem wir für die Nord-Süd-Seite der Geometrie sowie für die Ost-West-Seite jeweils eine Unterteilung vorgeben. Dadurch ist das Gitter eindeutig beschrieben.

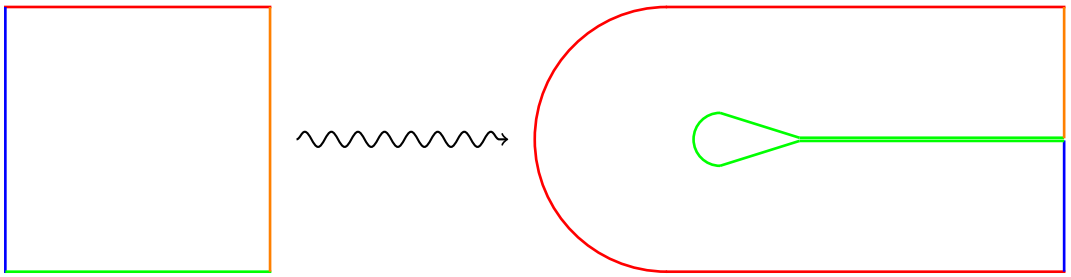

Abbildung 3.14: Beschreibung der Geometrie in Caffa.

Komplexere Geometrien werden durch Transformation der Ausgangs-Geometrie und passende Randbedingungen beschrieben. Dieses Vorgehen ist in der Abbildung 3.14 veranschaulicht.
Auf den jeweiligen Grundseiten, Nord, Süd, Ost und West, kann man die Randbedingungen abschnittsweise festlegen. So können wir beispielsweise den beiden horizontalen Abschnitten des Nord-Randes in Abbildung 3.14 Gleitrandbedingungen zuweisen und dem Bogen-Segment des Nord-Randes Einströmbedingungen.
Für bewegte Gitter wird das Gitter direkt im Caffa-Code transformiert. Dies geschieht über eine Verschiebung der entsprechenden Gebietsränder. Anschließend werden die Kontrollvolumina bezüglich der Änderung angepaßt und die entsprechend notwendigen Größen neu berechnet.

Approximationsmethoden

Die Berechnung der Gradienten wird nach der auf dem Satz von Gauß basierenden Variante berechnet. Dieses Approximations Schema haben wir im Abschnitt 3.2.3 unter dem Punkt *Satz von Gauß* beschrieben. Die Gleichung (3.90) spezialisiert sich bei Caffa insoweit, dass immer vier Flächen auftreten. Wir müssen nur die Zellen am Rand gesondert betrachten. In der Gleichung (3.100) betrachten wir die spezialisierte Form.

$$\nabla p = \begin{pmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \end{pmatrix} :\approx \frac{1}{V} \left(p_{\text{Nord}} \mathbf{s}_{\text{Nord}} + p_{\text{Ost}} \mathbf{s}_{\text{Ost}} + p_{\text{Süd}} \mathbf{s}_{\text{Süd}} + p_{\text{West}} \mathbf{s}_{\text{West}} \right), \tag{3.100}$$

$\mathbf{s}.$ ist der Oberflächenvektor, wie in Abbildung 3.13 dargestellt. Für die Werte $p.$ gilt, dass sie mit einer linearen Interpolation des linearen Taylorpolynoms für den aktuellen Zellmittelpunkt und die benachbarten Zellmittelpunkte berechnet werden. Die im Taylorpolynom verwendete Ableitung ist die Ableitung des letzten Zeitschrittes. Damit ergibt sich exemplarisch für den

Funktionswert am nördlichen Rand einer inneren Zelle:

$$\begin{aligned} f_{\text{Nord}} =& \lambda \left(f_{\mathcal{P}} + \frac{\partial f_{\mathcal{P}}}{\partial x}(\Delta x) + \frac{\partial f_{\mathcal{P}}}{\partial y}(\Delta y) \right) + \\ & (1-\lambda) \left(f_{\mathcal{P}_{\text{Nord}}} + \frac{\partial f_{\mathcal{P}_{\text{Nord}}}}{\partial x}(\Delta x) + \frac{\partial f_{\mathcal{P}_{\text{Nord}}}}{\partial y}(\Delta y) \right). \end{aligned} \tag{3.101}$$

Der nördliche Term einer Haftrand-Zelle vereinfacht sich zu

$$f_{\text{Nord}} = f_{\mathcal{P}}. \tag{3.102}$$

Die auftretenden Volumenintegrale und Flächenintegrale werden in Caffa mit der Mittelpunktsregel approximiert. Da die Funktionswerte im Zellmittelpunkt vorliegen, muss bei der Berechnung der Volumenintegrale keine Interpolation der Funktionswerte durchgeführt werden. Bei den Oberflächen-Integralen werden die Funktionswerte linear interpoliert.

Lösungsverfahren

Die auftretenden linearen Gleichungssysteme werden mit einer unvollständigen LU-Zerlegung[20] gelöst. Es wird die von Stone in *Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations* [Sto68] beschriebene SIP-Methode [21] verwendet.
Die SIP-Methode paßt die LU-Zerlegung so an, dass die dünne Besetztheitsstruktur der System-Matrix A auch in der unteren und oberen Dreiecksmatrix L und U erhalten bleibt.
Der Algorithmus 3.2.14 entspricht der in Caffa implementierten Version des SIP-Verfahrens.

Algorithm 3.2.14 SIP-Methode

for $i = 2, N_x$ **do**
  for $j = 2, N_y$ **do**
    $\mathsf{L}_{\text{West},i,j} = \mathsf{A}_{\text{West},i,j}/(1 + \alpha * \mathsf{U}_{\text{Nord},i,j-1})$
    $\mathsf{L}_{\text{Süd},i,j} = \mathsf{A}_{\text{Süd},i,j}/(1 + \alpha * \mathsf{U}_{\text{Ost},i-1,j})$
    $P_1 = \alpha \mathsf{L}_{\text{West},i,j} \mathsf{U}_{\text{Nord},i,j-1}$
    $P_2 = \alpha \mathsf{L}_{\text{Süd},i,j} \mathsf{U}_{\text{Ost},i-1,j}$
    $\mathsf{L}_{\text{P},i,j} = 1./(\mathsf{A}_{\text{P},i,j} + P_1 + P_2 - \mathsf{L}_{\text{West},i,j}\mathsf{L}_{\text{Ost},i,j-1} - \mathsf{L}_{\text{Süd},i,j}\mathsf{U}_{\text{Nord},i-1,j} + \epsilon)$
    $\mathsf{U}_{\text{Nord},i,j} = (\mathsf{A}_{\text{Nord},i,j} - P_1)\mathsf{L}_{\text{P},i,j}$
    $\mathsf{U}_{\text{Ost},i,j} = (\mathsf{A}_{\text{Ost},i,j} - P_2)\mathsf{L}_{\text{P},i,j}$
  end for
end for
while $|\mathbf{r}| > tol$ **do**
  Löse: $Ly = b$ //Vorwärtseinsetzen
  Löse: $Ux = y$ //Rückwärtseinsetzen
  berechne das Residuum **r**
end while

Die im Algorithmus auftretende Variable α ist ein Dämpfungsparameter, üblicherweise wird $\alpha = 0.92$ gewählt.

[20] Die unvollständige LU-Zerlegung wird häufig mit ILU-Zerlegung, für incomplete LU-Zerlegung, abgekürzt.
[21] SIP-Methode steht für Strongly Implicit Procedure und wird auch das Stone's Verfahren genannt.

Bemerkung 3.2.15 (ILU-Verfahren)
Die ILU-Verfahren werden bei der numerischen Simulation der Navier-Stokes-Gleichungen gerne auch als Vorkonditionierer eingesetzt.

Da Caffa mit nicht versetzten Gittern arbeitet wird zur Lösung der Gleichungen das SIMPLE-Verfahren für nicht versetzte Gitter verwendet, wie es im Buch von Ferziger und Perić [FP08] beschrieben ist.

Den Code von Caffa haben wir im Rahmen dieser Arbeit für Untersuchungen mit dem Automatischen Differenzieren verwendet. Das betrachtete Beispiel für die Simulation wird im Abschnitt 3.3 *Strömungssimulationen und Automatisches Differenzieren* beschrieben und analysiert.

Bemerkung 3.2.16 (Programmcode)
Eine Kopie des Programmcodes von Caffa ist auf der beiliegenden CD im Verzeichnis `Programme/caffa` *enthalten. In dem Verzeichnis finden sich auch die zugehörigen Demo-Konfigurationen.*

Comet

Im Rahmen dieser Arbeit lag der Quellcode der kommerziellen Software Comet unter Berücksichtigung von Verschwiegenheitsvereinbarungen vor. Comet ist eine von CD-Adapco entwickelte kommerzielle Software und wurde in den Programmiersprachen C und Fortran geschrieben. Einige Erweiterungen sind auch in C++ umgesetzt worden. Ebenso wie das zuvor beschriebene Tool Caffa, löst auch Comet die Navier-Stokes-Gleichungen mit der Finiten-Volumen-Methode. Allerdings ist in Comet neben komplexeren Geometrien und unterschiedlicher Gitterarten vor allem die drei-dimensionale Betrachtung von Strömungen möglich. Zusätzlich unterstützt Comet auch noch die Berechnung von strukturmechanischen Problemen und ist damit ein sogenanntes CCM[22]-Tool.
Im Folgenden betrachten wir zuerst den für uns interessanten Hauptalgorithmus von Comet im Algorithmus 3.2.17, anschließend betrachten wir die für uns relevanten Bereiche etwas näher.

Gittererzeugung

Im Gegensatz zu Caffa ist die Gittererzeugung und Modifikation aus dem eigentlichen Programm heraus gelöst. Dieser Ansatz hat ein paar Vorteile, auf die wir noch zu sprechen kommen. Allerdings schränkt er auch die Möglichkeit der simultionsinternen Modifikation des Gitters ein. Wir gehen darauf noch näher ein.
Comet bietet, was die Gittererzeugung angeht, das Tool `Cometpp` an. Dies ist ein eigenständiges Programm, mit dem man Geometrien beschreiben und Gitter erstellen kann. Das Tool ermöglicht ebenfalls die Beschreibung der Strömungskonfiguration für Randwerte, physikalische und numerische Eigenschaften. Zudem dient das Tool auch dem Einlesen von Gittern, die mit einem anderen Programm erzeugt wurden. Hierzu seien die Programme NASTRAN, PATRAN und STAR-CD, genannt.

[22] computational continuum mechanic

Algorithm 3.2.17 Comet Hauptalgorithmus

```
1: readData()
2: initProgram()
3: for iter = 1 → maxIter do
4:    varold()
5:    timeada()
6:    time+ = Δ_time
7:    gridMoving()
8:    //calculateVelocity
9:    solve: A_velocity v = rhs
10:   //calculate Mass
11:   solve: A_pressure p = rhs
12:   //calculateTemperature
13:   solve: A_temperature p = rhs //test
14:   postProcess()
15: end for
```

Comet unterstützt vier Grundtypen an Zellen: Hexaeder, Prisma, Pyramiden und Tetraeder. Durch die vielseitigere Unterstützung von Gitter-Typen sind wir in Comet mit der Beschreibung der Geometrie sehr frei. Ein Benefit dieser Freiheit ist, dass die Erzeugung der Gitter deutlich komfortabler gestaltet werden kann. Der Preis, den wir dafür zahlen müssen, betrifft die Verwendung des Automatischen Differenzierens bei bewegten Gittern, da die Mechanismen zur Gitterbewegung vom Simulationsprogramm Comet getrennt ist.

Bemerkung 3.2.18 (Anleitung Cometpp)
Eine Anleitung zu `Cometpp` *findet man im User Manual [Ins01]. Dort findet man auch Tutorials zur Verwendung von Comet.*

Approximation

Die Approximationsmethoden kann man als Anwender in Comet auswählen. Bei der Berechnung der Gradienten haben wir die Auswahl zwischen folgenden Approximationsmethoden: Der Kleinsten-Fehlerquadraten-Methode[23] oder der auf dem Satz von Gauß basierende Approximation. Das Standardverfahren ist die Kleinste-Fehlerquadrate-Methode, die wir im Abschnitt *Einführung* unter *Kleinste-Fehlerquadrat-Methode* beschrieben haben.
Zur Approximation der Zeitdiskretisierung stehen das implizite Euler-Verfahren oder ein Verfahren zweiter Ordnung welches drei Zeitschritte verwendet, zur Auswahl. Eine detaillierte Beschreibung und Herleitung des impliziten Euler Verfahrens findet man beispielsweise in Stoer und Bullrisch [SB00].
Im Algorithmus 3.2.19 ist das implizite Euler-Verfahren dargestellt. Abhängig von der Funktion f entspricht das "Lösen" in Zeile 3, dem Lösen eines linearen oder nicht-linearen Gleichungssystems.

[23] Dieses Verfahren wird in Comet als Polyfit Methode bezeichnet.

Algorithm 3.2.19 Implizites Euler-Verfahren

for $k = 1 \rightarrow K$ **do**
  $t_k = t_0 + k \cdot \Delta t$
  Löse: $x_k = x_{k-1} + \Delta t f(t_k, x_k)$
end for

Der Algorithmus 3.2.20 beschreibt das verwendete drei-Term-Verfahren für äquidistante Zeitschritte. Das drei-Term-Verfahren ist ein Verfahren zweiter Ordnung. Das Verfahren ist in dem *Comet User Manual* [Ins01] beschrieben.

Algorithm 3.2.20 Implizites Drei-Term-Verfahren

for $k = 1 \rightarrow K$ **do**
  $t_k = t_0 + k \cdot \Delta t$
  Löse: $x_k = x_{k-1} + \frac{2}{3}\Delta t f(t_k, x_k) + \frac{1}{3}\left(x_{k-1} - x_{x-2}\right)$
end for

Lösungsverfahren

Comet bietet eine Auswahl an Lösungsmethoden zum Lösen der entstehenden Gleichungssysteme. Zusätzlich werden die linearen Gleichungssysteme vorkonditioniert. Auch hier hat man die Wahl, was das Verfahren betrifft. Zum Lösen der symmetrischen linearen Gleichungssysteme stehen folgende Verfahren zur Verfügung:

- CG: ist das Konjungierte-Gradienten-Verfahren.
- CGSTAB: entspricht dem im Buch [Mei05] als BiCGSTAB beschriebenen Verfahren.
- AMG: ist ein Algebraische-Multi-Grid-Verfahren.

Auf die hier beschriebenen Verfahren möchten wir nicht näher eingehen. Hierzu sei auf die bereits erwähnte Literatur von Meister [Mei05] und Kanzow [Kan00] verwiesen. Zu dem AMG-Verfahren sei noch erwähnt, dass die Realisierung innerhalb von Comet über eine Schnittstelle zu C++ realisiert ist.
Bei den nicht-symmetrischen Verfahren steht nur das CGSTAB-Verfahren zur Verfügung. Als Vorkonditionierer stehen erneut drei Verfahren zur Auswahl:

- IC: steht für die unvollständige Cholesky-Zerlegung (engl. incomplete Cholesky).
- J: steht für das Jacobi-Verfahren als Vorkonditionierer.
- AMG: verwendet das Algebraische-Multigrid-Verfahren als Vorkonditionierer.

Bei der unvollständigen Cholesky-Zerlegung wählt man die Vorkonditionierungsmatrix B wie folgt:

$$\mathsf{B} := (\mathsf{L} + \bar{\mathsf{D}})\bar{\mathsf{D}}(\bar{\mathsf{D}} + L^T), \tag{3.103}$$

wobei in der Gleichung (3.103) für L und $\bar{\mathsf{D}}$ folgendes gilt:

$$\begin{aligned} \mathsf{L} &:= A_{i,j} \quad \text{für } i = j - 1 \\ \bar{\mathsf{D}} &:= \mathbf{diag}\left(\mathsf{A}_{i,i} - \sum_{k=1}^{i-1} \mathsf{L}_{i,k}, \mathsf{L}_{k,i}\bar{\mathsf{D}}_{k,k}\right). \end{aligned} \tag{3.104}$$

Diese Wahl als Vorkonditionierungsmatrix ist im Normalfall besser als die Jacobi-Vorkonditionierung geeignet, da bei der Jacobi-Vorkonditionierung nur die Inverse der Diagonale von der Matrix A verwendet wird, das heißt:

$$B := \mathbf{diag}\mathsf{A}. \tag{3.105}$$

Die Voreinstellungen wählen das CG-Verfahren für die symmetrischen linearen Gleichungssysteme, das BiCGSTAB-Verfahren für die nicht symmetrischen linearen Gleichungssysteme und als Vorkonditionierer die unvollständige Cholesky-Zerlegung

Comet stellt dem Anwender die Möglichkeit, einen benutzerspezifischen Code über fest definierte Schnittstellen einzubinden. Über diese Schnittstellen können wir bei bewegten Gittern die Bewegungen realisieren sowie Zielfunktionale, die über die Strömungsgrößen definiert werden, zur Laufzeit der Simulation mitberechnen. Weitere Schnittstellen sind zum Beispiel: Möglichkeiten zur Erweiterung der Transportgleichungen, benutzerabhängige Initialisierung oder nutzerdefinierte Randbedingungen.

Weiterhin unterstützt Comet die Parallelisierung der Simulation. Dieser Aspekt ist gerade bei großen und aufwendigen Simulationen von entscheidender Bedeutung. Da es allerdings in unserem Kontext keine Rolle spielt, gehen wir hierauf nicht weiter ein.

3.3 Strömungssimulationen und Automatisches Differenzieren

Bei der Betrachtung von Strömungssimulationen besteht häufig ein Interesse das simulierte Problem zu optimieren. Da die Simulationsprogramme uns nur die Zielgrößen und nicht die Ableitungswerte zur Verfügung stellen, müssen wir uns auf Ableitungsfreie Optimierungsverfahren beschränken. Um diese Problematik zu lösen, untersuchen wir im folgenden die Anwendung des Automatischen Differenzierens auf Strömungssimulationen. Als Basis für unsere Untersuchungen verwenden wir die zwei vorgestellten Programme Caffa und Comet.
Die Analyse des Verhaltens des Automatische Differenzierens in der Strömungsmechanik, ist nach dem verwendeten Simulationsprogramm gegliedert. Damit ergibt sich ein Abschnitt für *Caffa* und ein Abschnitt für *Comet*. Alle hier untersuchten Fälle verwenden bewegte Gitter. Im Rahmen der Diplomarbeit *Automatic Differentiation and flow simulation* von Ralf Leidenberger [Lei07] wurde der Aspekt von festen Gittern mit Comet und dem Automatischen Differenzieren bereits untersucht.

3.3.1 Caffa

Als Fallstudie für die Simulation ziehen wir uns auf einen bereits untersuchten Versuchsaufbau zurück, da wir durch die eingehende Betrachtung dieses Beispiels Rückschlüsse auf die auftretenden Phänomene bei Comet gewinnen möchten.

Simultionskonfiguration

Da Caffa, wie im Abschnitt *Caffa* , betrachtet nur eine eingeschränkte Fähigkeit zur Beschreibung der Geometrie besitzt, sind wir, was die Auswahl des Problems angeht, ebenfalls eingeschränkt. Wir ziehen uns auf die Betrachtung des folgenden Problems zurück:Wir betrachten

das Auftriebs Verhalten eines Tragflügel bezüglich der Änderung seines Anstellwinkels zur Anströmung.
Diese Konfigurationen der Strömungsgeometrie haben wir in der Abbildung 3.15 dargestellt.

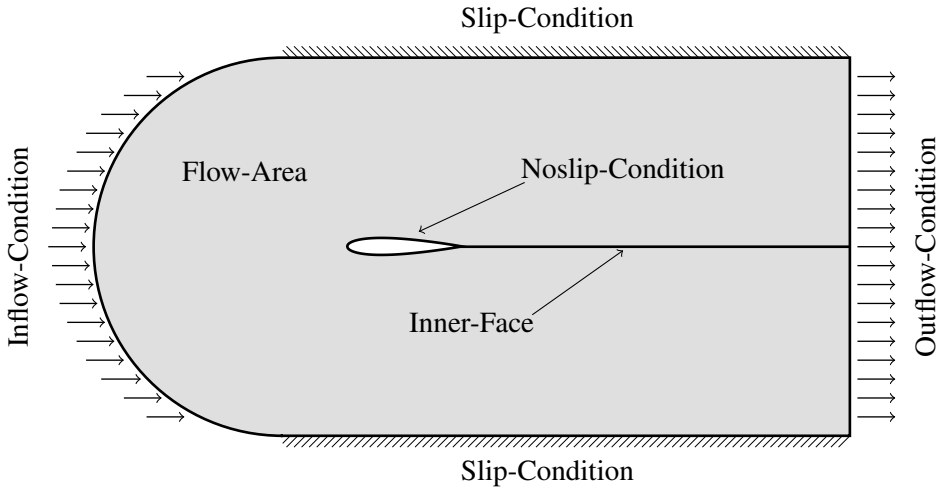

Abbildung 3.15: Strömungsgebiet des betrachteten Caffa-Testfalls.

Die linke halbrunde Seite des Strömungsgebiets ist als Einströmbedingung konfiguriert, die gegenüberliegende Seite haben wir auf Ausströmbedingungen gesetzt. Für die oberen und unteren Seiten haben wir Gleitrandbedingungen gewählt. Für die Oberfläche des Tragflügels mit einem Naca0015-Profil haben wir die Haftrandbedingung gewählt. Aufgrund der Geometriebeschreibung in Caffa mussten wir noch eine innere Oberfläche festlegen. Diese Zellen werden dann nicht als Randzellen, sondern als innere Zelle behandelt und sie korrespondieren mit der entsprechenden gegenüberliegenden Zelle.
Als Medium für das Strömungsfluid haben wir eine Dichte von 1.0 mit Viskosität 0.0002 und einer Prandtl-Zahl von 10.0 angenommen. Für die Umgebungsgröße Temperatur wurde 0.1 und für die Gravitationskräfte wurde 0 angenommen. Dies entspricht der Konfiguration des Testfalls von Caffa. Als Einströmbedingungen wurde eine konstante Geschwindigkeit in x-Richtung von 1.0 angenommen.

Das Verhalten dieser Problemstellung ist seitens der Physik und der Strömungsdynamik sehr gut bekannt. Die Auftriebskraft hat bei konstanten Einströmbedingungen und einem variablen Anstellwinkel einen konkaven Verlauf, wenn wir innerhalb gewisser Schranken bleiben, sodass die Strömung am Tragflügel nicht abreißt. Die Auftriebskraft berechnet sich über den auf den Tragflügel wirkenden Druck.

$$\mathbf{f} = \int_{\Omega_{\text{Flügel}}} p \cdot \nu dS, \tag{3.106}$$

Die y-Komponente des Kraftvektors $\mathbf{f}$ ist die gesuchte wirkende Auftriebskraft. In Gleichung (3.106) beschreibt p den Druck, ν ist die äußere Normale und $\Omega_{\text{Flügel}}$ repräsentiert die Oberfläche des Tragflügels. Das Druck-Integral über den Tragflügel lässt sich in der Simulation als die y-Komponente der Summe der äußeren Normalen multipliziert mit dem Druck auf jedem Oberfläche-Element des Tragflügels berechnen (3.107), da wir unser Gitter aus geraden

Stücken approximieren.

$$\mathbf{f} = \sum_{i=1}^{N_{\text{Flügel}}} p_i \cdot \nu_i, \tag{3.107}$$

wobei N_i die Anzahl der geraden Stücke des Tragflügel ist und p_i und ν_i sind der auf dem Oberflächenstück wirkende Druck, beziehungsweise die äußere Normale mit der Länge des geraden Stücks.

Die Bewegung des Tragflügels wird über den Anstellwinkel α in folgender Form beschrieben:

$$\mathbf{x}_{i,neu} = x_{i,alt,x}\cos(\alpha) + x_{i,alt,y}\sin(\alpha). \tag{3.108}$$

Die Gleichung wird auf alle Randpunkte des Nordrandes, an deren Oberflächen die Haftrandbedingung gesetzt ist angewendet. Für die Bereiche des Nordrandes, an denen die Innere-Oberflächen-Bedingung gelten, werden die Knotenpunkte von der Tragflügel-Endleiste zu ihrer Position am Ausstromrand linear angepaßt.
Diese Geometrie-Konfiguration wurde bereits im Rahmen der Diplomarbeit *Automatic Differentiation in Flow Simulation* von Leidenberger [Lei07] und in dem Artikel von Leidenberger und Urban *Automatic Differentiation for the Optimization of a Ship Propulsion and Steering System: A proof of concept* [LU11] untersucht. In diesen Arbeiten stand allerdings die generelle Machbarkeit des Ansatzes im Vordergrund. Im Rahmen der vorliegenden Arbeit betrachten wir diese Konfiguration mit dem Fokus auf Phänomene, in denen sich diese Caffa-Simulation zu einer vergleichbaren Simulation in Comet unterscheidet. Diese Unterschiede sind der Grund weshalb in Comet kein stabiler Zustand erreicht wird.

Im nächsten Abschnitt befassen wir uns mit der Anwendung des Automatischen Differenzierens auf Caffa für diesen Testfall.

Automatisches Differenzieren

Da Caffa in Fortran geschrieben ist, bietet sich die Verwendung des Tools Tapenade, welches wir im Kapitel 2 *Automatisches Differenzieren* bereits erwähnt haben, zur Erzeugung des differenzierten Codes an.
Die Änderung des Anstellwinkels wird durch den Parameter α beschrieben, daher differenzieren wir die Auftriebskraft f_y bezüglich α. Aus der Gleichung (3.108) geht hervor, dass α nur Einfluss auf die Geometrie und somit auf das Gitter hat und so nur indirekt auf die Auftriebskraft wirkt. Da das Gitter in die Berechnung der Zustandsgrößen der Strömung eingeht, hat es somit auch auf den Druck p Einfluss. Da aus der Gleichung (3.107) hervorgeht, dass zur Berechnung der auf den Tragflügel wirkenden Kraft $\mathbf{f}$ außer dem Druck noch die äußere Normale Einfluss hat, ergibt sich für die Ableitung der Auftriebskraft:

$$\frac{\partial f}{\partial \alpha} = f^D = \sum_{i=1}^{N_{\text{Flügel}}} {p^D}_i \cdot \nu_i + p_i \cdot {\nu^D}_i. \tag{3.109}$$

Hier bezeichnen wir mit $\cdot^D$ die durch das Automatisch Differenzieren berechnete Ableitung. Die Beschreibung der Änderung der äußeren Normalen ν_i bezüglich α ergibt sich direkt aus den zur Geometrie gehörenden Zustandsgrößen. Die Ableitung des Druckes p nach dem Anstellwinkel p ergibt sich durch Differenzierung der kompletten Simulation.

Da Caffa vollständig in Fortran geschrieben ist gibt es bei der Anwendung von Tapenade keine Probleme. Nach der Differentiation muss nur noch die Initialisierung des Anstellwinkel, α und die Ausgabe der Ableitung der Auftriebskraft angepaßt werden.

Um eine Divergenz der Simulation zu vermeiden, führen wir zunächst 600 Zeit-Schritte mit einem Anstellwinkel von 0° durch. Erst dann drehen wir den Flügel in die gewünschte Position, da ansonsten der Löser aufgrund einer noch nicht stabilen Strömung nach einem Zeitschritt nicht konvergiert. Eine Drehung des Flügels direkt bei Erzeugung der Gittergeometrie führt zwar dazu, dass die ursprünglichen Strömungsgrößen konvergieren, allerdings erreichen wir so keine Konvergenz der Ableitungswerte, da die anfängliche große Variation der Strömungsgrößen die Ableitungswerte zu stark stören.

Ein vollständiges lauffähiges Caffa-Programm inklusive der differenzierten Version finden wir auf der beiliegenden CD im Verzeichnis `Programme/caffa`.

Untersuchungen

In diesem Abschnitt befassen wir uns mit der Analyse des differenzierten Caffa-Codes, um mögliche Stellen, an denen die Berechnung der Ableitungswerte schief gehen kann, zu lokalisieren.

Die Gradienten-Berechnung benötigt gesonderte Betrachtung, da die Berechnung des Gradienten zur Klasse eines Problems mit invariantem Quotient gehören kann. Dies ist davon abhängig, ob das entsprechende Volumen vom Anstellwinkel abhängig ist.
Die Berechnung des Gradienten mit Caffa erfolgt nach der in Gleichung (3.100) beschriebenen Form. Hier sehen wir direkt, dass neben dem Volumen der Zelle noch die Oberflächen-Vektoren eingehen. Da das Gitter bei jeder Änderung des Anstellwinkels neu erzeugt wird, ändern sich alle Zell-Volumina in der näheren Umgebung des Flügels mit sehr hoher Wahrscheinlichkeit, wenn auch nur in einem sehr geringen Bereich.

Beispiel 3.3.1 (Änderung Zell-Volumina)
Wir untersuchen die durchschnittliche Änderung der Zell Volumina bei der Änderung des Anstellwinkels um 0.1° *bei der in Abbildung 3.15 dargestellten Gitterkonfiguration.*
In der Abbildung 3.16 ist die prozentuale Änderung der Volumina für jede Zelle in der Gittergeometrie veranschaulicht. Wir haben für jede Zelle an ihrer gemittelten Position $\mathbf{x}_m := \frac{1}{2}(\mathbf{x}_{old+}\mathbf{x}_{new})$ *einen Punkt gezeichnet, wobei die Farbe die Größe der Änderung verdeutlicht. Rot steht für eine sehr kleine Abweichungen, dann geht es über Gelb- zu Grün-Tönen.*
Die betrachteten Volumina sind in der Größenordnung von 10^{-5}*, und die Änderungen sind im Bereich von ungefähr* $0,1$ *bis* 3 *Promille.*

Als nächstes betrachten wir das Verhalten der Ableitungswerte in den ersten Zeit-Schritten nach der Änderung des Anstellwinkels.

