Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstrasse 38
D-70569 Stuttgart

Bachelorarbeit Nr. 303

Die Kombinationstechnik als Zeitintegrator in Parareal

Anna Kulischkin

Studiengang: Informatik

Prüfer/in: Jun.-Prof. Dr. Dirk Pflüger
Betreuer/in: Dr. Stefan Zimmer

Beginn am: 1. Februar 2016
Beendet am: 2. August 2016

CR-Nummer: F.2.2 G.1.1 G.1.8 G.1.0 G.1.10 G.1.m

Abstract

Um die Rechenzeit beim Lösen zeitabhängiger partieller Differentialgleichungen zu reduzieren, werden Parallelisierungsmethoden verwendet. Durch die Zeitparallelisierung können verschiedene Zeitabschnitte parallel berechnet werden. Dazu wird das Parareal-Verfahren eingesetzt, bei dem grobe und feine Zeitintegratoren eine Zeitparallelisierung ermöglichen. Das Ziel der vorliegenden Bachelorarbeit war es, die Kombinationstechnik als groben Zeitintegrator für die Parareal-Implementierung zu untersuchen. Bei der Kombinationstechnik werden Linearkombinationen von Lösungen auf verschieden großen Gittern gebildet. Dafür wurde die Kombinationstechnik für das Modellproblem der Wärmeleitungsgleichung in der Programmiersprache Python implementiert. Als feiner Zeitintegrator wurde die Lösung auf einem vollen Gitter verwendet. Das dafür aufgestellte lineare Gleichungssystem wurde mit dem impliziten Euler-Verfahren gelöst. Zum Auswerten der Ergebnisse wurde die Rechenzeit und die Anzahl der Iterationen sowie der dabei entstehende Fehler betrachtet. Ein wichtiger Punkt dabei war die Konvergenzgrenze zu berücksichtigen, die einen Einfluss auf die Anzahl der Iterationen und somit auf die Rechenzeit hat.

Inhaltsverzeichnis

1 Einleitung **7**

2 Parareal-Algorithmus **9**
2.1 Notation . . . 9
2.2 Einführung . . . 9
2.3 Algorithmus . . . 10
2.4 Programmierung . . . 11
2.4.1 Klassenübersicht . . . 11
2.4.2 Interface Beschreibung . . . 13

3 Kombinationstechnik **15**
3.1 Notation . . . 15
3.2 Einführung . . . 16
3.3 Kombinationstechnik mit Ausdünnung . . . 16

4 Kombinationstechnik als Zeitintegrator **19**
4.1 Einführung . . . 19
4.1.1 Wärmeleitungsgleichung . . . 19
4.1.2 implizite Euler-Verfahren . . . 21
4.1.3 Bilineare Interpolation . . . 21
4.2 Programmierung . . . 22
4.2.1 Klassenübersicht . . . 23
4.2.2 Interface Beschreibung . . . 23
4.3 Experimente . . . 25

5 Ausblick **31**

1 Einleitung

Die Idee, eine partielle Differentialgleichung zeitparallel zu lösen, hatte Nievergelt im Jahre 1964. Er war der Erste, der eine Zeitzerlegung für die parallele Lösung von Evaluationsproblemen berücksichtigte. Dafür zerlegte er das Intervall in mehrere Zeitabschnitte, auf denen die Berechnung parallel durchgeführt werden konnte, wobei nur die Vergangenheitswerte des vorangegangenen Zeitabschnitts in die Berechnung eingehen. Nievergelt präsentierte eine Methode zur Parallelisierung von numerischen Integrationen gewöhnlicher Differentialgleichungen. Seine Methode und die Weiterentwicklung der Zeitparallelisierung wird in [2] vorgestellt.

Im Jahr 2001 erfanden Lions, Maday und Turinie den Parareal-Algorithmus. Der Algorithmus baut auf der Idee der Intervallzerlegung von Nievergelt auf. Die Grundidee ist einen groben Zeitschritt zu wählen und die Lösung zuerst annähernd zu berechnen. Anschließend wird die Lösung durch Verkleinerung des Zeitschritts verfeinert. Die Verfeinerung der Lösung kann durch die Intervallzerlegung in allen Zeitabschnitten parallel ausgeführt werden. Der Parareal-Algorithmus löst Differentialgleichungen zeitparallel und zählt zu den am häufigsten untersuchten Algorithmen, die in der Zeit parallel durchgeführt werden. In der Arbeit von Lions, Maday und Turinie [5] wird der Algorithmus genauer beschrieben. 2015 beschäftigte sich Dr. Martin Schreiber an der Universität Exeter mit dem Parareal-Algorithmus. In seiner Arbeit [6] stellt er den Algorithmus und seine Parareal-Implementierung (pypint) in der Programmiersprache Python vor.

Durch eine Zeitdiskretisierung der Differentialgleichungen wird das Problem auf äquidistandem Gitter dargestellt. Um die Genauigkeit zu erhöhen wird das Diskretisierungsgitter verfeinert, dadurch kann das Gleichungssystem sehr groß werden. Mit der Größe des Gleichungssystems wächst aber auch der Rechenaufwand sowie der benötigte Speicherplatz. Der verfügbare Speicher von Computern ist jedoch beschränkt. Griebel, Schneider und Zenger entwickelten die Kombinationstechnik [3], um dieses Problem zu reduzieren. Bei der Kombinationstechnik wird eine Linearkombinationen diskreter Lösungen des Problems auf unterschiedlich großen rechteckigen Gittern gebildet. Die Kombinationslösung stimmt zwar mit der Lösung auf dünnen Gittern nicht überein, der Fehler besitzt jedoch dieselbe Ordnung. Mit dieser Technik kann der Rechenaufwand und der Speicherplatz deutlich verringert werden. Christoph Kranz untersucht in seiner Dissertation [4] die Kombinationstechnik bei der numerischen Strömungssimulation, unter anderem beschreibt er die Kombinationstechnik auf dünnen Gittern.

Die Kombinationstechnik eignet sich sehr gut für parallele Berechnungen. Diese Bachelorarbeit verfolgt das Ziel, die Kombinationstechnik als groben Zeitintegrator im Parareal-Algorithmus zu untersuchen. Als Grundlage für die Parareal-Implementierung dient die Arbeit von Martin Schreiber [6]. Die Kombinationstechnik wird für das Modellproblem der Wärmeleitungsgleichung implementiert und als grober Zeitintegrator in pypint integriert. Im Rahmen dieser Bachelorarbeit wurde die Rechenzeit und die benötigten Iterationsschritte sowie der dabei entstehende Fehler untersucht und ausgwewertet.

Diese Bachelorarbeit ist folgendermaßen gegliedert: In Kapitel 2 wird eine kurze Einführung

in den Parareal-Algorithmus gegeben. Anhand eines Beispiels wird der Algorithmus Schritt für Schritt erklärt. In Abschnitt 2.4 ist eine Übersicht der Parareal-Implementierung gegeben. Hier werden nur die für diese Bachelorarbeit wichtigen Abschnitte vorgestellt. Die Arbeit von Martin Schreiber [6] bildet die Grundlage dieses Kapitels. In Kapitel 3 werden grundlegende Formeln für die Kombinationstechnik vorgestellt. Die Kombinationstechnik mit Ausdünnung wird in Abschnitt 3.3 beschrieben und der dafür verwendete Ausdünnungsparameter eingeführt. Dieses Kapitel basiert auf [4]. Das darauffolgende Kapitel (Kapitel 4 ist das Kernstück der Bachelorarbeit. Es enthält die relevanten Ergebnisse, der durchgeführten Tests. Das Modellproblem der Wärmeleitungsgleichung sowie das implizite Euler-Verfahren zum Lösen der partiellen Differentialgleichungen werden erläutert. Die für die Kombinationstechnik verwendete Interpolation wird in Abschnitt 4.1.3 beschrieben. Die Erweiterung und die Beschreibung der Parareal-Implementierung folgen in Abschnitt 4.2. Anschließend werden die Ergebnisse durchgeführter Experimente vorgestellt und miteinander verglichen sowie ausgewertet. Ein abschließender Ausblick schafft Anknüpfungspunkte für weitere Untersuchungen.

2 Parareal-Algorithmus

Um die Rechenzeit beim Lösen von partiellen Differentialgleichungen zu minimieren, wird nicht nur die Zerlegung des Raumgebiets eingesetzt sondern auch eine Methode der Zeitparallelisierung. Eines der bekanntesten Methoden ist der Parareal-Algorithmus. Bei diesem Verfahren werden grobe und feine Zeitintegratoren eingesetzt, um mehrere Zeitabschnitte parallel berechnen zu können. In diesem Kapitel wird der Algorithmus erläutert. Die dazu verwendeten Notationen sind im nächsten Abschnitt aufgeführt. Anschließend wird die Parareal-Implementierung erläutert.

2.1 Notation

Nachfolgend sind die verwendeten Notationen in dieser Arbeit aufgelistet.

