FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Implementierung des TGV Algorithmus mithilfe von ExaSlang

Max Gerecke

Matrikelnummer 21503710

Bachelorarbeit

Implementierung des TGV Algorithmus mithilfe von ExaSlang

Max Gerecke

Bachelorarbeit

Aufgabensteller:	PD Dr.-Ing. habil. Harald Köstler
Betreuer:	M. Sc. Sebastian Kuckuk
Bearbeitungszeitraum:	1.6.2017 - 31.10.2017

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30.10.2017 .....

Inhaltsverzeichnis

1 Einleitung

ExaStencil ist ein Forschungsprojekt mit dem Ziel aus einer Problemspezifikation durch automatische Generierung eine möglichst optimierte Implementierung zu erzeugen. Im Rahmen dieses Forschungsprojektes wurde die Domaine Specific Language ExaSlang erstellt. In dieser Arbeit soll die Qualität der von ExaSlang erzeugten Implementierungen eines Problems mit herkömmlichen Implementierungen verglichen werden. Bei der dafür ausgewählten Problemspezifikation handelt es sich um die in der Veröffentlichung [1] vorgestellten Methoden L2-TGV und L1-TGV, die durch Anwendung der preconditioned Douglas-Rachford Iteration umgesetzt wurden. Dies sind Methoden der Bildverarbeitung und wurden im Rahmen der Arbeit mit ähnlichen Methoden verglichen und als vorteilhaft empfunden. Das Ziel dieser Arbeit soll es sein diese Methoden anhand der Problemspezifikation mithilfe von ExaSlang zu implementieren und die Ergebnisse dieser Implementierungen mit den vorgestellten Beispielimplementierungen zu vergleichen. Dabei soll gezeigt werden, dass durch automatisierte Optimierung die Implementierung der Problemspezifikation deutlich beschleunigt werden kann.

2 ExaStencil

ExaStencil steht für "Advanced Stencil-Code Engineering". Bei dem Forschungsprojekt handelt es sich um eine Arbeitsgemeinschaft von 6 verschiedenen Lehrstühlen an vier verschiedenen Orten. Die Mitglieder sind der Chair of Programming der Universität Passau, der Chair of Software Engineering der Universität Passau, der Chair of System Simulation der Friedrich-Alexander Universität Erlangen-Nürnberg, der Chair of Hardware-Software-Co-Design der Friedrich-Alexander Universität Erlangen-Nürnberg, der Chair of Applied Computer Science der Universität Wuppertal sowie der Chair of Creative Informatics der University of Tokyo.

Ziel des Projektes ist die Entwicklung einer Software, die die automatische Generierung von Code erlaubt, der Probleme im ausgewählten Problemfeld mit exascale Leistung lösen kann. Exascale bedeutet dabei eine Anwendung mit mindestens einem exaFLOP oder einer Milliarde Rechenschritten pro Sekunde. Dieses Ziel soll durch die Nutzung einer Domaine Specific Language(DSL) umgesetzt werden, welche durch Code Erzeugung so unterstützt, dass das Ergebnis exascale Leistung verwendet.

Als spezifisches Problemfeld, für das die Code Erzeugung stattfinden soll, wurde dafür Stencil Codes ausgewählt. Dabei handelt es sich um Algorithmen die einen hohen Rechenaufwand besitzen. Stencil Codes sind dadurch gekennzeichnet, dass jede Zelle eine Vektorfeldes wiederholt durch die Eigenschaften seiner Nachbarzellen neu berechnet wird. Das verwendete Muster für die wiederholte Berechnung der Zelle wird als Stencil bezeichnet.

Um die automatische Generierung des Codes zu unterstützen wurde im Rahmen des ExaStencil Projektes eine eigene DSL entwickelt. Diese wird als ExaSlang bezeichnet. Das macht vor allem dann Sinn, wenn man in Betracht zieht, dass die zu lösenden Probleme teilweise extreme Rechenleistungen erfordern, die nur durch die Verwendung von großen Rechnerclustern, die parallel auf dem Problem arbeiten, bereitgestellt werden können. Dabei handelt es sich aber meist um heterogene Systeme aus hunderten von Rechnerknoten die sich jeweils in Architektur und Leistung unterscheiden können. Hier ist die Verwendung von automatisch erzeugtem Code auf Basis einer DSL sehr hilfreich. Das Konzept erlaubt es eine einzige Problemspezifikation in der verwendeten DSL zu entwickeln. Diese Spezifikation wird dann abhängig von den lokalen Eigenschaften des Knotens unabhängig voneinander optimiert und in die Hostsprache der DSL übersetzt. Dadurch wird für jeden einzelnen der verwendeten Knoten die möglichst optimale und effizienteste Umsetzung der Problemspezifikation erreicht.

2.1 ExaSlang

ExaSlang ist die im Rahmen des ExaStencil Projekt entwickelte und verwendete DSL. Das Ziel von ExaSlang ist die Lösung von Stencil Codes und im besonderen die Lösung von Stencil Codes im Rahmen von Mehrgitterverfahren. Es geht allerdings nicht nur um die Lösung dieser Probleme sondern mithilfe der Spezifikation des Problems soll durch automatische Generierung und Optimierung eine möglichst leistungsstarke Implementierung dieser Lösung, für die gegebene Zielplattform, Erzeugt werden.

Um ExaSlang für eine breitere Menge von Nutzern verfügbar zu machen besteht eine Auswahl von mehreren Schichten auf denen gearbeitet werden kann. Es stehen dafür vier verschiedene Schichten zu Verfügung. Wie man in (Abbildung 1) sehen kann unterscheiden sich die Schichten im Grad der Abstraktion und der Anzahl der konkret zu Verfügung stehenden Programmiermethoden. So besitzt die erste Schicht einen hohen Grad der Abstraktion aber wenige konkrete Programmiermethoden, was es erlaubt, dass sie von Wissenschaftlern und Experten des Problemfeldes genutzt wird ohne das diese über Wissen im Bereich der Computerprogrammierung verfügen. Die vierte Schicht hingegen besitzt einen sehr geringen Grad der Abstraktion aber dafür stehen alle von der DSL bereitgestellten Methoden zur Verfügung. Diese Schicht erlaubt es zwar eine möglichst umfangreiche Problemspezifikation festzulegen dafür benötigt der Nutzer allerdings umfangreichere Kenntnisse über die verwendeten Funktionen.

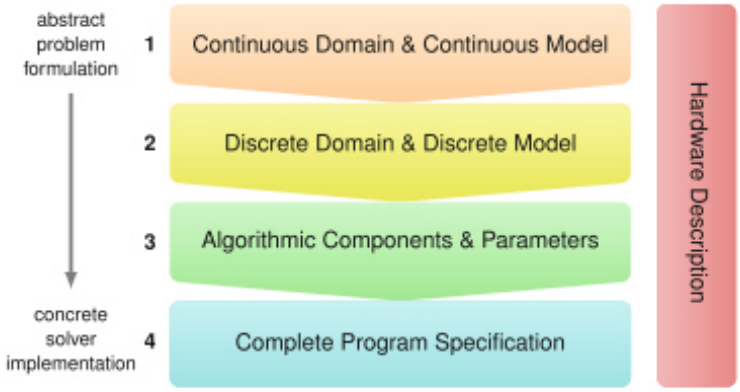

Abbildung 1 : vier Schichten von ExaSlang

Zusätzlich werden bei der Kompilierung von ExaSlang in die Hostsprache C++ die verschiedensten Optimierungsverfahren verwendet. Ein solches Verfahren ist zum Beispiel die lokale Fourier Analysis. Diese ermöglichst es schon während der Kompilierung von ExaSlang eine Vorhersage über die Konvergenz einer Iteration zu treffen solang das Konvergenzkriterium bekannt ist. Dadurch ist es möglich, schon im Voraus durch die Wahl der Implementierung der Iteration eine Optimierung durchzuführen.

Eine weitere verwendete Optimierungsmethode von ExaSlang ist die Verwendung von Software Product Line Verfahren .Dabei wird eine Anwendung als Teil einer Menge von Anwendungen angesehen. Die Mitglieder dieser Menge haben bestimmte Gemeinsamkeiten, zum Beispiel die gleichen Hardwarevoraussetzungen, aber auch eine Anzahl von Freiheitsgraden. Durch die Erfahrung der Erzeugung von vorhergehenden Anwendungen dieser Menge können neue Anwendungen, der Menge, als Variation von schon bekannten Anwendungen erzeugt werden . Somit können neue Teile des Erzeugungs- und Optimierungsprozesses wiederverwendet werden.

Eine weitere Komponente von ExaSlang ist die Nutzung einer target platform description language(TPDL). Mithilfe der TPDL werden einerseits vorhandene Hardwarekomponenten wie Speicher, Prozessor und Architektur der Zielplattform beschrieben aber auch die vorhanden Softwarekomponenten, wie Compiler, die zur Verfügung stehen. Auch diese Informationen werden für die Erzeugung einer möglichst optimierten Anwendung genutzt.

Weitere Informationen zum Thema ExaSlang finden sich in [3] sowie in [4].

3 Theoretische Grundlagen

In der Veröffentlichung [1] werden mehrere neue Algorithmen vorgestellt. Diese basieren auf einer Anwendung des in [5] vorgestellten preconditioned Douglas-Rachford (PDR) Algorithmus für konvex-konkave Sattelpunktprobleme auf Probleme aus dem Bereich der Bildverarbeitung,

im speziellen auf die Probleme Total Variation (TV) und Total Generalized Variation (TGV). Vorgestellt werden iterative Algorithmen für L^2-TV-denoising, L^1-TV-denoising, L^2-TV-deblurring, L^1-TV-deblurring, L^2-TGV-denoising sowie L^1-TGV-denoising. Dabei wird für denoising Algorithmen ein Rot-Schwarzer-Gauß- Seidel Preconditioner verwendet und für deblurring Probleme ein Fast Fourier Transformation(FFT) Preconditioner.

3.1 Douglas-Rachford Iteration

In der Veröffentlichung [1] wird die preconditioned-Douglas Rachford (PDR) Iteration vorgestellt. Dabei handelt es sich um einen Algorithmus zur Lösung von konvex-konkaven Sattelpunktproblemen. Diese basiert auf der Methode aus [5] Die zu lösenden Sattelpunktprobleme haben dabei die Form wie von

(1)

$$\min_{x \in domF} \max_{y \in domG} \langle Kx, y \rangle + F(x) - G(y).$$

Dabei handelt es sich bei X sowohl als auch bei Y um reelle Hilbert-Räume und bei $K : X \to Y$ um eine stetige lineare Abbildung. Die Funktionen

$$F : X \to R_\infty \text{ und } G : Y \to R_\infty \text{ , mit } R_\infty = (-\infty, +\infty)$$

sind echt, konvex und linksseitig stetig. Hinzukommt, dass auch die Fenchel Konjugierten

$$F^* : X \to R_\infty, G^* : Y \to R_\infty$$

die definiert sind durch

$$F^*(x) = \sup_{x' \in X} \langle x, x' \rangle - F(x')$$

$$G^*(x) = \sup_{y' \in Y} \langle y, Y' \rangle - G(y'),$$

ebenfalls echt, konvex und linksseitig stetig sind. Somit kann das Sattelpunktproblem(1) als die Fenchel-Rockafellar-Dualität angesehen werden mit den primären und sekundären Problemen in (2). Diese Probleme werden häufig in der Bildverarbeitung bei Methoden des Image denoising und deblurring verwendet. Dort ist K der ∇ Operator und $G^*(Kx)$ bezieht sich auf ein Problem der Totalen Variation.

(2)

$$\min_{x \in X} F(x) + G^*(Kx), \max_{y \in Y} -F^*(-K^*y) - G(y)$$

Unter der Bedingung, dass ein Minimum und Maximum für das primäre und sekundäre Problem existiert und diese übereinstimmen ist das Lösungspaar

$$(x^*, y^*) \in X \times Y$$

für das primäre und sekundäre Problem gleichzeitig die Lösung des Sattelpunktproblems (1).