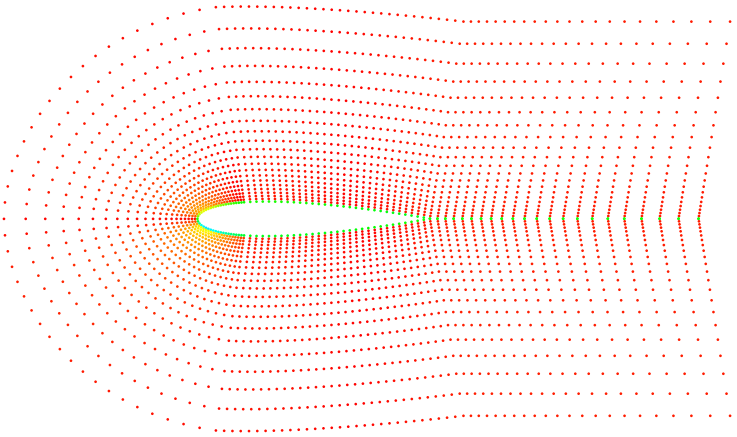

Abbildung 3.16: Volumenänderung der Kontrollvolumen bei einer Drehung um $0.1°$.

Beispiel 3.3.2 (Kraft-Gradienten-Werte)
Analog zum Beispiel 3.3.1 stellen wir die berechneten Werte grafisch aufbereitet dar, da es in dieser Betrachtung nicht um die exakten Werte geht, sondern um das Wissen, in welcher Größenordnung sich die Werte befinden und welche Bereiche kritischer bezüglich des Anstellwinkels sind.
Wir betrachten erneut eine Änderung des Anstellwinkels um $0,1°$. *In der Abbildung 3.17 sehen wir die Kraft-Gradienten auf den Zell-Oberflächen des Tragflügels im zeitlichen Verlauf. Zu beachten ist, dass die Gradienten in ihrer Größe skaliert sind, damit die Flügelgeometrie immer gleich groß dargestellt wird. Wir können folgende Aussagen über das Verhalten der Kraft-Gradienten treffen. Direkt nach der Drehung sind die Gradienten relativ groß und schwanken stark in ihrer Ausrichtung. Abgesehen von den Gradienten im Nasen- und Endleisten-Bereich, weisen die Gradienten eine hauptsächlich vertikale Ausrichtung auf. Daran lässt sich die Auswirkung des Anstellwinkels auf die Auftriebskraft deutlich erkennen.*

Erkenntnisse

Die zentrale Erkenntnis ist, dass der Ansatz des Automatischen Differenzierens bei Caffa zu vernünftigen Ergebnissen kommt. Wir haben die Werte mit den zugehörigen Finiten-Differenzen verglichen. Es hat sich gezeigt, dass die Ergebnisse, die wir mit dem Automatischen Differenzieren berechnet haben, eine glättende Eigenschaft haben, bezüglich des Rauschverhaltens, welches bei Finiten Differenzen auftritt, haben.
Weiter sehen wir, dass das Automatische Differenzieren bei der Berechnung der Ableitungen innerhalb eines bewegten Gitters kein grundsätzliches Problem hat, solange die Bewegung des Gitters durch eine differenzierbare Funktion beschrieben wird. Dies ist in unserem Fall mit einer Drehbewegung des Flügels gegeben.
Allerdings muss berücksichtigt werden, dass Caffa ein reiner 2D-Löser ist und sich somit gewisse Strukturen im Vergleich zu einem 2D/3D-Löser stark vereinfachen. Durch die gleichzeitige starke Einschränkung auf relativ einfache Gitter kann man damit gerade bei der Berechnung der Gradienten eine übersichtlichere und bessere numerische Formulierung realisieren.

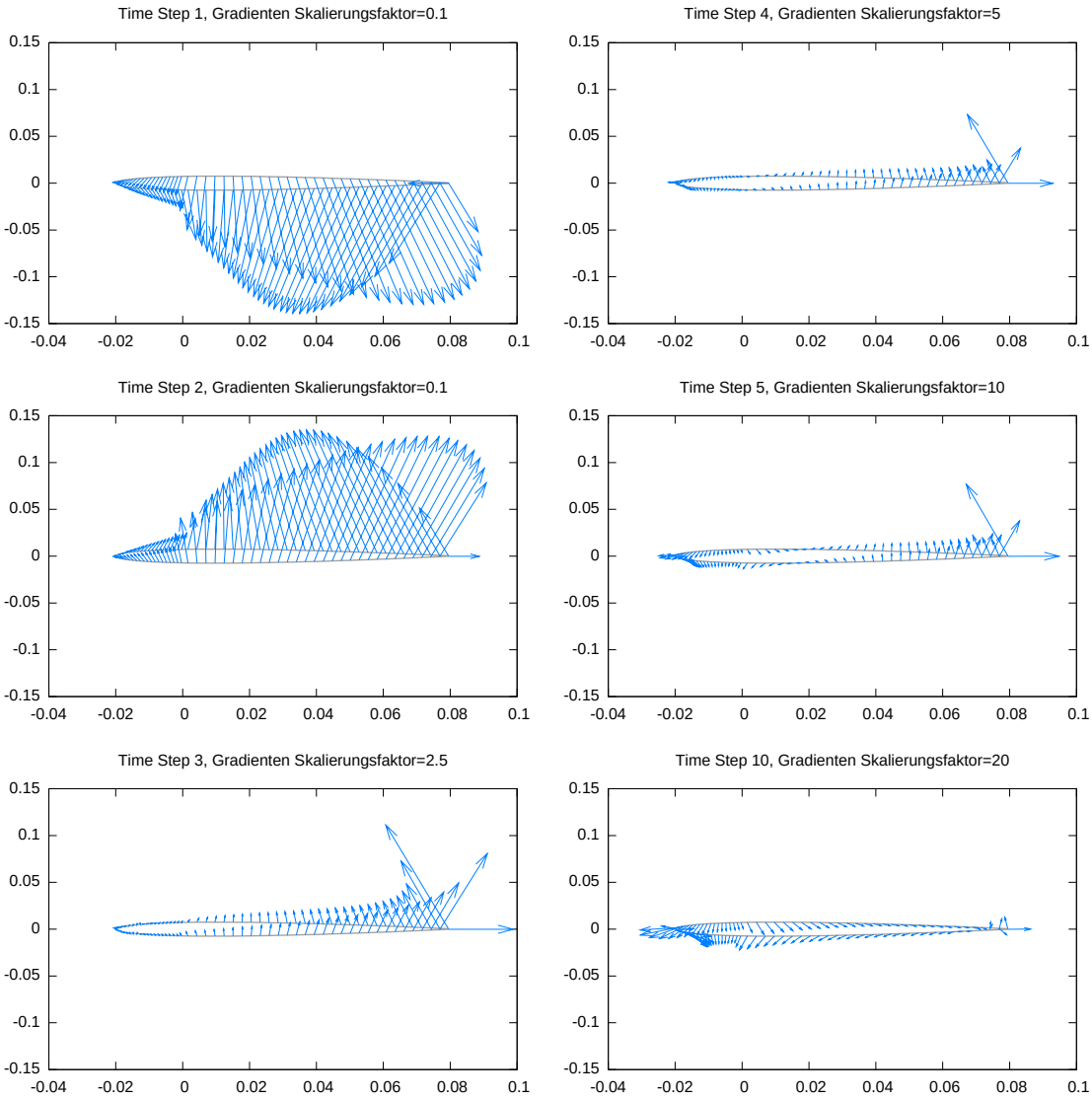

Abbildung 3.17: Zeitliche Änderung der Kraft-Gradienten nach einer Drehung um $0.1°$.

3.3.2 Comet

Im Folgenden untersuchen wir das Verhalten von Comet, wenn wir mittels dem Automatischen Differenzieren die Ableitungswerte berechnen. Hierfür betrachten wir zwei grundsätzlich unterschiedliche Konfigurationen. Als erstes betrachten wir die Anwendung auf ein *festes Gitter* und danach die Anwendung bei *bewegten Gittern*. Innerhalb der bewegten Gitter haben wir erneut zwei, vom Konzept her unterschiedliche, Konfigurationen. Im ersten Fall betrachten wir hier ein großes Gitter, in dessen Inneren ein bewegter Gitterblock zum Einsatz kommt. Die beiden Gitterblöcke werden mit einem sogenannten *sliding Interface* gekoppelt. Im zweiten Fall haben wir die Gittergeometrie so gewählt, dass wir das gesamte Gitter bewegen können. Dieser Fall entspricht von der Konfiguration der Geometrie her ungefähr dem Fall mit dem

festen Gitter.

Neben den drei Geometrie-Konfigurationen verwenden wir auch bei den internen Lösungsmethoden Variationen. So haben wir beim Lösen der linearen Gleichungssysteme sowohl den direkten Ansatz gewählt, das heißt wir lösen $\mathbf{A}\mathbf{x} = \mathbf{b}$ mit einem differenzierten Lösungsalgorithmus, als auch den im Abschnitt 2.2.3 beschrieben modifizierten Lösungsansatz. Bei der Gradienten-Berechnung haben wir beide von Comet zur Verfügung gestellten Verfahren getestet.

Wir betrachten analog zu Caffa erneut einen umströmten Körper und berechnen die aufgrund der Strömung auftretenden Kräfte auf diesen Körper. Für die auftretenden Kräfte gilt, dass sie sich als das Integral über das Produkt von Druck und der äußeren Normalen entlang der Körperoberfläche ergeben.
Wenn wir im Folgenden von den wirkenden Kräften sprechen, meinen wir immer die durch die Strömung erzeugten auf den umströmten Körper wirkenden Kräfte. Unter der Bezeichnung der Ableitungswerte sind ab jetzt immer, sofern nicht explizit etwas anderes gesagt wird, die Ableitungswerte der auf den Körper wirkenden Kraft bezüglich des Anstellwinkels gemeint.

Festes Gitter

Eine Konfiguration zur Berechnung der Auftriebskräfte an einem Tragflügel mit einer starren Gitterkonfiguration haben wir bereits in [Lei07] und [LU11] betrachtet. Wir beziehen uns noch einmal auf diesen Fall, um anschließend einen Vergleich zu den bewegten Gittern machen zu können.

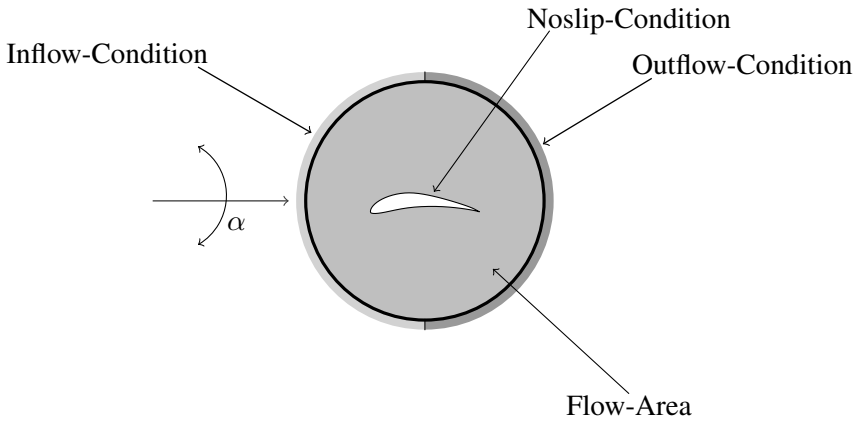

Abbildung 3.18: Testfall Strömungssimulation: Tragflügel, mit festem Gitter.

In der Abbildung 3.18 ist die Konfiguration des betrachteten Problemfalls veranschaulicht. Da wir ein festes Gitter betrachten, wird die Änderung des Anstellwinkels durch eine Änderung des Winkels der Einströmbedingung realisiert.

Bemerkung 3.3.3 (Anstellwinkel–Anströmwinkel)
Es gilt zu beachten, dass der Anstellwinkel dem negativen Anströmwinkel entspricht. Im Folgenden verwenden wir den Anstellwinkel als Referenzgröße.

Die Untersuchungen im Rahmen der oben angegebenen Quellen haben gezeigt, dass es grundsätzlich möglich ist, in Comet Ableitungswerte mit dem Automatischen Differenzieren zu berechnen. An dieser Stelle verweisen wir für eine Betrachtung der Simulationsergebnisse dieses Falles auf die beiden bereits genannten Literaturangaben [Lei07] und [LU11].

Bewegtes Gitter

Innerhalb der bewegten Gitter betrachten wir drei verschiedene Gitter und Bewegungskonfigurationen. Wie wir bereits erwähnt haben, treffen wir eine Unterscheidung, ob wir das Gitter vollständig drehen oder ob wir nur einen Teil des Gitters drehen und es mit einem Sliding Interface koppeln.

Ein *sliding Interface* ist eine Randbedingung die paarweise auftritt. Damit ist gemeint, dass wir zwei Gitterblöcke haben, die an einem Rand aufeinander liegen. Allerdings gibt es keine feste Verbindung, sodass man nicht jeder Zelle des einen Block eine fest zugehörige Zelle des anderen Blocks zuweisen kann. Diese Randbedingung wird gewählt, wenn ein Block bewegt wird und sein Rand an dem Rand des anderen Blocks entlang gleitet.

In jeder Gitter-Konfiguration betrachten wir die Gradienten-Berechnung mit der kleinsten-Fehlerquadrate-Methode,[24] die wir in den Gleichungen (3.92) bis (3.97) beschrieben haben und zusätzlich betrachten wir einen Fall mit der Gauß-Methode, welche wir in den Gleichungen (3.87) bis (3.90) betrachtet haben und die auch in Caffa zum Einsatz kommt.

Die Bewegung des Gitters ist in Comet, anders als in Caffa, normalerweise extern realisiert. Dies bedeutet, dass wir das Gitter über die Meta-Sprache von dem zugehörigen Präprozessor-Tool `Cometpp` bewegen. Die Programmroutine, die die Bewegung ausführt, startet nun das Tool `Cometpp`, modifiziert mit diesem das Gitter und liest das so erzeugte Gitter erneut in Comet ein. Dieser Ansatz ist bei der Verwendung des Automatischen Differenzierens nicht möglich, da wir sonst sämtliche Informationen über die Bewegung des Gitters verlieren würden. Dies resultiert aus der Tatsache, dass nur das eigentliche Programm differenziert wird.
Deshalb greifen wir tiefer in die Struktur von Comet ein. Wir bewegen jede Zelle mit ihren zugehörigen Größen von Hand innerhalb der laufenden Simulation. Da die Verwaltung des Gitters wiederum nicht in einer homogenen Struktur vorliegt, müssen wir gewisse Dinge berücksichtigen. Ein Teil der Verwaltung des Gitters ist in C anstatt Fortran geschrieben. Wie wir aus dem Abschnitt *Verschiedene Programmiersprachen* im Kapitel 2 wissen, ist die Verknüpfung des Automatischen Differenzierens mit unterschiedlichen Programmiersprachen nicht so einfach und in diesem speziellen Fall ist es aufgrund des Umfanges des Programms und der Unübersichtlichkeit nicht realisierbar, da man mit globalen Speicherbereichen arbeitet und Comet nicht für die direkte Manipulation der Gitter ausgelegt ist. Wollte man diese Einschränkung beheben, müsste Comet zu großen Teilen um- beziehungsweise neugeschrieben werden. Um die Probleme der Vermischung der Programmiersprachen zu vermeiden, haben wir bei der Gitterbewegung die Einschränkung, dass wir die Volumen der Zellen nicht verändern dürfen. Da wir als Modellproblem die Änderung der Auftriebskräfte bezüglich des Anstellwinkels betrachten, ist dies für uns keine wirkliche Einschränkung, da die Volumen der Zellen invariant bezüglich der Rotation sind.

[24] Dies ist der Standardfall in Comet.

Wir betrachten die Gitter-Konfigurationen in der Reihenfolge, wie wir sie untersucht haben und welche Probleme wir mit der entsprechenden Änderung untersucht haben.

Testfall 1

Bei dieser Testkonfiguration haben wir uns an der bereits mit Caffa untersuchten Geometrieanordnung orientiert. Die signifikanteste Änderung betrifft das Profil des Tragflügels. Während wir bei Caffa ein Naca0015-Profil betrachtet haben, verwenden wir hier ein Eppler-Profil mit der Nummer E420. Diese Änderung beruht auf der Erkenntnis, dass das verwendete Eppler-Profil bessere[25] Eigenschaften besitzt.

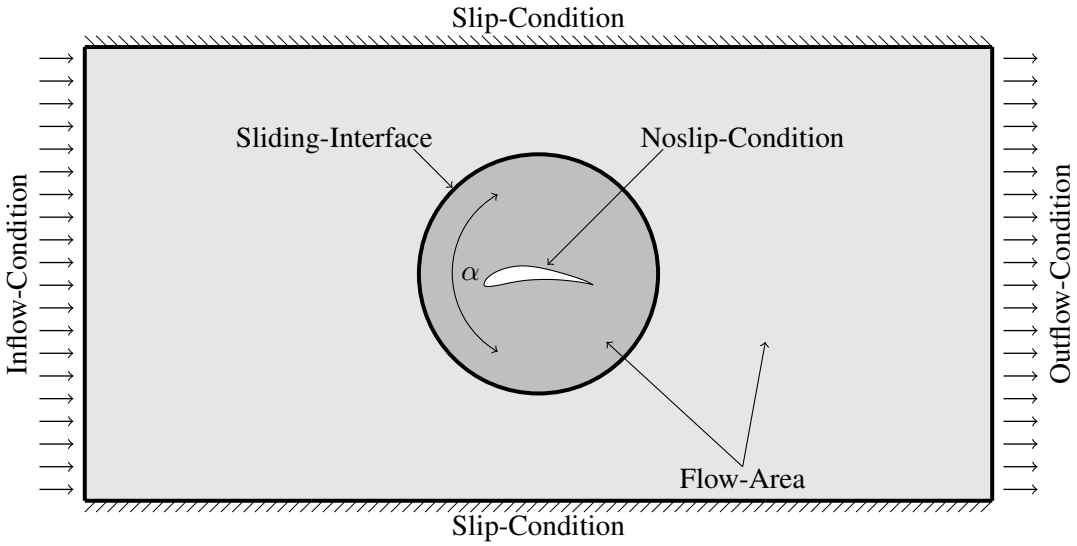

Abbildung 3.19: Testfall Strömungssimulation: Tragflügel.

Ein weiterer Unterschied zur Konfiguration von Caffa betrifft die Art der Gittererzeugung und die Art der Gitterbewegung in Comet. Im Detail heißt dies, dass die Geometrie in Caffa den Bogen nur aus Gründen der Geometrie-Beschreibung benötigt, in Comet haben wir, wie in der Abbildung 3.19 zu sehen ist, eine gerade Einströmfront. Da wir die Gitterbewegung in Comet mit einem anderen Ansatz realisieren müssen, ist das Gebiet in zwei Blöcke aufgeteilt: Einen starren Block und einen rotierenden Block, in welchem der Tragflügel eingebettet ist. Die beiden Blöcke sind mit einem *sliding Interface* gekoppelt.
Die Erwartung, dass sich die beschriebene Geometrie ähnlich wie der Testfall in Caffa verhält, konnte leider nicht erfüllt werden. Die Strömungsgrößen haben zwar ein physikalisches Verhalten gezeigt, sodass wir einen gravierenden Fehler in der Beschreibung der Geometrie und der Parameterwahl ausschließen können, allerdings hat sich dieses Verhalten nicht auf die Ableitungswerte übertragen.

Die Konfiguration der Lösungsmethoden sowie die der Strömungsparameter beschreiben wir im Anschluss an die Beschreibung der drei Geometrie-Varianten.

[25] In diesem Fall ist mit „besseren“ Eigenschaften „gutmütigere“ Eigenschaften gemeint.

Testfall 2

Im zweiten Testfall modifizieren wir den Testfall 1, sodass wir die auftretenden Probleme mit den Ableitungswerten aufgrund schlechter Zellengeometrie ausschließen können. Um dies zu gewährleisten, haben wir den Tragflügel durch einen Zylinder als umströmten Körper ersetzt.

Da ein Zylinder ein rotationssymmetrischer Körper ist, kennen wir das physikalische Verhalten bezüglich einer Änderung des Anstellwinkels, wobei hier die Bezeichnung Anstellwinkel nur im direkten Vergleich mit dem Testfall 1 passend ist. Da es sich um einen rotationssymmetrischen Körper handelt sind die auf ihn wirkenden Kräfte unabhängig des „Anstellwinkels“.

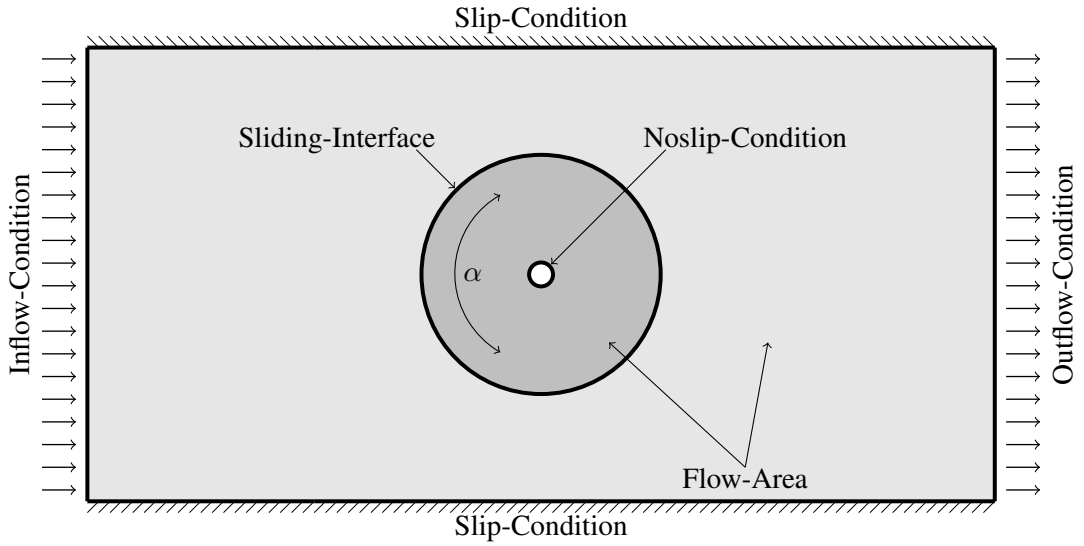

Abbildung 3.20: Testfall Strömungssimulation: Kreis.

In der Abbildung 3.20 sehen wir die Konfiguration des Testfalls 2. Wir sehen direkt, dass wir - bis auf den umströmten Körper - die gleiche Geometrie wie bei Testfall 1 gewählt haben. Die numerischen Experimente mit dieser Konfiguration haben uns gezeigt, dass die Divergenz der Ableitungswerte aus dem ersten Testfall nicht auf schlechte Gittereigenschaften zurückzuführen ist.

Testfall 3

Da wir mit dem zweiten Testfall die Zellengeometrie bei einem umströmten Tragflügel als Ursache ausschließen konnten, betrachten wir in diesem Fall erneut ein Eppler E420 als Tragflügel. In der Abbildung 3.21 ist die Geometrie dieses Testfalls veranschaulicht.

Wir sehen direkt, dass wir in diesem Testfall kein *sliding Interface* haben. In diesem Testfall möchten wir überprüfen, ob das *sliding Interface* eine mögliche Ursache für die auftretenden Probleme bei der Berechnung der Ableitungswerte ist.

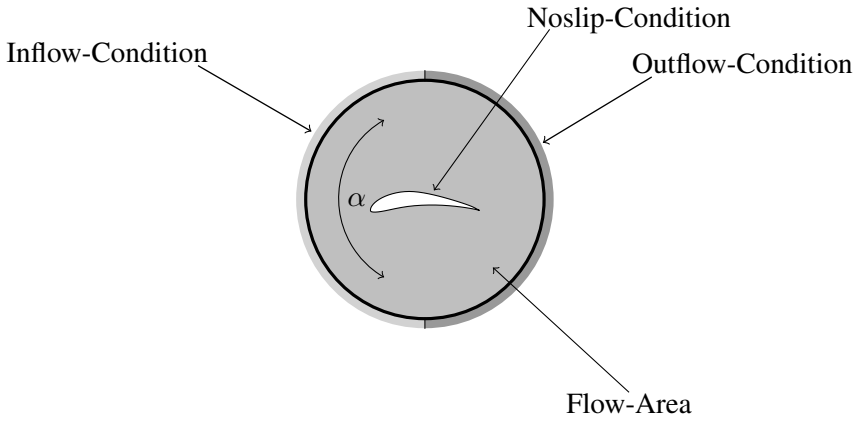

Abbildung 3.21: Testfall Strömungssimulation: Tragflügel rundes Simulationsgebiet.

Die Konfiguration der Gittergeometrie entspricht der aus dem Fall des festen Gitters. Aber anstatt dass wir die Einströmbedingung in der Richtung variieren, drehen wir in diesem Fall das gesamte Gitter.

Untersuchungen

Da die Anwendung des Automatischen Differenzierens bei Comet in Kombination mit bewegten Gittern nicht funktioniert hat, haben wir eine Vielzahl von Methoden und Verfahren getestet, um die Ursache beziehungsweise die Ursachen für das Nichtfunktionieren zu lokalisieren.

Wir klassifizieren die Methoden und Verfahren nun zuerst und betrachten diese dann detailliert. Neben den beschriebenen Geometrien haben wir die Lösung der Gleichungssysteme und die Art der Gradienten-Berechnung sowie die Strömungsparameter der Simulationen variiert und untersucht.

Gitter-Konfiguration

Wir haben drei Gittervarianten untersucht, um auszuschließen, dass die numerischen Experimente aufgrund einer schlechten Geometrieeigenschaft bei der Berechnung der Ableitungswerte misslingen.
Da die Berechnung der direkt simulierten Größe und die auf den Körper wirkenden Kraft in allen Geometrievarianten stabil berechnet werden kann, können wir ausschließen, dass eine grundsätzliche Fehlkonfiguration der Geometrie beziehungsweise der Geometriegrößen vorliegt.
Um die naheliegende mögliche Ursache für dieses Verhalten zu untersuchen, haben wir die Geometrie des umströmten Körpers durch einen Zylinder ersetzt. So ist sichergestellt, dass alle Zellen gute Eigenschaften besitzen und nicht verzerrt sind oder ungünstige Eigenschaften aufgrund spitzer Winkel besitzen. Da diese Modifikation keine Änderung im Verhalten bei der Berechnung der Ableitungswerte bewirkt hat, müssen wir weitere Untersuchungen anstellen.
Damit verbleibt noch die Möglichkeit, dass das sliding Interface als mögliche Problemstelle bei der Berechnung der Ableitungswerte fungiert. Dies haben wir durch die Verwendung der

bereits erprobte Geometrie aus dem unbewegten Comet-Fall zur Berechnung der Ableitungswerte mittels der Variation des Anströmwinkels untersucht.
Dass das Problem im Testfall in dem wir das Gitter vollständig drehen, an den Eigenschaften der Zellen liegt, kann aufgrund der positiven Ergebnisse im unbewegten Fall ausgeschlossen werden. Damit erübrigt sich die Betrachtung eines vollständig bewegten Gitters in Verbindung mit einem umströmten Zylinder.

Lösung linearer Gleichungssysteme

Den Einfluss der Lösungsverfahren für die auftretenden linearen Gleichungssysteme haben wir bei der Problemanalyse berücksichtigt. Da wir den Quellcode von Comet mittels dem Source-Transformations-Ansatz differenziert haben, wurden die Ableitungswerte der Gleichungssysteme zunächst einfach mitberechnet. Da es bei diesem Vorgehen das Risiko gibt, dass zwar die Lösung der Funktionswerte konvergiert hat, aber die zu den Ableitungswerten gehörende Lösung noch nicht konvergiert hat, haben wir ebenfalls die in der Bemerkung 2.2.12 beschriebene Transformation angewandt. Damit haben wir zum Einem einen geringeren Aufwand beim Lösen der Gleichungssysteme, zum Anderen können wir die Konvergenz der Ableitungswerte kontrollieren und die Vorkonditionierungsmethoden des ursprünglichen Lösungsansatzes können auch auf das Problem angewandt werden.
Bei der Simulation konnten wir sicherstellen, dass sowohl die Lösung des ursprünglichen Problems als auch die Lösung der Ableitung konvergieren. Bei den Ableitungswerten traten nach vorangeschrittener Simulationszeit allerdings Divergenzprobleme auf, da die Werte des rechten Seiten Vektors ein extremes Oszillationsverhalten bekommen haben. Wir konnten dieses Verhalten durch eine Erhöhung der Anzahl der Iterationen kurzzeitig beheben. Einige Zeitschritte später ist dieses Verhalten allerdings erneut aufgetreten.

Gradienten-Approximation

Hier untersuchen wir den Einfluss, den die Verwendung der Gradienten-Approximationsmethode auf die Güte der Simulationsergebnisse hat.
Comet stellt zwei Verfahren zur Gradienten-Approximation zu Verfügung. Die Standardkonfiguration, mit der wir bisher alle Untersuchungen durchgeführt haben ist die sogenannte *Kleinste-Quadrat-Methode*. Hier testen wir explizit die Verwendung der alternativen Approximationsmethode, die den Satz von Gauß verwendet. Beide Verfahren haben wir bereits beschrieben.
Der Vergleich der Simulationen mit der Gradienten-Approximation mit dem Ansatz der Kleinsten-Quadrate zu der Methode mit dem Satz von Gauß, welche auch in Caffa verwendet wird, hat zwei Ergebnisse hervorgebracht. Die Ableitungswerte neigen nicht zur vollständigen Divergenz, allerdings konvergieren sie auch nicht gegen einen festen Wert, sondern zeigen ein oszillierendes Verhalten, sodass wir nicht bestimmen können, ob die Werte stimmen. Die berechneten Kraftwerte zeigen ebenfalls ein leicht oszillierendes Verhalten, sodass wir keine Werte mit Finiten-Differenzen berechnen können, die Aussagekräftig sind.