N	Anzahl der groben Zeitintervalle
$y(t)$	Lösung zum Zeitpunkt t
y_0	Anfangswert zum Zeitpunkt $t = 0$
$y_{k,i}$	korregierter grober Wert des Segments k für die Iteration i
$y^G_{k,i}$	Lösung des groben Integrators am Ende des Segments k für Iteration i
$y^F_{k,i}$	Lösung des feinen Integrators am Ende des Segments k für Iteration i
$G_{\Delta T}(y^G_{k,i})$	grobe Integrator: Berechnung der Lösung vom Zeitpunkt an der Stelle i bis $t + \Delta T$
$F_{\Delta T}(y^G_{k,i})$	feine Integrator: Berechnung der Lösung vom Zeitpunkt an der Stelle i bis $t + \Delta T$
ΔT	grober Zeitschritt
Δt	feiner Zeitschritt
$d_{k,i}$	Differenz zwischen der groben und feinen Lösung

2.2 Einführung

Die Grundidee des Parareal-Algorithmus ist die Lösung annähernd mit einem groben Zeitintegrator zu berechnen und anschließend diese iterativ zu verfeinern. Die Besonderheit des Algorithmus ist die zeitparallele Berechnung der Lösung. Dafür wird das gesamte Zeitintervall in mehrere Zeitabschnitte, die sogenannten Segmente, aufgeteilt. Im Folgenden bezeichnet N die Anzahl der Segmente. Im ersten Schritt wird die Lösung, ausgehend von einem gegebenen Anfangswert, mit einem groben Zeitschritt seriell berechnet. Die Lösungen am Ende der Segmente repräsentieren die Anfangswerte für weitere Berechnungen. Als Nächstes wird die Schrittweite verkleinert um genauere Lösungen zu erzielen. Dadurch, dass das Zeitintervall in N Segmente aufgeteilt wurde und die Anfangswerte für jedes Segment durch die erste grobe Berechnung gegeben sind, kann die Verfeinerung im zweiten Schritt parallel erfolgen. Der Algorithmus iteriert so lange bis alle Segmente konvergieren. Ein Segment wird als konvergent bezeichnet, wenn

die Differenz der feinen Lösung zwei aufeinangerfolgenden Iterationen die Konvergenzgrenze nicht überschreitet. Dies bedeutet, dass die absolute Differenz der beiden Werte $y^F_{k-1,i}$ und $y^F_{k,i}$ betrachtet wird. Die Konvergenzgrenze ist das Konvergenzkriterium, das vor Beginn der Durchführung festgelegt wird. Wird keine Konvergenzgrenze angegeben, wird diese auf Null gesetzt. Die Konvergenzgrenze hat einen Einfluss auf die Anzahl der Iterationen. Je größer die Konvergenzgrenze, desto weniger Iterationen werden für die Berechnung benötigt. In Abschnitt 4.3 wird die Auswirkung der unterschiedlichen Konvergenzgrenzen auf die Anzahl der Iterationen und den Fehler untersucht.

2.3 Algorithmus

Der Algorithmus setzt sich aus zwei Teilen zusammen, zum Einen aus der Initialisierung und zum Anderen aus der Iteration. Bei der Initialisierung wird die Lösung zuerst annähernd mit einem groben Zeitschritt berechnet und die Anfangswerte für die einzelnen Segmente gesetzt. Bevor die erste Berechnung durchgeführt werden kann, wird der grobe Integrator mit einem Anfangswert für $t = 0$ initialisiert. Anschließend erfolgt eine serielle Berechnung der Näherungswerte, Abbildung 2.1(a). Im zweiten Schritt der Initialisierung wird die Lösung, durch das Verkleinern der Zeitschritte verfeinert. Die Werte der groben Näherungslösung dienen als Anfangswerte für den feinen Zeitintegrator nachfolgender Iteration. Da die Anfangswerte für den feinen Integrator alle gegeben sind, kann die Berechnung der verfeinerten Lösung parallel durchgeführt werden. Abbildung 2.1(b) zeigt das Ergebnis nach dem Initialisierungsschritt. Für weitere Durchführungen und Konvergenzentscheidungen werden die Werte nach der Initialisierung und auch nach jedem Iterationsschritt am Ende jedes Segments gespeichert, also zu den Zeitpunkten $t_0 + i \cdot \Delta T$, wobei i das i-te Segment ist. Ein Programmausschnitt des Initialisierungsschritts wird in Listing 2.1 dargestellt.

```
1 | y^G_{0,0} := y_0
2 | for i in [1,N]:
3 |     y^G_{0,i} := G_{ΔT}(y^G_{0,i-1})
4 |
5 | for i in [1,N]:
6 |     y^F_{0,i} := F_{ΔT}(y^G_{0,i-1})
```

Listing 2.1: Initialisierung

Nachdem der grobe und feine Integrator ihre erste Berechnung beendet haben und alle Anfangswerte gesetzt sind, beginnt der erste Iterationsschritt. Da sich die Anfangswerte y_0 im ersten Segment nicht ändern und hierdurch auch nicht die Lösung $y^F_{1,1}$, wird die Konvergenzgrenze nicht überschritten. Dadurch kann das erste Segment als konvergent angenommen werden. Der Algorithmus fängt die erste Iteration im zweiten Segment an. Dafür wird das Endergebnis des feinen Integrators des ersten Segments, also $y^F_{0,1}$, als Anfangswert für die erste Iteration des zweiten Segments angenommen und die Lösung grob berechnet, Abbildung 2.1(c). Als Nächstes wird die Differenz zwischen der feinen und groben Lösung der vorherigen Iteration ermittelt (Abbildung 2.1(d)). Dieser Wert wird für die Korrektur der groben (ungenauen) Lösung verwendet. Für die Korrektur wird der ermittelte Differenzwert an die neu berechnete grobe Lösung addiert, um somit die Differenz auszugleichen. Dieser Korrekturschritt ist sehr wichtig, da eine stetige Funktion als Ergebnis erwünscht wird. Um die Berechnung fortzuführen muss die berechnete Differenz ausgeglichen werden. Mit diesem neuen korrigierten Wert wird weiter gerechnet

(Abbildung 2.1(e)). Dieser Korrekturschritt und die grobe Berechnung wird für alle Segmente wiederholt, wie in Abbildungen 2.1(f) und 2.1(g) zu sehen ist. Nachdem alle Segmente grob berechnet wurden, durchläuft der feine Integrator alle Segmente parallel und errechnet die feine Lösung. Dabei werden alle konvergenten Segmente ausgelassen. Das Ergebnis nach dem ersten Iterationsschritt wird in Abbildung 2.1(h) dargestellt. Listing 2.2 beschreibt den Iterationsschritt.

Für die Überprüfung der Konvergenz werden die Werte $y^F_{k,i}$ und $y^F_{k-1,i}$ betrachtet. Dazu wird die Differenz dieser beider Werte gebildet und anschließend mit der Konvergenzgrenze verglichen. Liegt die Differenz unter der Konvergenzgrenze, so gilt das Segment als konvergent. Aufgrund der Konvergenzbedingung können mehrere Segmente gleichzeitig konvergieren. Bei einer Konvergenzgrenze von Null wird nach jedem Iterationsschritt nur ein Segment als konvergent angenommen. Somit durchläuft der Algorithmus maximal $N-1$ Iterationen. Neben den $N-1$ Iterationen wird ein Initialisierungsschritt durchgeführt, zusammen sind es N Durchläufe, die der Algorithmus braucht, um die Lösung der partiellen Differentialgleichung zu berechnen. Je größer die Konvergenzgrenze ist, desto weniger Iterationen werden benötigt. Der Fehler kann mit abnehmender Iterationenanzahl steigen, dies wird in Kapitel 4 untersucht.

```
for k in [1,N[:                                                    1
  n:= k+1  # Zeitschritt nach dem zuletzt konvergierten Zeitschritt 2
  y^G_{k,n} := G_ΔT(y^F_{k-1,n-1})                                 3
                                                                   4
  for i in [n+1,N]:                                                5
    d_{k,i-1} := y^F_{k-1,i-1} - y^G_{k-1,i-1}                     6
    y_{k,i-1} := y^G_{k,i-1} + d_{k,i-1}                           7
    y^G_{k,i} := G_ΔT(y_{k,i-1})                                   8
                                                                   9
  for i in [n,N]:                                                  10
    y^F_{k,i} := F_ΔT(y^G_{k,i}) #parallel                         11
```

Listing 2.2: Iteration

2.4 Programmierung

In diesem Kapitel wird die Parareal-Implementierung erläutert. Bei dem Programm handelt es sich um das von Martin Schreiber implementierte Parareal-Algorithmus in der Programmiersprache Python. Es werden nur für diese Bachelorarbeit relevante Klassen und Funktionen dargestellt. Zuerst wird in Abschnitt 2.4.1 der Programmaufbau beschrieben und wie die existierenden Klassen miteinander zusammenhängen. Die Funktionen, die für den Algorithmus benötigt werden, sind in Abschnitt 2.4.2 aufgelistet und definiert.

2.4.1 Klassenübersicht

In Abbildung 2.2 ist eine grobe Übersicht über die Klassen am Beispiel einer gewöhnlichen Differentialgleichung mit einer MPI-Parallelisierung gegeben. Die Script-Datei ist ein sehr wichtiger Bestandteil des Programms. Zum Lösen der Differentialgleichungen und zur Ausführung des Programms sind benutzerdefinierte Parameter notwendig. Diese Parameter werden in der