Dabei gibt es zwei verschiedene Varianten der preconditioned-Douglas-Rachford Iteration. Die erste, als PDR bezeichnete Variante, ist anwendbar auf Probleme in denen die Resolvente von F, G eindeutig gelöst werden kann. Bei der zweiten Variante handelt es sich um die sogenannte PDRQ Methode. Diese wird dann angewendet, wenn entweder das primäre oder sekundäre Problem nur quadratisch linear lösbar ist.

Beide Varianten wenden einen linearen Preconditioner an. Dieser wird dazu verwendet die Gleichung

$$Tx = b$$

zu lösen, wobei T durch das zu lösende Problem gegeben ist. Der Preconditioner wird auf das Problem angewendet durch

$$x^{new} = x^{old} + M^{-1}(b - T\,x^{old}).$$

T ist abhängig von K und K^* . Deren Anwendung auf Vektoren ist berechenbar. Deshalb wird auch der Preconditioner M so gewählt, dass seine Anwendung M^{-1} einfach zu berechnen ist. Damit die Konvergenz der Methode sichergestellt ist muss M weitere Eigenschaft besitzen. Die erste Bedingung ist, dass $T, M : X \to X$ linear, stetig, symmetrisch und positiv definit sein müssen und zusätzlich muss $M - T$ positiv semidefinit sein, damit M ein zulässiger Preconditioner für T ist.

Positiv definit bedeutet alle Eigenwerte müssen größer als null sein. Positiv semidefinit hingegen bedeutet, dass alle Eigenwerte größer oder gleich sein müssen.

Die PDR Variante wird nun Initialisiert mit

$$T = T_{PDR} = I + \sigma^2 K^* K,$$

einer Schrittgröße von $\sigma > 0$ und M als zulässigen Preconditioner für T. Dadurch entsteht die Douglas-Rachford Iterationsmethode wie in (3).

PDR Variante:

(3)

$$Zu\ lösen: \min_{x \in dom\, F} \max_{y \in domG} \langle Kx, y\rangle + F(x) - G(y)$$

$$Initialisierung: (x^0, \bar{x}^0, \bar{y}^0) \in X \times X \times Y, \sigma > 0\ Schrittgröße$$

$$T = I + \sigma^2 K^* K, M - T \geq 0$$

$$Iteration: \begin{cases} b^k = \bar{x}^k - \sigma K^* \bar{y}^k \\ x^{k+1} = x^k + M^{-1}(b^k - Tx^k) \\ y^{k+1} = \bar{y}^k + \sigma K x^{k+1} \\ \bar{x}^{k+1} = \bar{x}^k + (I + \sigma \partial F)^{-1}[2x^{k+1} - \bar{x}^k] - x^{k+1} \\ \bar{y}^{k+1} = \bar{y}^k + (I + \sigma \partial G)^{-1}[2y^{k+1} - \bar{y}^k] - y^{k+1} \end{cases}$$

Für die Variante PDRQ hingegen wird vorausgesetzt, dass entweder das primäre oder sekundäre Problem nur quadratisch linear lösbar ist. Im Beispiel aus [1] wird nur der Fall, dass das primäre Problem quadratisch linear lösbar ist betrachtet. Somit gilt

$$F(x) = \frac{1}{2}\langle Qx, x\rangle - \langle f_0, x\rangle\ mit\ Q: X \to X, f_0 \in X$$

und Q ist linear, stetig, symmetrisch und positiv definit. M ist ein zulässiger Preconditioner für T wenn

$$T = T_{PDRQ} = \sigma Q + \sigma^2 K^* K \text{ und } \sigma > 0$$

gilt. Dadurch entsteht die Douglas-Rachford Iterationsmethode für PDRQ wie in (4) zu sehen ist.

PDRQ Variante:

(4)

$$Zu\ lösen: \min_{x \in X} \max_{y \in domG} \langle Kx, y\rangle + (\frac{1}{2}Qx - f_0, x) - G(y)$$

$$Initialisierung: (x^0, \bar{y}^0) \in X \times Y, \sigma > 0\ Schrittgröße$$

$$T = \sigma Q + \sigma^2 K^* K, M - T \geq 0$$

$$Iteration: \begin{cases} b^k = -\sigma K^* \bar{y}^k + \sigma f_0 \\ x^{k+1} = x^k + M^{-1}(b^k - Tx^k) \\ y^{k+1} = \bar{y}^k + \sigma K x^{k+1} \\ \bar{y}^{k+1} = \bar{y}^k + (I + \sigma \partial G)^{-1}[2y^{k+1} - \bar{y}^k] - y^{k+1} \end{cases}$$

Auch die Konvergenz für beide Methoden wird in [1] beschrieben. So gilt, dass solang M ein zulässiger Preconditioner ist und ein Sattelpunkt für das Sattelpunktproblem (1) existiert, so konvergieren sowohl PDR als auch PDRQ schwach gegen diesen Sattelpunkt. Ebenfalls wird die Art der Konvergenz bewiesen. So handelt es sich bei Anwendungen von sowohl PDR als auch PDRQ, auf das Sattelpunktproblem, um eine ergodische Konvergenzrate. Ein möglicher und auch in den nachfolgenden Problemen verwendeter zulässiger Preconditioner ist die Symmetrische Gauß-Seidel Methode. Für weitere Informationen zu der Herleitung und Verwendung der preconditioned Douglas-Rachford Iterationsmethode wird auf die [1] und [5] verwiesen.

3.2 Einführung in TV und TGV Probleme

Rauschen oder Noise in Bildern sind Bildfehler im Bild die nichts mit dem eigentlichen Bildinhalt zu tun haben. Die verrauschten Pixel unterscheiden sich an dieser Stelle von dem Originalbild. Dies kann unterschiedliche Gründe haben. Um die erzeugten Bilder trotzdem mit größtmöglicher Genauigkeit bearbeiten zu können müssen diese noise Effekte entfernt oder so weit möglich herausgefiltert werden. Dafür gibt es sogenannte denoising und deblurring Verfahren. Einige Verfahren in diesem Bereich basieren auf der totalen Variation (TV). Bei TV und TGV handelt es sich um Probleme aus dem Variational Image Processing, es geht dabei um das Entfernen von Rauschen aus Bildern. Das beste Ergebnis wird durch das Finden eines Minimums einer Energiefunktion erreicht. Allerdings besteht das Problem darin, dass entweder kein Minimum oder mehrere Minima gefunden werden und dass das Rauschen das Ergebnis unvorhergesehen beeinflusst. Verfahren die Totale Variation zum denoising verwenden basieren auf der Idee, dass Bilder mit überflüssigen und falschen Details eine hohe Totale Variation besitzen. Das bedeutet, dass das Integral des Gradienten über das Signal hoch ist. Basierend auf dieser Idee wird durch die Senkung der Totalen Variation des Signals eine Entfernung von unnötigen Details erreicht, bei Erhaltung aller Kanten. Dieses Prinzip wurde erstmal in [6] vorgestellt.

Die preconditioned Douglas-Rachford Iteration aus dem vorherigen Teil soll nun auf Probleme der Art TV und TGV angewendet werden. Ein typisches diskretes TV normiertes Probleme ist von der Art

$$\min_{u \in U} F(u) + \alpha TV(u) = \min_{u \in U} F(u) + \alpha \|\nabla u\|_1.$$

Dabei handelt es sich bei U um einen diskreten Bildraum und bei ∇ um den diskreten Gradienten. Bei $\|\cdot\|_1$ handelt es sich um eine beliebige Norm. Ein zu bearbeitendes Bild wird über die Menge der Farbwerte alle Pixel dargestellt. Diese werden dafür in einem diskreten Zellgatter mit den Eigenschaften

$$\Omega \subset \mathbb{Z}^2, \Omega = \{(i,j) | i,j \in \mathbb{N}, 0 \leq i \leq N_x - 1, 0 \leq j \leq N_y - 1\}$$

gespeichert. Hierbei sind N_x, N_y die Grenzen des Gatters in x und y Richtung. Der Fall $N_x < 3$ und $N_y < 3$ wird ausgeschlossen, da dieser ein Sonderfall ist und getrennt behandelt werden muss. Der Bildraum U wird definiert als

$$U = \{u : \Omega \rightarrow \mathbb{R}\}$$

Der verwendete Gradient hingegen wird definiert als

$$(\nabla u) = \begin{pmatrix} \partial_x^+ u \\ \partial_y^+ u \end{pmatrix}$$

Der dafür notwendige Vorwärtsdifferenzenquotient ist definiert als

$$(\partial_x^+ u)_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j}, & falls\ 0 \leq i < N_x - 1 \\ 0, & falls\ i = N_x - 1 \end{cases}$$

$$(\partial_y^+ u)_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j}, & falls\ 0 \leq j < N_y - 1 \\ 0, & falls\ j = N_y - 1 \end{cases}$$

Das Ergebnis der Gradienten Operation ist nun nicht mehr von der Art U, sondern von der Art $V = U^2$ und somit gilt $\nabla: U \rightarrow V$. Die Divergenz ist das negative Adjunkt des Gradienten und ist somit die Abbildung $div : V \rightarrow U$ die

$$\langle \nabla u, p \rangle_V = \langle u, \nabla^* p \rangle_U = -\langle u, div\, p \rangle_U, \qquad \forall u \in U, p \in V$$

erfüllt. Die Divergenz kann somit auch geschrieben werden als

$$div\, p = \partial_x^- p^1 + \partial_y^- p^2,$$

wobei p^1, p^2 das erste und zweite Skalar des Vektors p sind. Der für die Divergenz notwendige Rückwärtsdifferenzenquotient wird definiert als

$$(\partial_x^- u)_{i,j} = \begin{cases} u_{0,j}, & falls\ i = 0 \\ u_{i,j} - u_{i-1,j}, & falls\ 0 < i < N_x - 1 \\ -u_{N_x-1,j}, & falls\ i = N_x - 1 \end{cases}$$

$$(\partial_x^- u)_{i,j} = \begin{cases} u_{0,j}, & falls\ i = 0 \\ u_{i,j} - u_{i-1,j}, & falls\ 0 < i < N_x - 1 \\ -u_{N_x-1,j}, & falls\ i = N_x - 1 \end{cases}$$

Im weiteren werden für die Verwendung der TV und TGV Methoden die L^1 , L^2 und L^∞ Normen benötigt. Diese werden definiert als

$$\|u\|_t = \Big(\sum_{(i,j)\in\Omega} \big|u_{i,j}\big|^t\Big)^{1/t}, \|u\|_\infty = \max_{(i,j)\in\Omega} \big|u_{i,j}\big|$$

$$\|p\|_t = \Big(\sum_{(i,j)\in\Omega} \big(\big(p_{i,j}^1\big)^2 + (p_{i,j}^2)^2\big)^{t/2}\Big)^{1/t}$$

$$\|p\|_\infty = \max_{(i,j)\in\Omega} \sqrt{(p_{i,j}^1)^2 + (p_{i,j}^2)^2}$$

$$\text{mit } u \in U, p = (p^1, p^2) \in V, 1 \leq t \leq \infty$$

3.2.1 TV normierte denoising Probleme

Die bisherigen Definitionen sollen nun auf das TV denoising Problem für L^1 und L^2 angewendet werden.

$$\min_{u\in U} F(u) + \alpha\|\nabla u\|_1$$

$$F(u) = \begin{cases} \frac{1}{2}\|u-f\|_2^2 \, für\, L2-denoising \\ \|u-f\|_1 \, für\, L1-denoising \end{cases}$$