In der Abbildung 3.22 sehen wir das im vorherigen Abschnitt beschriebene oszillierende Verhalten, dass auf die Verwendung des Gauß'schen Gradienten-Approximationsansatzes zurückzuführen ist.

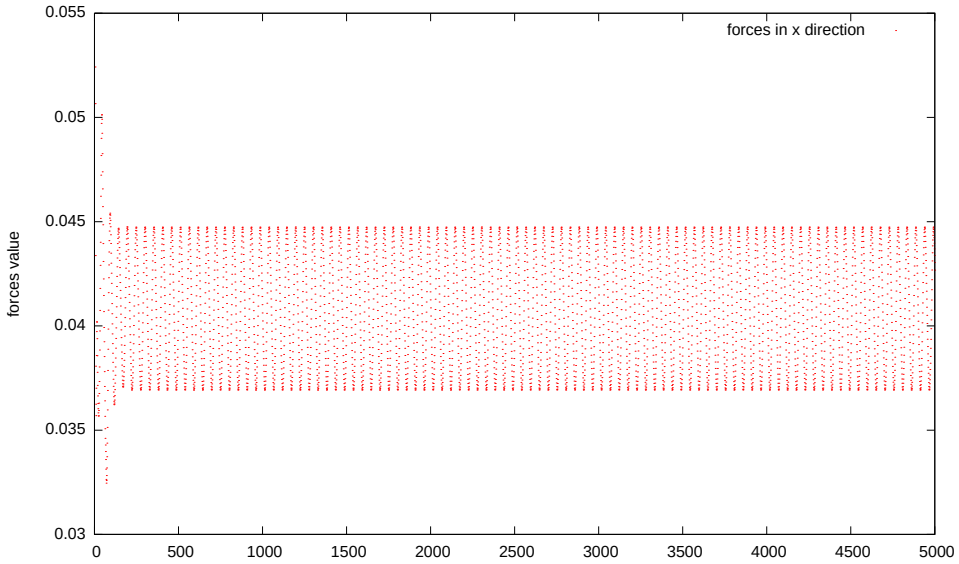

Abbildung 3.22: Wirkende Kraft auf den Tragflügel in x-Richtung.

Das beobachtete Verhalten hat uns gezeigt, dass die Verwendung dieses Gradienten-Approximationsansatzes zwar eine gewisse Verbesserung der Berechnung der Ableitungswerte bewirkt, aber die Ergebnisse der ursprünglich beobachteten Größen sind zu ungenau, um eine befriedigendes Ergebnis zu erhalten und die Ergebnisse beurteilen zu können.
Wir können mit der Wahl der Methode zur Gradientenberechnung innerhalb der Finiten-Volumen-Methode bei Comet zwar Einfluss auf das Verhalten der Ableitungsberechnung mittels dem Automatischen Differenzieren nehmen, allerdings sind bei der Methode, die auf dem Satz von Gauß basiert, die ursprünglichen Werte in unserem Fall zu instabil. Somit können wir auch mit diesem Mechanismus keine zufriedenstellenden Simulationsergebnisse erzielen.

Kontrolle der Programmstruktur

Um sicherzustellen, dass die auftretenden Probleme nicht durch einen Fehler der Programmstruktur - damit ist gemeint, dass bei der Anwendung des Automatischen Differenzierens versehentlich in Speicherbereiche geschrieben wird, die dafür nicht vorgesehen sind - verursacht werden, haben wir die Bedatung so gewählt, dass wir ein triviales Ergebnis erhalten. Unter der Bezeichnung „triviales Ergebnis“ verstehen wir die konstante Null-Lösung sowohl bei den ursprünglichen Größen, als auch bei den abgeleiteten Größen.
Dies haben wir in zwei Variationen gemacht zuerst haben wir die Null-Einströmbedingung gewählt, als zweite Variante haben wir bei einer Simulation mit einer von Null verschiedenen Bedatung die Druckwerte in jedem Iterationsschritt fix auf den Wert Null gesetzt. In beiden Fällen waren die beobachteten Ableitungswerte und auch die ursprünglichen Größen stabil.
Wurde beim Setzen der Druckwerte während der Simulation ein von Null verschiedener Wert gewählt, waren die abgeleiteten Größen einige Iterationen stabil, bis sie langsam angefangen haben, beginnend durch ein leichtes Rauschen, zu divergieren.
Somit können wir ein grundsätzliches Problem in der Programmstruktur ausschließen, da die

ursprünglichen Größen stabil sind und wir bei Bedatung mit einer trivialen Lösung diese auch erhalten und wir, wie bereits im Punkt *Lösung lineare Gleichungssysteme* erwähnt, sicherstellen können, dass die auftretenden linearen Gleichungssysteme auch für die zugehörigen Ableitungswerte korrekt gelöst werden.

Parameter-Konfiguration

Die Parameter Konfiguration umfasst eine ganze Reihe an Festlegungen, die die Simulation beschreiben.
Zum einen haben wir dort das Fluid auf die Eigenschaften der Luft festgelegt. Diese Wahl haben wir nicht variiert, da das betrachtete Profil für Strömungen in der Luft ausgelegt ist. Ebenfalls variieren wir die Temperatur nicht und halten diesen Wert bei einer Temperatur von $10°C$ fest, da eine Temperatur in dieser Größenordnung einen zu vernachlässigenden Einfluss auf das Verhalten der Strömung hat.
Als variable Größe haben wir zwei Werte genommen. Zuerst die Geschwindigkeit, da sie den größten Einfluss auf die Strömung hat. Als zweite variable Größe haben wir den Anstellwinkel variiert um auszuschließen, dass wir uns in einem Bereich des Anstellwinkels befinden, in dem das Profil ein nicht differenzierbares oder gar ein unstetiges Verhalten aufweist.
Da wir mit dem Ansatz des festen Gitters bei gleichen Parametern stabile Werte sowohl bei den ursprünglichen Größen als auch bei den Ableitungen erhalten haben und sich dieses Verhalten bei der Betrachtung in bewegten Gittern auf die ursprünglichen Größen übertragen hat, können wir ausschließen, dass die aufgetretenen Probleme nur aufgrund einer schlechten Wahl der Parameter herrühren.

Erkenntnisse

In diesem Abschnitt erläutern wir die Erkenntnisse über die Anwendbarkeit des Automatischen Differenzierens bei der Strömungssimulation mit dem Programmen Comet. Zusammenfassend können wir die Erkenntnisse der Berechnung von Ableitungswerte innerhalb von Comet mit dem Automatischen Differenzieren in zwei Punkte aufteilen.

Festes Gitter

Bei der Betrachtung einer Simulation mit einem festen Gitter ist es möglich, Ableitungswerte stabil mit dem Automatischen Differenzieren zu berechnen, da die zugrundeliegende Simulation für diese Berechnung stabil ist. Diese Simulationskonfiguration wurde bereits im Rahmen der Diplomarbeit *Automatic Differentiation in flow Simulation* [Lei07] durchgeführt und validiert. Im Rahmen dieser Arbeit haben wir diese Ergebnisse als Referenz zur Beurteilung der zusätzlichen Konfigurationen hinzugezogen.

Bewegtes Gitter

Für die Betrachtung einer Simulation mit bewegtem Gitter müssen wir leider festhalten, dass es in dieser Form bei Comet nicht möglich ist, Ableitungswerte mit dem Automatischen Differenzieren stabil zu berechnen, da aufgrund einer Vielzahl von unterschiedlichen Problemen die berechneten Ableitungswerte nicht stabil sind und somit nicht dem wahren Ableitungswert entsprechen. Allerdings konnten wir feststellen, dass dieses Phänomen nicht auf strukturelle

Probleme im Programmfluss oder der Bedatung der Strömungsgrößen basiert, sondern auf - sich verstärkende - numerische Problemen beruht. Um diese Probleme zu beheben wäre es notwendig Comet, in weiten Bereichen neu zu schreiben bzw. zu entwickeln um sicher zu stellen, dass die Probleme behoben sind. Hierfür erwähnen wir exemplarisch die Bewegung der Gittergeometrie.

3.4 Ergebnisse

Die Beurteilung der Simulationsergebnisse fällt zweigeteilt aus. Zum einen können wir die Aussage treffen, dass die Anwendung des Automatischen Differenzierens grundsätzlich im Bereich der Strömungssimulation mit Finiten-Volumen möglich ist, wie wir aus den Ergebnissen der Testreihe mit dem Programm Caffa gesehen haben. Allerdings ist die Anwendung des Automatischen Differenzierens bei Comet mit bewegten Gittern fehlgeschlagen, da massiv numerische Probleme aufgetreten sind.
Im Folgenden betrachten wir die Ergebnisse der beiden Programme nochmals etwas differenzierter.

3.4.1 Caffa

Die Beurteilung der Simulation mit Automatischen Differenzieren bei Caffa fällt durchweg positiv aus.

Wir haben, aufgrund der homogenen Programmstruktur eine sehr einfache Anwendbarkeit des Automatischen Differenzierens mit dem Quellcode-Transformations-Tool Tapenade.
Die innere Programmstruktur - im Speziellen, dass die Gitterbewegung vollständig in dem Quellcode von Caffa realisiert ist - vereinfacht die Anwendung des Automatischen Differenzierens bei bewegten Gittern. Da die Zellen nicht nur durch die Bewegung des Randes transformiert werden, sondern das Gitter noch geglättet wird und dies für die Berechnung der Ableitung berücksichtigt wird, hat das Gitter weiterhin gute Eigenschaften.
Dadurch ist gewährleistet, dass durch die Drehung die Konvergenz der ursprünglichen Größen im Vergleich zur Simulation ohne AD nicht beeinflusst wird und alle Einflüsse der Gitterbewegung im abgeleiteten Code berücksichtigt werden. Dadurch ist ein vollständiger und transparenter Informationsfluss gewährleistet.

In den Abbildungen 3.23 und 3.24, die im Rahmen der Diplomarbeit *Automatic Differentiation in flow simulation* [Lei07] erstellt wurden, sehen wir die guten Simulationsergebnisse von Caffa.

In der Abbildung 3.23 ist die Auftriebskraft, die auf den Tragflügel in Abhängigkeit des Anstellwinkels wirkt dargestellt.

In Abbildung 3.24 sehen wir Ableitungswerte, die mit dem Automatischen Differenzieren berechnet wurden im Vergleich zu den berechneten Finiten Differenzen. Wir sehen direkt, dass die mit dem Automatischen Differenzieren berechneten Ableitungswerte realistisch sind und im Vergleich zu den den Finiten-Differenzen-Werten einen glättenden Effekt besitzen.

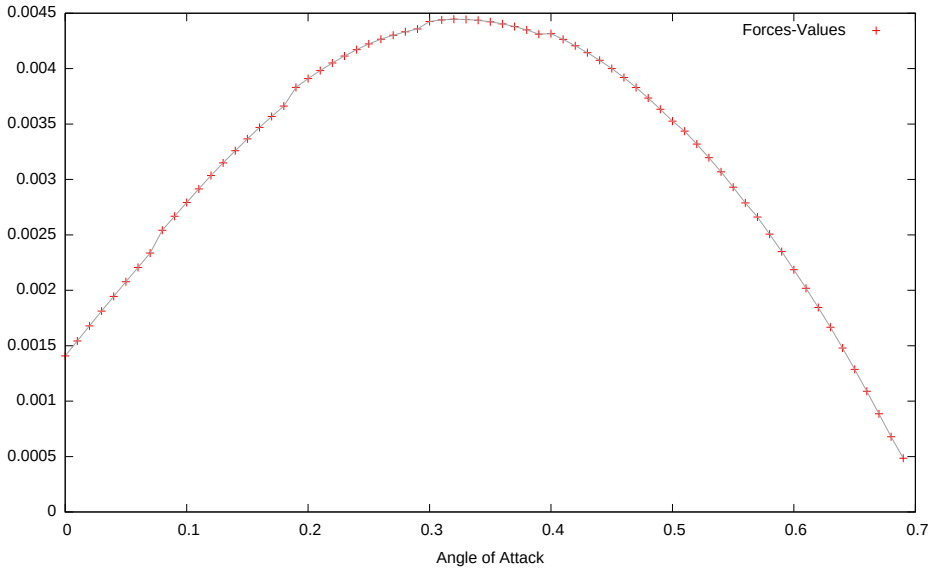

Abbildung 3.23: Auftriebskraft am Tragflügel.

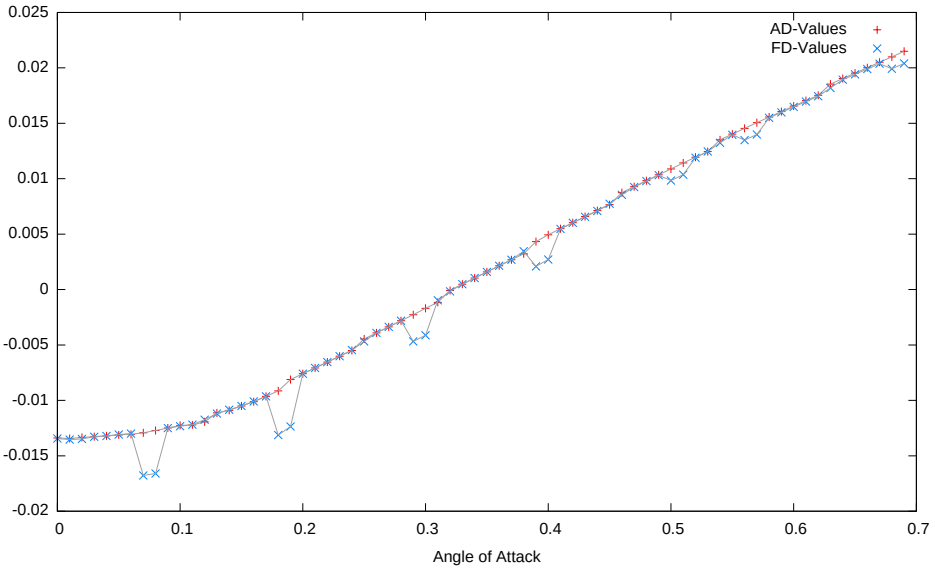

Abbildung 3.24: Ableitungswerte mittels AD und Finiten-Differenzen.

3.4.2 Comet

Bei Comet haben wir keine so positive Ergebnisbilanz wie bei Caffa. Für die abschließende Beurteilung der Ergebnisse beziehen wir auch Erkenntnisse und Untersuchungen mit ein, die im Rahmen der Diplomarbeit [Lei07] gewonnen wurden, da sie nach den weiteren Untersuchungen im Rahmen dieser Arbeit in einem neuen Licht erscheinen.
Wir unterteilen die Erkenntnisse in zwei Bereiche. Zuerst fassen wir zusammen, welche Ansät-

ze funktioniert haben und welche Ansätze nicht funktioniert haben. Anschließend beschäftigen wir uns mit den Gründen für die Ergebnisse.

Festes Gitter

Diese Konfiguration hatten wir ja schon, wie oben beschrieben, in der Arbeit [Lei07] untersucht. Zum jetzigen Zeitpunkt können wir die Ergebnisse, die wir damals gewonnen haben, besser und vollständiger beurteilen.

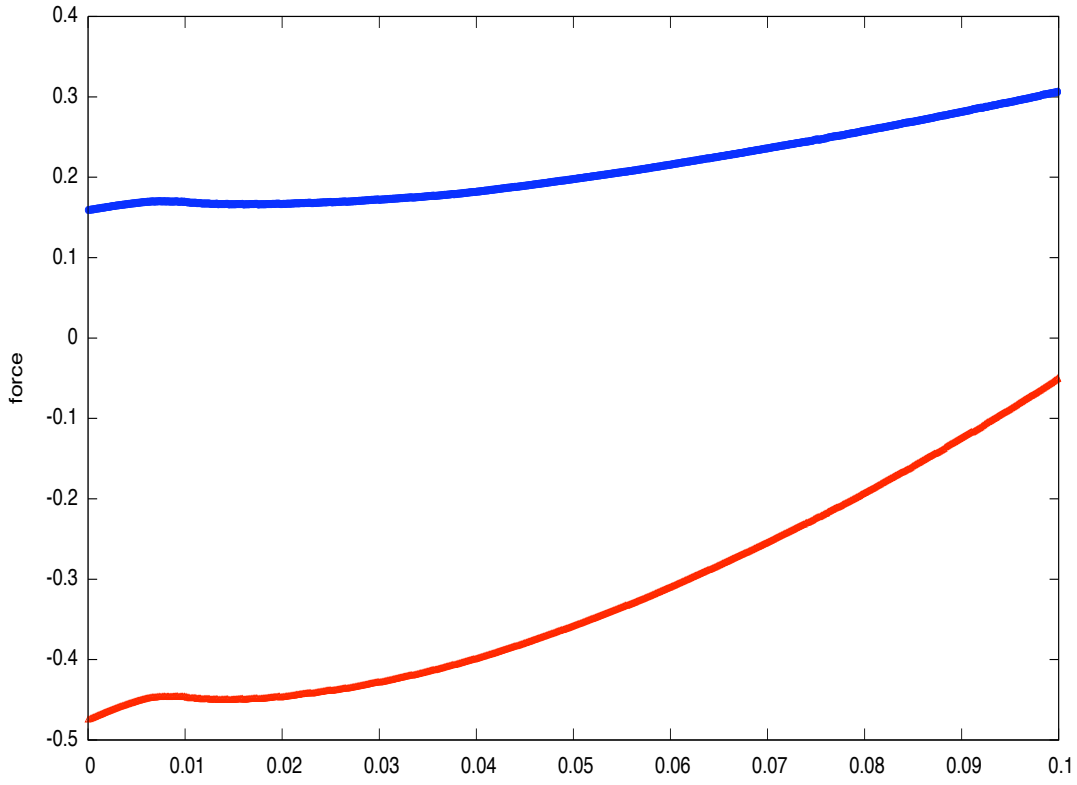

Abbildung 3.25: Kräfte in x- und y-Richtung, die auf den Tragflügel in Abhängigkeit des Anströmwinkels wirken.

Wie wir in den Abbildungen 3.25 und 3.26 sehen, haben wir hier - wie bei der Simulation mit Caffa - gute Simulationsergebnisse, sowohl für die Auftriebskraft als auch für die Ableitungswerte. Allerdings wissen wir nach umfangreicheren Untersuchungen, dass diese guten Simulationsergebnisse durch die verwendeten Algorithmen zur Bewegung des Gitter zunichte gemacht werden und keine grundsätzlichen guten Eigenschaften von Comet für dieses Verhalten verantwortlich sind. Diese Programmstruktur ist auch bei der Berechnung der Ableitungswerte stabil, da Comet hochoptimiert auf Stabilität und Geschwindigkeit ist und wir nicht in die inneren Strukturen und Mechanismen eingreifen müssen.

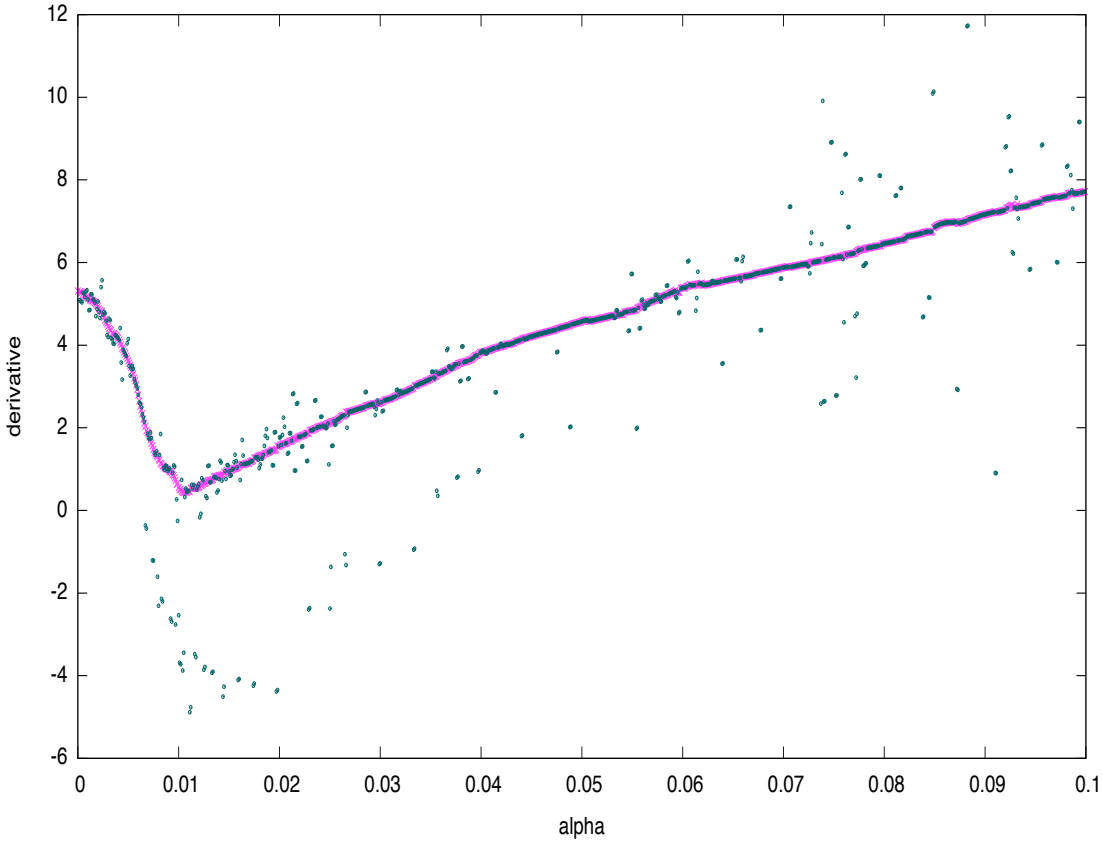

Abbildung 3.26: Ableitungswerte in y-Richtung mittels AD und Finiten-Differenzen.

Bewegtes Gitter

Bei der Untersuchung von Simulationen mit einem bewegten Gitter können wir festhalten, dass diese Konfigurationen bei Comet in dieser Form nicht möglich sind. Dies basiert auf einer Reihe von unterschiedlichen Ereignissen. Wir können allerdings mit Sicherheit festhalten, dass keine grundlegenden Fehler bei der Anwendung vom AD bei Comet geschehen sind. Dies haben wir in den vorher beschriebenen Untersuchungen ausgeschlossen.

Gründe

Wie wir schon im vorherigen Abschnitt beschrieben haben, können wir nicht einen Grund alleine als Grund für das Scheitern der Berechnung von Ableitungswerten mit dem Automatischen Differenzieren identifizieren. Als zusammenfassende Erkenntnis können wir festhalten, dass die Bewegung der Gitter, wie sie in Comet realisiert ist, zu Instabilitäten bei der Berechnung der zugehörigen differenzierten Größen führt. Auf dieses Verhalten können wir nicht ohne die Struktur von Comet grundlegend zu ändern, Einfluss nehmen. Um die Gitterbewegung überhaupt für das Automatische Differenzieren erfassbar machen zu können, müssen wir schon etwas in die innere Struktur von Comet eingreifen da die Gittebewegung normalerweise über

einen externen Mechanismus mit einer Metasprache realisiert ist. Dadurch können keine stabilisierenden Algorithmen auf die Gitterbewegung wirken. Zudem ist die verwendete Methode zur Gradienten-Berechnung in der Finiten-Volumen-Simulation zwar, was die Gitterbewegung betrifft, sehr stabil, allerdings nicht für die Berechnung der Ableitungswerte.

4 Finanzmathematik

In diesem Kapitel betrachten wir finanzmathematische Simulationen zur Berechnung von Optionspreisen. Das Kapitel hat folgenden Aufbau: Zuerst betrachten wir einige elementare Grundlagen der Finanzmathematik und bauen auf diesen Grundlagen anschließend das Black-Scholes und das Heston-Modell auf. Zuletzt wenden wir auf diese Modelle das Automatische Differenzieren an und diskutieren die Resultate.

4.1 Grundlagen

In diesem Abschnitt befassen wir uns mit den Optionen und der Integration.

4.1.1 Optionen

In diesem Abschnitt stellen wir die wesentlichen Arten und Eigenschaften von finanzmathematischen Optionen vor. Wir beginnen mit einer allgemeinen Definition einer Option, bevor wir uns mit der Klassifizierung befassen. Der folgende Abschnitt basiert auf den Vorlesungsskripten *Financial Mathematics I* von Kiesel [Kie04], und *Numerische Methoden der Finanzmathematik* von Grüne [Grü09].

Definition 4.1.1 (Option)
Unter einer Option im Finanzbereich versteht man, dass der Käufer der Option ein Recht kauft, ein gewisses Produkt zu vorher vereinbarten Konditionen zu kaufen beziehungsweise zu verkaufen. Dies nennt man das Recht, die Option auszuüben. Allerdings hat der Käufer nicht die Pflicht die Option auszuüben.

Im Allgemeinen handelt es sich um das Recht eine Aktie, Rohstoffe oder Währungen zu einem bestimmten Kurs oder Preis zu kaufen beziehungsweise zu verkaufen. Die Option ist damit als Instrument zur Absicherung gegen Kursschwankungen gedacht.

Optionen können wir nach zwei Kriterien unterscheiden. Zum einen nach der Art des Ausübungsrechtes, das heißt, ob wir das Recht haben etwas zu kaufen (Call[1]) oder zu Verkaufen (Put[2]). Das andere Kriterium ist die Fälligkeit, wann wir die Option ausüben dürfen. Übliche Formen sind die Europäische-Option und die Amerikanische-Option, seltener ist die Bermuda-Option. Weitere Optionsarten sind zum Beispiel die Asiatische-Option oder die Barrier-Option. Die genannten Varianten gibt es sowohl als Put-Optionen, wie auch als Call-Optionen.
Wir befassen uns nun etwas ausführlicher mit den unterschiedlichen Typen an Optionen, bevor wir uns mit der Frage, wie man den Preis einer Option berechnet, befassen.

[1] Eine Call-Option wird auch Kaufoption genannt, allerdings hat sich die englische Bezeichnung auch im deutschen Sprachgebrauch durchgesetzt.
[2] Die deutsche Bezeichnung ist Verkaufsoption, auch hier ist die englische Bezeichnung, Put-Option, üblich.

Call-Option

Nun betrachten wir die Call-Option detailliert. Zunächst analysieren wir, welche Einflussfaktoren in eine Call-Option einfließen. Dazu betrachten wir zuerst die Definition einer Call-Option.

Definition 4.1.2 (Call-Option)
Die Call-Option gibt dem Käufer der Option das Recht, das vereinbarte Produkt zu einem oder mehreren festgelegten Zeitpunkten, beziehungsweise in einem Zeitfenster zu einem vorher vereinbarten Preis zu **kaufen**.

Aus der Definition ergibt sich direkt, dass der aktuelle Marktpreis des Produktes und der vereinbarte Kaufpreis des Produktes den Wert der Option beeinflussen.

Bemerkung 4.1.3 (Eigenschaften der Call-Option)
Die Auszahlungsfunktion einer Call-Option hat folgende Charakteristik: Ist der Preis des Produktes zum Ausübungszeitpunkt kleiner oder gleich dem vereinbarten Preis, ist der Wert der Option Null. Für einen Preis der größer dem vereinbarten Preis ist, ist damit der Wert gleich der Differenz zwischen dem vereinbarten Preis und dem tatsächlichen Preis. Die grafische Darstellung dieses Verhaltens sehen wir in der Grafik 4.1.

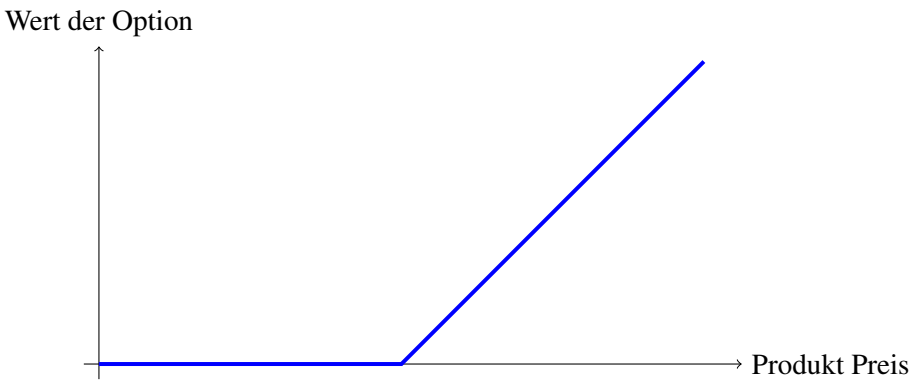

Abbildung 4.1: Auszahlungsfunktion einer Call-Option.

Damit ergibt sich folgender Ausdruck für den Wert der Call-Option:

$$c = \max(0, S - S_v), \tag{4.1}$$

wobei S_v den vereinbarten Kaufpreis und S den Preis beim Ausübungszeitpunkt bezeichnet.

Put-Option

Nun betrachten wir die Put-Option, die deutsche Bezeichnung ist Verkaufs-Option.

Definition 4.1.4 (Put-Option)
Die Put-Option gibt dem Käufer der Option das Recht, das vereinbarte Produkt zu einem oder mehreren festgelegten Zeitpunkten beziehungsweise in einem Zeitfenster zu einem vorher vereinbarten Preis zu **verkaufen**.

Analog zur Call-Option ergibt sich der Wert der Put-Option auch aus dem aktuellen Preis des Produktes und dem vereinbarten Verkaufspreis des Produktes.

Bemerkung 4.1.5 (Eigenschaften der Put-Option)
Die Auszahlungsfunktion einer Put-Option hat folgende Charakteristik: Ist der Preis des Produktes zum Ausübungszeitpunkt größer oder gleich dem vereinbarten Preis. so ist der Wert der Option Null. Für einen Preis der kleiner dem vereinbarten Preis ist, ist damit der Wert gleich der Differenz zwischen dem vereinbarten Preis und dem tatsächlichen Preis. Die grafische Darstellung dieses Verhaltens sehen wir in der Grafik 4.2.

