Adaptive Mehrgitterverfahren in Raum und Zeit

Harald Wörndl-Aichriedler[1]

Lehrstuhl für Systemsimulation (Informatik X),
Institut für Mathematische Maschinen und Datenverarbeitung
Friedrich–Alexander–Universität Erlangen–Nürnberg

Erlangen, den 17. Dezember 1999

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.–Inf. Markus Kowarschik
Bearbeitungszeitraum:	20. Juni 1999 — 17. Dezember 1999

[1] mailto://Harald.Woerndl@spin.at

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.

Erlangen, den 17. Dezember 1999 Harald Wörndl-Aichriedler

Zusammenfassung

Adaptive Verfahren werden in Kombination mit Mehrgitterverfahren und iterativen Lösern bereits eingesetzt. Anhand einer C++–Implementierung der *Fully Adaptive Multigrid* Methode FAMe zur Lösung von elliptischen partiellen Differentialgleichungen (PDE) wurde untersucht, inwiefern sich die FAMe–Methode auf parabolische PDEs anwenden läßt. Das betrachtete zeitabhängige parabolische Modellproblem verlangte eine Neudefinition der Transportoperatoren und eine abstraktere Sichtweise auf das adaptive Verfeinerungskriterium. Um die Konvergenz zu gewährleisten, wurde zusätzlich der Aufbau der Gitterhierarchie gesteuert, damit anisotrope Differenzenoperatoren vermieden werden. Die numerischen Ergebnisse zeigen, daß die adaptive FAMe–Methode viele Einschränkungen bei der Lösung von parabolischen PDEs mit herkömmlichen Mehrgitterverfahren aufhebt.

Abstract

Adaptive methods combined with iterative multigrid methods have already been used for some problems. This paper describes the *fully adaptive multigrid* method FAMe applied to parabolic partial differential equations (PDE) resulting in a C++implementation. The FAMe method applied on the time related parabolic model problem needs a redefinition of the transport operators and a more abstract view on adaptive refinement. To ensure the convergenze of the method, the construction of the grid hierarchy has to be controlled to avoid anisotropic operators. Finally, numerical results allow to conclude that the adaptive FAMe method recalls many restrictions of solving a parabolic PDE with an ordinary multigrid method.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Von der stationären Wärmeverteilung eines Stabes bis hin zur Klimaentwicklung der Erde werden viele naturwissenschaftliche Probleme durch partielle Differentialgleichungssysteme beschrieben. Da eine analytische Lösung derartiger Aufgaben nicht immer gefunden werden kann, muß sie von einem numerischen Ergebnis approximiert werden.

Für die numerische Lösung existieren verschiedene Methoden. Sie basieren auf der Diskretisierung des Problems durch finite Elemente, finite Differenzen oder finite Volumina. In allen Fällen entsteht ein Gleichungssystem, das direkt oder iterativ gelöst wird. Direkte Löser arbeiten robust, jedoch nur mit hohem Entwicklungsaufwand auch performant. Iterative Löser konvergieren nur zur Lösung und verhalten sich nicht bei allen Problemen robust. Die iterativen Mehrgitter– oder Multiscale–Verfahren sind den direkten Lösern bei vielen Problemen überlegen, wenn nur eine Näherung der Lösung gesucht wird. Ein pauschaler Vergleich ist jedoch nicht möglich.

In dieser Arbeit werden iterative Mehrgitterverfahren (MG) betrachtet. Deren Einsatz in elliptischen linearen partiellen Differentialgleichungen ist weitgehend erforscht[1]. Sie werden im Kapitel 2 mit den Modellproblemen vorgestellt. Das Problem von singulären Stellen ist mit den herkömmlichen MG jedoch nicht in den Griff zu bekommen — der Diskretisierungsfehler nahe einer singulären Stelle verfälscht das Ergebnis auf dem ganzen Gebiet. Die Arbeit [Rüd88] von U. Rüde bietet dafür eine sehr gute Lösung. Eine andere Methode ist eine Verfeinerung der Diskretisierung um die Singularität. Dieser adaptive Ansatz ist automatisierbar und wird in Kapitel 3 weiter besprochen.

Ein weiterer adaptiver Ansatz ist, das iterative Verfahren auf einer Gitterebene zu optimieren. Es wird, wie schon von Gauß entdeckt und in [PR93] mit MG–Verfahren vereint, die Reihenfolge und Anzahl der iterativen Lösungsschritte von der Genauigkeit der Approximation gesteuert. Die

[1] Die gundlegende Arbeit zu MG ist von A. Brandt [Bra77]. Das Tutorial [Bri87] von W. Briggs ist eine ausführliche Einführung in Mehrgitterverfahren.

Kombination beider adaptiver Verfahren ergibt das in [Rüd93b] vorgestellte „Fully adaptive Multigrid“, FAMe Verfahren.