Hierbei ist f das Eingabebild, somit die verrauschte Variante eines Originalbildes, und definiert als $f : \Omega \to \mathbb{R}$, während es sich bei $\alpha > 0$ um einen normierenden Parameter handelt. Da das verwendete Gatter begrenzt ist in der Anzahl seiner Zellen, handelt es sich sowohl bei F als auch bei $\alpha\|\cdot\|_1$ um stetige Funktionen. Deshalb kann das Schema für die Fenchel-Rockafellar Dualität(2) aus dem vorherigen Teil angewendet werden um ein Sattelpunktproblem(1) zu erhalten. Dafür gelten die Zuweisungen aus

$$X = U, Y = V, \mathcal{F} = F, K = \nabla \, und \, \mathcal{G} = \mathrm{I}_{\{\|v\|_\infty \leq \alpha\}}$$

sowie

$$I_C(x) = \begin{cases} 0 \, falls\, x \in C \\ \infty \quad sonst \end{cases}.$$

Mit diesen kann die preconditioned Douglas-Rachford Methode auf das TV Problem angewendet werden. Wie in [5] gezeigt muss für den L^2 Fall die PDRQ Variante angewendet werden. Dafür gilt

$$K = \nabla, Q = I, f_0 = f, \mathcal{G} = \mathrm{I}_{\{\|p\|_\infty \leq \alpha\}}$$

Die dafür verwendete Resolvente ist deshalb

(5)

$$\mathcal{P}_\alpha(p) = \frac{p}{\max(1, \frac{|p|}{\alpha})}$$

$$\text{mit } |p| = \sqrt{(p^1)^2 + (p^2)^2}$$

Im Falle von L^1 wiederrum kann die Variante PDR angewendet werden. Vorteilhaft ist es dafür einen Faktor $\tau > 0$ zu verwenden. Deshalb gilt

$$K = \tau\nabla, \mathcal{F} = \|-f\|_1, \mathcal{G} = \mathrm{I}_{\{\|p\|_\infty \leq \alpha/\tau\}}$$

und die Resolvente für ∂F ist

$$S_\sigma(u, f) = f + sign(u - f) \cdot \max(0, |u - f| - \sigma)$$

Für die Resolvente von ∂G gilt hingegen weiterhin die Definition von (5).

Als Preconditioner für sowohl L^1 als auch L^2 wird ein Rot-Schwarz-Gauß-Seidel Schema verwendet. Dabei sind die Positionen der roten und schwarzen Zellen im Gatter definiert durch

$$\Omega_{rot} = \{(i,j) \in \Omega \mid \mathrm{i} + \mathrm{j}\ \text{gerade}\}$$

$$\Omega_{schwarz} = \{(i,j) \in \Omega \mid \mathrm{i} + \mathrm{j}\ \text{ungerade}\}$$

Für das Gauß-Seidel Schema werden die neuen Zellwerte durch einen Fünf-Punkt-Stencil berechnet. Da alle Nachbarn der jeweils zu bearbeitenden Zelle von der anderen Farbe sind können alle Zellen, einer Farbe, parallel zueinander bearbeitet werden. Die gesamte Methode wird wie in (6) beschrieben durchgeführt.

(6)

SRBGS Methode:

$$\begin{cases} SRBGS^n_{\lambda,\mu}\left(u^k, b^k\right) = \left(u^{k+1}_{rot}, u^{k+1}_{schwarz}\right) \\ \left(u^{k+(v+\frac{1}{2})/n}_{rot}\right)_{i,j} = \dfrac{1}{\lambda + c_{i,j}\mu}\left(b^k_{i,j} + \mu \sum_{(i',j')\in N(i,j)} \left(u^{k+\frac{v}{n}}_{schwarz}\right)_{i',j'}\right) mit\ (i,j) \in \Omega_{rot} \\ \left(u^{k+(v+1)/n}_{schwarz}\right)_{i,j} = \dfrac{1}{\lambda + c_{i,j}\mu}\left(b^k_{i,j} + \mu \sum_{(i',j')\in N(i,j)} \left(u^{k+\frac{v+\frac{1}{2}}{n}}_{rot}\right)_{i',j'}\right) mit (i,j) \in \Omega_{schwarz} \\ \left(u^{k+1}_{rot}\right)_{i,j} = \dfrac{1}{\lambda + c_{i,j}\mu}\left(b^k_{i,j} + \mu \sum_{(i',j')\in N(i,j)} \left(u^{k+1}_{schwarz}\right)_{i',j'}\right) mit\ (i,j) \in \Omega_{rot} \end{cases}$$

Dabei unterscheidet sich das zentrale Element $c_{i,j}$ abhängig davon an welcher Position sich die Zelle befindet wie in (Abbildung 2) gezeigt.

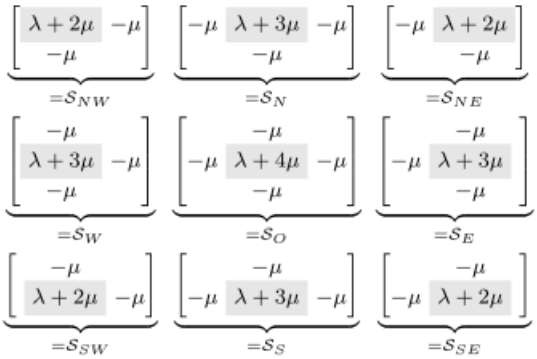

Abbildung 2 TV zentrales Element

Alle Werte außerhalb von Ω sind mit 0 belegt und verändern somit das Ergebnis nicht. Somit entstehen, für L2-TV denoising durch Verwendung von PDRQ, und für L1-TV denoising mit Verwendung von PDR zwei verschiedene Algorithmen die im Folgenden beschrieben werden.

Für L2-TV denoising mit PDRQ:

(7)

$$Ziel\ L^2 - TV\ denoising\ \min_{u\in U}\frac{1}{2}\|u-f\|_2^2 + \alpha\|\nabla u\|_1$$

$$Initialisierung:\ (u^0,\bar{p}^0)\in U\times V, \sigma>0\ Schrittgröße$$

$$n\geq 1\ innere\ Iterationen\ von\ Gauss-Seidel$$

$$Iterationen:\quad \begin{aligned} u^{k+1} &= SRBGS^n_{\sigma,\sigma^2}(u^k,\sigma(f+div\,\bar{p}^k))\\ p^{k+1} &= \bar{p}^k + \sigma\nabla u^{k+1}\\ p^{k+1}_{test} &= \mathcal{P}_\alpha(2p^{k+1}-\bar{p}^k)\\ \bar{p}^{k+1} &= \bar{p}^k + p^{k+1}_{test} - p^{k+1}\end{aligned}$$

Für L1-TV denoising mit PDR :

(8)

$$Ziel\ L^1 - TV\ denoising\ \min_{u\in U}\|u-f\|_1 + \alpha\|\nabla u\|_1$$

$$Initialisierung:\ (u^0,\bar{u}^0,\bar{p}^0)\in U\times U\times V, \sigma>0\ Schrittgröße, \tau>0$$

$$n\geq 1\ innere\ Iterationen\ von\ Gauss-Seidel$$

$$Iterationen:\quad \begin{aligned} u^{k+1} &= SRBGS^n_{1,(\sigma\tau)^2}(u^k,\bar{u}^k+\sigma\tau\,div\bar{p}^k)\\ p^{k+1} &= \bar{p}^k + \tau\sigma\nabla u^{k+1}\\ u^{k+1}_{test} &= S_\sigma(2u^{k+1}-\bar{u}^k)\\ \bar{u}^{k+1} &= \bar{u}^k + u^{k+1}_{test} - u^{k+1}\\ p^{k+1}_{test} &= \mathcal{P}_{\alpha/\tau}(2p^{k+1}-\bar{p}^k)\\ \bar{p}^{k+1} &= \bar{p}^k + p^{k+1}_{test} - p^{k+1}\end{aligned}$$

3.2.2 TGV normierte denoising Probleme

Das Prinzip von TGV wurde in [7] eingeführt. Es handelt sich dabei um eine Funktion die auf TV aufbaut und ähnliche Eigenschaften besitzt. Genau wie TV besitzt TGV die Möglichkeit Kanten zu erkennen und Bildfehler herauszufiltern. Hinzu kommt, dass anders als TV, bei TGV der staircasing Effekt signifikant reduziert wird. TGV wird definiert durch :

$\Omega \subset \mathbb{R}^d$ ist eine gebundene Lipschitz Domäne

$\alpha = (\alpha_0, \alpha_1) > 0$ sind normierungs Parameter

dann ist die Total Generalized Variation der zweiten Ordnung definiert durch

$$TGV_\alpha^2(u) = \sup\left\{\int_\Omega u\, div^2 q\, dx \mid q \in C_c^2(\Omega, \mathrm{S}^{\mathrm{d}\times\mathrm{d}}), \|\mathrm{q}\|_\infty \leq \alpha_0, \|\mathrm{div\ q}\|_\infty \leq \alpha_1\right\}.$$

Für die Anwendung der preconditioned Douglas-Rachford Iteration auf TGV ist von entscheidender Bedeutung, dass TGV auch als Minimum ausgedrückt werden kann, was in [8] gezeigt wird. Daraus entsteht:

$$TGV_\alpha^2(u) = \min_{w \in BD(\Omega)} \alpha_1 \|\nabla u - w\|_{\mathcal{M}} + \alpha_0 \|\mathcal{E}\mathrm{w}\|_{\mathcal{M}}$$

Für die verwendete Methode haben nur die diskreten denoising Probleme der Art

$$\min_{u \in U} F(u) + \alpha_1 \|\nabla u - w\|_1 + \alpha_0 \|\mathcal{E}\mathrm{w}\|_1$$

$$F(u) = \begin{cases} \frac{1}{2}\|u - f\|_2^2 \ f\ddot{\mathrm{u}}\mathrm{r}\ L2 - denoising \\ \|u - f\|_1 \ f\ddot{\mathrm{u}}\mathrm{r}\ L1 - denoising \end{cases}$$

eine Bedeutung.

Es gelten auch weiterhin die von TV bekannten Vorwärts- und Rückwärtsdifferenzenquotienten. Zusätzlich zu den schon von TV bekannten Definitionen von V und U wird nun auch der Raum W benötigt. Sie sind definiert als

$$U = \{u : \Omega \to \mathrm{R}\}, V = \{u : \Omega \to \mathrm{R}^2\}, W = \{u : \Omega \to \mathrm{S}^{2\times 2}\}$$

und somit gilt $q \in W\ mit\ q = (q^1, q^2, q^3) \in U^3$.

Der symmetrische Differenzenquotient $\mathcal{E}$ wird definiert als

$$\mathcal{E}w = \begin{bmatrix} \partial_x^+ w^1 \\ \partial_y^+ w^2 \\ \frac{1}{2}(\partial_y^+ w^1 + \partial_x^+ w^2) \end{bmatrix}.$$

Die Divergenz von Vektoren der Art q ist definiert als

$$div\ q = \begin{bmatrix} \partial_x^- q^1 + \partial_y^- q^3 \\ \partial_x^- q^3 + \partial_y^- q^2 \end{bmatrix}$$

Zusätzlich wird eine Definition für die Normen benötigt.

$$\|q\|_t = \Big(\sum_{(i,j)\in\Omega} \left(\left(q_{i,j}^1\right)^2 + \left(q_{i,j}^2\right)^2 + (q_{i,j}^3)^2 \right)^{t/2} \Big)^{1/t}$$

$$\|q\|_\infty = \max_{(i,j)\in\Omega} \sqrt{\left(q_{i,j}^1\right)^2 + \left(q_{i,j}^2\right)^2 + \left(q_{i,j}^3\right)^2}$$

$$\text{mit } 1 \leq t < \infty, q \in W$$

Mit diesen Informationen kann genau wie zuvor bei TV das Minimierungsproblem als Sattelpunktproblem dargestellt werden.

$$\min_{u\in U, w\in V} \max_{p\in V, q\in W} \langle \nabla u - w, p \rangle v + \langle \mathcal{E} w, q \rangle w + F(u)$$

Als verwendete Resolventen werden

$$\mathcal{P}_{\alpha 1}(p) = \frac{p}{\max\left(1, \frac{|p|}{\alpha_1}\right)}$$

und

$$\mathcal{P}_{\alpha 0}(q) = \frac{q}{\max\left(1, \frac{|q|}{\alpha_0}\right)}$$

hergeleitet mit $|p| = \sqrt{(p^1)^2 + (p^2)^2}$ und $|q| = \sqrt{(q^1)^2 + (q^2)^2 + 2(q^3)^2}$ für $p \in V$ und $q \in W$.