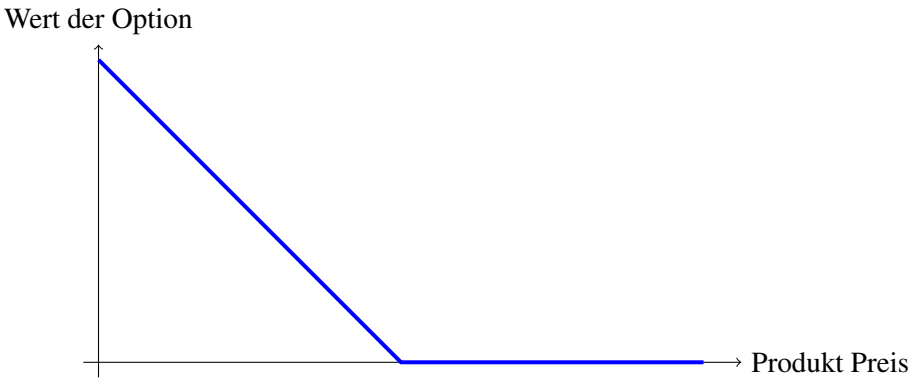

Abbildung 4.2: Auszahlungsfunktion einer Put-Option.

Damit ergibt sich folgender Ausdruck für den Wert der Call-Option:

$$p = \max(0, S_v - S), \tag{4.2}$$

wobei S_v den vereinbarten Verkaufspreis und S den Preis beim Ausübungszeitpunkt bezeichnet. Im Folgenden betrachten wir Optionsarten, die sich im Ausübungszeitpunkt unterscheiden.

Europäische Option

Wir betrachten nun eine Call-Option. Die Aussagen sind analog auf die Put-Option übertragbar. Diese Einschränkung treffen wir nur um die Notation einfacher zu halten.

Definition 4.1.6 (Europäische-Option)
Bei einer Europäischen-Option erwirbt der Käufer zum Zeitpunkt $t = 0$ das Recht, die Option zum vorher vereinbarten Endzeitpunkt $t = T$ auszuüben.

Diese Art der Option ist sinnvoll, wenn man sich gegen Kursschwankungen zu einem festen Zeitpunkt absichern möchte. Als Beispiel könnte man folgende Situation betrachten: Wenn man weiß, dass man zum Begin des Winters seinen Heizöltank füllen muss, könnte man beispielsweise rechtzeitig einen Höchstbetrag vereinbaren.

Bemerkung 4.1.7 (Eigenschaften der Europäischen-Option)
Durch den fest vereinbarten Ausübungszeitpunkt ist die Betrachtung im Vergleich zur

Amerikanischen- oder Bermuda-Option einfach zu berechnen, da nur ein fester Zeitpunkt betrachtet werden muss.
Für den tatsächlichen Wert der Option spielt nur der Endzeitpunkt eine Rolle.

Eine Option mit flexiblmn Ausübungszeitpunkt betrachten wir nun im Anschluss mit der Amerikanischen-Option.

Amerikanische-Option

Analog zur Europäischen-Option beschränken wir uns hier auf die Betrachtung einer Call-Option.

Definition 4.1.8 (Amerikanische-Option)
Bei einer Amerikanischen Option erwirbt der Käufer das Recht, die Option im Zeitintervall $I_T := [0, T]$ auszuüben. Damit steht der Ausübungszeitpunkt beim Kauf der Option noch nicht fest. Lediglich der späteste Ausübungszeitpunkt ist damit festgelegt.

Bemerkung 4.1.9 (Eigenschaften der Amerikanischen-Option)
Die komplette Preisentwicklung im Intervall I_T beeinflusst den Wert der Option. Damit ist die Berechnung des Wertes einer Amerikanischen-Option um einiges komplexer als die Berechnung des Wertes einer Europäischen-Option.
Man kann die Berechnung auf ein Hindernisproblem, wie zum Beispiel in Numerische Lösung des parabolischen Hindernisproblems mit einer inneren Punkte-Methode von Wick [Wic00] beschrieben, zurückführen.

Zuletzt betrachten wir noch einige sogenannte *exotische* Optionen.

Andere Optionen

In diesem Abschnitt geben wir einen kurze Überblick über andere verbreitete Arten von Optionen, sowohl was den Ausübungszeitpunkt als auch die Auszahlungsfunktion betrifft.

Bermuda-Option

Die Bermuda-Option ist eine Zwischenform aus Amerikanischer- und Europäischer-Option, woher auch der Bergriff Bermuda-Option kommt, da die Bermudas zwischen Amerika und Europa liegen. Bei der Bermuda-Option darf der Käufer zu mehreren fixen Zeitpunkten sein Optionsrecht ausüben. Damit gibt es mehr mögliche Ausübungszeitpunkte als bei der Europäischen-Option und weniger als bei der Amerikanischen-Option.

Asiatische-Option

Anders als es die *Kontinentale*-Bezeichnung vermuten lässt, unterscheidet sich die Asiatische-Option nicht im Ausübungszeitpunkt von den bisherigen Optionen.
Die Asiatische-Option betrachtet nicht den Wert eines Produktes zu einem Zeitpunkt sondern den Mittelwert des Produktes über die gesamte Laufzeit der Option. Man unterscheidet innerhalb der Asiatischen-Optionen noch die Art der Durchschnittsbildung. Hierzu seien die Arithmetische-Asiatische-Option und die Geometrische-Asiatische-Option erwähnt.

Barrier-Option

Hier betrachtet man zwei grundlegend unterschiedliche Varianten:
Die *Knock-Out-Option* ist eine Option, die verfällt, wenn der Preis des Produktes innerhalb der Laufzeit eine gewisse Schranke überschreitet.

Die *Knock-In-Option* ist eine Option, die erst aktiv wird, wenn der Preis des Produktes innerhalb der Laufzeit eine gewisse Schranke überschreitet. Ansonsten verfällt die Option.

Weitere Exotische-Optionen, auf die wir nicht weiter eingehen werden: sind: Chooser-Optionen, Cliquets-Optionen, Lookback-Optionen, Range-Optionen oder Russische-Optionen.

Einführungen in die Theorie der Optionen bieten folgende Quellen: *Vorlesungsskript Finanzmathematik* von Frey und Schmidt [FS05], *Financial Mathematics I* von Kiesel [Kie04] oder auch *Numerische Methoden der Finanzmathematik* von Lars Grüne [Grü09].

4.1.2 Integration

In diesem Abschnitt befassen wir uns mit einigen numerischen Integrationsverfahren, die in der Finanzmathematik Anwendung finden. Als erstes betrachten wir das wohl bekannteste Verfahren innerhalb der Finanzmathematik: die Monte-Carlo-Simulation. Anschließend betrachten wir den darauf basierenden Algorithmus der Quasi-Monte-Carlo-Simulation. Zuletzt betrachten wir noch drei Quadratur-Regeln, die nicht auf einem der ersten beiden Verfahren basieren: Die Gauß-Quadratur[3], die Laguerre-Quadratur[4] und die Kronrod-Quadratur[5].

Monte-Carlo-Simulation

Monte-Carlo-Methoden sind Algorithmen, die auf der sehr häufigen Wiederholung von Zufallsexperimenten beruhen. Die grundlegende Idee dieser Verfahrensklasse geht auf Enrico Fermi[6] zurück, während die ersten Arbeiten auf Stanislaw Ulam[7] und John von Neumann[8] zurückgehen. Hierzu sei folgende Arbeit genannt *Statistical Methods in Neutron Diffusion* [NR47]. Die Geschichte der Entwicklung von Monte-Carlo-Methoden findet man im Artikel *THE BEGINNING of the MONTE CARLO METHOD* von N. Metropolis [Met87]. Die Grundzüge der Monte-Carlo Methoden wurden im *Los Alamos National Laboratory* in den 1940er Jahren im Rahmen des Manhattan-Project entwickelt.
Hier beschränken wir uns mit der Betrachtung der Monte-Carlo-Simulation zur Berechnung von Integralwerten. Im folgenden sei

$$I_\Omega(f) := \int_\Omega f(x)dx \tag{4.3}$$

der exakte Wert des Integrals. Für die Gleichung (4.3) gilt, dass $f(x)$ eine integrierbare Funktion auf dem Gebiet Ω ist, für die gilt $f : \Omega_0 \to \mathbb{R}$. Weiter ist $\Omega \subset \Omega_0 \subset \mathbb{R}^d$, $d \in \mathbb{N}$.

[3] Benannt nach dem deutschen Mathematiker Johann Carl Friedrich Gauß (1777-1855).
[4] Benannt nach dem französischen Mathematiker Edmond Nicolas Laguerre (1834-1886).
[5] Benannt nach dem russischen Mathematiker Aleksandr Semenovich Kronrod (1921-1986).
[6] 1901-1954 Kernphysiker
[7] 1909-1984 Mathematiker
[8] 1903-1957 Mathematiker

Den Wert der numerischen Integration beschreiben wir mit $Q_{N,\Omega}(f)$, wobei N die Feinheit der Approximation angibt. Für die Monte-Carlo-Simulation ergibt sich damit folgender Ausdruck:

$$Q_{N,\Omega}(f) := \frac{|\Omega|}{N} \sum_{n=1}^{N} f(x_i). \tag{4.4}$$

Dass für $Q_N(f)$ aus Gleichung (4.4) der Zusammenhang:

$$Q_{N,\Omega}(f) \overset{N\to\infty}{\longrightarrow} I_\Omega(f), \tag{4.5}$$

folgt, müssen folgende Voraussetzungen gelten:

$$\begin{aligned} &x_i \text{ ist eine gleichverteilte Folge, } x_i \in \Omega \text{ und} \\ &f \in \text{ der Riemann-integrierbaren Funktionen.} \end{aligned} \tag{4.6}$$

Die obige Aussage können wir in einem Satz zusammenfassen:

Satz 4.1.10 (Konvergenz der Monte-Carlo-Simulation)
Gelten die Bedingungen aus den Gleichungen (4.6)*, dann gilt, dass $Q_{N,\Omega}$ aus der Gleichung* (4.4) *die Konvergenzeigenschaft von Gleichung* (4.5) *erfüllt.*

Nun betrachten wir ein Beispiel zur Monte-Carlo-Simulation:

Beispiel 4.1.11 (Monte-Carlo-Quadratur)
In diesem Beispiel betrachten wir zunächst folgende Funktion:

$$f(x) := -3x^2 + 2, \tag{4.7}$$

auf dem Integrations Interval $\Omega := [-1, 1]$. Die Stammfunktion der Funktion $f(x)$ aus Gleichung (4.7) *ist*

$$F(x) = -x^3 + 2 \cdot x. \tag{4.8}$$

Für das Integral ergibt sich mit der Stammfunktion (4.8)*,*

$$I_[\Omega] = \int_\Omega f(x)dx = 2. \tag{4.9}$$

Wir berechnen nun den Integralwert zusätzlich mit der Monte-Carlo-Simulation. Dazu verwenden wir ein C++-Programm, welches wir auf der CD im Verzeichnis `Programme/ex:MonteCarlo01` *finden.*

$$Q_{\Omega,N} := \frac{|\Omega|}{1} \sum_{n=1} N f(x_n). \tag{4.10}$$

Die Punkte x_n sind gleichverteilte Pseudo-Zufallszahlen auf dem Intervall $[-1, 1]$, die wir mit der C++-Funktion `rand()` *erzeugt haben.*
Als Ergebnis betrachten wir die Differenz zwischen dem exakten Ergebnis und dem Simulations-Ergebnis $Q_{\Omega,N}$.
Die Funktion $f(x)$ aus Gleichung (4.7) *auf dem Intervall $\Omega = [-1, 1]$ sehen wir in der Abbildung 4.3.*

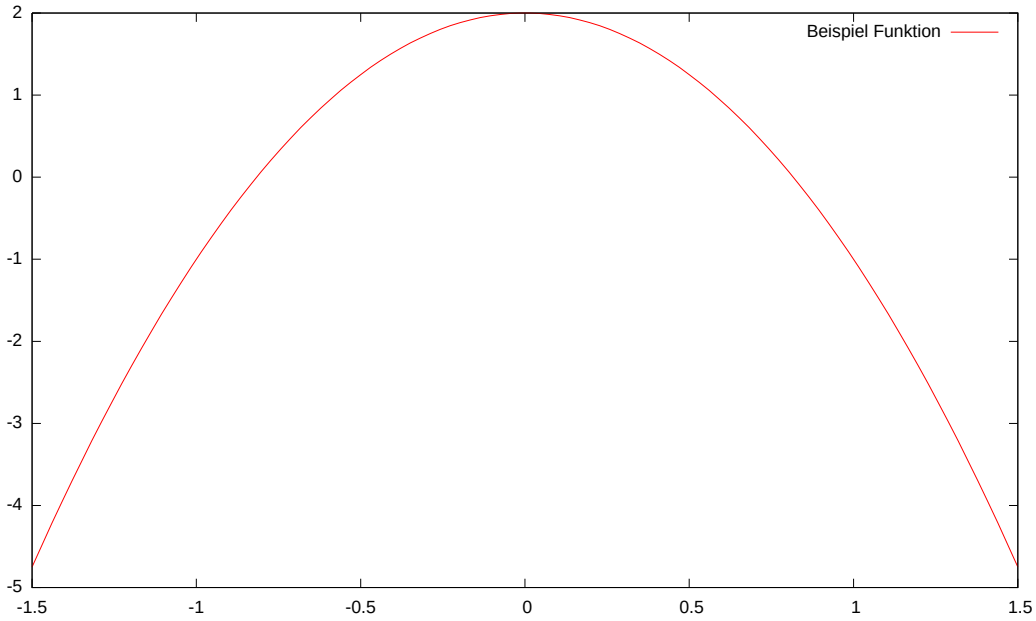

Abbildung 4.3: Verlauf der Beispielfunktion.

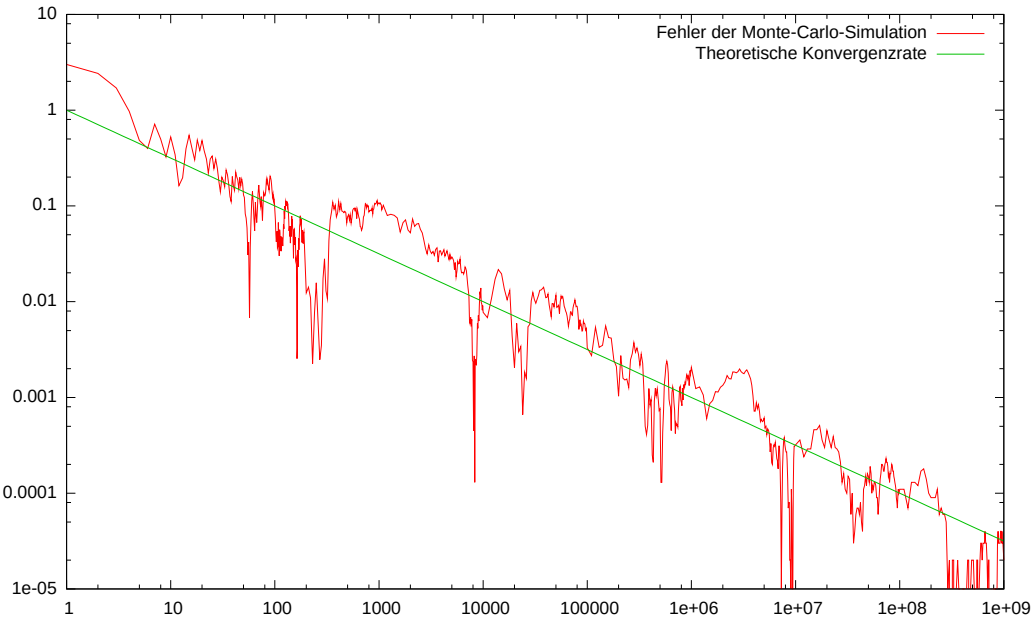

Abbildung 4.4: Vergleich der gemessenen Konvergenz und dem theoretischen Konvergenzverlaufs.

In der Abbildung 4.4 sehen wir den Absolutbetrag des Fehlers aus dem Simulationsergebnis und dem theoretischen Wert, im Vergleich zu der theoretischen Konvergenzrate der Monte-

Carlo-Simulation im Doppel-Logarithmischen Plot.
Wir sehen, dass die Konvergenzrate der Monte-Carlo-Simulation nicht allzu gut ist. Als nächstes betrachten wir eine Funktion, an der man erkennt, weshalb die Monte-Carlo-Simulation im Bereich Finanzmathematik so weit verbreitet ist.
Die nun betrachte Funktion können wir als mehrdimensionale Erweiterung der bereits betrachteten Funktion (4.7) *auffassen:*

$$g(\mathbf{x}) := (\sum_{i=0}^{d} -3x_i^2) + 2, \tag{4.11}$$

wobei für $\mathbf{x}$ *gilt, dass* $\mathbf{x} \in \mathbb{R}^d$ *ist.*

Die Konvergenzordnung für die Monte-Carlo-Simulation betrachten wir im nächsten Satz:

Satz 4.1.12 (Konvergenzordnung Monte-Carlo-Simulation)
Für eine Funktion f*, die nach dem Satz 4.1.10 konvergiert, gilt zudem folgende Aussage zur Konvergenzordnung:*

$$\lim_{n\to\infty} P(\frac{\sigma_f}{\sqrt{n}}a < R_n(f) < \frac{\sigma_f}{\sqrt{n}}b) = \frac{1}{\sqrt{2\pi}} \int_a^b e^{-\frac{t^2}{2}}\, dt. \tag{4.12}$$

Damit gilt für die Monte-Carlo-Simulation, dass sie die asymtotische Konvergenzordnung $\frac{1}{\sqrt{n}}$ *besitzt.*

Beweis. Die Aussage folgt direkt aus dem *starken Gesetz der großen Zahlen*. Einen ausführlichen Beweis dieser Aussage findet man in Einführung in die Wahrscheinlichkeitstheorie von Andreas Eberle [Ebe10]. □

Die Konvergenzgeschwindigkeit ist mit der Ordnung $\sqrt{N}$ nicht allzu gut, allerdings besitzt die Monte-Carlo-Simulation die positive Eigenschaft, dass die Konvergenzgeschwindigkeit unabhängig von der Dimension des Integrals ist. Somit gilt der sogenannte *Fluch der Dimensionen (engl. curse of dimensionality)* nicht für die Anwendung der Monte-Carlo-Simulation.

Bemerkung 4.1.13 (Fluch der Dimensionen)
Es gibt noch andere Möglichkeiten, den Fluch der Dimensionen zu kontrollieren, die wir im Rahmen dieser Arbeit nicht näher betrachten möchten. Der Vollständigkeit halber seien hierzu die Dünngittermethoden erwähnt. Die Quasi-Monte-Carlo-Methoden werden wir im nächsten Abschnitt betrachten.

Im Folgenden kombinieren wir die Monte-Carlo-Simulation mit dem Automatischen Differenzieren. Dazu betrachten wir folgendes Beispiel.

Ableitung bezüglich der Parameter

Das Interesse an der Sensitivität des Wertes eines Integrals bezüglich der Funktionsparameter ist eine zentrale Frage innerhalb der Finanzmathematik. Im *Black-Scholes-Modell* wird die Sensitivität bezüglich der Parameter mit die *Griechen* bezeichnet. Auf diese Aspekte gehen wir

im Abschnitt 4.2.2 näher ein.
Hier beschränken wir uns auf die Betrachtung eines einfachen ein-dimensionalen Problems. Wir betrachten eine Funktion deren Stammfunktion wir kennen.

Beispiel 4.1.14
Wir betrachten die Funktion

$$f(x) := -ax^2 + 1, \tag{4.13}$$

für x gilt $x \in \mathbb{R}$, und approximieren den Wert des Integrals auf dem Intervall $I := [-1, 1]$ mit der Monte-Carlo-Simulation. Simultan dazu möchten wir die Empfindlichkeit des Integralwerts bezüglich des Parameters a ermitteln.
Zunächst bestimmen wir beide Größen auf analytischem Wege. Die Stammfunktion von $f(x)$ ist:

$$F(x) = -a/3\, x^3 + x \tag{4.14}$$

und für die Sensitivität bezüglich a gilt:

$$\frac{dF}{da}(x) = -\frac{1}{3}x^3 \tag{4.15}$$

Für das Integral auf dem Intervall I ergibt sich damit eine Sensitivität vom Wert $-2/3$.

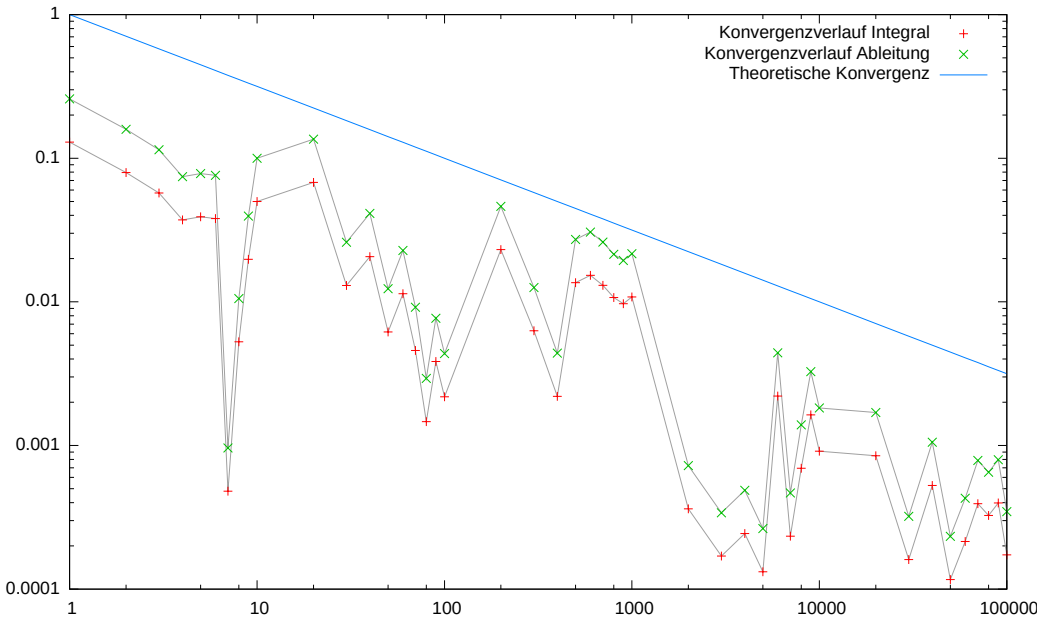

Abbildung 4.5: Vergleich der gemessenen Konvergenz und dem theoretischen Konvergenzverlauf, für das Integral und die Ableitung des Integrals bezüglich a.

In der Grafik 4.5 sehen wir, dass die Konvergenz der Ableitung des Integrals bezüglich des Parameters direkt mit der Konvergenz des Integrals korreliert ist.

Quasi-Monte-Carlo-Simulation

Das Quasi-Monte-Carlo-Verfahren ist eine Modifikation des Monte-Carlo-Verfahrens in dem man keine Zufallszahlen verwendet, sondern *Quasi-Zufallszahlen* mit besseren Eigenschaften. Dadurch erhält man eine besser Konvergenzrate Statt $\frac{1}{\sqrt{n}}$ erhält man $\frac{1}{n}$. Allerdings ist diese Konvergenzrate nicht vollständig unabhängig von der Dimension des Integrationsproblems.
Auf die Quasi-Monte-Carlo-Simulation wollen wir hier gar nicht näher eingehen, sondern verweisen beispielsweise auf *Monte Carlo and quasi-Monte Carlo methods* von Caflisch [Caf98] oder *Numerische Methoden in der Optionspreistheorie: Monte Carlo und Quasi-Monte Carlo Methoden* von Leippold [Lei97].

Gauß-Quadratur

Das Gauß-Quadratur-Verfahren[9] ist ein numerisches Integrationsverfahren, bei dem die Stützstellen und die Gewichte so gewählt werden, dass der Grad der Approximation maximal wird. Der maximale Approximationsgrad bei einer vorgegebenen Anzahl von n Stützstellen ist $2n-1$, da wir die Stützstellen x_i und die Gewichte α_i frei wählen können und wir somit $2n$ Freiheitsgrade haben. Damit ergibt sich, dass wir ein eindeutig bestimmtes Interpolationspolynom vom Grade $n-1$ erhalten können. Für das Polynom gilt, dass wir es mit n Stützstellen exakt integrieren können.
Formal können wir diese Aussage in folgendem Satz ausdrücken:

Satz 4.1.15 (Gauß-Quadratur)
Sei $f \in \mathcal{C}^{2n+2}$, auf dem Intervall $I := [a,b]$. Ohne Beschränkung der Allgemeinheit betrachten wir im Folgenden das Intervall $I := [-1,1]$, ansonsten betrachten wir eine Variablentransformation, sodass wir das gegebene Intervall auf $[-1,1]$ abbilden. Dann existieren Stützstellen $x_1, x_2, \ldots, x_{n+1} \in I$ und dazu gehörige Gewichte $\alpha_1, \alpha_2, \ldots, \alpha_{n+1}$, so dass

$$\int_{-1} 1 f(x) dx = 2 \sum_{i=1} n + 1 \alpha_i f(x_i) + R(f), \tag{4.16}$$

mit $R(p) = 0$ für alle $p \in \mathcal{P}_{2n+1}$ ist. Weiter gilt für das Restglied aus Gleichung (4.16)*:*

$$|R(f)| = \frac{[(n+1)!]^4}{[(n+1)!]^3(2n+2)} 2^{2n+2} |f^(2n+2)(\chi)| \tag{4.17}$$

für eine Zwischenstelle $\chi \in (-1,1)$.

Eine ausführliche Betrachtung der Herleitung finden wir zum Beispiel in *Numerische Mathematik 1* von Stoer [Sto02] oder im Vorlesungsskript *Numerik 1b* von Urban [Urb06].
In den folgenden drei Abschnitten betrachten wir drei gebräuchliche Varianten zur Konstruktion der Stützstellen und der zugehörigen Gewichte.

Tschebyscheff-Quadratur

Als Stützstellen für die Gauß-Quadratur auf dem Intervall $[-1,1]$ verwenden wir im Folgenden die Nullstellen der Tschebyscheff-Polynome.

[9] Benannt nach dem Mathematiker Carl Friedrich Gauß (1777-1855).

Diese Variante ist auch unter *Gauß-Tschebyscheff-Quadratur* oder einfach nur unter dem Namen *Gauß-Quadratur* bekannt. Eine gebräuchliche Darstellung der Tschebyscheff-Polynome ist:

$$T_n(x) = \cos(n \arccos x) \tag{4.18}$$

für $x \in [-1, 1]$ und

$$T_n(x) = \cosh(n \operatorname{arccosh} x) \tag{4.19}$$

für $|x| > 1$. Die zur Gleichung (4.18) gehörigen Nullstellen $x_{i,n}$ lassen sich über

$$x_{i,n} := \cos(\frac{2i-1}{2n}\pi), \tag{4.20}$$

berechnen. Die entsprechende Gewichtsfunktion lautet:

$$\omega(x) = \frac{1}{\sqrt{1-x^2}}, \tag{4.21}$$

damit ergibt sich für die Gewichte folgender Ausdruck:

$$\alpha_{i,n} := \frac{\pi}{n}. \tag{4.22}$$

Damit können wir in folgender Tabelle 4.1 die Stützstellen und Gewichte für die Tschebyscheff-Quadratur angeben:

Beispiel 4.1.16 (Gauß-Quadratur)
Als Beispielfunktion betrachten wir das Integral der Funktion:

$$f(x) := 2 \cos 2x e^{\sin 2x} \tag{4.23}$$

auf dem Interval $\Omega := [-1, 1]$. *Die Stammfunktion der Funktion* (4.23) *ist:*

$$F(x) = e^{\sin 2x}. \tag{4.24}$$

Damit ergibt sich für das Integral

$$I = \int_\Omega f(x)dx, \tag{4.25}$$

aus der Stammfunktion (4.24)*:*

$$I = F(1) - F(-1) = 2.07977. \tag{4.26}$$

Berechnet mit der Tschebyscheff-Quadratur ergibt sich für $n = 5$

$$Q_5 = 2.02298 \tag{4.27}$$

In der Abbildung 4.6 sehen wir den Konvergenzverlauf der Tschebyscheff-Quadratur bezüglich n *der Anzahl der Stützstellen.*

Ordnung	x_i	α_i	
$N = 2$			
	0.7071067811865	1.570796326795	1.11072073454
	-0.7071067811865	1.570796326795	1.11072073454
$N = 3$			
	0.8660254037844	1.047197551197	0.5235987755983
	6.123233995737e-17	1.047197551197	1.047197551197
	-0.8660254037844	1.047197551197	0.5235987755983
$N = 4$			
	0.9238795325113	0.7853981633974	0.3005588649422
	0.3826834323651	0.7853981633974	0.7256132880349
	-0.3826834323651	0.7853981633974	0.7256132880349
	-0.9238795325113	0.7853981633974	0.3005588649422
$N = 5$			
	0.9510565162952	0.628318530718	0.1941611038725
	0.5877852522925	0.628318530718	0.5083203692315
	6.123233995737e-17	0.628318530718	0.628318530718
	-0.5877852522925	0.628318530718	0.5083203692315
	-0.9510565162952	0.628318530718	0.1941611038725
$N = 6$			
	0.9659258262891	0.5235987755983	0.1355173351172
	0.7071067811865	0.5235987755983	0.3702402448465
	0.2588190451025	0.5235987755983	0.5057575799637
	-0.2588190451025	0.5235987755983	0.5057575799637
	-0.7071067811865	0.5235987755983	0.3702402448465
	-0.9659258262891	0.5235987755983	0.1355173351172
$N = 7$			
	0.9749279121818	0.4487989505128	0.09986716162673
	0.781831482468	0.4487989505128	0.2798215687297
	0.4338837391176	0.4487989505128	0.4043538823593
	6.123233995737e-17	0.4487989505128	0.4487989505128
	-0.4338837391176	0.4487989505128	0.4043538823593
	-0.781831482468	0.4487989505128	0.2798215687297
	-0.9749279121818	0.4487989505128	0.09986716162673
$N = 8$			
	0.9807852804032	0.3926990816987	0.07661179030404
	0.8314696123025	0.3926990816987	0.2181719203259
	0.5555702330196	0.3926990816987	0.3265173532116
	0.1950903220161	0.3926990816987	0.385153478958
	-0.1950903220161	0.3926990816987	0.385153478958
	-0.5555702330196	0.3926990816987	0.3265173532116
	-0.8314696123025	0.3926990816987	0.2181719203259
	-0.9807852804032	0.3926990816987	0.07661179030404

Tabelle 4.1: Gewichte für die Gauß-Tschebyscheff-Quadratur.