Die zu dieser Arbeit gehörende Implementierung basiert auf einem abgewandelten FAMe Algorithmus. Nur die Fehlerschätzung für die adaptive Verfeinerung wurde anders implementiert. Abschnitt 3.1 befaßt sich mit der neuen Verfeinerungsstrategie. Zur in Kapitel 5 beschriebenen Implementierung gehört auch die Speicherung der diskretisierten Daten. Bei adaptiver Verfeinerung entstehen dabei dünnbesetzte Gitter, die nicht mehr mit herkömmlichen Verfahren gespeichert werden können. Zur Speicherung wurden, wie schon in [GZ97], Hashtabellen verwendet. Zusammen mit der „Lazy Evaluation“ Technik wurde in C++das FAMe–Verfahren mit dem neuem Fehlerschätzer implementiert.

Das Ziel der Arbeit ist die Anwendung all dieser Techniken auf lineare parabolische Probleme. Von diesem Typ sind viele zeitabhängige Differntialgleichungen wie das zweite Modellproblem. Kapitel 4 befaßt sich mit der Diskretisierung und der Anwendung des MG–Verfahrens auf derartige Probleme und stützt sich dabei auf die Arbeit [HV95]. Zusätzlich konnten die adaptiven Verfahren auf parabolische Probleme angewandt werden. Erst dadurch werden mehrdimensionale parabolische Differenialgleichungssysteme mit der gleichen Effizienz lösbar wie elliptische Probleme im Raum.

Kapitel 2

Mathematische Grundlagen

Dieses Kapitel soll erstens die Idee, Mehrgittermethoden zur Lösung von Differentialgleichungen zu verwenden, nachvollziehen, und zweitens die Standardverfahren und Algorithmen erläutern.

Das Basiswerk zur hier weiterbehandelten Theorie ist die Arbeit von Achi Brandt [Bra77]. Eine sehr gute englischsprachige Einführung zum Thema Mehrgittermethoden bietet das Tutorium von Briggs [Bri87]. Das Thema Differentialgleichungen wird in vielen mathematischen Werken ausführlich behandelt. Die Nomenklatur der Diffentialrechnung wurde aus [BSMM95] übernommen. Die verwendete Notation ist am Ende des Kapitels auf Seite 19 aufgelistet.

2.1 Modellprobleme

Als modellhaftes Beispiel für viele in der Ingenieursmathematik vorhandene Probleme wird die *Wärmeverteilung* betrachtet. Für den zeitunabhängigen Fall ist die *stationäre* Wärmeleitungsgleichung im zweidimensionalen Raum,

$$u_{xx} + u_{yy} = f, \tag{2.1}$$

das erste Modellproblem. Die zeitabhängige eindimensionale Wärmeausbreitung in einem gleichförmigen Stab, definiert durch

$$u_t - a u_{xx} = f, \qquad a > 0, \tag{2.2}$$

ist das zweite Modellproblem. Diese Differentialgleichungen müssen auf dem Definitionsbereich Ω mit Randbedingungen und Anfangswerten für $\Gamma = \partial\Omega$ gelöst werden.

2.1.1 Lineare Partielle Differentialgleichungen

In der Physik werden viele Probleme durch *partielle Differentialgleichungen* oder auch PDEs („partial differential equations“) beschrieben. Eine analytische Lösung dieser oft in der zweiten Ordnung auftretenden Diffentialgleichungen ist meist nicht zu ermitteln. Die allgemeine Form von PDEs 2. Ordnung mit n Veränderlichen hat die Gestalt

$$\sum_{i,k} a_{i,k} \frac{\partial^2 u}{\partial x_i \partial x_k} + F\left(x_1, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n}\right) = 0,$$

wobei $a_{i,k}$ gegebene Funktionen der unabhängigen Varaiablen sind (Definition aus [BSMM95]). Wenn die Koeffizienten $a_{i,k}$ konstant sind, dann ist durch eine lineare homogene Transformation der unabhängigen Variablen die Umformung in die einfachere Normalform

$$\sum_i \kappa_i \frac{\partial^2 u}{\partial x_i^2} + \ldots = 0,$$

möglich, in der alle Koeffizienten κ_i gleich ± 1 oder 0 sind. Daraus ergegeben sich mehrere charakteristische Fälle:

Elliptischer Typ. Wenn alle κ_i vom selben Vorzeichen und ungleich 0 sind, handelt es sich um eine lineare PDE 2. Ordnung des elliptischen Typs. Wegen ihrer Häufigkeit in der Physik sind sie von besonderem Interesse. Gleichungen der oft vorkommenden, einfachen Form

$$\sum_i a_i \frac{\partial^2 v}{\partial x_i^2} + \sum_i b_i \frac{\partial v}{\partial x_i} + cv = g \qquad a_i \text{ und } b_i \text{ konstant, } a_i \neq 0 \ \forall i$$

lassen sich im elliptischen Fall mit

$$u = ve^{-\frac{1}{2}\sum \frac{b_i}{a_i} x_i}$$

in die noch einfachere Form

$$\Delta u + du = f \tag{2.3}$$

umformen, wobei der n–dimensionale *Laplace–Operator* Δ als

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \ldots + \frac{\partial^2 u}{\partial x_n^2} \tag{2.4}$$

definiert ist.

