Präkonditionierung der gekoppelten One-Shot Iteration bei Discontinuous Galerkin für Navier-Stokes

Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Mathematik

eingereicht von: Max Sagebaum

geb.: am 29.07.1986 in: Berlin

Betreuer: Prof. Dr. Nicolas R. Gauger

Berlin, den 11. Januar 2012

Ich möchte mich bei allen Menschen bedanken,
die mich bei dieser Arbeit
und
meinem bisherigem Lebensweg
unterstützt haben.

Besonders möchte ich mich bei

Herrn Ortwin Fromm

bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

0. Einleitung

Die Arbeit wird im Rahmen des DGHPOPT Projekts geschrieben. Das Ziel des Teilprojekts an der RWTH Aachen ist die Implementierung einer One-Shot Methode, die während der Berechnung eine Adaption des Gitters durchführt.

Das Ziel dieser Arbeit ist es den Präkonditionierer, den wir für die Konvergenz in einem One-Shot Verfahren benötigen, zu berechnen. Der Code, in dem das One-Shot Verfahren aufgebaut werden soll, ist der Padge Code vom DLR Braunschweig.

Der Präkonditionierer setzt sich aus den einzelnen Komponenten in der One-Shot Iteration zusammen. Die Komponenten beschreiben die Updates im Primalen, im Adjungierten und im Design. Die Updates für diese drei Komponenten enthalten Ableitungen bezüglich der Design- und der Zustandsvariablen, die auch für den Präkonditionierer eine Rolle spielen. Für die Berechnung des Präkonditionierers muss man die Ableitung der zu Grunde liegenden Gleichungen des Verfahrens im Padge Code beschaffen.

Der Padge Code ist ein Strömungslöser für die kompressiblen Navier-Stokes-Gleichungen, der diese Gleichungen mit einem diskontinuierlichen Galerkin Verfahren löst. Die Berechnung von Ableitungen ist im Page Code nicht vorgesehen. Um den Präkonditionierer zu erhalten, müssen wir den Code so umschreiben, dass er auch Ableitungen berechnen kann. Die Ableitungen sollen mit einem Tool zum Automatischen Differenzieren bereitgestellt werden.

Bisher wurde das One-Shot Verfahren nur für Finite Volumen Methoden implementiert. Bei diesen Methoden ist eine Adaption des Gitters nur schwer möglich. Für die diskontinuierlichen Galerkin Verfahren hat man eine sehr gute Grundlage, eine Adaption des Gitters basierend auf der Adjungierten für ein bestimmtes Zielfunktional herzuleiten.

In der Arbeit werden deshalb die Gebiete des Automatischen Differenzierens, der diskontinuierlichen Galerkin Verfahren und der One-Shot Optimierung behandelt. Die ersten 3 Kapitel beschreiben eine Einführung in diese Gebiete und geben eine Übersicht für die mathematiche Herangehensweise. Dabei beschreiben wir aber alle Grundlagen, die wir für die Berechnung des Präkonditionierers brauchen.

Kapitel 4 beschreibt den Aufbau des Padge Codes und die Schritte für die Bereitstellung des Präkonditionierers. Das Kapitel enthält auch eine Übersicht über die Funktionen und die Funktionalität des Padge Codes, die für den Präkonditionierer benötigt werden.

Im letztem beiden Kapitel werden die Ergebnisse und die Validierungen für die Ableitung des Padge Codes präsentiert. Hier werden wir auch die Implementierung

für die Berechnung des Präkonditionierers beschreiben.

1. Automatisches Differenzieren

In vielen mathematischen Problemstellungen und Lösungsansätzen in der Numerik sowie in der Optimierung werden die Ableitungen von Funktionen benötigt. Die Ableitung einer Funktion kann man auf verschiedene Art und Weise beschaffen.

Die traditionelle Variante ist das herleiten der Ableitung auf Basis der Funktionsformel. Dadurch berechnet man die exakte Ableitung in Maschinengenauigkeit. Aber der Ausdruck für die Ableitung kann selbst bei einfachen Formeln sehr schnell zu einer Größe anwachsen, die man nicht mehr handhaben kann. Nachdem man den Ausdruck für die Ableitung hergeleitet hat, fehlt noch die Implementation der Formel für die Ableitung in einer Umgebung oder Programmiersprache, sodass man diese Ableitung in seinen Berechnungen verwenden kann. Der dadurch entstandene Aufwand und die benötigte Zeit sind meist sehr groß und werden nur deshalb betrieben weil bessere Alternativen fehlen.

Die einfachste Alternative zum direkten Herleiten der Ableitung sind die Finiten Differenzen. Die Finiten Differenzen werden berechnet, indem man die Funktion an zwei Punkten auswertet und durch den Abstand der beiden Punkte teilt. In der Theorie kann man dadurch die Ableitung in einer beliebigen Genauigkeit approximieren. Auf dem Rechner ist der Abstand zwischen den Punkten nicht beliebig klein wählbar. Der Fehler durch die Auslöschung stört die Finite Differenz immer mehr je kleiner der Abstand wird. Durch einen zu großen Abstand der beiden Punkte wird der Fehler auch immer größer. Für eine optimale Genauigkeit der Finiten Differenzen muss ein optimaler Abstand der zwei Punkte gefunden werden. Dieses Optimum ist von Funktion zu Funktion unterschiedlich und kann auch je nach Anordnung der Punkte variieren. Dafür lassen sich die Finiten Differenzen nahezu ohne weiteren Aufwand implementieren, wenn man die Funktion schon in einem Code vorliegen hat. Die Genauigkeit der Ableitung ist bei den Finiten Differenzen nicht so gut wie bei der analytischen Ableitung.

Als dritten Weg möchten wir in diesem Kapitel das Automatische Differenzieren (AD) herleiten. Hier verändert man das Programm so, dass beim Durchlaufen des Codes die Ableitung in eine Richtung automatisch mit berechnet wird. Diese Ableitung ist die exakte Ableitung der Funktion in Maschinengenauigkeit. Der Aufwand für das Einbringen von AD in den Code ist je nach Code unterschiedlich. Es kommt hauptsächlich darauf an, wie stark der Typ, der für die Berechnungen benutzt wird, abstrahiert ist. In C++ kann er als Templateparameter, Typendefinition oder als Basistyp wie double gegeben sein. Bei einem Templateparameter und der Typendefinition sollte es reichen diese zu ändern. Wenn ein Basistyp im Code benutzt wird, muss man den Code umschreiben. Wie man das am besten macht, ist dem Programmierer überlassen. Wenn man den Code auf eine der gerade genannten Arten differenziert

hat, ergibt sich der Vorteil, dass man diese Arbeit nicht wiederholen muss. Jede Erweiterung das Codes wird automatisch mit in die Differenziation einbezogen, wenn man sich an die Regeln für das Verwenden eines AD Tools hält.

Mit AD ergibt sich die Möglichkeit, für einen Code die exakte Ableitung in Maschinengenauigkeit zu berechnen. Der damit verbundene Aufwand ist sehr gering. Der Aufwand kann einmalig sehr hoch sein, wenn man keine Möglichkeit hat, den Rechentypen umzudefinieren.

Für die Herleitung werden zuerst einige Definitionen gegeben, mit denen wir den Vorwärtsmodus von AD herleiten. Danach wird der Rückwärtsmodus eingeführt. Zum Abschluss wird auf Besonderheiten eingegangen, die wir für den Padge Code benötigen. In dem gesamten Kapitel über das Automatische Differenzieren halte ich mich an das Buch [4] von A. Griewank und A. Walther.

1.1. Voraussetzungen

Die folgenden Definitionen stammen aus dem Königsberger [9].

Definition 1.1 (Richtungsableitung)
Es sei $f = (f_1, f_2, ..., f_m) : U \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion. Dann ist die Richtungsableitung von f in Richtung $d \in \mathbb{R}^n$ an der Stelle $x_0 \in \mathbb{R}^n$ definiert durch

$$\frac{\partial f}{\partial d}(x_0) := \lim_{h \to 0} \frac{f(x_0 + h \cdot d) - f(x_0)}{h} \ .$$

Definition 1.2 (Partielle Ableitung)
Es sei $f = (f_1, f_2, ..., f_m) : U \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion. Dann sind die partielle Ableitung an der Stelle $x_0 \in \mathbb{R}^n$ gegeben durch die Richtungsableitung in die Richtung $e_i \in \mathbb{R}^n$. e_i ist für alle $i = 1...n$ definiert durch $[e_i]_j = \delta_{ij}, j = 1...n$. Wir erhalten

$$\frac{\partial f}{\partial x_i}(x_0) := \frac{\partial f}{\partial e_i}(x_0) = \lim_{h \to 0} \frac{f(x_0 + h \cdot e_i) - f(x_0)}{h} \qquad i = 1...n \ .$$

Definition 1.3 (Gradient)
Es sei $f : U \subset \mathbb{R}^n \to \mathbb{R}$ eine differenzierbare Funktion. Der Gradient von f an der Stelle $x_0 \in \mathbb{R}^n$ ist der Vektor der partiellen Ableitungen von f.

$$\nabla f(x_0) := gradf(x_0) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x_0) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_0) \end{pmatrix}$$

Definition 1.4 (Jacobi Matrix)
Es sei $f = (f_1, f_2, ..., f_n) : U \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion. Die

Jacobi Matrix von f an der Stelle $x_0 \in \mathbb{R}^n$ ist durch die partiellen Ableitungen von f definiert.

$$f'(x_0) := \frac{df}{dx} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_0) & \frac{\partial f_1}{\partial x_2}(x_0) & \cdots & \frac{\partial f_1}{\partial x_n}(x_0) \\ \frac{\partial f_2}{\partial x_1}(x_0) & \frac{\partial f_2}{\partial x_2}(x_0) & \cdots & \frac{\partial f_2}{\partial x_n}(x_0) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x_0) & \frac{\partial f_m}{\partial x_2}(x_0) & \cdots & \frac{\partial f_m}{\partial x_n}(x_0) \end{pmatrix}$$

Für eine reelle Funktion $f : U \subset \mathbb{R}^n \to \mathbb{R}$ gilt folgende Beziehung zwischen Jacobi-Matrix und Gradient

$$\nabla f(x) = \frac{df}{dx}^T(x) \ .$$

Definition 1.5 (Kettenregel)
Es seien $X \subset \mathbb{R}^n, Y \subset \mathbb{R}^m, Z \subset \mathbb{R}^k$ offen. Sei $f : X \to Y$ und $g : Y \to Z$ zwei reelle Funktionen, dann können wir für die Funktion $h := g \circ f$ die Ableitung an der Stelle $x \in X$ als die Kettenregel

$$h'(x) = (g \circ f)(x) = g'(f(x)) \cdot f'(x)$$

definieren.

Damit haben wir die grundlegenden Begriffe für die Ableitungen einer Funktion eingeführt. Nun gehen wir auf den Begriff der Differentialform oder kurz 1-Form ein. Diese Begriffe werden dann für den Spezialfall einer reellen Funktion erklärt und genauer definiert.

Definition 1.6 (Differentialform)
Unter einer Differentialform versteht man eine Abbildung

$$\omega : U \subset \mathbb{R}^n \to L(\mathbb{R}^n, \mathbb{R}^m),$$

die jedem $x \in U$ eine lineare Abbildung $\omega(x) : \mathbb{R}^n \to \mathbb{R}^m$ zuordnet.

Für eine Funktion $f = (f_1, f_2, ..., f_n) : U \subset \mathbb{R}^n \to \mathbb{R}^m$ können wir eine Differentialform bilden indem wir die Jacobi-Matrix als lineare Abbildung heranziehen. Wir erhalten dadurch die Differentialform

$$\begin{aligned} \frac{df}{dx}(x) : \mathbb{R}^n &\to \mathbb{R}^m \\ h &\mapsto \frac{df}{dx}(x)h \ . \end{aligned}$$

Wenn wir für die Koordinatenfunktionen $\xi_i : \mathbb{R}^n \to \mathbb{R}, \xi_i(x) = x_i$ für $i = 1...n$ die Differentialform bilden, bekommen wir die linearen Abbildungen

$$dx_i(x)h := \frac{d\xi_i(x)}{dx}h = h_i \ . \tag{1.1}$$

Sei nun ω eine beliebige 1-Form auf einer Menge U mit $\omega(x) : \mathbb{R}^n \to \mathbb{R}^m$. Diese 1-Form können wir an jeder Stelle $x \in U$ in die Richtung der Standardbasis $e_1, e_2, ..., e_n \in \mathbb{R}^n$ auswerten. Dadurch erhalten wir die Funktionen

$$\begin{aligned} a_i : U &\to \mathbb{R}^m \qquad i = 1...n \\ x &\mapsto \omega(x)e_i \ . \end{aligned}$$

Den Ausdruck $\omega(x)h$ können wir mit der Liniarität von $\omega(x)$ und der Identität $d\xi_i(x)h = h_i$ in Abhängigkeit der Koordinatenfunktionen darstellen. Durch die Umformung

$$\begin{aligned} \omega(x)h &= \sum_{i=1}^{n} \omega(x)e_i \cdot h_i = \sum_{i=1}^{n} a_i(x) \cdot h_i = \sum_{i=1}^{n} a_i(x) \cdot d\xi_i(x)h \\ &= \sum_{i=1}^{n} a_i(x) \cdot dx_i(x)h \end{aligned}$$

erhalten wir eine Darstellung in den a_i's. Die Kurzschreibweise für diese Darstellung lautet

$$\omega = a_1 dx_1 + ... + a_n dx_n \tag{1.2}$$

oder

$$\omega = A \cdot dx \text{ mit } A = (a_1, a_2, ..., a_n) \text{ und } dx = \begin{pmatrix} dx_1 \\ \vdots \\ dx_n \end{pmatrix} \ . \tag{1.3}$$

Die Funktionen $a_1, ..., a_n$ heißen Koeffizienten der 1-Form ω bezüglich $dx_1, ..., dx_n$. Mit dieser Darstellung können wir nun die Differentialform einer reellen Funktion besser definieren.

Satz 1.7 (Differentialform einer reellen Funktion)
Es sei $f = (f_1, f_2, ..., f_m) : U \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion. Die 1-Form df hat bezüglich $dx_1, ..., dx_n$ die Koeffizienten $a_i = \frac{\partial f}{\partial x_i}$ und damit die Darstellung

$$df = \frac{\partial f}{\partial x_1} dx_1 + \cdots + \frac{\partial f}{\partial x_n} dx_n$$

oder

$$df = \frac{df}{dx} dx \ .$$

Beweis. Wir definieren wie oben zu f die 1-Form $\omega(x)h := \frac{df}{dx}(x)h$ für alle Punkte $x \in X$.

Durch die Definition von a_i erhalten wir $a_i(x) = w(x)e_i = \frac{df}{dx}(x)e_i = \frac{\partial f}{\partial x_i}$ für alle $i = 1...n$. □

Mit der in Satz 1.7 definierten 1-Form einer reellen Funktion, haben wir eine gute Handhabe, wie wir die lineare Funktion in jedem Punkt x von einer Menge U darstellen können. Die 1-Formen $dx_1, ..., dx_n$ geben uns die Möglichkeit, das Differential in eine Richtung h auszuwerten. Diese Auswertung ist, wie in (1.1) definiert, nur die Abbildung auf die i-te Koordinate von h. Eine noch zu klärende Frage ist, wie sich die Differentiale zweier Funktionen verhalten, die wir miteinander verknüpfen.

Satz 1.8 (Kettenregel für Differentialformen)
Es seien $X \subset \mathbb{R}^n, Y \subset \mathbb{R}^m, Z \subset \mathbb{R}^k$ offen. Sei $f : X \to Y, g : Y \to Z$ zwei reelle Funktionen, dann können wir für die Funktion $a := g \circ f$ das Differential wie folgt definieren:

$$x \in X \qquad da(x) = dg(f(x)) \circ df(x)$$

Beweis. Wir definieren wie oben zu a die 1-Form $da := \frac{da}{dx}dx$ für alle Punkte $x \in X$.
Mit der Definition der Kettenregel für reelle Funktionen bekommen wir

$$da(x) = \frac{dg}{dy}(f(x))\frac{df}{dx}(x)dx \ .$$

Wenn wir die Differentialform $dg = \frac{dg}{dy}dy$ an der Stelle $f(x)$ auswerten erhalten wir die lineare Abbildung $dg(f(x)) = \frac{dg}{dy}(f(x))dy$. dy ist dabei die Identität von einem Vektor $h \in \mathbb{R}^m$. Wir erhalten also $dy(h) = h$. Dadurch gilt $dy(df(x)(h')) = \frac{df}{dx}dx(h')$ mit $h' \in \mathbb{R}^n$ und dem Differential $df = \frac{df}{dx}dx$. Jetzt können wir die Ersetzung

$$\begin{aligned} da(x)(h) &= \frac{dg}{dy}(f(x))\frac{df}{dx}(x)dx(h) \\ &= \frac{dg}{dy}(f(x))dy(df(x)(h)) \\ &= dg(f(x)) \circ df(x)(h) \qquad \text{mit } h \in \mathbb{R}^n \text{ bel.} \end{aligned}$$

vornehmen. Damit haben wir die Identität

$$da(x) = dg(f(x)) \circ df(x) \qquad \text{für } x \in X$$

gezeigt. □

In dem Beweis haben wir die Identität $dy(df(x)(h')) = \frac{df}{dx}dx(h')$ verwendet. Da diese Gleichung für alle $h' \in \mathbb{R}^n$ gilt, können wir auch $dy = \frac{df}{dx}dx$ schreiben.

1.2. Der Vorwärtsmodus des Automatischen Differenzierens

Um ein Programm oder einen Code differenzieren zu können, müssen wir zuerst beschreiben wie wir das Programm mathematisch darstellen wollen. Angenommen das Programm berechnet eine Funktion $f : X \subset \mathbb{R}^n \to Y \subset \mathbb{R}^m$. Dann haben wir in dem

Programm gegebene Eingabewerte x aus der Menge X, die Ausgabewerte y in der Menge Y und alle Zwischenwerte u in einer Menge U. Die Zwischenwerte entstehen bei der Berechnung von y aus den Eingabewerten x. Für die Zwischenwerte u und den Ausgabewerte y nehmen wir an, dass sie durch Elementaroperationen ϕ berechnet werden. Wir definieren nun zuerst den Raum der Programmauswertung.

Definition 1.9 (Programmraum)
X, Y und U werden definiert als:

- $X \subset \mathbb{R}^n$ ist der Raum der Eingabewerte.
- $Y \subset \mathbb{R}^m$ ist der Raum der Ausgabewerte.
- $U \subset \mathbb{R}^l$ ist der Raum der Zwischenwerte.

Dann ist

$$V := X \times U \times Y \subset \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R}^m = \mathbb{R}^{n+l+m}$$

der Raum der Programmauswertung. Den Vektor $v \in V$ werden wir zum Teil in den Komponenten $v = (x, u, y)$ schreiben.

Mit der Definition des Programmraumes kann man nun die Elementaroperationen eines Programms näher beschreiben.

Definition 1.10 (Elementarabbildung, Elementaroperation)
Eine Elementarabbildung ist eine Funktion $\Phi_k : V \to V$ mit $k \in \{1, ..., l+m\}$. Diese Elementarabbildung ist definiert durch

$$\Phi_k(v) = (v_1, ..., v_n, v_{n+1}, ..., v_{n+k-1}, \phi_k(v_1, ..., v_n, v_{n+1}, ..., v_{n+k-1}, 0, 0, ..., 0), 0, ..., 0)$$

Zu der Elementarabbildung gehört die Elementaroperation $\phi_k : V \to \mathbb{R}$, diese hängt nur von den ersten $n + k - 1$ Element von V ab.

Für unsere Funktion f benötigen wir noch die Einbettung von X in V

$$\begin{aligned} I_x : X &\to V \\ x &\mapsto (x, 0, 0)^T \end{aligned}$$

und die Projektion von V nach Y

$$\begin{aligned} P_y : \quad V &\to Y \\ (x, u, y)^T &\mapsto y \ . \end{aligned}$$

Damit können wir nun im mathematischen Sinne die Funktion f definieren, die von unserem Programm ausgewertet wird.

Definition 1.11 (Programmfunktion)
Es seien $x \in X$ die Eingabewerte des Programmes, $y \in Y$ die Ausgabewerte. Das Programm besteht aus $k \in \mathbb{N}$ Elementaroperationen, die die Funktion $f : X \to Y$ berechnen.

Dann sagen wir f ist die Programmfunktion wenn die Gleichung

$$f(x) = P_y \circ \Phi_k \circ \Phi_{k-1} \circ ... \circ \Phi_2 \circ \Phi_1 \circ I_x(x)$$

gilt.

Durch die Programmfunktion haben wir nun die Möglichkeit, ein Programm in einem mathematischen Kontext zu betrachten. Die spannende Frage lautet: Können wir *df* berechnen und wie sieht *df* aus? Dazu müssen wir noch eine entscheidende Annahme treffen.

Annahme 1.12 (Differenzierbarkeit)
Die Elementaroperationen $\phi_i : V \to V$ sind bis zu einem gewissen Grad d differenzierbar.

Mit dieser Annahme hat sich die Frage nach der Berechenbarkeit von f erübrigt. Durch die Differenzierbarkeit von ϕ_i sind die Φ_i auch d-mal differenzierbar. Die Kettenregel besagt, dass die Verknüpfung von differenzierbaren Funktionen auch differenzierbar ist. Wir wissen also, dass f eine differenzierbare Funktion ist und können *df* bilden. Dazu benötigen wir die Differentiale dI_x, dP_y und $d\Phi_i$.

Für $I_x(x) = \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}$ erhalten wir:

$$dI_x = \frac{dI_x}{dx} dx = \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} dx \ .$$

Für $P_y(x, u, y) = y$ erhalten wir:

$$dP_y = \frac{dP_y}{dv} dv = \begin{pmatrix} 0 & 0 & I \end{pmatrix} dv \ .$$

Für die Elementarabbildung Φ_i und die dazugehörige Elementaroperation ϕ_i bilden wir für $\Phi_i(v) = (v_1, ..., v_n, v_{n+1}, ..., v_{n+i-1}, \phi_i(v_1, ..., v_n, v_{n+1}, ..., v_{n+i-1}, 0, ..., 0), 0, ..., 0)$ zuerst die Jacobi-Matrix der einzelnen Komponenten. Dabei bezeichnen wir mit $\Phi_{ij}(v) := (\Phi_i(v))_j$.

- Sei j < i + n, dann ist $\Phi_{ij}(v) = v_j$ und somit

$$\frac{d\Phi_{ij}}{dv} = (0, ..., 0, \underbrace{1}_{\text{j-te}}, 0, ..., 0) \ .$$

- Sei j > i + n, dann ist $\Phi_{ij}(v) = 0$ und somit

$$\frac{d\Phi_{ij}}{dv} = (0, ..., 0) \ .$$

- Sei i + n = j, dann ist $\Phi_{ij}(v) = \phi_i(v)$ und somit

$$\frac{d\Phi_{ij}}{dv} = \frac{d\phi_i}{dv} = \left(\frac{\partial \phi_i}{\partial v_1}, ..., \frac{\partial \phi_i}{\partial v_{i-1}}, \underbrace{0}_{\text{j-te}}, 0, ..., 0 \right) \ .$$

Das Differential von Φ_i können wir nun schreiben als:

$$d\Phi_i = \frac{d\Phi_i}{dv} dv = \begin{pmatrix} I & 0 & 0 \\ \frac{d\phi_i}{dv} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} dv \ .$$

Zuerst wollen wir die Verkettung von zwei Elementarabbildungen betrachten. Sei dazu $\Phi_1 : V \to V$ und $\Phi_2 : V \to V$ zwei Elementarabbildungen. Diese werten wir an der Stelle $v \in V$ aus. Wir erhalten dadurch $w := \Phi_1(v)$, $u := \Phi_2(w)$ und die Verknüpfung $u = \Phi_2(\Phi_1(v))$. Die entsprechenden Differenziale lauten

- $dw = d\Phi_1(v)dv = \frac{d\Phi_1}{dv} dv$
- $du = d\Phi_2(w)dw = \frac{d\Phi_2}{dw} dw$
- $du = d\Phi_2(\Phi_1(v))d\Phi_1 dv = \frac{d\Phi_2}{dw} \frac{d\Phi_1}{dv} dv$

Wir sehen nun aufgrund der Kettenregel, dass es egal ist, ob wir die Jacobi-Matrix von $\Phi_2 \circ \Phi_1$ auswerten oder die Jacobi-Matrizen $d\Phi_1$ und $d\Phi_2$ einzeln auswerten. Für die Auswertung der Funktion und der Ableitung ergibt sich später eine sehr effiziente Darstellung. Wenn wir das Differential von f bilden erhalten wir die Formel

$$\begin{aligned} dy(x) = df(x) &= \frac{df}{dx}(x)dx = \frac{dP_y \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \circ I_x}{dx}(x)dx \\ &= \frac{dP_y \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2 \circ \Phi_1}{dv}(I_x(x)) \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} dx \\ &= \frac{dP_y \circ \Phi_k \circ \Phi_{k-1} \circ \cdots \circ \Phi_2}{dv}(\Phi_1 \circ I_x(x)) \frac{d\Phi_1}{dv}(I_x(x)) \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} dx \\ &\vdots \\ &= \frac{dP_y}{dv} \frac{d\Phi_k}{dv}(\Phi_{k-1} \circ \cdots \circ \Phi_1 \circ I_x(x)) \cdot \ldots \cdot \frac{d\Phi_1}{dv}(I_x(x)) \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} dx \ . \end{aligned}$$

Diese Auswertung können wir als Iteration schreiben

$$\begin{aligned}
v_{(0)} &= I_x(x) & dv_{(0)} &= \frac{dI_x}{dx}(x)dx \\
v_{(1)} &= \Phi_1(v_{(0)}) & dv_{(1)} &= \frac{d\Phi_1}{dv}(v_{(0)})dv_{(0)} \\
v_{(2)} &= \Phi_2(v_{(1)}) & dv_{(2)} &= \frac{d\Phi_2}{dv}(v_{(1)})dv_{(1)} \\
&\vdots & &\vdots \\
v_{(i)} &= \Phi_i(v_{(i-1)}) & dv_{(i)} &= \frac{d\Phi_i}{dv}(v_{(i-1)})dv_{(i-1)} \\
&\vdots & &\vdots \\
v_{(k)} &= \Phi_l(v_{(k-1)}) & dv_{(k)} &= \frac{d\Phi_k}{dv}(v_{(k-1)})dv_{(k-1)} \\
y &= P_y(v_{(k)}) & dy &= \frac{dP_y}{dv}dv_{(k)} \ .
\end{aligned} \tag{1.4}$$

Die $v_{(i)}$ sind Vektoren in V und die $dv_{(i)}$ sind 1-Formen in $L(\mathbb{R}^{n+l+m}, \mathbb{R}^{n+l+m})$. In dieser Auflistung sehen wir, dass alle Jacobi-Matrizen der Elementarabbildungen einzeln ausgewertet werden. Wir brauchen nicht die Verknüpfung aller Jacobi-Matrizen zu berechnen sondern nur die Verknüpfung mit einem Vektor.

Als ein kleines Beispiel wollen wir den Einheitsvektor e_1 und f als eine reellwertige Funktion wählen. Wenn wir nun die 1-Form dx in die Richtung e_1 auswerten, erhalten wir $dx = e_1$. Die Kettenregel besagt, dass wir durch die Berechnung der Vektoren $v_{(i)}$ und $dv_{(i)}$ den Ausdruck $dy = \frac{df}{dx}e_1$ berechnen. Da wir für f angenommen haben, dass die Funktion reellwertig ist, erhalten wir nach der Definition von $\frac{df}{dx}$

$$dy = \frac{df}{dx}e_1 = \nabla f(x)^T e_1 = \frac{\partial f}{\partial x_1}$$

also die Ableitung von f nach der ersten Variablen.

Mathematisch sind das Ergebnis des Beispiels und die Auflistung der Kettenregel als Iteration nichts Weltbewegendes. Durch eine andere Interpretation dieser Ergebnisse erhalten wir aber den entscheidenden Sprung zum Automatischen Differenzieren und damit ein neues und sehr mächtiges Tool in der Numerik.

Wenn wir auf einen Rechner schauen, der dieses Programm auswertet, erhalten wir eine Vorschrift, wie wir die Anweisungen für die Ableitung zu einer normalen Anweisung dazu schreiben. In einem Programm werden wir aber nicht die Auswertungen der Elementarabbildungen vorfinden, sondern die Auswertungen der Elementaroperationen. Das Programm wird auch nicht nach jeder Elementaroperation einen neuen Vektor $v_{(i)}$ erzeugen, sondern auf den bestehenden Variablen arbeiten und dabei eine neue Variable setzen.

Dadurch können wir den Ablauf in einem Programm kürzer als in der Iteration für die Kettenregel beschreiben. Auf dem Rechner kann man nicht mit den 1-Formen rechnen, deshalb müssen wir eine Ersetzung vornehmen.

	Funktion	Normale Anweisung	AD Anweisung
Input:	$I_x(x)$	$v_i = x_i, \forall i = 1...n$	$\dot{v}_i = \dot{x}_i, \forall i = 1...n$
	$\Phi_1(v)$	$v_{n+1} = \phi_1(v)$	$\dot{v}_{n+1} = \frac{\partial \phi_1}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_1}{\partial v_n}\dot{v}_n$
	$\Phi_2(v)$	$v_{n+2} = \phi_2(v)$	$\dot{v}_{n+2} = \frac{\partial \phi_2}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_2}{\partial v_{n+1}}\dot{v}_{n+1}$
	$\vdots$		
	$\Phi_i(v)$	$v_{n+i} = \phi_i(v)$	$\dot{v}_{n+i} = \frac{\partial \phi_i}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_i}{\partial v_{n+i-1}}\dot{v}_{n+i-1}$
	$\vdots$		
	$\Phi_k(v)$	$v_{n+k} = \phi_k(v)$	$\dot{v}_{n+k} = \frac{\partial \phi_k}{\partial v_1}\dot{v}_1 + ... + \frac{\partial \phi_k}{\partial v_{n+k-1}}\dot{v}_{n+k-1}$
Output:	$P_y(v)$	$y_i = v_{l-m+i}, \forall i = 1...m$	$\dot{y}_i = \dot{v}_{l-m+i}, \forall i = 1...m$

Abbildung 1.1.: AD Vorwärtsmodus

Sei $\dot{x} \in \mathbb{R}^n$ eine beliebige Richtung, in die wir unser Programm ableiten wollen. Dazu ist

$$\xi(x) = x \in \mathbb{R}^n \text{die Koordinatenabildung in x und}$$
$$dx(h) = h \in \mathbb{R}^n \text{das Differential der Koordinatenabildung.}$$

Für die Richtung $\dot{x}$ erhalten wir $dx(\dot{x}) = \dot{x}$. Damit können wir die Umwandlung eines Differentials in die AD Vorwärtsform definieren.