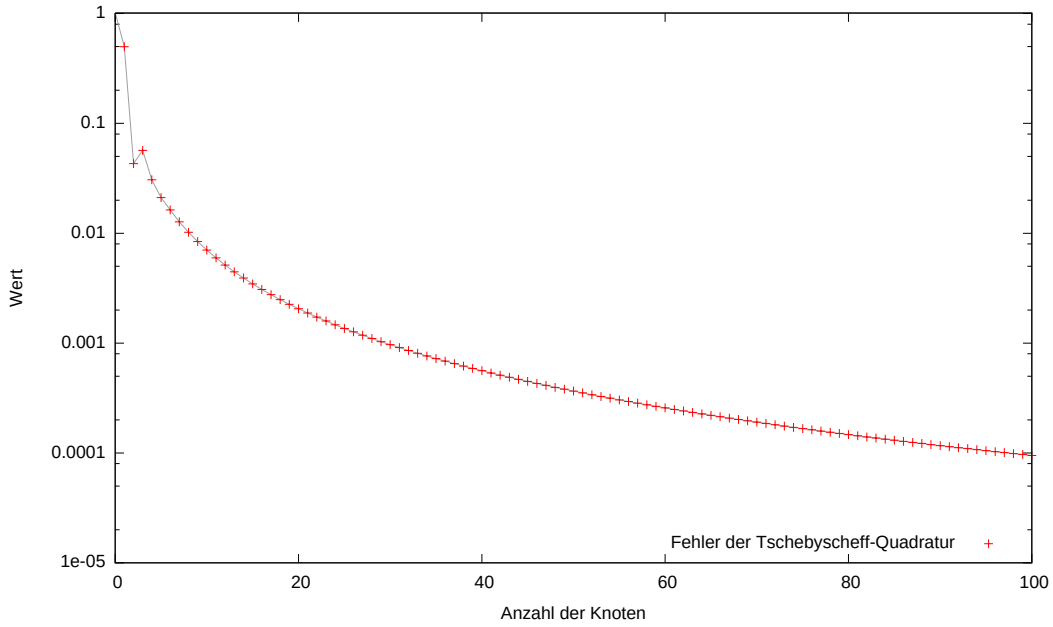

Abbildung 4.6: Konvergenzverlauf der Tschebyscheff-Quadratur in Abhängigkeit der Stützstellen für das Integral (4.25).

Legendre-Quadratur

Die Gauß-Quadratur für das Intervall $[a, b]$ mit den Nullstellen der Legendre Polynome wird auch als Legendre-Quadratur bezeichnet.

Für die Gewichtsfunktion der Legendre Polynome gilt:

$$\omega(x) = 1. \tag{4.28}$$

Mit der Gewichtsfunktion (4.28) ergibt sich für die Gewichte ω_i folgende Form:

$$\omega_i = \frac{2}{(1 - x_i^2)(P_n'(x_i))^2}. \tag{4.29}$$

Die Darstellung (4.29) findet man beispielsweise auch in *NUMERICAL RECIPES IN C: THE ART OF SCIENTIFIC COMPUTING* [NRC88].

Die in Tabelle 4.2 angegebenen Stützstellen x_i und der zugehörigen Gewichte α_i haben wir mit der rekursiven Legendre-Polynom-Darstellung und einem Bisektionsverfahren berechnet.

Das Programm finden wir auf der CD im Verzeichnis `Programme\ex:Legendre`.

Beispiel 4.1.17 (Legendre-Quadratur)
Als Beispielfunktion betrachten wir das Integral der Funktion:

$$f(x) := 2 \cos 2x e^{\sin 2x}, \tag{4.30}$$

auf dem Interval $\Omega := [-1, 1]$.
Dies ist dieselbe Funktion und Konfiguration wie im Beispiel 4.16. Damit gilt für die Stammfunktion der Funktion (4.23) *ebenfalls:*

$$F(x) = e^{\sin 2x}. \tag{4.31}$$

und das Integral ergibt sich zu dem Term:

$$I = \int_{\Omega} f(x)dx. \tag{4.32}$$

Daraus ergibt sich mit der Stammfunktion (4.31)*:*

$$I = F(1) - F(-1) = 2.07977. \tag{4.33}$$

Berechnet über die Legendre-Quadratur ergibt sich für $n = 5$

$$Q_5 = 2,02298. \tag{4.34}$$

In der Abbildung 4.7 sehen wir den Konvergenzverlauf der Legendre-Quadratur bezüglich n *der Anzahl der Stützstellen.*

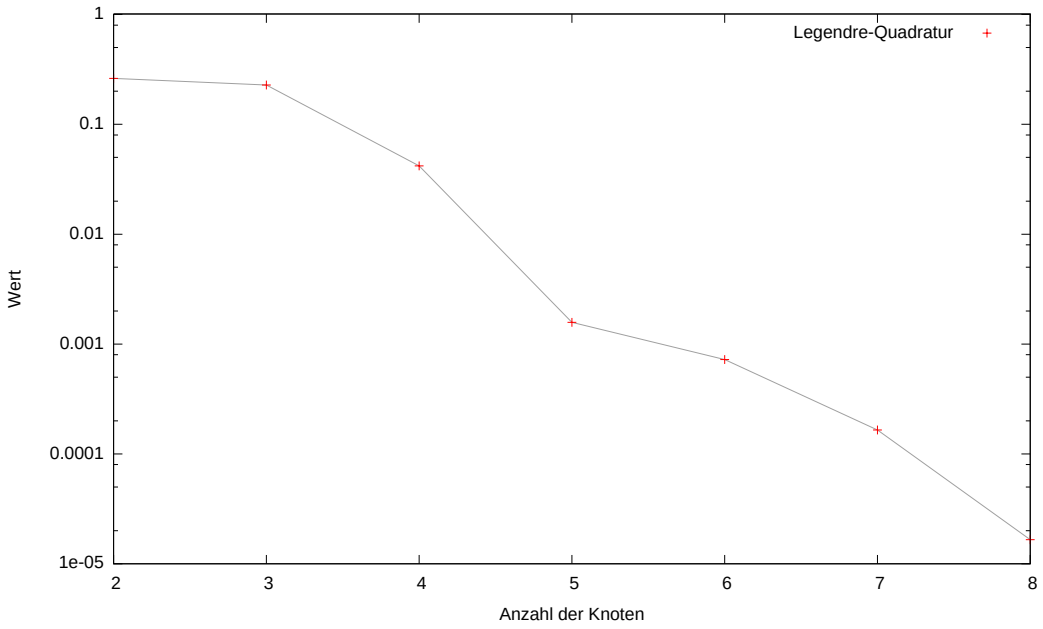

Abbildung 4.7: Konvergenzverlauf der Legendre-Quadratur in Abhängigkeit der Stützstellen für das Integral (4.25).

Ordnung	x_i	α_i
$N = 1$		
	0	2
$N = 2$		
	-0.57735026919	1
	0.57735026919	1
$N = 3$		
	-0.774596669241	0.555555555556
	$-8.30201112542e-19$	0.888888888889
	0.774596669241	0.555555555556
$N = 4$		
	-0.861136311594	0.347854845137
	-0.339981043585	0.652145154863
	0.339981043585	0.652145154863
	0.861136311594	0.347854845137
$N = 5$		
	-0.906179845939	0.236926885056
	-0.538469310106	0.478628670499
	$-8.30201112542e-19$	0.568888888889
	0.538469310106	0.478628670499
	0.906179845939	0.236926885056
$N = 6$		
	-0.932469514203	0.171324492379
	-0.661209386466	0.360761573048
	-0.238619186083	0.467913934573
	0.238619186083	0.467913934573
	0.661209386466	0.360761573048
	0.932469514203	0.171324492379
$N = 7$		
	-0.949107912343	0.129484966169
	-0.741531185599	0.279705391489
	-0.405845151377	0.381830050505
	$-8.30201112542e-19$	0.417959183673
	0.405845151377	0.381830050505
	0.741531185599	0.279705391489
	0.949107912343	0.129484966169
$N = 8$		
	-0.960289856498	0.10122853629
	-0.796666477414	0.222381034453
	-0.525532409916	0.313706645878
	-0.183434642496	0.362683783378
	0.183434642496	0.362683783378
	0.525532409916	0.313706645878
	0.796666477414	0.222381034453
	0.960289856498	0.10122853629

Tabelle 4.2: Gewichte für die Legendre-Quadratur.

Laguerre-Quadratur

Die Laguerre[10]-Quadratur ist eine Variante der Gauß-Quadratur, die die zu untersuchende Funktion auf dem Intervall von $[0, \infty]$ integriert.
Sie ist auch unter den Namen Gauß-Laguerre-Integration oder Gauß-Laguerre-Quadratur bekannt.
Wir betrachten nun das Integral:

$$\int_0^\infty f(x)dx. \tag{4.35}$$

Wir formen nun das Integral (4.35) in

$$I = \int_0^\infty \omega(x)e^x f(x)dx \tag{4.36}$$

um. Die Gewichtsfunktionen $\omega(x)$ sind die orthogonalen Laguerre-Polynome. Die Laguerre-Polynome lassen sich durch drei gebräuchliche Formen darstellen:

- Über die Rodrigues[11]-Formel:

$$L_n(x) := \frac{e^x}{n!}\frac{d^n}{dx^n}(x^n e^{-x}), \tag{4.37}$$

- Eine alternative Darstellung ist die rekursive Definition

$$L_n(x) = \frac{(2n+1-x)L_n(x) - nL_{n-1}(x)}{n+1}, \tag{4.38}$$

  für den Rekursionsanfang gilt

$$\begin{aligned} L_0(x) &= 1, \\ L_1(x) &= -x+1, \end{aligned} \tag{4.39}$$

  und $n = 2, 3, \cdots$.

- Die letzte gebräuchliche Definition ist die Lösung der Differentialgleichung:

$$xy''(x) + (1-x)y'(x) + ny(x) = 0, \tag{4.40}$$

  für n gilt $n \in \mathbb{N}_0$.

Das Integral (4.36) approximieren wir nun zur numerischen Auswertung durch die Summe

$$Q := \sum_{i=}^{n} f(x_i)e^{x_i}\alpha_i. \tag{4.41}$$

Die betrachteten Stützstellen x_i sind die Nullstellen des Laguerre-Polynoms der entsprechenden Ordnung und für die Gewichte gilt:

$$\alpha_i := \frac{x_i}{((n+1)L_{n+1}(x_i))}. \tag{4.42}$$

[10] Benannt nach dem französischen Mathematiker Edmond Nicolas Laguerre (1834-1886).
[11] Benannt nach dem französischen Mathematiker Benjamin Olinde Rodrigues (1795-1851).

Ordnung	x_i	α_i	$\alpha_i e^x$
$N = 2$			
	0.585786437627	0.853553390593	1.533326033119
	3.414213562373	0.146446609407	4.450957335055
$N = 3$			
	0.415774556783	0.711093009929	1.077692859271
	2.294280360279	0.278517733569	2.762142961902
	6.289945082937	0.010389256502	5.601094625434
$N = 4$			
	0.322547689619	0.603154104342	0.832739123838
	1.745761101158	0.357418692438	2.048102438454
	4.536620296921	0.038887908515	3.631146305822
	9.395070912301	0.000539294706	6.487145084408
$N = 5$			
	0.263560319718	0.521755610583	0.679094042208
	1.413403059107	0.398666811083	1.638487873603
	3.596425771041	0.075942449682	2.769443242371
	7.085810005859	0.003611758680	4.315656900921
	12.640800844276	2.33699724e-05	7.219186354354
$N = 6$			
	0.222846604179	0.458964673950	0.573535507423
	1.188932101673	0.417000830772	1.369252590712
	2.992736326059	0.113373382074	2.260684593383
	5.775143569105	0.010399197453	3.350524582355
	9.837467418383	0.000261017202	4.886826800211
	15.982873980602	8.98547906e-07	7.849015945596
$N = 7$			
	0.193043676560	0.409318951701	0.496477597540
	1.026664895339	0.421831277862	1.177643060861
	2.567876744951	0.147126348658	1.918249781660
	4.900353084526	0.020633514469	2.771848636232
	8.182153444563	0.001074010143	3.841249122489
	12.734180291798	1.58654643e-05	5.380678207922
	19.395727862263	3.17031548e-08	8.405432486828
$N = 8$			
	0.170279632305	0.369188589342	0.437723410493
	0.903701776799	0.418786780814	1.033869347666
	2.251086629866	0.175794986637	1.669709765659
	4.266700170288	0.033343492261	2.376924701759
	7.045905402393	0.002794536235	3.208540913348
	10.758516010181	9.07650877e-05	4.268575510825
	15.740678641278	8.48574672e-07	5.818083368672
	22.863131736889	1.04800117e-09	8.906226215292

Tabelle 4.3: Gewichte für die Laguerre-Quadratur.

Bemerkung 4.1.18 (Nullstellenberechnung Laguerre-Polynome)
Die Stützstellen in Tabelle 4.3 wurden mit einem C++-Programm berechnet, in dem wir die Laguerre-Polynome über die rekursive Definition (4.38) *berechnet haben und die Nullstellen über einen einfachen Bisektions-Algorithmus gesucht haben.*
Das Programm liegt im Verzeichnis `Programme/ex:Laguerre`.

Beispiel 4.1.19 (Laguerre-Quadratur)
Wir betrachten das Integral der Funktion

$$f(x) := e^{-x}, \tag{4.43}$$

auf dem Interval $\Omega := [0, \infty)$. *In der Abbildung 4.8 sehen wir den Konvergenzverlauf des approximierten Integralwert gegen den exakten Wert* $I(f, \Omega) = 1$ *in Abhängigkeit von der Anzahl der verwendeten Stützstellen.*

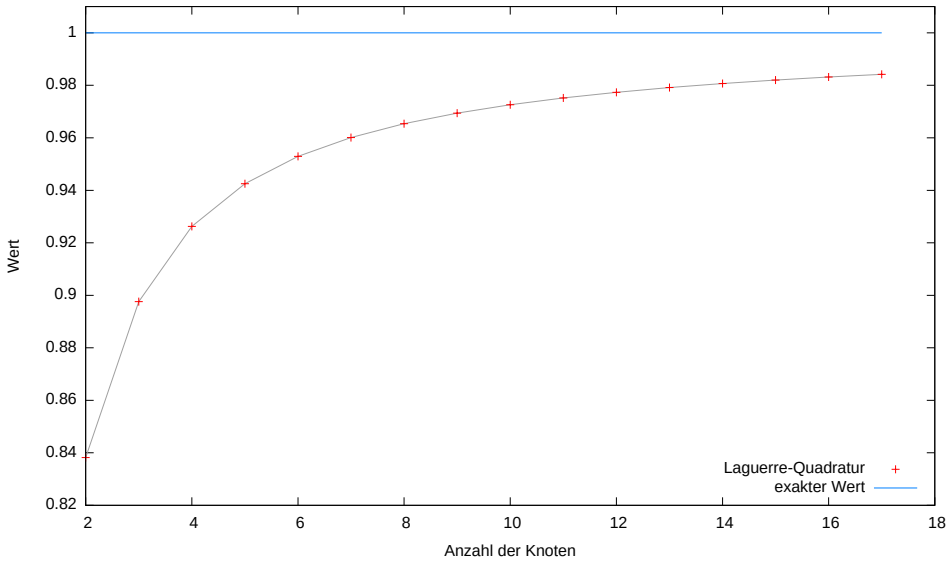

Abbildung 4.8: Konvergenzverlauf der Funktion (4.43) mit der Laguerre-Quadratur.

Kronrod-Quadratur

Die Kronrod-Quadratur ist eine Variante der Gauß-Quadratur. Hierbei wird die Gauß-Quadratur um zusätzliche Stützstellen, die Nullstellen der Stieltjes-Polynome, erweitert. Durch diesen Ansatz lässt sich ein Verfahren mit zugehörigem Fehlerschätzer konstruieren.

Eine gebräuchliche Variante ist die *G7,K15* auf dem Intervall $[-1, 1]$. Hierbei wird die 7-Punkt Gauß-Quadratur mit einer 15-Punkt Kronrod-Quadratur als Fehlerschätzer verknüpft. Für die Knoten und Gewichte gilt:

Gauß		Kronrod	
Knoten	Gewichte	Knoten	Gewichte
		± 0.991455371120813	0.022935322010529
± 0.949107912342759	0.129484966168870	± 0.949107912342759	0.063092092629979
		± 0.864864423359769	0.104790010322250
± 0.741531185599394	0.279705391489277	± 0.741531185599394	0.140653259715525
		± 0.586087235467691	0.169004726639267
± 0.405845151377397	0.381830050505119	± 0.405845151377397	0.190350578064785
		± 0.207784955007898	0.204432940075298
0	0.417959183673469	0	0.209482141084728

Tabelle 4.4: Gauß-Kronrod Knoten und Gewichte für die G7,K15 Variante.

Für nähere Betrachtungen sei hierzu auf folgende Arbeit verwiesen: *CALCULATION OF GAUSS-KRONROD QUADRATURE RULES* von Dirk P. Laurie [Lau97].
Diese Quadratur-Regel wird gerne zur Berechnung des Hestons-Modell verwendet.

4.2 Das Black-Scholes-Modell

Das Black-Scholes-Modell[12] wurde 1973 erstmals in der Arbeit *The Pricing of Options and Corporate Liabilities* von Black und Scholes [BS73] veröffentlicht.

4.2.1 Das Modell

Auf eine detaillierte Herleitung des Black-Scholes Modell wollen wir an dieser Stelle verzichten. Wir betrachten die Gleichungen des Modells und erläutern diese. Eine Herleitung des Black-Scholes Modell finden wir beispielsweise in der Originalarbeit von Black und Scholes [BS73] oder von Merton [Mer73].

Zunächst betrachten wir die Call-Option:

$$C(S,t) = S\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2). \tag{4.44}$$

Für die Put-Option gilt:

$$P(S,t) = Ke^{-r(T-t)}\Phi(-d_2) - S\Phi(-d_1), \tag{4.45}$$

wobei die Größen d_1, d_2 und $\Phi(x)$ wie folgt erklärt sind:

$$d_1 = \frac{\ln(S/K) + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} \tag{4.46}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{4.47}$$

$$\Phi(x) = \frac{1}{2\pi}\int_{-\infty}^{x} e^{-z^2/2}\, dz \tag{4.48}$$

[12] Benannt nach Fischer Black (1938-1995) und Myron Samuel Scholes (1941).

Die Größen d_1 und d_2 werden zur einfacheren Darstellung verwendet. $\Phi(x)$ ist die Verteilungsfunktion der Standardnormalverteilung. Folgende fünf Parameter beschreiben damit den Wert einer Option im Black-Scholes-Modell:

- Der aktuelle Aktienkurs S,
- der vereinbarte Basispreis K,
- der Zinssatz r,
- die angenommene Volatilität σ des Basiswerts,
- sowie die Restlaufzeit $T - t$ die sich aus dem Endzeitpunkt T und der aktuellen Zeit t berechnet.

Als nächstes befassen wir uns mit den Ableitungen des Optionspreises bezüglich seiner Parameter, dies sind die Sensitivitäten des Modells. Diese Sensitivitäten werden als die "Griechen" bezeichnet.

4.2.2 Die "Griechen"

Wir betrachten nun die sogenannten *Griechen* (engl. the Greeks). Sie werden aufgrund ihrer üblichen Bezeichnung mit den griechischen Buchstaben Δ, Γ, Λ, Θ, P und Ω so genannt.

Formal betrachtet sind die *Griechen* die Ableitungen der Black-Scholes-Formel (4.44) beziehungsweise (4.45) bezüglich der Parameter. Sie repräsentieren damit die Sensitivität des Modells bezüglich des entsprechenden Parameters. Im Folgenden geben wir nun die ökonomische Repräsentation der Griechen wieder.

Δ (Delta)

Delta Δ gibt die Sensitivität des Optionspreises bezüglich der Änderungen des Basiswertes an. Im Detail bedeutet dies die Abhängigkeit des Optionspreises bei einer Änderung des zugrundeliegenden Basiswerts um eine Einheit.

Damit gilt für eine europäische Call-Option:

$$\Delta_C = \frac{\partial C}{\partial S} = \Phi(d_1) \geq 0. \tag{4.49}$$

Analog folgt für die europäische Put-Option:

$$\Delta_P = \frac{\partial P}{\partial S} = -\Phi(-d_1) = \Phi(d_1) - 1 \leq 1. \tag{4.50}$$

Häufig wird Δ als wichtigste Sensitivität einer Option betrachtet, da der Basispreis die volatilste Größe ist.

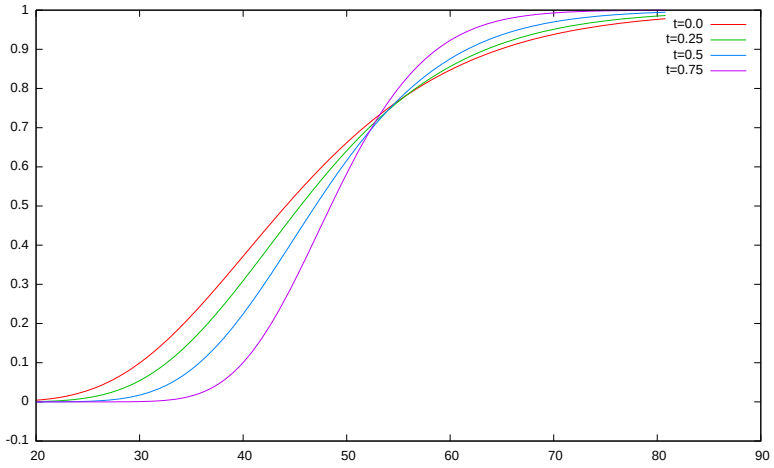

Abbildung 4.9: Delta.

In der Abbildung 4.9 sehen wir den Verlauf von Δ für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

Γ (Gamma)

Gamma Γ nimmt eine Sonderrolle unter den Griechen ein, da es als einzigste Größe eine zweite Ableitung beschreibt und damit nicht direkt die Sensitivität bezüglich einer Einflussgröße wiedergibt, sondern die Krümmung. In diesem Fall beschreibt Γ die Schwankungen von Δ. Aus den Gleichungen (4.49) und (4.50) folgt, dass Γ für Call und Put identisch ist:

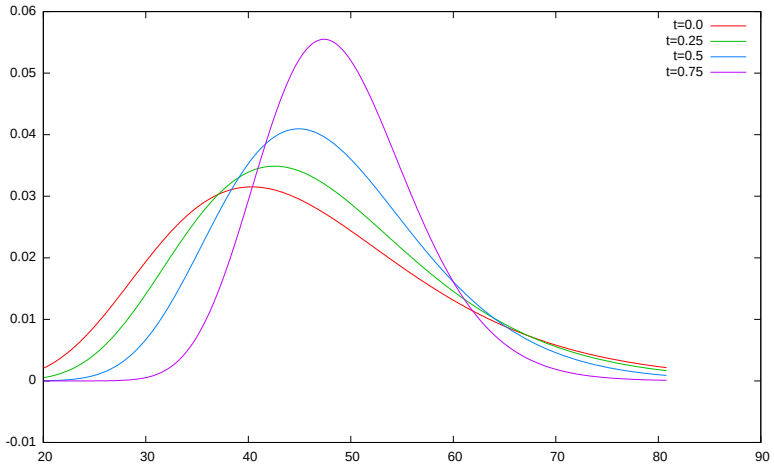

Abbildung 4.10: Gamma.

$$\Gamma = \frac{\partial^2 C}{\partial S^2} = \frac{\partial^2 P}{\partial S^2} = \frac{\Phi'(d1)}{S\sigma\sqrt{T-t}} \geq 0. \tag{4.51}$$

In der Abbildung 4.10 sehen wir den Verlauf von Γ für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

Λ (Lambda)

Die Größe Lambda Λ wird gelegentlich auch mit Kappa κ oder Vega bezeichnet. Λ gibt die Sensitivität des Optionspreises bezüglich der Volatilität σ an. Damit beschreibt es das Verhalten des Optionspreises bezüglich der Unsicherheit der Volatilität. Für Lambda gilt, dass es unabhängig von der Betrachtung eines Calls beziehungsweise eines Puts ist. Damit folgt:

$$\Lambda = \frac{\partial C}{\partial \sigma} = \frac{\partial P}{\partial \sigma} = S\Phi'(d_1)\sqrt{T-t} \geq 0. \tag{4.52}$$

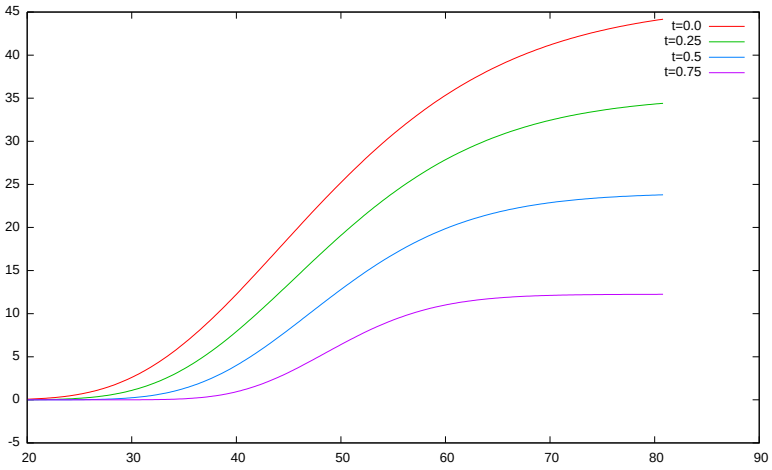

Abbildung 4.11: Lambda.

In der Abbildung 4.11 sehen wir den Verlauf von Λ, für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

Θ (Theta)

Der Griechische Buchstaben Theta Θ beschreibt die Sensitivität des Black-Scholes-Modells bezüglich der Zeit.
Für eine Europäische-Option ergibt sich damit aus der Gleichung (4.44):

$$\Theta_C = \frac{\partial C}{\partial} = \frac{S\Phi'(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}\Phi(d_2). \tag{4.53}$$

Da sich der Payoff mit der Zeit dem Auszahlungswert annähert, ist Θ unter der Voraussetzung, dass alle anderen Größen unverändert bleiben, nie positiv.
Analog können wir Θ auch für einen Europäischen Put angeben. Dies folgt direkt aus der Gleichung (4.45):

$$\Theta_P = \frac{\partial P}{\partial} = \frac{S\Phi'(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}\Phi(-d_2) \tag{4.54}$$

Das Vorzeichen für eine Put-Option können wir nicht so eindeutig angeben wie für die Call-Option, da es von unterschiedlichen Parametern abhängig ist.

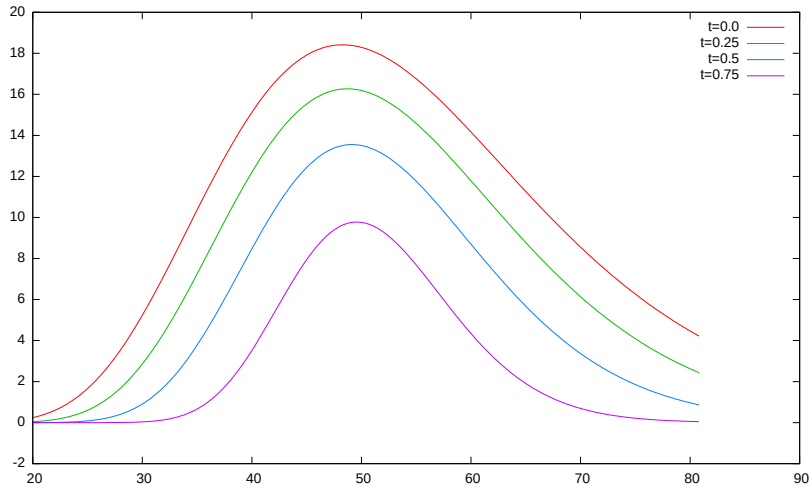

Abbildung 4.12: Theta.

In der Abbildung 4.12 sehen wir den Verlauf von Θ für eine Europäische-Call-Option für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

P (Rho)

Die Unsicherheiten durch mögliche Zinsschwankungen werden durch die Größe Rho P beschrieben. Die Ableitung des Optionspreises ist für den Call und den Put unterschiedlich. Für den Call gilt:

$$\mathsf{P}_C = \frac{\partial C}{\partial r} = (T-t)Ke^{-r(T-t)}\Phi(d_2) \geq 0. \tag{4.55}$$

P ist für eine Europäische-Call-Option immer positiv.
Für die Put-Option ist P durch folgende Gleichung beschrieben:

$$\mathsf{P}_P = \frac{\partial P}{\partial r} = -(T-t)Ke^{-r(T-t)}\Phi(-d_2) \leq 0 \tag{4.56}$$

und P nimmt nur negative Werte an.