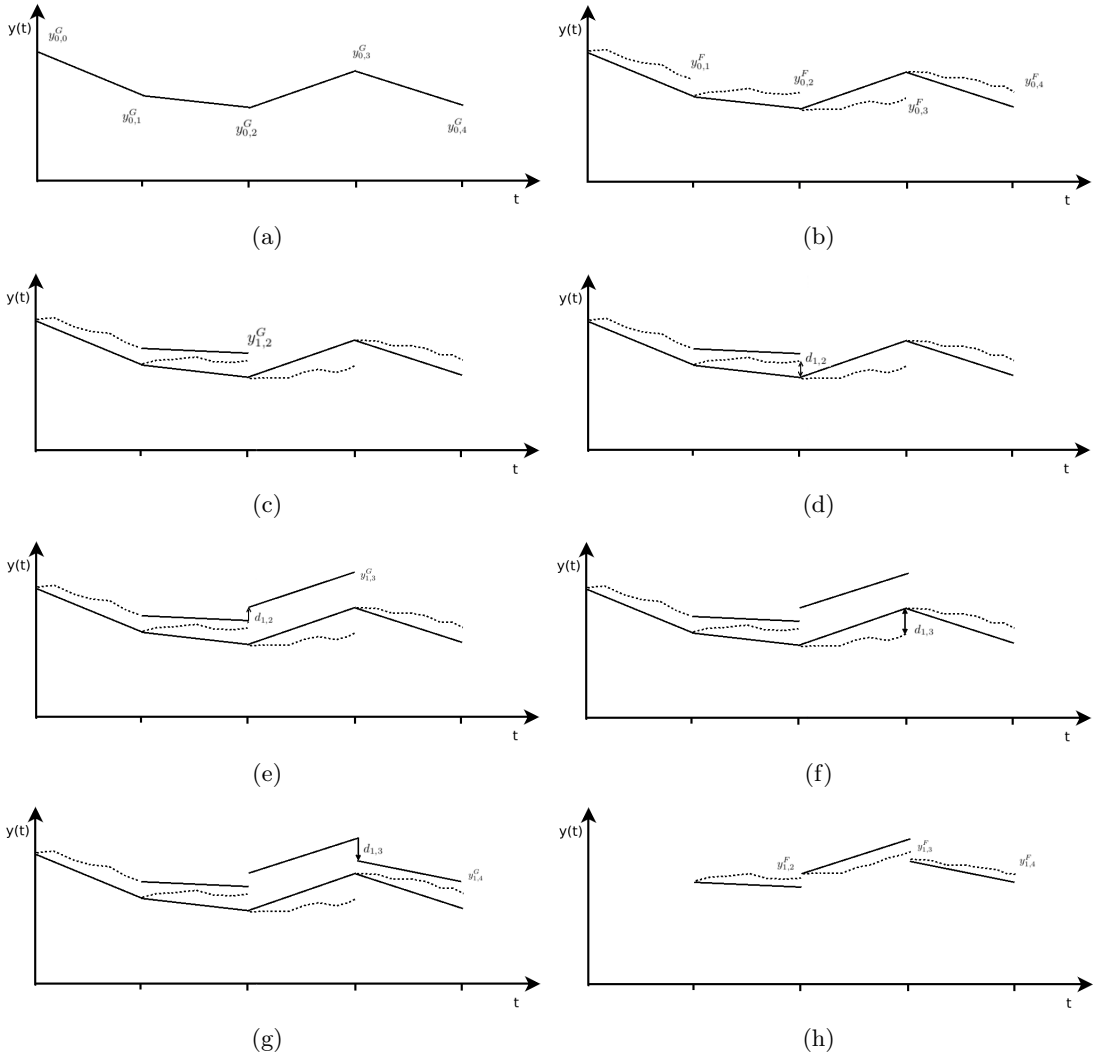


Abbildung 2.1: Parareal-Algorithmus an einem Beispiel

Script-Datei gespeichert, die im Laufe der Ausführung eingelesen werden. Die wichtigsten Parameter sind mit der dazugehörigen Beschreibung in der Tabelle 2.1 aufgelistet.

Nach der Ausführung der Script-Datei wird die Main-Klasse aufgerufen. Die Main-Klasse liest zuerst die Parameter für die Wahl der Simulation und des PinT-Drivers ein und ruft die entsprechenden .py-Dateien auf. Die beiden ausgewählten Klassen werden zunächst initialisiert, dafür werden weitere Parameter eingelesen und gespeichert. Der PinT-Driver beschreibt den Ablauf des Programms, beispielsweise die Reihenfolge der auszuführenden Funktionen. In der Simulationsklasse werden die einzelnen Funktionen, die für den Parareal-Algorithmus benötigt werden, implementiert. Die wichtigsten Funktionen werden im nächsten Abschnitt definiert. Je nachdem welcher Driver ausgewählt wird, erfolgt die Berechnung parallel oder seriell.

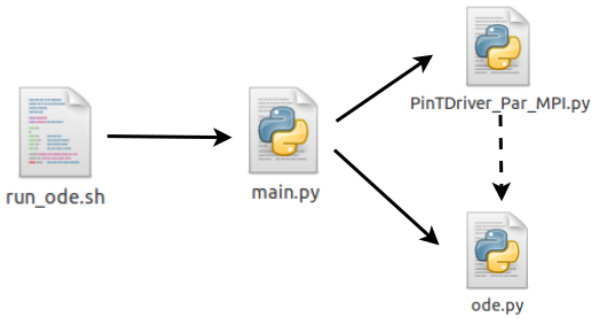


Abbildung 2.2: Klassenübersicht am Beispiel einer gewöhlichen Differentialgleichung und MPI-Parallelisierung

sim	Wahl der Simulation
pint	Wahl des PinT-Drivers
ctni	Anzahl der groben Intervalle
time	Zeitintervall
ctis	Größe eines Segments
ctt	Konvergenzgrenze
plot_ylim_min	minimale y-Wert
plot_ylim_max	maximale y-Wert

Tabelle 2.1: Parameter

2.4.2 Interface Beschreibung

Das Interface lässt sich in drei wichtige Gruppen aufteilen: (1) Initialisierung, (2) Berechnung und (3) Korrektur. Im Folgenden werden die Aufgaben der Gruppen und die dazugehörigen Funktionen aufgelistet und beschrieben.

Initialisierung

Bevor die ersten Berechnungen durchgeführt werden können, müssen die Zeitrahmen für jedes Segment festgelegt werden. Für das Setzen der Anfangswerte existieren zwei verschiedene Funktionen. Die erste Funktion (*set_initial_value()*) setzt den vorgegebenen Anfangswert für das erste Segment zum Zeitpunkt $t = 0$. Weitere Anfangswerte müssen zuerst vom groben Integrator berechnet werden. Das Ergebnis am Ende eines Abschnitts bildet den Anfangswert für das nachfolgende Segment. Die Initialisierung weiterer Segmente erfolgt mit der Funktion *set_simulation_data()*.

- set_simulation_timeframe(i_timeframe):
  setzt das Zeitfenster für alle Segment

- setup_initial_value():
  setzt den Anfangswert zum Zeitpunkt $t = 0$

- set_simulation_data(i_sim_data):
  setzt die Anfangswerte für weitere Segmente zum Zeitpunkt $t \neq 0$

Berechnung

Zum Berechnen der Lösung werden zwei Funktionen verwendet, (1) *run_timestep_coarse()* und (2) *run_timestep_fine()*. Die erste Funktion berechnet die Näherungslösung und die zweite Funktion verfeinert die grob berechneten Ergebnisse. Die zwei weiteren Funktionen geben die berechneten Lösungen des groben und des feinen Integrators zurück.

- run_timestep_fine():
  berechnet die feine Lösung
- get_data_timestep_fine():
  gibt die feine Lösung zurück. Wird z.B. bei der Korrektur verwendet
- run_timestep_coarse():
  berechnet die grobe Lösung
- get_data_timestep_coarse(): gibt die Lösung der groben Berechnung zurück. Wird z.B. bei der Differenzrechnung verwendet

Korrektur

Wie schon erwähnt, ist die Korrektur ein sehr wichtiger Schritt. Ohne den Korrekturschritt würde die Lösungsfunktion nicht stetig verlaufen. Ein weiterer Grund für die Korrektur ist, dass die Lösungen genauer berechnet werden können. Hierdurch wird der Fehler mittels der Korrektur verringert. Dafür muss zuerst die Differenz der Werte $y_{k,i}^G$ und $y_{k,i}^F$ bestimmt (*compute_difference()*) und anschließend mit dem neuen groben Wert verrechnet werden (*compute_output_data()*). Eine weitere Funktion, die zu der Gruppe Korrektur gehört, ist *return_error_estimation()*. Diese berechnet den Wert für den Konvergenztest.

- compute_difference():
  berechnet die Differenz zwischen der feinen und groben Lösung
- compute_output_data():
  berechnet den neuen Wert für die nächste Iteration
- get_output_data():
  gibt den korrigierten Wert aus
- return_error_estimation():
  gibt den Wert für die Fehlerabschätzung aus, z.B. für den Konvergenztest

3 Kombinationstechnik

Angenommen es soll ein Problem in Raum und Zeit gelöst werden, das als Modell durch ein System partieller Differentialgleichungen beschrieben ist. Die Lösung der Differentialgleichungen muss numerisch berechnet werden. Dazu werden die Differentialgleichungen auf einem Gitter diskretisiert und in ein lineares Gleichungssystem überführt. Um eine hohe Genauigkeit zu erzielen werden die Gitter verfeinert, das heißt es kommen Gitterpunkte hinzu. Daraus folgen sehr große Gleichungssysteme. Mit der Größe der Gleichungssysteme steigt die Rechenzeit und der Speicherbedarf. Um dieses Problem zu lösen wird die Kombinationstechnik auf dünnen Gittern verwendet. Eine ausführliche Beschreibung der dünnen Gitter und der Kombinationstechnik gibt es in der Dissertation von Christoph Kranz [4].

Die Idee der Kombinationstechnik ist es, nicht auf einem einzelnen Vollgitter zu rechnen, sondern das Problem auf mehrere gröbere Gitter mit unterschiedlicher Maschenweite in alle Raumrichtungen zu verteilen. Das Ergebnis der Kombinationslösung ist zwar nicht gleich der Lösung auf einem vollen Gitter, liegt jedoch sehr nahe. Die Anzahl der Gitterpunkte aller gröberen Gittern ist wesentlich kleiner als die Anzahl der Gitterpunkte auf einem vollen Gitter und somit wird der Speicherbedarf und die Rechenzeit reduziert.