Als Preconditioner wird auch bei der Umsetzung von TGV eine Rot-Schwarz-Gauß-Seidel Methode gewählt. Allerdings verändert sich nun abhängig von der Position der Zell nicht mehr nur das zentrale Element, sondern eine zentrale 3x3 Matrix wie in (Abbildung 3) gezeigt.

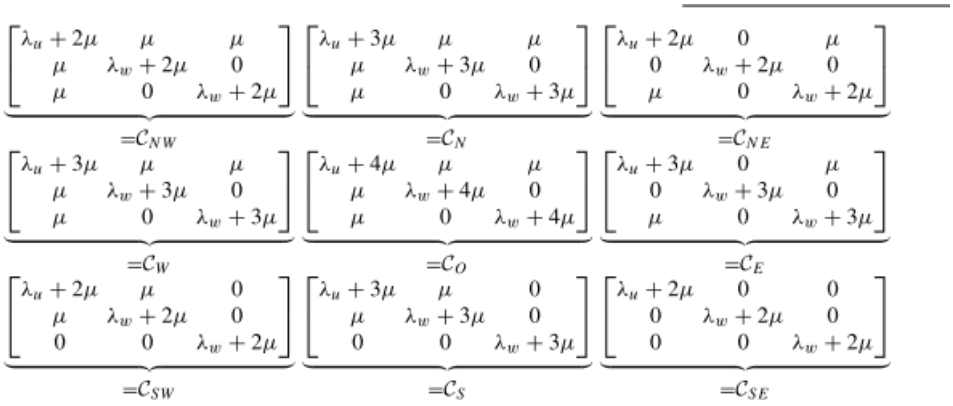

Abbildung 3 TGV zentrales Element

Außerdem verändert sich die Einbeziehung der Nachbarn. Da nun an jeder Position drei Werte gespeichert werden entscheidet die Position des Nachbarn zur aktuellen Zelle welche dieser drei Werte einbezogen werden. Dies wird dargestellt in (Abbildung 4).

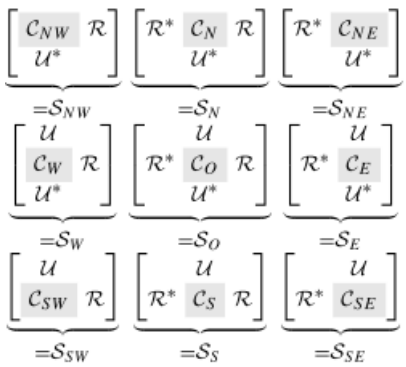

Abbildung 4 Block Stencil TGV

Dabei handelt es sich bei R und U um die 3x3 Matrizen die die Einbeziehung der Nachbarn beschreiben. R* sowie U* sind die adjungierten Matrizen von R und U. R,R*,U und U* sind in (9) definiert.

(9)

$$R = \begin{bmatrix} -\mu & 0 & 0 \\ -\mu & -\mu & 0 \\ 0 & 0 & -\mu \end{bmatrix},\ R^* = \begin{bmatrix} -\mu & -\mu & 0 \\ 0 & -\mu & 0 \\ 0 & 0 & -\mu \end{bmatrix},\ U = \begin{bmatrix} -\mu & 0 & -\mu \\ 0 & -\mu & 0 \\ 0 & 0 & -\mu \end{bmatrix},\ U^* = \begin{bmatrix} -\mu & 0 & 0 \\ 0 & -\mu & 0 \\ -\mu & 0 & -\mu \end{bmatrix}$$

Mit bekannten Matrizen R und U sowie mit den berechenbaren Inversen Matrizen des Zentralen Elementes kann die verwendete blocksymmetrische Rot-Schwarze-Gauß-Seidel Methode wie folgt beschrieben werden.

BSRBGS Methode:

(10)

$$\begin{cases} BSRBGS^{n}_{\lambda u,\lambda w,\mu}\left(x^{k},b^{k}\right)=\left(x^{k+1}_{rot},x^{k+1}_{schwarz}\right) \\ \left(x^{k+(v+\frac{1}{2})/n}_{rot}\right)_{i,j}=\ C^{-1}_{i,j}\left(b_{i,j}-\ R^{*}(x^{k+\frac{v}{n}}_{schwarz})_{i-1,j}-R(x^{k+\frac{v}{n}}_{schwarz})_{i+1,j}-\ U(x^{k+\frac{v}{n}}_{schwarz})_{i,j-1}-\ U^{*}(x^{k+\frac{v}{n}}_{schwarz})_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\mathrm{rot}} \\ \left(x^{k+(v+1)/n}_{schwarz}\right)_{i,j}=\ C^{-1}_{i,j}\left(b_{i,j}-\ R^{*}(x^{k+\frac{v+1/2}{n}}_{rot})_{i-1,j}-R(x^{k+\frac{v+1/2}{n}}_{rot})_{i+1,j}-\ U(x^{k+\frac{v+1/2}{n}}_{rot})_{i,j-1}-\ U^{*}(x^{k+\frac{v+1/2}{n}}_{rot})_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\mathrm{schwarz}} \\ \left(x^{k+1}_{rot}\right)_{i,j}=\ C^{-1}_{i,j}\left(b_{i,j}-\ R^{*}(x^{k+1}_{schwarz})_{i-1,j}-R(x^{k+1}_{schwarz})_{i+1,j}-\ U(x^{k+1}_{schwarz})_{i,j-1}-\ U^{*}(x^{k+1}_{schwarz})_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\mathrm{rot}} \end{cases}$$

Dabei ist $C^{-1}_{i,j}$ das inverse zentrale Element und $b_{i,j}=((b_u)_{i,j},(b^1_w)_{i,j},(b^2_w)_{i,j})$ wird in wie in (11) bzw. (12) initialisiert.

Mit der BSRBGS Methode als Preconditioner können nun die PDRQ Methode für L2-TGV(11) und die PDR Methode für L1-TGV(12) umgesetzt werden.

Für L2-TGV:

(11)

$$Ziel\ L^2-TGV\ denoising\ \min_{u\in U}\frac{1}{2}\|u-f\|_2^2+\ TGV^2_\alpha(u)$$

$$Initialisierung:\ (u^0,w^0,\bar{p}^0,\bar{q}^0)\in U\times V\times V\times W,\sigma>0\ Schrittgröße$$

$$n\geq 1\ innere\ Iterationen\ von\ Gauss-Seidel$$

$$Iterationen:$$

$$b^k_u=\ \sigma(f+div\,\bar{p}^k)$$

$$b^k_w=\ \sigma\left(\left(\bar{p}^k+\ div\,\bar{q}^k\right)+\frac{\sigma}{2}(\nabla^- div^+-\ \Delta)w^k\right)$$

$$\left(u^{k+1},w^{k+1}\right)=BSRBGS^{n}_{\sigma,\sigma^2,\sigma^2}\left(u^k,w^k,b^k_u,b^k_w\right)$$

$$p^{k+1}=\ \bar{p}^k+\ \sigma(\nabla u^{k+1}-\ w^{k+1})$$

$$q^{k+1}=\ \bar{q}^k+\ \sigma\mathcal{E}w^{k+1}$$

$$p^{k+1}_{test}=\ \mathcal{P}_{\alpha 1}\left(2p^{k+1}-\ \bar{p}^k\right)$$

$$\bar{p}^{k+1}=\ \bar{p}^k+\ p^{k+1}_{test}-\ p^{k+1}$$

$$q^{k+1}_{test}=\ \mathcal{P}_{\alpha 0}\left(2q^{k+1}-\ \bar{q}^k\right)$$

$$\bar{q}^{k+1} = \bar{q}^k + q_{test}^{k+1} - q^{k+1}$$

Für L1-TGV

(12)

$$Ziel\ L^2 - TGV\ denoising\ \min_{u \in U} \|u - f\|_1 + TGV_\alpha^2(u)$$

$$Initialisierung: (u^0, w^0, \bar{u}^0, \bar{p}^0, \bar{q}^0) \in U \times V \times U \times V \times W, \sigma > 0\ Schrittgröße, \tau > 0$$

$$n \geq 1\ innere\ Iterationen\ von\ Gauss - Seidel$$

$$Iterationen:$$

$$b_u^k = \bar{u}^k\ \sigma\tau div\ \bar{p}^k)$$

$$b_w^k = w^k + \sigma\tau\ ((\bar{p}^k + div\ \bar{q}^k) + \frac{\sigma\tau}{2}(\nabla^- div^+ - \Delta) w^k)$$

$$(u^{k+1}, w^{k+1}) = BSRBGS_{1,1+(\sigma\tau)^2,(\sigma\tau)^2}^n (u^k, w^k, b_u^k, b_w^k)$$

$$p^{k+1} = \bar{p}^k + \sigma\tau(\nabla u^{k+1} - w^{k+1})$$

$$q^{k+1} = \bar{q}^k + \sigma\tau\mathcal{E} w^{k+1}$$

$$u_{test}^{k+1} = S_\sigma(2u^{k+1} - \bar{u}^k)$$

$$\bar{u}^{k+1} = \bar{u}^k + u_{test}^{k+1} - u^{k+1}$$

$$p_{test}^{k+1} = \mathcal{P}_{\alpha 1/\tau}(2p^{k+1} - \bar{p}^k)$$

$$\bar{p}^{k+1} = \bar{p}^k + p_{test}^{k+1} - p^{k+1}$$

$$q_{test}^{k+1} = \mathcal{P}_{\alpha 0/\tau}(2q^{k+1} - \bar{q}^k)$$

$$\bar{q}^{k+1} = \bar{q}^k + q_{test}^{k+1} - q^{k+1}$$

Um eine Aussage über die Anzahl der benötigten Iterationen zu treffen die notwendig sind um Konvergenz zu erreichen wird für TGV nun ein Konvergenzkriterium basierend auf dem Abstand zwischen dem Minimum des primären Problems und dem Maximum des sekundären Problems gesucht(2) Für die in dieser Arbeit gesucht Lösung hat dieses Konvergenzkriterium allerdings wenig Bedeutung. Das liegt daran, dass in dieser Arbeit der Vergleich von bisherigen Implementierungen, deren Anzahl von Iteration gegeben ist, mit einer ExaSlang Implementierung im Mittelpunkt steht. Somit wird in allen Testfällen statt einem Konvergenzkriterium eine feste Anzahl von Iterationen als Abbruchbedingung verwendet.

4 Implementierung

4.1 Implementierung des L2-TGV Algorithmus

Die Implementierung des L2-TGV Algorithmus basiert auf der in (11) vorgestellten Methode aus der Veröffentlichung [1] Für die Berechnung werden Vektorfelder mit Vektoren bestehend aus einem, zwei oder drei Skalaren verwendet. Dies wird in ExaSlang durch die Verwendung von Feldern dargestellt. Ein Vektorfeld mit Vektoren bestehend aus drei Skalaren wird somit zum Beispiel durch drei voneinander getrennte Felder dargestellt die jeweils den Wert eines Skalar für jede Zellposition speichern.