In der Abbildung 4.13 sehen wir den Verlauf von P für eine Europäische-Call-Option für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

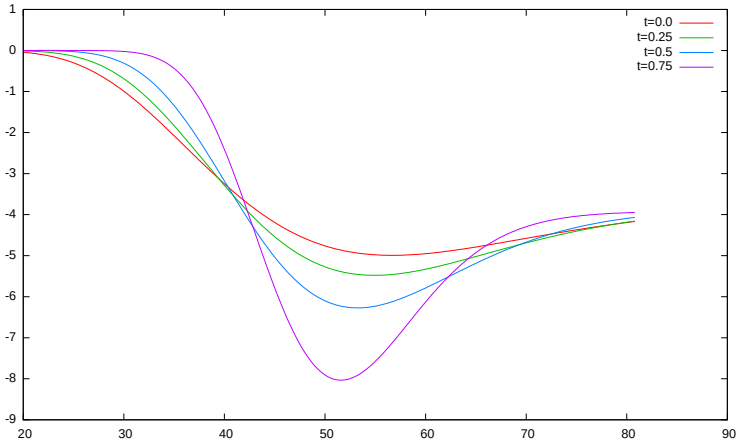

Abbildung 4.13: Rho.

Ω (Omega)

Ω beschreibt das Verhältnis einer normierten Differenz des Optionspreises $\frac{\Delta C}{C}$ zur normierten Differenz des zugrundeliegenden Aktienpreises $\frac{\Delta S}{S}$. Damit beschreibt Ω die Elastizität der Option oder kann auch als eine prozentuale Sensitivität interpretiert werden. Für eine Call-Option gilt damit:

$$\Omega_C = \frac{\frac{\Delta C}{C}}{\frac{\Delta S}{S}} = \Phi(d_1)\frac{S}{C} > 0. \tag{4.57}$$

Bei einer Put-Option ergibt sich für Ω:

$$\Omega_P = \frac{\frac{\Delta P}{P}}{\frac{\Delta S}{S}} = (\Phi(d_1) - 1)\frac{S}{P} < 0. \tag{4.58}$$

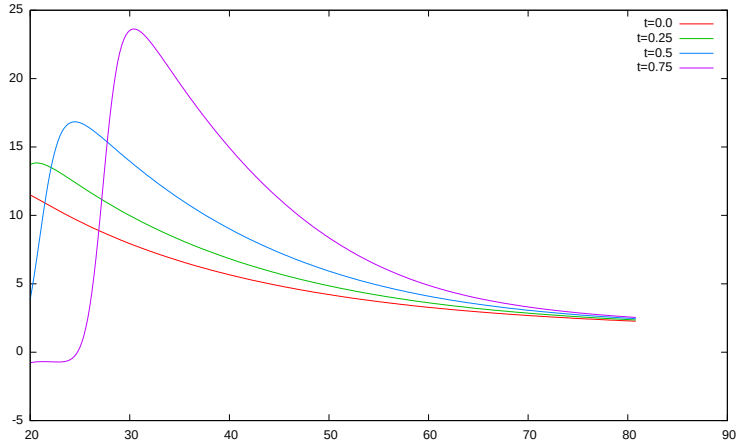

Abbildung 4.14: Omega.

In der Abbildung 4.14 sehen wir den Verlauf von Ω für eine Europäische-Call-Option für unterschiedliche Zeitpunkte t in Abhängigkeit des Basiswerts.

4.2.3 Die Berechnung

Da das Black-Scholes-Modell vollständig in einer geschlossenen Formel vorliegt, ist die Berechnung eines Optionspreises mit diesem Modell relativ einfach. Dies hat sicher zur starken Verbreitung des Modelles beigetragen.
In diesem Teilabschnitt gehen wir nur kurz auf die Berechnung des Wertes einer Europäischen-Call-Option mit dem Black-Scholes-Modell ein. Die Vorgehensweise ist für eine Put-Option analog.

Algorithmus 4.2.1 beschreibt die Berechnung des Wertes einer Europäischen Call Option. Die Funktion $Q(\cdot)$ in Zeile 8 steht stellvertretend für ein geeignetes Quadraturverfahren zur Berechnung der Funktion $\Phi(\cdot)$ aus Gleichung (4.48).
Im folgenden Beispiel betrachten wir die Umsetzung und die Ergebnisse des Algorithmus 4.2.1.

Algorithm 4.2.1 Europäische Call Option mit dem Black-Scholes Modell

1: set: r, T, K, σ,
2: set S, t
3: $d_1 = \frac{\ln(S/K)+(r+\sigma^2/2)(T-t)}{\sigma\sqrt{T-t}}$
4: $d_2 = d_1 - \sigma\sqrt{T-t}$
5: $C(S,t) = S\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2)$
6:
7: $\Phi(x)$
8: $\phi := Q(\int_{-\infty}^{x} e^{-z^2/2}\, dz)$
9: return $\frac{1}{2\pi}\phi$

Beispiel 4.2.2 (Black-Scholes-Berechnung)
Wir betrachten in diesem Beispiel eine Europäische-Call-Option, die wir nach dem in Algorithmus 4.2.1 beschriebenen Verfahren berechnen.

Die Implementierung der Zeilen 1-5 aus dem Algorithmus 4.2.1 benötigt keine nähren Betrachtung, da wir sie direkt umsetzen können. Interessanter ist die Berechnung der Funktion Φ, genauer gesagt die Wahl des Quadraturverfahrens.

Wir nutzen die Eigenschaften der zu integrierenden Funktion aus und bezeichnen diese im Folgenden mit:

$$\phi(z) := e^{-\frac{z}{2}}. \tag{4.59}$$

Da die Funktion (4.59) *bezüglich der Null symmetrisch ist, können wir das Integral $\Phi(x)$wie folgt schreiben:*

$$\Phi(x) := \frac{1}{2\pi}\int_{-\infty}^{x}\phi(z)dz = \int_{0}^{\infty} + \int_{0}^{x}\phi(z)dz.$$

Des Weiteren gilt:

$$\phi(z) \overset{|z|\to\infty}{\to} 0.$$

Damit müssen wir nur den Bereich nahe um die Null sehr genau approximieren. Für den Bereich $x \to \infty$ können wir eine einfachere Approximation verwenden. Konkret haben wir folgende Approximation verwendet:

$$\Phi(x) \approx Q(\phi(\cdot), x) = Q_1 + Q_2 + Q_3 + Q_4(x) \tag{4.60}$$

$$\begin{aligned} Q_1 &:= I_{Laguerre,16,\Omega=(10,\infty,]}(\phi) \\ Q_2 &:= I_{Legendre,8,\Omega=[2,10]}(\phi) \\ Q_3 &:= I_{Legendre,8,\Omega=[0,2]}(\phi) \\ Q_4(x) &:= I_{Legendre,8,\Omega=[0,x]}(\phi) \end{aligned} \tag{4.61}$$

Wir betrachten eine Europäische-Call-Option mit folgenden Parametern:

Größe	*Wert*	*Beschreibung*
r	*0.08*	*risikoloser Zins*
T	*1.*	*Gesamtlaufzeit der Option (in Jahren)*
K	*50*	*Kaufpreis*
σ	*0.25*	*Volatilität*

Wir betrachten nun den Optionspreis für folgende Zeitpunkte $t = 0$, $t = 0.25$, $t = 0.5$ und $t = 0.75$, für Aktienpreise aus dem Bereich $S \in (20,\ 80)$. Die Optionspreise sind in der Abbildung 4.15 abgebildet.

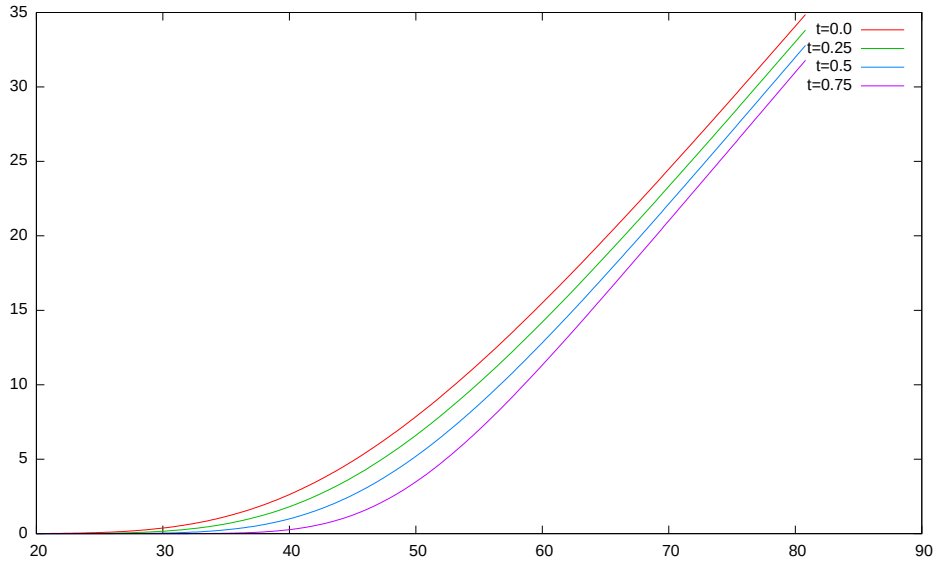

Abbildung 4.15: Optionspreise einer Europäischen Option, mit dem Black-Scholes-Modell.

Wir sehen, dass die Optionspreise mit kürzer werdender Restlaufzeit [13] gegen den Ausübungswert konvergiert.
Die Implementierung für unser Beispiel ist in C++ realisiert. Wir haben die Berechnung des Optionspreises und der Griechen in einer Klasse zusammengefasst. Das Programm kann neben einer Europäischen-Call-Option auch den Wert einer Europäischen-Put-Option und den jeweils zugehörigen Griechen berechnen.
Die Implementierung liegt im Verzeichniss `Programme\ex:BlackScholes`.

Im nächsten Abschnitt befassen wir uns mit dem Grund und den Möglichkeiten die das Automatische Differenzieren bei der Optionspreisberechnung mit dem Black-Scholes-Modell bietet.

4.2.4 Automatisches Differenzieren im Black-Scholes-Modell

Da die Ableitungen des Black-Scholes-Modells die sogenannten *Griechen*, schon in geschlossener Form vorliegen, ist es grundsätzlich nicht mehr notwendig Berechnung der Ableitungswerte über das Automatische Differenzieren zu bestimmen.
Allerdings bietet sich die Verwendung des Automatischen Differenzierens in diesem Fall trotzdem an, da die wir bei der Verwendung einer bereits getesteten AD-Bibliothek zum einen den Implementierungsaufwand zur Berechnung der *Griechen* reduzieren und zum anderen reduzieren wir auch mögliche Fehler bei der Implementierung der *Griechen* Berechnung.
Wir betrachten nun die Berechnung der *Griechen* durch Erweiterung des Beispieles 4.2.2 mit dem Konzept des Automatischen Differenzierens zur Berechnung der *Griechen*. Als Grundlage verwenden wir das Programm aus dem vorherigen Abschnitt. Da wir die dort entwickelte Klasse zur Berechnung einer Europäischen-Option als Template-Klasse implementiert haben, reicht es nun aus, dass wir den bisher verwendeten Gleitkomma-Typ[14] durch einen AD-Typ zuersetzen und die Variablen, bezüglich derer wir differenzieren möchten, entsprechend zu initialisieren.

Beispiel 4.2.3 (Black-Scholes Griechen Berechnung mit AD)
Wir betrachten nun folgendes Ausgangsproblem: Im Folgenden berechnen wir nun den Wert der Europäischen-Call-Option mit dem Black-Scholes-Modell und simultan dazu berechnen wir Θ, also die Sensitivität bezüglich t, mit dem Automatischen Differenzieren. In der Abbildung 4.16 sehen wir den Preis der Option für unterschiedliche Parameter. In der Abbildung 4.17 sehen wir die zugehörigen Sensitivitäten.
Die Programme, die wir für dieses Beispiel verwendet haben, finden wir im Verzeichnis `Programme\ex:BlackScholes`.

4.3 Das Heston-Modell

Das Heston-Modell ist als Erweiterung des Black-Scholes-Modell zu verstehen. Es beseitigt die bekannte Problematik, beziehungsweise die Schwachstellen der konstanten Volatilität des Black-Scholes-Modells.

[13] In Unserem Fall heißt dies $t \to 1$.
[14] Im vorherigen Abschnitt haben wir `double` verwendet.

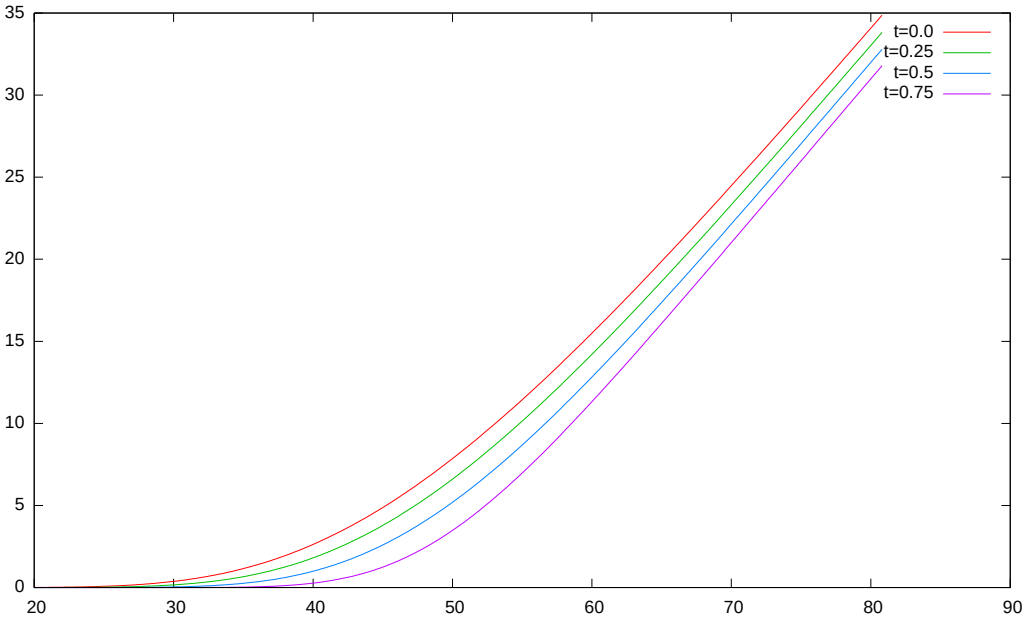

Abbildung 4.16: Europäische-Call-Option für unterschiedliche Parameter T.

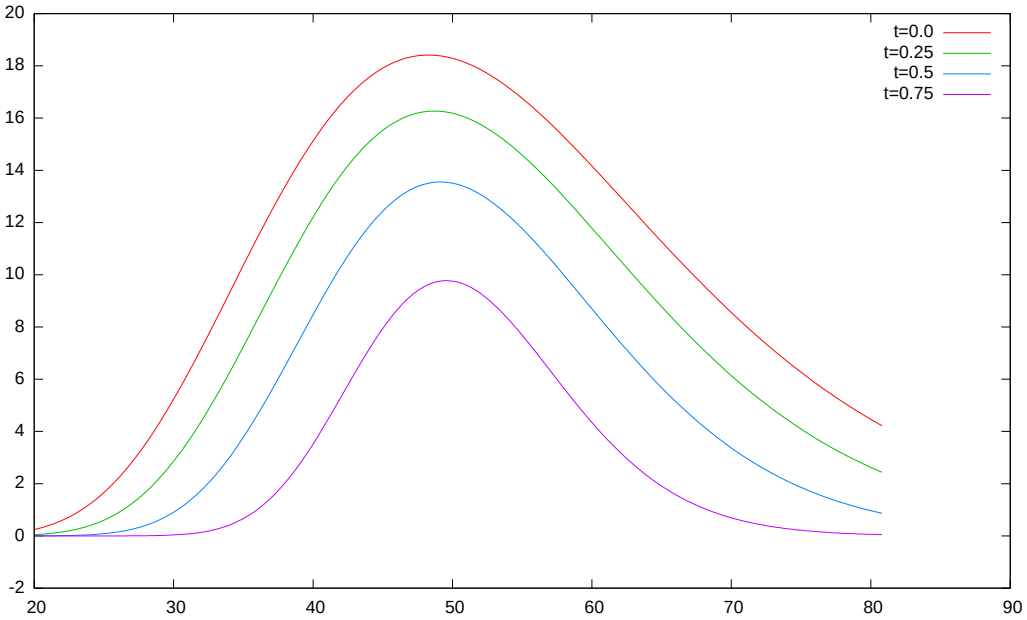

Abbildung 4.17: Zur Abbildung 4.16 gehörige Sensitivitäten Θ.

Das erste Mal wurde es im Jahr 1993 von Steven L. Heston in der Arbeit *A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options* [Hes93] vorgestellt.

4.3.1 Das Modell

In diesem Abschnitt beschreiben wir nun das Kapitalmarkt-Modell nach Heston. Dazu betrachten wir zunächst noch einmal das Kapitalmarkt-Modell nach Black und Scholes (4.44), (4.46), (4.47) und (4.48). Wir sehen, dass die Volatilität in die Größen d_1, d_2 und damit indirekt in $\Phi(x)$ einfließt. Da wir die Schwachstelle der konstanten Volatilität im Black-Scholes Modell beheben möchten, ändern wir die Konstante Volatilität σ in eine dynamische Größe. Dies realisieren wir durch einen *Wurzel-Diffusions-Prozess*[15] :

$$dX_t = \kappa(\Theta - X_t)dt + \sigma\sqrt{X_t}dW_t, \tag{4.62}$$

κ, σ, t und Θ sind reelle Größen, die folgende Bedingungen erfüllen:

$$\kappa \geq 0,$$
$$t \geq 0,$$
$$\Theta > 0.$$

(W_t) ist eine *Brownsche Bewegung*[16] . Die Lösung der Stochastischen-Differential-Gleichung (4.62) X_t nennt man einen Wurzel-Diffusions-Prozess.
Ein Wurzel-Diffusions-Prozess hat folgende Eigenschaft: $X_t \geq 0$ und die Parameter lassen sich in folgender Weise interpretieren:

- σ beschreibt die Volatilität des Prozesses.
- Θ legt das Niveau des Gleichgewichts für den Prozess fest.
- κ bestimmt die Geschwindigkeit der Rückkehr des Prozesses zu seinem Gleichgewichtszustand. Damit ist κ ein Regulierungsparameter[17]

Auf eine tiefergreifende theoretische Untersuchung des Wurzel-Diffusions-Prozess möchten wir an dieser Stellen verzichten und verweisen auf folgende Quelle [Ger06].
Für den Integrand ergibt sich, mit der über den Wurzel-Diffusions-Prozess modellierten Volatilität, folgender Ausdruck:

$$\phi(u,t) := e^{\mathbf{i}u(\ln S_0 + (r-q)t)} e^{a(u,t)} e^{b(u,t)}. \tag{4.63}$$

Für die Größen $a(u,t)$ und $b(u,t)$ aus der Gleichung 4.63 gilt:

$$\begin{aligned}
a(u,t) &:= rt\mathbf{i}u + \frac{\nu\kappa}{\sigma^2}\left((c_j - \rho\sigma\mathbf{i}u + d)t - 2\ln\frac{1-ge^{dt}}{1-g}\right) \\
b(u,t) &:= \frac{1}{\sigma^2}(c_j - \rho\sigma\mathbf{i}u + d)\frac{1-e^{dt}}{1-ge^{dt}} \\
c_j &:= \kappa + \sigma_0 - (2-j)\rho\sigma \text{ für } j = 1,2 \\
d &:= \sqrt{(\rho\sigma\mathbf{i}u - c_j)^2 - \sigma^2(k_j\mathbf{i}u + \mathbf{i}u^2)} \\
g &:= \frac{c_j - \rho\sigma\mathbf{i}u + d}{c_j - \rho\sigma\mathbf{i}u - d} \\
k_j &:= -1^{1+j}, \text{ für } j = 1,2
\end{aligned} \tag{4.64}$$

[15] Die englische Bezeichnung square-root diffusion ist im Deutschen ebenfalls gebräuchlich.
[16] Die *Brownsche Bewegung* wird auch als *Brownsche Molekularbewegung* bezeichnet und ist nach dem schottischen Botaniker Robert Brown (1773-1858) benannt.
[17] Im Englischen wird κ als mean reversion speed bezeichnet.

Auf die anschauliche Bedeutung der eingehenden Parameter gehen wir nun im Einzelnen ein.

- ν repräsentiert die langfristige Varianz des Modells.
- κ gibt die Dämpfung der dynamischen Volatilität an.
- ρ ist ein Korrelationsparameter.
- σ beschreibt die Volatilität der Varianz.
- σ_0 gibt die Volatilitätsniveau der dynamischen Volatilität an.

Der Wert einer Europäischen-*Call*-Option ergibt sich nun analog zum Black-Scholes-Modell, mit der Formel:

$$C(u,t) = \frac{e^{-\alpha \log K}}{\pi} \int_0^\infty \text{Re}\phi(u,t)du. \tag{4.65}$$

Für eine *Put*-Option können wir genauso vorgehen und kommen zu folgendem Ergebnis:

$$P(u,t) = \frac{e^{-\alpha \log K}}{\pi} \int_0^\infty \text{Re}\phi(u,t)du. \tag{4.66}$$

Für detailliertere Betrachtungen und Herleitungen möchten wir an dieser Stelle auf die verwendeten Arbeiten von Sergei Mikhailov und Ulrich Nögel *Heston's Stochastic Volatility Model Implementation, Calibration and Some Extensions* [MN05], Nimalin Moodley *The Heston Model: A Practical Approach* [Moo05] und Patrik Karlsson *The Heston Model* [Kar09] verweisen. Im nächsten Abschnitt gehen wir noch etwas vertieft auf die Bedeutung und Modellierung der Volatilität im Heston-Modell ein.

4.3.2 Volatilität des Heston-Modells

In diesem Abschnitt untersuchen wir die Auswirkungen der Erweiterung um eine nicht konstante Volatilität im Heston-Modell im Vergleich zum Black-Scholes-Modell mit konstanter Volatilität.
In einem ersten Schritt gehen wir nochmals auf die Problematik der konstanten Volatilität σ im Black-Scholes-Modell ein und stellen die dynamische Modellierung der Volatilität mit dem Wurzel-Diffusions-Prozess gegenüber.

Modellierung mit dynamischer und konstanter Volatilität

Das bekannteste Finanzmarkt-Modell, das Black-Scholes-Modell, modelliert die Volatilität des Marktes konstant. Dies bedeutet, dass die Flexibilität der Dynamik über die gesamte Laufzeit als konstant angenommen wird.
Diese Voraussetzung beschreibt das Verhalten des Marktes nur in speziellen Fällen ausreichend. Damit dies erfüllt ist, dürfen wir nur kurze Laufzeiten betrachten, da bei einer kurzen Laufzeit die Dynamik des Marktes näherungsweise konstant ist. Der andere Fall ist die Betrachtung des Marktes zu einer sehr ruhigen Zeit. Damit ist gemeint, dass keine strukturellen Veränderungen und auch keine plötzlichem Nervositäten den Markt beunruhigen.
Da die Einschränkung, nur kurze Laufzeiten betrachten zu können, nicht ausreichend ist. Die Finanzkrise von 2007/2008 sowie die Schuldenkrise im Euroraum hat gezeigt, dass unsere Finanzmärkte plötzlich und unerwartet sehr nervös und empfindlich auf Ereignisse reagieren

können. Somit ist die Annahme der konstanten Volatilität nicht ausreichend. Im Heston-Modell wird versucht, dieses Problem des Black-Scholes Modells durch eine dynamische Volatilität zu beheben.
Das Heston-Modell modelliert die Volatilität durch einen Wurzel-Diffusions-Prozess. Damit können wir die Volatilität besser beschreiben, da es neben einem Volatilitätsniveau einen Dämpfungsparameter und eine Volatilität der Markt-Volatilität gibt. Zusätzlich werden im Heston-Modell noch eine langfristige Varianz und eine zugehörige Korrelation als Größen eingeführt.
Den Preis, den wir durch die bessere Modellierung der Volatilität zahlen müssen, ist ein gestiegener Aufwand bei der Implementierung und der Berechnung. Zudem wird die Kalibrierung des Modells durch die gestiegene Anzahl an Parametern, die nicht direkt im Markt beobachtbar sind, schwieriger.
Durch die erhöhte Komplexität des Modells steigt der Nutzen von Methoden zur Effizienzsteigerung bei der Kalibrierung und der Berechnung von Sensitivitäten. Im Rahmen dieser Arbeit verwenden wir den Ansatz des Automatischen Differenzierens, um die Sensitivitäten zu berechnen. Zudem verwenden wir die berechneten Ableitungswerte auch zu Kalibrierungszwecken. Dies werden wir im Abschnitt 4.3.4 und 4.3.5 näher betrachten.

Vergleich der dynamischen und konstanten Volatilität

Hier stellen wir, um die Auswirkungen der dynamischen Volatilität im Heston-Modell zu veranschaulichen, ein vergleichbar Konfiguriertes Black-Scholes-Modell zur Optionspreis-Berechnung gegenüber.

Als Referenz-Beispiel verwenden wir eine Europäische-Call-Option mit folgenden initialen Parametern:

Größe	Wert	Beschreibung
r	0.08	risikoloser Zins
T	1.	Gesamtlaufzeit der Option (in Jahren)
K	50	Kaufpreis

Die Volatilität im Black-Scholes-Modell setzen wir auf den Wert $\sigma = 0.25$. Im Heston-Modell wählen wir für die Modellierung der Volatilität folgende Größen:

Größe	1.Fall	2. Fall	3. Fall	4. Fall	5. Fall
ν	0	0	0	0	0.05
κ	0	0	0	3	3
ρ	0	0	-0.5	-0.5	-0.5
σ	0	0.25	0.25	0.25	0.25
σ_0	0.25	0.25	0.25	0.25	0.25

Die Ergebnisse der einzelnen Fälle haben wir in der Abbildung 4.18 für eine Europäische-Call-Option veranschaulicht.

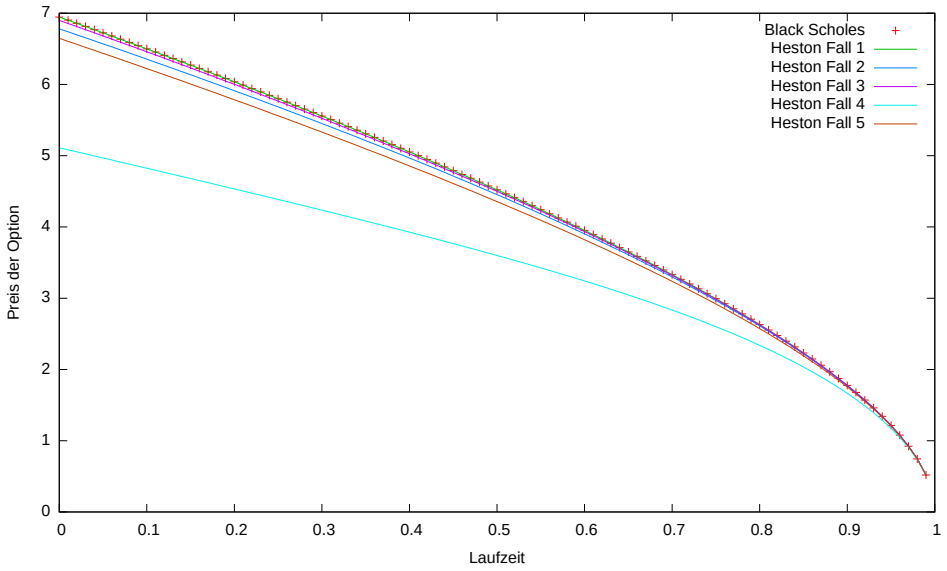

Abbildung 4.18: Europäische-Call-Option: Heston- und Black-Scholes-Modell im Vergleich.

Im 1. Fall sollten das Black-Scholes- und das Heston-Modell übereinstimmen. In den anderen Fällen können wir die Auswirkungen der einzelnen Parameter der dynamischen Volatilität auf den Optionspreis erkennen.

4.3.3 Berechnung

In diesem Abschnitt befassen wir uns mit der numerischen Berechnung des Heston-Modells zur Bestimmung des Preises einer Europäischen-Call-Option. Dazu verwenden wir die Formulierung des Heston-Modells in der Formulierung aus den Gleichungen (4.63) und (4.63).
Für die Berechnung des Integrals:

$$C(u,t) = \frac{e^{-\alpha \log K}}{\pi} \int_0^\infty \mathrm{Re}\phi(u,t)du, \tag{4.67}$$

greifen wir auf die im Abschnitt 4.1.2 betrachteten Quadratur-Varianten zurück, beziehungsweise wir verknüpfen die einzelnen Methoden geschickt miteinander.