3.1 Notation

Für die Kombinationstechnik verwendeten Notationen sind in diesem Abschnitt aufgelistet.

n	(n, n)-Level
$\overline{\Omega}$	Raumgebiet
Ω_n^c	dünnes Gitter
Ω_n	volles Gitter
Ω_n^s	ausgedünntes volles Gitter
u_n^c	Kombinationslösung auf dem dünnen Gitter Ω_n^c
$u_{\underline{l}}$	Funktion auf dem Vollgitter $\Omega_{\underline{l}}$
u_n^s	Kombinationslösung auf dem ausgedünnten vollen Gitter Ω_n^s
d	Dimension
$\underline{l}$	Level des Gitters
s	Ausdünnungsparameter
$\underline{1}$	Einheitsvektor
h_{l_j}	Maschenweite des Gitters in j-Richtung
$I_{\underline{l}}^s$	Multiindexmenge

3.2 Einführung

Bei der Dünngitterinterpolation wird in ein Raumgebiet $\overline{\Omega}$ ein dünnes Gitter Ω_n^c gelegt, das weniger Gitterpunkte enthält als ein volles Gitter Ω_n. Dabei wird der Interpolationsfehler nur minimal vergrößert. Bei einer Verkleinerung der Maschenweite h_{l_j} wird das Gitter verfeinert, das heißt die Anzahl der Gitterpunkte steigt. Die Maschenweite ist definiert als $h_{l_j} = 2^{-l_j}$. Für die hier verwendete Kombinationstechnik wird das zweidimensionale Gebiet $\overline{\Omega} = [0,1]^2$ betrachtet. Um zu einer Problemlösung auf einem dünnen Gitter zu gelangen wird die Kombinationstechnik verwendet. Das Prinzip der Kombinationtechnik ist, durch geeignete Linearkombination verschiedener Vollgitterlösungen $u_{\underline{l}}$, eine Lösung u_n^c auf dem dünnen Gitter Ω_n^c zu berechnen. Die Kombinationslösung ist nicht identisch mit der Lösung, die direkt auf dem dünnen Gitter berechnet wird. Der Fehler der Kombinationslösung besitzt aber dieselbe Fehlerordnung wie die Dünngitterlösung.

Die allgemeine Formel zur Bildung der Linearkombination für isotrope Gitter lautet:

$$u_n^c = \sum_{j=0}^{d-1} (-1)^j \binom{d-1}{j} \sum_{|\underline{l}|_1 = n-j} u_{\underline{l}} \tag{3.1}$$

Im zweidimensionalen Fall ergibt sich folgende Formel:

$$u_{(n,n)}^c = \sum_{l_1+l_2=n} u_{(l_1,l_2)} - \sum_{l_1+l_2=n-1} u_{(l_1,l_2)} \tag{3.2}$$

Zum Beispiel würde die Kombinationslösung für ein $(3,3)$-Level Gitter folgendermaßen aussehen:

$$u_3^c = u_{(3,0)} + u_{(2,1)} + u_{(1,2)} + u_{(0,3)} - u_{(2,0)} - u_{(1,1)} - u_{(0,2)}$$

Das Level beschreibt die Größe eines Vollgitters. Daraus kann die Gittergröße $(2^n+1, 2^n+1)$ und die Maschenweite $(2^{-n}, 2^{-n})$ berechnet werden. Welche Gitter für Linearkombination verwendet werden, sind in Abbildung 3.1 eingezeichnet. Für die Kombinationslösung werden die Gitter auf der Hauptdiagonale addiert und die Gitter auf der Nebendiagonale subtrahiert.

3.3 Kombinationstechnik mit Ausdünnung

Eine weitere Methode der Kombinationstechnik ist das Ausdünnungsverfahren. Der Unterschied zur normalen Kombinationstechnik liegt an der Wahl der zu kombinierenden Gitter. Um die Namensherkunft der Ausdünnung zu verdeutlichen, betrachten wir zunächst das volle Gitter. Das Problem wird aber nicht auf dem vollen Gitter gelöst, sondern es wird eine Linearkombination von gröberen Gittern gebildet. Die gröberen Gitter werden durch das Weglassen von Gitterpunkten in den unterschiedlichen Raumrichtungen erhalten, das heißt es werden Gitter niedrigeren Levels zum Kombinieren verwendet. Dies bedeutet, dass das volle Gitter ausgedünnt wird, daher wird diese Methode das Ausdünnungsverfahren genannt. Der so genannte Ausdünnungsparameter gibt an wie weit ausgedünnt werden soll. Bei einem maximalen Ausdünnungsparameter erhält man die normale Kombinationstechnik. Die Formel für die Kombinationstechnik mit Ausdünnung lautet:

$$u_n^s = \sum_{j=o}^{d-1} (-1)^j \binom{d-1}{j} \sum_{\underline{k} \in I_{n-j}^{s-j}} u_{\underline{k}} \tag{3.3}$$

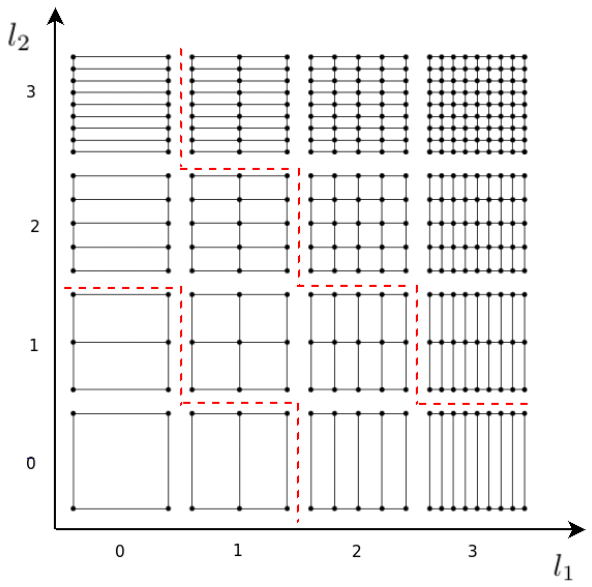


Abbildung 3.1: Kombinationstechnik an einem Beispiel eines (3,3)-Level Gitters

Mit der Multiindexmenge:

$$I_n^s := \left\{(k_1, ..., k_d) \middle| |\underline{k}|_1 = dn - (d-1)s, \quad n - s \leq k_j \leq n, \quad j = 1, ..., d\right\} \tag{3.4}$$

Die Multiindexmenge gibt an, welche Gitter für die Kombinationstechnik mit Ausdünnung verwendet werden. Das Prinzip des Ausdünnungsverfahrens ist an einem Beispiel für das $(3, 3)$-Level Gitter mit Ausdünnungsparameter $s = 2$ in Abbildung 3.2 dargestellt. Je kleiner der Ausdünnungsparameter ist, desto mehr Punkte erhält das Gitter, daraus resultiert eine genauere Lösung. Abbildung 3.3 zeigt die Kombinationslösungen für verschiedene Ausdünnungsparameter im Vergleich. Der Ausdünnungsparameter bei diesem Verfahren kann maximal n groß sein, also maximal so groß wie das Level des Lösungsgitters. Für den maximalen Ausdünnungsparameter $s = n$ wird die normale Kombinationstechnik erhalten und für $s = 0$ das volle Gitter. Der Unterschied zwischen den verschiedenen Gittern ist in Abbildung 3.4 zu erkennen.

Nun sind aber nicht alle Gitter isotrop. Für anisotrope Gitter lässt sich die Formel für die Kombinationstechnik leicht anpassen, so dass wir folgende Folmel erhalten:

$$u_{\underline{l}}^s = \sum_{j=o}^{d-1} (-1)^j \binom{d-1}{j} \sum_{\underline{k} \in I_{\underline{l}-j\cdot\underline{1}}^{s-j}} u_{\underline{k}} \tag{3.5}$$

Mit der Multiindexmenge:

$$I_{\underline{l}}^s := \left\{(k_1, ..., k_d) \middle| |\underline{k}|_1 = |\underline{l}|_1 - (d-1)s, \quad l_j - s \leq k_j \leq l_j, \quad j = 1, ..., d\right\} \tag{3.6}$$

Bei anisotropen Gittern ist der maximale Ausdünnungsparameter definiert als: $s = \min(l_1, l_2)$.

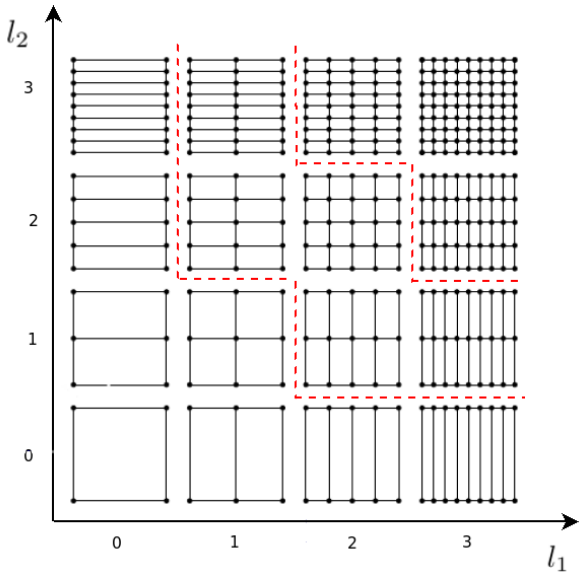
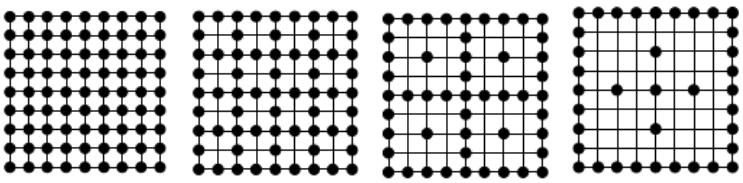
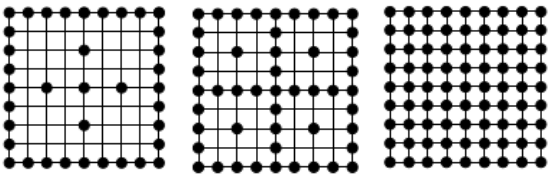


Abbildung 3.2: Prinzip der Kombinationstechnik mit Ausdünnung am Beispiel eines (3,3)-Level Gitters mit einem Ausdünnungsparameter s=2



(a) s=0 (b) s=1 (c) s=2 (d) s=3

Abbildung 3.3: Kombinationslösungen mit verschiedenen Ausdünnungsparametern im Vergleich



(a) (b) (c)

Abbildung 3.4: Unterschiedliche Gitter im Vergleich: (a) Dünnes Gitter, (b) Ausgedünntes volles Gitter, (c) Volles Gitter

4 Kombinationstechnik als Zeitintegrator

In diesem Kapitel wird die Kombinationstechnik als Zeitintegrator im Parareal-Algorithmus untersucht. Im nächsten Abschnitt werden die Unterschiede zum normalen Parareal-Algorithmus erläutert. Weiter wird die Integrierung der Kombinationstechnik in die Parareal-Implementierung beschrieben. Anschließend werden die für diese Bachelorarbeit durchgeführten Experimente beschrieben und ausgewertet.