Die Vektorfelder $u, b_u, f, \epsilon\, U$ besitzen Vektoren bestehend aus einem einzelnen Skalar, die Vektorfelder $p, \bar{p}, p_{test}, w, b_w\, \epsilon\, V$ besitzen Vektoren bestehend aus zwei Skalaren und die Vektorfelder $q, \bar{q}, q_{test}, b\ \in W$ bestehen aus drei Skalaren. Somit wird ein einzelnes Layout für die Felder verwendet welches ein Skalar repräsentiert. Alle Vektorfelder werden am Anfang mit 0 als Wert initialisiert. Bei $\sigma, \alpha_1, \alpha_0$ handelt es sich um globale Konstanten die als Parameter abhängig vom verwendeten Testfall sind.

Zu Beginn werden die zu bearbeitenden Bildwerte in das Feld f eingelesen. Dabei ist das Eingabebild ein Graubild mit Werten von 0 für die Farbe Schwarz bis zu 255 für die Farbe Weiß. Die Anzahl der Zellen des Feldes entsprechen dafür der Anzahl der Pixel des Bildes und muss in den meisten Fällen im verwendeten Layout des Feldes festgelegt werden. Hinzukommt, dass beim einlesen des Bildes die Position des Koordinatenursprungs verschoben wird. Während in der Bildverarbeitung häufig ein Koordinatensystem mit dem Koordinatenursprung in der linken oberen Ecke verwendet wird, welches von links nach rechts und oben nach unten durchlaufen wird, wird in ExaSlang ein kartesisches Koordinatensystem verwendet. Hier ist der Koordinatenursprung in der linken unteren Ecke und das Bild wird von links nach rechts und unten nach oben durchlaufen. Die Reihenfolge der Bildpunkte nach dem einlesen in die Zellen des Feldes ist also die gleiche aber das Bild wurde horizontal gespiegelt.

Im Laufe des Algorithmus werden nun die vorher definierten Funktionen verwendet. Der Gradient wird definiert als

$$\nabla u = \begin{pmatrix} u_x^+ \\ u_y^+ \end{pmatrix} mit\ u\ \in U$$

mit dem Vorwärtsdifferenzquotient

(13)

$$(\partial_x^+ u)_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j}, & falls\ 0 \le i < N_x - 1 \\ 0, & falls\ i = N_x - 1 \end{cases}$$

$$(\partial_y^+ u)_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j}, & falls\ 0 \le j < N_y - 1 \\ 0, & falls\ j = N_y - 1 \end{cases}$$

Die Divergenz für Vektorräume mit zwei Skalaren wird definiert als

$$div\ p = \partial_x^- p^1 + \partial_y^- p^2\ , mit\ p \in V$$

und dem Rückwärtsdifferenzquotient der definiert ist als

(14)

$$(\partial_x^- u)_{i,j} = \begin{cases} u_{0,j}, & falls\ i = 0 \\ u_{i,j} - u_{i-1,j}, & falls\ 0 < i < N_x - 1 \\ -u_{N_x-1,j}, & falls\ i = N_x - 1 \end{cases}$$

$$(\partial_x^- u)_{i,j} = \begin{cases} u_{0,j}, & falls\ i = 0 \\ u_{i,j} - u_{i-1,j}, & falls\ 0 < i < N_x - 1 \\ -u_{N_x-1,j}, & falls\ i = N_x - 1 \end{cases}$$

Die Divergenz für Vektorräume mit drei Skalaren wird definiert als

$$div\ q = \begin{bmatrix} \partial_x^- q^1 + \partial_y^- q^3 \\ \partial_x^- q^3 + \partial_y^- q^2 \end{bmatrix}, mit\ q \in W.$$

$\mathcal{E}w$ wurde definiert als

$$\mathcal{E}w = \begin{bmatrix} \partial_x^+ w^1 \\ \partial_y^+ w^2 \\ \frac{1}{2}\left(\partial_y^+ w^1 + \partial_x^+ w^2\right) \end{bmatrix}, mit\ w \in V.$$

$\mathcal{P}_{\alpha_1}(p)$ ist definiert als

$$\mathcal{P}_{\alpha_1}(p) = \frac{p}{\max\left(1, \frac{|p|}{\alpha_1}\right)}, mit\ p \epsilon V\ und\ |p| = \sqrt{(p^1)^2 + (p^2)^2}$$

$\mathcal{P}_{\alpha_0}(q)$ ist definiert als

$$\mathcal{P}_{\alpha_0}(q) = \frac{q}{\max\left(1, \frac{|q|}{\alpha_0}\right)}, mit\ q \epsilon W\ und\ |q| = \sqrt{(q^1)^2 + (q^2)^2 + 2(q^3)^2}$$

Außerdem wird der Laplace Operator, der als $\Delta = div\,\nabla$ definiert ist, auf w als $\Delta w = (\Delta w^1, \Delta w^2)$ definiert. Zusätzlich wird definiert, dass ∇^- der diskrete Gradient mithilfe des Rückwärtsdifferenzquotienten ist sowie div^+die diskrete Divergenz mithilfe des Vorwärtsdifferenzquotienten ist.

Die Differenzenquotienten werden dabei in ExaSlang als Stencil implementiert. So wird der Vorwärtsdifferenzenquotient in x Richtung repräsentiert durch den Stencil

```
Stencil forwardDifferenceX@all
{
    [0,0] => (-1.0)
    [1,0] => ( 1.0)
}
```

Für die Verwendung der Differenzenquotienten spielt es keine Rolle, dass das Bild beim einlesen horizontal gespiegelt wurde. Das liegt daran, dass es bei der Berechnung der Differenzenquotienten nicht auf die Position der Zellen zueinander ankommt sondern auf die Reihenfolge. Allerdings ist laut der Definition für den Vorwärtsdifferenzenquotienten in (13) eine Randbehandlung notwendig. Damit an den Rändern $i = N_X - 1$ beziehungsweise $j = Ny - 1$ jeweils 0 herauskommt muss das Ghost Layer an dieser Stelle um den Wert von $u_{i,j}$ erweitert werden. Deshalb muss bei allen Feldern auf die ein Vorwärtsdifferenzenquotient angewendet wird eine Neumann Grenzbehandlung durchgeführt werden und nach einer Werteänderung dieses Feldes die *apply bc to()* Funktion aufgerufen werden.

```
Stencil backwardDifferenceX@all
{
    [ 0,0] => ( 1.0)
    [-1,0] => (-1.0)
}
```

Der Rückwärtsdifferenzenquotient der durch den Stencil *backwardDifferenceX* repräsentiert wird und in (14) definiert ist, benötigt einen weiteren Stencil um seine Randbehandlung bei $i = N_X - 1$ und $j = Ny - 1$ zu erfüllen. Es wird bei jeder Schleife die den Rückwärtsdifferenzenquotient aufruft getestet ob sich die Zelle am rechten Rand beziehungsweise am oberen Rand oder in der rechten oberen Ecke befindet. In der Beschreibung von Bild 2.3 soll in diesem Fall $(\partial_x^- u)_{i,j} = -u_{N_x-1,j}\,, bei\; i = N_x - 1$ und $(\partial_y^- u)_{i,j} = -u_{i,N_y-1}\,, bei\; j = N_y - 1$ ausgegeben werden. Allerdings wird im vorhandenen Referenzcode zu TV, in dem der gleiche Rückwärtsdifferenzenquotient verwendet wird, stattdessen eine Implementierung der Art wie in (Abbildung 5) verwendet.

```
void dxm(double *outMatrix, double *xMatrix, int dimN, int dimM)
{
  int i, j, dimN_, ind;

  dimN_ = dimN - 1;
#pragma omp parallel for private (i, ind)
  for(j=0; j<dimM; j++) {
    ind = j*dimN;
    outMatrix[ind] = xMatrix[ind]; ind++;
    for(i=1; i<dimN_; i++, ind++)
      outMatrix[ind] = xMatrix[ind] - xMatrix[ind-1];
    outMatrix[ind] = -xMatrix[ind-1];
  }
}
```

Abbildung 5 Matlab TV Rückwärtsdifferenz

Damit wird stattdessen $(\partial_x^- u)_{i,j} = -u_{N_x-2,j}\,, bei\; i = N_x - 1$ erreicht. Im Testfall kann gezeigt werden, dass es keinen nennenswerten Unterschied bei den entstehenden Bildern gibt, weshalb die Version wie sie im Referenzcode vorhanden ist implementiert wurde. Somit werden im Falle der Randposition die Stencil *backwardDifferenceBoundaryX* und *backwardDifferenceBoundaryY* verwendet.

```
Stencil backwardDifferenceBoundaryX@all
{
    [-1,0] => (-1.0)
}

Stencil backwardDifferenceBoundaryY@all
{
    [0,-1] => (-1.0)
}
```

Die in (10)beschriebene Blocksymmetrische-Rot-Schwarz-Gauß-Seidel Funktion(BSRBGS) wird auf zwei verschiedene Arten umgesetzt, was zu zwei verschiedenen Varianten von L2-TGV führt.

(10)

$$\begin{cases} BSRBGS^{n}_{\lambda u,\lambda w,\mu}\left(x^k, b^k\right) = \left(x^{k+1}_{rot}, x^{k+1}_{schwarz}\right) \\ \left(x_{rot}^{k+(v+\frac{1}{2})/n}\right)_{i,j} = C_{i,j}^{-1}\left(b_{i,j} - R^*\left(x^{k+\frac{v}{n}}_{schwarz}\right)_{i-1,j} - R\left(x^{k+\frac{v}{n}}_{schwarz}\right)_{i+1,j} - U\left(x^{k+\frac{v}{n}}_{schwarz}\right)_{i,j-1} - U^*\left(x^{k+\frac{v}{n}}_{schwarz}\right)_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\text{rot}} \\ \left(x_{schwarz}^{k+(v+1)/n}\right)_{i,j} = C_{i,j}^{-1}\left(b_{i,j} - R^*\left(x^{k+\frac{v+1/2}{n}}_{rot}\right)_{i-1,j} - R\left(x^{k+\frac{v+1/2}{n}}_{rot}\right)_{i+1,j} - U\left(x^{k+\frac{v+1/2}{n}}_{rot}\right)_{i,j-1} - U^*\left(x^{k+\frac{v+1/2}{n}}_{rot}\right)_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\text{schwarz}} \\ \left(x_{rot}^{k+1}\right)_{i,j} = C_{i,j}^{-1}\left(b_{i,j} - R^*\left(x^{k+1}_{schwarz}\right)_{i-1,j} - R\left(x^{k+1}_{schwarz}\right)_{i+1,j} - U\left(x^{k+1}_{schwarz}\right)_{i,j-1} - U^*\left(x^{k+1}_{schwarz}\right)_{i,j+1}\right) \\ mit\ (i,j)\ \in \Omega_{\text{rot}} \end{cases}$$

Die erste Variante der BSRBGS Funktion verwendet auch hier nur einfache mit einzelnen Skalaren belegte Vektorfelder. Zusätzlich werden die verwendeten Inversen der Zentralen Elemente direkt als Konstanten verwendet. Beim Aufruf der Funktion wird ein Duplikat der Bildwerte angelegt und mit diesem weitergerechnet. Das liegt daran, dass das Feld mit den ursprünglichen Bildwerten eine Neumann Grenzbehandlung verwendet. Da in dieser ersten Version eine Auswahl des Zentralen Elementes abhängig von der Position der Zelle stattfindet ist keine Grenzbehandlung notwendig. In drei aufeinanderfolgenden Schleifen werden zuerst alle roten Elemente, dann alle schwarzen Elemente und zuletzt noch einmal alle roten Elemente berechnet. Damit in jeder Schleife nur auf Elemente der richtigen Farbe zugegriffen wird werden die Schleifen mit *where 0 == ((i0 + i1) % 2)* für rote Elemente und *where 1 == ((i0 + i1) % 2)* für schwarze Elemente aufgerufen. Wie man sehen kann, hängt das Ergebnis von Zellen einer Farbe nur von Zellen der jeweils anderen Farbe ab. Dadurch kann jeder Punkt einer Farbe undabhängig voneinander parallel verarbeitet werden. Dann wird zuerst die Klammer mithilfe von Stencil wie *rightStencil* ,für den rechtsseitigen Nachbarn, auf Basis von R,U,R*,U* gelesen . Das Ergebnis wird mit dem zusammengesetzten Vektor b addiert.

```
Stencil rightStencil@all
{
    [1,0] => (-1.0 * mu)
}
```

Dann wird die Position der aktuellen Zelle ermittelt, ob sie sich am Rand an einer Ecke oder im Inneren befindet. Mithilfe dieser Position wird das Inverse des Zentralen Elementes gewählt, aus (Abbildung 3) und eine Matrixmultiplikation von diesem und dem Ergebnis der Klammer ausgerechnet. Zuletzt wird das Vektorfeld, welches zu Beginn dupliziert wurde, aktualisiert. Die Implementierung mit dieser Variante der BSRBGS Funktion wird im Folgenden als L2-TGV ohne VC bezeichnet.