Der wichtigste Vertreter der elliptischen PDEs ist die sogenannte *Poisson–* oder *Potentialgleichung* mit den verschiedensten Einsatzgebieten, unter anderem in der Berechnung der Wärmeverteilung:

$$\Delta u = f$$

Deren homogene Form $\Delta u = 0$ ist die *Laplace–Gleichung*. Auch das erste Modellproblem, die zweidimensionale stationäre Wärmeverteilung (2.1) ist eine Poisson–Gleichung. Es wird im weiteren als das *elliptische Modellproblem* bezeichnet.

Parabolischer Typ. Eine weitere Gruppe der PDEs zweiter Ordnung sind Parabolische Differentialgleichungen. Sie ähneln den elliptischen, wobei jedoch genau ein $\kappa_j = 0$ ist und alle anderen κ_i das gleiche Vorzeichen haben. Zeitabhängige PDEs der Form

$$\Delta u + b\frac{\partial u}{\partial t} + cu = f$$

sind leicht erkennbar parabolischen Typs. Das zweite Modellproblem (2.2) der zeitabhängigen Wärmeausbreitung hat diese Form und wird deshalb auch als das *parabolische Modellproblem* bezeichnet.

Randwerte. Um bei Versuchen den Fehler $u^* - u$ zwischen der exakten Lösung u^* und einer Approximation u bestimmen zu können, ist es ratsam, eine Funktion u zu wählen, die die Laplace–Gleichung erfüllt. Die Randwerte

$$u(x, y), (x, y) \in \Gamma, \qquad \Gamma = \partial\Omega$$

werden dann direkt durch die gegebene Funktion bestimmt und der Fehler über Ω kann berechnet werden. In Definition (3.1) in Abschnitt 3.1 wird für das elliptische Modellproblem eine singuläre Lösung mit singulären Randwerten verwendet.

2.1.2 Diskretisierte lineare partielle Differentialgleichungen

Die erste, sehr allgemeine Idee ist, die Lösung von PDEs numerisch, mit Hilfe einer Vielzahl von diskreten Gleichungen zu approximieren. Durch die Diskretisierung eines Problems erhält man mehrere diskrete Gleichungen an verschiedenen Punkten x_i. Man kann die Punkte so anordnen, daß eine Menge von untereinander abhängigen Gleichungen entsteht wie in Abbildung 2.1 rechts dargestellt ist. Die diskreten Gleichungen einzelner *Punkte* bilden zusammen ein lineares Gleichungssystem.

Für den eindimensionalen Fall aus Abbildung 2.2 kann das Gleichungssystem für das Problem $u_{xx} = f$ mit den gegebenen Randwerten u_0 und u_4 und einer uniformen Diskretisierung $x_i = h \cdot i$ für die inneren Punkte mittels *zentraler Differenzen* notiert werden als

$$\begin{aligned}
\frac{u(x_0) - 2u(x_1) + u(x_2)}{h^2} &= f(x_1),\\
\frac{u(x_1) - 2u(x_2) + u(x_3)}{h^2} &= f(x_2),\\
\frac{u(x_2) - 2u(x_3) + u(x_4)}{h^2} &= f(x_3).
\end{aligned}$$

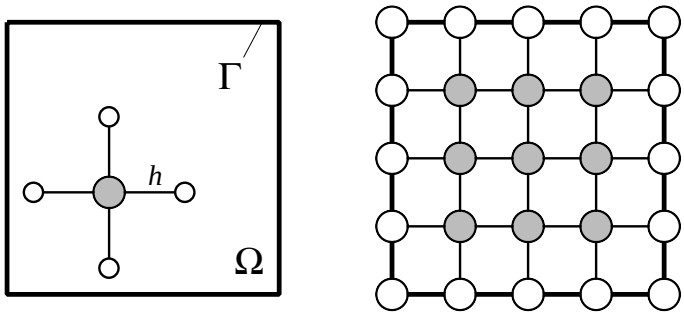


Abbildung 2.1: *Links* der Definitionsbereich Ω, der Rand Γ und ein Punkt eines mit der Gitterweite h diskretisierten zweidimensionalen Problems. *Rechts* das gleichmäßig diskretisierte Problem mit grauen inneren Punkten und weißen Randpunkten.