4.1 Einführung

Im Parareal-Algorithmus geht es darum die Lösung zuerst grob zu berechnen und diese dann zu verfeinern und zwar unabhängig vom Zeitschritt. Wir führen die Kombinationstechnik als groben Zeitintegrator ein. Der Unterschied zum normalen Algorithmus besteht darin, nicht den Zeitschritt zu verfeinern, sondern die Mascheinweite des Gitters. Dies bedeutet, dass die Kombinationstechnik mit Ausdünnung die grobe Lösung darstellt und die Lösung auf vollem Gitter die Verfeinerung. Für die Grobgitterlösung muss der Ausdünnungsparameter also größer Null sein, wobei er für die Feingitterlösung konstant auf Null gesetzt ist. Der Zeitschritt ΔT bleibt hier unverändert. Nach mehreren Untersuchungen fällt auf, dass die besten Ergebnisse für $\Delta_t = 0,01$ erzielt werden. Dieser Wert wird für alle Tests als fester Wert angenommen. Im Folgenden wird das zweidimensionale Modellproblem der Wärmeleitungsgleichung betrachtet.

4.1.1 Wärmeleitungsgleichung

Die hier verwendete Wärmeleitungsgleichung lautet:

$$\frac{\partial u}{\partial t}(\vec{x}, t) = \kappa \Delta u(\vec{x}, t) \tag{4.1}$$

wobei κ die Wärmeleitfähigkeit beschreibt. Da es in dieser Bachelorarbeit nicht relevant ist, wird es aus Vereinfachungsgründen auf Eins gesetzt.
Wenn der Laplace-Operator ($\Delta = div\nabla$) ausgeschrieben wird, kann die Formel wie folgt geschrieben werden:

$$\frac{\partial u}{\partial t}(\vec{x}, t) = \frac{\partial^2 u(\vec{x}, t)}{\partial x_1^2} + \frac{\partial^2 u(\vec{x}, t)}{\partial x_2^2} \tag{4.2}$$

Führt man eine Ortsdiskretisierung dieser Gleichungen durch, lässt sich die Gleichung 4.2 umschreiben in:

$$\frac{\partial u_{i,j}(t)}{\partial t} \approx \frac{u_{i-1,j}(t) - 2u_{i,j}(t) + u_{i+1,j}(t)}{\Delta x_1^2} + \frac{u_{i,j-1}(t) - 2u_{i,j}(t) + u_{i,j+1}(t)}{\Delta x_2^2} \tag{4.3}$$

Für $0 \leq i \leq n$ und $0 \leq j \leq m$.
Um ein lineares Gleichungssystem zu erhalten, wird die Gleichung in eine Matrixschreibweise gebracht:

$$\frac{\partial u(t)}{\partial t} \approx \frac{1}{\Delta x_1^2} A_1 u(t) + \frac{1}{\Delta x_2^2} A_2 u(t) \tag{4.4}$$

Als Randwerte wird die Dirichlet-Randbedingung verwendet, das heißt alle Randwerte werden gleich Null gesetzt:

$$u(\vec{x}, t)|_\Gamma \equiv 0 \tag{4.5}$$

Da die Randwerte Null sind, können sie in den Matrizen A_1 und A_2 weggelassen werden. Durch das Wegglassen der Randwerte werden die Matrizen A_1 und A_2 verkleinert. Die Verkleinerung der Matrizen spart Rechenzeit und Speicherplatz. Die beiden Matrizen der Gleichung 4.4 sehen folgendermaßen aus:

$$A_1 = \begin{pmatrix} \begin{array}{ccc|ccc} -2 & & & 1 & & \\ & \ddots & & & \ddots & \\ & & -2 & & & 1 \\ \hline 1 & & & -2 & & \\ & \ddots & & & \ddots & \\ & & 1 & & & -2 \end{array} & & 0 \\ & \ddots & \\ 0 & & \begin{array}{ccc|ccc} -2 & & & 1 & & \\ & \ddots & & & \ddots & \\ & & -2 & & & 1 \\ \hline 1 & & & -2 & & \\ & \ddots & & & \ddots & \\ & & 1 & & & -2 \end{array} \end{pmatrix}$$

$$A_2 = \begin{pmatrix} \begin{array}{cccc} -2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{array} & & 0 \\ & \ddots & \\ 0 & & \begin{array}{cccc} -2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{array} \end{pmatrix}$$

Jede Blockmatrix ist $m \times m$ groß und jede Matrix besteht aus insgesamt $n \times n$ Blockmatrizen, also sind A_1 und A_2 $nm \times nm$ groß.
Folgende Funktion dient als Anfangswert der partiellen Differentialgleichung:

$$u(\vec{x}, 0) = \sin(x\pi) \cdot \sin(y\pi) \tag{4.6}$$

Die Wärmeleitungsgleichung konvergiert gegen den Wert:

$$u(\vec{x}, t) = \exp^{-2\pi^2 t} \cdot u(\vec{x}, 0) \tag{4.7}$$

Diese exakten Werte werden später für die Fehlerberechnung verwendet. Es hat sich herausgestellt, dass eine andere Möglichkeit den Fehler zu berechnen bessere Ergebnisse liefert, dazu mehr im Abschnitt 4.3.

4.1.2 implizite Euler-Verfahren

Mit Hilfe des Euler-Verfahrens werden die partiellen Differentialgleichungen gelöst. Da die Randwerte alle Null sind, vereinfacht es die Darstellung des Problems.

Gegeben sei ein Anfangswertproblem:

$$\dot{x} = f(x,t)$$

$$x(t_0) = x_0$$

Für eine Differentialgleichung wird eine Diskretisierungsschrittweite $h > 0$ gewählt. Betrachtet werden die diskreten Zeitpunkte:

$$t_k = t_0 + k \cdot h, \quad k = 0, 1, 2, ...$$

Gesucht sind die iterierten Werte:

$$x_{k+1} = x_k + f(x_{k+1}, t_{k+1})$$

Sei die Wärmeleitungsgleichung 4.4 das Anfangswertproblem. Wir führen nun eine Zeitdiskretisierungen durch Differenzenquotienten in der Zeit ein. u^k beschreibt die diskrete Lösung zum Zeitpunkt t_k.

$$u^{k+1} = u^k + \frac{\Delta t}{\Delta x_1^2} A_1 u^{k+1} + \frac{\Delta t}{\Delta x_2^2} A_2 u^{k+1} \tag{4.8}$$

Nach Umstellen der Gleichung 4.8, wird ein lineares Gleichungssystem erhalten:

$$\left(I - \frac{\Delta t}{\Delta x_1^2} A_1 - \frac{\Delta t}{\Delta x_2^2} A_2\right) u^{k+1} = u^k \tag{4.9}$$

Die Matrixwerte werden zum Lösen der Differentialgleichung spaltenweise ohne Randwerte in einen $(n-2) \cdot (m-2)$ großen Vektor gespeichert. Die Vektoren des Gleichungssystems sehen folgendermaßen aus:

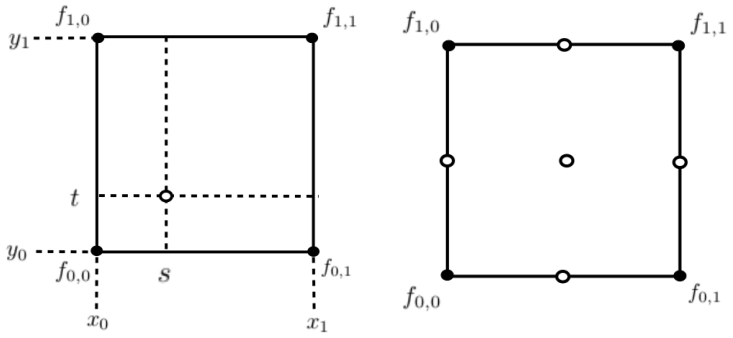
$$u = \begin{pmatrix} u_{1,1} \\ \vdots \\ u_{1,m-1} \\ \vdots \\ u_{n-1,1} \\ \vdots \\ u_{n-1,m-1} \end{pmatrix}$$

Nach dem Lösen des Gleichungssystems müssen die Werte in die Matrix zurückgeschrieben werden. Das implizite Euler-Verfahren wird auf alle Gitter der Kombinationtechnik angewendet.

4.1.3 Bilineare Interpolation

Mit Hilfe der bilinearen Interpolation werden die Gitter mit unterschiedlichen Größen miteinander verrechnet. Das Schema der bilinearen Interpolation ist in 4.1(a) abgebildet. Die zu interpolierenden Werte werden mit der folgenden Formel berechnet:

$$f_{s,t} = (1-\alpha)\big((1-\beta) f_{0,0} + \beta f_{1,0}\big) + \alpha\big((1-\beta) f_{0,1} + \beta f_{1,1}\big) \tag{4.10}$$



(a) (b)

Abbildung 4.1: (a) Schema bilinearer Interpolation, (b) die zu interpolierenden Punkte für das Gitter

Für die beiden Variablen α und β gilt:

$$\alpha = \frac{s - x_0}{x_1 - x_0}, \quad \beta = \frac{t - y_0}{y_1 - y_0}, \quad 0 \leq \alpha, \beta \leq 1$$

Eine $(n \times m)$ Matrix wird solange vergrößert bis sie die Größe der Ergebnismatrix erreicht hat. Dabei wird jede zweite Spalte bzw. Zeile eine neue Spalte oder Zeile eingefügt. Zum Beispiel wird aus einer 3×3-Matrix ein e5 $\times$ 5-Matrix. Die neu hinzugefügten Spalten/Zeilen werden mittels der bilinearen Interpolation befüllt.