Die zweite Variante hingegen verwendet Felder mit dem *flowLayout*.

```
Layout flowLayout < ColumnVector<Real, 3>, Cell> @all {
    ghostLayers = [ 1, 1 ]
    duplicateLayers = [ 0, 0 ]

}
```

Dieses Layout ermöglicht es, dass in einem Vektorfeld drei Skalare pro Zelle gespeichert werden. Somit kann anstatt von drei verschiedenen Feldern ein einzelnes verwendet werden. Durch die Anwendung eines einzelnen Feldes kann auch mit einem einzigen Stencil *bsrbgsStencil* der Inhalt der Klammer berechnet werden, die den Block Stencil (Abbildung 4) repräsentiert.

```
Stencil bsrbgsStencil@all
{
    [ 0,0] => {{ 0.0, 0.0, 0.0}, {0.0, 0.0, 0.0}, { 0.0, 0.0, 0.0}}
    [ 1,0] => {{-1.0 *mu, 0.0, 0.0}, {-1.0*mu,-1.0*mu, 0.0}, {0.0, 0.0, -1.0*mu}} //R
    [-1,0] => {{-1.0*mu, -1.0*mu, 0.0}, {0.0,-1.0*mu, 0.0}, {0.0, 0.0, -1.0*mu}} //R*
    [ 0, 1] => {{-1.0*mu, 0.0, 0.0}, {0.0,-1.0*mu, 0.0}, {-1.0*mu, 0.0, -1.0*mu}} //U*
    [ 0,-1] => {{-1.0*mu, 0.0, -1.0*mu}, {0.0,-1.0*mu, 0.0}, {0.0, 0.0, -1.0*mu}} //U

}
```

Auch hier, genau wie in der ersten Variante, wird abhängig von der Position der Zelle das zentrale Element gewählt. Allerdings wird das Inverse in dieser Variante dynamisch, abhängig von $\lambda_u, \lambda_w, \mu$, berechnet. Damit werden die aktualisierten roten und schwarzen Werte berechnet. Allerdings gibt es bisher in ExaSlang noch nicht die Möglichkeit direkt auf einen Wert des *flowLayouts* zuzugreifen. Um sie trotzdem auslesen zu können wird eine Skalarmultiplikation des Feldes mit den drei Einheitsvektoren e_x, e_y, e_z durchgeführt. Die Implementierung mit dieser Variante der BSRBGS Funktion wird im Folgenden als L2-TGV mit VC bezeichnet.

Der Vorteil dieser Version ist, dass mit der Anpassung der Parameter in den Globalen Variablen die inversen zentralen Elemente berechnet werden, anstatt sie als Konstante manuell anzupassen wie in der ersten Variante notwendig ist. Allerdings werden die Inversen bei jeder Iteration neu berechnet. Das kann vor allem bei einer hohen Anzahl von Iterationen zu einem spürbaren Geschwindigkeitsverlust führen.

In der Beschreibung des Algorithmus von L2-TGV (11) wird

$$p_{test} = \mathcal{P}_{\alpha_1}(2p - \bar{p}) \text{ , } \bar{p}^{k+1} = \bar{p}^k + p_{test}^{k+1} + p^{k+1}$$

sowie

$$q_{test} = \mathcal{P}_{\alpha_1}(2q - \bar{q}), \bar{q}^{k+1} = q^k + q_{test}^{k+1} + q^{k+1}$$

verwendet. Wenn der Algorithmus auf diese Art und Weise implementiert wird, ist das Ergebnisbild zum Zeitpunkt der Konvergenz allerdings nahezu dem Eingabebild entsprechend. Es kommt zu keiner Verringerung des Rauschens. In den vorhandenen Beispielimplementierungen des TV Algorithmus, in dem die gleichen Rechnungen vorkommen, wird stattdessen

$$p_{test} = \mathcal{P}_{\alpha_1}(2p + \bar{p}) \text{ , } \bar{p}^{k+1} = p_{test}^{k+1} + p^{k+1}$$

sowie

$$q_{test} = \mathcal{P}_{\alpha_1}(2q + \bar{q}), \bar{q}^{k+1} = q_{test}^{k+1} + q^{k+1}$$

verwendet. Dies führt im Testfall zu deutlich besseren Ergebnissen und wird deshalb auch hier weiterhin so verwendet.

4.2 Implementierung des L1-TGV Algorithmus

Die Implementierung des L1-TGV Algorithmus basiert zu einem überwiegenden Teil auf der Implementierung des L2-TGV Algorithmus. Er wird in der Veröffentlichung wie in (12) beschrieben. Da die zwei Verschiedenen Varianten der Implementierung bei L2-TGV verglichen werden ist dies bei L1-TGV nicht noch einmal notwendig, da der BSRBGS Teil, der die beiden Varianten unterscheidet, der selbe ist. Die Implementierung von L1-TGV basiert auf L2-TGV mit VC.

Hinzu kommt allerdings die Verwendung eines Parameters τ sowie der zwei Variablen $\bar{u}$ und u_{test}. Ansonsten werden aus L2-TGV bekannte Funktionen verwendet nur S_σ kommt hinzu und ist definiert als

$$S_\sigma(u, f) = f + sign(u - f) \cdot \max(0, |u - f| - \sigma)$$

Somit kann basierend auf der L2-TGV Implementierung relativ leicht eine L1-TGV Implementierung umgesetzt werden.

5 Berechnete Testfälle und Auswertung

Im Folgenden werden die zwei verschiedenen Implementierungen von L2-TGV sowie die Implementierung von L1-TGV getestet. Dabei werden die verschiedenen Implementierungen mit den Ergebnissen der Implementierungen der Veröffentlichung [1] verglichen sowie einer vorhandenen ExaSlang Implementierung des L2-TV Algorithmus. Anders als in der zu Grunde liegenden Veröffentlichung wird bei der Erzeugung von Testfällen nicht mit einem Konvergenzkriterium gearbeitet. Das entscheidende in dieser Arbeit ist der Vergleich zwischen den Implementierungen in Matlab und den automatisch leistungsoptimierten Implementierungen in ExaSlang.

Alle Tests und Ergebnisse sind dabei unter Verwendung eines Systems mit acht Kernen je 3,60GHz entstanden.

5.1 L2-TGV Testfälle und Auswertung

Der erste Test vergleicht die Ergebnisse der Experimente aus der Veröffentlichung (REF 1) zum Thema L2-TGV mit den zwei verschiedenen ExaSlang Implementierungen von L2-TGV. Um einen Test mit möglichst vergleichbaren Werten zu erzeugen wird deshalb das gleiche Originalbild (Abbildung 6) und Eingabebild (Abbildung 7) verwendet. Die Bilder besitzen die Dimensionen 512x357 Pixel und das Eingabebild besitzt 10% gaussschem Rauschen, das herausgefiltert werden soll. Die Parameter für den Test wurden ebenfalls aus der Veröffentlichung übernommen. So betragen $\sigma = 3.0$, $\alpha_1 = 0.1, \alpha_0 = 2\alpha_1$, $\lambda_u = \sigma$, $\lambda_w = \sigma^2$, $\mu = \sigma^2$. Die Anzahl der Iterationen auf die getestet wird beträgt 88 und 1778 damit diese mit den

Ergebnissen der Veröffentlichung verglichen werden können. Die Anzahl der Iterationen des Preconditioners beträgt wie bei allen weiteren Iterationen $n = 1$.

Die Ergebnisse sind (Abbildung 8) mit 88 Iterationen und (Abbildung 9) mit 1778 Iterationen für die L2-TGV Implementierung ohne Vector Column (VC) und (Abbildung 10) mit 88 Iterationen und (Abbildung 11) mit 1778 Iterationen für die L2-TGV Implementierung mit VC. Hinzukommt (Abbildung 12) für das Ergebnis der Implementierung aus der Veröffentlichung mit 1778 Iterationen.

Abbildung 6 Original 512x357

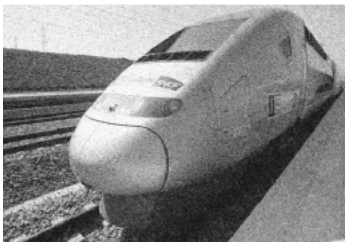

Abbildung 7 Eingabebild 512x357

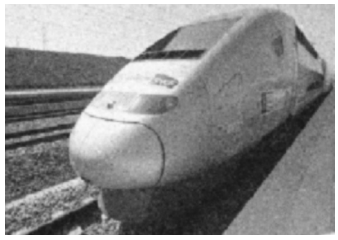

Abbildung 8 L2 ohne VC 88 Iterationen

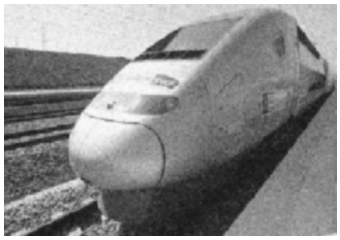

Abbildung 9 L2 ohne VC 1778 Iterationen

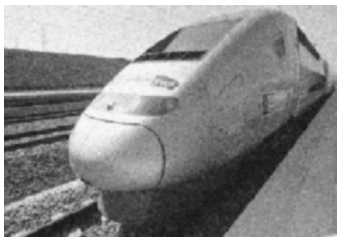

Abbildung 10 L2 mit VC 88 Iterationen

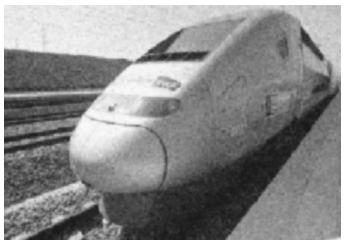

Abbildung 11 L2 mit VC 1778 Iterationen

Abbildung 12 Vergleichsbild Veröffentlichung 512x357

	ExaSlang Compiler	C++ Compiler	Ausführungszeit
mit VC	7300	26024	1127
ohne VC	6200	33891	1086
Veröffentlichung			4100

Tabelle 1 L2-TGV mit 88 Iterationen: Zeit in ms, Mittelwert aus Messergebnisse von 10 Testdurchläufen

	ExaSlang Compiler	C++ Compiler	Ausführungszeit
mit VC	7400	24665	21272
ohne VC	6100	25443	21411
Veröffentlichung			82300

Tabelle 2 L2-TGV mit 1778 Iterationen: Zeit in ms, Mittelwert aus Messergebnissen von 10 Testdurchläufen

Die angegebenen Messwerte in den Tabelle 1 und 2 wurden durch 10-faches Ausführen der Implementierungen mit VC und ohne VC und der darauf folgenden Bildung des Mittelwertes der 10 Ergebnisse.