- Kronrod $\int_a^b f(x)dx \approx \sum_{i=1}^n f(x_i)\omega_i$
- Laguerre $\int_0^\infty f(x)dx \approx \sum_{i=1}^n f(x_i)e^{x_i}\alpha_i$

$$\begin{aligned} C(u,t) &= \tfrac{e^{-\alpha \log K}}{\pi}(\textstyle\int_0^b \mathrm{Re}\phi(u,t)du + \int_0^\infty \mathrm{Re}\phi(u+b,t)du) \\ &\approx \tfrac{e^{-\alpha \log K}}{\pi}(\textstyle\sum_{i=1}^{n_k} \mathrm{Re}\phi(u,t)\omega_i + \sum_{i=1}^{n_l} \mathrm{Re}\phi(u+b,t)e^{x_i}\alpha_i) \end{aligned}$$

Für die Berechnung des Optionspreises können wir zwei verschiedene Strategien verfolgen: Beim ersten Ansatz betrachten wir die Funktion $\phi(u,t)$ direkt als Komplexe-Funktion unter der Verwendung von komplexen Datentypen. Ein naheliegender Vorteil dieses Vorgehens ist

die einfache und direkte Umsetzung, da wir die Gleichung (4.63) unter der Verwendung der Größen (4.64) direkt implementieren können. Ein möglicher Nachteil dieses Vorgehens ist, dass die Effizienz der komplexen Datentypen häufig schlechter ist als die der reellen Datentypen.
Die zweite Strategie berücksichtigt von Anfang an, dass wir am Ende nur den Realteil der Funktion $\phi(u,t)$ benötigen. Dies verwenden wir direkt, indem wir von der Funktion $\phi(u,t)$ nur den Realteil berechnen. Dazu betrachten wir alle Teilgrößen aufgeteilt in ihren Realteil und Imaginärteil und berechnen so den Realteil der Funktion $\phi(u,t)$ nur über die Verwendung reeller Datentypen. Der offensichtliche Nachteil dieses Vorgehens ist der deutlich erhöhte Aufwand die Gleichungen so aufzustellen und zu implementieren. Allerdings ergibt sich damit der Vorteil, dass wir den Berechnungsaufwand reduzieren können und ausschließlich auf Standard-Datentypen zurückgreifen müssen.
Im Folgenden betrachten wir den vollständigen Algorithmus zur Berechnung einer Europäischen-Call-Option mit dem Heston-Modell. Im Algorithmus 4.3.1 betrachten wir die komplexe Variante und im Algorithmus 4.3.2 die Variante, die nur auf den reellen Datentypen aufbaut.

Algorithm 4.3.1 Heston-Modell - Komplex

$P_j(i) = \frac{1}{2} + \frac{1}{\pi} Gauss(i)$
$C() = S_t P_j(0) - K e^{-r\Delta_T P_j(1)}$
for i=1,N **do**
    $Q+ = f(x)$
end for
$P_{Call} := \frac{e^{-\alpha \log K}}{\pi} Q$

function $f(j,\phi)$
$\tau = \rho\sigma\phi I - b_j$
$d = \sqrt{\tau^2 - \sigma^2(2u_j\phi I - \phi^2)}$
$g = \frac{b_j - \rho\sigma\phi I + d}{b_j - \rho\sigma\phi I - d}$
$f_x = \frac{e^{-I\phi \log(K) C(j,T-t,\phi) + D(j,T-t,\phi) V_t + I\phi x}}{I\phi}$
return $real(f_x)$

function $C(j,\Delta_t,\phi)$
return $r\phi I\Delta_t + \frac{a}{\sigma^2}\frac{(b_j\rho\sigma\phi I + d)\Delta_t - 2\log(1-ge^{d\Delta_t})}{1-g}$

function $D(j,\Delta_t,\phi)$
return $\frac{b_j - \rho\sigma\phi I + d}{\sigma^2}\frac{1-e^{d\Delta_t}}{1-ge^{d\Delta_t}}$

Der Algorithmus 4.3.1 beschreibt die direkte Umsetzung der Gleichungen (4.63) und (4.64). Die Berechnung des Integrals wurde durch eine Quadratur-Regel ersetzt. Analog zum Algorithmus 4.3.1 haben wir im Algorithmus 4.3.2 den Optionspreis berechnet. Allerdings haben wir bei der Formulierung des Algorithmus die komplexwertigen Größen in ihren Real- und Imaginär-Anteil aufgeteilt und so eine reelle Version des Algorithmus erhalten.

Algorithm 4.3.2 Heston-Modell - Reell

for i=1,N **do**
  $Q+ = f(x)$
end for
$P_{Call} := \frac{e^{-\alpha \log K}}{\pi} Q$

function $f(x)$
$[\phi_R, \phi_I] := \phi(x, -(\alpha+1), T)$
$\tau_R := (\alpha^2 + \alpha - x^2)$ $\quad$ $\tau_I := (2\alpha+1)x$
$D := \tau_R^2 + \tau_I^2$
return $(cos(-x\log(K))(\phi_R\tau_R + \phi_I\tau_I) - \sin(-x\log(K))(\phi_I\tau_R - \phi_I\tau_R))\frac{e^{-rT}}{D}$

function $\phi(x_R, x_I, t)$
$\tau_{1,R} = -\rho\lambda x_I - \kappa$ $\quad$ $\tau_{1,I} = \rho\lambda x_R$
$\tau_{2,R} = \lambda^2(-x_I + x_R^2 - x_I)$ $\quad$ $\tau_{2,I} = \lambda^2(x_R + 2x_R x_I)$
$\tau_{3,R} = \tau_{1,R}^2 - \tau_{1,I} + \tau_{2,R}$ $\quad$ $\tau_{3,I} = 2\tau_{1,T}\tau_{1,I} + \tau_{2,I}$
$\tau_4 = \sqrt{\tau_{3,R}^2 + \tau_{3,I}^2}$ $\quad$ $\tau_5 = \arg \tau_{3,R}, \tau_{3,I}$
$d_R = \sqrt{\tau_4}\cos(\tau_5/2)$ $\quad$ $d_I = \sqrt{\tau_4}\sin(\tau_5/2)$
$\tau_{1,R} = \kappa\rho\lambda x_I - d_R$ $\quad$ $\tau_{1,I} = -\rho\lambda x_R - d_I$
$\tau_{2,R} = \kappa\rho\lambda x_I + d_R$ $\quad$ $\tau_{2,I} = -\rho\lambda x_R + d_I$
$\tau_3 = \tau_{2,R}^2 + \tau_{2,I}^2$
$g_R := \tau_{1,R}\tau_{2,R} + \tau_{1,I}\tau_{2,I}/\tau_3$ $\quad$ $g_I := \tau_{1,R}\tau_{2,R} - \tau_{1,I}\tau_{2,I} < ++ >/\tau_3$
$\tau_2 = e^{-d_R t}$
$\tau_{3,R} = \cos(d_I t)$ $\quad$ $\tau_{3,I} = \sin(d_I t)$
$\tau_{4,R} = 1 - \tau_2(g_R\tau_{3,R} - g_I\tau_{3,I})$ $\quad$ $\tau_{4,R} = -\tau_2(g_R\tau_{3,I} + g_I\tau_{3,R})$
$\tau_2 = \log(S_0) + (r-q)t$
$\tau_{3,R} = -x_I\tau_R$ $\quad$ $\tau_{3,I} = x_R\tau_R$
$\tau_5 = e^{\tau_{3,R}}$
$\xi_{1,R} := \tau_5\cos(\tau_{3,I})$ $\quad$ $\xi_{1,I} := \tau_5\sin(\tau_{3,I})$
$\tau_2 = \frac{\nu\kappa}{\lambda^2}$
$\tau_3 = (1-g_R)^2 + g_I^2$ $\quad$ $\tau_5 = \tau_3^2 + g_I^2$
$\tau_{6,R} = \tau_{4,R}\tau_3 - \tau_{4,I}g_I/\tau_3$ $\quad$ $\tau_{6,I} = \tau_{4,I}\tau_3 + \tau_{4,R}g_I/\tau_3$
$\tau_7 = \sqrt{\tau_{6,R}^2 + \tau_{6,I}^2}$ $\quad$ $\tau_8 = \arg(\tau_{6,R}, \tau_{6,I})$
$\tau_{9,R} = \tau_2(\tau_{1,R}t - 2\log(\tau_7))$ $\quad$ $\tau_{9,I} = \tau_2(\tau_{1,I}t - 2\tau_8)$
$\tau_{10} = e^{\tau_{9,R}}$
$\xi_{2,R} := \tau_{10}\cos(\tau_{9,I})$ $\quad$ $\xi_{2,I} := \tau_{10}\sin(\tau_{9,I})$
$\tau_2 = \frac{\sigma^2}{\lambda^2}$
$\tau_3 = \tau_{4,R} + \tau_{4,I}$ $\quad$ $\tau_5 = e^{-d_R t}$
$\tau_{6,R} = \cos(-d_I t)$ $\quad$ $\tau_{6,I} = \sin -d_I t$
$\tau_{7,R} = \tau_5\tau_{6,R}$ $\quad$ $\tau_{7,I} = \tau_5\tau_{6,I}$
$\tau_{8,R} = \tau_{1,R}(1-\tau_{7,R}) - \tau_{1,I}(-\tau_{7,I})$ $\quad$ $\tau_{8,I} = \tau_{1,R}(-\tau_{7,I}) + \tau_{1,I}(1-\tau_{7,R})$
$\tau_{9,R} = \tau_2\tau_{8,R}\tau_{4,R} + \tau_{8,I}\tau_{4,I}/\tau_3$ $\quad$ $\tau_{9,I} = \tau_2\tau_{8,I}\tau_{4,R} - \tau_{8,R}\tau_{4,I}/\tau_3$
$\tau_{10} = e^{\tau_{9,R}}$
$\xi_{3,R} := \tau_{10}\cos(\tau_{9,I})$ $\quad$ $\xi_{3,I} := \tau_{10}\sin(\tau_{9,I})$
$\phi_R := (\xi_{1,R}\xi_{2,R} - \xi_{1,I}\xi_{2,I})\xi_{3,R} - (\xi_{1,I}\xi_{2,R} + \xi_{1,R}\xi_{2,I})\xi_{3,I}$
$\phi_I := (\xi_{1,R}\xi_{2,R} - \xi_{1,I}\xi_{2,I})\xi_{3,I} + (\xi_{1,I}\xi_{2,R} + \xi_{1,R}\xi_{2,I})\xi_{3,R}$
return $[\phi_R, \phi_I]$

Nachdem wir nun beide Algorithmen zur Berechnung eines Optionspreises mit dem Heston-Modell betrachtet haben, sehen wir, dass der Aufwand zur Implementierung für eine rein reelle Umsetzung deutlich höher ist. Allerdings können wir bei der reellen Variante des Algorithmus auf einfachem Weg die Sensitivitäten berechnen. Prinzipiell könnten wir nun den Algorithmus von Hand bezüglich der Eingangsgrößen analog zum Vorgehen beim Black-Scholes Modell differenzieren. Bei der offensichtlichen Komplexität des Algorithmus ist aber der Aufwand und die Fehleranfälligkeit signifikant, sodass es effizienter ist, wenn wir den Ansatz des Automatischen Differenzierens wählen. Hierauf gehen wir im Abschnitt 4.3.4 näher ein.
Als Test-Methode, um unsere Implementierung zu überprüfen, wählen wir die Größen so, dass wir als Spezialfall einen Optionspreis nach dem Black-Scholes-Modell erhalten. Dies ist bereits in der Betrachtung von den Auswirkungen der dynamischen Volatilität im Abschnitt 4.3.2 geschehen.
Die Implementierung finden wir auf der CD im Verzeichnis `Programme/ex:Heston01`. Dort ist sowohl eine komplexe als auch eine reelle Version zur Berechnung einer Europäischen-Option mit dem Heston-Modell vorhanden.

4.3.4 Sensitivitäten Berechnung

In diesem Kapitel untersuchen wir die Sensitivitäten des Heston-Modells. Analog zum Black-Scholes-Modell untersuchen wir die Abhängigkeit des Preises bezüglich der Eingangsgrößen, bevor wir im Folgenden die Sensitivitäten betrachten und ihre Bedeutung für das Modell erläutern, beschreiben wir das Vorgehen bei der Berechnung.
Aus der Komplexität des Algorithmus resultiert, dass das Risiko Fehler beim manuellen Erstellen der Sensitivitätsableitungen zu machen, deutlich steigt. Des Weiteren ist der Zeitaufwand für das Erstellen der Ableitung nicht unerheblich. Daher bietet sich das Ableiten des Algorithmus' mit dem Automatischen Differenzieren an. Wir verwenden die reelle Version des Algorithmus' da die Berechnung der Ableitungen bei komplexwertigen Funktionen im Vergleich zu reellwertigen Funktionen generell komplexer ist und in unserem Fall die Funktion auch nicht holomorph bezüglich aller relevanten Einflussgrößen ist. Die Sensitivitäten erhalten wir nun durch die Verwendung eines AD-Datentypes in unserer Implementierung.
In unserer Implementierung haben wir dies von Anfang an bei der Berechnung der Sensitivitäten berücksichtigt und die Klasse als Template-Klasse realisiert. Somit müssen wir nur den Template-Parameter nur von `double` auf `AD` ändern und die AD-Variablen entsprechend der gewünschten Sensitivitäten initialisieren.
Die Implementierung finden wir auf der CD im Verzeichnis `Programme/ex:Heston02`. Das Programm basiert auf der reellen Version, zur Berechnung einer Europäischen Option mit dem Heston-Modell.
Nun geben wir noch einen Überblick über die einzelnen Sensitivitäten des Heston-Modells. Wir haben die Bezeichnung der einzelnen Sensitivitäten analog zu den Bezeichnungen der Griechen im Black-Scholes Modell gewählt.

Δ (Delta)

Die Sensitivität Δ (Delta) ist die Ableitung des Optionspreises bezüglich des Aktienpreises (Basiswert).

$$\Delta_C = \frac{\partial C}{\partial S}$$

Damit ist das Δ ein Maß für die Empfindlichkeit gegenüber Kursschwankungen.

Λ (Lambda)

Die Größe Λ (Lambda)[18] gibt die Sensitivität nach dem Volatilitätsniveau an.

$$\Lambda_C = \frac{\partial C}{\partial \sigma_0}$$

Λ kann somit als Indikator für die Empfindlichkeit des im Modell zugrundeliegenden Volatilitätsniveaus aufgefasst werden. In diesem Punkt unterscheidet sich die genaue Interpretation der Sensitivität vom Black-Scholes-Modell, da anstatt einer konstanten Volatilität eine dynamische Volatilität verwendet wird.

Θ (Theta)

Die Ableitung des Optionspreises nach der Zeit bezeichnen wir mit Θ (Theta).

$$\Theta_C = \frac{\partial C}{\partial T}$$

So ist Θ die Sensitivität bezüglich der Zeit und gibt damit die Empfindlichkeit des Optionspreises nach der bereits vergangenen Zeit an.

P (Rho)

Die Sensitivität bezüglich des im Modell berücksichtigten Zinses wird durch P (Rho) beschrieben.

$$\mathsf{P}_C = \frac{\partial C}{\partial r}.$$

Damit ergibt sich eine Kenngröße zur Beurteilung von Zinsrisiken.

Wir verzichten hier auf eine grafische Darstellung der Sensitivitäten, da wir den Fokus beim Heston-Modell mehr auf die Kalibrierung des Modells gelegt haben. Dazu betrachten wir die Sensitivitäten als Ableitungswerte, um ein gradientenbasiertes Verfahren zur Kalibrierung des Modells verwenden zu können. Dieses Vorgehen wird im Abschnitt 4.3.5 betrachtet.

4.3.5 Kalibrierung

In diesem Abschnitt betrachten wir anhand eines einfachen exemplarischen Beispiels, wie wir das Heston-Modell mit Hilfe der Ableitungen kalibrieren können. Wir betrachten im Folgenden eine Europäische-Call-Option, für die wir uns einen Referenzpreis $\mathcal{P}^\star$ vorgeben. Damit ergibt sich unsere Zielfunktion zu:

$$F_d(\sigma_0, \sigma, \rho, \kappa\nu) := \sum_{i=1}^{N} |P(K_i, T_i, \sigma_0, \sigma, \rho, \kappa, \nu) - \mathcal{P}^\star(K_i, T_i)|^d \text{ für } d > 0. \tag{4.68}$$

Als nächstes müssen wir nun die Funktion $F_d(\sigma_0, \sigma, \rho, \kappa\nu)$ minimieren. Durch die Wahl von d können wir das Konvergenzverhalten steuern.

[18] Λ (Lambda) wird gelegentlich auch mit Vega oder Kappa bezeichnet.

Bemerkung 4.3.3
Seien $\sigma_0^\star, \sigma^\star, \rho^\star, \kappa^\star$ und $\nu^\star$ die Parameter, die das Minimum von $F_d(\sigma_0, \sigma, \rho, \kappa\nu)$ beschreiben, ist dies unabhängig von der Wahl des Parameters d und es gilt $F_d(\sigma_0^\star, \sigma^\star, \rho^\star, \kappa^\star\nu^\star) = 0$.

Als Optimierungsmethode wählen wir das Gradienten-Verfahren. Auf eine Beschreibung dieses Verfahrens möchten wir an dieser Stelle verzichten und auf das Buch von Wilhelm Alt *Nichtlineare Optimierung* [Alt02] verweisen.

Optimierungsparameter d

Zuerst betrachten wir die Variation des Optimierungsparameters d und die aus ihm resultierende Konvergenzgeschwindigkeit.

In der Abbildung 4.19 haben wir die folgenden Parameter verwendet:

Parameter	Zielwert	Startwert
r	0	0
q	0	0
S_0	100	100
T	$\left[1/6\ 1/3\ 1/2\ 2/3\ 5/6\ 1\right]$	$\left[1/6\ 1/3\ 1/2\ 2/3\ 5/6\ 1\right]$
K	[90100110]	[90100110]
α	0.75	0.75
κ	3.00	3.5
λ	0.25	0.15
ν	0.0625	0.0625
ρ	-0.5	-0.5
σ	0.2	0.1

Wir lassen $\kappa,\ \lambda,\ \nu,\ \rho$ und σ als Optimierungsparameter zu. Für die Wahl von d haben wir die Werte 2 und 4 als Vertreter gewählt. Die Fehlerdarstellung in der Grafik 4.19 zeigt den Fehlervektor $\mathbf{e}$, der Anhand des exakten Lösungsvektors berechnet und mit der 2-Norm $\|\mathbf{e}\|_2$ in normierter Form geplottet wurde.

Asymptotisch betrachtet ist das Konvergenzverhalten identisch. Betrachten wir gezielt unser exemplarisches Beispiel, nimmt der absolute Fehler mit dem Parameter $d = 2$ am Anfang schneller ab. Da wir im Folgenden, das obige Beispiel als Grundlage für unterschiedlichen Modifikationen verwenden, setzen wir den Parameter d im Folgenden immer auf den Wert zwei. So sind die unterschiedlichen Ergebnisse leichter vergleichbar sind.

Wir betrachten nun das Konvergenzverhalten für 2 Parameter κ, und λ. Den Optimierungsparameter d haben wir, wie oben beschrieben, auf 2 gesetzt.

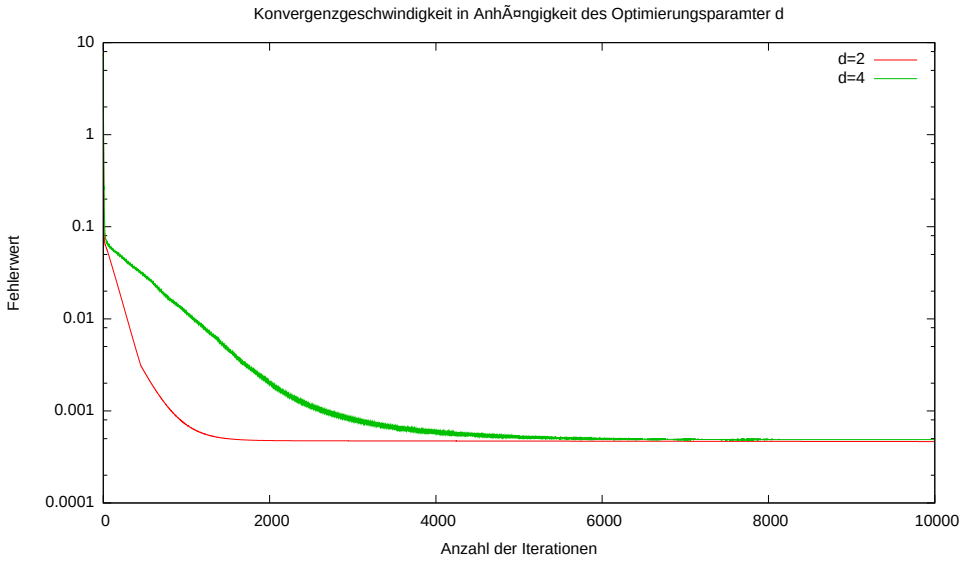

Abbildung 4.19: Verlauf des Kalibrierungsfehlers für unterschiedliche d in normierter Darstellung.

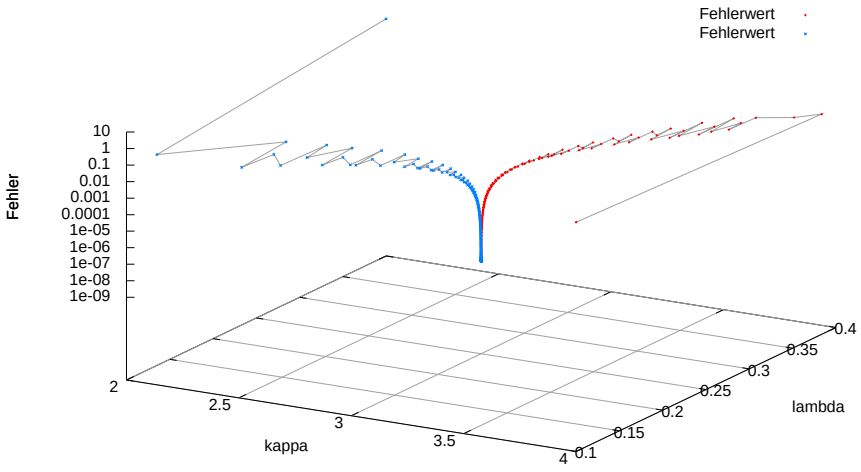

Abbildung 4.20: Verlauf des Kalibrierungsfehlers bezüglich κ und λ.

Zunächst betrachten wir erst eine Optimierung bezüglich der Parameter κ und λ, bevor wir für alle fünf Parameter $\kappa,\ \lambda,\ \nu,\ \rho$ und σ unseres Ausgangsproblem kalibrieren.
In der Abbildung 4.20 ist das Verhalten des Kalibrierungsfehlers für unterschiedliche Startwerte abgebildet. Die x-Komponente entspricht dem aktuellen κ-Wert und die y-Komponente dem λ-Wert. Die z-Komponente gibt den Fehlerwert in einer logarithmischen Darstellung wieder.

Wir sehen, dass beide Kalibrierungsläufe gegen den gleichen Wert konvergieren. Anschließend vergleichen wir in der Grafik 4.21 die Konvergenzgeschwindigkeit für die beiden unterschiedlichen Startwerte.

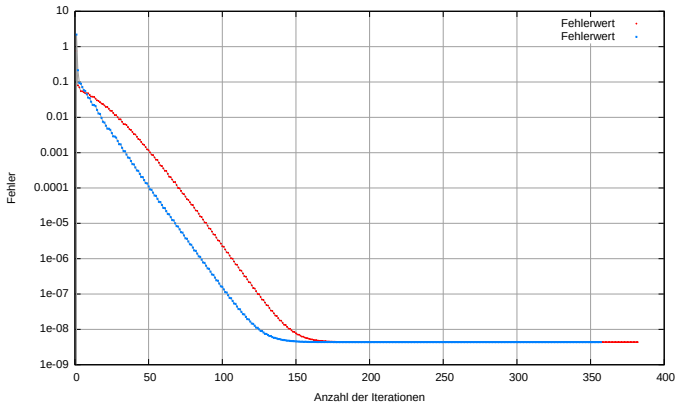

Abbildung 4.21: Konvergenzgeschwindigkeit bezüglich κ und λ.

Wir sehen in der Abbildung 4.21, dass das anfängliche Konvergenzverhalten etwas von der Wahl des Startwertes abhängig ist. Allerdings gleicht sich das asymtotische Verhalten sehr schnell an.
Als Nächstes Beispiel betrachten wir die Kalibrierung nach allen fünf Parametern: κ, λ, ν, ρ und σ.

Der sich daraus ergebende Verlauf des Kalibrierungsfehlers ist in der Abbildung 4.22 abgebildet. Der Parameter d ist wie beschrieben aus Gründen der Vergleichbarkeit ebenfalls 2.

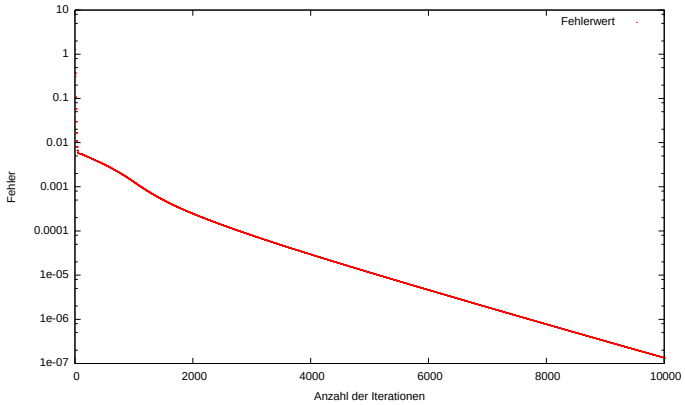

Abbildung 4.22: Verlauf des Kalibrierungsfehlers für eine Kalibrierung nach allen fünf Parametern.

Wir sehen, dass verglichen mit der Optimierung nach 2 Parametern die Anzahl der benötigten Iterationen deutlich, steigt.

4.4 Richtlinien zur Sensitivitätenermittlung in der Finanzmathematik

Hier fassen wir alle gesammelten Erkenntnisse zusammen, die das Anwenden des Automatischen Differenzierens zur Sensitivitäten Ermittlung im Bereich der Finanzmathematik vereinfacht.

4.4.1 Voraussetzungen

Die Voraussetzungen teilen sich in zwei Bereiche auf. Zum einen sind es grundsätzliche Bedingungen für das Automatische Differenzieren, zum Anderen sind es Voraussetzungen, die gerade bei finanzmathematischen Problemen von Relevanz sind.
Zu den elementaren Voraussetzungen gehört, dass das betrachtete Ausgangsproblem differenzierbar bezüglich der zu untersuchenden Eingangsgröße ist. Des Weiteren muss der Quellcode zur Verfügung stehen. Für die Programmiersprache muss zusätzlich noch eine AD-Implementierung vorhanden sein.
Sind diese Bedingungen erfüllt, müssen wir noch Voraussetzungen an den Algorithmus stellen. Folgende Punkte muss die Implementierung des Algorithmus erfüllen:
Er muss in in einer rein reellen Form vorliegen. Wenn der Grundalgorithmus auf einem komplexen Ansatz beruht müssen wir ihn analog zum Vorgehen beim Heston-Modell in eine reelle Form übersetzen und implementieren. Dieser Abschnitt kann relativ aufwendig sein und birgt auch das Risiko, das Fehler entstehen. So bietet es sich an, die kompaktere komplexe Formulierung parallel zu entwickeln und als Referenzimplementierung zu verwenden.
Eine weitere Eigenschaft, die gewährleistet sein muss, ist bei einem iterativen Algorithmus, - zum Beispiel bei einer Monte-Carlo-Simulation - dass wir sicherstellen können, dass der Funktionswert, aber auch die berechnete Ableitung eine vorausgesetzte Güte erfüllen.

4.4.2 Effizienz

Häufig gibt es gerade bei der Berechnung von Finanzprodukten zwei Phasen. In der ersten Phase möchte man das Modell und seine Implementierung testen und seine Sensitivitäten bezüglich unterschiedlichster Einflussfaktoren ermitteln. In dieser Phase ist es vertretbar oder sogar zwingend notwendig, die Sensitivitäten immer mit zu berechnen. In der Anwendungsphase, der zweiten Phase, sind wir nur noch an dem Preis des Produktes interessiert und eventuell noch an einzelnen Sensitivitäten. Allerdings ist zu diesem Zeitpunkt die Laufzeit sehr wichtig, sodass es sich anbietet, nur die benötigten Werte zu berechnen. Dies kann relativ einfach in C++ durch die Verwendung von Templates realisiert werden, indem wir einen generischen Datentyp wählen und ihn bei Bedarf als AD-Datentyp oder als reinen Floating-Datentyp zur Kompilezeit festlegen.
Ein Vorteil dieser Vorgehensweise ist es, dass der Code nur einmal implementiert werden muss und wir durch die Wahl des Datentypen beziehungsweise dessen Bedatung die Funktionalität beeinflussen können.

4.5 Zusammenfassung

Zuletzt fassen wir die Ergebnisse dieses Kapitels zusammen. Wir haben uns auf zwei Methoden zur Bewertung von Optionspreisen beschränkt und uns mit gängigen Implementierungsvarian-

ten für beide Methoden befasst. Bei der ganzen Untersuchung war das Ziel, die Sensitivitäten der Simulationen bestimmen zu können. Die Beurteilung der Sensitivitäten ist von zentraler Bedeutung innerhalb der Finanzmathematik. Gerade durch die Finanzkrise hat die Frage nach der Verlässlichkeit der Methoden und Bewertungsansätze stark an Bedeutung gewonnen.

Hierzu müssen wir zwei Aspekte betrachten. Die erste Frage befasst sich mit der Modellierung des Modells. Ist das verwendete Modell zuverlässig, beschreibt es die Realität genau genug. Oder gibt es Grenzen bei denen das Modell versagt? Diese Frage können wir mit der hier vorgestellten Methode nicht beantworten. Dies war auch nicht das Ziel dieser Arbeit.

Im Rahmen dieser Arbeit haben wir ein Werkzeug untersucht und vorgestellt, mit dem wir auf relativ einfache Weise den Einfluss der einzelnen Eingangsparameter auf das Ergebnis einer Simulation bestimmen können. Diese so ermittelten Sensitivitäten geben uns somit einen Indikator, ob das Modell bezüglich möglicher Schwankungen der Parameter empfindlich oder robust reagiert. Des Weiteren kann man über diese Sensitivitäten auch ermitteln, in welchem Umfang und mit welchem Aufwand wir die Parameter bestimmen, beziehungsweise schätzen müssen, um ein Ergebnis mit einer gewünschten Zuverlässigkeit zu erhalten.