Definition 1.13 (AD Vorwärtsform eines Differentials)
Es sei $\phi : X \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion, $x \in X$ und $\dot{x} \in \mathbb{R}^n$. Dann ist die Auswertung des Differentials $d\phi$ von ϕ an der Stelle x in die Richtung $\dot{x}$ die Richtungsableitung von ϕ an der Stelle x in Richtung $\dot{x}$. Wir definieren $\dot{y}$ als

$$\dot{y} := dy(\dot{y}) := d\phi(\dot{x}) = \frac{d\phi}{dx}dx(\dot{x}) = \frac{d\phi}{dx}\dot{x},$$

wobei $\dot{y}$ von x abhängt.

In Abbildung 1.1 ist der Vorwärtslauf des Automatischen Differenzierens auf Basis der Iteration der Kettenregel in (1.4) beschrieben. In einem Programm werden zuerst die Eingabewerte gesetzt. Beim Durchlaufen der Operationen wird zu jeder Operation die Abgeleitete ausgeführt und zum Schluss werden die Ausgabewerte zurückgegeben. Als Rückgabe erhalten wir die Auswertung der Funktion an der Stelle x und die Richtungsableitung der Funktion in Richtung $\dot{x}$. Die Ableitung wird bei einem Vorwärtsdurchlauf des Programms gebildet, daher nennen wir diesen AD-Modus den Vorwärtsmodus. Dieses Ergebnis halten wir in einem Satz fest.

Elementaroperation	Vorwärtsauswertung AD
$u = v + w$	$\dot{u} = \dot{v} + \dot{w}$
$u = v * w$	$\dot{u} = w \cdot \dot{v} + v \cdot \dot{w}$
$u = \frac{1}{v}$	$\dot{u} = -\frac{1}{v^2} \cdot \dot{v}$
$u = \sin(v)$	$\dot{u} = \cos(v) \cdot \dot{v}$
$u = e^v$	$\dot{u} = e^v \cdot \dot{v}$
$u = \ln(v)$	$\dot{u} = \frac{1}{v} \cdot \dot{v}$
$u = v^p$	$\dot{u} = pv^{p-1} \cdot \dot{v}$

Abbildung 1.2.: AD Vorwärtsmodus für ausgewählte Operationen

Satz 1.14 (AD Vorwärtsmodus)
Es sei ein Programm durch $k \in \mathbb{N}$ Elementaroperation $\phi_i : V \to V$ mit $i = 1...k$ gegeben. Es seien $x \in X$ die Eingabewerte und $\dot{x} \in \mathbb{R}^n$ eine Richtung. Dann erhalten wir durch den in Abbildung 1.1 definierten Ablauf des Programmes die Richtungsableitung in die Richtung $\dot{x}$ an der Stelle x.

Falls das Programm für eine Funktion $f : X \subset \mathbb{R}^n \to \mathbb{R}^m$ steht, erhalten wir mit

$$\dot{y} = \frac{df}{dx}(x)\dot{x}$$

die Richtungsableitung von f in Richtung $\dot{x}$.

Beweis. Wenn wir uns jede Zeile in der Iteration für die Kettenregel in 1.4 anschauen, brauchen wir nur die Elementaroperation für jede Elementarabbildung ausführen. Dadurch erhalten wir die entsprechende Zeile in Abbildung 1.1. □

In Abbildung 1.2 sind die Vorwärtsauswertungen für ein einige ausgewählte Operationen beschrieben.

1.3. Der Rückwärtsmodus des Automatischen Differenzierens

Sei $f : X \subset \mathbb{R}^n \to \mathbb{R}^m$ dieselbe Funktion, wie wir sie im Vorwärtsmodus betrachtet haben, das heißt f ist durch $f(x) = P_y \circ \Phi_k \circ ... \circ \Phi_1 \circ I_x(x)$ beschrieben. Für den Vorwärtsmodus haben wir die Gleichung $\dot{y} = \frac{df}{dx}(x)\dot{x}$ bekommen. Um den Rückwärtsmodus herzuleiten, benötigen wir einen Vektor $\bar{y} \in R^m$ und die Standard-Skalarprodukte $\langle \cdot, \cdot \rangle_y$ und $\langle \cdot, \cdot \rangle_x$ im $\mathbb{R}^m$ bzw. $\mathbb{R}^n$. Wir betrachten jetzt die Gleichung

$$\langle \bar{y}, \dot{y} \rangle_y = \langle \bar{y}, \frac{df}{dx}\dot{x} \rangle_y = \langle \frac{df}{dx}^T \bar{y}, \dot{x} \rangle_x \qquad \text{für alle } \bar{y} \in \mathbb{R}^m$$

Durch das Herüberschieben von $\frac{df}{dx}$ auf die linke Seite erhalten wir das Skalarprodukt in R^n. Die Richtung $\dot{x}$ ist in einem Programmdurchlauf fixiert, man kann die

Richtung aber ohne eine Beschränkung wählen. Daraus folgt, dass die Gleichung

$$\langle \frac{df}{dx}^T \bar{y}, \dot{x} \rangle_x = \langle \bar{x}, \dot{x} \rangle_x$$

für alle $\dot{x} \in \mathbb{R}^n$ gelten muss. Wir können also $\bar{x} \in R^n$ durch

$$\bar{x} := \frac{df}{dx}^T \bar{y} \tag{1.5}$$

definieren. Was sagt $\bar{x}$ aus?

Wenn wir zum Beispiel $m = 1$ annehmen, erhalten wir $\bar{x} = \nabla f \bar{y}$. Für $\bar{y} = 1$ erhalten wir den Gradient von f. Im Kapitel zum Vorwärtsmodus brauchten wir im gleichen Beispiel n Durchläufe, um den gesamten Gradienten zu berechnen.

Was bedeutet (1.5) für die Jacobi-Matrix von f und im Besonderen für die Elementarabbildungen Φ_i? Wenn wir die Stelle der Auswertung weglassen, erhalten wir für die Jacobi-Matrix von f

$$\frac{df}{dx} = \frac{dP_y}{dv} \cdot \frac{\Phi_k}{dv} \cdot \ldots \cdot \frac{\Phi_1}{dv} \cdot \frac{dI_x}{dx} \ .$$

Daraus ergibt sich für die Transponierte

$$\frac{df}{dx}^T = \frac{dI_x}{dx}^T \cdot \frac{\Phi_1}{dv}^T \cdot \ldots \cdot \frac{\Phi_k}{dv}^T \cdot \frac{dP_y}{dv}^T \ .$$

Die Jacobi-Verknüpfungen werden also rückwärts durchlaufen. Daher bezeichnet man diese Methode der Auswertung auch als Rückwärtsmodus. Doch eine Jacobi-Matrix zu transponieren, ist erstmal nichts Neues. Wir müssen uns noch anschauen, was die Transponierte für die Elementarabbildungen Φ_i, die Einbettung I_x und die Projektion P_y bedeuten.

Für $I_x(x) = \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}$ erhalten wir:

$$\frac{dI_x}{dx} = \begin{pmatrix} I \\ 0 \\ 0 \end{pmatrix} \qquad \text{daraus folgt} \qquad \frac{dI_x}{dx}^T = \begin{pmatrix} I & 0 & 0 \end{pmatrix} \ .$$

Für $P_y(x, u, y) = y$ erhalten wir:

$$\frac{dP_y}{dv} = \begin{pmatrix} 0 & 0 & I \end{pmatrix} \qquad \text{daraus folgt} \qquad \frac{dP_y}{dv}^T = \begin{pmatrix} 0 \\ 0 \\ I \end{pmatrix} \ .$$

Durch das Transponieren vertauschen die Jacobimatrizen der Einbettung und der Projektion ihre Rollen.

Für die Elementarabbildung Φ_i und die dazugehörige Elementaroperation ϕ_i erhalten wir wie im Abschnitt zum Vorwärtsmodus:

$$\frac{d\Phi_i}{dv} = \begin{pmatrix} I & 0 & 0 \\ \frac{d\phi_i}{dv} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Durch Transponieren ergibt sich

$$\frac{d\Phi_i}{dv}^T = \begin{pmatrix} I & \frac{d\phi_i}{dv}^T & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} .$$

Wie man sieht, steht in der ersten Zeile der Matrix die Identität und die Jacobi-Matrix der Elementarabbildung. Für die j-te Komponente mit $j < i+n$ des Produktes $\bar{w} = \frac{d\Phi_i^T}{dv}\bar{v}$ ergibt sich

$$\bar{w}_j = \bar{v}_j + \frac{\partial \phi_i}{\partial v_j}\bar{v}_i \ .$$

Die Auswertung von $\frac{df}{dx}^T$ kann man wieder in einer Rekursion schreiben, wobei wir zuerst die Funktion vorwärts auswerten und danach die Jacobi-Matrizen rückwärts.

$$\begin{aligned}
v_{(0)} &= I_x(x) \\
v_{(1)} &= \Phi_1(v_{(0)}) \\
v_{(2)} &= \Phi_2(v_{(1)}) \\
&\vdots \\
v_{(i)} &= \Phi_i(v_{(i-1)}) \\
&\vdots \\
v_{(k)} &= \Phi_k(v_{(k-1)}) \\
y &= P_y(v_{(k)})
\end{aligned}$$

$$\begin{aligned}
\bar{v}_{(k)} &= \frac{dP_y}{dv}^T \bar{y} \\
\bar{v}_{(k-1)} &= \frac{d\Phi_k}{dv}^T (v_{(k-1)})\bar{v}_{(k)} \qquad (1.6) \\
&\vdots \\
\bar{v}_{(i-1)} &= \frac{d\Phi_i}{dv}^T (v_{(i-1)})\bar{v}_{(i)} \\
&\vdots \\
\bar{v}_{(1)} &= \frac{d\Phi_2}{dv}^T (v_{(1)})\bar{v}_{(2)} \\
\bar{v}_{(0)} &= \frac{d\Phi_1}{dv}^T (v_{(0)})\bar{v}_{(1)} \\
\bar{x} &= \frac{dI_x}{dx}^T \bar{v}_{(0)}
\end{aligned}$$

Da in einem Programm diese Auswertung so nicht vorgenommen wird, schauen wir uns die vereinfachte Auswertung an. In dem Code wird nicht jedes mal ein neuer Vektor $v_{(k)}$ gebildet, es wird mit einem Vektor v gearbeitet, in dem nur durch die Elementarabbildungen ϕ_i die einzelnen Komponenten gesetzt werden. Aus der Gleichung

$$\bar{w_j} = \bar{v_j} + \frac{\partial \phi_i}{\partial v_j}\bar{v_i}$$

wird

$$\bar{v_j}' = \bar{v_j} + \frac{\partial \phi_i}{\partial v_j}\bar{v_i} \ .$$

Dieses Update für v_j beschreiben wir mit dem Operator $+=$. Damit ergibt sich die Gleichung

$$\bar{v_j} \ += \frac{\partial \phi_i}{\partial v_j}\bar{v_i} \ .$$

In Abbildung 1.3 wurde aus der Iteration (1.6) der AD Rückwärtsmodus hergeleitet.

Der Vorteil des Rückwärtsmodus gegenüber dem Vorwärtsmodus besteht darin, dass wir bei einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$, die von vielen Eingabewerten abhängt und nur von wenigen Ausgabewerten $m << n$, für die Bestimmung der Jacobi-Matrix nur m-mal den Rückwärtslauf durchführen und nicht n-mal den Vorwärtslauf . Das Problem des Rückwärtsmodus liegt allerdings im Speicheraufwand. So wie in Abbildung 1.3 beschrieben, müssen wir alle Zwischenwerte v_i speichern, um die partiellen Ableitungen von ϕ_j auswerten zu können. Die gefundenen Aussagen fassen wir nun noch zusammen.

Definition 1.15 (AD Rückwärtsform eines Differentials)
Es sei $\phi : X \subset \mathbb{R}^n \to \mathbb{R}^m$ eine differenzierbare Funktion, $x \in X$ und $\bar{y} \in \mathbb{R}^m$. Dann haben wir durch $\dot{y} = \frac{d\phi}{\dot{x}}\dot{x}$ die Vorwärtsform der Abbildung gegeben. Die Rückwärtsform erhalten wir durch Transponieren der Jacobi-Matrix

$$\bar{x} \ += \frac{d\phi}{dx}^T \bar{y},$$

wobei $\bar{x}$ von x abhängt.

Satz 1.16 (AD Rückwärtsmodus)
Es sei ein Programm gegeben durch $k \in \mathbb{N}$ Elementaroperation $\phi_i : V \to V$ mit $i = 1...k$. Es seien $x \in X$ die Eingabewerte und $\bar{y} \in \mathbb{R}^m$ eine Richtung. Dann erhalten wir durch den in Abbildung 1.3 definierten Ablauf die transponierte Jacobi-Matrix multipliziert mit der Richtung $\bar{y}$.

Falls das Programm für eine Funktion $f : X \subset \mathbb{R}^n \to \mathbb{R}^m$ steht, erhalten wir

$$\bar{x} = \frac{df}{dx}^T (x)\bar{y} \ .$$

In Abbildung 1.4 sind die Rückwärtsauswertung für einige ausgewählte Operationen beschrieben.

	Funktion	Normale Anweisung	AD Anweisung
Input:	$I_x(x)$	$v_i = x_i \quad \forall i = 1...n$	
	$\Phi_1(v)$	$v_{n+1} = \phi_1(v)$	
	$\Phi_2(v)$	$v_{n+2} = \phi_2(v)$	
	$\vdots$		
	$\Phi_i(v)$	$v_{n+i} = \phi_i(v)$	
	$\vdots$		
	$\Phi_k(v)$	$v_{n+k} = \phi_k(v)$	
Output:	$P_y(v)$	$y_i = v_{l-m+i} \quad \forall i = 1...m$	
Input:	$P_y(v)$		$\bar{v}_{l-m+i} = \bar{y}_i \quad \forall i = 1...m$
	$\Phi_k(v)$		$\bar{v}_j \mathrel{+}= \frac{\partial \phi_k}{\partial v_j} \bar{v}_{n+k} \quad \forall j = 1...(n+l-1)$
			$\bar{v}_{n+k} = 0$
	$\vdots$		
	$\Phi_i(v)$		$\bar{v}_j \mathrel{+}= \frac{\partial \phi_i}{\partial v_j} \bar{v}_{n+i} \quad \forall j = 1...(n+i-1)$
			$\bar{v}_{n+i} = 0$
	$\vdots$		
	$\Phi_1(v)$		$\bar{v}_j \mathrel{+}= \frac{\partial \phi_1}{\partial v_j} \bar{v}_{n+1} \quad \forall j = 1...n$
			$\bar{v}_{n+1} = 0$
Output:	$I_x(x)$		$\bar{x}_i = \bar{v}_i \quad \forall i = 1...n$

Abbildung 1.3.: AD Rückwärtsmodus

Elementaroperation	Rückwärtsauswertung AD
$u = v + w$	$\bar{v} \mathrel{+}= \bar{u}, \quad \bar{w} \mathrel{+}= \bar{u}, \quad \bar{u} = 0$
$u = v \cdot w$	$\bar{v} \mathrel{+}= w \cdot \bar{u}, \quad \bar{w} \mathrel{+}= v \cdot \bar{u}, \quad \bar{u} = 0$
$u = \frac{1}{v}$	$\bar{v} \mathrel{+}= -\frac{1}{v^2} \cdot \bar{u}, \quad \bar{u} = 0$
$u = \sin(v)$	$\bar{v} \mathrel{+}= \cos(v) \cdot \bar{u}, \quad \bar{u} = 0$
$u = e^v$	$\bar{v} \mathrel{+}= e^v \cdot \bar{u}, \quad \bar{u} = 0$
$u = \ln(v)$	$\bar{v} \mathrel{+}= \frac{1}{v} \cdot \bar{u}, \quad \bar{u} = 0$
$u = v^p$	$\bar{v} \mathrel{+}= pv^{p-1} \cdot \bar{u}, \quad \bar{u} = 0$

Abbildung 1.4.: AD Rückwärtsmodus für ausgewählte Operationen

1.4. Lineare Gleichungssysteme, Iterationen und das Newtonverfahren

In diesem Abschnitt schauen wir uns fortgeschrittene Elementaroperationen an, die wir später für die Anwendung auf den Padge Code verwenden. Dafür benötigen wir noch die Produktregel für Differentialformen. Da wir in diesem Kapitel hauptsächlich mit Matrizen arbeiten, müssen wir die Produktregel auch in ihrer vollen Allgemeinheit herleiten.

Satz 1.17 (Produktregel für Differentialformen)
Es sei $A : X \subset \mathbb{R}^n \to \mathbb{R}^{a\times l}$ und $B : X \subset \mathbb{R}^n \to \mathbb{R}^{l\times b}$. Dann ist die Abbildung $f(x) := A(x) \cdot B(x)$ differenzierbar und das Differential lautet

$$df(x) = dA(x) \cdot B(x) + A(x) \cdot dB(x) = \frac{dA}{dx}(x)dx \cdot B(x) + A(x) \cdot \frac{dB}{dx}(x)dx \ .$$

Beweis. Wir betrachten

$$f(x)_{i,j} = \sum_{k=1}^{l} A(x)_{i,k} \cdot B(x)_{k,j} \ .$$

Hierfür können wir das Differential bilden

$$\begin{aligned}
df(x)_{i,j} &= \sum_{p=1}^{n}\sum_{k=1}^{l}\left(\frac{dA(x)_{i,k}}{dx_p} \cdot B(x)_{k,j}dx_p + A(x)_{i,k} \cdot \frac{dB(x)_{k,j}}{dx_p}dx_p\right) \\
&= \sum_{p=1}^{n}\sum_{k=1}^{l}\left(\frac{dA(x)_{i,k}}{dx_p}dx_p \cdot B(x)_{k,j} + A(x)_{i,k} \cdot \frac{dB(x)_{k,j}}{dx_p}dx_p\right) \\
&= \sum_{k=1}^{l}\left(\sum_{p=1}^{n}\frac{dA(x)_{i,k}}{dx_p}dx_p \cdot B(x)_{k,j} + A(x)_{i,k} \cdot \sum_{p=1}^{n}\frac{dB(x)_{k,j}}{dx_p}dx_p\right) \\
&= \sum_{k=1}^{l}\left(dA(x)_{i,k} \cdot B(x)_{k,j} + A(x)_{i,k} \cdot dB(x)_{k,j}\right) \ .
\end{aligned}$$

Die lineare Abbildung dx_p können wir mit dem $B(x)_{k,j}$ vertauschen, da wir uns im skalarem Fall befinden. Wenn wir jeweils die letzte und vorletzte Gleichung betrachten, können wir diese wieder zu einer Matrix zusammensetzen. Damit erhalten wir die Gleichung für die Produktregel. □

Mit dem Satz 1.17 können wir nun eine Aussage über das Differential der inversen Abbildung herleiten.

Satz 1.18 (Differential der Inversen Abbildung)
Sei $A : \mathbb{R}^m \to \mathbb{R}^{n\times n}$ stetig differenzierbar und es existiere $A(x)^{-1}$ für alle $x \in X$ mit X offen. Dann existiert

$$\frac{dA^{-1}}{dx}(x)$$

und das Differential von A^{-1} hat die Form

$$dA^{-1} = -A^{-1} \cdot dA \cdot A^{-1}$$
$$dA^{-1}(x) = -A^{-1}(x) \cdot \frac{dA}{dx}(x) \cdot A^{-1}(x) \ .$$

Beweis. Für den Beweis benutzen wir den Satz über die implizite Funktionen. Dazu definieren wir für ein fixiertes $b \in \mathbb{R}^n$

$$F(x, a) := A(x)a - b \qquad \text{mit } a \in \mathbb{R}^n.$$

Für diese Funktion existiert für jedes $x \in X$ eine Nullstelle, da nach Voraussetzung $A(x)^{-1}$ existiert und $F(x, A(x)^{-1}b) = 0$ ist. Wenn wir uns die Ableitung von F nach a anschauen, erhalten wir

$$\frac{dF}{da}(x, a) = A(x)$$

und sehen, dass $\frac{dF}{da}$ für alle $x \in \mathbb{R}^m$ und $a \in \mathbb{R}^n$ invertierbar ist. Der Satz über implizite Funktionen besagt nun, dass wir für alle $(x, a) \in X \times \mathbb{R}^n$, Umgebungen $U_1(x) \subset X$ und $U_2(a) \subset \mathbb{R}^n$ finden sowie eine Abbildung $g : U_1 \to U_2$ mit der Eigenschaft, dass

$$F(x, g(x)) = 0 \tag{*}$$

ist. Wenn wir für diese Gleichung das Differential bilden, erhalten wir durch die Kettenregel

$$\begin{aligned} 0 = dF(x, g(x)) &= \frac{dF}{dx}(x, g(x))dx + \frac{dF}{da}(x, g(x)) \cdot \frac{dg}{dx}(x)dx \\ &= \frac{dA}{dx}(x)dx \cdot g(x) + A(x)\frac{dg}{dx}(x)dx \ . \end{aligned}$$

Diese Gleichung können wir nun nach $\frac{dg}{dx}dx$ umformen

$$\frac{dg}{dx}(x)dx = -A^{-1}(x)\frac{dA}{dx}(x)dx \cdot g(x) \tag{**}$$

Wenn wir uns nun noch einmal (*) anschauen, erhalten wir

$$\begin{aligned} 0 = F(x, g(x)) &= A(x)g(x) - b \\ \Leftrightarrow b &= A(x)g(x) \\ \Leftrightarrow g(x) &= A(x)^{-1}b \end{aligned}$$

und für das Differential von $g(x)$

$$dg(x) = \frac{dg}{dx}(x)dx = \frac{dA^{-1}}{dx}(x)dx \cdot b = dA^{-1}(x)b \ .$$

Wir können nun die letzten beiden Gleichungen in (**) einsetzen und erhalten

$$dA^{-1}(x)b = \frac{dg}{dx}(x)dx = -A^{-1}(x)\frac{dA}{dx}(x)dx \ A^{-1}(x)b \ .$$

Elementaroperation	Vorwärtsauswertung AD
$u = W^{-1}v$	$\dot{u} = W^{-1}(\dot{v} - \dot{W}u)$

Abbildung 1.5.: AD Vorwärtsmodus für das Lösen eines linearen Gleichungssystems

Da $b \in \mathbb{R}^n$ beliebig gewählt werden kann, erhalten wir das Ergebnis

$$dA^{-1} = -A^{-1}\frac{dA}{dx}dxA^{-1} \ .$$

□

Wir kennen nun das Differential einer Inversen Matrix, damit können wir jetzt eine Aussage über das Differential eines linearen Gleichungssystems machen.

Satz 1.19 (Differential eines linearen Gleichungssystems)
Gegeben sei das Gleichungssystem $W(x) \cdot u(x) = v(x)$ mit $W : X \subset \mathbb{R}^n \to \mathbb{R}^{m\times m}$ und $u, v : X \subset \mathbb{R}^n \to \mathbb{R}^m$. W, u und v seien stetig differenzierbar und W^{-1} existiere für alle $x \in \mathbb{R}^n$.

Wir definieren $u(x) := W^{-1}(x)v(x)$, dann ist das Differential von u gegeben durch

$$du = W^{-1}(dv - dWu) \ .$$

Beweis. Für u erhalten wir durch die Produktregel und das Differential der Inversen Abbildung

$$\begin{aligned} du &= dW^{-1}v + W^{-1}dv \\ &= -W^{-1}dWW^{-1}v + W^{-1}dv \\ &= W^{-1}\left(-dW\underbrace{W^{-1}v}_{=u} + dv\right) \\ &= W^{-1}\left(dv - dWu\right) \ . \end{aligned}$$

□

Wenn wir $\phi(x) := u(x) = W^{-1}(x)v(x)$ als Elementaroperation betrachten, können wir für den AD Vorwärtsmodus die Anweisung für ϕ wie in Abbildung 1.5 schreiben.

Für den Rückwärtsmodus müssen wir diese Gleichung noch umschreiben. Sei $T \equiv W^{-1}$, A_{*i} die i-te Spalte einer Matrix A und A_{i*} die i-te Zeile einer Matrix A. Damit

Elementaroperation	Rückwärtsauswertung AD
$u = W^{-1}v$	$s = W^{-1^T}\bar{u}$ $\bar{W}$ += $-u \cdot s^T$ $\bar{v}$ += s $\bar{u} = 0$

Abbildung 1.6.: AD Rückwärtsmodus für das Lösen eines linearen Gleichungssystems

bekommen wir für die i-te Komponente von $p := T\dot{W}u$

$$\begin{aligned} p_i &= \sum_{j,l=1}^{m} T_{ij}\dot{W}_{jl}u_l = \sum_{j,l=1}^{m} u_l T_{ij}\dot{W}_{jl} \\ &= u_1 \sum_{j=1}^{m} T_{ij}\dot{W}_{j1} + ... + u_m \sum_{j=1}^{m} T_{ij}\dot{W}_{jm} \\ &= u_1 T_{i*}\dot{W}_{*1} + ... + u_m T_{i*}\dot{W}_{*m} \ . \end{aligned}$$

Daraus folgt für den Vektor p

$$p = u_1 T\dot{W}_{*1} + ... + u_m T\dot{W}_{*m} \ .$$

Für $\dot{u}$ ergibt sich

$$\dot{u} = T\dot{v} - u_1 T\dot{W}_{*1} - ... - u_m T\dot{W}_{*m} \ .$$

und dadurch nach der Vorschrift für den Rückwärtsmodus

$$\begin{aligned} \bar{v} \ &+= T^T\bar{u} \\ \bar{W}_{*1} \ &+= -\ u_1 T^T\bar{u} \\ &\vdots \\ \bar{W}_{*m} \ &+= -\ u_m T^T\bar{u} \ . \end{aligned}$$

Das Produkt $T^T\bar{u}$ kommt in jeder Gleichung vor, daher brauchen wir es in einer Berechnung nur einmal ausrechnen. Das Aufsummieren der einzelnen Spalten von $\bar{W}$ können wir auch kürzer fassen, indem man $\bar{W}$ += $-u(T^T\bar{u})^T$ berechnet.

Damit erhalten wir für die Elememtaroperation $\phi(x) := u(x) = W^{-1}(x)v(x)$ die Anweisung für den Rückwärtsmodus, wie es in Abbildung 1.6 dargestellt ist.

Wir schauen uns nun einen beliebigen iterativen Löser an. Sei $G : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ ein iterativer Löser. Wir haben also die Rekursion

$$z_{k+1} = G(z_k, x) \quad \text{mit } z_0 \text{ geg.} \ .$$

Das Differential von G hat die Form

$$dz_{k+1} = dG(z_k, x) = \frac{\partial G}{\partial z}(z_k, x)dz_k + \frac{\partial G}{\partial x}(z_k, x)dx \ .$$

Dadurch ergibt sich für den Vorwärtsmodus

$$\dot{z}_{k+1} = \frac{\partial G}{\partial z}\dot{z}_k + \frac{\partial G}{\partial x}\dot{x} \ .$$

und für den Rückwärtsmodus

$$\begin{aligned} \bar{z}_k \ &+= \frac{\partial G}{\partial z}\bar{z}_{k+1} \\ \bar{x} \ &+= \frac{\partial G}{\partial x}\bar{z}_{k+1} \ . \end{aligned}$$

Für den Vorwärtsmodus sehen wir, dass man ohne weiteres die Anweisung für die Ableitung aufschreiben kann. Die Iterierte $\dot{z}_{k+1}$ hängt nur von x, dx und den Iterierten aus dem Schritt k ab.

Für den Rückwärtsmodus scheint dasselbe zu gelten, jedoch tritt das Speicherproblem hier im besonderem Maße auf. Wir müssen die Vektoren z_k speichern, um sie beim Rückwärtslauf für die Auswertung der Jacobi-Matrizen zur Hand zu haben. Wenn wir ein sehr langsam konvergierendes Verfahren haben, können das mehrere hundert Vektoren sein. Außerdem berechnen wir die Abhängigkeit von unserem Ausgabewert y nach einem frei wählbaren Parameter z_0. Wir interessieren uns aber nur für die Abhängigkeit der Ausgabewerte y nach den Eingabewerten x. Daher macht die Rückwärtsauswertung im Vergleich zu dem Aufwand wenig Sinn. Es ergibt sich auch das Problem, dass man das iterative Verfahren nur bis zu einer bestimmten Genauigkeit ausführt. Wenn man nun nach dem Abbruch den Rückwärtslauf ausführt, kann man keine Aussage darüber treffen, wie genau die Ableitung berechnet worden ist. Man sollte daher für ein iteratives Verfahren lieber den Vorwärtsmodus wählen.

Mit einer anderen Herleitung des Rückwärtsmodus für iterative Verfahren kann man eine bessere Berechnung durchführen. Der sog. „incremental reverse“ wird in Griewank et al. [4] beschrieben.

Da wir nun wissen, wie ein iteratives Verfahren differenziert wird, können wir uns das Newtonverfahren anschauen. Für das Newtonverfahren haben wir eine Funktion $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$, für die wir eine Nullstelle finden wollen. Die Jacobimarix f' von f bezeichnen wir mit F. F ist also eine Funktion $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{n\times n}$ mit $F = f'$.

Für F setzten wir voraus, dass die Inverse F^{-1} existiert. Dann ist das Newtonverfahren durch die Gleichung

$$z_{k+1} = G(z_k, x) = z_k - F(z_k, x)^{-1} f(z_k, x)$$

definiert. Die partiellen Ableitungen von G sind

Elementaroperation	Vorwärtsauswertung AD
$y = f(z_k, x)$	$\dot{y} = \dot{f} = \frac{\partial f}{\partial z}\dot{z}_k + \frac{\partial f}{\partial x}\dot{x}$
$Y = F(z_k, x)$	$\dot{Y} = \dot{F} = \frac{\partial F}{\partial z}\dot{z}_k + \frac{\partial F}{\partial x}\dot{x}$
$d = Y^{-1}y$	$\dot{d} = Y^{-1}\left(\dot{y} - \dot{Y}d\right)$
$z_{k+1} = z_k - d$	$\dot{z}_{k+1} = \dot{z}_k - \dot{d}$

Abbildung 1.7.: Anweisungen für die Ableitung des Newton Schritts als Elementaroperation

$$\begin{aligned}
\frac{\partial G}{\partial z} &= I - \frac{\partial (F^{-1})}{\partial z} f - F^{-1}\frac{\partial f}{\partial z} = I + F^{-1}\frac{\partial F}{\partial z}F^{-1}f - F^{-1}\frac{\partial f}{\partial z} \\
&= I + F^{-1}\left(\frac{\partial F}{\partial z}F^{-1}f - \frac{\partial f}{\partial z}\right) \\
\frac{\partial G}{\partial x} &= -\frac{\partial (F^{-}1)}{\partial x} f - F^{-1}\frac{\partial f}{\partial x} = F^{-1}\frac{\partial F}{\partial x}F^{-1}f - F^{-1}\frac{\partial f}{\partial x} \\
&= F^{-1}\left(\frac{\partial F}{\partial x}F^{-1}f - \frac{\partial f}{\partial x}\right) \ .
\end{aligned}$$

Dadurch sieht das Differential von G folgendermaßen aus

$$\begin{aligned}
dz_{k+1} = & dz_k + F^{-1}\left(\frac{\partial F}{\partial z}dz_k F^{-1}f - \frac{\partial f}{\partial z}dz_k\right) \\
& + F^{-1}\left(\frac{\partial F}{\partial x}dx F^{-1}f - \frac{\partial f}{\partial x}dx\right) \\
= & dz_k + F^{-1}\left[\underbrace{\left(\frac{\partial F}{\partial z}dz_k + \frac{\partial F}{\partial x}dx\right)}_{=dF} F^{-1}f(z,x) - \underbrace{\left(\frac{\partial f}{\partial z}dz_k + \frac{\partial f}{\partial x}dx\right)}_{=df}\right] \ .
\end{aligned}$$

In dem Differential von G können wir die Differentiale von F und f verwenden. Der Ausdruck $F^{-1}f(z,x)$ kommt schon in der Auswertung von G vor. Wenn wir diesen Ausdruck ersetzen, erhalten wir die Anweisung für den Vorwärtsmodus von AD mit einem Newton Schritt als Elementaroperation. In Abbildung 1.7 werden die Anweisungen aufgeführt.