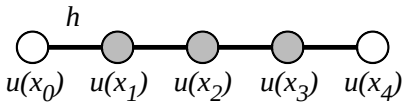


Abbildung 2.2: Die uniforme Diskretisierung eines eindimensionalen Problems.

Numeriert man die Punkte der Diskretisierungen von u und f durch, so erhält man die Form

$$\begin{aligned} h^{-2}(u_0 - 2u_1 + u_2) &= f_1 \\ h^{-2}(u_1 - 2u_2 + u_3) &= f_2 \\ h^{-2}(u_2 - 2u_3 + u_4) &= f_3, \end{aligned} \tag{2.5}$$

die sich auch in der Matrizenschreibweise $Au = f$ notieren läßt:

$$\frac{1}{h^2}\begin{pmatrix} 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

Die verbleibenden Unbekannten sind die inneren Werte u_1, u_2 und u_3. Die Funktionswerte f_0 und f_4 am Rand werden nicht weiter benötigt. Man sieht, daß jede Unbekannte u_i über das Gleichungssystem mit ihren Nachbarpunkten in Beziehung steht. Ist $Au = f$ und A die Koeffizientenmatrix, so definiert die Zeile i von A den *Sternoperator* für eine Unbekannte. Die einzelnen Gleichungen des Systems lassen sich mit dem Sternoperator für dieses Beispiel vereinfacht notieren als

$$h^{-2}[\ 1 \quad -2 \quad 1\ ] \cdot u_i = f_i, \qquad 1 \le i \le 3,$$

und sind äquivalent zu den drei Formeln in (2.5).

Für das elliptische Modellproblem im Raum wurde folgender Sternoperator für die Diskretisierung des Δ–Operators gewählt, wobei $h_x \neq h_y$ sein kann:

$$\begin{bmatrix} & h_y^{-2} & \\ h_x^{-2} & -2h_x^{-2} - 2h_y^{-2} & h_x^{-2} \\ & h_y^{-2} & \end{bmatrix} \tag{2.6}$$

Es gilt nun, das Gleichungssystem für alle Unbekannten zu lösen, beziehungsweise die Lösung zu approximieren.

2.2 Lösung der diskretisierten Modellprobleme

Für die Lösung des Gleichungssystems $Au = f$ der Diskretisierung eines Problems stehen mehrere Möglichkeiten zur Auswahl. Man kann das System mit einem *direkten Verfahren* lösen. Ein einfacher Vertreter dieser *direct solvers* ist die *Gaußsche Elimination*. Die Anzahl der Unbekannten n ist meistens sehr groß und die Martix A eine m–Bandmatrix, die jedoch anfangs nur mit $\mathcal{O}(n)$ Koeffizienten ungleich 0 besetzt ist (Streifen). In einem k–dimensionalen Gitter von Unbekannten eines mit h uniform diskretisierten Problems der Seitenlänge $N = 1 + \frac{1}{h}$ ist $n = \mathcal{O}(N^k)$ und $m = \mathcal{O}(N^{k-1})$. Wird die Gaußsche Elimination angewandt, so füllt sich die Bandmatrix auf $\mathcal{O}(n \cdot m)$ Koeffizienten ungleich 0 an, womit die Speicherkomplexität steigt. In der Praxis werden andere direkte Löser verwendet, die an die Struktur des Gleichungssystems besser angepaßt sind. Da die Lösung jedoch im Rahmen der numerischen Darstellung exakt ist und derartige Löser ein abschätzbares Laufzeitverhalten haben, wird dieser immense Aufwand oft betrieben.

Als Alternative existieren die sogenannten *iterativen Verfahren*. Sie wurden vor über 175 Jahren von C. Gauß als eine „indirekte Eliminationsmethode für 4×4 Matrizen“ vorgeschlagen[1]. Die Methode beruhte schon damals auf einer aufeinanderfolgenden Entspannung (*Relaxation*) der einzelnen Gleichungen durch das Lösen nach der betreffenden Unbekannten. Im ursprünglichen Algorithmus wird die Gleichung i mit dem größten Residuum r_i zur weiteren Relaxation gewählt, wobei $r = f - Au$ ist. Dieser Algorithmus heißt *Gauß–Southwell relaxation*.

Die Reihenfolge und die Art der Relaxation differenzieren die verschiedenen iterativen Algorithmen. Zusätzlich kann die Anforderung existieren, die Diskretisierung mit einem Fehlerschätzer zu kombinieren, um somit ein an das Problem angepaßtes Gitter zu generieren. Iterative und direkte Lösungsverfahren sind nicht unmittelbar vergleichbar, da iterative Verfahren zur Lösung konvergieren, direkte Löser aber unmittelbar die exakte Lösung im

[1] Entnommen aus der Einführung von [PR93].