Da es sich um Gitter mit äquidistandem Abstand handelt, können α und β nur drei verschiedene Werte 0, 1 oder $\frac{1}{2}$ annehmen. Die Erweiterung der Gitter erfolg immer nur in eine Richtung. Dadurch ist mindestens eine der Variablen gleich 0 oder 1. Daraus folgt, dass nur zwei Werte für die Interpolation benötigt werden. Dies gleicht der linearen Interpolation:

$$f_x = f_0 \frac{x_1 - x}{x_1 - x_0} + f_1 \frac{x - x_0}{x_1 - x_0} \tag{4.11}$$

Aufgrund äquidistanter Abstände zwischen den x-Werten, folgt:

$$f_x = \frac{1}{2} f_0 + \frac{1}{2} f_1 = \frac{f_0 + f_1}{2} \tag{4.12}$$

Der im Bild 4.1(b) eingezeichnete Mittelpunkt kann aus zwei interpolierten Werten berechnet werden. Somit ist nur die lineare Interpolation von Bedeutung.

4.2 Programmierung

Wie schon in der Einleitung erwähnt, wird das Programm von Martin Schreiber erweitert und an die Kombinationstechnik angepasst. In diesem Abschnitt werden die für diese Bachelorarbeit benötigten Klassen und die Einordnung in das schon existierende Programm beschrieben. Abschnitt 4.2.2 gibt einen Überblick über die zusätzlichen Funktionen. Das Interface ist nicht nur um Funktionen erweitert worden, sondern auch um einige Parameter. Die für die Kombinationstechnik verwendeten Parameter sind in der Tabelle 4.1 aufgelistet. Neben der Kombinationstechnik wird eine weitere Technik eingeführt, die sogenannte Rekombination. Die Rekombination wird verwendet, um nach dem Kombinieren die verwendeten vollen Gitter zu aktualisieren. Die Methode der Rekombination wird in Abschnitt 4.3 vorgestellt.

tp	Ausdünnungsparameter
grid_level_x	Gitterlevel in x-Richtung
grid_level_y	Gitterlevel in y-Richtung
ret	mit Rekombination
eec	Fehlerberechnung, entweder zur exakten Lösung oder Feingitterlösung

Tabelle 4.1: Parameterübersicht

4.2.1 Klassenübersicht

In der schon vorgestellten Klassenübersicht ändert sich lediglich nur der Simulationsteil. Vom Ablauf bleibt alles unverändert. Die komplette Klassenübersicht ist in Abbildung 4.2 abgebildet. Die unveränderten Klassen von Martin Schreiber sind in der Abbildung 4.2 gekennzeichnet.

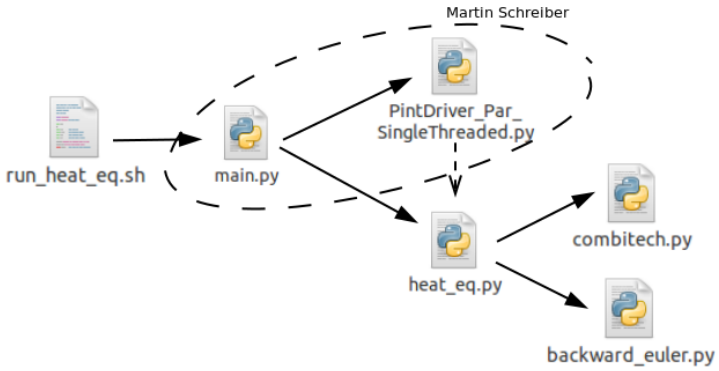


Abbildung 4.2: Klassenübersicht am Beispiel der Wärmeleitungsgleichung

Die Script-Datei wird nur um die in Tabelle 4.1 aufgelisteten Parameter erweitert. Für die Durchführung der Tests reicht eine serielle Ausführung des Programms aus, deshalb wird der *PinTDriver-SingleThreaded* als Standard ausgewählt. Als Simulation wird hier die in Abshnitt 4.1.1 vorgestellte Wärmeleitungsgleichung verwendet. Für die Kombinationstechnik und für das implizite Euler-Verfahren werden separate Klassen erstellt. Diese werden zum Berechnen und Kombinieren von der Simulations-Datei aufgerufen. In der Simulationsklasse selbst sind die Funkionen aus Abschnitt 2.4.2 ausimplementiert. Funktionen, die für die Kombinationtechnik und für das implizite Euler-Verfahren verwendet werden, sind im nächsten Abschnitt aufgeführt und kurz erläutert.

4.2.2 Interface Beschreibung

Wie auch in der letzten Interface Beschreibung lassen sich die Funktionen in drei Gruppen aufteilen. Für die Auswertung der Ergebnisse werden die Fehlerwerte betrachtet. Die Funktionen, die dafür benötigt werden, bilden eine weitere wichtige Gruppe, die Fehlerberechnung.

Initialisierung

- initial(grid):
  füllt die Matrix mit Anfangswerten der Gleichung 4.6

- get_index(thin_par, grid_size):
  berechnet die Indexmenge 3.4 und gibt diese aus

Berechnung

- get_grid_list():
  gibt die Liste mit allen Matrizen wieder
- coeff(count):
  gibt die Koeffizienten-Liste der Gleichung 3.3 aus
- combi(grid_list, result, coeff_list):
  berechnet die Kombinationslösung 3.3
- recombi(grid_list, result):
  Rekombination der Lösung
- grid_calculate(grid_list, delta_t):
  Löst das lineare Gleichungssystem 4.9

Fehlerberechnung

- get_error(error,i_time,i_data):
  berechnet den Fehler und gibt diesen aus

Vor der Initialisierung werden zuerst die Indexmenge 3.4 und die Koeffizienten aus der Gleichung 3.3 (*coeff()*), die in einer Liste gespeichert werden, berechnet. Im zweidimensionalen Raum können die Koeffizienten nur zwei Werte annehmen, 1 oder -1. Die Berechnung der Indizes erfolgt mit der Funktion *get_index()*, die in der Klasse der Kombinationstechnik implementiert ist. Die Menge der Indizes wird bei der Initialisierung (*setup_initial_value()* (s. Kapitel 2.4 für die Erstellung der Matrizen verwendet. Anhand der Indizes kann die Größe der Matrizen für die Kombinationstechnik errechnet werden. Nachdem alle Matrizen aufgestellt und in eine Liste gespeichert wurden, werden sie mit Hilfe *initial()* mit Anfangswerten aus der Gleichung 4.6 gefüllt. Sobald alle Matrizen initialisiert sind kann die Berechnung beginnen. Zum Aufrufen der Matrizen wird die Funktion *get_grid_list()* verwendet. Mit dem impliziten Euler-Verfahren können die neuen Werte errechnet werden. Zum Aufstellen des linearen Gleichungssystems 4.9 werden die Werte der Matrix spaltenweise in einen Vektor gespeichert. Als Nächstes löst die Funktion *grid_calculate()* das lineare Gleichungssystem für jede einzelne Matrix aus der Liste und überschreibt diese mit den neu berechneten Werten. Zum Kombinieren der Matrizen wird die Funktion *combi()* aufgerufen. Hier werden die Matrizen mit Hilfe der linearen Interpolation (s. Kapitel 4.1.3) und den vorher berechneten Koeffizienten addiert.

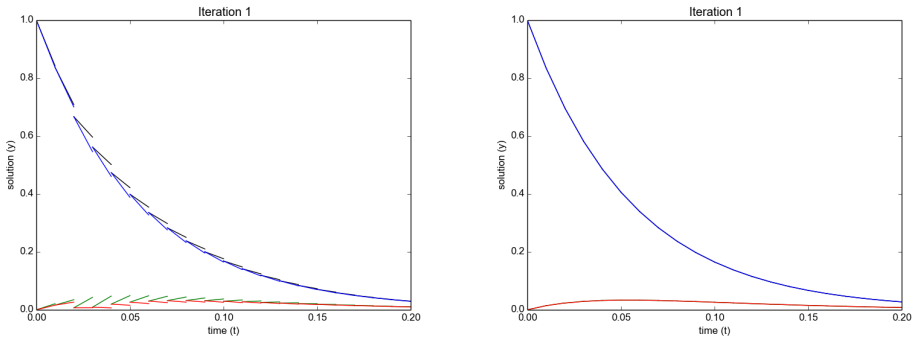
Wie schon erwähnt, bietet sich die Möglichkeit die Matrizen zu aktualisieren. Dies geschieht mit der Technik der Rekombination. Bei dieser Technik passiert genau das Gegenteil der Kombination. Die neu kombinierten Werte werden in alle Matrizen gespeichert. Bei der Rekombination wird mit den neuen Werten weiter gerechnet. Die andere Möglichkeit ist, mit den alten Werten weiter zu rechnen. Welche der beiden Möglichkeiten verwendet wird, legt der Parameter *ret* fest. Für diese Methode wurde die Funktion *recombi()* hinzugefügt. Ist der Parameter *ret* auf *true* gesetzt, wird die Funktion *recombi()* nach der Kombination aufgerufen. Mehr zu der Rekombination und der Auswirkung auf das Ergebnis gibt es im Abschnitt 4.3.

Nach einem Iterationsschritt wird der Fehler berechnet (*get_error()*). Für die Fehlerberechnung existieren zwei Möglichkeiten. Die Differenz kann zum einen zur exakten Lösung und zum anderen zur Feingitterlösung berechnet werden. Weitere Details zur Fehlerberechnung gibt es im Abschnitt 4.3. Welche der beide Varianten verwendet wird, gibt der Parameter *eec* an. Nach jeder Iteration werden die Fehlerwerte von konvergierten Segmenten addiert. Nachdem alle Segmente konvergiert sind, wird die mittlere absolute Differenz (MAD) berechnet, sowohl für die grobe als auch für die feine Lösung. Für die Auswertung der Ergebnisse wird nur der Mittelpunkt betrachtet. Dies spart nicht nur Speicher sondern auch Rechenzeit.

4.3 Experimente

Alle Experimente wurden mit der Schrittweite $\Delta T = 0,01$ und insgesamt 20 Zeitabschnitten durchgeführt. Kombiniert wurde nach jedem Zeitschritt. Für die Auswertung der Ergebnisse wird nur der Mittelpunkt betrachtet. Gezeichnet werden insgesamt 4 Funktionen: (1) die Feingitterlösung (blau), (2) die Grobgitterlösung (schwarz), (3) der Fehler der Feingitterlösung (rot) und (4) der Fehler der Grobgitterlösung (grün). Da nach der ersten Iteration die größten Unterschiede zu erkennen sind, werden hier fast nur die Plots nach dem ersten Iterationsschritt gezeigt.