Bei dieser Aufstellung muss man beachten, dass die ersten beiden Zeilen gleichzeitig ausgeführt werden. Da man in dem Programm die differenzierten Versionen von f und F aufruft, werden y und $\dot{y}$ sowie Y und $\dot{Y}$ zur selben Zeit durch AD berechnet.

1.5. Der Vektormodus

Bisher haben wir nur darüber gesprochen, dass wir für eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ im Vorwärtsmodus

$$\dot{y} = \frac{df}{dx}\dot{x}$$

und im Rückwärtsmodus

$$\bar{x} = \frac{df}{dx}^T \bar{y}$$

auswerten. Wenn wir die Jacobi-Matrix von f haben, könnten wir auch anstatt des Vektors $\dot{x} \in \mathbb{R}^n$ eine Matrix $\dot{X} \in \mathbb{R}^{n\times l}$ wählen. Wir erhalten dann $\dot{Y} \in \mathbb{R}^{m\times l}$ und die Gleichung

$$\dot{Y} = \frac{df}{dx}\dot{X}.$$

Um diese Gleichung mit dem Kalkül der Differentialformen zu erklären, müssen wir die vektorwertige Differentialform definieren.

Definition 1.20 (Vektorwertige-Differentialform)
Unter einer vektorwertigen Differentialform versteht man eine Abbildung

$$\omega : U \subset \mathbb{R}^n \to L(\mathbb{R}^{n\times l}, \mathbb{R}^{m\times l}),$$

die jedem $x \in U$ eine lineare Abbildung $\omega(x) : \mathbb{R}^{n\times l} \to \mathbb{R}^{m\times l}$ zuordnet.

Das Differential der Koordinatenfunktionen $\xi_i : \mathbb{R}^n \to \mathbb{R}, \xi_i(x) = x_i$ für i von 1 bis n hat nun die Form

$$dx_i(x)(H) := d\xi_i(x)(H) = H_i.$$

Mit $x \in R^n$, $H \in \mathbb{R}^{n\times l}$ und H_i die i-te Zeile von H. Für eine Richtung $\dot{X}$ erhalten wir wieder die Identität:

$$dx(\dot{X}) = \dot{X} \ .$$

Wir können den AD Vektor-Vorwärtsmodus also genauso definieren, wie wir den AD Vorwärtsmodus definiert haben. Wir brauchen nur die Vektoren $\dot{x}$ durch die Matrizen $\dot{X}$ zu ersetzten und einzelne Komponenten $\dot{x}_i$ durch Spalten $\dot{X}_i$. Dadurch, dass wir den Rückwärtsmodus aus dem Vorwärtsmodus hergeleitet haben und dabei nur Argumente verwendet haben, die sich auf die Linearität der Skalarprodukte beziehen, können wir genauso den AD Vektor-Rückwärtsmodus aus dem dem Vektor-Vorwärtsmodus herleiten. Dadurch ergibt sich

$$\bar{X} = \frac{df}{dx}^T \bar{Y}$$

und in der Definition für den Rückwärtsmodus müssen wir genauso wie im Vorwärtsmodus Vektoren $\bar{x}$ durch Matrizen $\bar{X}$ ersetzten und einzelne Komponenten $\bar{x}_i$ durch Spalten $\bar{X}_i$.

1.6. Höhere Ableitungen

Bisher haben wir nur Ableitungen erster Ordnung gebildet. Zu einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ haben wir eine Methode entwickelt, wie wir die Richtungsableitung von f, auf Basis der Elementaroperationen ϕ, berechnen. Dadurch haben wir die Gleichungen

$$\dot{y} = \frac{df}{dx}\dot{x}, \qquad \bar{x} = \frac{df}{dx}^T \bar{y}$$

erhalten. In unserem Programm haben wir nun rein formell eine Methode für den Vorwärtsmodus

$$(y, \dot{y}) = F_V(x, \dot{x}) = \left(f(x), \frac{df}{dx}\dot{x}\right)$$

und für den Rückwärtsmodus

$$(y, \bar{x}) = F_R(x, \bar{y}) = \left(f(x), \frac{df}{dx}^T \bar{y}\right)$$

Für beide Funktionen können wir das Differential bilden

$$dF_V(x, \dot{x}) = \left(\frac{df}{dx}dx, \frac{d^2f}{d^2x}dx \cdot \dot{x} + \frac{df}{dx}d\dot{x}\right)$$
$$dF_R(x, \bar{y}) = \left(\frac{df}{dx}dx, \left[\frac{d^2f}{d^2x}dx\right]^T \bar{y} + \frac{df}{dx}^T d\bar{y}\right) .$$

Wir wählen nun $\hat{*}$ als Richtung für den Vorwärtsmodus, da wir sonst nicht zwischen $\dot{x}$ und $\hat{x}$ unterscheiden können. Die Funktionen für den Vorwärtsmodus haben nun die Form

$$(y, \dot{y}, \hat{y}, \hat{\dot{y}}) = F_{VV}(x, \dot{x}, \hat{x}, \hat{\dot{x}}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}\hat{x}, \frac{d^2f}{d^2x}\hat{x} \cdot \dot{x} + \frac{df}{dx}\hat{\dot{x}}\right)$$
$$(y, \bar{x}, \hat{y}, \hat{\bar{x}}) = F_{RV}(x, \bar{y}, \hat{x}, \hat{\bar{y}}) = \left(f(x), \frac{df}{dx}^T \bar{y}, \frac{df}{dx}\hat{x}, \left[\frac{d^2f}{d^2x}\hat{x}\right]^T \bar{y} + \frac{df}{dx}^T \hat{\bar{y}}\right) .$$

Um den Rückwärtsmodus einfacher nach der Vorschrift bilden zu können, schreiben wir die Formeln noch um in

$$(y, \dot{y}, \hat{y}, \hat{\dot{y}}) = F_{VV}(x, \dot{x}, \hat{x}, \hat{\dot{x}}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}\hat{x}, \frac{d^2f}{d^2x}\dot{x} \cdot \hat{x} + \frac{df}{dx}\hat{\dot{x}}\right)$$
$$(y, \bar{x}, \hat{y}, \hat{\bar{x}}) = F_{RV}(x, \bar{y}, \hat{x}, \hat{\bar{y}}) = \left(f(x), \frac{df}{dx}^T \bar{y}, \frac{df}{dx}\hat{x}, \bar{y}^T\frac{d^2f}{d^2x}\hat{x} + \frac{df}{dx}^T \hat{\bar{y}}\right) .$$

Für den Rückwärtsmodus verwenden wir das Symbol $\tilde{*}$. Damit erhalten wir

$$(y, \dot{y}, \tilde{x}, \tilde{\dot{x}}) = F_{VR}(x, \dot{x}, \tilde{y}, \tilde{\dot{y}}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}^T \tilde{y} + \left[\frac{d^2f}{d^2x}\dot{x}\right]^T \tilde{\dot{y}}, \frac{df}{dx}^T \tilde{\dot{y}}\right)$$
$$(y, \bar{x}, \tilde{x}, \tilde{\bar{y}}) = F_{RR}(x, \bar{y}, \tilde{y}, \tilde{\bar{x}}) = \left(f(x), \frac{df}{dx}^T \bar{y}, \frac{df}{dx}^T \tilde{y} + \left[\bar{y}^T\frac{d^2f}{d^2x}\right]^T \tilde{\bar{x}}, \frac{df}{dx}\tilde{\bar{x}}\right) .$$

Wenn wir in F_{VR}, F_{RV} und F_{RR} die Werte umstellen erhalten wir

$$(y, \hat{y}, \bar{x}, \hat{\bar{x}}) = F_{RV}(x, \hat{x}, \bar{y}, \hat{\bar{y}}) = \left(f(x), \frac{df}{dx}\hat{x}, \frac{df}{dx}^T \bar{y}, \left[\frac{d^2 f}{d^2 x}\hat{x}\right]^T \bar{y} + \frac{df}{dx}^T \hat{\bar{y}} \right)$$
$$(y, \dot{y}, \tilde{\dot{x}}, \tilde{x}) = F_{VR}(x, \dot{x}, \tilde{\dot{y}}, \tilde{y}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}^T \tilde{\dot{y}}, \left[\frac{d^2 f}{d^2 x}\dot{x}\right]^T \tilde{\dot{y}} + \frac{df}{dx}^T \tilde{y} \right)$$
$$(y, \tilde{\bar{y}}, \bar{x}, \tilde{x}) = F_{RR}(x, \tilde{\bar{x}}, \bar{y}, \tilde{y}) = \left(f(x), \frac{df}{dx}\tilde{\bar{x}}, \frac{df}{dx}^T \bar{y}, \left[\bar{y}^T \frac{d^2 f}{d^2 x}\right]^T \tilde{\bar{x}} + \frac{df}{dx}^T \tilde{y} \right)$$

und sehen, dass F_{RV}, F_{VR} und F_{RR} äquivalent sind. Deshalb ist es besser, wenn man für höhere Ableitungen zuerst den Vorwärts- oder Rückwärtsmodus anwendet und danach $d-1$ mal den Vorwärtsmodus.

Die mit diesem Vorschlag erzeugten Formeln sind die Folgenden:

$$(y, \dot{y}, \hat{y}, \hat{\dot{y}}) = F_{VV}(x, \dot{x}, \hat{x}, \hat{\dot{x}}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}\hat{x}, \frac{d^2 f}{d^2 x}\hat{x} \cdot \dot{x} + \frac{df}{dx}\hat{\dot{x}} \right)$$
$$(y, \hat{y}, \bar{x}, \hat{\bar{x}}) = F_{RV}(x, \hat{x}, \bar{y}, \hat{\bar{y}}) = \left(f(x), \frac{df}{dx}\hat{x}, \frac{df}{dx}^T \bar{y}, \left[\frac{d^2 f}{d^2 x}\hat{x}\right]^T \bar{y} + \frac{df}{dx}^T \hat{\bar{y}} \right) .$$

Das Symbol $\hat{*}$ können wir für F_{VV} nicht ersetzen. In F_{RV} können wir $\hat{*}$ durch $\dot{*}$ ersetzen um die normale Symbolik zu erhalten. Daraus folgt

$$(y, \dot{y}, \bar{x}, \dot{\bar{x}}) = F_{RV}(x, \dot{x}, \bar{y}, \dot{\bar{y}}) = \left(f(x), \frac{df}{dx}\dot{x}, \frac{df}{dx}^T \bar{y}, \left[\frac{d^2 f}{d^2 x}\dot{x}\right]^T \bar{y} + \frac{df}{dx}^T \dot{\bar{y}} \right) . \quad (1.7)$$

In (1.7) sieht man nun sehr gut, dass in den ersten beiden Komponenten y und $\dot{y}$ der normale Vorwärtsmodus berechnet wird. Die Komponenten $\bar{x}$ und $\dot{\bar{x}}$ beschreiben einen Vorwärtsschritt im Rückwärtsdurchlauf.

Die Theorie für den AD Rückwärts-Vorwärtsschritt kann man nun für die verschiedenen Elementaroperationen anwenden. Wir wollen uns hier die Elementaroperation für das Lösen eines linearen Gleichungssystems anschauen.

Für die Elementaroperation $u = W^{-1}v$ haben wir den Vorwärts- und Rückwärtsmodus berechnet. Wenn wir uns für die Rückwärtsauswertung die Formeln anschauen, können wir für diese die Differentiale bilden. Die Schritte für den Rückwärtsmodus sind

$$\begin{aligned} s &= {W^{-1}}^T \bar{u} \\ \bar{W} &+= -u \cdot s^T \\ \bar{v} &+= s \\ \bar{u} &= 0 . \end{aligned}$$

Elementaroperation	Vorwärtsauswertung AD
$u = W^{-1}v$	$\dot{u} = W^{-1}(\dot{v} - \dot{W}u)$
Rückwärtsauswertung AD	Rückwärtsvorwärtsauswertung AD
$s = W^{-1^T}\bar{u}$ $\bar{W}$ += $-u \cdot s^T$ $\bar{v}$ += s $\bar{u} = 0$	$\dot{s} = W^{-1^T}(\dot{\bar{u}} - \dot{W}^T s)$ $\dot{\bar{W}}$ += $-\dot{u} \cdot s^T - u \cdot \dot{s}^T$ $\dot{\bar{v}}$ += $\dot{s}$ $\dot{\bar{u}} = 0$

Abbildung 1.8.: AD Rückwärtsvorwärtsmodus für ein lineares Gleichungssystem

Die Differentiale für die einzelnen Schritte haben die Form

$$\begin{aligned} ds &= W^{-1^T}(d\bar{u} - dW^T s) \\ d\bar{W} &+= -du \cdot s^T - u \cdot ds^T \\ d\bar{v} &+= ds \\ d\bar{u} &= 0 \ . \end{aligned}$$

Dadurch ergibt sich die Rückwärtsvorwärtsauswertung von der Elementaroperation $u = W^{-1}v$ die in der Abbildung 1.8 dargestellt ist.

2. Discontinuous Galerkin

Diskontinuierliche Galerkin Verfahren (DG) sind eine Weiterentwicklung der kontinuierlichen Galerkin Verfahren, um partielle Differentialgleichungen zu lösen. Die Ansatzfunktionen im Diskontinuierlichen sind nicht mehr kontinuierlich, sondern erlauben Sprünge an den Kanten von zwei Elementen der Diskretisierung. Durch die Sprünge würde sich normalerweise kein vernünftiges Verfahren ergeben, da die Lösung der partiellen Differentialgleichung nicht global erfüllt wäre, sondern nur lokal für jedes einzelne Element. Um ein konsistentes und stabiles Verfahren, das die Gleichung global erfüllt, zu erhalten, müssen die Sprünge durch eine Modifikation der zugrundeliegenden Gleichungen, behandelt werden.

DG Verfahren haben den Vorteil gegenüber normalen Finiten-Element-Methoden (FEM), dass man für sie einfacher Verfahren höherer Ordnung herleiten kann. Diese Einfachheit geht damit einher, dass die Kommunikation zwischen den Elementen auf die Nachbarn eines Elementes beschränkt bleibt. Traditionelle FEM oder Finite-Volumen-Methoden müssen hierzu die Nachbarn der Nachbarn eines Elementes einbeziehen. Auf unstrukturierten Gittern ist das meist nur sehr schwer möglich. Die höhere Ordnung in den DG Verfahren hingegen ergibt sich durch den Grad der Ansatzfunktionen und der Größe der Elemente. Da diese beiden Werte für ein DG-Verfahren nicht global festgelegt sein müssen, ergibt sich der weitere Vorteil, dass man für DG-Verfahren sehr einfach Verfeinerungsstrategien implementieren kann. Durch verschiedene Arten von Verfeinerungen ergibt sich die Möglichkeit das Rechengebiet an die Erfordernisse der zu lösenden Gleichung anzupassen und dabei den Aufwand klein zu halten, aber die Ordnung des Verfahrens nicht zu mindern.

Die h Verfeinerung bezieht sich auf das Verkleinern der Rechenelement. Dadurch kann man die Lösung an sehr unregelmäßigen Stellen besser darstellen. Man spricht von der p-Verfeinerung, wenn man den Grad der Ansatzfunktionen auf einem Element erhöht. Diese Verfeinerung ist besonders dann sinnvoll, wenn sich die Lösung an einer Stelle polynomial verhält, aber die Funktionswerte stark voneinander abweichen. Zusammen bezeichnet man die beiden Verfeinerungsarten als hp-Verfeinerung.

In einem aerodynamischen Kontext ist es sinnvoll, für die Elemente an dem Flugzeugrand, die p-Verfeinerung zu benutzen. Die Geschwindigkeit der Luft ist an dem Flugzeugrand durch die No-Slip Bedingung nahe bei Null. Da sich das Flugzeug aber sehr schnell bewegt, ist die Luftgeschwindigkeit schon wenige Millimeter vom Rand entfernt sehr hoch. Mit einer p-Verfeinerung der Elemente nahe dem Rand kann, man diesen steilen Anstieg in der Geschwindigkeit gut auflösen. Eine h-Verfeinerung macht bei der Auflösung von Schocks Sinn. Nach dem Satz von Godonov muss ein Verfahren an dem Schock die Ordnung 1 haben. Wir können p daher nicht frei wählen. Um eine vernünftige Konvergenz zu erhalten, müssen wir in der Umgebung des Schocks die

Elemente verkleinern, dass heißt h-verfeinern. Ein DG-Verfahren, das diese Adaption der Schocks auf Basis eines Fehlerschätzers beinhaltet, ist in der Literatur noch nicht bekannt.

In folgendem Kapitel werden wir die Grundlagen zum diskontinuierlichem Galerkin Verfahren erklären und anhand einer verallgemeinerten Problemstellung das Verfahren herleiten sowie die Ergebnisse für spezielle Problemstellungen geben.

Zum Schluss werden wir auf die Implementierung eines Verfahrens eingehen um auch eine Betrachtung für die Eignung zum Automatischem Differenzieren zu erörtern.

Wir benutzen die Arbeit von Hartmann [8] als Vorbild für die Herleitung.

2.1. Problemstellung

Für das Modelproblem, dass wir in diesem Kapitel betrachten wollen, benötigen wir zuerst noch einige Definitionen.

Definition 2.1 (Gebiet und Zerlegung)
Es sei $\Omega \subset \mathbb{R}^k$ ein Gebiet, dass heißt Ω ist offen und wegezusammenhängend. Dann ist $\Gamma := \partial\Omega$ der Rand von Omega. $\mathcal{T}_h(\Omega)$ ist eine Zerlegung von Ω wenn für alle Elemente $\kappa \in \mathcal{T}_h$ gilt, dass

1. κ ist offen, für alle $\kappa \in \mathcal{T}_h$,
2. $\kappa_1 \cap \kappa_2 = \emptyset$, für alle $\kappa_1, \kappa_2 \in \mathcal{T}_h, \kappa_1 \neq \kappa_2$,
3. $\bigcup_{\kappa \in \mathcal{T}_h} \bar{\kappa} = \bar{\Omega}$,
4. Für alle $\kappa \in \mathcal{T}_h$ existiert eine stetig differenzierbare Abbildung $\phi : \hat{\kappa} \to \kappa$ mit $\hat{\kappa}$ ist ein k-Einheitssimplex, k-Einheitsquader oder ein anderes Einheitselement im $\mathbb{R}^k$.

Als nächstes benötigen wir noch den Raum, der unsere Testfunktionen beschreibt.

Definition 2.2 (Gebrochene Sobolev-Räume)
Es sei Ω ein Gebiet und $\mathcal{T}_h(\Omega)$ eine Zerlegung von Ω gegeben. Mit $H^m(\kappa)$ bezeichnen wir den Sobolev-Raum auf dem Gebiet $\kappa \in \mathcal{T}_h(\Omega)$ nach Alt [1].
Dann heißt

$$H^m(\mathcal{T}_h) = \left\{ v \in L^2(\Omega) : v|_\kappa \in H^m(\kappa), \kappa \in \mathcal{T}_h \right\}$$

gebrochener Sobolev Raum.

Definition 2.3 (Spuroperator)
Es sei Ω ein Gebiet und $\mathcal{T}_h(\Omega)$ eine Zerlegung von Ω gegeben. Dazu sei $H^m(\mathcal{T}_h)$ der gebrochene Sobolev-Raum. Für ein $v \in H^1(\mathcal{T}_h)$ ist der lokale Spuroperator

$T|_\kappa : H^1(\kappa) \to L^2(\partial\kappa)$ gegeben. Mit dem Raum $\partial L^2(\mathcal{T}_h) := \prod_{\kappa\in\mathcal{T}_h} L^2(\partial\kappa)$ können wir den globalen Spuroperator:

$$\begin{aligned} T : H^1(\mathcal{T}_h) &\to \partial L^2(\mathcal{T}_h) \\ v &\mapsto \prod_{\kappa\in\mathcal{T}_h} T|_\kappa (v|_\kappa) \end{aligned}$$

definieren.

Für eine Kante $d \subset \partial\kappa^+$ mit $\kappa^+ \in \mathcal{T}_h$ erhalten wir immer zwei Werte für T, wenn d nicht am Rand von Ω liegt. Sei κ^- das Element in $\mathcal{T}_h$, das d als Kante hat und für das gilt $\kappa^+ \cap \kappa^- = \emptyset$. Dann bezeichnen wir mit v^+ die Spur von $v \in H^1(\mathcal{T}_h)$ auf κ^+ und mit v^- die Spur auf κ^-.

Mit der Spur können wir nun folgendes einführen:

Definition 2.4 (Sprung- und Mittelwertoperator)
Es sei Ω ein Gebiet und $\mathcal{T}_h(\Omega)$ eine Zerlegung von Ω. Dazu sei $H^m(\mathcal{T}_h)$ der gebrochene Sobolev-Raum und Γ_I die inneren Kanten der Zerlegung $\mathcal{T}_h(\Omega)$. Sei d eine Kante aus Γ_I und $\kappa^+, \kappa^- \in \mathcal{T}_h$ die dazugehörigen Elemente auf beiden Seiten. Mit $n^+ \in \mathbb{R}^k$ und $n^- \in \mathbb{R}^k$ bezeichnen wir jeweils die äußeren Normalenvektoren. Es gilt $-n^+ = n^-$. Dann definieren wir den Sprungoperator $[\![\cdot]\!]$ und den Mittelwertoperator $\{\!\{\cdot\}\!\}$ für die Funktionen $\phi \in \partial L^2(\mathcal{T}_h)$ und $\Phi \in \left[\partial L^2(\mathcal{T}_h)\right]^k$

$$\begin{aligned} \{\!\{\phi\}\!\} &= \tfrac{1}{2}\left(\phi^+ + \phi^-\right), \quad [\![\phi]\!] &= \phi^+ n^+ + \phi^- n^-, \\ \{\!\{\Phi\}\!\} &= \tfrac{1}{2}\left(\Phi^+ + \Phi^-\right), \quad [\![\Phi]\!] &= \Phi^+ \cdot n^+ + \phi^- \cdot n^- \ . \end{aligned}$$

Auf einer Randkante d in Γ definieren wir

$$\begin{aligned} \{\!\{\phi\}\!\} &= \phi^+, \quad [\![\phi]\!] &= \phi^+ n^+, \\ \{\!\{\Phi\}\!\} &= \Phi^+, \quad [\![\Phi]\!] &= \Phi^+ \cdot n^+ \ . \end{aligned}$$

Um unsere Gleichungen einfacher aufschreiben zu können, definieren wir noch gebrochene Operatoren für die Divergenz, den Gradienten und den Laplaceoperator.

Definition 2.5 (Gebrochene Operatoren)
Es sei Ω ein Gebiet und $\mathcal{T}_h(\Omega)$ eine Zerlegung von Ω, dann definieren wir die gebrochenen Operatoren durch die Einschränkung auf ein Element $\kappa \in \mathcal{T}_h$.

- Der gebrochene Gradient $\nabla_h : H^1(\mathcal{T}_h) \to L^2(\mathcal{T}_h)$ ist definiert durch

$$(\nabla_h v)_{|\kappa} := \nabla(v_{|\kappa}), \qquad \text{für alle } \kappa \in \mathcal{T}_h \text{ und } v \in H^1(\mathcal{T}_h)$$

- Die gebrochene Divergenz $\mathrm{div}_h : \left[H^1(\mathcal{T}_h)\right]^k \to L^2(\mathcal{T}_h)$ ist definiert durch

$$(\mathrm{div}_h\, \tau)_{|\kappa} := \mathrm{div}(\tau_{|\kappa}), \qquad \text{für alle } \kappa \in \mathcal{T}_h \text{ und } \tau \in \left[H^1(\mathcal{T}_h)\right]^k$$

- Der gebrochene Laplace Operator $\Delta_h : H^1(\mathcal{T}_h) \to L^2(\mathcal{T}_h)$ ist definiert durch

$$(\Delta_h v)_{|\kappa} := \Delta(v_{|\kappa}), \qquad \text{für alle } \kappa \in \mathcal{T}_h \text{ und } v \in H^1(\mathcal{T}_h)$$

In den Umformungen der Modellgleichung brauchen wir die partielle Integration für die Divergenz. Deshalb leiten wir die Formel hier her, um die Herleitung in den Umformungen zu vermeiden.

Satz 2.6 (Partielle Integration für die Divergenz)
Es seien $v : U \subset \mathbb{R}^k \to \mathbb{R}^k$ und $u : U \subset \mathbb{R}^k \to \mathbb{R}$ zwei Funktionen, für die die partiellen Ableitungen existieren und die integrierbar über das Gebiet U sind. Dann gilt

$$\int_U u \cdot \operatorname{div} v \; dx = -\int_U \nabla u \cdot v \; dx + \int_{\partial U} un \cdot v \; ds \; .$$

Beweis. Aus Königsberger [9] haben wir den Gaußschen Integralsatz

$$\int_U \operatorname{div}(F) \; dx = \int_{\partial U} F \cdot n \; ds$$

für ein Vektorfeld $F : \mathbb{R}^k \to \mathbb{R}^k$. Wenn wir die Divergenz des Vektorfeldes $F := u \cdot v$ berechnen, erhalten wir

$$\begin{aligned} \operatorname{div}(u \cdot v) &= \sum_{i=1}^{k} \frac{\partial}{\partial x_i}(u \cdot v_i) = \sum_{i=1}^{k} \left(\frac{\partial u}{\partial x_i} v_i + u \frac{\partial v_i}{\partial x_i} \right) \\ &= \nabla u \cdot v + u \cdot \operatorname{div} v \; . \end{aligned}$$

Wenn wir diese Gleichung in die Gaußsche Integralformel einsetzen erhalten wir

$$\begin{aligned} \int_U \operatorname{div}(u \cdot v) \; dx &= \int_{\partial U} un \cdot v \; ds \\ \Leftrightarrow \int_U \nabla u \cdot v + u \cdot \operatorname{div} v \; dx &= \int_{\partial U} un \cdot v \; ds \\ \Leftrightarrow \int_U u \cdot \operatorname{div} v \; dx &= -\int_U \nabla u \cdot v \; dx + \int_{\partial U} un \cdot v \; ds \; . \end{aligned}$$

□

Nun können wir unser Modellproblem beschreiben und in den Kontext der DG Verfahren bringen.

Definition 2.7 (Problemstellung $\mathcal{P}$)
Es sei Ω ein Gebiet im $\mathbb{R}^k$ und $\mathcal{T}_h(\Omega)$ eine Zerlegung von Ω. Dann wollen wir die Gleichung

$$\begin{aligned} -\operatorname{div}(a(u)\nabla u) + \operatorname{div}(b(u)) &= f \quad \text{in } \Omega \\ u &= g_D \quad \text{auf } \Gamma_D \\ n \cdot a(u)\nabla u &= g_N \quad \text{auf } \Gamma_N \end{aligned}$$

für eine Funktion $u \in V$ lösen. Dabei sind $f \in L^2(\Omega), g_D \in L^2(\Gamma_D), b : V \to V^k$ und $a : V \to V^{k\times k}$ gegeben. Der Dirichlet Rand $\Gamma_D \subset \partial\Omega$ und der Neumann Rand $\Gamma_N \subset \partial\Omega$ sind disjunkt. Der Rand von Ω ist die Vereinigung $\Gamma := \partial\Omega = \Gamma_D \cup \Gamma_N \cup \Gamma_O$. Wobei Γ_O ein freier Rand ist, an dem keine Randbedingungen gesetzt sind. Für den Dirichlet-Rand soll gelten $\Gamma_D \neq \emptyset$.

Im weiterem Verlauf des Kapitel wird die Problemstellung $\mathcal{P}$ immer gelten und es wird nicht extra gesagt, dass wir $\mathcal{P}$ voraussetzen.

V ist ein Raum, den wir bei der Herleitung des DG Verfahrens noch genauer bestimmen müssen. Für einen klassischen Ansatz würde man V als $C^2(\Omega)$ wählen. Mit diesem Raum können wir aber nur wenig über die Existenz einer Lösung aussagen und die Beschreibung der Lösung ist hochgradig nicht trivial, wenn eine Lösung existiert.

Daher wollen wir als Raum V den gebrochenen Sobolev-Raum $H^2(\mathcal{T}_h)$ wählen. Mit dieser Wahl haben wir für die Funktion $v \in V$ einen Ausdruck für verschiedene Operatoren. Für die partiellen Ableitungen zweiter Ordnung haben wir eine Darstellung und für den Wert der Funktion auf dem Rand eines Elementes $\kappa \in \mathcal{T}_h$.

Die Problemstellung $\mathcal{P}$ schreiben wir zuerst in der äquivalenten Form:

$$\sigma = a(u)\nabla u \tag{I}$$
$$-\operatorname{div}\sigma + \operatorname{div} b(u) = f \tag{II}$$
$$u = g_D \quad \text{auf } \Gamma_D \tag{III}$$
$$n \cdot a(u)\nabla u = g_N \quad \text{auf } \Gamma_N \tag{IV}$$

Die Gleichung (I) wird mit einer Testfunktion $\tau \in \left[H^1(\mathcal{T}_h)\right]^k$ multipliziert und über $\kappa \in \mathcal{T}_h$ integriert.

$$\int_\kappa \sigma \cdot \tau \, dx = \int_\kappa a(u)\nabla u \cdot \tau \, dx \tag{I}$$

Durch die Anwendung des Satzes 2.6 auf der rechten Seite der Gleichung erhalten wir

$$\int_\kappa \sigma \cdot \tau \, dx = -\int_\kappa \operatorname{div}(a(u)^T\tau) \cdot u \, dx + \int_{\partial\kappa} ua(u)n \cdot \tau \, ds \, . \tag{I}$$

In dem Integral über den Rand von κ wird für u und τ jeweils die Spur der Funktionen auf dem Element verwendet. Da wir die einzelnen Elemente $k \in \mathcal{T}_h$ miteinander verbinden wollen, tauschen wir die Spur auf dem Rand durch eine Flussfunktion $\hat{u}$ aus. Wie diese Funktionen genau aussehen, werden wir später definieren. Es folgt

$$\int_\kappa \sigma \cdot \tau \, dx = -\int_\kappa \operatorname{div}(a(u)^T\tau) \cdot u \, dx + \int_{\partial\kappa} \hat{u}a(u)n \cdot \tau \, ds \, . \tag{I}$$

Wir benutzen Satz 2.6 ein zweites Mal, um die Divergenz von $a(u)^T\tau$ umzuwandeln.

$$\int_\kappa \sigma \cdot \tau \, dx = \int_\kappa a(u)\nabla u \cdot \tau \, dx + \int_{\partial\kappa} (\hat{u} - u)\, a(u)n \cdot \tau \, ds \tag{I}$$

Durch diese doppelte Partielle Integration haben wir uns einen neuen Term geschaffen, der ein Residuum auf dem Rand jeder Zelle beschreibt.