Rahmen der Maschinengenauigkeit berechnen. Ein Vergleich ist nur für eine konkrete Anwendung denkbar.

2.2.1 Einfache iterative Lösungsverfahren

Die folgenden Lösungsverfahren sind in dem Tutorium [Bri87] ausführlich erläutert und analysiert. Für weiteres Interesse bieten sowohl [Rüd93b, PR93] als auch Mathematische Lehrbücher einen guten Ausgangsspunkt.

Jacobi–Relaxation. Der einfachste Vertreter der iterativen Algorithmen ist die Jacobi–Relaxation. Dieses Verfahren arbeitet für ein gesuchtes u bei $Au = f$ schrittweise von Approximation $u^{(k)}$ zu $u^{(k+1)}$. Der Startwert $u^{(0)}$ ist gegeben. Ein Iterationsschritt berechnet für jede Unbekannte $u_i^{(k)}$ den Wert

$$u_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j,\ j \neq i} a_{ij} u_j^{(k)} \right) \tag{2.7}$$

um die nächste Approximation $u^{(k+1)}$ zu erhalten. Die Berechnung (2.7) ist in der Regel bei diskretisierten PDEs sehr einfach auszuführen. Für das eindimensionale Beispiel aus Abschnitt 2.1.2 gilt

$$u_i^{(k+1)} = \frac{1}{-2} \left(h^2 f_i - u_{i-1}^{(k)} - u_{i+1}^{(k)} \right).$$

Die Reihenfolge der Relaxationen in (2.7) ist egal, da immer nur auf Elemente der letzten Approximation zugegriffen wird.

Gauß–Seidel Relaxation. Unter der Annahme, daß jede Unbekannte der gesuchten Lösung ab einem gewissen Zeitpunkt zur Lösung konvergiert, kann man davon ausgehen, daß jüngere Werte näher an der exakten Lösung liegen. Bei der seriellen Relaxation der Gleichungen einer Bandmatrix sind im Durchschnitt bei jeder Relaxation schon 50% der Werte $u_j^{(k+1)}$ der neuen Lösung $u^{(k+1)}$ berechnet. Diesen Effekt benutzt die Gauß–Seidel Relaxation. Ein neu berechneter Wert u_i wird sofort in den darauffolgenden Relaxationen weiterverwendet. In der Praxis gibt es nur einen Lösungsvektor u, der während der Relaxation aller Punkte u_i modifiziert wird. Dadurch wird gegenüber der Jacobi–Relaxation nur etwa der halbe Speicher benötigt.

Nun hat die *Reihenfolge der Relaxationen* jedoch eine Bedeutung[2]. Deshalb gibt es mehrere Varianten des Gauß–Seidel Algorithmus. Die einfachste, sogenannte *reguläre* Reihenfolge ist, alle Punkte $1, \ldots, n$ der Reihe nach durchzuarbeiten. Die *symmetrische* Gauß–Seidel Methode arbeitet alle von $1, \ldots, n$ und danach von $n, \ldots, 1$ ab. Sehr effektiv ist die als „*red–black*“ bekannte Variante, in der zuerst alle „geraden“ roten Punkte und danach

[2] Die Numerierung der Punkte bleibt hierbei unverändert.

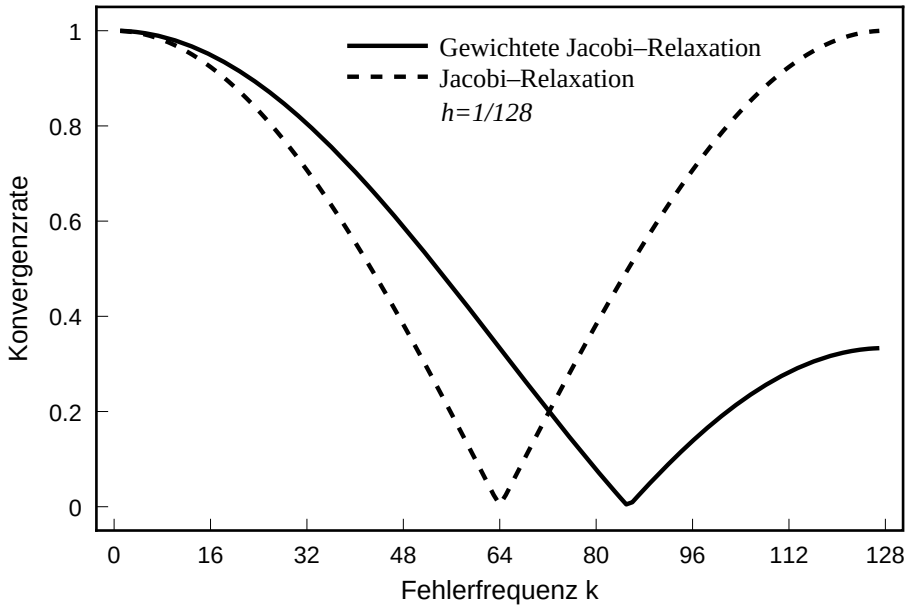


Abbildung 2.3: Die Konvergenzrate eines Jacobi–Verfahrens bei der Glättung des initialen Fehlers $v^{(k)}$, nicht gewichtet ($\omega = 1$) und gewichtet mit $\omega = 2/3$.