Wir haben uns auf zwei gängige Methoden beschränkt, um das eigentliche Werkzeug - das Automatische Differenzieren - gründlich zu untersuchen. Das Black-Scholes-Modell ist aufgrund seiner Einfachheit noch immer weit verbreitet, obwohl seine Schwächen bekannt sind. Für uns war dieses Modell sehr gut geeignet, da für dieses Modell die Sensitivitäten in geschlossener Form - die sogenannten *Griechen* - bekannt sind und mit einem moderaten Aufwand implementierbar sind. Anhand der *Griechen* hatten wir ein geeignetes Werkzeug unsere Sensitivitäten, die wir über das Automatische Differenzieren berechnet haben, zu überprüfen. Des Weiteren zeigt dies auch die Grenzen auf, die für den Ansatz des Automatischen Differenzierens gelten. Jede Sensitivität, die wir mittels dem AD-Ansatz bestimmen, könnte man auch durch manuelle Berechnung und Implementierung von Hand erhalten. Allerdings ist dieses Vorgehen aufwendig und anfällig für Fehler.
Das zweite betrachtete Modell war das Heston-Modell, welches eine Erweiterung des klassischen Black-Scholes Modell darstellt und die bekannte Schwäche der Konstanten-Volatilität behebt. Im Abschnitt 4.3 haben wir allerdings auch gleich die Schwierigkeiten gesehen, die bei der Berechnung der Optionspreises auftreten. Üblicherweise ist die Funktion $\phi(x)$ komplexwertig definiert. Im Komplexen ist die Ermittlung der Ableitung deutlich schwieriger, wenn sie überhaupt erklärt ist. Daher mussten wir um das Automatische Differenzieren anwenden zu können, den Algorithmus 4.3.1 in eine reelle Formulierung übertragen. Diese Formulierung ist im Algorithmus 4.3.2 abgebildet. Im Algorithmus 4.3.2 sehen wir schon, dass der Code aufgebläht und unübersichtlicher geworden ist, sodass der Aufwand zur Bestimmung der Sensitivitäten erhebliche Ausmaße annehmen würde, wollten wir dies manuell vornehmen.

Generell können wir für das Automatische Differenzieren im Bereich der Finanzmathematik festhalten, dass es ein geeignetes Werkzeug zur Ermittlung der Sensitivitäten ist, welches häufig mit einem vertretbaren Mehraufwand verwendet werden kann. Eine weitere Möglichkeit der Anwendung ist bei der Ermittlung von Modell-Parametern, da mit dem Automatischen Differenzieren die Berechnung der Ableitungswerte ermöglicht wird und wir somit ableitungsbasierte Optimierungs- beziehungsweise Suchverfahren einsetzen können. Eine gründliche analytische und theoretische Untersuchung eines neuen Modells zur Bestimmung der Modellgrenzen

nimmt es uns aber nicht ab, auch wenn die zur Verfügung stehenden Sensitivitäten eventuell helfen die Grenzen leichter zu ermitteln.
Die Erkenntnisse zur einfachen Bestimmung der Sensitivitäten in finanzmathematischen Anwendungen, die wir im Rahmen dieser Arbeit ermittelten, sind im Abschnitt 4.4 *Richtlinien zur Sensitivitätenermittlung in der Finanzmathematik* zusammengefasst.

5 Zusammenfassung

Im letzten Kapitel dieser Arbeit fassen wir die Ergebnisse dieser Arbeit zusammen und analysieren die Ergbenisse in einem gesammtheitlichen Kontext. Zuerst fassen wir die Resultate der einzelnen Kapiteln noch einmal zusammen, anschließend verknüpfen wir die Resultate zu einem Gesamtergebniss und stellen die neuen Erkenntnisse dieser Arbeit zusammen. Zuletzt geben wir einen Ausblick auf mögliche weitere Forschungs- und Entwicklungsmöglichkeiten.

5.1 Automatisches Differenzieren

Im Kapitel 2 haben wir die Grundlagen des Automatisches Differenzieren beschrieben und sind auf die unterschiedlichen Varianten vom AD eingegangen und haben die auftretenden Problematiken aus technischer und numerischer Sicht, sowie im Bereich der Effizienz eingehend betrachtet und analysiert. Zudem haben wir die im Rahmen dieser Arbeit entstandenen AD-Tools und Libraries vorgestellt und erläutert.
Mit GAuDi ist ein Framework im Rahmen der Arbeit entstanden, das zwei unterschiedliche Bereiche umfasst: Eine generelle Idee zur Umsetzung eines Programmiersprachen unabhängigen Source-Transformtion-Tools, als auch eine C++ Bibliothek, die eine integrierte Debbuging-Einheit zum Auffinden von numerischen Instabilitäten beinhaltet. Die Besonderheit der Debbunging-Funktionalität ist, dass sie ohne Modifikation per Compiler-Option zugeschaltet werden kann und bei Inaktivität keinen zusätzlichen Aufwand erzeugt.

5.2 Strömungsdynamik

Die Strömungsdynamik ist ein relevantes und anspruchsvolles Thema aus physikalischer und numerischer Sicht. In diesem Kapitel haben wir deshalb - ausgehend von der Physik - die wesentlichsten Aspekte der Strömungsmechanik zusammengestellt und die numerische Herangehensweisen erläutert, bevor wir auf die - aus Sicht dieser Arbeit - relevanten Fragestellungen, der Anwendung des Automatisches Differenzierens in numerischen Strömungssimulationen, eingegangen sind. Innerhalb dieses Bereichs haben wir zwei Implementierungen von CFD-Programmen verwendet eine 2D-Software *Caffa* und eine 2D/3D-Software *Comet*.
Bei der Verwendung des Automatischen Differenzierens in Caffa können wir zusammenfassend feststellen, dass die Anwendung von AD ohne Einschränkung umsetzbar ist. Bei der Kombination von Comet mit dem Automatischen Differenzieren ist ein abschließendes Urteil nicht so eindeutig. Wird ausschließlich ein starres Gitter betrachten, ist die Umsetzung möglich. Bei bewegten Gittern ist aufgrund der internen Struktur und der verwendeten Berechnungsmethoden von Comet eine Anwendung nicht möglich, da numerische Instabilitäten bei den Ableitungswerten auftreten.

5.3 Finanzmathematik

Analog zu den anderen Kapiteln haben wir im Bereich der Finanzmathematik (Kapitel 4) eine kurze Einführung in die für uns relevanten Methoden und Modelle, sowie ihre numerische Umsetzung, gegeben. Wir beschränken uns auf die Bewertung von Optionen. Dazu führen wir den Begriff der Option ein und befassen uns näher mit der numerischen Berechnung von Integralen, da dies für uns eine entscheidende Rolle spielt. Im Anschluss befassen wir uns mit dem am weitesten verbreiteten Modell zur Bewertung von Optionen, dem Black-Scholes-Modell und den Anwendungsmöglichkeiten des Automatischen Differenzierens in der Optionspreisberechnung. Zuletzt übertragen wir die gewonnenen Erkenntnisse auf das komplexere Heston-Modell, welches die größten Schwächen des Black-Scholes-Modells behebt und betrachten erneut den Einsatz des Automatischen Differenzierens im Senisitiväts- und Kalbibrierungskontext.
Abschließend hat sich gezeigt, dass die Verwendung vom AD im Bereich der Finanzmathematik sehr gut umsetzbar ist. Zum Teil muss man die gebräuchliche Formulierung in Complexer Notation in eine reelle Formulierung, aus Gründen der Anwendbarkeit der AD-Tools, umformen. Dieser zusätzliche Aufwand lohnt sich allerdings meist, schon da die Simulation in einer rein reellen Formulierung leichter von Hand optimiert werden kann. Zudem ist aufgrund der besser optimierten Umsetzung von Reellen-Datentypen innerhalb des Compilers die Laufzeit aufgrund der möglichen Optimierungen des Compilers in der Regel besser.

5.4 Gesamtergebnisse

Die Gesamtergebnisse dieser Arbeit gliedern sich in drei Bereiche:
Die elementaren Erkenntnisse, die sich im Bereich des Automatischen Differenzierens ergeben haben. Im Speziellen sind hier die in dieser Form nicht vorhandenen, beziehungsweise zusammengetragenen Erkenntnisse der auftretenden Problematiken, sowie die Entwicklung einfacher und trotzdem effizienter Tools, beziehungsweise Librarys, genannt.
Bei der Betrachtung der Strömungsmechanik liefern detaillierten Erkenntnisse der Anwendung des Automatischen Differenzierens auf bestehenden Code unter Verwendung verschiedener Ansätze neue Erkenntnisse zur erfolgreichen Integration vom Automatischen Differenzieren in Strömungssimulationen. Die Erkenntnisse sind allerdings auf eine Vielzahl von unterschiedlichen Simulationen in anderen Bereichen übertragbar.
Die Anwendung des Automatischen Differnzierens auf Problemstellungen aus dem finanzmathematischen Bereich im Rahmen dieser Arbeit zeigt die Flexibilät der bisher meist im Zusammenhang mit technisch-naturwissenschaftlich angewandten Methode. Die Kalibrierung des Heston-Modells in dieser Form wurde bisher noch nicht in dieser Art und Weise betrachtet.

5.5 Ausblick

In diesem Abschnitt fassen wir interessante Fragestellungen, die beim Erstellen dieser Arbeit aufgeworfen wurden, zusammen. Die Beantwortung dieser Fragen hätte allerdings den Rahmen dieser Promotionsarbeit gesprengt. Die Gliederung des folgenden Abschnitts ist analog zum restlichen Aufbau der vorliegenden Arbeit.

5.5.1 Automatisches Differenzieren

Ein möglicher Ausblick, damit die Verbreitung des Automatischen Differenzierens zunimmt, ist die Weiterentwicklung der betrachteten Tools sowie die Integration der Bibliotheken in andere, bereits bestehende Bibliotheken mit gezielten Erweiterungen. So kann eine Effizienzsteigerung - sowohl aus Sicht der Recheneffizienz, als auch aus Sicht des Anwenders - erreicht werden. Andere denkbare Weiterentwicklungen sind die Entwicklung von noch detaillierteren und effizienteren Methoden zur Erkennung von Singularitäten aus numerischer Sicht zur Beschleunigung der Laufzeit der entsprechenden Simulation, sowie die Entwicklung von Mechanismen zur Steigerung der Robustheit der Ableitungswerte beim Auftreten numerischer Singularitäten oder in deren unmittelbarer Umgebung.

5.5.2 Strömungsdynamik

Bei der Betrachtung des Automatischen Differenzierens im Bereich der Strömungsdynamik haben wir einige neue Ansätze aufgeworfen. Zum Beispiel die Entwicklung eines numerischen Strömungslösers der gezielt unter den Aspekten der Verwendung von bewegten Gittern und dem Automatischen Differenzieren entwickelt wird. Hierzu müsste man zunächst eine geeignete Datenstruktur für die Gittergeometrie und das Bewegen des Gitters entwickeln, die für das Automatische Differenzieren flexibel genug ist und auf der anderen Seite den Effizienzansprüchen für Speicher- und Recheneffizienz genügt, speziell für den Fall, dass die Funktionalität des Automatischen Differenzierens nicht benötigt wird. Dazu müsste man Stabilitätsbetrachtungen für die Simulation, sowohl für die primären Größen als auch die auftretenden sekundären Größen (Ableitungsgrößen), bei der Wahl der Lösungsalgorithmen berücksichtigen.

5.5.3 Finanzmathematik

Die Verwendung des Automatischen Differenzierens ist gerade bei nicht auf Monte-Carlo basierenden Methoden im Bereich der Finanzmathematik noch nicht sehr stark vertreten. Daraus folgt, dass das Feld für Ausblicke hier entsprechend groß ist. Man kann die Einsatzmöglichkeiten und Untersuchungen für das Automatische Differenzieren auf die meisten Berechnungen von finanzmathematischen Produkten und Fragestellungen ausweiten und generelle Erkenntnisse für den Einsatz vom AD in der Finanzmathematik aufstellen.

Lebenslauf

Vorname:		**Ralf**
Name:		**Leidenberger**
Schulbildung:	1990-1994	**Grundschule**,
	1994-1999	**Hauptschule**,
	1999-2000	**Werkrealschule**,
	2000-2003	**Technisches Gymnasium**,
Studium:	2003-2008	**Diplom Mathemtik** (Nebenfach Informatik), Universität Ulm
Promotion:	2008-2011	**Promotionsstipendiat**, Graduiertenkolleg 1100, Universität Ulm
Wissenschafliche Tätigkeiten:	2005-2008	**Forschungsstudent**, Graduiertenkolleg 1100, Universität Ulm
	2007-2008	**Tutor**, Numerik 1b, Universität Ulm
	2009	**Übungsleiter**, Einführung in die Strömungsmechanik, Universität Ulm
	2010	**Übungsleiter**, Einführung in die Strömungsmechanik, Universität Ulm
	2011	**Übungsleiter**, Scientific Computing, Universität Ulm
	2012	**Dozent**, Fahrzeugsystemtechnik, HTWG Konstanz
	2013	**Dozent**, Fahrzeugsystemtechnik, HTWG Konstanz

Der Lebenslauf ist in der Online-Version aus Gründen des Datenschutzes nur in einer gekürzten Version verfügbar.

Bezeichnungen

AD	Automatisches Differenzieren oder auch Algorithmisches Differenzieren (engl. *automatic differentiation*)
CFD	Computational Fluid Dynamics
CFG	Control Flow Graph
$\cdot^D$	symbolisiert die durch AD bestimmte zugehörige abgeleitete Variable
$\mathbf{div}\,\cdot$	Repräsentiert die Divergenz $\mathbf{div}\,\cdot := \sum_{i=1}^{n} \frac{\partial \cdot_i}{\partial x_i}$
elementare Operationen	gleichbedeutend mit *elementare Rechenoperationen*.
elementare Rechenoperationen	Dies beschreib alle nativ in einer Programmiersprache verfügbaren Rechenoperationen bzw. mathematische Funktionen.gleichbedeutend mit
FDM	Finite Differenzen Methode siehe *Finite-Differenzen-Methode* .
FEM	Finite Elemente Methoden siehe *Finite-Elemente-Methode* .
FVM	Finit Volumen Methoden siehe *Finite-Volumen-Methoden* .
GAuDi-STT	GNU Automatic Differentiation - Source Transformation Tool
$\mathbf{j}$	Stromdichte
κ (Strömungsmechanik)	Die Wärmeleitfähigkeit in $\frac{kg \cdot m}{s^3 \cdot K}$.
Ma	Die Mach-Zahl, ist die dimensionslose KennzahlMa $= \frac{v}{c}$, wobei c die Schallgeschwindigkeit beschreibt.
∇	$\nabla := (\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3})$im $\mathbb{R}^3$ mit kartesischen Koordinaten.
∇f	$\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3})$im $\mathbb{R}^3$ mit kartesischen Koordinaten und $f \in \mathcal{C}^1(\mathbb{R}^3)$.
$\nabla \cdot f$	$\nabla \cdot f = \mathbf{div}\, f$ im $\mathbb{R}^3$ mit kartesischen Koordinaten und $f \in \mathcal{C}^1(\mathbb{R}^3)$.
Pr	Die Prandtl-Zahl, ist die dimenensionslose KennzahlPr $= \frac{\eta c_p}{\kappa}$.
$\mathbb{R}^+$	$\mathbb{R}^+ := \{x \in \mathbb{R} : x > 0\}$
$\mathbb{R}_0^+$	$\mathbb{R}^+ := \{x \in \mathbb{R} : x \geq 0\}$
$\mathbb{R}^-$	$\mathbb{R}^+ := \{x \in \mathbb{R} : x < 0\}$
$\mathbb{R}_0^-$	$\mathbb{R}^+ := \{x \in \mathbb{R} : x \leq 0\}$
Re	Die Reynolds-Zahl, ist die dimenensionslose KennzahlRe $= \frac{\rho \cdot v \cdot d}{\eta}$.
V_M	Kontrollvolumen mit fester Masse.
V_V	Kontrollvolumen mit festem Volumen.

Literaturverzeichnis

[ADC99] AUBERT, Pierre ; DI CÉSARÉ, Nicolas: sl::Fad<A,O>. (1999)

[Alt02] ALT, Walter: *Nichtlineare Optimierung*. 1. Vieweg, 2002

[Bas08] BASTIAN, Peter: *Numerische Lösung partieller Differentialgleichungen*. Universität Stuttgart, 2008

[Bir11] BIRNTHALER, Thomas ; OSTC GMBH (Hrsg.): *Reguläre Ausdrücke: Beschreibung und Anwendung*. 1.39. OSTC GmbH, 05 2011

[BO04] BARTH, Timothy ; OHLBERGER, Mario: Finite volume methods: foundation and analysis. In: *Encyclopedia of Computational Mechanics* (2004)

[Bra00] BRAESS, Dietrich: *Finite Elemente*. 3. Springer, 2000

[BS73] BLACK, Fischer ; SCHOLES, Myron: The Pricing of Options and Corporate Liabilities. In: *Journal of Political Economy* 81 (1973), S. 503–504

[BS96] BENDTSEN, Claus ; STAUNING, Ole: FADBAD, a flexible C++ package for automatic differentiation / Technical University of Denmark. 1996. – Forschungsbericht

[BSD93] BSD (Hrsg.): *Manpage: tail*. BSD, 06 1993

[BSD99] BSD (Hrsg.): *Manpage: rm*. BSD, 01 1999

[BSD03] BSD (Hrsg.): *Manpage: echo*. BSD, 04 2003

[BSD04] BSD (Hrsg.): *Manpage: cat*. BSD, 03 2004

[BSD05a] BSD (Hrsg.): *Manpage: ps — process status*. BSD, 03 2005

[BSD05b] BSD (Hrsg.): *Manpage: wc*. BSD, 02 2005

[Buc14] BUCKINGHAM, Edgar: On physically similar systems; illustrations of the use of dimensional equations. In: *Physical Review* 4 (1914), Nr. 4, S. 345–376

[Buc15] BUCKINGHAM, Edgar: The principle of similitude. In: *Nature* 96 (1915), S. 396–397

[Caf98] CAFLISCH, Russel E.: Monte Carlo and quasi-Monte Carlo methods. In: *Acta Numerica* (1998), S. 1–49

[DB02] DEUFELHARD, Peter ; BORNEMANN, Folkmar: *Numerische Mathematik II*. 2. de Gruyter, 2002

[Ebe10] EBERLE, Andreas: *Einführung in die Wahrscheinlichkeitstheorie*. Universität Bonn, 2010

[Eul68] EULER, Leonhard: *Institutiones Calculi Integralis*. Petropoli, 1768

[Fis05] FISCHER, Herbert: *Algorithmisches Differenzieren*. Technische Universität München, 2005

[For05] FORSTER, Otto: *Analysis 2*. 6. Vieweg, 2005

[FP08] FERZIGER, Joel H. ; PERIC, Milovan: *Numerische Strömungsmechanik*. 1. Springer, 2008. – 509 S.

[Fre02] FREE SOFTWARE FOUNDATION, INC. (Hrsg.): *Manpage: gdb - The GNU Debugger*. Free Software Foundation, Inc., 05 2002

[FS05] FREY, Rüdiger ; SCHMIDT, Thorsten: *Vorlesungsskript Finanzmathematik I*. 1. Universität Leipzig, 2005

[GDN95] GRIEBEL, Michael ; DORNSEIFER, Thomas ; NEUNHOEFFER, Tilman: *Numerische Simulation in der Strömungsmechanik, eine praxisorientierte Einführung*. Braunschweig : Vieweg, 1995

[Ger06] GERIG, Vera: *Parameterschätzung stochastischer Prozesse aus der Finanzwelt mittels (Dünngitter-)Histogramm- Matching-Verfahren*, Rheinischen Friedrich-Wilhelms-Universität Bonn, Diplomarbeit, 2006

[GJU96] GRIEWANK, Andreas ; JUEDES, David ; UTKE, Jean: Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. In: *ACM Transactions on Mathematical Software* 22 (1996), S. 131–167

[GPF08] GRIMBERG, Gerard ; PAULS, Walter ; FRISCH, Uriel: Genesis of d'Alembert's paradox and analytical elaboration of the drag problem. In: *Physica D* 237 (2008), S. 1878–1886

[Grü09] GRÜNE, Lars: *Numerische Methoden der Finanzmathematik*. 2009

[GW08] GRIEWANK, Andreas ; WALTHER, Andrea: *Evaluating Derivatives*. 2. Siam, 2008. – 438 S.

[Her08] HERWIG, Heinz: *Strömungsmechanik*. 1. Vieweg-Teubner, 2008

[Hes93] HESTON, Steven L.: A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. In: *The Review of Financial Studies* 6 (1993), S. 327–343

[HP04] HASCOËT, Laurent ; PASCUAL, Valérie ; INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (Hrsg.): *TAPENADE 2.1 user's guide*. INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 2004. – tapenade21

[Ins01] INSTITUTE OF COMPUTATIONAL CONTINUUM MECHANICS GMBH (Hrsg.): *Comet*. 2.00. Bramfelder Straße 164 D-22 305 Hamburg: Institute of Computational Continuum Mechanics GmbH, 03 2001

[KA00] KNABNER, Peter ; ANGERMANN, Lutz: *Numerik partieller Differentialgleichungen*. 1. Springer, 2000

[Kae03] KAENEL, Roland von: *Large-Eddy Simulation of Compressible Flows using the Finite-Volume MethodLarge-Eddy Simulation of Compressible Flows using the Finite-Volume Methods*, ETH, Diss., 2003

[Kan00] KANZOW, Christian: *Numerik linearer Gleichungssysteme*. 1. Springer, 2000

[Kar09] KARLSSON, Patrik: *The Heston Model*, Lund University, Bachelorarbeit, 2009

[Kie04] KIESEL, Rüdiger: *Financial Mathematics I*. Universität Ulm, 2004

[Kö00] KÖNIGSBERGER, Konrad: *Analysis 1*. Bd. 1. 6. Springer, 2000

[Kö04] KÖNIGSBERGER, Konrad: *Analysis 2*. Bd. 2. 5. Springer, 2004

[Lau97] LAURIE, Dirk P.: Calculation of Gauss-Kronrod Quadrature Rules. In: *Mathematics of Computation* 66 (1997), S. 1133–1146

[Lei97] LEIPPOLD, Markus: Numerische Methoden in der Optionspreistheorie: Monte Carlo und Quasi-Monte Carlo Methoden. In: *Finazmarkt und Portfolio Management* 2 (1997), S. 179–196

[Lei07] LEIDENBERGER, Ralf: *Automatic differentiation in flow simulation*, Ulm University, Diplomarbeit, 2007

[Leo79] LEONARD, Brian P.: A stable and accurate convective modeling procedure based on quardratic upstream interpolation. In: *Computer Methods in Applied Mechanics and Engineering* 19 (1979), S. 59–98

[Lia99] LIANG, Sheng: *The Java Native Interface*. Addison-Wesley, 1999

[LU11] LEIDENBERGER, Ralf ; URBAN, Karsten: Automatic Differentiation for the Optimization of a Ship Propulsion and Steering System: A proof of concept. In: *Journal of Global Optimization* 49 (2011), S. 497–504

[MB06] MARTIN BÜCKER, Paul Hovland Uwe Naumann Boyana N. George Corliss C. George Corliss (Hrsg.): *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006

[McK10] MCKINNEY, Gordon: *Regular Expression Quick Reference v1.2*. 1.2, 2010

[Mei05] MEISTER, Andreas: *Numerik linearer Gleichungssysteme*. 2. Vieweg, 2005. – 252 S.

[Mer73] MERTON, Robert C.: Theory of Rational Option Pricing. In: *The Bell Journal of Economics and Management Science* 4 (1973), S. 141–183

[Met87] METROPOLIS, Nicholas: The Beginning of the Monte Carlo Methods. In: *Los Alamos Science Special Issue* (1987), S. 125–130

[MN05] MIKHAILOV, Sergei ; NÖGEL, Ulrich: Heston's Stochastic Volatility Model Implementation, Calibration and Some Extensions. In: *Wilmott magazine* (2005), S. 74–79

[Moo05] MOODLEY, Nimalin: *The Heston Model: A Practical Approach*, University of the Witwatersrand, Johannesburg, Bachelorarbeit, 2005

[Nel] NELSON, Philip A.: *Manpage: bc*. 1.06

[Nov03] NOVILLO, Diego: Tree SSA a new Optimization Infrastructure for GCC. In: *GCC Developers Summit* (2003), S. 181–194

[NR47] NEUMANN, John von ; RICHTMYER, Robert D.: Statistical Methods in Neutron Diffusion. In: *LAMS* 551 (1947), S. 17–26

[NRC88] *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988

[OjBD06] OERTEL JR., Herbert ; BOHLE, Martin ; DOHRMANN, Ulrich: *Strömungsmechanik*. 4. Vieweg, 2006

[Ran99] RANNACHER, Rolf: *Finite Element Methods for the Incompressible Navier-Stokes Equations*. University of Heidelberg, 1999

[RMB10] RONALD M. BARRON, Bashar Z.: A Finite Difference Calculation Procedure for the Navier-Stokes Equations on a Staggered Curvilinear Grid. In: *World Academy of Science, Engineering and Technology* 71 (2010), S. 849–852

[SA07] SPURK, Joseph H. ; AKSEL, Nuri: *Strömungslehre*. 7. Springer, 2007

[SB00] STOER, Josef ; BULRISCH, Roland: *Numerische Mathematik 2*. Bd. 2. 4. Springer, 2000

[Sch01] SCHÖNING, Uwe: *Algorithmik*. 1. Spektrum, 2001

[Seg11] SEGAL, Ir. A.: *Finite element methods for the incompressible Navier-Stokes equations*. Delft University of Technology, 2011

[SG05] SCHLICHTING, Hermann ; GERSTEN, Klaus: *Grenzschicht-Theorie*. 10. Springer, 2005

[SGDC08] STALLMAN, Richard M. ; GCC DEVELOPER COMMUNITY the ; FREE SOFTWARE FOUNDATION (Hrsg.): *GNU Compiler Collection Internals*. Free Software Foundation, 2008. `http://gcc.gnu.org/onlinedocs/gccint/`. – For gcc version 4.5.0 (pre-release)

[SGDC10] STALLMAN, Richard M. ; GCC DEVELOPER COMMUNITY the ; GNU PRESS (Hrsg.): *Using the GNU Compiler Collection*. For gcc version 4.6.1. GNU Press, 2010

[SJGS99] SREEDHAR, Vugranam C. ; JU, Roy Dz-Ching ; GILLIES, David M. ; SANTHANAM, Vatsa: Translating Out of Static Single Assignment Form. In: *Lecture Notes in Computer Science* 1694 (1999), S. 849–849

[Sla05] SLAWIG, Thomas: Algorithmisches Differenzieren in Simulation und Optimalsteuerung von Differentialgleichungen / Technische Universität Berlin. 2005. – Forschungsbericht

[Spi05] SPINELLIS, Diomidis D.: *Manpage: sed — stream editor*, 05 2005. – BSD General Commands Manual

[SS86] SAAD, Youcef ; SCHULTZ, Martin H.: GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. In: *SIAM Journal on Scientific and Statistical Computing* 7 (1986), S. 856 – 869

[Sto45] STOKES, George G.: On the theories of the internal friction of fluids in motion. In: *Transactions of the Cambridge Philosophical Society* 8 (1845), S. 287–305

[Sto68] STONE, Herbert L.: Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations. In: *SIAM Journal on Numerical Analysis* 5 (1968), 9, Nr. 3, S. 530–558

[Sto02] STOER, Josef: *Numerische Mathematik 1*. Bd. 1. 8. Springer, 2002

[Str05] STROUSTRUP, Bjarne: *The C++ Programming Language*. 3. Addison-Wesley, 2005

[tea11] TEAM, The gfortran ; FREE SOFTWARE FOUNDATION (Hrsg.): *Using GNU Fortran*. For gcc version 4.7.0 (pre-release). Free Software Foundation, 2011

[Thu02] THURSO, Jens: *Numerische Simulation des Grenzschichtverhalten in Turbinengittern unter periodisch-instationären Strömungsbedingungen*, TU-Darmstadt, Diss., 2002

[Tru05] TRUSKETT, Iain ; FORD & MASON LTD. (Hrsg.): *Perl Regular Expression Quick Reference Card*. 0.1. Ford & Mason Ltd., 2005

[Tru08a] TRUCKENBRODT, Erich: *Fluidmechanik*. Bd. 1. Springer, 2008. – Band 1: Grundlagen und elementare Strömungsvorgänge dichtebeständiger Fluide

[Tru08b] TRUCKENBRODT, Erich: *Fluidmechanik*. Bd. 2. Springer, 2008. – Band 2: Elementare Strömungsvorgänge dichteveränderlicher Fluide sowie Potential- und Grenzschichtströmungen

[UNL11] UTKE, Jean ; NAUMANN, Uwe ; LYONS, Andrew: *OpenAD/F: User Manual*, 2011

[Urb06] URBAN, Karsten: *Numerik 1b*. Universität Ulm, 2006

[Wec05] WECKESSER, Warren: More About Nondimensionalization / Department of Mathematics, Colgate University. 2005. – Math 312 Lectures 6 and 7

[Wic00] WICK, Matthias M.: *Numerische Lösung des parabolischen Hindernisproblems mit einer inneren Punkte Methode*, Universität Basel, Diss., 2000

[ZB08] ZIEREP, Jürgen ; BÜHLER, Karl: *Grundzüge der Strömungslehre*. 7. Teubner, 2008

[ZSMM06] ZHANG, Kenn K. Q. ; SHOTORBAN, Babak ; MINKOWYCZ, W. J. ; MASHAYEK, Farzad: A Compact Finite Difference Method on Staggered Grid for Navier-Stokes Flows. In: *International Journal for Numerical Methods in Fluids* 52 (2006), S. 867–881

Namen- und Sachverzeichnis

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe; die aus fremden Quellen direkt oder indirek übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Zudem wurde die Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis beachtet.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ravensburg, den 10. Dezember 2013 _____

(Unterschrift)