Nun wollen wir uns (II) zuwenden. Auch hier multiplizieren wir die Gleichung mit einer Testfunktion $v \in H^1(\mathcal{T}_h)$ und integrieren über $\kappa \in \mathcal{T}_h$.

$$-\int_\kappa \operatorname{div} \sigma \cdot v \; dx + \int_\kappa \operatorname{div} b(u) \cdot v \; dx = \int_\kappa f \cdot v \; dx \tag{II}$$

Durch partielle Integration der Terme auf der linken Seite erhalten wir

$$\int_\kappa \sigma \cdot \nabla v \; dx - \int_\kappa b(u) \cdot \nabla v \; dx - \int_{\partial\kappa} \sigma \cdot vn \; ds + \int_{\partial\kappa} b(u) \cdot vn \; ds = \int_\kappa f \cdot v \; dx \tag{II}$$

Auch hier führen wir wieder die Flussgrößen $\hat{\sigma}$ und $\hat{b}(u)$auf dem Rand von κ ein um eine Koppelung der Elemente zu erhalten.

$$\int_\kappa \sigma \cdot \nabla v \; dx - \int_\kappa b(u) \cdot \nabla v \; dx - \int_{\partial\kappa} \hat{\sigma} \cdot vn \; ds + \int_{\partial\kappa} \hat{b}(u) \cdot vn \; dx = \int_\kappa f \cdot v \; dx \tag{II}$$

Da wir τ frei wählen können, ersetzen wir τ durch ∇v in Gleichung (I).

$$\int_\kappa \sigma \cdot \nabla v \; dx = \int_\kappa a(u) \nabla u \cdot \nabla v \; dx + \int_{\partial\kappa} (\hat{u} - u) \, a(u) n \cdot \nabla v \; ds \tag{I}$$

Wir können nun (I) in (II) einsetzen. Dadurch fügen wir das System von Gleichungen wieder zusammen und brauchen σ nicht mehr bestimmen.

$$\begin{aligned}\int_\kappa f \cdot v \; dx = &\int_\kappa a(u) \nabla u \cdot \nabla v \; dx - \int_\kappa b(u) \cdot \nabla v \; dx \\ &+ \int_{\partial\kappa} (\hat{u} - u) \, a(u) n \cdot \nabla v \; ds - \int_{\partial\kappa} \hat{\sigma} \cdot vn \; ds + \int_{\partial\kappa} \hat{b}(u) \cdot vn \; ds\end{aligned}$$

Jetzt summieren wir über alle $\kappa \in \mathcal{T}_h$ und erhalten

$$\begin{aligned}\int_\Omega f \cdot v \; dx = &\int_\Omega a(u) \nabla_h u \cdot \nabla_h v \; dx - \int_\Omega b(u) \cdot \nabla_h v \; dx \\ &+ \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} (\hat{u} - u) \, a(u) n \cdot \nabla v \; ds \\ &- \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \hat{\sigma} \cdot vn \; ds + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \hat{b}(u) \cdot vn \; ds \; .\end{aligned}$$

Dadurch erhalten wir die *erste Flussformulierung*: Finde $u \in H^2(\mathcal{T}_h)$, so dass

$$\hat{A}(u, v) + \hat{B}(u, v) = \int_\Omega f \cdot v \; dx \qquad \text{für alle } v \in H^2(\mathcal{T}_h) \tag{DG}$$

gilt. Mit $\hat{A} : H^2(\mathcal{T}_h) \times H^2(\mathcal{T}_h) \to \mathbb{R}$ und $\hat{B} : H^2(\mathcal{T}_h) \times H^2(\mathcal{T}_h) \to \mathbb{R}$ definiert als

$$\hat{A}(u, v) = \int_\Omega a(u) \nabla_h u \cdot \nabla_h v \; dx + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} (\hat{u} - u) \, a(u) n \cdot \nabla v \; ds - \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \hat{\sigma} \cdot vn \; ds$$

$$\hat{B}(u, v) = -\int_\Omega b(u) \cdot \nabla_h v \; dx + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \hat{b}(u) \cdot vn \; ds \; .$$

Diese beiden Ausdrücke sind noch in Elementen κ von $\mathcal{T}_h$ geschrieben. Wir wollen die Ausdrücke umformen, so dass sie auf den Kanten von der Zerlegung $\mathcal{T}_h$ beruhen. Dafür definieren wir

$$\Gamma_I := \bigcup_{\kappa \in \mathcal{T}_h} \partial\kappa \setminus \Gamma$$

als die inneren Kanten der Zerlegung. In jeder Summe kommt eine innere Kante zweimal vor. Dafür beweisen wir jetzt zwei Formeln, mit denen wir die elementbasierte Form in eine kantenbasierte umschreiben können.

Satz 2.8
Sei $q \in \partial L^2(\mathcal{T}_h)$ und $\Phi \in \left[\partial L^2(\mathcal{T}_h)\right]^k$, dann ist

$$\sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa \setminus \Gamma} \Phi^+ \cdot q^+ n^+ \, ds = \int_{\Gamma_I} \{\{\Phi\}\} \cdot [\![q]\!] \, ds + \int_{\Gamma_I} [\![\Phi]\!] \{\{q\}\} \, ds$$

$$\sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \Phi^+ \cdot q^+ n^+ \, ds = \int_{\Gamma_I \cup \Gamma} \{\{\Phi\}\} \cdot [\![q]\!] \, ds + \int_{\Gamma_I} [\![\Phi]\!] \{\{q\}\} \, ds \; .$$

Beweis. Wir schauen uns nur die Ausdrücke in den Integralen an, dadurch erhalten wir für eine Kante aus Γ_I

$$\begin{aligned}
\{\{\Phi\}\} \cdot [\![q]\!] + [\![\Phi]\!] \{\{q\}\} =& \frac{1}{2}\left(\Phi^+ + \Phi^-\right) \cdot \left(q^+ n^+ + q^- n^-\right) \\
&+ \frac{1}{2}\left(\Phi^+ \cdot n^+ + \Phi^- \cdot n^-\right)\left(q^+ + q^-\right) \\
=& \frac{1}{2}\left(\Phi^+ \cdot q^+ n^+ + \Phi^- \cdot q^+ n^+ + \Phi^+ \cdot q^- n^- + \Phi^- \cdot q^- n^-\right) \\
&+ \frac{1}{2}\left(\Phi^+ \cdot q^+ n^+ + \Phi^- \cdot q^+ n^- + \Phi^+ \cdot q^- n^+ + \Phi^- \cdot q^- n^-\right) \\
=& \Phi^+ \cdot q^+ n^+ + \Phi^- \cdot q^- n^- \; .
\end{aligned}$$

. Die mittleren Terme heben sich wegen $n^+ = -n^-$ auf. Dadruch erhalten wir die erste Gleichung in dem Satz. Auf einer Randkante haben wir $\Phi^+ \cdot q^+ n^+ = \{\{\Phi\}\} \cdot [\![q]\!]$ nach der Definition der Operatoren auf dem Rand. □

Mit diesen Formeln können wir $\hat{A}$ umschreiben, indem wir den Satz einmal mit $a(u)^T \nabla v$ und $\hat{u} - u$ verwenden und einmal mit $\hat{\sigma}$ und v.

$$\begin{aligned}
\hat{A}(u,v) =& \int_\Omega a(u) \nabla_h u \cdot \nabla_h v \, dx + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} (\hat{u} - u)\, a(u) n \cdot \nabla v \, ds - \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa} \hat{\sigma} \cdot v n \, ds \\
=& \int_\Omega a(u) \nabla_h u \cdot \nabla_h v \, dx \\
&+ \int_{\Gamma_I \cup \Gamma} \left\{\left\{a(u)^T \nabla v\right\}\right\} \cdot [\![\hat{u} - u]\!] \, ds + \int_{\Gamma_I} \left[\left[a(u)^T \nabla v\right]\right] \{\{\hat{u} - u\}\} \, ds \\
&- \int_{\Gamma_I \cup \Gamma} \{\{\hat{\sigma}\}\} \cdot [\![v]\!] \, ds - \int_{\Gamma_I} [\![\hat{\sigma}]\!] \{\{v\}\} \, ds
\end{aligned}$$

Für $\hat{B}$ benutzen wir die Formel mit $\hat{b}(u)$ und v.

$$\begin{aligned}\hat{B}(u,v) &= -\int_\Omega b(u)\cdot\nabla_h v\ dx + \sum_{\kappa\in\mathcal{T}_h}\int_{\partial\kappa}\hat{b}(u)\cdot vn\ ds \\ &= -\int_\Omega b(u)\cdot\nabla_h v\ dx + \int_{\Gamma_I\cup\Gamma}\{\!\{\hat{b}(u)\}\!\}\cdot[\![v]\!]\ ds + \int_{\Gamma_I}[\![\hat{b}(u)]\!]\,\{\!\{v\}\!\}\ ds\end{aligned}$$

Mit diesen beiden Darstellungen von $\hat{A}$ und $\hat{B}$ haben wir eine Form erreicht, die es uns ermöglicht, Aussagen über die Konsistenz und Stabilität des Verfahrens herzuleiten.

Diese Arbeit beschreibt nur eine Einführung in das Forschungsgebiet der diskontinuierliche Galerkin Verfahren. Deshalb werden wir die Analyse zur Stabilität und Konsistenz hier nicht durchführen. Ebenso stellen die Flussoperatoren $\hat{u}$, $\hat{\sigma}$ und $\hat{b}(u)$ ein komplexes Forschungsgebiet dar. Deshalb verweisen wir für eine tiefer gehende Herleitung auf die Arbeit von Hartmann [8].

Wir werden aber die Konvergenzaussagen aus Hartmann [8] hier für zwei Beispiele wiedergeben. Diese Aussagen basieren auf einer Diskretisierung des Raums $H^2(\mathcal{T}_h)$ durch den Raum $V_{h,p}^k$.

Definition 2.9 ($V_{h,p}^k$ der gebrochene Polynomialraum)
Es sei $p \geq 0$. h steht für die Zerlegung $\mathcal{T}_h(\Omega)$. Dann ist $V_{h,p}^k$ der Raum der diskontinuierlichen stückweisen Polynome vom Grad p:

$$V_{h,p}^k = \left\{v_h \in L^2(\Omega) : v_h|_\kappa \circ \phi_\kappa \in B_p(\hat{\kappa}), \kappa \in \mathcal{T}_h\right\}$$

$B_p(\hat{\kappa})$ steht für die Polynomiale Basis des Einheitselements $\hat{\kappa}$ für κ. ϕ_κ ist die Transformation von $\hat{\kappa}$ auf κ.

2.2. Beispiel 1 - Die lineare Advektionsgleichung

Wenn wir in unserer Problemstellung $a(u) = 0$ setzen und $b(u) = bu$ definieren, b ist linear in u, erhalten wir die lineare Advektionsgleichung. Für diese Gleichung bekommen wir ein konsistentes und stabiles Verfahren, wenn wir die Flussfunktion $\hat{b}$ wie folgt definieren:

$$\hat{b}(u) = \hat{b}(u^+, u^-, n^+) = b\cdot n\frac{1}{2}(u^+ + u^-) + \frac{1}{2}|b\cdot n|(u^+ - u^-)$$

Mit diesem $\hat{b}$ kann man beweisen, dass folgendes gilt.

Satz 2.10 (Fehlerabschätzung mit der exakten Lösung)
Sei $u \in H^{p+1}(\Omega)$ die exakte Lösung der linearen Advektionsgleichung

$$\begin{aligned}\operatorname{div}(bu) &= f \quad \text{in } \Omega \\ u &= g_D \quad \text{auf } \Gamma_D = \Gamma_- \\ \Gamma_- &:= \{x \in \Gamma : b(x)\cdot n(x) < 0\} \\ \Gamma_+ &:= \Gamma \setminus \Gamma_-\end{aligned}$$

Sei $u_h \in V_{h,p}^k$ die Lösung zu der DG Gleichung

$$B_h(u_h, v_h) = F(v_h), \qquad \text{für alle } v_h \in V_{h,p}^d$$

mit

$$\begin{aligned} B_h(u,v) = &- \int_\Omega bu \cdot \nabla_h v \; dx + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa \setminus \Gamma} \left(b \cdot n \frac{1}{2}(u^+ + u^-) + \frac{1}{2}|b \cdot n|(u^+ - u^-) \right) v \; ds \\ &+ \int_{\Gamma_+} b \cdot nuv \; ds \\ F_h(v) = &\int_\Omega fv \; dx - \int_{\Gamma_-} b \cdot n g_D v \; ds \end{aligned}$$

Dann erhalten wir

$$|||u - u_h|||_{b_0} \leq Ch^{p+1/2}|u|_{H^{p+1}(\Omega)},$$

wobei die Norm $|||\cdot|||_{b_0}$ gegeben ist durch

$$|||v|||_{b_0}^2 = c_0||v||^2 + \sum_{e \in \Gamma_I} \int_e \frac{1}{2}|b \cdot n|(v^+ - v^-)^2 \; ds + \frac{1}{2}\int_\Gamma |b \cdot n| v^2 \; ds \; .$$

Dieses Ergebnis ist direkt aus Hartmann [8] Seite 30 übernommen. Die Definition von B_h beruht auf der elementbasierten Form von $\hat{B}$. In die Gleichung wurde auch die Randbedingung eingebracht und verarbeitet. Der Satz besagt, dass wir mit der entsprechenden Wahl von $\hat{b}$ ein stabiles, konsistentes und konvergentes Verfahren erhalten.

2.3. Beispiel 2 - Die Poission Gleichung

Wenn wir in unserer Problemstellung $a(u) = I$ setzen und $b(u) = 0$ betrachten, erhalten wir die Poission Gleichung mit Dirichlet- und Neumannrand. Mit der folgenden Definition von $\hat{u}$ und $\hat{\sigma}$ bekommen wir ein stabiles und konsistentes Verfahren, bekannt unter dem Namen *SIPG*.

$$\begin{array}{llllll} \hat{u}_h & = & \{\{u_h\}\}, & \hat{\sigma}_h & = & \{\{\nabla_h u_h\}\} - \delta^{ip}(u_h) \quad \text{auf } \Gamma_I \\ \hat{u}_h & = & g_D, & \hat{\sigma}_h & = & \nabla_h u_h - \delta_\Gamma^{ip}(u_h) \quad \text{auf } \Gamma_D \\ \hat{u}_h & = & u_h, & \hat{\sigma}_h & = & g_N n \quad \text{auf } \Gamma_N \end{array}$$

mit

$$\begin{array}{lll} \delta^{ip}(u_h) & = & \delta \, [\![u_h]\!] = C_{IP}\frac{p^2}{h} \, [\![u_h]\!] \quad \text{auf } \Gamma_I \\ \delta_\Gamma^{ip}(u_h) & = & \delta(u_h - g_D)n = C_{IP}\frac{p^2}{h}(u_h - g_D)n \quad \text{auf } \Gamma_D \; . \end{array}$$

Damit können wir nun die folgende Fehlerabschätzung beweisen.

Satz 2.11 (L^2-Fehlerabschätzung)
Sei $u \in H^{p+1}(\Omega)$ die exakte Lösung der Poission Gleichung

$$\begin{aligned} \operatorname{div}(\nabla u) =& f \quad && \text{in } \Omega \\ u =& g_D && \text{auf } \Gamma_D \\ n \cdot \nabla u =& g_N && \text{auf } \Gamma_N \ . \end{aligned}$$

Sei $u_h \in V_{h,p}^k$ die Lösung zu der DG Gleichung

$$A_h(u_h, v_h) = F(v_h), \qquad \text{für alle } v_h \in V_{h,p}^d$$

mit

$$\begin{aligned} A_h(u,v) =& \int_\Omega \nabla_h u \cdot \nabla_h v \ dx \\ &+ \int_{\Gamma_I \cup \Gamma_D} (- [\![u]\!] \cdot \{\!\{\nabla_h v\}\!\} - \{\!\{\nabla_h u\}\!\} \cdot [\![v]\!]) \ ds \\ &+ \int_{\Gamma_I \cup \Gamma_D} \delta [\![u]\!] \cdot [\![v]\!] \ ds, \\ F_h(v) =& \int_\Omega f v \ dx + \int_{\Gamma_D} -g_D n \cdot \nabla v \ ds + \int_{\Gamma_D} \delta g_D v \ ds + \int_{\Gamma_N} g_N v \ ds \ . \end{aligned}$$

Dann erhalten wir

$$||u - u_h||_{L^2(\Omega)} \leq C h^{p+1} |u|_{H^{p+1}(\Omega)} \ .$$

Dieser Satz ist direkt aus Hartmann [8] Seite 52 übernommen. A_h basiert auf der Kantenformulierung von $\hat{A}$. In A_h wurden die Definition von $\hat{\sigma}$ und $\hat{u}$ eingesetzt sowie die Gleichungen auf dem Rand behandelt. Der Satz besagt, dass wir ein stabiles, konsistentes und konvergentes Verfahren erhalten.

2.4. Die kompressiblen Navier Stokes Gleichungen

Die kompressiblen Navier Stokes Gleichungen beschreiben die Strömung eines kompressiblen Gases oder einer Flüssigkeit. Die Gleichungen beziehen sich auf die Erhaltung von Masse, Impuls und Energie. Sie lauten

$$\operatorname{div}\left(\mathcal{F}^c(u) - \mathcal{F}^v(u, \nabla u)\right) \equiv \sum_{i=1}^{k} \left[\frac{\partial}{\partial x_i} f_i^c(u) - \frac{\partial}{\partial x_i} f_i^v(u, \nabla u) \right] = 0 \qquad \text{in } \Omega \ . \quad \text{(NV)}$$

Dabei ist u der Vektor der konservativen Variablen, F^c der konvektive Fluss und F^v der viskose Fluss. Die Formeln für die Flüsse und u lauten:

$$u = \begin{pmatrix} \rho \\ \rho v_1 \\ \vdots \\ \rho v_k \\ \rho E \end{pmatrix}, \quad f_i^c(u) = \begin{pmatrix} \rho v_i \\ \rho v_1 v_i + \delta_{1i} p \\ \vdots \\ \rho v_k v_i + \delta_{ki} p \\ \rho H v_i \end{pmatrix}, \quad f_i^v(u, \nabla u) = \begin{pmatrix} 0 \\ \tau_{1i} \\ \vdots \\ \tau_{ki} \\ \sum_{j=1}^k \tau_{ij} v_j + \frac{\partial}{\partial x_i} \mathcal{K} T \end{pmatrix}$$

für i von 1 bis k. Die Variablen sind durch die Dichte ρ, die Geschwindigkeit $v = (v_1, \ldots, v_k)^T$ und die totale Energie E definiert. Aus den Variablen setzen sich der Druck p, der viskosen Stresstensor τ, die Temperatur T und die totale Enthalpie H zusammen. Für H, p, τ und $\mathcal{K}T$ gelten die Gleichungen

$$\begin{aligned} H =& E + \frac{p}{\rho} = e + \frac{1}{2}v^2 + \frac{p}{\rho}, \\ p =& (\gamma - 1)\rho e, \\ \gamma =& \frac{c_p}{c_v}, \\ \tau =& \mu\left(\nabla v + (\nabla v)^T - \frac{2}{3}\operatorname{div}(v) I\right), \\ \mathcal{K}T =& \frac{\mu\gamma}{Pr}\left(E - \frac{1}{2}v^2\right), \end{aligned}$$

In den Gleichungen beschreibt Pr die Prandtl Nummer, μ den dynamischen Viskositäts-Faktor und $\mathcal{K}$ die Wärmeleitfähigkeit. $\gamma = \frac{c_p}{c_v}$ beschreibt das Verhältnis der spezifischen Wärmekapazität bei konstantem Druck, c_p, und konstantem Volumen, c_v. Für Luft ist $\gamma = 1,4$.

Die Navier Stokes Gleichung (NV) hat noch nicht die Form, wie wir sie in $\mathcal{P}$ festgelegt haben. Man kann die Gleichung in die Form

$$\sum_{i=1}^{k} \frac{\partial}{\partial x_i} f_k^c(u) - \sum_{i,j=1}^{k} \frac{\partial}{\partial x_i}\left(G_{ij}(u)\frac{\partial u}{\partial x_j}\right) = 0 \qquad \text{auf } \Omega \tag{NV}$$

bringen. $G_{ij}(u)$ ist durch $G_{ij}(u) = \frac{\partial}{\partial x_j} f_k^v(u, \nabla v)$ definiert. Dadurch erhalten wir eine Gleichung, die dieselbe Form hat wie in der Problemstellung $\mathcal{P}$, nur dass wir keine skalare sondern eine vektorielle Gleichung haben. Die Umformungen, die wir mit der Ausgangsgleichung des Problems $\mathcal{P}$ gemacht haben, lassen sich genauso mit der Gleichung (NV) durchführen. Diesmal müssen wir aber die Testfunktionen vektoriell wählen und die Sprungoperatoren anpassen. Am Ende erhalten wir aber eine Gleichung, die genau dieselbe Form hat wie die Gleichung (DG). Wir können also sagen, dass wir ein diskontinuierliches Galerkin Verfahren für die Navier Stokes Gleichungen aufschreiben können. Wie die Behandlung der Ränder erfolgt und wie die Flussfunktionen definiert werden, möchten wir hier nicht näher betrachten. In Hartmann [8] wurde kein Beweis für die Stabilität und das Konvergenzverhalten gegeben, daher können wir auch nicht ohne weiteres einen Satz wiedergeben oder herleiten. Die numerischen Ergebnisse in Hartmann [8] Seite 99f lassen auf ein stabiles und konvergentes Verfahren der Ordnung $\mathcal{O}(h^{p+1})$ schließen.

2.5. Implementation und Ableitung von DG Verfahren

Wir können erst mit dem Automatischen Differenzieren von DG Verfahren anfangen, wenn ein Code geschrieben wurde, der eine Lösung erzeugt. Deshalb schauen wir uns zuerst die Diskretisierung des DG Verfahrens an. Ausgehend von der Gleichung

$$\hat{A}(u,v) + \hat{B}(u,v) = \int_\Omega f \cdot v \; dx \qquad \text{für alle } v \in H^2(\mathcal{T}_h) \tag{DG}$$

erzeugen wir eine diskrete Lösung in dem Raum $V_{h,p}^d$, indem wir die Testfunktion v und die Lösung u in diesem Raum ansetzten. Dadurch erhalten wir die Testfunktionen $v_h \in V_{h,p}^d$ und die Lösung $u_h \in V_{h,p}^d$. Für $V_{h,p}^d$ seien $\phi_1, \ldots, \phi_m : V_{h,p}^d \to \mathbb{R}$ eine endliche Basis. Dann können wir u_h darstellen als

$$u_h := \sum_{i=1}^m \lambda_i \phi_i \quad \text{ mit } \lambda_i \in \mathbb{R}.$$

Mit u_h erhalten wir

$$\hat{A}(u_h,v_h) + \hat{B}(u_h,v_h) = \int_\Omega f \cdot v_h \; dx \qquad \text{für alle } v_h \in V_{h,p}^d \; .$$

Diese Gleichung können wir in eine Residuumsgleichung umschreiben, indem wir die rechte Seite auf die linke schieben. Wir erhalten

$$\hat{R}(u_h,v_h) = \hat{A}(u_h,v_h) + \hat{B}(u_h,v_h) - \int_\Omega f \cdot v_h \; dx = 0 \qquad \text{für alle } v_h \in V_{h,p}^d \; .$$

Nun wollen wir noch die Abhängigkeit von v_h eliminieren. Sei $v_h = \sum_{i=1}^m \alpha_i \phi_i$ mit $\alpha_i \in \mathbb{R}$. Da $\hat{A}$, $\hat{B}$ und das Integral linear in v_h sind, bekommt man

$$\sum_{i=1}^m \alpha_i \hat{R}(u_h,\phi_i) = \sum_{i=1}^m \alpha_i \left(\hat{A}(u_h,\phi_i) + \hat{B}(u_h,\phi_i) - \int_\Omega f \cdot \phi_i \; dx \right) = 0 \qquad \text{für alle } \alpha_i \in \mathbb{R}.$$

Die ϕ_i bilden eine Basis von $V_{h,p}^d$, dann ist die Gleichung nur dann erfüllt, wenn

$$\hat{R}(u_h,\phi_i) = \hat{A}(u_h,\phi_i) + \hat{B}(u_h,\phi_i) - \int_\Omega f \cdot \phi_i \; dx = 0 \qquad \text{für alle } i = 1 \ldots m$$

gilt. Das Residuum muss also für jede einzelne Basisfunktion ϕ_i Null werden. Wenn wir jetzt das Residuum als Vektor

$$R(u_h) = R(\sum_{i=1}^m \lambda_i \phi) = \begin{pmatrix} \hat{R}(u_h,\phi_1) \\ \vdots \\ \hat{R}(u_h,\phi_m) \end{pmatrix} = 0$$

schreiben, erhalten wir eine Funktion $R : \mathbb{R}^m \to \mathbb{R}^m$.

Dieses Gleichungssystem kann man nun mit einem Löser für ein nichtlineares Gleichungssystem lösen. Hier können wir z.B. das Newton Verfahren verwenden und erhalten die Iterationsvorschrift:

$$u_{k+1} = u_k - \frac{dR}{du}^{-1} R(u_k)$$

Damit zeigt sich auch schon die erste Möglichkeit, bei der wir AD verwenden können. Die Ableitung $\frac{dR}{du}$ ist eine quadratische Matrix, daher ist es egal, ob wir sie im Vorwärts- oder Rückwärtsmodus berechnen. Im normalen Vorwärtsmodus müssen wir m mal $R(u)$ berechnen, um die ganze Jacobimatrix zu erhalten. Wenn wir AD im Vektormodus ausführen, braucht man sogar nur eine Vorwärtsauswertung von $R(u)$. Die Speichergröße für diese Vorwärtsauswertung steigt aber im Vergleich zu der Speichergröße der normalen Auswertung sehr stark an.

Für jeden skalaren Wert in der normalen Funktion benötigen wir m weitere skalare Werte für die Speicherung der Ableitung. Man braucht deshalb eine sehr effektive Sparse-Implementierung des AD Vecktormodus.

In Padge ist $\frac{dR}{du}$ als Funktion implementiert. Dadurch können wir den Newton-Schritt als

$$u_{k+1} = u_k - \mathrm{JR}(u_k)^{-1} R(u_k)$$

schreiben. Mit $\mathrm{JR}(u_k)$ ist die Implementierung von $\frac{dR}{du}(u_k)$ gemeint. Für unsere Anwendung wollen wir eine Formoptimierung bezüglich des Randes $\Gamma_\omega \subset \partial\Omega$ durchführen. Das Residuum R muss also noch zusätzlich von einem weitere Parameter x abhängen. Angenommen wir können den Rand durch $x \in \mathbb{R}^n$ parametrisieren, dann erhalten wir für unser Residuum

$$R(u_h, x) = R(\sum_{i=1}^{m} \lambda_i \phi, x) = \begin{pmatrix} \hat{R}(u_h, \phi_1, x) \\ \vdots \\ \hat{R}(u_h, \phi_m, x) \end{pmatrix} = 0$$

In den Formeln für $\hat{A}$ und $\hat{B}$ geht x nicht über die Gleichungen, sondern nur über das Integralgebiet bzw. den Rand ein. Wenn wir z.B. einen Term

$$w(u_h, \phi) = \int_\kappa h(u_h, \phi)\ dx$$

haben, sehen die Gleichungen durch den neuen Parameter x folgendermaßen aus

$$w(u_h, \phi, x) = \int_{\kappa(x)} h(u_h, \phi)\ dx$$

Für diese Formel müssten wir eine Ableitung für das Integrationsgebiet eines Integrals entwickeln. In dem Code werden aber nicht diese Integrale berechnet, sondern die Integrale werden durch eine Quadratur ausgewertet. Die Quadratur besteht aus den Quadraturpunkten $p_i \in R^k$, die auf einem Einheitselement $\hat{\kappa}$ definiert sind und aus zugehörigen Gewichten $\lambda_i \in \mathbb{R}$. Die Quadraturpunkte p_i werden durch eine Transformationsabbildung Φ_κ von dem Einheitselement $\hat{\kappa}$ auf ein Element $\kappa \in \mathcal{T}_h(\Omega)$ übertragen. Die Transformationsabbildung Φ_κ hängt von dem Zielelement κ ab. κ kann auf dem Rand liegen und somit von x abhängen. Dadurch ergibt sich für w

$$w(u_h, \phi_i, x) = \sum_i \lambda_i h(u_h(\Phi_{\kappa(x)}(p_i), \phi(\Phi_{\kappa(x)}(p_i))\ .$$

Da das Element κ meist nicht direkt benutzt wird, kann man auch schreiben

$$w(u_h, \phi_i, x) = \sum_i \lambda_i h(u_h(\Phi_\kappa(x, p_i), \phi(\Phi_\kappa(x, p_i)) \ .$$

In der Funktion für die Auswertung von R und JR wird die Randparametrisierung x direkt eingehen. Die Ausdrücke $\frac{\partial R}{\partial x}(u_h, x)$, $\frac{\partial \, \mathrm{JR}}{\partial x}(u_h, x)$ sind dadruch wohldefiniert.

Die Auswertung des DG Residuums können wir dadruch nach dem Parameter x ableiten, selbst wenn sich dieser Parameter auf das Gebiet Ω bezieht.

3. One Shot Optimierung

In der Optimierung besteht eine Problemstellung meist aus einem zu optimierenden Zielfunktional und einer Nebenbedingung. Für die von uns betrachteten Fälle ist die Nebenbedingung eine Partielle Differential Gleichung (PDE). Um diese PDE zu lösen, kann man in den meisten Fällen nur ein iteratives Verfahren benutzen, da man die analytische Lösung der PDE nicht ermitteln kann. Ein zweites iteratives Verfahren kommt durch die Lösung der adjungierten PDE hinzu. Die adjungierte PDE beschreibt den Einfluss des Gebietes auf das Zielfunktional. Nachdem man die primale PDE und die adjungierte PDE in jeweils ihrer eigenen Iteration gelöst hat, findet ein Update in den Parametern für die Optimierung statt. Deshalb müssen die primale PDE und adjungierte PDE erneut mit einer Iteration gelöst werden. Danach folgt wieder ein Update in den Parametern der Optimierung. Diese Schleife wird so oft wiederholt, bis man das Optimum oder eine Verbesserung erreicht hat, die groß genug ist.

Es gibt zwei Wege, diese Optimierung zu beschleunigen. Als erstes kann man die Startwerte für die Iterationen der PDE's geschickt wählen, sodass man näher an dem Fixpunkt der Iteration startet. Als zweites kann man für Lösungen zu den PDE's, die noch nicht viel mit dem Optimum zu tun haben, eine geringere Genauigkeit verlangen, da man von der echten Lösung noch sehr weit entfernt ist. In einem One Shot Verfahren möchte man für beide Beschleunigungen die extreme Variante verwenden. Man macht in jeder Iteration für die Partiellen Differentialgleichungen nur einen Schritt, und basierend darauf, das Update in den Optimierungvariablen. Die Startwerte für die Iterationen gibt es nicht mehr, da man immer das Ergebnis des letzten Schrittes verwendet.