Die ersten Tests wurden für unterschiedliche Gitterlevel durchgeführt. Wie es zu erwarten war, ist der Unterschied zwischen der Grobgitterlösung und der Feingitterlösung bei einem niedrigerem Gitterlevel deutlicher zu erkennen. Für das Beispiel werden das $(3, 3)$-Levelgitter mit dem Ausdünnungsparameter $s = 3$ und das $(7, 7)$-Level Gitter mit dem maximalen Ausdünnungsparameter $s = 7$ miteinander verglichen. In beiden Fällen wird der Fehler zur exakten Lösung betrachtet und keine Rekombination verwendet. In Abbildung 4.3 ist der Unterschied dargestellt. Im linken Bild ist die Differenz zwischen der groben Lösung (schwarz) und der feinen Lösung (rot) deutlich zu sehen, wobei im rechten Bild der Unterschied so klein ist, dass er nicht zu erkennen ist.



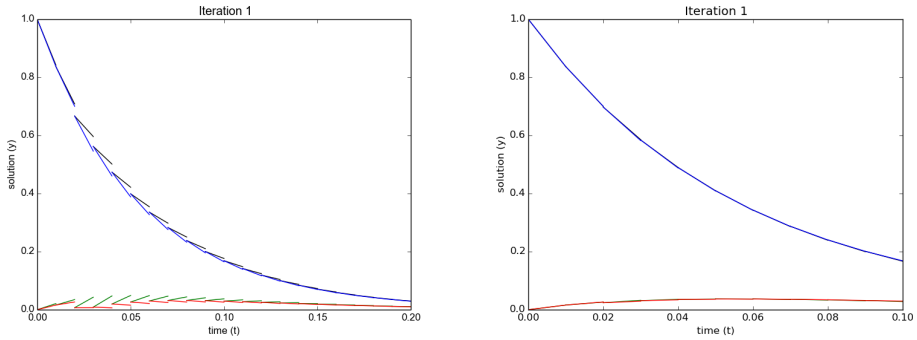
(a) (b)

Abbildung 4.3: (a) $(3, 3)$-Level Gitter mit Ausdünnungsparameter $s = 3$, (b) $(7, 7)$-Level Gitter mit Ausdünnungsparameter $s = 7$

Wie schon erwähnt wird die Kombinationstechnik als grober Zeitintegrator verwendet. Nun kann der Ausdünnungsparameter variieren. Je kleiner der Ausdünnungsparameter ist, desto

näher liegt die Lösung an der Feingitterlösung, welche die Vollgitterlösung ist. Das heißt aber nicht, dass der Fehler gegen die exakte Lösung minimiert wird. Der Unterschied zwischen Ergebnissen unterschiedlicher Ausdünnung, ist an einem $(3, 3)$-Level Gitter in Abbildunjg 4.4 gezeigt. Im Bild 4.4(b) ist zu sehen, dass der Grobgitterlösung fast keinen Unterschied zur Feingitterlösung aufweist und die Funktionen flacher abfallen. Die Differenz zur exakten Lösung ist aber etwas größer im Vergleich des Fehlers in 4.4(a).

In Kapitel 3 wurden die einzelnen Gitter mit unterschiedlichen Ausdünnungsparametern miteinander verglichen. Es hat sich gezeigt, dass das Gitter mit einem kleineren Ausdünnungsparameter mehr Gitterpunkte enthält. Dies hat zur Folge, dass die im Programm verwendeten Matrizen größer werden und dadurch der Speicherverbrauch vergrößert wird. Das heißt der Speicherverbrauch und somit auch die Rechenzeit wächst mit abnehmendem Ausdünnungsparameter. Im gegenzug sinkt die Rechenzeit mit einem größerem Ausdünnungsparameter.

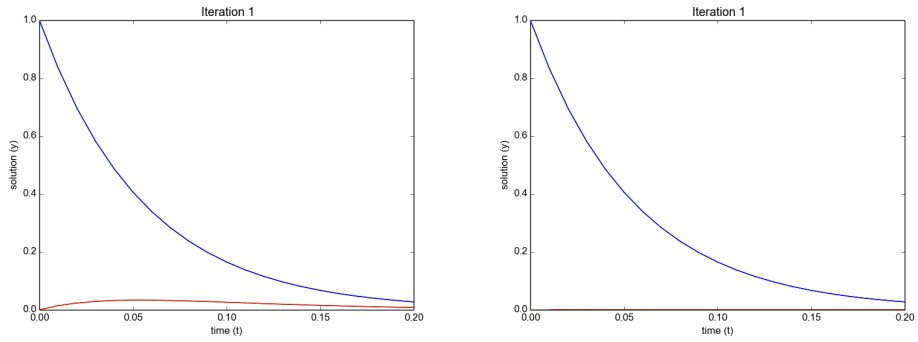


(a) (b)

Abbildung 4.4: Ein $(3, 3)$-Level Gitter mit unterschiedlichem Ausdünnungsparameter: (a) $s = 3$ (b) $s = 2$

Wie in den vorherigen Beispielen zu sehen ist, konvergiert der Fehler nicht gegen Null, sondern verkleinert sich. Der Grund dafür ist, dass allein schon bei der Diskretisierung ein Fehler zustande kommt. Mit dem Parareal-Algorithmus wird eine weitere Möglichkeit den Fehler zu bestimmen eingeführt. Man berechnet die Differenz zur Feingitterlösung. Dazu werden die Ergebnisse des feinen Integrators bei der Initialisierung, also nach dem ersten Durchlauf, gespeichert und später für die Fehlerberechnung verwendet. Um zu vermeiden, dass noch mehr große Matrizen gespeichert werden, wird hier nur der Mittelpunkt betrachtet. Durch den Vergleich der Ergebnisse mit der Feingitterlösung ist der Fehler deutlich geringer. Der Unterschied ist an den beiden roten Funktionen in Abbildung 4.5 zu erkennen. Wobei die rote Funktion der Fehler des feinen Integrators ist. Zum Vergleich wurde ein $(7, 7)$-Level Gitter mit dem maximalen Ausdünnungsparameter $s = 7$ verwendet. Es wurde keine Rekombination durchgeführt.

Bei den bisher durchgeführten Beispielen wurde mit den für die Kombinationstechnik berechnete Matrizen weiter gerechnet. Was passiert aber, wenn die Werte nach jedem Iterationsschritt aktualisiert werden? Der Gedanke dabei ist, nach jeder Kombination, die Matrizen mit den Werten der kombinierten Lösung zu überschreiben. Dabei passiert genau das Gegenteil von der Kombination. Die Matrizen bekommen die Werte der kombinierten Lösung, so dass man gleiche Ergebnisse bekommt, wenn alle Matrizen mittels der biliniaren Interpolation auf die gleiche



(a) (b)

Abbildung 4.5: (a) Differenz zur exakten Lösung (b) Differenz zur Feingitterlösung

Größe erweitert werden. Diese Technik wird Rekombination genannt. Es lässt sich behaupten, dass die Lösungen durch die Rekombination genauer berechnet werden und den Fehler damit senken. Doch folgende Tests zeigen, dass eine Aktualisierung der Werte nicht immer sinnvoll ist. Da der Unterschied feinere Gitter bildlich nicht zu erkennen ist, wird die Rekombination am Beispiel eines $(3, 3)$-Level Gitters gezeigt (Abbildung 4.6). Der Ausdünnungsparameter wurde hier auf $s = 3$ gesetzt. Rechts im Bild ist zu erkennen, dass die Lösung nach im ersten Iterationsschritt sehr stark abfällt. Nach der zweiten Iteration sinkt der Fehler zwar sehr stark ab, betrachtet man aber die mittlere absolute Differenz des groben Integrators, so ist zu erkennen, dass die Berechnung mit der Rekombination fast um das Doppelte schlechter ist. Daraus lässt sich schließen, dass es nicht immer gut ist die Technik der Rekombination anzuwenden. Aus diesem Grund werden weitere Tests ohne Rekombination durchgeführt.

Gitterlevel	**K.-Grenze**	**Iterationen**	**Zeit in sec**	**MAD grob**	**MAD fein**
3,3	0	19	3,0388479233	0,0086026118	0,0060601738
3,3	0,001	11	2,0870370865	0,0095054386	0,0077192017
3,3	0,005	4	1,0351269245	0,0116407518	0,0109252579
5,5	0	19	10,65386796	0,0006085911	0,0005205943
5,5	0,0001	12	8,8453371525	0,0006124347	0,0005721492
5,5	0,001	3	3,3884859085	0,0009320228	0,0008599163
7,7	0	19	3506,2605799	$5,60 \cdot 10^{-5}$	$4,97 \cdot 10^{-5}$
7,7	0,00001	12	2965,94414902	$5,25 \cdot 10^{-5}$	$4,97 \cdot 10^{-5}$
7,7	0,00005	4	1436,68316197	$7,17 \cdot 10^{-5}$	$7,02 \cdot 10^{-5}$
7,7	0,0001	2	906,495019913	$8,30 \cdot 10^{-5}$	$7,07 \cdot 10^{-5}$

Tabelle 4.2: Ergebnisse mit unterschiedlicher Konvergenzgrenze im Vergleich

Als Nächstes wird die Konvergenzgrenze untersucht. Mit der Konvergenzgrenze kann die Rechenzeit und die Anzahl der benötigten Iterationen reduziert werden. Für die bisher durchgeführen Experimente wurde die Konvergenzgreze auf Null gesetzt. Dabei benötigt der Algorithmus 19 Iterationen bei 20 Zeitabschnitten. Für ein $(7, 7)$-Level Gitter mit einem Ausdünnungsparameter $s = 7$ braucht der Rechner ca. 3550 Sekunden um alle 19 Iterationen zu berechnen. Durch eine geringere Anzahl an Iterationen wird eine geringere Rechenzeit erreicht. Die Anzahl der Zeitab-