Wie man sehen kann gleichen sich die Abbildungen 8 und Abbildung 10 nahezu. Das liegt daran, dass Beide Implementierungen, sowohl die Implementierung mit VC als auch die Implementierung ohne VC, letztendlich den gleichen Algorithmus implementieren. Sie unterscheiden sich nur in der Implementierung des Preconditioners des Rot-Schwarzen-Gauß-Seidel Algorithmus. Das gleiche gilt für die Abbildungen 9 und 11. Zwischen Abbildung 8 und Abbildung 9 sowie zwischen Abbildung 10 und Abbildung 11 besteht ebenfalls nur ein minimaler Unterschied. Das liegt daran, dass der Algorithmus nach 88 Iterationen schon ein hinreichend kleines Konvergenzkriterium erreicht hat. Es kommt also bis zu 1778 Iterationen nicht mehr zu Unterschieden die für das menschliche Auge sichtbar sind. Der Test mit 1778 Iterationen wurde somit eher als ein Vergleich für die Zeit hinzugefügt. Allerdings gibt es einen Unterschied zwischen den Ergebnissen der Implementierung mithilfe von ExaSlang und dem Ergebnis der Beispielimplementierung(Abbildung 12) aus der Veröffentlichung. In Abbildung 12 wurde das Rauschen wesentlich besser beseitigt und auch die aus der Rauschentfernung resultierende Unschärfe ist wesentlich geringer. Trotz der Verwendung des in der Veröffentlich vorgestellten Algorithmus und der Anwendung der gleichen Parameter konnten nicht die gleichen Ergebnisse erzielt werden. Es ist zwar eine deutliche Rauschbeseitigung erkenntlich aber nicht in dem Masse wie erwartet. Das liegt zum einen daran, dass die Implementierung in ExaSlang mithilfe der Beschreibung des Algorithmus aus der Veröffentlichung stattgefunden hat und auf der Matlab Umsetzung des TV Algorithmus aus der Veröffentlichung basierte. Es stand keine Matlab Umsetzung des TGV Algorithmus zum Vergleich zur Verfügung. Zum anderen ist es möglich, dass auch nach extensiver Fehlersuche weiterhin Fehler bei der Implementierung bestehen. Die somit entstandenen Bilder erfüllt somit nicht die Erwartungen im Bezug auf ihre Qualität.

Bei der Zeit hingegen ist, wie in Tabelle 1 und Tabelle 2 zu sehen ist, ein deutlicher Vorteil der Implementierung durch ExaSlang zu erkennen. So ist die Implementierung mithilfe von ExaSlang fast vier Mal so schnell, sowohl bei 88 Iterationen als auch bei 1778 Iterationen, wie die Beispielimplementierung die mithilfe von Matlab entstanden ist. Da die maximale Optimierung eines der Hauptziele von ExaSlang ist, kann man sagen, dass dieses Ziel erreicht wurde. Der Nachteil von eine Implementierung mit ExaSlang ist allerdings das einmalige zusätzliche Kompilieren in die Hostsprache C++, im Vergleich zu einer direkten Implementierung in einer Zielsprache. Mit bis zu 7400ms ist der Zeitaufwand für das Kompilieren ungefähr ein viertel des Zeitaufwandes der Zielsprache und kann somit vor allem bei Implementierungen mit geringer Iterationsmenge von Bedeutung sein, in denen der Zeitaufwand für das Kompilieren im allgemeinen einen höheren Anteil besitzt.

Bei einem weiteren Testfall wurde mit einem Bild mit den Dimensionen 512x512 Pixel gearbeitet. Bei diesem Testfall geht es im wesentlichen um den Vergleich zwischen den beiden unterschiedlichen Implementierungen von L2-TGV aber auch um den Vergleich mit verfügbaren Implementierungen von TV. Bei dem Originalbild handelt es sich um (Abbildung 13), bei dem Eingabebild um (Abbildung 14). Da das Ergebnis von sowohl L2-TGV mit VC und L2-TGV ohne VC nahezu übereinstimmt verwenden wir nur das Ergebnis von L2-TGV mit VC (Abbildung 15). Die Anzahl der Iterationen ist auf 1800 beschränkt. Zum Vergleich werden eine vorhandene Implementierung des TV Algorithmus in ExaSlang (Abbildung 16) verwendet sowie eine Matlab Implementierung des TV Algorithmus. Bei den Parametern handelt es sich wie im letzten Testfall um $\sigma = 3.0\,, \alpha_1 = 0.1, \alpha_0 = 2\alpha_1\,, \lambda_u = \sigma\,, \lambda_w = \sigma^2\,, \mu = \sigma^2$.

Abbildung 13 Original 512x512

Abbildung 14 Eingabebild 10% Rauschen 512x512

Abbildung 15 L2 TGV mit VC 512x512

Abbildung 16 TV ExaSlang 512x512

	mit VC	ohne VC	ExaSlang TV	Matlab TV
Zeit in ms	31862	33387	5950	39276

Tabelle 3 L2-TGV mit 1800 Iterationen: Ergebnisse als Mittelwerte von 10 Testläufen entstanden

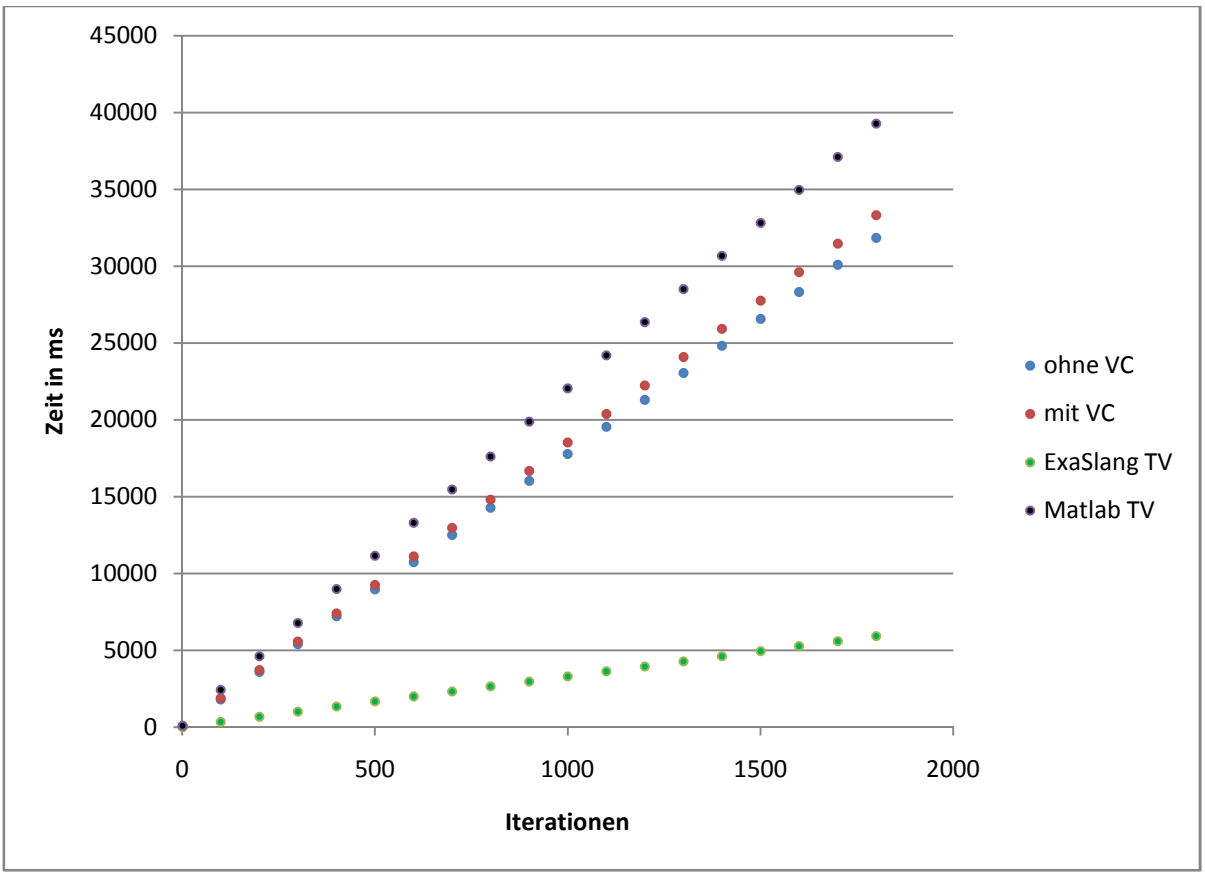

Abbildung 17Diagramm Zeit/Iterationen für L2-TGV

Wie man in Tabelle 3 und Abbildung 17 sehen kann ist der ExaSlang Algorithmus für TV wesentlich schneller als die Algorithmen für TGV. Das liegt vor allem daran, dass der Aufwand

für die Berechnung des Preconditioners bei TV wesentlich geringer ist. Das liegt am Aufbau des Preconditioners. Während eine Rot Schleife bei TV durch einmaliges Aufrufen eines Stencils berechnet wird, sowie einiger einfachen Additionen und Multiplikationen in einer Schleife, kommt es bei TGV zu mehreren Schleifendurchläufen. Hinzukommt eine Matrixmultiplikation mit dem zentralen Element. Während der Anteil für die Aktualisierung bei TGV etwa ein Drittel(10984ms / 33387ms) der gesamten Bearbeitungszeit einnimmt, kostet dieser Teil bei TV fast zwei Drittel(3682ms / 5950ms), da der Preconditioner so billig ist. Außerdem ist der Aufwand für die Aktualisierung der restlichen Variablen nach durchlaufen des Rot-Schwarz-Gauß-Seidel Blocks bei TGV aufwendiger.

Außerdem sieht man in Tabelle 3, dass die Implementierung von L2-TGV ohne VC durchgängig etwas schneller ist, als die Variante mit VC. Dieser Unterschied steigt mit wachsender Anzahl der Iterationen so(Abbildung 17) ist er bei 100 Iterationen etwa 100ms groß. Bei 1800 Iterationen hingegen beträgt der Zeitunterschied 1500ms. Dies ist den Implementierungsunterschieden der beiden Versionen geschuldet. Während es die Version mit VC ermöglicht alle verwendeten Parameter in den globalen Variablen zu ändern, erhöht dies den Aufwand bei der Berechnung des inversen zentralen Elementes. Dieses wird in jeder Iteration neu berechnet. Dadurch wächst der Aufwand mit der Anzahl der Iterationen für diese Berechnung. Bei der Version ohne VC hingegen ist eine Änderung der Parameter mit einer manuellen neuen Berechnung der verwendeten Inversen verbunden. Diese müssen in die Fallunterscheidungen für die aktuelle Position der Zelle einmalig eingetragen werden.

Es ist aber sichtbar(Abbildung 17), dass trotz des geringeren Aufwands von TV die ExaSlang Implementierung von TGV schneller ist als die Matlab Implementierung von TV. Die Implementierung von ExaSlang TV ist etwa 6,5-Fach so schnell wie die Matlab Implementierung von TV. Da keine Referenzimplementierung von TGV in Matlab zur Verfügung stand ist es nicht möglich eine genaue Aussage über die Verbesserung der Geschwindigkeit, durch die ExaSlang Implementierung, zu treffen.

In wie weit die Ausführungszeit von der Bildgröße abhängt kann gezeigt werden, indem für verschiedene Bildgrößen die gesamte Ausführungszeit, bei gleicher Anzahl Iterationen, durch die Anzahl der zu bearbeitenden Zellen geteilt wird. Wie man in Tabelle 4 sehen kann ist der Unterschied in der Ausführungszeit je Zelle gering. Die Bildgröße verändert also den Betrag der Ausführungszeit nur als einfacher Faktor.