In diesem Kapitel werden wir das One Shot Verfahren nach den Papern von Gauger et al. [15], Hamdi et al. [6] und Griewank et al. [5] herleiten. Dabei wird auf das One Shot Verfahren an sich und auf den Präkonditionierer für das Design Update eingangen.

3.1. Das One Shot Verfahren

Für das One Shot Verfahren benennen wir zuerst das Minimierungsproblem.

Definition 3.1 (Problemstellung (P))
Es sei $f : Y \times U \to \mathbb{R}$ das Zielfunktional. $Y \subset \mathbb{R}^m$ ist konvex und beschreibt den Zustand des Systems. $U \subset \mathbb{R}^n$ ist der Raum der Designvariablen. Die Nebenbedingung $c(y, u) = 0$ wird durch den Fixpunkt der Iterationsvorschrift $G : Y \times U \to Y$ beschrieben. f und G seien zweimal stetig differenzierbar.

Dann hat das Optimierungsproblem die Form

$$\min_{u \in U} f(y,u), \quad \text{so dass } y = G(y,u)\ . \tag{P}$$

G hat den Kontraktionfaktor $\rho < 1$. Es gilt für fixes $u \in \mathbb{R}^n$

$$\|G_y(y,u)\| = \left\|G_y(y,u)^T\right\| \leq \rho < 1 \qquad \text{für alle } y \in \mathbb{R}^m\ .$$

Die Voraussetzung, dass der Kontraktionsfaktor G kleiner als 1 ist, ergibt die Schlussfolgerung.

Satz 3.2
P soll gelten, dann haben wir für $y_1, y_2 \in Y$ und $u \in U$ beliebig die Relation

$$\|G(y_1,u) - G(y_2,u)\| \leq \rho \|y_1 - y_2\|$$

Daraus folgt, das für alle $u \in U$ ein $y^* \in Y$ existier, sodass

$$y^*(u) = G(y^*(u), u)$$

ein eindeutiger Fixpunkt von G ist.

Beweis. Für $y_1, y_2 \in Y$ und $d \in \mathbb{R}^m$ definieren wir die Funktion $f(y) := G(y,u) \cdot d$.Wir wissen auf Grund der Konvexität von Y, dass auch die Strecke $\overline{y_1 y_2}$ in Y liegt. Wenn wir den Mittelwertsatz für reelle Funktionen auf f anwenden, ergibt sich daraus, dass ein $\zeta \in \overline{y_1 y_2}$ existiert, so dass

$$\begin{aligned} f(y_2) - f(y_1) &= f_y(\zeta)(y_2 - y_1) \\ \Leftrightarrow (G(y_2,u) - G(y_1,u)) \cdot d &= d^T G_y(\zeta,u)(y_2 - y_1) \end{aligned}$$

gilt. Wir setzten jetzt $d = G(y_2,u) - G(y_1,u)$ und schätzen mit der Cauchy Schwarzen Ungleichung (CSU) ab

$$\begin{aligned} \|G(y_2,u) - G(y_1,u)\|_2^2 = &\ (G(y_2,u) - G(y_1,u))^T G_y(\zeta,u)(y_2 - y_1) \\ \overset{CSU}{\leq} &\ \|G(y_2,u) - G(y_1,u)\|_2 \|G_y(\zeta,u)(y_2 - y_1)\|_2 \\ \leq &\ \|G(y_2,u) - G(y_1,u)\|_2 \|G_y(\zeta,u)\|_2 \|y_2 - y_1\|_2 \\ \leq &\ \|G(y_2,u) - G(y_1,u)\|_2 \rho \|y_2 - y_1\|_2 \end{aligned}$$

Für $\|G(y_2,u) - G(y_1,u)\| \neq 0$ können wir beide Seiten durch $\|G(y_2,u) - G(y_1,u)\|$ teilen und erhalten die erste Gleichung

$$\|G(y_1,u) - G(y_2,u)\| \leq \rho \|y_1 - y_2\|$$

Diese Gleichung gibt uns die Lipschitzstetigkeit von G mit einer Liptschitzkonstante $\lambda = \rho < 1$. Dadurch folgt aus dem Banachschen Fixpunktsatz, dass für jedes $u \in U$ ein eindeutiges $y^* \in Y$ existiert, so dass $y^* = G(y^*,u)$ gilt. □

Wir bilden nun die Lagrangefunktion für das Problem P

$$L(u, y, \lambda) = f(y, u) + (G(y, u) - y)^T \lambda = N(y, \lambda, u) - y^T \lambda \ .$$

Die Lagrangefunktion schreiben wir mit dem verschobenen Lagrangeterm

$$N(y, \lambda, u) := f(y, u) + G(y, u)^T \lambda \ .$$

Nach den notwendigen Bedingungen für einen Extramal-Punkt (y^*, λ^*, u^*) des Problems P muss die Bedingung $\nabla L(y^*, \lambda^*, u^*) = 0$ gelten. Es folgt

$$\begin{aligned}
\frac{\partial L}{\partial \lambda}(y^*, \lambda^*, u^*) &= N_\lambda(y^*, \lambda^*, u^*)^T - y^* = G(y^*, u^*) - y^* = 0 \\
\frac{\partial L}{\partial y}(y^*, \lambda^*, u^*) &= N_y(y^*, \lambda^*, u^*)^T - \lambda^* = 0 \\
\frac{\partial L}{\partial u}(y^*, \lambda^*, u^*) &= N_u(y^*, \lambda^*, u^*)^T = 0
\end{aligned}$$

Wir erhalten das Gleichungssystem

$$\begin{aligned}
y^* &= G(y^*, u^*) && \text{(Primal)} \\
\lambda^* &= N_y(y^*, \lambda^*, u^*)^T && \text{(Adjungiert)} \\
0 &= N_u(y^*, \lambda^*, u^*)^T && \text{(Design)}
\end{aligned}$$

Diese drei Gleichungen verwendet man, um eine gekoppelte Iteration für die Variablen y, λ und u durchzuführen. In dem Paper Griewank et al. [5] wird gezeigt wie, man diese gekoppelte Iteration herleitet.

Das Update in den einzelnen Variablen sieht dann folgerndermaßen aus:

$$\begin{bmatrix} y_{k+1} \\ \lambda_{k+1} \\ u_{k+1} \end{bmatrix} = \begin{bmatrix} G(y_k, u_k) \\ N_y(y_k, \lambda_k, u_k)^T \\ u_k - B_k^{-1} N_u(y_k, \lambda_k, u_k)^T \end{bmatrix}$$

Der Updateschritt ist durch

$$s(y, \lambda, u) = \begin{bmatrix} \Delta y \\ \Delta \lambda \\ \Delta u \end{bmatrix} = \begin{bmatrix} G(y, u) - y \\ N_y(y, \lambda, u)^T - \lambda \\ -B^{-1} N_u(y, \lambda, u)^T \end{bmatrix}$$

gegeben.

In dem Update für u wird der Präkonditionierer B benutzt. Zur Zeit ist B noch nicht genauer spezifiziert. Es ist das Ziel, alle Voraussetzungen herzuleiten, um für B eine klare Definition zu haben und die Berechnung von B auf Basis von Padge zu ermöglichen. Wir werden genauer auf den Präkonditionierer in den folgenden Kapiteln eingehen.

Zuerst beschäftigen wir uns mit dem Update im Primalen und Adjungierten, denn für diese beiden Gleichungen wird in Hamdi et al. [6] der flogende Satz bewiesen:

Satz 3.3 (Adjungierte Verzögerung)
Angenommen für ein fixiertes $u \in U$, sind die Funktionen f und G einmal differenzierbar und die Ableitungen sind Lipschitzstetig bezüglich y nahe eines Fixpunktes $y^* = y^*(u)$. Für die primalen Iterierten y_k soll gelten, dass sie zu einer Lösung y^* konvergieren. Es gilt

$$\lim_{k\to\infty} \frac{\|\Delta y_k\|}{\|\Delta y_{k-1}\|} = \rho^* \equiv \|G_y(y^*, u)\| .$$

Wir definieren für ein beliebiges $\epsilon > 0$ und $\alpha, \beta \in \mathbb{R}^+$ das kleinste Paar natürlicher Zahlen $(l_p^\epsilon, l_d^\epsilon) \in \mathbb{N}^2$ mit der Eigenschaft

$$\sqrt{\alpha} \left\| \Delta y_{l_p^\epsilon} \right\| \leq \epsilon \qquad \textit{und} \qquad \sqrt{\beta} \left\| \Delta \lambda_{l_d^\epsilon} \right\| \leq \epsilon$$

Dann erhalten wir

$$\lim_{k\to\infty} \sup \frac{l_d^\epsilon}{l_p^\epsilon} \leq 1$$

Der Satz besagt, dass bei demselben Fehler der Index für die primale Iteration immer etwas kleiner als der Index für die adjungierte Iteration ist. Diese Verzögerung bleibt aber wegen der lim sup Aussage beschränkt und vergrößert sich nicht weiter. Die primale und adjugierte Iteration konvergieren deshalb mit derselben Geschwindigkeit ρ, nur dass die adjungierte Iteration etwas verzögert konvergiert. Diese Aussage gilt aber nur für ein fixiertes u.

Für ein Update in allen drei Komponenten ist es mit der Konvergenzaussage schwieriger. In Griewank et al. [5] wird gezeigt, dass man Konvergenz erhält, wenn der Präkonditionirer B_k groß genug gewählt wird. Durch die Jacobimatrix des Updates

$$J^* = \frac{\partial(y_{k+1}, \lambda_{k+1}, u_{k+})}{\partial(y_k, \lambda_k, u_k)}_{|_{(y^*,\lambda^*,u^*)}} = \begin{bmatrix} G_y & 0 & G_u \\ N_{yy} & G_y & n_{yu} \\ -B^{-1}N_{uy} & -B^{-1}G_u & I - B^{-1}N_{uu} \end{bmatrix}$$

erhalten wir eine Konvergenzaussage, wenn der Spektralradius $\hat{\rho}$ von J^* kleiner als 1 ist. Dadurch würden wir wie bei G die Existenz und Eindeutigkeit eines Fixpunktes erhalten. In Hamdi et al. [6] wird beschrieben, dass es leider noch nicht gelungen ist, die Eigenwerte von J^* hinreichend durch Bedingungen an den Präkonditionierer B zu beschränken, wenn man das Problem P betrachtet. Deshalb wird in Hamdi et al. [6] ein anderer Weg vorgeschlagen, so dass man hinreichende Bedingungen für B erhält.

3.2. Die erweiterte Lagrangefunktion

Wir definieren die erweiterte Lagrangefunktion wie in Hamdi et al. [6]

$$L^a(y, \lambda, u) = \frac{\alpha}{2} \|G(y, u) - y\|^2 + \frac{\beta}{2} \|N_y(y, \lambda, u) - \lambda\|^2 + N(y, \lambda, u) - \lambda^T y .$$

Für $\alpha = 0$ und $\beta = 0$ ist die erweiterte Lagrangefunktion gleich der normalen Lagrangefunktion. In dem Paper wird nun gezeigt, dass L^a eine exakte Straffunktion ist.

Straffunktionen sind Funktionen, die in der Optimierung genutzt werden um Nebenbedingungen in einem Optimierungsproblem nicht gesondert behandeln zu müssen. Man kann durch die Straffunktion die Nebenbedingung direkt in die zu optimierende Funktion einbringen. Das Exakt bezieht sich darauf, dass jeder Extremalpunkt von L^a unter bestimmten Bedingungen auch ein Extremalpunkt der selben Art von L ist.

In den folgenden Sätzen aus Hamdi et al. [6] wird $\Delta G_y := I - G_y$ verwendet. Diese Matrix ist wegen der Bedingung $\|G_y\| < \rho$ invertierbar. Die Invertierbarkeit folgt aus dem Störungslemma in Werner [19].

Satz 3.4 (Übereinstimmungsbedingung)
Es existiert eine Eins-zu-Eins Übereinstimmung der stationären Punkte von L^a und den Nullstellen des Updateschritts s wenn gilt

$$\det(\alpha\beta\Delta G_y^T \Delta G_y - I - \beta N_{yy}) \neq 0 \ .$$

Für diese Bedingung ist es hinreichend, dass

$$\alpha\beta(1-\rho)^2 > 1 + \theta \tag{I}$$

mit $\theta = \|N_{yy}\|$ erfüllt ist. Wenn (I) *gilt, ist die Hessematrix von L^a positiv definit.*

Da eine Nullstelle von s ein stationärer Punkt von L ist, folgt aus dem Satz, dass unter der Bedingung (I) die stationären Punkte von L^a den stationären Punkten von L entsprechen. Durch den Satz ergibt sich zusätzlich, dass die stationären Punkte von L^a Minima sind. Als nächstes wird eine Bedingung für den Abstieg hergeleitet.

Satz 3.5 (Abstiegsbedingung)
Mit $\Delta\bar{G}_y = \frac{1}{2}\left(\Delta G_y + \Delta G_y^T\right)$ bekommen wir für den Updateschritt $s(y, \lambda, u)$ einen Abstieg in L^a für alle großen positiven B, wenn die Bedingung

$$\alpha\beta\Delta\bar{G}_y > \left(I + \frac{\beta}{2}N_{yy}\right)\left(\Delta\bar{G}_y\right)^{-1}\left(I + \frac{\beta}{2}N_{yy}\right)$$

erfüllt ist. Dafür ist es hinreichend wenn

$$\sqrt{\alpha\beta}(1-\rho) > 1 + \frac{\beta}{2}\theta \tag{II}$$

gilt. Die Ungleichung (II) *ist etwas stärker als die Ungleichung* (I).

Wenn die Bedingung (II) erfüllt ist, bekommen wir einen Abstieg in der erweiterten Lagrangefunktion L^a. Der Schritt s führt somit zu einem Abstiegsverfahren in der Funktion L^a. Für einen stationären Punkt ($s = 0$) ergibt sich, dass L^a an dieser Stelle ein Minimum hat. Dadurch ergeben sich für das One Shot Verfahren zwei Möglichkeiten. Zum einen kann es gegen $-\infty$ divergieren oder es konvergiert gegen ein Minimum.

In Satz 3.5 wird immer noch ein B vorausgesetzt, das groß genug ist. Darauf wollen wir im nächsten Abschnitt eingehen.

3.3. Der Präkonditionierer B

Um einen Präkonditionierer herzuleiten, schauen wir uns wie in Hamdi et al. [7] das Minimierungsproblem

$$\min_{\Delta u} L^a(y + \Delta y, \lambda + \Delta\lambda, u + \Delta u) \tag{3.1}$$

an. Aus der Lösung Δu kann man durch die Gleichung $\Delta u = -B^{-1}N_u^T$ den Präkonditionierer B identifizieren. Um eine Lösung für das Problem (3.1) zu erhalten, betrachten wir die quadratische Approximation von L^a:

$$\min_{\Delta u} s^T \nabla L^a(y, \lambda, u) + \frac{1}{2} s^T \nabla^2 L^a(y, \lambda, u) s. \tag{3.2}$$

s ist wie oben der Updateschritt. Die Minimierung findet nur für den Paramter Δu statt. Deshalb genügt es, wenn man sich nur die Komponenten anschaut in denen Δu vorkommt. Wir erhalten die Funktion

$$\begin{aligned} E(\Delta u) =& \Delta u^T \nabla_u L^a + \frac{1}{2}(\Delta y^T \nabla_{yu} L^a \Delta u + \Delta\lambda^T \nabla_{\lambda u} L^a \Delta u + \Delta u^T \nabla_{uu} L^a \Delta u \\ &+ \Delta u^T \nabla_{uy} L^a \Delta y + \Delta u^T \nabla_{u\lambda} L^a \Delta\lambda) \ . \end{aligned}$$

Die Hessematrix von L^a ist symmetrisch, damit lassen sich die gemischten Terme zusammenfassen

$$\begin{aligned} E(\Delta u) =& \Delta u^T \left(\nabla_u L^a + \nabla_{uy} L^a \Delta y + \nabla_{u\lambda} L^a \Delta\lambda\right) + \frac{1}{2} \Delta u^T \nabla_{uu} L^a \Delta u \\ \approx& \Delta u^T \nabla_u L^a (y + \Delta y, \lambda + \Delta\lambda, u) + \frac{1}{2} \Delta u^T \nabla_{uu} L^a \Delta u \ . \end{aligned}$$

Die Terme $\nabla_{uy} L^a \Delta y$ und $\nabla_{u\lambda} L^a \Delta\lambda$ können als die Entwicklung von $\nabla_u L^a$ gesehen werden, wodurch sich die Approximation ergibt.

Die Ableitungen von $E(\Delta u)$ lauten

$$\begin{aligned} E'(\Delta u) &= \nabla_u L^a(y + \Delta y, \lambda + \Delta\lambda, u) + \nabla_{uu} L^a \Delta u \\ E''(\Delta u) &= \nabla_{uu} L^a. \end{aligned}$$

Für das Problem (3.2) existiert ein Minimum, wenn $\nabla_{uu} L^a$ positiv definit ist, dann existiert auch $(\nabla_{uu} L^a)^{-1}$ und wir erhalten für das Designupdate

$$\Delta u = -(\nabla_{uu} L^a)^{-1} \nabla_u L^a.$$

In einem stationärem Punkt von L^a gilt, dass $\nabla_u L^a$ gleich N_u^T ist. Damit ergibt sich die Gleichung

$$\Delta u = -(\nabla_{uu} L^a)^{-1} N_u(y, \lambda, u)^T. \tag{3.3}$$

Den Präkonditionierer B können wir nun mit $\nabla_{uu} L^a$ identifizieren. Wir wissen aber noch nicht, ob diese Wahl „groß genug“ ist. Eine Bedingung dafür wollen wir nun nach Hamdi et al. [7] herleiten.

Für die Herleitung nehmen wir an, dass B eine symmetrische positiv definite Matrix ist. Der Gradient von L^a hat die Form

$$\nabla L^a(y,\lambda,u) = \begin{pmatrix} \alpha(G_y - I)^T(G-y) + \beta N_{yy}(N_y - \lambda) + N_y - \lambda \\ \beta(N_{y\lambda} - I)(N_y - \lambda) + N_\lambda - y \\ \alpha G_u^T(G-y) + \beta N_{yu}^T(N_y - \lambda) + N_u \end{pmatrix}.$$

Mit $N_{y\lambda} = G_y$, $N_\lambda = G$ und $\Delta G_y = I - G_y$ erhalten wir eine einfachere Darstellung als Matrix Vektorprodukt

$$\nabla L^a(y,\lambda,u) = -Ms(y,\lambda,u), \text{ mit } M = \begin{pmatrix} \alpha\Delta G_y^T & -I - \beta N_{yy} & 0 \\ -I & \beta\Delta G_y^T & 0 \\ -\alpha G_u^T & -\beta N_{yu}^T & B \end{pmatrix}.$$

Der Vektor s ist der Updateschritt. Die symmetrische Matrix M_s ist durch $M_s = \frac{1}{2}(M^T + M)$ gegeben und hat die Form

$$M_S = \begin{pmatrix} \alpha\Delta\bar{G}_y & -I - \frac{\beta}{2}N_{yy} & -\frac{\alpha}{2}G_u \\ -I - \frac{\beta}{2}N_{yy} & \beta\Delta\bar{G}_y & -\frac{\beta}{2}N_{yu} \\ -\frac{\alpha}{2}G_u^T & -\frac{\beta}{2}N_{yu}^T & B \end{pmatrix}.$$

$\Delta\bar{G}_y$ ist wie in Satz 3.5 definiert. Durch M_S können wir eine neue Beziehung zu ∇L^a herstellen, denn wir haben

$$-s^T M_S s = -\frac{1}{2}s^T M^T s - \frac{1}{2}s^T M s = \frac{1}{2}(\nabla L^a)^T s + \frac{1}{2}s^T \nabla L^a = s^T \nabla L^a \tag{3.4}$$

wegen der Symmetrie des Skalarproduktes.

Da B symmetrisch und positiv definit ist, können wir für B mit $B^{\frac{1}{2}}$ die Cholesky Zerlegung nach Werner [19] beschreiben. $B^{\frac{1}{2}}$ erfüllt die Gleichung $B = B^{\frac{T}{2}}B^{\frac{1}{2}}$.

$B^{\frac{1}{2}}$ verwenden wir um u zu skalieren. Wir definieren $\tilde{u} := B^{\frac{1}{2}}u$. Der Updateschritt s wird so zu dem Updateschritt $\tilde{s}$ indem wir die dritte Komponente $\Delta u = -B^{-1}N_u^T$ ersetzen durch $\Delta\tilde{u} = B^{\frac{1}{2}}\Delta u = -B^{-\frac{1}{2}}N_u^T$. Für s ergibt sich

$$s = \operatorname{diag}(I, I, B^{-\frac{1}{2}})\tilde{s}.$$

Daraus folgt für $s^T M_S s$

$$s^T M_S s = \tilde{s}^T \operatorname{diag}(I, I, B^{-\frac{T}{2}}) M_S \operatorname{diag}(I, I, B^{-\frac{1}{2}})\tilde{s} = \tilde{s}^T \tilde{M}_S \tilde{s}$$

mit

$$\tilde{M}_S = \begin{pmatrix} \alpha\Delta\bar{G}_y & -I - \frac{\beta}{2}N_{yy} & -\frac{\alpha}{2}G_{\tilde{u}} \\ -I - \frac{\beta}{2}N_{yy} & \beta\Delta\bar{G}_y & -\frac{\beta}{2}N_{y\tilde{u}} \\ -\frac{\alpha}{2}G_{\tilde{u}}^T & -\frac{\beta}{2}N_{y\tilde{u}}^T & I \end{pmatrix}.$$

In $\tilde{M}_S$ verwenden wir die Abkürzungen $G_{\tilde{u}} = G_u B^{-\frac{1}{2}}$ und $N_{y\tilde{u}} = N_{yu}B^{-\frac{1}{2}}$. Mit diesen Definitionen wurde in Hamdi et al. [7] der folgende Satz bewiesen.

Satz 3.6
D_C ist durch die 3×3 Matrix

$$D_C = \begin{pmatrix} \alpha(1-\rho) & -1 - \frac{\beta}{2}\|N_{yy}\| & -\frac{\alpha}{2}\|G_{\tilde{u}}\| \\ -1 - \frac{\beta}{2}\|N_{yy}\| & \beta(1-\rho) & -\frac{\beta}{2}\|N_{y\tilde{u}}\| \\ -\frac{\alpha}{2}\|G_{\tilde{u}}\| & -\frac{\beta}{2}\|N_{y\tilde{u}}\| & 1 \end{pmatrix}$$

definiert. Dann gilt für alle $v_1, v_2 \in \mathbb{R}^m$ und für alle $v_3 \in \mathbb{R}^n$

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}^T \tilde{M}_S \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \geq \begin{bmatrix} \|v_1\| \\ \|v_2\| \\ \|v_3\| \end{bmatrix}^T D_C \begin{bmatrix} \|v_1\| \\ \|v_2\| \\ \|v_3\| \end{bmatrix}$$

Die positive Definitheit von D_C wird in folgendem Satz beschrieben.

Satz 3.7
Wenn die Ungleichung

$$\left(\sqrt{\frac{\alpha}{2}}\|G_{\tilde{u}}\| + \sqrt{\frac{\beta}{2}}\|N_{y\tilde{u}}\|\right)^2 \leq (1-\rho) - \frac{(1+\frac{\theta}{2}\beta)^2}{\alpha\beta(1-\rho)} =: \sigma$$

gilt, dann ist D_C eine positiv definite Matrix.

Jetzt können wir eine Bedingung für B herleiten. Die linke Seite der Ungleichung in Satz 3.7 schätzen wir nach oben ab

$$\begin{aligned} \frac{1}{2}\left(\sqrt{\alpha}\|G_{\tilde{u}}\| + \sqrt{\beta}\|N_{y\tilde{u}}\|\right) &\leq \max\left\{\sqrt{\alpha}\|G_{\tilde{u}}\|, \sqrt{\beta}\|N_{y\tilde{u}}\|\right\} \\ &\leq \left\|\begin{matrix}\sqrt{\alpha}G_{\tilde{u}} \\ \sqrt{\beta}N_{y\tilde{u}}\end{matrix}\right\|_2 = \left\|\begin{pmatrix}\sqrt{\alpha}G_u \\ \sqrt{\beta}N_{yu}\end{pmatrix} B^{-\frac{1}{2}}\right\|_2 \end{aligned}$$

Für die Matrix $\begin{pmatrix}\sqrt{\alpha}G_u & \sqrt{\beta}N_{yu}\end{pmatrix}$ benötigen wir die QR Zerlegung nach Werner [19].

$$\begin{pmatrix}\sqrt{\alpha}G_u \\ \sqrt{\beta}N_{yu}\end{pmatrix} = QR$$

mit $Q \in \mathbb{R}^{(n+m)\times n}$ und der oberen Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$. Für Q haben wir die Identität $Q^TQ = I$. Wenn wir QR nun in der Spektralnorm verwenden, bekommen wir

$$\begin{aligned} \left\|\begin{pmatrix}\sqrt{\alpha}G_u \\ \sqrt{\beta}N_{yu}\end{pmatrix} B^{-\frac{1}{2}}\right\|_2^2 &= \left\|QRB^{-\frac{1}{2}}\right\|_2^2 = \sqrt{\lambda_{\max}(B^{-\frac{T}{2}}R^TQ^TQRB^{-\frac{1}{2}})} \\ &= \sqrt{\lambda_{\max}(B^{-\frac{T}{2}}R^TRB^{-\frac{1}{2}})} = \left\|RB^{-\frac{1}{2}}\right\|_2^2 . \end{aligned}$$

Mit dem Satz A.1 erhalten wir nun, indem wir $A = RB^{-\frac{1}{2}}$ setzten, die Gleichung

$$\left\|RB^{-\frac{1}{2}}\right\|_2^2 = \left\|RB^{-1}R^T\right\|_2 \ .$$

Wir wollen diese Gleichung nach oben durch σ beschränken um die Bedingung aus Satz 3.7 zu erfüllen.

$$\begin{aligned} & \left\|RB^{-\frac{1}{2}}\right\|_2^2 = \left\|RB^{-1}R^T\right\|_2 \leq \|\sigma I\|_2 = \sigma \\ \Leftrightarrow \quad & RB^{-1}R^T \preceq \sigma I \\ \Leftrightarrow \quad & B \succeq \frac{1}{\sigma} R^T R = \frac{1}{\sigma} R^T Q^T Q R \ . \end{aligned}$$

Damit ergibt sich ein Ausdruck für B_0 in $\mathbb{R}^{n\times n}$ mit

$$B \succeq B_0 := \frac{1}{\sigma}(\alpha G_u^T G_u + \beta N_{yu}^T N_{yu}) \ .$$

Die Parameter α und β müssen für diese Gleichung bestimmt werden. Mit der Definition von B_0 wurde in Hamdi et al. [7] der folgende Satz gezeigt.

Satz 3.8
Es sei $\|G_u\|_2^2 \neq 0$ *dann wird die Funktion*

$$\phi(\alpha,\beta) = \frac{\|G_u\|_2^2\,\alpha + \|N_{yu}\|_2^2\,\beta}{1-\rho-\frac{(1+\frac{\theta}{2})^2}{\alpha\beta(1-\rho)}} \geq \|B_0\|_2$$

durch

$$\beta = \frac{3}{\sqrt{\theta^2 + 3\frac{\|N_{yu}\|_2^2}{\|G_u\|_2^2}(1-\rho)^2} + \frac{\theta}{2}} \ \textit{und}\ \alpha = \frac{\|N_{yu}\|_2^2\,\beta(1+\frac{\theta}{2}\beta)}{\|G_u\|_2^2\,(1-\frac{\theta}{2}\beta)}$$

minimiert. Diese Wahl von β *und* α *erfüllt die Abstiegsbedingung* (II).

Das α und β aus Satz 3.8 minimieren die Bedingung für B_0 und damit erhalten wir für ein bestimmtes B ein Minimum. Dieses minimale B resultiert in ein möglichst großes B^{-1}. Der Updateschritt Δu wird somit auch möglichst groß.

In Gleichung (3.3) haben wir B als $\nabla_{uu}L^a$ identifiziert. Für $\nabla_{uu}L^a$ haben wir angenommen, dass es positiv definit ist. Die Formel für $\nabla_{uu}L^a$ lautet

$$\nabla_{uu}L^a = \alpha G_u^T G_u + \beta N_{yu}^T N_{yu} + N_{uu}.$$

Angenommen N_{uu} ist positiv definit, dann erhalten wir

$$\nabla_{uu}L^a \succeq \alpha G_u^T G_u + \beta N_{yu}^T N_{yu}$$

und somit

$$\frac{1}{\sigma}\nabla_{uu}L^a \succeq \frac{1}{\sigma}(\alpha G_u^T G_u + \beta N_{yu}^T N_{yu}) \tag{3.5}$$

Damit können wir den Präkonditionierer festlegen

Definition 3.9 (Der Präkonditionierer)
B definiert durch

$$B := \frac{1}{\sigma}\nabla_{uu}L^a \tag{3.6}$$

ist nach Satz 3.7 und 3.8 ein Präkonditionierer für das Problem P, der einen Abstieg für den Schritt s garantiert.

Zur Verdeutlichung der Definition schauen wir uns nocheinmal die Gleichung (3.4) an. Mit dem definierten B wissen wir, dass die Matrix M_S positiv definit ist. Daraus folgt, dass $s^T\nabla L^a$ kleiner als 0 ist. Durch die Tailorentwicklung von L^a in Richtung s

$$L^a(y_{k+1}, \lambda_{k+1}, u_{k+1}) \approx L^a(y_k, \lambda_k, u_k) + s^T\nabla L^a$$

bekommen wir die Ungleichung

$$L^a(y_{k+1}, \lambda_{k+1}, u_{k+1}) < L^a(y_k, \lambda_k, u_k) \ .$$

Aus dieser Ungleichung sehen wir, dass wir für das gewählte B einen Abstieg in L^a erhalten.

3.4. Berechnung des Präkonditionierers

Da die Hessematrix von L^a sehr aufwändig zu berechnen ist, wollen wir sie durch ein BFGS Update wie in Gauger et al. [15] annähren. Die Annahme für das BFGS Update lautet

$$B\Delta u = \nabla_{uu}L^a(y, \lambda, u)\Delta u \approx \nabla_u L^a(y, \lambda, u + \Delta u) - \nabla L^a(y, \lambda, u)$$

Der Ansatz ist, dass wir eine Matrix $H^k \in \mathbb{R}^{n\times n}$ definieren, die die Gleichung

$$H_{k+1}R_k = \Delta u_k \text{ mit } R_k := \nabla_u L^a(y_k, \lambda_k, u_k + \Delta u_k) - \nabla_u L^a(y_k, \lambda_k, u_k).$$

erfüllen muss.