alle „ungeraden“ schwarzen Punkte relaxiert werden. Im zweidimensionalen Fall ist die Anordnung der roten und schwarzen Punkte schachbrettartig. Ein Vorteil dieser Variante ist die sehr gute Parallelisierbarkeit, da bei kompakten Stern–Operatoren wie in (2.6) nur die unmittelbar benachbarten Punkte in die Berechnung eingehen. Bei diesem Zweischritt–Verfahren gibt es dadurch keine Abhängigkeiten zwischen Relaxationen in einem Berechnungsschritt — eine Voraussetzung für effiziente Parallelisierbarkeit.

Gewichtete Verfahren. In [Bri87] wird für diese Standard–Verfahren die Konvergenzrate anhand diskretisierter Fourier–Moden

$$v_i^{(k)} = \sin\left(\frac{ik\pi}{N}\right), \qquad 0 \le i \le N, \quad 1 \le k \le N-1$$

untersucht. Abhängig von der Frequenz k konvergiert die Lösung des Problems $u_{xx} = 0, u_0 = u_N = 0$ vom Startwert $u_i^{(0)} = v_i^{(k)}$ unterschiedlich schnell zur Lösung $u \equiv 0$. Die Abbildung 2.3[3] veranschaulicht die Konvergenzrate für das herkömmliche Jacobi–Verfahren in Abhängigkeit der Frequenz der initialen Fehlerfunktion v. Zusätzlich ist die Konvergenzrate für das sogenannte *gewichtete Jacobi–Verfahren* abgebildet. Dabei bestimmt der

[3] Die Tests wurden mit dem Hilfsprogramm `jacobiKonv1D.c` durchgeführt.

Faktor ω während der Relaxation das Gewicht des neu errechneten Wertes beziehungsweise der Korrektur, so daß sie ähnlich (2.7) definiert ist als

$$\begin{aligned} \hat{u}_i^{(k+1)} &= \frac{1}{a_{ii}} \left(f_i - \sum_{j,\ j \neq i} a_{ij} u_j^{(k)} \right), \\ u_i^{(k+1)} &= \omega \cdot \hat{u}_i^{(k+1)} + (1-\omega) \cdot u_i^{(k)}. \end{aligned}$$

Analog kann man die Gewichtung auch auf das Gauß–Seidel Verfahren anwenden und erhält somit das *SOR–Verfahren*.

Ungewichtetes Glätten mit dem Jacobi–Verfahren hat den Nachteil, daß nur Fehler aus dem mittleren Frequenzbereich gedämpft werden. Die Gewichtung verbessert das Verfahren im hohen Frequenzbereich, während die niederfrequenten Fehler etwas schlechter gedämpft werden. Die Wahl von $\omega = 2/3$ hat sich im eindimensionalen Fall als sehr robust erwiesen und wird auch in der Abbildung 2.3 für das gewichtete Jacobi–Verfahren verwendet.

2.2.2 Mehrgitterverfahren

Iterative Standardverfahren beseitigen mittel– und hochfrequente Fehler schnell mit sehr wenigen Iterationen. Das Frequenzspektrum der Glättung hängt, wie Abbildung 2.3 vermuten läßt, stark von der Gitterweite h der Diskretisierung ab. Die Idee ist nun,

durch Wahl eines gröberen h das Frequenzspektrum zu skalieren, so daß niederfrequente Fehler in den gut konvergierenden Teil des Konvergenzspektrums fallen.

Man sollte nun also durch die Wahl einer gröberen Diskretisierung auch niederfrequente Fehler eliminieren können. Wird dies für verschiedene h_i gemacht, so ergibt sich eine gemeinsame Konvergenzrate wie in Abbildung 2.4. Die dort eingetragene Gesamtkonvergenrate ist das Produkt der Konvergenzraten h_4 bis h_7. Das ist der Grundgedanke von *Mehrgittermethoden* (im Englischen auch als „Multigrid–“ oder „Multiscale–Methods“ bezeichnet). Die Symmetrien und Wiederholungen bei $h > 1/128$ entstehen durch die laut dem Abtasttheorem zu grobe Diskretisierung der Fehler v, da $k \geq N = 1/h$ wird, und

$$\begin{aligned} v_i^{(k)} &= \sin\left(\frac{k}{N} i\pi\right) = \sin\left(\frac{2N \cdot m + k}{N} i\pi\right) = -\sin\left(\frac{2N \cdot m - k}{N} i\pi\right) \\ &\Downarrow \\ v_i^{(k)} &= v_i^{(2N \cdot m + k)} = -v_i^{(2N \cdot m - k)} \qquad \forall i, k, m \in \mathbb{Z} \end{aligned}$$

gilt. Somit sind die Konvergenzraten bei $u^{(0)}$ gleich den diskretisierten Fourier–Moden $v^{(2N \cdot m + k)}$ oder $v^{(2N \cdot n - k)}$ für jedes m und n identisch.