	Gesamtzeit in ms	Gesamtzeit/Zellen in ms
512 × 357 Zellen	21272	0.1163
512 × 512 Zellen	33103	0.1262

Tabelle 4 Skalierter Aufwand 1778 Iterationen

5.2 L1-TGV Auswertung

Das Ziel des Testfalls von L1-TGV ist es die ExaSlang Implementierung mit den Ergebnissen aus der Veröffentlichung zu vergleichen. Deshalb werden soweit möglich die gleichen Eingabeparameter und das gleiche Eingabebild verwendet. Somit ist das Originalbild (Abbildung 18) ein Bild von 510×383 Pixeln. Das Eingabebild ist ein durch 30% salt-and-pepper Rauschen bearbeitetes Bild (Abbildung 19). Als Parameter für den L1-TGV Algorithmus werden $\sigma = 0.1, \tau = \frac{3}{\sigma}, \alpha_1 = 1.0,\ \alpha_0 = 2\alpha_1, \lambda_u = 1, \lambda_w = 1 + \ (\sigma\tau)^2, \mu = (\sigma\tau)^2$ gewählt. Die Anzahl der Iterationen beträgt im ersten Fall 396 und im zweiten Fall 1408. Bei dem in der Veröffentlichung gezeigten Ergebnis handelt es sich um Abbildung 20. Die aus der ExaSlang Implementierung entstandenen Bilder sind in Abbildung 21 und Abbildung 22 zu sehen.

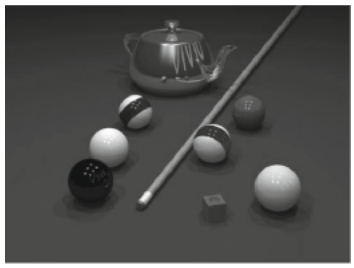

Abbildung 18 Originalbild L1-TGV

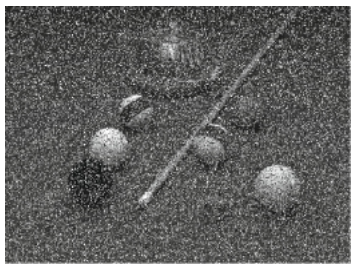

Abbildung 19 Eingabebild L1-TGV 30 % Rauschen

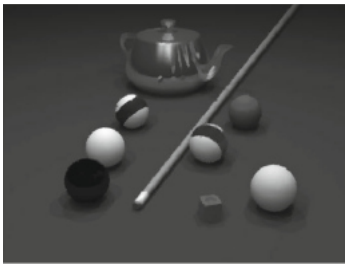

Abbildung 20 L1-TGV Veröffentlichung

Abbildung 21 L1-TGV ExaSlang 396 Iterationen

Abbildung 22 L1-TGV ExaSlang 1408 Iterationen

Wie man sehen kann handelt es sich bei den Ergebnissen der ExaSlang Implementierung um schwarze Bilder. Man muss also davon ausgehen, dass diese Implementierung nicht den Inhalt des vorgestellten L1-TGV Methode und somit der Problemspezifikation, entspricht obwohl sie nach dessen Beschreibung erstellt wurde. Die Gründe dafür sind schwer nachzuvollziehen, da keine Beispielimplementierung des L1-TGV Algorithmus vorliegt um die Implementierungen zu vergleichen. Es kann einerseits an Wiedersprüchen zwischen der Beispielimplementierung und der Beschreibung in der Veröffentlichung liegen, welche bei TV und L2-TGV aufgetreten sind, liegen oder aber auch an Fehlern bei der Implementierung der Beschreibung.

In diesem Sinne ist fragwürdig in wie weit die entstandenen Messwerte der beiden Testfälle einen Vergleich der beiden Implementierungen ermöglichen.

	ExaSlang L1-TGV	Beispiel L1-TGV
396 Iterationen	5810 ms	19460 ms
1408 Iterationen	20804 ms	69670 ms

Tabelle 5 L1-TGV : Ergebnisse als Mittelwerte von 10 Testläufen entstanden

5.3 Probleme bei der Implementierung

Bei der Implementierung der in der Veröffentlichung [1]vorgestellten TGV Algorithmen ist es zu einigen Problemen gekommen. Der Umfang der Dokumentation für ExaSlang ist aktuell sehr

begrenzt. Somit basiert die Einarbeitung in die Verwendung von ExaSlang4 im großen Maße auf den Veröffentlichungen [2] und [3] zum Thema und auf vorhandenen Implementierungen die in Codeform vorliegen, sowie auf Nachfragen bei Experten. Somit ist es schwierig sich einen Überblick über vorhandene Funktionen und verfügbare Parameter zu verschaffen. Ein weiteres Problem bei der Implementierung durch ExaSlang sind die beim Kompilieren erzeugten Protokolle. Für einfache Fehler sind sie sehr aufschlussreich und ermöglichen eine schnelle Beseitigung. Für andere Fehler sind sie wenig hilfreich. So ist die Fehlermeldung "Trying to access current outside of a valid scope", die auftritt wenn ein Feld mit "@current" ohne vorherige Definition des Levels aufgerufen wird, nur schwer zu beheben wenn keine Position des Fehlers angegeben wird. Es ist zwar nützlich um die Art des Fehlers zu erkennen aber ohne die Position ist die Suche nach dem Fehler trotzdem schwierig. Auch die Debug-Meldungen die in dem Protokoll ausgegeben werden, sind nicht auf eine Art und Weise dargestellt, dass ein Nutzer ohne Expertenwissen zu ExaSlang diese nutzen kann.

Ein weiteres Problem ist die veränderte Definition des verwendeten Gatters. In den meisten Fällen wird in der Bildverarbeitung ein System verwendet, in dem sich der Koordinatenursprung (0,0) in der linken oberen Ecke befindet. In der Mathematik hingegen ist das kartesische Koordinatensystem verbreitet. In diesem befindet sich der Ursprung links unten. ExaSlang verwendet ein kartesisches Koordinatensystem obwohl es zur Bearbeitung von Bildern verwendet wird. Das sorgt dafür, dass beim Einlesen eines Bildes die Bildwerte horizontal gespiegelt werden. Das ist dann kein Problem wenn nur die Reihenfolge der Zellen eine Rolle spielt, zum Beispiel bei der Vorwärts- und Rückwärtsdifferenz. Wenn es aber darum geht den richtigen Nachbarn auszuwählen sorgt diese Spiegelung für Verwirrung, da auch jeder verwendete Stencil horizontal gespiegelt werden muss. Hinzukommt, dass auch das Ergebnis in umgekehrte Art und Weise gespiegelt werden muss damit es sinnvoll ausgegeben werden kann. Bei Projekt bei dem die maximale Effizienz ein Zentrales Ziel ist lässt sich dies möglicherweise vermeiden.

Ein Problem des ExaSlang Compilers welches bei der Implementierung aufgetreten ist, war der mehrmalige Aufruf der Funktion inverse(diag()) auf den selben Stencil. Dies führte dazu, dass das Ergebnis erst beim letzten Aufruf berechnet wurde und alle weiteren Aufrufe auf diese Berechnung verwiesen haben. Somit haben Funktionsaufrufe die sequenziell vor der Berechnung stattfinden versucht die Lösung der Berechnung, die noch nicht vorhanden ist aufzurufen. Dieser Fehler ist in einer neueren Version des ExaSlang Compilers behoben.

Ein weiteres Problem bei der Implementierung lag daran, dass es in ExaSlang keine Möglichkeit des direkten Zugriffs auf eine bestimmte Zelle gibt. Für jede Zelle die gelesen oder geschrieben werden soll, muss das gesamte Feld in einer Schleife durchlaufen werden.

Auch die Nutzung des Layouts mit der Eigenschaft Vector Column ist bisher nicht ohne Probleme. So ist die einzige mir bekannte Möglichkeit Werte aus einem Vektor Column Feld in

ein Feld mit Real Werten zu übertragen, der Weg über das Skalarprodukt mit Einheitsvektoren. Dies sorgt für weitere Felder die definiert, initialisiert und durchlaufen werden müssen.

Das größte Problem für die Implementierung war allerdings , dass keine Referenzimplementierungen für sowohl L1-TGV als auch T2-TGV zur Verfügung standen. Dadurch war einerseits der Vergleich der Implementierungen nur begrenzt möglich aber auch das Erzeugen der Implementierung an sich. Das lag zum einen daran, dass bei der vorhandenen Referenzimplementierung von L2-TV Wiedersprüche zu der Beschreibung der Methode in der Veröffentlichung auftraten. Ohne Referenzimplementierung für L1-TGV und L2-TGV war es nicht möglich diese Wiedersprüche zu finden oder, falls es Wiedersprüche waren die auch in der Referenzimplementierung von L2-TV auftraten, eine Entscheidung über die richtige Implementierung zu treffen. Hinzukommt das ohne Referenzimplementierungen die Fehlersuche wesentlich erschwert wurde.

6 Fazit

Sowohl für L2-TGV als auch für L1-TGV wurden Implementierungen der Probleme in ExaSlang umgesetzt und getestet. Trotz Problemen bei der Implementierung kann zumindest für L2-TGV, anhand von Testergebnissen, gezeigt werden, dass durch die automatische Optimierung mit ExaSlang die Ausführungszeit der Implementierung um etwa ein vierfaches gesenkt werden konnte. Durch weitere Optimierung der Spezifikation ist möglicherweise sogar eine weitere Beschleunigung möglich. Für die Implementierung von L1-TGV konnte dies allerdings nicht gezeigt werden, da die Implementierung nicht die Problemspezifikation wiederspiegelte. Aber es wurde gezeigt das durch Optimierung des Codes eine deutliche Beschleunigung möglich ist. Dies steht allerdings gegenüber den gesteigerten Kosten für die Erzeugung der durch ExaSlang umgesetzten Problemspezifikation. Dabei handelt es sich vor allem um die Kosten sich in eine neue Sprache einzuarbeiten die nur für ein begrenztes Problemfeld anwendbar ist. Allerdings ist es möglich, dass diese Kosten in Zukunft deutlich gesenkt werden. Dies könnte zum Beispiel im Rahmen einer umfangreichen Dokumentation für ExaSlang stattfinden oder aber auch durch das zur Verfügung stellen von zusätzlichen Referenzcodes, so dass die Einarbeitung in ExaSlang vereinfacht wird.

7 Literaturverzeichnis

Alle Referenzen im Text wurden mit der zugehörigen Nummer, der Quelle, aus dem Literaturverzeichnis gekennzeichent .[NR]

[1] Bredies/ Kristian , Sun/Hong Peng: Preconditioned Douglas-Rachford Algorithms for TV- and TGV- Regularized Variational Imaging Problems, New York bei Springer Science 2015

[2] "Advanced Stencil-Code Engineering", unter http://exastencils.org/ (abgerufen am 29.10.2017)

[3] Schmitt/Christian, Kuckuk/Sebastian, Hannig/Frank, Teich/Jürgen, Köstler/Harald, Rüde/Ulrich, Lengauer/Christian: Systems of Partial Differenzial Equations in ExaSlang,Online bei Springer 2016

[4] Schmitt/Christian, Kuckuk/Sebastian, Hannig/Frank, Teich/Jürgen, Köstler/Harald : ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers, Fourth International Workshop on Domain-Specific Languages and High Level Frameworks for High Performance Computing 2014

[5] Bredies/Kristian, Sun/Hong Peng: Preconditioned Douglas-Rachford splitting methods for convex-concave saddle-point problems. SIAM J. Numer. Anal.(2014, in Press)

[6] Rudin/L.I.,Osher/S.,Fatemi/E.,: Nonlinear total variation based noise removal algorithms. Physica D60(1-4), 259-268 (1992)

[7] Bredies/Kristian, Kunisch/K.,Pock/T: Total generalized variation. SIAM J., Imaging SCI. 3(3), 492-526(2010)

[8] Bredies/k., Valkonen/T., : Inverse problems with second-order total generalized variation constraints. In: Proceedings of SampTA 2011-- 9th Internation Conference on Sampling Theory and Apllications, Singapore(2011)

8 Abbildungsverzeichnis

Alle referenzierten Gleichungen wurden nummeriert und werden mit (NR) gekennzeichnet.