Nach Nocedal et al. [13] erhalten wir durch

$$H_{k+1} = (I - r_k\Delta u_k R_k^T)H_k(I - r_k R_k \Delta u_k^T) + r_k\Delta u_k\Delta u_k^T \text{ mit } r_k = \frac{1}{R_k^T\Delta u_k} \tag{3.7}$$

eine Annährung für B^{-1}. Die Berechnung von R_k enthält $\nabla_u L^a$. Der Gradient hat die Form

$$\nabla_u L^a = \alpha\Delta y^T G_u + \beta\Delta\lambda^T N_{yu} + N_u \ .$$

Die Ableitungen in $\nabla_u L^a$ werden mit AD berechnet. $\Delta y^T G_u$ kann im Rückwärtsmodus berechnet werden, da wir sofort das Produkt $\Delta y^T G_u$ berechnen können. Die zweite Ableitung $\Delta\lambda^T N_{yu}$ sollte im Rückwärts-Vorwärtsmodus aufgerufen werden. Damit können wir, wie für $\Delta y^T G_u$, sofort die Verknüpfung der Hessematrix N_{yu} mit $\Delta\lambda$ berechnen. N_u wertet man am Besten mit dem Rückwärtsmodus aus, weil die Funktion N keine vektorwertige Funktion ist.

Die Faktoren α und β sind durch die Wahl von B gegeben.

ρ und θ sind die einzigen Unbekannten in den Gleichungen für α und β . Für die Abschätzung von ρ und θ schauen wir uns wie in Hamdi et al. [6] die Änderung in y und λ an. Die Änderung in u wird dort vernachlässigt.

ρ beschreibt die Konvergenzrate des iterativen Verfahrens G. Daher haben wir für ρ die Ungleichung

$$\|G(y_k, u) - G(y_{k-1}, u)\| \leq \rho \|y_k - y_{k-1}\| \ .$$

Wir können diese Ungleichung nach ρ umstellen und durch Ausdrücke Δy ersetzen, denn in unserem Update ist $y_k = G(y_{k-1}, u_{k-1})$. Mit einem Startwert ρ_0 ist das Update von ρ_k mit $\tau \in (0, 1)$ durch

$$\rho_{k+1} = \max\{\frac{\|\Delta y_k\|}{\|\Delta y_{k-1}\|}, \tau\rho_k\}$$

definiert.

Die Approximation von θ erhalten wir aus den ersten beiden Komponenten für das Update s. Die Komponenten sind definiert durch

$$\begin{aligned}\begin{bmatrix}\Delta y_{k+1}\\ \Delta\lambda_{k+1}\end{bmatrix} &= \begin{bmatrix}G(y_k, u_k) - y_k\\ N_y(y_k, \lambda_k, u_k)^T - \lambda_k\end{bmatrix}\\ &= \begin{bmatrix}G(y_{k-1} + \Delta y_k, u_k) - G(y_{k-1}, u_{k-1})\\ N_y(y_{k-1} + \Delta y_k, \Delta\lambda_{k-1} + \Delta\lambda_k, u_k)^T - N_y(y_{k-1}, \lambda_{k-1}, u_{k-1})^T\end{bmatrix} \ .\end{aligned}$$

Wir können nun in die Differenzen eine Approximation mit dem Gradienten einsetzen. Es folgt

$$\begin{bmatrix}\Delta y_{k+1}\\ \Delta\lambda_{k+1}\end{bmatrix} \approx \begin{bmatrix}G_y(y_k, \lambda_k, u_k) & 0\\ N_{yy}(y_k, \lambda_k, u_k) & G_y(y_k, \lambda_k, u_k)^T\end{bmatrix} \begin{bmatrix}\Delta y_k\\ \Delta\lambda_k\end{bmatrix} \ .$$

Diesen Vektor multiplizieren wir nun mit $(\Delta\lambda_k, \Delta y_k)^T$. Es entsteht

$$\begin{aligned}\begin{bmatrix}\Delta\lambda_k\\ -\Delta y_k\end{bmatrix}^T \begin{bmatrix}\Delta y_{k+1}\\ \Delta\lambda_{k+1}\end{bmatrix} =& \Delta\lambda_k^T\Delta y_{k+1} - \Delta y_k^T\Delta\lambda_{k+1}\\ \approx& \Delta\lambda_k^T G_y\Delta y_k + \Delta y_k^T N_{yy}\Delta y_k - \Delta y_k^T G_y^T\Delta\lambda_k \ .\end{aligned}$$

Aufgrund der Symmetrie des Skalarproduktes erhalten wir die Ungleichung

$$\Delta\lambda_k^T\Delta y_{k+1} - \Delta y_k^T\Delta\lambda_{k+1} \approx \Delta y_k^T N_{yy}\Delta y_k \ .$$

Diese Ungleichung kann man nun mit der Cauchy-Schwarzen Ungleichung abschätzen

$$|\Delta\lambda_k^T\Delta y_{k+1} - \Delta y_k^T\Delta\lambda_{k+1}| \approx |\Delta y_k^T N_{yy}\Delta y_k| \leq \|\Delta y_k\|_2 \|N_{yy}\Delta y_k\| \leq \|N_{yy}\| \|\Delta y_k\|_2^2 .$$

Es sei $\tau \in (0,1)$ wie oben gewählt und θ_0 ein Startwert. Dann ist

$$\theta_{k+1} = \max\{\frac{|\Delta\lambda_k^T \Delta y_{k+1} - \Delta y_k^T \Delta\lambda_{k+1}|}{\|\Delta y_k\|_2^2}, \tau\theta_k\}$$

das Update für θ.

Bisher wurde noch keine Approximation von $\|G_u\|_2$ und $\|N_{yu}\|_2$ vorgeschlagen, die wir zur Berechnung von α und β benötigen. Deshalb schlagen wir eine Approximation auf Basis der induzierten Matrixnorm einer Vektornorm vor. Die induzierte 2-Norm einer Matrix A ist definiert durch

$$\|A\|_2 = \sup_{v \neq 0} \frac{\|Av\|_2}{\|v\|_2} \ .$$

Für diese Gleichung gilt die Ungleichung

$$\frac{\|Au\|}{\|u\|} \leq \sup_{v \neq 0} \frac{\|Av\|}{\|v\|} \qquad \text{für alle } u.$$

Bei der Berechnung des Präkonditionierers B berechnen wir die Produkte $\Delta y^T G_u$ und $\Delta\lambda^T N_{yu}$, wir können also eine Update für $\mu = \|G_u\|_2$ und $\nu = \|N_{yu}\|_2$ definieren durch

$$\mu_{k+1} = \max\left\{\frac{\left\|\Delta y_k^T G_u\right\|_2}{\|\Delta y_k\|_2}, \tau\mu_k\right\},$$

$$\nu_{k+1} = \max\left\{\frac{\left\|\Delta\lambda_k^T N_{yu}\right\|_2}{\|\Delta\lambda_k\|_2}, \tau\nu_k\right\} \ .$$

Die Updates μ_k und ν_k unterschätzen die exakten Werte μ und ν. μ und ν werden in der Berechnung von α und β als Bruch $\frac{\nu}{\mu}$ verwendet. Beide Werte werden unterschätzt, dadurch können wir ohne weitere Voraussetzungen keine Aussage darüber treffen ob wir den Bruch über- oder unterschätzen. Weiterhin wird der Bruch im Zähler von α und im Nenner von β verwendet. Damit können wir nicht von vornherein sagen ob eine Über- oder Unterschätzung des Bruchs sinnvoller ist. Bei der Verwendung der Updates μ_k und ν_k ist deshalb Vorsicht geboten und eine genauere Analyse für das zu lösende Problem erforderlich.

Damit haben wir sämtliche Gleichungen und Definition zusammen, um den Präkonditionierer $B = \frac{1}{\sigma}\nabla_{uu}L^a$ zu berechnen. In Kapitel 6 werden wir an diese Stelle zurückkehren und die Berechnung von B in dem Padge Code beschreiben. Dort findet sich auch eine Übersicht über alle Formeln, die wir für den Präkonditionierer benötigen.

4. Differentiation von Padge

Der Padge Code ist uns im Rahmen des BMBF Projekts DGHPOPT vom DLR Braunschweig zur Verfügung gestellt worden. Der Code ist für die Lösung kompressibler Strömungen geschrieben, für Problemstellungen, die keine Trivialbeispiele sind, sondern in der Größenordnung von mehreren Millionen Freiheitsgraden liegen. Um die Probleme in einer vernünftigen Zeit lösen zu können, ist die Unterstützung für verteiltes Rechnen implementiert. Teile des Codes werden daher parallel ausgeführt. Deshalb hängt der Code von mehreren anderen Bibliotheken ab und kann nicht ohne diese ausgeführt werden.

Dieses Kapitel wird zuerst eine Übersicht über den Code geben. Dabei werden die Abhängigkeiten erklärt und es wird auf die Struktur eingegangen. Darauf basierend werden wir erklären, welchen Weg wir für die Ableitung des Codes gewählt haben und weshalb andere Varianten nicht in Frage kommen. Wir werden auch beschreiben, welche Probleme bei der Ableitung auftraten und wie man diese hätte verhindern können.

4.1. Struktur von Padge

Bei der Struktur wollen wir mit den Abhängigkeiten des Padge Codes beginnen, da diese maßgeblich die Struktur und den Aufbau des Codes beeinflussen.

- *NetCDF* (Network Common Data Form) [12] ist eine Bibliothek zum Laden und Speichern von Daten. Diese Daten werden maschinenunabhängig gespeichert. Jede so erzeugte Datei enthält alle Daten um sich selber zu beschreiben.

- *METIS* [10] stellt Funktionen für die Zerlegung von Graphen und Finite Elemente Gittern bereit.

- *OpenCascade* [14]ist ein Softwarepaket, das Methoden zum Einlesen und Bearbeiten von CAD Daten bereitstellt. Von dieser Bibliothek werden nicht alle Pakte verwendet, sondern nur 20 von insgesamt 81.

- *Sacado* [17] ist Teil des Trilinos Projekts. Das Tool dient zum automatischen Differenzieren von C++ Code. In Padge wird das Paket in vereinzelten Funktionen für die Berechnung der Ableitung verwendet.

- *deal.II* (Differential Equations Analysis Library) [3] ist eine Bibliothek für die Berechnungen auf Finiten Elementen. Sie enthält alle Funktionen um eine Finite-Element-Methode zu implementieren und zu lösen.

- *PETSc* (Portable, Extensible Toolkit for Scientific Computation) [16] ist eine umfangreiche Bibliothek für das Lösen linearer und nicht linearer Probleme. Das Paket definiert dafür seine eigenen Datenstrukturen. Die Löser benutzen diese Strukturen, um alle Berechnungen durchzuführen.

- *MPI* (Message Parsing Interface) [11] ist ein Toolset, um ein Programm auf mehreren Rechner parallel auszuführen. MPI übernimmt dabei die Kommunikation zwischen den verschiedenen Prozessen.

Alle diese Pakete werden direkt in Padge benutzt. Da deal.II die Grundlagen für Padge liefert, ist es nicht verwunderlich, dass PETSc und MPI zum größten Teil durch die Architektur von deal.II gefordert werden.

Bei jedem Code der numerische Probleme löst, kann man die Bestandteile in zwei Gruppen unterteilen. Die erste Gruppe setzt sich aus allen Klassen und Methoden zusammen, die für die Berechnung des numerischen Problems zuständig sind. Als zweite Gruppe können wir die restlichen Bestandteile zusammenfassen. Hierbei handelt es sich meist um Ein- und Ausgabe von Informationen sowie die Aufarbeitung der Dateien, die das zu lösenden Problem beschreiben.

Für Padge können wir diese Unterteilung schon für die Bibliotheken durchführen, von denen der Code abhängt.

In die zweite Kategorie fallen NetCDF, METIS und OpenCascade. Jede dieser Bibliotheken wird deshalb für die Ableitung keine Rolle spielen. Probleme sind rein theoretisch nur bei der Übergabe von Daten an die abgeleiteten Bestandteile zu erwarten.

Die erste Kategorie wird durch deal.II, PETSc, Sacado und MPI vertreten. Das heißt, dass wir uns für die Ableitung von Padge auch mit diesen Bibliotheken befassen müssen.

Intern ist Padge in keine Pakete oder verschiedene Namensbereiche (namespaces) eingeteilt. Die logische Aufteilung findet durch eine Ordnerstruktur statt. Diese Ordner beinhalten die Kategorien *assemble, base, discretization, error_ estimation, geometry, postprocess, refinment, solutions, solver* und *test_ routines.*

assemble umfasst alle Klassen, die die zu lösenden Probleme beschreiben. Die Klasse UserProblem (Benutzer Problem) ist das Kernstück des Codes. Hier werden die Parameter, die das Problem beschreiben, eingelesen. Basierend auf diesen Parametern wird das Programm initialisiert und die Berechnung wird gestartet.

Der Einstiegspunkt befindet sich in *base.* Dieser Ordner definiert hauptsächlich grundlegende Werte und Funktionen die von allen Klassen verwendet werden.

geometry, refinement, discretization behandeln hauptsächlich die Verwaltung und Erstellung der Gitterstruktur. Die restlichen Ordner *error_ estimation, postprocess, solution* und *solver* behandeln die Aufstellung der Matrizen, der Vektoren und das Lösen der Problemstellungen.

4.2. Das AD Tool DCO

DCO [2] ist ein Werkzeug zum Automatischen Differenzieren, das an der RWTH Aachen von der Gruppe STCE (*Software and Tools for Computational Engineering*) um Uwe Naumannn entwickelt wird.

DCO befindet sich zur Zeit noch in der Entwicklung und wurde uns vom STCE im Rahmen dieser Arbeit zur Verfügung gestellt. Die verschiedenen Bereiche von DCO decken den Vorwärtsmodus *dco::t1s::type*, den Vecktorvorwärtsmodus *dco::t1v::type*, den Rückwärtsmodus *dco::a1s::type* und den Rückwärtsvorwärtsmodus *dco::t2s_ a1s-::type* ab. Alle Modi können eine Abhängigkeitsanalyse während des Vorwärtslaufes durchführen. Der Vorteil der Abhängigkeitsanalyse ist, dass man nur die Berechnungen durchführt und die Werte speichert, die am Ende für die Berechnung der Ableitung gebraucht werden. Die Tapegröße wird so klein gehalten und ermöglich damit die Ableitung größerer Codeteile.

Ein weiterer Vorteil besteht in der Möglichkeit externe Funktionen in DCO einzubinden. Eine externe Funktion ist eine Codeblock, der nicht direkt von DCO differenziert wird, sondern wo der Benutzer eine differenzierte Variante bereitstellt. DCO verwendet dann die von dem Benutzer bereitgestellte Ableitung der externen Funktion für die Berechnung der vollständigen Ableitung.

4.3. Möglichkeiten für die Ableitung

In Kapitel 1 über das Automatische Differenzieren wurden zwei Wege beschrieben, wie man AD in einen Code einbringen kann. In einem templatisierten Code ist es möglich, den Basistypen zur Berechnung über das Templateargument zu bestimmen. Ansonsten kann man durch ein Makro oder einen Typedef den globalen Basistypen umschalten.

In Padge ist keine dieser beiden Varianten vorgesehen. Es wurde kein Basistyp durch einen Typedef festgelegt, es wurden keine Klassen in einer templatisierten Form bereitgestellt. Wir müssen uns also für einen der beiden Wege entscheiden und diesen dann im gesamten Code umsetzen.

Für die Variante der Templates spricht ihre bessere Eignung für einen effizienten Code. Wenn man einen Algorithmus schreibt, der nur von einigen Funktionen die Ableitung braucht, kann man so für jede Funktion einzeln entscheiden, wie man sie aufruft. Dadurch ergibt sich der Vorteil, dass man für jede Berechnung nur den Aufwand betreibt, den man für sein Ergebnis benötigt.

Mit der Definition des Basistypen durch einen Typedef hat man diese Freiheit nicht. Für jeden Teil des Codes, der etwas ausrechnet, wir automatisch die Ableitung aktiviert. Besonders im Vorwärtsmodus wird immer die Ableitung mit berechnet, auch wenn man sie für keine der aufgerufen Funktionen benötigt. Der Rückwärtsmodus erzeugt weniger unnötigen Aufwand. Die Abhängigkeitsanalyse in DCO reduziert den Aufwand nochmals um einen Faktor für den Vorwärts- und Rückwärtsmodus, jedoch

ist der Aufwand auch dann höher, als wenn man nur das gewünschte berechnet.

Für welche Methode man sich entscheidet, hängt noch von einem weiteren Faktor ab. Je nachdem, wie gut man den zu differenzierenden Code kennt, kann man besser beurteilen, welche Variante die bessere ist und wieviel Arbeit es machen wird, eine der zwei Varianten zu implementieren.

Der Padge Code ist für uns vollkommen unbekannt. Darunter fallen auch die Abhängigkeiten PETSc, Sacado und MPI. Nur im Umgang mit deal.II gibt es grundlegende Erfahrungen. Basierend auf dieser Unerfahrenheit, kann man kein Urteil über den Aufwand der beiden Möglichkeiten treffen. Auch eine Sondierung der einzelnen Klassen in die Kategorien, welche abgeleitet werden müssen und welche nicht, wird man so nicht treffen können.

Deshalb haben wir uns für eine Blackbox-Differentiation des gesamten Padge Codes und des deal.II Codes entschieden. Diese Differentiation werden wir nicht mit Templates durchführen, sonden mit einem Typdef. Die Gründe für diese Entscheidung sind die Folgenden. Für jede Klasse einzeln zu entscheiden, ob sie differenziert werden muss, ist ohne Kenntnis über die Strukturen des Programmes sehr schwierig. Diese Klasse dann in eine Templateklasse umzuwandeln, ist schon recht aufwändig und lässt sich nicht sehr gut automatisieren. Danach in dem gesamten Code nach Abhängigkeiten für diese Klasse zu suchen und jede Abhängigkeit so umzuschreiben, dass die neue Form der Klasse in korrekter Weise aufgerufen wird, ist nochmals mit sehr viel Aufwand verbunden. Man kann das Suchen nach Abhängigkeiten umgehen, indem für das Template der Klasse ein Standart-Wert angegeben wird. Aber solche Hilfskonstrukte sollte man vermeiden.

Tauscht man in jeder Klasse den Rechentyp durch den Typedef aus, bereitet man jede Klasse schon indirekt auf die Templatisierung vor. Die gewählte Variante ist deshalb eine Vorstufe der Aufarbeitung mit Templates. Wenn der Code mit dem Typedef läuft, ergeben sich auch neue Analysemöglichkeiten, die vorher nicht gegeben waren. Der Rechentyp muss nicht ein double oder Ähnliches sein. Es muss auch kein sogenannter aktiver Typ für AD sein. Der Rechentyp kann auch aus einer Analysestruktur bestehen, die Informationen über das Programm sammelt. Die Differentiation mittels des Typedef's ist daher keine vergebliche Arbeit, sondern eine Vorstufe für die bessere Differentiation mit Templates.

PETSc, Sacado und MPI finden oben keine Erwähnung, da wir sie aus der Differenzierung ausklammern möchten. Bei PETSc handelt es sich um einen Code, der um ein vielfaches größer und komplexer ist als der Padge Code. Diesen abzuleiten ist sicherlich möglich, aber an sich schon eine Mammutaufgabe. Nach dem ersten Verständnis wird PETSc nur dafür verwendet lineare Gleichungssysteme zu lösen. Diese Gleichungssysteme werden aus AD Sicht als Elementaroperation betrachtet. Diese Elementaroperation leitet man separat ab um, die Ableitung im Padge Code mit den Funktionen von PETSc zu implementieren.

MPI ist für die Parallelisierung der Berechnung zuständig. Da es noch keine fertige differenzierte Version von MPI gibt, müsste man MPI auch differenzieren. Jedoch

verhält es sich hier wie bei PETSc, dass diese Aufgabe sehr schwierig ist. Wir wissen auch nicht, ob der Padge Code überhaupt differenziert werden kann. Deshalb ist es sinnvoll in einem ersten Schritt, die Ableitungen für nur einen Prozess durchzuführen und erst danach für mehrere Prozesse.

Da Sacado selbst ein AD Werkzeug ist, muss man annehmen, dass Sacado in den betreffenden Komponenten durch DCO ausgetauscht wird. Wir haben uns aber dafür entschieden, Sacado mit DCO abzuleiten. In Kapitel 1.6 wird erklärt, dass diese Ableitung bei einer theoretischen Betrachtung ohne Probleme funktionieren sollte und beschrieben, dass ein abgeleitetes Programm, erneut durch den Vorwärts- oder Rückwärtsmodus ableitbar ist. Wenn diese erneute Ableitung durch ein anderes AD Tool stattfindet, macht das keinen Unterschied.

In den folgenden Abschnitten werden wir uns mit der Implementierung der Ableitung durch DCO beschäftigen. deal.II, Padge und Sacado werden so vollständig differenziert. Auf die Sonderbehandlungen für PETSc und andere Codebestandteile werden wir auch eingehen.

4.4. Die Differentiation von deal.II

deal.II besteht wie Padge aus kleineren Programmteilen, auf die wir nicht näher eingehen. In diesem Abschnitt besprechen wir nur Besonderheiten des Codes, die wir bei der Differentiation zu beachten hatten.

Begonnen wurde mit der Definition des Basistyps. deal.II wird mit den *autoconf* Tool konfiguriert. Dabei wird aus der Datei *config.h.in* die Datei *config.h* erstellt. Diese Konfigurationsdatei enthält alle grundlegenden Einstellungen für das Projekt. Wenn wir in *config.h.in* den Typedef für den Rechentypen einfügen, sollte er in allen Dateien des Projekts sichtbar sein. Ansonsten bindet man die Datei *config.h* ein. Die Definition sieht folgendermaßen aus:

```
 basetype DCO declaration
#include "../../../dco/include/dco.hpp"
#include "dco_add.h"

typedef dco::t1s::type BASE_TYPE;
typedef dco::t1s::type LONG_BASE_TYPE;
```

In dieser Typendeklaration ist *dco::t1s::type* durch einen beliebigen anderen Typen ersetzbar. *dco_add.h* enthält speziellen Funktionen, die nicht in DCO definiert wurden, die wir aber für deal.II benötigen. Dabei handelt es sich um Konvertierungen zu einem Integer. Da sich DCO noch in der Entwicklung befindet, sind noch keine unterschiedlichen Rechentypen möglich. Deshalb bilden wir die zwei Genauigkeiten auf dieselbe Präzision ab. Auf die Definition eines *SHORT_BASE_TYPE* wurde verzichtet. Unter den jetzigen Bedingungen entstehen zu viele doppelte Deklarationen.

Nachdem wir die Typendeklaration für den Basistypen eingefügt haben, wurde jedes

Vorkommen von *long double* durch *LONG_BASE_TYPE* ersetzt und *double* durch *BASE_TYPE*. Mit diesem Schritt beginnt nun das Erstellen des Projekts. Dafür sollte man vorher einen Schritt zurück gehen und in dem typedef die normalen Typen verwenden. Dadurch findet man alle Fehler, die durch die reine Ersetzung entstanden sind.

Beim Erstellen des Programmcodes mit den DCO Typen ergeben sich je nach Codestruktur andere Fehler. Der häufigste besteht in den expliziten Konstruktoren von Klassen. Die Zuweisungen

```
Matrix A = 0.0;
Matrix B(0.0);
```

werden nicht mehr funktionieren. Die Matrix benötigt für den Konstruktor einen BASE_TYPE. Im oberen Fall erhält sie aber einen *double* Wert. Solche Ausdrücke muss man durch

```
Matrix A = BASE_TYPE(0.0);
Matrix B(BASE_TYPE(0.0));
```

ersetzen.

deal.II spezifisch ist die Handhabung der Vektor- und Matrixklassen. Diese können durch einen Templateparamter einen beliebigen Rechentypen benutzen. Die Klassen wurde aber nicht im Header implementiert, wie es für Templateklassen üblich ist, sondern durch explizite Template-Instantierung in einer Sourcecodedatei. Normalerweise ist das kein Problem aber in deal.II werden auch die Verbindungen von verschiedenen Rechentypen deklariert. Die Spezialisierungen werden explizit in den Sourcecodedateien instanziiert. Zum Beispiel werden in der Datei *precondition_block_ez.cc* die folgenden Instanziierungen für die Methode *vmult* vorgenommen:

```
template void
  PreconditionBlockJacobi<SparseMatrixEZ<float>, float>
  ::vmult<BASE_TYPE> (Vector<BASE_TYPE> &, const Vector<BASE_TYPE> &)
      const;
template void
  PreconditionBlockJacobi<SparseMatrixEZ<BASE_TYPE>, float>
  ::vmult<float> (Vector<float> &, const Vector<float> &) const;
template void
  PreconditionBlockJacobi<SparseMatrixEZ<BASE_TYPE>, float>
  ::vmult<BASE_TYPE> (Vector<BASE_TYPE> &, const Vector<BASE_TYPE> &)
      const;
template void
  PreconditionBlockJacobi<SparseMatrixEZ<BASE_TYPE>, BASE_TYPE>
  ::vmult<float> (Vector<float> &, const Vector<float> &) const;
```

Man sieht hier die Beziehungen zwischen *float* und vorher *double*. Durch die Ersetzung von *double* haben wir nun das Problem, dass die Konvertierung unserer aktiven Typen zu float nicht existiert. Daher müssen solche Codebestandteile auskommentiert werden.

Die Spezialisierungen für verschiedene Rechentypen werden ansonsten durch folgende Dateienarten generiert:

```
for (S1, S2 : REAL_SCALARS)
  {
    ...
    template S1 Vector<S1>::DEAL_II_MEMBER_OP_TEMPLATE_INST
      operator*<S2>(const Vector<S2>&) const;
    template
      void Vector<S1>::reinit<S2>(const Vector<S2>&, const bool);
    template
      void Vector<S1>::equ<S2>(const S1, const Vector<S2>&);
    template
      void Vector<S1>::scale<S2>(const Vector<S2>&);
    ...
  }
```

REAL_SCALARS ist eine Liste mit verschiedenen Typen. Die *for* Schleife wird für jede Kombination aller, der in der Liste enthaltenen Typen, aufgerufen.

Als Beispiel sollen in der Liste *REAL_SCALARS* die zwei Werte *double* und *float* stehen. Dann wird der Textblock für die Kombinationen *(double, double)*, *(float, double)*, *(double, float)* und *(float, float)* erstellt.

Zur Zeit brauchen wir nur die Version, die die Elemente mit sich selbst verknüpft. Deshalb wurde das zweite Argument aus der *for* Schleife entfernt. Dadurch erhalten wir:

```
for (S : REAL_SCALARS)
  {
    template bool Vector<S>::DEAL_II_MEMBER_OP_TEMPLATE_INST
      operator==<S>(const Vector<S>&) const;
    ...
  }
```

Wie in jeder Portierung gibt es noch weitere kleinere Probleme. Meist sind diese Sachen sehr C++ spezifisch und lassen sich nur mit den Feinheiten von C++ [18] lösen.

Als Abschluss geben wir noch eine Statistik über die Codezeilen an.

Codezeilen + Kommentare	ohne Boost	mit Boost
Insgesamt	219235	955674
Verändert nach Ersetzung	5440 (2,5%)	5440 (0,5%)
Verändert nach erfolgreicher Compilierung	8719 (4,0%)	8793 (0,9%)

Die Werte wurden mit dem Tool *cloc* erstellt und dienen zur Anschauung des Aufwandes. Leerzeichen wurden ignoriert. Die Unterscheidung mit und ohne Boost wurde wegen des Umfangs der Boost Dateien vorgenommen. In Padge werden nicht alle Dateien der Boost-Bibliothek verwendet. Deshalb verwischt diese Bibliothek die Verhältnisse in dem Code. Für den Rest der Programmbestandteile gilt, dass die Umstellung auf fast alles einen Einfluss hat.

4.5. Grundlegende Änderungen in Padge

Mit dem Padge Code wurde nach demselben Prinzip wie bei deal.II verfahren. In der Datei *global_include.h* wurde wieder *BASE_TYPE* und *LONG_BASE_TYPE* definiert und *long double* sowie *double* wurden durch die entsprechenden Typen ersetzt. Danach begann die Compilierung des Programmcodes.

Hier trat sehr viel häufiger das Problem der expliziten Konstruktoren auf als bei deal.II. Durch die fehlende Templatisierung gab es keine Probleme mit Typenkonvertierungen. Dafür traten Probleme an den Stellen auf, an denen die durch deal.II bereitgestellten Konvertierungen genutzt wurden. Dort musste die Berechnung in einer geringeren Genauigkeit entfernt werden. Die Berechnung wurde dann in der normalen Genauigkeit ausgeführt.

Andere Probleme gab es hauptsächlich mit Sacado und PETSc, die wir in eigenen Absätzen behandeln. Weitere kleinere Probleme traten auf, diese sind aber meist trivial nachdem man sich mit den Feinheiten von C++ [18] beschäftigt hat. Deshalb werde wir auf diese nicht weiter eingehen.

4.6. Differenzieren von Sacado in Padge

Die Methoden, in denen Sacado benutzt wird, wurden so geschrieben, dass der Sacado Typ als ein Typedef oder Templateargument vorlag. Dadurch hätte man sehr einfach den Sacado Typen durch einen DCO Typen ersetzten können. DCO unterstützt noch keine Verschachtelungen von AD Typen.

Mit der Verschachtelung ist gemeint, dass wir in einer Methode einen AD Typen verwenden, um ein Ergebnis zu berechnen, das als double zurückgegeben wird. Wir wollen den Code jetzt so ändern, dass die Methode mit einem beliebigen Typen rechnet. Der Rückgabewert muss dann dem neuen Typen entsprechen. Als Schlussfolgerung ergibt sich, dass das AD Tool, das in der Methode verwendet wird, mit dem neuen Typen rechnen muss.

Dieses Ineinander-Einsetzen von AD Typen ist in DCO noch nicht möglich. Die einzige Lösung wäre, dass man für den Basistypen einen Ableitungsgrad wählt, der alle Möglichkeiten von Ableitungen beinhaltet. Der Nachteil dieser Lösungsmöglichkeit besteht allerdings darin, dass der Rechenaufwand für die gesamte Anwendung steigt und dass man mehr Sonderbehandlungen für die Ableitungen benötigt.

Sacado wurde so geschrieben, dass man verschiedene Sacado Typen ineinander einsetzen kann. Als Beispiel kann der normale Vorwärtsmodus von Sacado genutzt werden, um zweite Ableitungen zu berechnen. Normalerweise wird in dem Sacado Vorwärtstypen ein *double* als Templateparameter übergeben. Wenn wir anstelle des *double*'s den Sacado Vorwärtstypen übergeben, können wir zweite Ableitungen mit Sacado berechnen.

Intern arbeitet Sacado sehr viel mit diesen Strukturen und erlaubt so eine gute Anpassung an die Bedingungen des Benutzers. Bei der Verwendung eines DCO Typen als Templateparameter für die Sacado Typen bekommt man Fehlermeldungen des

```
#define SACADO_BUILTIN_SPECIALIZATION(t)                    \
  template <> struct ScalarType< t > {                      \
    typedef t type;                                         \
  };                                                        \
  template <> struct ValueType< t > {                       \
    typedef t type;                                         \
  };                                                        \
  template <> struct ScalarValueType< t > {                 \
    typedef t type;                                         \
  };                                                        \
  template <> struct IsADType< t > {                        \
    static const bool value = false;                        \
  };                                                        \
  template <> struct IsScalarType< t > {                    \
    static const bool value = true;                         \
  };                                                        \
  template <> struct Value< t > {                           \
    static const t& eval(const t& x) { return x; }          \
  };
```

Abbildung 4.1.: Sacado Makro für die Definition der Basistypen (Version 9.0.8)

Compilers, dass er Werte wie *Sacado::ScalarType<dco::t1s::type>::type* nicht finden kann.