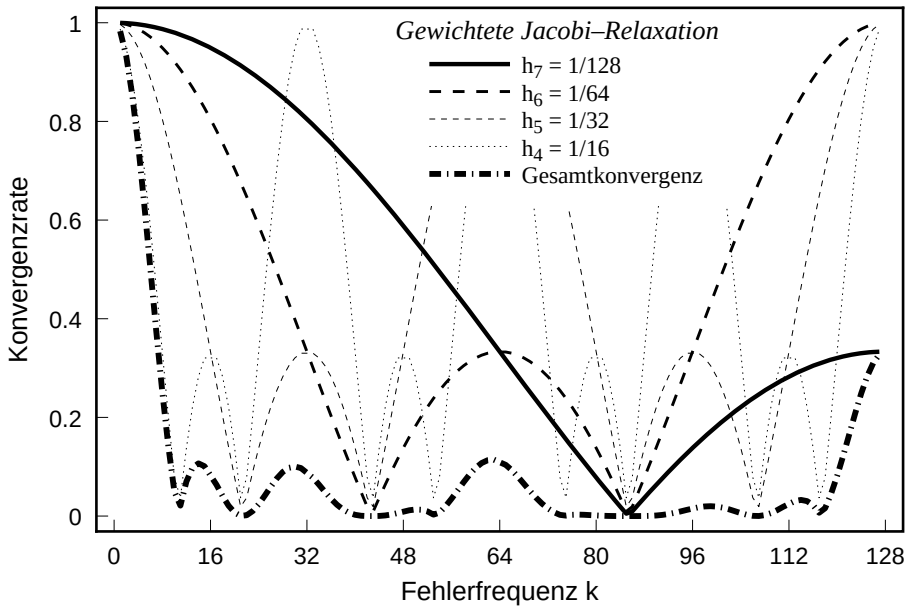


Abbildung 2.4: Die Konvergenzrate $c_h(k)$ des mit $\omega = 2/3$ gewichteten Jacobi–Verfahrens in Abhängigkeit von der Gitterweite h. Werden die Konvergenzraten überlagert, ergibt sich als *Gesamtkonvergenz* das Produkt $c_{h_4} \cdot c_{h_5} \cdot c_{h_6} \cdot c_{h_7}$.

1. Relaxiere v_1 mal das Feingittersystem $A_h u_h = f_h$.
2. Bilde das Residuum $r_h \leftarrow f_h - A_h u_h$.
3. Bilde $f_{2h} \leftarrow I_h^{2h} r_h$.
4. Löse das grobe System $A_{2h} u_{2h} = f_{2h}$ exakt.
5. Korrigiere $u_h \leftarrow u_h + I_{2h}^h u_{2h}$.

Abbildung 2.5: Der *Correction Scheme* Zweigitter–Algorithmus.

Um nun verschiedene diskrete Approximationen u_h zu vereinigen, gibt es verschiedene Algorithmen. Die wichtigsten Mehrgitterverfahren sind in [Bri87, Gri90] erläutert. Die Basis von Mehrgitter–Algorithmen ist oft eine einfache Zweigitter–Methode wie das *Correction Scheme* in Abbildung 2.5. Es approximiert den Fehler e_h der Approximation u_h für $A_h u_h = f_h$ und korrigiert u_h damit. Der Fehler e_h ist definiert durch $A_h e_h = r_h$, wobei r_h das Residuum ist. Eine direkte Berechnung des Fehlers ist gleich aufwendig wie die Berechnung von u_h, e_h ist aber nach der Vorglättung meist glatt. Er wird deshalb in einem gröberen und dadurch kleineren Gleichungssystem $A_{2h} e_{2h} = I_h^{2h} r_h$ berechnet und danach auf die feinere Diskretisierung „hochgerechnet". Dazu sind sogenannte Transportoperatoren zwischen verschiedenen Diskretisierungsstufen notwendig.

Transportoperatoren. Für den Transport vom feinen System h zum groben System $2h$ wird die *Restriktion* I_h^{2h} benötigt, für den umgekehrten Fall die *Prolongation* I_{2h}^h von einem groben in ein feines System.