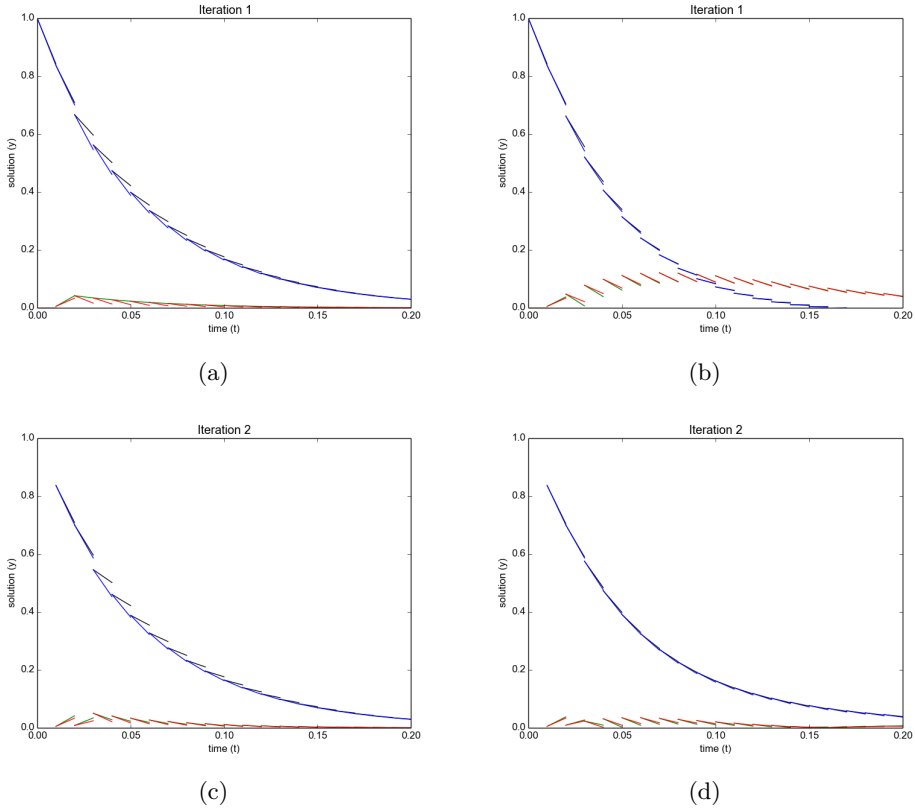


Abbildung 4.6: (a) Ergebnis ohne Rekombination nach der ersten Iteration , (b) Ergebnis mit Rekombination nach der ersten Iteration, (c) Ergebnis ohne Rekombination nach der zweiten Iteration, (d) Ergebnis mit Rekombination nach der zweiten Iteration

schnitte soll jedoch gleich bleiben. Durch die Erhöhung der Konvergenzgrenze kann eine Verminderung der Iterationen erzielt werden. Für den Vergleich werden drei verschiedene Gitterlevel betrachtet, $(3,3)$, $(5,5)$ und $(7,7)$. In allen Fällen wird der maximale Ausdünnungsparameter, also die normale Kombinationstechnik, verwendet. Die Durchführung erfolgt ohne die Technik der Rekombination. Für die Auswertungen wird die Differenz zur Feingitterlösung betrachtet. Verglichen werden die mittleren absoluten Differenzen. Geändert wird nur die Konvergenzgrenze, andere Parameter bleiben unverändert.

Mit einer Konvergenzgrenze von Null braucht jede Berechnung 19 Iterationen. Die Rechenzeit steigt logischerweise mit der Größe der Matrizen. Zum Beispiel braucht der Rechner für ein $(3,3)$-Level Gitter ca. 3 Sekunden und für ein $(7,7)$-Level Gitter ca. 3506 Sekunden. Wird die Konvergenzgrenze auf 0.001 vergrößert, so wird die Anzahl der Iterationen für das $(3,3)$-Level Gitter auf 11 Iterationen gesenkt. Der dabei entstehende Fehler vergrößert sich um ca. 0.001, sowohl für den feinen als auch für den groben Integrator. Im Vergleich dazu benötigt der Algorithmus für das $(5,5)$-Level Gitter nur noch 3 Iterationen und der Fehler steigt um ca. 0.0003. Um die Iterationen für das $(3,3)$-Level Gitter weiter zu verringern, muss die Konvergenzgrenze vergrößert werden. Setzen wir also die Konvergenzgrenze auf 0.005, so stellen wir fest, dass nur noch 4 Iterationen benötigt werden. Dabei steigt der Fehler von 0.0086 (Konvergenzgrenze gleich

Null) auf 0.011. Für feinere Gitter, also für Gitter höheren Levels, kann die Konvergenzgrenze weiter verkleinert werden. Nachfolgend wird die Konvergenzgrenze auf 0.0001 gesetzt. Für das $(3, 3)$-Level Gitter hat dieser Wert keine Auswirkungen, da die Differenzen der Werte $y^F_{k,j}$ und $y^F_{k-1,j}$ im höheren Bereich liegen. Betrachtet wird das $(5, 5)$-Level Gitter mit der Konvergenzgrenze 0.0001. Es werden zwar mehr Iterationen benötigt als bei einer Konvergenzgrenze von 0.001 aber immernoch weniger als 19 Iterationen. Für die gleiche Konvergenzgrenze braucht der Algorithmus für das $(7, 7)$-Level Gitter schon 2 Iterationen. Dies ist die minimale Anzahl an Iterationen, die erreicht werden kann. Der Rechner braucht für diese Berechnung nur ein Viertel der Zeit, die er für 19 Iterationen benötigt. Die mittlere absolute Differenz verschlechtert sich dabei um das 0,7-fache. Da der Fehler allgemein schon sehr klein ist, ist diese Verschlechterung kaum bemerkbar. Für das $(7, 7)$-Level Gitter wird die Konvergenzgrenze weiter untersucht, um zu sehen wie weit diese verringert werden kann und trotzdem schnellere Ergebnisse erzielen. Wird die Konvergenzgrenze 0.0001 um die Hälfte verkleinert, werden trotzdem sehr wenige Iterationen benötigt. Daraus folgt, dass bei einem sehr feinen Gitter die Konvergenzgrenze sehr klein gewählt werden kann, um ein schnelleres Ergebnis zu erzielen. Die wichtigsten Ergebnisse sind in Tabelle 4.2 aufgelistet.

5 Ausblick

Die Ergebnisse der durchgeführten Experimente haben gezeigt, dass mit Hilfe der Kombinationstechnik und des Parareal-Algorithmus partielle Differentialgleichungen viel schneller gelöst werden können als mit gewöhnlichen Lösungsverfahren. Durch den Parareal-Algorithmus und der Konvergenzgrenze kann die Lösung für sehr große Gitter in kurzer Zeit berechnet werden. Dafür reicht es aus die Konvergenzgrenze nur minimal zu vergrößern.

Die gewonnenen Erkenntnisse lassen sich durch weitere Untersuchungen ergänzen. Die Methode der Rekombination kann näher analysiert werden. Zum Beispiel kann die Häufigkeit der Rekombinationsmethode und deren Auswirkungen auf das Ergebnis untersucht werden. Ein weiterer interessanter Punkt ist, für den feinen Zeitintegrator ebenfalls die Kombinationstechnik mit Ausdünnung zu verwenden. Dabei soll beachtet werden, dass der Ausdünnungsparameter für die Feingitterlösung kleiner gewählt wird als für die Grobgitterlösung. Ausgehend davon wäre zu fragen, welcher Wert zur Berechnung des Fehlers dabei geeignet ist. Welche Auswirkung hat es auf die Rechenzeit und den dabei entstehenden Fehler? In dieser Bachelorarebeit wurde die Kombinationstechnik auf den zweidimensionalen Raum beschränkt. Für weitere Untersuchungen lässt sich die Kombinationstechnik auf mehrdimensionale Räume erweitern. Aufgrund des beschränkten Speicherplatzes des zur Verfügung stehenden Rechners, konnten die Tests für maximal $(7, 7)$-Level-Gitter durchgeführt werden. Bietet sich eine Möglichkeit an, Berechnungen auf einem leistungsstarken Rechner auszuführen, so kann die Maschenweite des Gitters weiter verkleinert werden.

Eine Optimierungsmöglichkeit besteht darin, die durch den Parareal-Algorithmus ermöglichte Zeitparallelisierung auszunutzen. Hierbei werden alle Zeitabschnitte parallel ausgeführt. Dadurch kann die Rechenzeit enorm minimiert werden. Dafür bietet die Parareal-Implementierung von Martin Schreiber [6] mehrere Möglichkeiten, wie zum Beispiel eine MPI-Parallelisierung mit oder ohne Sliding Window.

Literaturverzeichnis

[1] H.-J. Bungartz, S. Zimmer, M. Buchholz, and D. Pflüger. *Modellbildung und Simulation: Eine anwendungsorientierte Einführung.* Springer Spektrum, 2 edition, 2013. ISBN 978-3-642-37655-9.

[2] M. J. Gander. 50 years of time parallel time integration. Technical report, Section de Mathématiques, Université de Genève, 2015.

[3] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In P. de Groen and R. Beauwens, editors, *Proceedings of the IMACS International Symposium on Iterative Methods in Linear Algebra: Brussels, Belgium, 2 - 4 April, 1991.* IMACS, North Holland, 1992.

[4] C. Kranz. Untersuchungen zur kombinationstechnik bei der numerischen strömungssimulation auf versetzten gittern. Technical report, Technische Universität München, 2002.

[5] J.-L. Lions, Y. Maday, and G. Turinici. A "parareal" in time discretization of pde's. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 332(7):661—-668, 2001.

[6] M. Schreiber et al. Parallelization in time: An hpc and software-driven overview. Technical report, University of Exeter, 2015.

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 02.08.2016

Unterschrift