In der Dokumentation von Sacado ist nicht direkt beschrieben, wie ein eigener Typ für die Berechnung in Sacado hinzugefügt wird. Ein Blick in die Doxygen Dokumentation der Klasse *Sacado::Fad::DFad* zeigt, dass diese Klasse *Sacado::ScalarType<t>* definiert. *ScalarType* wird in der Datei *Sacado_ Traits.hpp* beschrieben. Diese Datei beinhaltet das Makro aus der Abbildung 4.1. Mit diesem Makro können wir einen neuen Typen für die aktiven DCO Typen einfügen indem wir in der Datei *global_ include.h*

```
SACADO_BUILTIN_SPECIALIZATION(BASE_TYPE)
```

aufrufen. Die fehlenden Information für Sacado werden mit diesem Makro definiert und der Compiler hat für Sacado keinen Fehler mehr produziert.

4.7. Differenzieren von PETSc in Padge

Eingangs wurde erwähnt, dass PETSc wahrscheinlich nur für das Lösen linearer Gleichungssystem benötigt wird. Diese Vermutung hat sich beim Kompilieren des Codes nicht bestätigt. In Padge wurde die MPI Fähigkeit über PETSc hinzugefügt. In Teilen des Codes wird deshalb von einer Berechnung mit Basistypen auf eine Berechnung mit PETSc Vektoren umgestiegen und teilweise wird der Vektor- und Matrixtyp für die Berechnungen als Templateargument bzw. Typedef angelegt. Der Code kann so mit deal.II Vektoren oder PETSc Vektoren aufgerufen werden. Der Code enthält spezielle Weichen, die die Ausführung mit MPI und PETSc konfigurieren und handhaben.

```
            // Basic typedefs:
#if defined(USE_PETSC)
    typedef dealii::PETScWrappers::MPI::SparseMatrix_SEQBAIJ
        PETSc_SparseMatrix;
    typedef dealii::PETScWrappers::MPI::Vector PETSc_Vector;
#endif

    typedef SparseMatrix<MTYPE> DealII_SparseMatrix;
    typedef Vector<double> DealII_Vector;


            // Typedefs depending on compiler
            // options:

#if defined(USE_PETSC) && !defined(PETSC_USE_SINGLE)
    typedef PETSc_SparseMatrix  PSparseMatrix;
    typedef PETSc_Vector        PVector;
#endif

#if defined(USE_PETSC) && defined(PETSC_USE_SINGLE)
    typedef PETSc_SparseMatrix  PSparseMatrix;
    typedef DealII_Vector       PVector;
#endif

#if !defined(USE_PETSC)
    typedef DealII_SparseMatrix PSparseMatrix;
    typedef DealII_Vector       PVector;
#endif
```

Abbildung 4.2.: Definitionen der Matrizen und Vektoren für verschiedene Compileroptionen von Padge

Der PETSc Vektor ist nicht der in PETSc definierte Pseudotyp *Vec*, sondern eine Wrapperklasse aus deal.II. Diese Wrapperklasse stellt ein PETSc *Vec* Objekt als einen deal.II konformen Vektor bereit. Einem Code, der mit deal.II arbeitet, wird dadurch ermöglicht, MPI und PETSc zu verwenden. Dieselben Möglichkeiten werden auch für Matrizen bereitgestellt.

Durch die Entscheidung, PETSc nicht mit AD zu behandeln, konnten wir die PETSc Vektoren in der AD Version von Padge nicht mehr verwenden. In der Datei *vector_type.h* werden für verschiedene Compilereinstellungen die Definition für die Vektoren und Matrizen gesetzt. In ihrer Grundform enthält die Datei die in Abbildung 4.2 gezeigten Definitionen. Das P in *PSparseMatrix* und *PVector* steht nicht für PETSc, sondern für Parallel. Man kann anhand der Definitionen sehen, dass bei einer Erstellung von Padge ohne PETSc die normalen deal.II Varianten benutzt werden. In der Erstellung mit PETSc werden die deal.II Wrapper für PETSc verwendet.

Wir haben nun probiert Padge, ohne PETSc zu kompilieren. Dieser Test hat ergeben, dass einige Funktionalitäten in Padge nicht mehr verfügbar sind. Darunter fiel auch das Backward Euler Verfahren, das wir zum Lösen der nichtlinearen Navier

Stokes Gleichungen benutzen wollten. In diesem Verfahren wird auch der Löser für die entstehenden linearen Gleichungssysteme bereitgestellt.

Man konnte auch nicht die deal.II Vektoren und Matrizen in denCodepfaden für die PETSc Variante von Padge benutzen, da in diesen Funktionen der PETSc Wrapper fehlen.

Das Problem wurde mit dem Einsatz von zwei neuen Klassen für die Vektoren und Matrizen gelöst. Die Klasse des Vektors wurde von dem deal.II Vektor abgeleitet, die der Matrix von der deal.II Sparse Matrix. Die zusätzliche Funktionalität beschränkt sich auf die Verwaltung der MPI spezifischen Werte und das Füllen der Matrix. Dabei handelt es sich um das Kommunikationsobjekt für die verschiedenen MPI Prozesse und die lokale Größe des Vektors oder der Matrix. Die lokale Größe gibt an, wie viele Einträge des Objekts für den lokalen Prozess gespeichert werden. In unserem Fall ist die lokale Größe immer gleich der Dimension des Vektors oder der Matrix, da wir uns auf einen MPI Prozess beschränken.

Die Matrix Klasse hatte noch die Besonderheit, dass die Sparse-Struktur für die normalen deal.II Matrizen anders aufgebaut wird als für die PETSc Matrizen. Dieses besondere Verhalten, musste in der Implementierung berücksichtigt werden.

Die beiden neuen Klassen konnten ohne größere Schwierigkeiten in dem Padge Code verwendet werden. An einigen Stellen im Code wurden anstatt des *PVectors* oder der *PSparseMatrix* die PETSc Varianten direkt verwendet. An diesen Stellen musste die globale Definition eingefügt werden, dadurch waren auch diese Stellen kein Problem mehr.

Mit diesen neuen Vektoren und Matrizen konnte nun der gesamte Padge Code kompiliert werden. Die einzigen Ausnahmen waren Codebestandteile, die auf Routinen von PETSc zurückgreifen. In diesen Routinen werden die PETSc eigenen Vektoren und Matrizen verlangt. Die von uns neu erstellten Klassen funktionieren an diesen Stellen nicht. Man muss nun die Ableitung der PETSc Routinen selbst implementieren. Diese manuelle Implementierung wurde nur für den nichtlinearen Löser in Padge vorgenommen. An zwei anderen Stellen für einen generellen Löser und einen Matrixfreien wurden die PETSc Befehle ausdokumentiert und ein Fehler wird erzeugt, falls eine Konfiguration von Padge diese Methode aufruft.

Der nichtlineare Löser in Padge heißt *snes_newton*. Er greift sehr stark auf die Routinen des PETSc SNES Lösers zurück. Mathematisch beschreiben wir den Newton Schritt durch

$$\begin{aligned} d =& J(x_k)^{-1} R(x_k) \qquad \text{mit } d \in \mathbb{R}^n, R(x) \in \mathbb{R}^n, J(x) \in \mathbb{R}^{n \times n} \text{ und} \\ x_{k+1} =& x_k - d \qquad \text{mit } x_k \in \mathbb{R}^n. \end{aligned}$$

Für den PETSc SNES wird dieser Schritt in drei Methoden aufgeteilt. *formFunction* berechnet $R(x)$, *formJacobi* $J(x)$ und *lineSearch* führt den Schritt $x_{k+1} = x_k - d$ aus. Durch diese Strukturierung in PETSc haben wir auch ähnliche Funktionen in Padge.

In dem Kapitel über AD haben wir in der Abbildung 1.7 die Ableitung eines Newton

```
PetscErrorCode NewtonAlgorithm::formFunction(SNES /*snes*/, Vec x, Vec
    f, void* dummy) {
    SolverStatus<ProblemType, VECTOR>* status =
      reinterpret_cast<SolverStatus<ProblemType, VECTOR>*>(dummy);
      ...

              // compute residual:
    ProblemType& op = status->equation_operator;
    op.compute_residual(x);
            // copy contents back again:
    PetscErrorCode ierr;
    ierr = VecCopy(op.get_residual(), f);
    Assert (ierr == 0, ExcPETScError(ierr));

    return 0;
  }
```

Abbildung 4.3.: Berechnung des Residuums in Padge

Schritts für den Vorwärtsmodus aufgeführt. Die Funktionen f und F in der Abbildung 1.7 können wir mit dem Aufruf von *formFunction* und *formJacobi* identifizieren.

Für das Residuum R wird der Code in Abbildung 4.3 verwendet. Die Abbildung 4.4 zeigt die differenzierte Variante. Wir sehen, dass der Unterschied zwischen der normalen und der differenzierten Variante in den Methoden *petscVectorToDCOVector* und *dcoVectorToPetscVector* besteht. Für die Funktion *formJacobian* bekommen wir genau dieselbe Veränderung.

Damit sind die ersten beiden Zeilen in Abbildung 1.7 abgehandelt. Nachdem der SNES Löser diese beiden Methoden aufgerufen hat, löst er das Gleichungssystem. Das resultierende d aus der Abbildung wird der Funktion *lineSearch* übergeben. In dieser Funktion wird das Update in x ausgeführt. Der Code der normalen Version wird in Abbildung 4.5 etwas verkürzt dargestellt. In diesem Code müssen wir $\dot{d}$ berechnen und das Update in $\dot{x}$ durchführen. Die so entstandene differenzierte Variante wird in Abbildung 4.6 gezeigt.

Die differenzierte Version unterscheidet sich nur in den neuen Berechnungen von der normalen Version.

4.8. Abschluss

Durch die beschriebenen Schritte konnte der differenzierte Padge Code kompiliert werden. Die größten aufgetretenen Probleme wurden in vorausgehenden Abschnitten beschrieben.

Der nun für die Ableitungen aufbereitete Code muss noch getestet werden. Wie das gemacht wird und was zu beachten ist, schauen wir uns im nächsten Kapitel an.

Zum Abschluss dieselbe Statistik wie für den deal.II Code über die veränderten Codezeilen. Die Zahlen wurden mit dem Tool *cloc* erstellt. Leerzeichen wurden ignoriert.

```
PetscErrorCode NewtonAlgorithm::formFunction(SNES /*snes*/, Vec x, Vec
    f, void* dummy) {
    NewtonAlgorithm* algorithm =
        reinterpret_cast<NewtonAlgorithm*>(dummy);
    SolverStatus<ProblemType, VECTOR>* status = (&algorithm->status);
    ...

    algorithm->solutionVector = x;
    petscVectorToDCOVector(algorithm->solutionVector,
        algorithm->ad_solutionVector, algorithm->solution);

    // compute residual:
    ProblemType& op = status->equation_operator;
    op.compute_residual(algorithm->solution);

    dcoVectorToPetscVector(op.get_residual(),
        algorithm->residualVector, algorithm->ad_residualVector);

    // copy contents back again:
    PetscErrorCode ierr;
    ierr = VecCopy(algorithm->residualVector, f);
    Assert (ierr == 0, ExcPETScError(ierr));

    return 0;
  }
```

Abbildung 4.4.: Berechnung des Residuums im differenzierten Padge

```
PetscErrorCode NewtonAlgorithm::lineSearch(SNES snes,
                ...
  {
    ...
   double omega = 1.0;
    ...

    // x = x_k, y = d, w = x_k+1
    stepsize_estimation->slcontrol.perform_step(x, omega, y, w);
    ...
  }
```

Abbildung 4.5.: Durchführung des Newtonschrits in Padge

```
PetscErrorCode NewtonAlgorithm::lineSearch(SNES snes,
                                                    ...
  {
    ...
    // calculate the ad version of  the direction
    algorithm->ad_jacobiMatrix.vmult(v_ad, v);
    v_ad -= algorithm->ad_residualVector;

    algorithm->ierr = KSPSolve(algorithm->ksp, v_ad, v_ad);
    ...
    BASE_TYPE omega = BASE_TYPE(1.0);
    ...
    // x = x_k, y = d, w = x_k+1
    stepsize_estimation->slcontrol.perform_step(x, omega.v, y, w);
    ...

    // do the step for the the ad computation
    stepsize_estimation->slcontrol.perform_step(
        algorithm->ad_solutionVector, omega.v, y_ad, w_ad);
    algorithm->ad_solutionVector = w_ad;

    ..
  }
```

Abbildung 4.6.: Durchführung des Newtonschrits im differenzierten Padge

Codezeilen + Kommentare	Anzahl
Insgesamt	132586
Verändert nach Ersetzung	6962 (5,3%)
Verändert nach erfolgreicher Compilierung	8874 (6,7%)

5. Verifikation des differenzierten Codes

5.1. Möglichkeiten der Verifikation

Die Ableitung für eine reell analytisch gegebene Funktion lässt sich durch die Ableitungsvorschriften herleiten, wenn wir wissen, dass für die Funktion eine Ableitung existiert.

Für die Ableitung, die wir in einem Programm bilden, haben wir zwei Möglichkeiten zur Validierung. Falls das Programm nur für die Auswertung der Funktion zuständig ist, besteht die erste Möglichkeit darin, dass die durch AD berechnete Ableitung mit der analytischen verglichen wird.

Sobald das Programm die Funktion nicht mehr exakt ausrechnen kann, einen anderen Weg zur Berechnung beschreitet oder keine Funktion gegeben ist, die die Berechnung des Programmes beschreibt, kann man diesen Vergleich nicht mehr durchführen.

Bei der Lösung dieses Problems helfen uns die Finiten Differenzen weiter. In der Einleitung zu dem Kapitel über das Automatische Differenzieren haben wir kurz die Probleme der Finiten Differenzen dargestellt und dargestellt wieso man sie nicht verwendet. Da sie sich aber sehr einfach berechnen lassen, kann man sie als Vergleichsobjekt heranziehen.

Definition 1.1 beschreibt die Richtungsableitung. Eine Formel für die Finiten Differenzen bekommen wir, indem wir ein $h \in \mathbb{R}$ fixieren und eine Richtung $d \in \mathbb{R}^n$ mit $\|d\| = 1$ festlegen. Dadurch entsteht

$$f'(x_0)^T d \approx \frac{f(x_0 + hd) - f(x_0)}{h} \ . \tag{FD}$$

Wenn wir in diese Formel für d die Einheitsvektoren $e_1, ..., e_n$ einsetzen erhalten wir eine Approximation des Gradienten ∇f der Funktion. Wir nennen diese Approximation $\nabla_{FD} f$.

Für die Berechnung mit AD haben wir in den Definitionen 1.13 und 1.15 den Vorwärts- und Rückwärtsmodus beschrieben. Aus den jeweils folgenden Sätzen wissen wir, dass mit AD die exakte Ableitung der Funktion in Maschinengenauigkeit berechnet wird. Damit haben wir die zwei Formeln

$$\begin{aligned} \dot{y} &= \nabla f(x)\dot{x}, \\ \bar{x} &= \nabla f^T(x)\bar{y} \ . \end{aligned}$$

Durch eine geeignete Wahl von $\dot{x}$ und $\bar{y}$ bekommen wir mit diesen Gleichungen den Gradienten von f. Den Gradienten durch den Vorwärtsmodus nennen wir $\nabla_F f$ und den durch den Rückwärtsmodus $\nabla_R f$.

Für diese drei Gradienten muss nun die Gleichung

$$\nabla_F f(x) = \nabla_R f(x) \approx \nabla_{FD} f(x)$$

für ein $x \in \mathbb{R}^n$ erfüllt sein. Mit dieser Gleichung können wir die Ableitung, die wir mit AD berechnen mit den Ableitungen, die durch eine Finite Differenz berechnet werden, vergleichen.

Für das Programm, das wir ableiten, können wir annehmen, dass es hinreichend gut getestet ist. Es wird in seinen Berechnungen keine Fehler haben und somit sollten die Finiten Differenzen ohne Probleme berechenbar sein und eine gute Approximation der Ableitung im Bereich ihrer Möglichkeiten geben.

Damit haben wir eine Möglichkeit, die Ableitung des differenzierten Programmes zu testen. Doch welche Erwartungen kann man an das Ergebnis dieses Tests stellen?

Es sei $f : \mathbb{R}^n \to \mathbb{R}$ eine differenzierbare Funktion, dann ist der Gradient von f ein Vektor im $\mathbb{R}^n$. Die Normen $\|\nabla_F f(x) - \nabla_{FD} f(x)\|_2$ und $\|\nabla_R f(x) - \nabla_{FD} f(x)\|_2$ beschreiben den Fehler der Gradienten. $\nabla_F f(x)$ und $\nabla_R f(x)$ sollten sehr nahe bei dem exakten Gradienten liegen. $\nabla_{FD} f(x)$ liegt je nach gewähltem h näher oder weiter weg von dem exakten Gradienten. Für große h erwarten wir deshalb aufgrund des großen Abstandes eine schlechte Approximation des Gradienten. Für kleine h erwarten wir aufgrund von Auslöschung ebenfalls eine schlechte Approximation des Gradienten. Für eine Abbildung des Fehlers gegenüber der Schrittweite erwarten wir deshalb eine V-Kurve.

Wenn wir unsere durch AD berechneten Ableitungen mit den Finiten Differenzen vergleichen, sollten wir diese V-Kurve erhalten, wenn die Berechnungen keinen Fehler enthalten. In Abbildung 5.1 wurde für die Funktion

$$f_d(x_1, ..., x_n) = x_1^d \cdot x_2^d \cdot .... \cdot x_n^d$$

der Fehler der Finiten Differenzen gegen die exakte Ableitung für $d = 1, 2$ und 5 gezeichnet. Auf der x-Achse sind die Schrittgrößen für die Finiten Differenzen angegeben. Die y-Achse stellt den Fehler $\|\nabla f_d - \nabla_{FD} f_d\|$ dar. Für 2 und 5 kann man deutlich die V-Kurve sehen. In dem Fall $d = 1$ ist f eine lineare Funktion für die einzelnen Komponenten. Bei linearen Funktionen sind die Finiten Differenzen aufgrund der Definition der Finiten Differenzen sehr genau.

5.2. deal.II

Die Verifikation der Ableitungen von deal.II haben wir anhand des 12. Beispiels aus der deal.II Dokumentation durchgeführt. Das 12. Beispiel löst eine lineare Transportgleichung auf dem Einheitsquadrat. Die Definition des Problems lautet

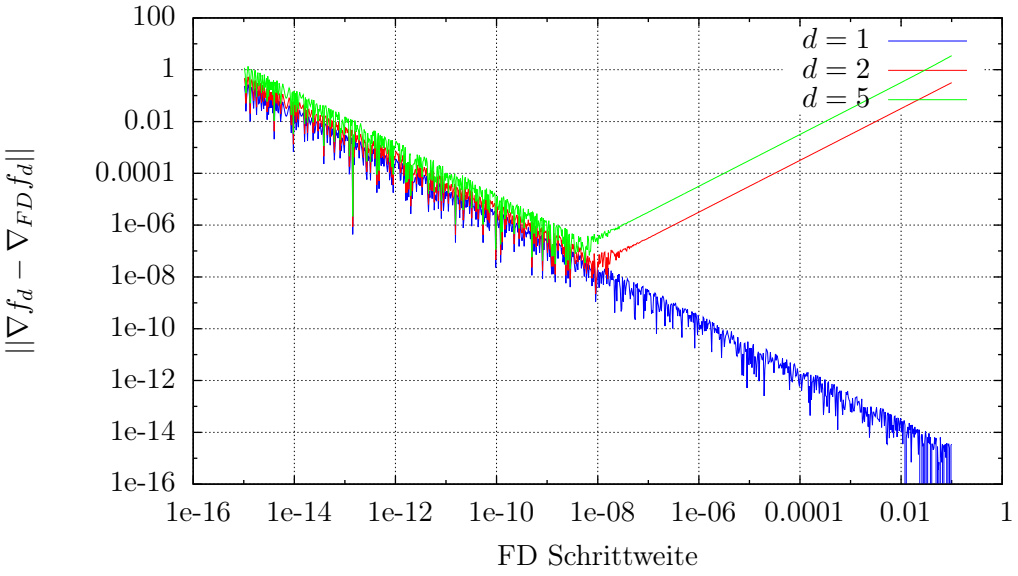

Abbildung 5.1.: Fehler der Finiten Differenzen für die Testfunktion $f_d = x_1^d \cdot x_2^d \cdot .... \cdot x_n^d$

$$\begin{aligned} \operatorname{div}(\beta u) &= 0 \qquad \text{auf } \Omega := [0,1]^2, \\ u &= g \qquad \text{auf } \Gamma_- \ . \end{aligned}$$

β beschreibt die Strömungsrichtung und ist definiert durch

$$\beta(x) := \frac{1}{|x|}(-x_2, x_1) \ .$$

Der Rand Γ_- ist durch die Randelemente bestimmt an denen wir eine Einströmung haben.

$$\begin{aligned} \Gamma_- :=& \{x \in \Gamma, \beta(x) \cdot n(x) < 0\} \\ g =& 1 \text{ für } x \in \Gamma_-^1 := [0, 0.5] \times \{0\} \\ g =& 0 \text{ für } x \in \Gamma_- \setminus \Gamma_-^1 \end{aligned}$$

In dem Beispiel wurden für die Berechnung der Finiten Differenzen und der Ableitung das gröbste Gitter gewählt. Als Ergebnis erhalten wir für das Transportproblem die Funktion $u : \Omega \to \mathbb{R}$, die durch die diskrete Lösung u_h dargestellt wird. Diese diskrete Lösung wird aus den Basisfunktionen der Finiten Elemente und den Gewichten λ berechnet.

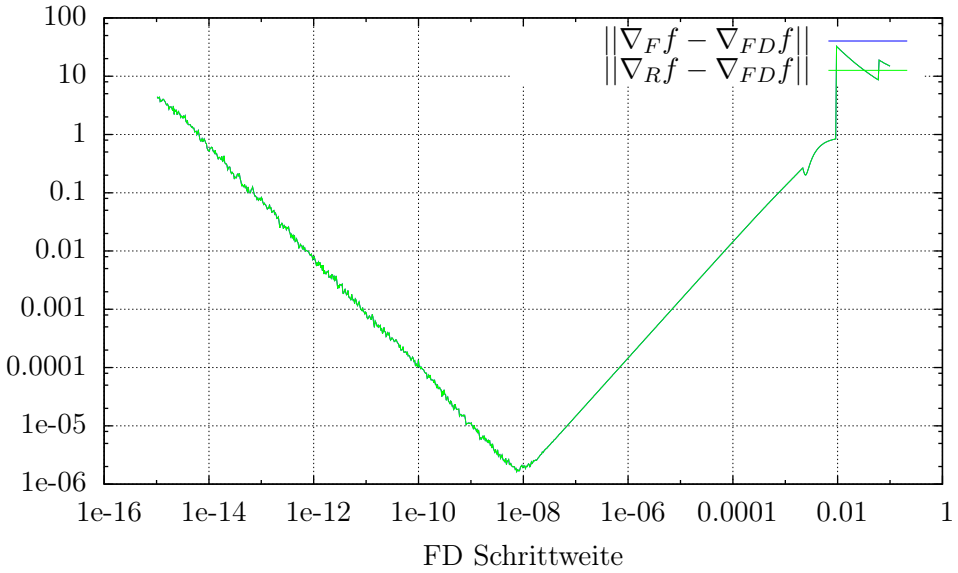

Abbildung 5.2.: Finiten Differenzen gegen AD Vorwärts und Rückwärts für deal.II. Die Kurven überlagern sich, zeigen aber die erwartete V-Kurve.

Die Parameter, nach denen wir ableiten wollen, sind die Gitterpunkte der Diskretisierung. Jeder Gitterpunkt liefert 2 Freiheitsgrade: einen für die x-Richtung, einen für die y-Richtung. Die Freiheitsgrade der Gitterpunkte bezeichnen wir mit dem Vektor $x \in \mathbb{R}^n$. Damit können wir für λ schreiben, dass es von x abhängt, da das Gitter in die Berechnung der Finiten Elemente eingeht. Die Definition der Testfunktion

$$f(x) = \|\lambda(x)\|_2$$

gibt uns eine reelle Funktion, für die wir die Finiten Differenzen mit dem Vorwärts- und Rückwärtsmodus von AD vergleichen können.

Die Finiten Differenzen wurden für jeden Freiheitsgrad in x an 100 Stellen ausgewertet. Die Schrittweite liegt im Bereich von $[10^{-15}, 10^{-1}]$.

Die Grafik 5.2 zeigt eine sehr schöne V-Kurve für den Rückwärts- sowie für den Vorwärtsmodus. Damit können wir sicher sein, dass bei der Differentiation des deal.II Codes keine groben Fehler entstanden sind.

AD sollte für $\nabla_R f$ und $\nabla_F f$ gleiche Werte liefern. Der Fehler für $\|\nabla_F f - \nabla_R f\|$ liegt bei 10^{-15}. Damit unterscheiden sich die Kurven kaum. Beide Modi funktionieren daher gleich gut.

5.3. Padge

Die Problemstellung in Padge besteht aus dem Gebiet $\Omega \subset \mathbb{R}^2$, für das die Ränder Γ_∞ den Fernfeldrand und Γ_ω das Flügelprofiel beschreiben. Für die Ränder gilt

$$\partial\Omega = \Gamma_\infty \dot\cup \Gamma_\omega \ .$$

Für das Gebiet Ω soll die Navier-Stokes-Gleichung (NV) erfüllt sein. Damit gilt die Gleichung

$$\mathrm{Nv}(y) = 0 \qquad \text{auf } \Omega \ .$$

y beschreibt den Vektor der konservativen Variablen.

Für die Strömung um den Flügel legen wir einen Anstellwinkel α fest. Der Anstellwinkel beschreibt, wie der Flügel im Verhältnis zum Luftstrom liegt. Dadurch können wir die Randbedingungen für das Gebiet Ω beschreiben.

Auf dem Fernfeldrand Γ_∞ setzen wir an den Einströmungsstellen die freie Anströmung, das heißt

$$\begin{aligned} v =& v_\infty \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix} \qquad \text{auf } \Gamma_\infty^- \\ \Gamma_\infty^- :=& \left\{ x \in \Gamma_\infty | \quad n(x) \cdot (\cos(\alpha), \sin(\alpha))^T < 0 \right\} \ . \end{aligned}$$

v_∞ beschreibt die Geschwindigkeit der freien Anströmung. Auf dem Flügelprofil setzten wir die No-Slip Bedingung

$$0 = v \cdot n(x) \qquad \text{auf } \Gamma_\omega \ .$$

Für eine Lösung y^* der Navier-Stokes Gleichungen können wir die aerodynamischen Beiwerte c_d für den Widerstand und c_l für den Auftrieb berechnen. Für die Beiwerte gilt die Gleichung

$$J(y) = \frac{1}{C_\infty} \int_{\Gamma_\omega} (pn - \tau n) \cdot \psi \ ds \tag{5.1}$$

mit der Konstanten $C_\infty = \frac{1}{2}\gamma p_\infty M_\infty^2 l$. l ist die Referenzlänge des Flügels, M_∞ die Machzahl am Fernfeldrand und p_∞ der Druck am Fernfeldrand. Für $\psi_d = (\cos(\alpha), \sin(\alpha))^T$ und $\psi_l = (-\sin(\alpha), \cos(\alpha))^T$ bekommen wir den Widerstands- und Auftriebsbeiwert.

Die Navier-Stokes Gleichungen werden in Padge mit Hilfe eines nichtlinearen Lösers gelöst. Das Residuum

$$R : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m \tag{Residuum}$$

berechnet die Navier-Stokes Gleichungen auf dem Strömungsgebiet. In diesem Fall ist $R(x, y) = \mathrm{Nv}(x, y)$. In dem Residuum $R(y, x)$ bezeichnen wir mit $x \in \mathbb{R}^n$ die Parametrisierung des Flügelprofils. In unseren Tests benutzen wir für das Profil die „Freenode“ Parametrisierung, bei der wir jeden Punkt p auf dem Profil frei bewegen

können. y beschreibt den Zustand des Systems. Die Jacobi-Matrix von R nach y ist in Padge durch

$$J : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{m \times m} \tag{Jacobi}$$

definiert. Zusammen beschreiben sie das Update im Status y als Newton-Schritt

$$y_{k+1} = y_k - J^{-1}(y_k, x)R(y_k, x). \tag{5.2}$$

Für die Funktionen R und J können wir mit dem differenzierten Code die Ableitungen nach y und x berechnen. Die Tests führen wir mit den zwei Funktionen $f_R(y, x) = \|R(y, x)\|_2^2$ und $f_J(y, x) = \|J(y, x)\|_F^2$ durch. Mit $\|\cdot\|_F$ kennzeichnen wir die Frobenius-Norm einer Matrix. Die Validierung unserer Ableitungen durch die Finiten Differenzen können wir mit diesen zwei Funktionen durchführen.

In den Abbildungen 5.3 und 5.4 wurde für f_R und somit für das Residuum R die Ableitung nach y und x gebildet. Für beide Ableitungen sieht man die erhoffte V-Kurve sehr deutlich. Der Unterschied der Gradienten

$$\nabla f_R = \frac{\partial \|R\|_2^2}{\partial y} \text{ und } \nabla f_R = \frac{\partial \|R\|_2^2}{\partial x}$$

beträgt für den Vorwärts- und Rückwärtsmodus 10^{-16}. Deshalb kann man für die zwei Kurven keinen Unterschied erkennen.

Für die Jacobi-Matrix ergibt sich dasselbe Bild. Die Gradienten

$$\nabla f_J = \frac{\partial \|J\|_F^2}{\partial y} \text{ und } \nabla f_J = \frac{\partial \|J\|_J^2}{\partial x}$$

wurden für die verschiedenen Methoden miteinander verglichen. Die Abbildungen 5.5 und 5.6 stellen diesen Vergleich dar. Hier kann man auch keinen Unterschied in den beiden Kurven erkennen, da die Differenz zwischen dem AD Vorwärts- und Rückwärtsmodus kleiner als 10^{-13} ist.

Für J und R haben wir somit die Ableitungen durch AD verifiziert. Nun wollen wir noch die Validierung des Updateschritts in (5.2) und der aerodynamischen Beiwerte durchführen. Der Widerstandsbeiwert c_d hängt durch seine Definition von der Lösung y^* der Iteration (5.2) ab und von dem Rand Γ_ω, damit auch von x. Wir können deshalb c_d in Abhängigkeit von x und y schreiben. Die Lösung y^* hängt wegen der Parametrisierung des Randes Γ_ω durch x auch von x ab.

In dem Test wurde deshalb nur nach dem Parameter x differenziert. Das Ergebnis für die Testfunktion $f(x) = C_d(x)$ ist in der Abbildung 5.7 dargestellt. Hier haben wir nur den AD Vorwärtsmodus betrachtet. Wir sehen, dass die Ableitung wegen der V-Kurve korrekt ist.

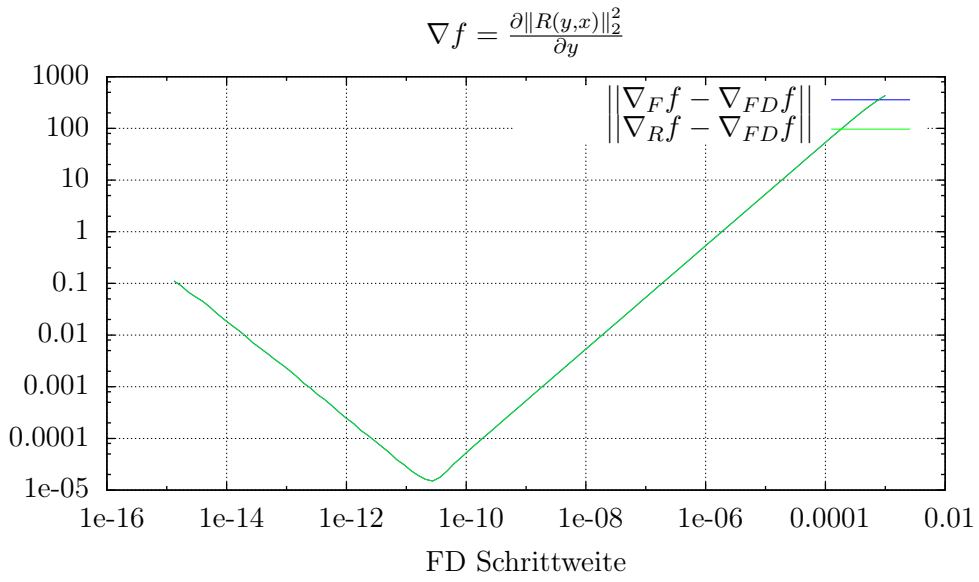

Abbildung 5.3.: Finiten Differenzen gegen AD Vorwärts und Rückwärts für das Residuum der diskretisierten Navier-Stokes Gleichungen in Padge nach y abgeleitet. Die beiden Kurven überlagern sich.

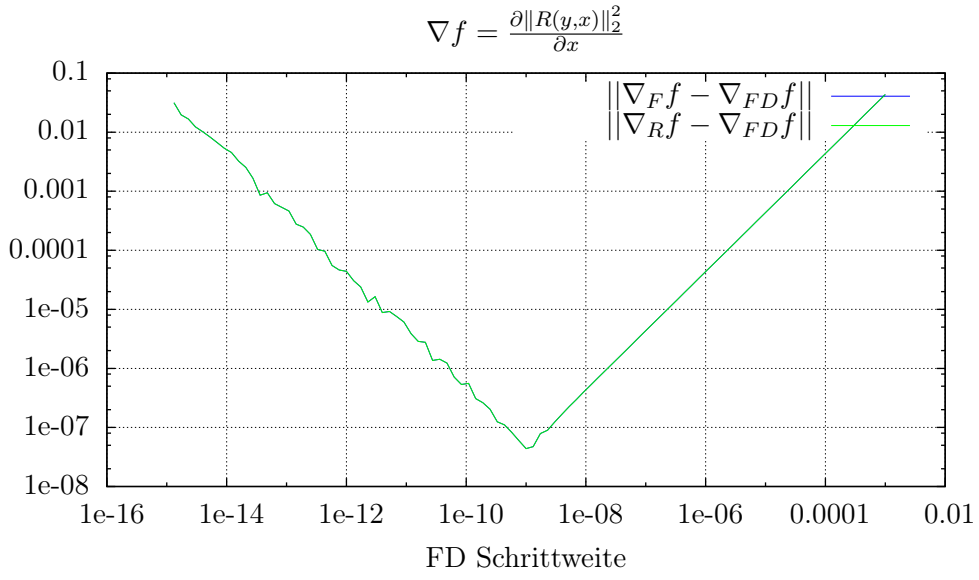

Abbildung 5.4.: Finiten Differenzen gegen AD Vorwärts und Rückwärts für das Residuum der diskretisierten Navier-Stokes Gleichungen in Padge nach x abgeleitet. Die beiden Kurven überlagern sich.

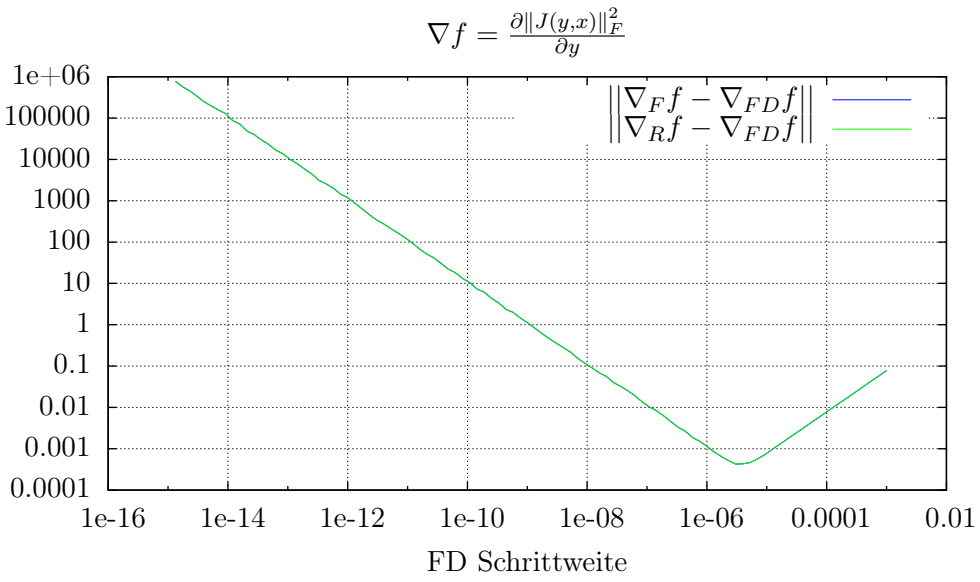

Abbildung 5.5.: Finiten Differenzen gegen AD Vorwärts und Rückwärts für die Jakobimatrix der diskretisierten Navier-Stokes Gleichungen in Padge nach y abgeleitet. Die beiden Kurven überlagern sich.

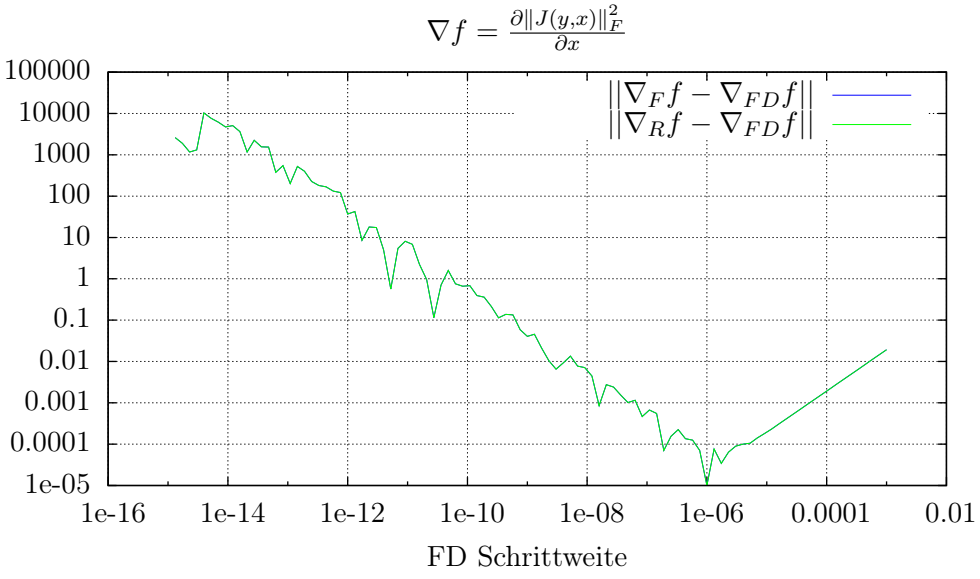

Abbildung 5.6.: Finiten Differenzen gegen AD Vorwärts und Rückwärts für die Jakobimatrix der diskretisierten Navier-Stokes Gleichungen in Padge nach x abgeleitet. Die beiden Kurven überlagern sich.

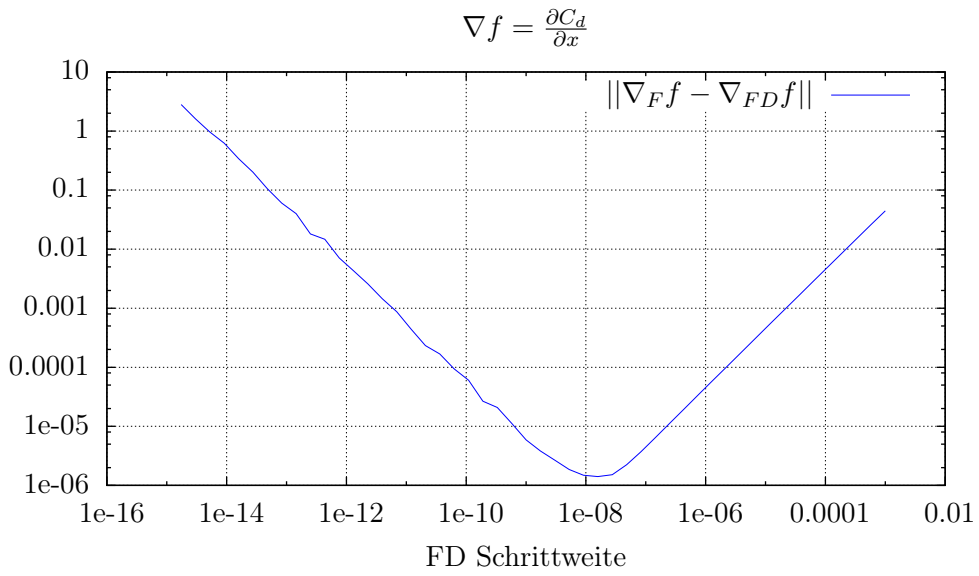

Abbildung 5.7.: Finiten Differenzen gegen AD Vorwärts für den Widerstandsbeiwert nach einem vollen Padge-Durchlauf.

6. Berechnung des Präkonditionierers in Padge

Das Ziel dieser Arbeit ist es den Präkonditionierer für das One Shot Verfahren in Padge bereit zu stellen. Deshalb geben wir in diesem Kapitel nochmal eine Übersicht aller Formeln, die wir für die Berechnung des Präkonditionierers benötigen. Danach wird die Implementierung in Padge beschrieben.

Für den Präkonditionierer B haben wir in (3.6) die Formel

$$B := \frac{1}{\sigma}\nabla_{uu}L^a \tag{6.1}$$

erhalten. σ wurde in Satz 3.7 als

$$\sigma := (1-\rho) - \frac{(1+\frac{\theta}{2}\beta)^2}{\alpha\beta(1-\rho)} \tag{6.2}$$

definiert. Da wir die Hessematrix $\nabla_{uu}L^a$ vermeiden wollen, nähren wir B mit einem BFGS Update an. Das Update für B^{-1} wurde in (3.7) als

$$H_{k+1} = (I - r_k\Delta u_k R_k^T)H_k(I - r_k R_k \Delta u_k^T) + r_k\Delta u_k\Delta u_k^T \text{ mit } r_k = \frac{1}{R_k^T\Delta u_k} \tag{6.3}$$

beschrieben. R_k wurde durch

$$R_k := \nabla_u L^a(y_k, \lambda_k, u_k + \Delta u_k) - \nabla_u L^a(y_k, \lambda_k, u_k) \tag{6.4}$$

definiert. Der Gradient von der erweiterten Lagrangefunktion L^a hat die Form

$$\nabla_u L^a = \alpha\Delta y^T G_u + \beta\Delta\lambda^T N_{yu} + N_u \ . \tag{6.5}$$

Die Koeffizienten α und β in (6.2) und (6.5) wurden im Satz 3.8 als

$$\beta = \frac{3}{\sqrt{\theta^2 + 3\frac{\|N_{yu}\|_2^2}{\|G_u\|_2^2}(1-\rho)^2} + \frac{\theta}{2}} \text{ und } \alpha = \frac{\|N_{yu}\|_2^2\,\beta(1+\frac{\theta}{2}\beta)}{\|G_u\|_2^2\,(1-\frac{\theta}{2}\beta)} \tag{6.6}$$

definiert. Als letztes fehlen noch die Skalare ρ, θ, $\mu = \|G_u\|_2$ und $\nu = \|N_{yu}\|_2$, die in dem Abschnitt 3.4 definiert wurden. Die Updates haben die Form

$$\rho_{k+1} = \max\{\frac{\|\Delta y_k\|}{\|\Delta y_{k-1}\|}, \tau\rho_k\}, \tag{6.7}$$

$$\theta_{k+1} = \max\{\frac{|\Delta\lambda_k^T \Delta y_{k+1} - \Delta y_k^T \Delta\lambda_{k+1}|}{\|\Delta y_k\|_2^2}, \tau\theta_k\}, \tag{6.8}$$

$$\mu_{k+1} = \max\left\{\frac{\left\|\Delta y_k^T G_u\right\|_2}{\|\Delta y_k\|_2}, \tau\mu_k\right\} \text{ und} \tag{6.9}$$

$$\nu_{k+1} = \max\left\{\frac{\left\|\Delta\lambda_k^T N_{yu}\right\|_2}{\|\Delta\lambda_k\|_2}, \tau\nu_k\right\} \tag{6.10}$$

mit $\tau \in (0, 1)$.

Für die Berechnung der Funktionen brauchen wir in Padge eine Formulierung für den verschobenen Lagrangeterm $N(y, \lambda, u)$. Da dieser Term sich aus f und G zusammensetzt, haben wir in Padge drei Funktionen implementiert, die $f(y, u)$, $G(y, u)$ und $N(y, \lambda, u)$ berechnen. In Abbildung 6.1 werden die drei Funktionen dargestellt. u kommt in den Funktionen als Argument nicht vor, da es sich um die Parametrisierung des Flugzeugrandes handelt, gibt es in Padge noch nicht die Möglichkeit die Parametrisierung als Vektor zu übergeben. Deshalb müssen Änderungen für u vor dem Aufruf einer Funktion durchgeführt werden.

Bevor wir angeben, in welcher Reihenfolge wir die einzelnen Terme berechnen, müssen wir uns noch anschauen, wie wir in (6.5) den Term $\Delta\lambda^T N_{yu}$ auswerten.

Dazu bilden wir im ersten Schritt den Gradienten

$$\nabla N = \begin{pmatrix} N_y \\ N_\lambda \\ N_u \end{pmatrix} \in \mathbb{R}^{m+m+n}$$

und die Hessematrix

$$\nabla^2 N = \begin{pmatrix} N_{yy} & N_{y\lambda} & N_{yu} \\ N_{\lambda y} & N_{\lambda\lambda} & N_{\lambda u} \\ N_{uy} & N_{u\lambda} & N_{uu} \end{pmatrix} \in \mathbb{R}^{m+m+n \times m+m+n}$$

von N. In (1.7) haben wir für einen AD Rückwärtsvorwärtsschritt die Gleichung

$$\dot{\bar{x}} = \left[\frac{d^2 f}{d^2 x}\dot{x}\right]^T \bar{y} + \frac{df}{dx}^T \dot{\bar{y}}$$

für die Berechnung der zweiten Ableitung. In dieser Gleichung wurde $y = f(x)$ verwendet. Für die Identität $f \equiv N$ bekommen wir für die Parameter die Identität $(y, \lambda, u) \equiv x$ und für die Ergebnisse $w \equiv y$. Damit ergibt sich die Gleichung

```
template <int dim> VECTOR UserProblem<dim>::calculate_G(VECTOR &y) {
  ProblemType& op = status->equation_operator;

  // calculate residual
  op.flag_recompute();
  op.compute_residual(y);
  VECTOR& residual = op.get_residual();

  // calculate the jacobian
  op.flag_recompute_Jacobian();
  MATRIX& jacobi = op.NegativeJacobian(y);

  // calculate d
  VECTOR d(residual); // clone the vector
  petsc_solve(jacobi, residual, d, 1e-80);

  // return y + d not y - d because we calculate the negative jacobian
  return y + d;

}

template <int dim> BASE_TYPE UserProblem<dim>::calculate_f(VECTOR &y) {
  std::map<typename ForceCoefficients<dim>::Type,BASE_TYPE> values;
  this->force_coefficients->evaluate(*Primal::discretization.dof_handler,
      this->mapping.collection, Primal::discretization.quadrature.face,
      valueG, values);

//  return values[ForceCoefficients<dim>::lift_force];
  return values[ForceCoefficients<dim>::drag_force];
}

template <int dim> BASE_TYPE UserProblem<dim>::calculate_N(VECTOR &y,
    VECTOR &lambda) {
  VECTOR g = calculate_G(y);

  BASE_TYPE f = calculate_f(y);

  return f + g * lambda;
}
```

Abbildung 6.1.: Funktion für die Berechnung des verschobenen Lagrangeterms in Padge

$$(\dot{\bar{y}}, \dot{\bar{\lambda}}, \dot{\bar{u}}) = \left[\nabla^2 N \begin{pmatrix} \dot{y} \\ \dot{\lambda} \\ \dot{u} \end{pmatrix} \right]^T \bar{w} + \nabla N^T \dot{\bar{w}} \ .$$

Wenn wir $\dot{\bar{w}}$ auf 0 setzten erhalten wir einen Ausdruck für die Hessematrix von N

$$\begin{pmatrix} \dot{\bar{y}} \\ \dot{\bar{\lambda}} \\ \dot{\bar{u}} \end{pmatrix} = \bar{w} \begin{pmatrix} N_{yy} & N_{y\lambda} & N_{yu} \\ N_{\lambda y} & N_{\lambda\lambda} & N_{\lambda u} \\ N_{uy} & N_{u\lambda} & N_{uu} \end{pmatrix} \begin{pmatrix} \dot{y} \\ \dot{\lambda} \\ \dot{u} \end{pmatrix} \ .$$

Wir müssen jetzt entscheiden, wie wir die Parameter $\bar{w}, \dot{y}, \dot{\lambda}$ und $\dot{u}$ wählen, um den Term $\Delta\lambda^T N_{yu}$ zu berechnen. Durch die Wahl von $\bar{w} = 1$ und $(\dot{y}, \dot{\lambda}, \dot{u}) = (\Delta\lambda, 0, 0)$ erhalten wir

$$\begin{pmatrix} \dot{\bar{y}} \\ \dot{\bar{\lambda}} \\ \dot{\bar{u}} \end{pmatrix} = \begin{pmatrix} N_{yy}\Delta\lambda \\ N_{\lambda y}\Delta\lambda \\ N_{uy}\Delta\lambda \end{pmatrix} \ .$$

In der dritten Komponente steht die Gleichung $\dot{\bar{u}} = N_{uy}\Delta\lambda$. Da die Hessematrix symmetrisch ist, muss gelten, dass $N_{uy}^T = N_{yu}$ ist. Damit erhalten wir die Gleichung

$$\dot{\bar{u}}^T = \Delta\lambda^T N_{yu}$$

und somit wissen wir nun, dass wenn wir u als aktive Variable behandeln und $\dot{y} = \Delta\lambda$ setzten, dass nach einem AD Rückwärtsvorwärtslauf in $\dot{\bar{u}}$ das Ergebnis von $\Delta\lambda^T N_{yu}$ steht.

Um zu verdeutlichen, wie man mit DCO die Ableitungen berechnet, geben wir nun für die Berechnung von $\Delta\lambda^T N_{yu}$ den Code an. In der Abbildung 6.2 sehen wir, wie zuerst u als aktive Variable gesetzt wird. Danach wird y als aktiv registriert und wir setzten $\dot{y}$ auf $\Delta\lambda$. Nach der Berechnung von N können wir uns von DCO die Ableitung ausgeben lassen.

In der Tabelle 6.1 ist nun Angegeben, wie wir den Präkonditionierer B in Padge berechnen. Unter *Berechnung* steht jeweils die Anweisung, die wir in Padge durchführen. Die Terme auf der linken Seite werden gesetzt. Die Spalte *AD Variablen* beschreibt welche Variablen wir *aktiv* setzten, um für diese Variablen die Ableitung zu berechnen. Sie enthält auch die Information über die Werte, die wir für die Vorwärts- und Rückwärtsvariablen setzten. In *Ergebnis* stehen die Zuweisungen, die wir aus dem AD lauf bekommen. Die letzte Spalte *Gleichung* beschreibt die Formel, die wir in dieser Zeile berechnen.

Wir haben nun alles Implementiert, was wir für die Berechnung des Präkonditionierers benötigen.

```
template <int dim> BASE_TYPE UserProblem<dim>::eval_N_uy(...) {
  dco::t2s_a1s::global_tape = dco::t2s_a1s::tape::create(1e8);

  // register u // pseudo code u is no vector
  for(int i = 0; i < u.size(); ++i) {
    dco::t2s_a1s::global_tape->register_variable(u(i));
  }

  // register y and set y dot to delta lambda
  for(int i = 0; i < y.size(); ++i) {
    dco::t2s_a1s::global_tape->register_variable(y(i));
    dco::t2s_a1s::set(y(i), yBar(i), 0, 2);
  }

  BASE_TYPE w = calculate_N(y, lambda);

  // set w bar to 1
  dco::t2s_a1s::set(w, 1.0, -1);

  dco::t2s_a1s::global_tape->interpret_adjoint();

  // get y dot bar which is Delta lambda^T N_{yu}
  for(int i = 0; i < y.size(); ++i) {
    dco::t2s_a1s::get(y(i), yDotBar, -1, 2);
  }
}
```

Abbildung 6.2.: Die Berechnung von $w = \Delta\lambda^T N_{yu}$ mit DCO. Es werden $\dot{y} = \Delta\lambda$ und $\bar{w} = 1$ gesetzt.

Tabelle 6.1.: Berechnung des Präkonditionierers in Padge

Berechnung	AD Variablen	Ergebnis	Gleichung
$w_k = N(y_k\lambda_k, u_k)$	$\bar{w} = 1$ Aktiv: y, λ, u	$w_y = N_y$ $w_\lambda = N_\lambda = G$ $w_{u1} = N_u$	 (6.5) 3. Term
$\Delta y_k = w_\lambda - y_k$ $\Delta\lambda_k = w_y - \lambda_k$ $\Delta u_k = -H_k w_{u1}$			
$d = G(y_k, u_k)$	$\bar{d} = \Delta y_k$ Aktiv: u	$a_1 = \Delta y_k^T G_u$	(6.5) 1. Term
$w_k = N(y_k, \lambda_k, u_k)$	$\bar{w} = 1, \dot{y} = \Delta\lambda_k$ Aktiv: u	$b_1 = \Delta\lambda_k^T N_{yu}$	(6.5) 2. Term
$u_{k+1} = u_k + \Delta u_k$!! Gitter Anpassung !!			
$d = G(y_k, u_{k+1})$	$\bar{d} = \Delta y_k$ Aktiv: u	$a_2 = \Delta y_k^T G_u$	(6.5) 1. Term
$w_k = N(y_k, \lambda_k, u_{k+1})$	$\bar{w} = 1, \dot{y} = \Delta\lambda_k$ Aktiv: u	$b_2 = \Delta\lambda_k^T N_{yu}$ $w_{u2} = N_u$	(6.5) 2. Term (6.5) 3. Term
$\beta = \dfrac{3}{\sqrt{\theta_k^2 + 3\dfrac{\nu_k^2}{\mu_k^2}(1-\rho_k)^2} + \dfrac{\theta_k}{2}}$ $\alpha = \dfrac{\nu_k^2\beta(1 + \dfrac{\theta_k}{2}\beta)}{\mu_k^2(1 - \dfrac{\theta_k}{2}\beta)}$			(6.6) (6.6)

Tabelle 6.1.: Berechnung des Präkonditionierers in Padge

Berechnung	AD Variablen	Ergebnis	Gleichung
$\sigma = (1-\rho_k) - \dfrac{(1+\frac{\theta_k}{2}\beta)^2}{\alpha\beta(1-\rho_k)}$			(6.2)
$L_1 = \alpha a_1 + \beta b_1 + w_{u1}$			(6.5)
$L_2 = \alpha a_2 + \beta b_2 + w_{u2}$			(6.5)
$R_k = L_2 - L_1$			(6.4)
$r_k = \dfrac{1}{R_k^T \Delta u_k}$			(6.3)
$H_{k+1} = (I - r_k \Delta u_k R_k^T) H_k (I - r_k R_k \Delta u_k^T) + r_k \Delta u_k \Delta u_k^T$			(6.3)
$\rho_{k+1} = \max\left\{ \dfrac{\|\Delta y_k\|_2}{\|\Delta y_{k-1}\|_2}, \tau\rho_k \right\}$			(6.7)
$\theta_{k+1} = \max\left\{ \dfrac{\lvert\Delta\lambda_k^T \Delta y_{k+1} - \Delta y_k^T \Delta\lambda_{k+1}\rvert}{\|\Delta y_k\|_2^2}, \tau\theta_k \right\}$			(6.8)
$\mu_{k+1} = \max\left\{ \dfrac{\|b_1\|_2}{\|\Delta y_k\|_2}, \tau\mu_k \right\}$			(6.9)
$\nu_{k+1} = \max\left\{ \dfrac{\|b_2\|_2}{\|\Delta\lambda_k\|_2}, \tau\nu_k \right\}$			(6.10)

7. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit ist es den Präkonditionierer auf der Basis von Padge bereitzustellen.

Die ersten zwei Kapitel dienten nur zur Einarbeitung in die verwendeten Hilfsmittel und sind aus den angegeben Quellen erstellt worden. Das erste Kapitel nimmt eine besondere Rolle ein, da es die Kernmethodik darstellt, die wir zum Ableiten verwendet haben, deshalb lag dort ein großer Fokus.

Das dritte Kapitel war auch nur eine Zusammenstellung aus den angegebenen Quellen und diente dazu alle Voraussetzungen für die Berechnung des Präkonditionierers darzustellen. In der Literatur war dafür alles vorhanden, wir mussten nur für die Approximation von $\|G_u\|_2$ und $\|N_{yu}\|_2$ eine Möglichkeit finden.

Für die Ableitung eines Codes kann man keine Anleitung für alle möglichen Probleme schreiben. Das Einzige, an das man sich halten kann, sind Richtlinien. Diese Richtlinien helfen jedoch nicht bei den meisten Problemen, die bei der Differentiation von Padge und deal.II aufgetreten sind. Wir konnten vorher nicht absehen welche Probleme auftreten werden, da der Code für uns vollkommen fremd war.

Der Weg unserer Bemühungen wurde in Kapitel 4 dargestellt und der erfolgreiche Abschluss in Kapitel 5. Damit wurden alle Voraussetzungen geschaffen um den Präkonditionierer für die One-Shot Optimierung zu berechnen.

Als nächsten Schritt kann man Padge seine Fähigkeit auf mehreren Rechnern zu rechnen zurückgeben. Da man hierfür die Ableitung von MPI braucht, muss man MPI mit DCO ableiten. Die Programmierer von DCO arbeiten zurzeit an dieser Ableitung, sodass man in nächster Zeit Padge auf mehreren Rechnern rechnen lassen kann.

Ein anderer Schritt ist die Einführung von Templates in den Page Code. Bisher haben wir „nur“ den Rechentypen global getauscht und dadurch deal.II bzw. Padge differenziert. Damit wurde aber ein großer Schritt für die Einführung von Templates in die Klassen getan.

A. Anhang

Satz A.1
Es sei $A \in \mathbb{R}^{n\times m}$ eine beliebige Matrix. Dann gilt für die Spektralnorm $\|\cdot\|_2$ von A

$$\|A\|_2^2 = \left\|AA^T\right\|_2$$

Beweis. Die Spektralnorm von A ist definiert durch

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^TA)}.$$

Da A^TA eine reelle symmetrische Matrix ist, existiert die Eigenwertzerlegung von A^TA in $A^TA = G\Lambda G^T$. $G \in \mathbb{R}^{m\times m}$ ist eine Orthogonale Matrix, d.h. es gilt $G^TG = I$. $\Lambda \in \mathbb{R}^{m\times m}$ ist eine Diagonalmatrix mit den Eigenwerten von A^TA auf der Diagonalen. Da die Eigenwerte für ähnliche Matrizen gleich sind, erhalten wir für die Spektralnorm

$$\|A\|_2^2 = \lambda_{\max}(A^TA) = \lambda_{\max}(G\Lambda G^T) = \lambda_{\max}(\Lambda).$$

Für die Spektralnorm von AA^T bekommen wir

$$\begin{aligned}
\left\|AA^T\right\|_2 &= \sqrt{\lambda_{\max}((AA^T)^TAA^T)} \\
&= \sqrt{\lambda_{\max}(AA^TAA^T)} \\
&= \sqrt{\lambda_{\max}(G\Lambda G^TG\Lambda G^T)} \\
&= \sqrt{\lambda_{\max}(\Lambda^2)} \\
&= \lambda_{\max}(\Lambda) \\
&= \|A\|_2^2 .
\end{aligned}$$

□

Literaturverzeichnis

[1] Hans Wilhelm Alt. *Lineare Funktionalanalysis*. Springer, 2006.

[2] DCO. `http://wiki.stce.rwth-aachen.de/content/software/dco_cpp.html`.

[3] deal.II. `http://www.dealii.org`.

[4] A. Griewank and A. Walther. *Evaluating Derivatives, second edition*. SIAM, 2008.

[5] Andreas Griewank, Nicolas Gauger, and J. Riehme. Extension of fixed point pde solvers for optimal design by one-shot method. *European Journal of Computational Mechanics*, 17:87–102, 2008.

[6] A. Hamdi and A. Griewank. Properties of an augmented lagrangian for design optimization. *to appeear in Optimization Methods and Software*, 2008.

[7] A. Hamdi and A. Griewank. Reduced quasi-newton method for simultaneous design and optimization. *Computational Optimization and Applications*, submitted, 2008.

[8] Ralf Hartmann. Numerical analysis of higher order discontinuous Galerkin finite element methods. In H. Deconinck, editor, *VKI LS 2008-08: CFD - ADIGMA course on very high order discretization methods, Oct. 13-17, 2008*. Von Karman Institute for Fluid Dynamics, Rhode Saint Genèse, Belgium, 2008.

[9] Konrad Königsberger. *Analysis 2, 5. korrigierte Auflage*. Springer, 2004.

[10] METIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[11] MPI. `http://www.mcs.anl.gov/research/projects/mpi`.

[12] NetCDF. `http://www.unidata.ucar.edu/software/netcdf`.

[13] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operational Research, 1999.

[14] OpenCascade. `http://www.opencascade.org`.

[15] E. Özkaya and N. Gauger. Automatic transition from simulation to one-shot shape optimization with navier-stokes equations. *GAMM-Mitt.*, 33(2):133–147, 2010.

[16] PETSc. `http://www.mcs.anl.gov/petsc/petsc-as`.

[17] Sacado. `http://trilinos.sandia.gov/packages/docs/dev/packages/sacado/doc/html/index.html`.

[18] Bjarne Stroustrup. *Die C++ Programmiersprache*. ADDISON-WESLEY, 4 edition, 2000.

[19] J. Werner. *Numerische Mathematik 1: Lineare und nichtlineare Gleichungssysteme, Interpolation, numerische Integration*. Friedr. Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, Deutschland, 1992.

Selbständigkeitserklärung

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den

_____

Datum, Unterschrift