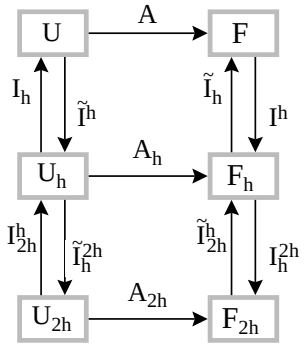


Abbildung 2.6: Die verschiedenen Transportoperatoren zwischen den Funktionenräumen in einer Übersicht.

Die Übersicht in Abbildung 2.6 zeigt, daß die Transportoperatoren sowohl zwischen den diskretisierten Räumen wie $U_h \to U_{2h}$ als auch zwischen kontinuierlichen und diskreten Räumen abbilden wie zum Beispiel $U \to U_h$. Der diskrete Grobgitteroperator A_{2h} kann in bezug auf diese Operatoren auf zwei Arten gebildet werden[4].

- Erstens durch die sogenannte *Galerkin–Approximation*: Der Grobgitteroperator A_{2h} wird in Abhängigkeit des Feingitteroperators A_h besimmt, sodaß

$$A_{2h} = I_h^{2h} A_h I_{2h}^h. \tag{2.8}$$

[4]Diese Definitionen gelten genauso für den allgemeinen Fall zwischen zwei beliebigen Diskretisierungen h_1 und h_2. Genaueres dazu in [Gri90].

- Zweitens, wenn der Operator A_h durch eine Diskretisierung $A_h = I^h A I_h$ direkt aus dem kontinuierlichen Operator erzeugt wurde, kann analog auch der Grobgitteroperator auf diese Weise erzeugt werden:

$$A_{2h} = I^{2h} A I_{2h} \tag{2.9}$$

Um die durch die Galerkin–Approximation gewonnenen Operatoren in Zusammenhang mit der direkten Diskretisierung zu bringen, fordert man, daß beide so gewonnenen Grobgitteroperatoren identisch sind. So muß

$$A_{2h} \overset{(2.9)}{=} I^{2h} A I_{2h} \overset{(2.8)}{=} I_h^{2h} A_h I_{2h}^h$$

gelten. Mit $A_h = I^h A I_h$ ergeben sich für die noch unbestimmten Transportoperatoren I_h^{2h} und I_{2h}^h folgende zu erfüllende Bedingungen:

$$I^{2h} = I_h^{2h} I^h, \qquad I_{2h} = I_h I_{2h}^h \tag{2.10}$$

Sind diese Bedingungen erfüllt, so sind die Diskretisierungsverfahren und die gewählte Prolongation und Restriktion zueinander *kompatibel*[5]. Die so definierten diskrete Operatoren A_h sind dadurch auch immer eine direkte Diskretisierung des kontinuiertlichen Operators A. So ist für eine Implementierung gewährleistet, daß A_h immer eine gleichartige Struktur aufweist. In der Praxis soll zum Beispiel der Laplace–Operator aus (2.4) in jeder Diskretisierung nur von h abhängig sein und einem 5er–Stern gleichen.

Das Multigrid Correction Scheme — CS. Das in Abbildung 2.5 veranschaulichte zweistufige Korrekturschema ist die Grundlage für den Aufbau zum einfachsten Mehrgitter–Algorithmus, dem Multigrid CS. Der vierte Schritt, die exakte Berechnung von u_{2h} ist nicht weiter definiert. Es wird nun die exakte Berechnung von u_{2h} durch eine Approximation mit der CS–Methode ersetzt. Daraus ergibt sich eine Rekursion über h bis hin zum gröbsten Gitter $u_{h_{max}}$. Dieses letzte Gleichungssystem $A_{h_{max}} u_{h_{max}} = f_{h_{max}}$ ist in der Regel direkt lösbar. Existiert darin, wie es bei den Modellproblemen der Fall ist, nur eine Unbekannte, dann löst eine Relaxation ohne Gewichtung das System.

Dem Mehrgitter–Algorithmus aus Abbildung 2.7 werden drei Parameter v_1, v_2 und γ übergeben, die den Ablauf steuern. Konkret geben v_1 und v_2 die Anzahl der Relaxationen vor und nach der Rekursion an. Diese Relaxationen werden auch als *Vorglättung* und *Nachglättung* bezeichnet. Die Nachglättung ist nötig, da im Gegensatz zum Zweigitter–Verfahren die Lösung u_{2h} nicht mehr korrekt ist. Auf jeden Fall werden damit hochfrequente, durch die Prolongation erzeugte Fehler gedämpft. Der Parameter γ steuert den Rekursionszyklus an sich. Ein $\gamma = 1$ erzeugt dabei einen V–Zyklus, ein $\gamma = 2$

```
proc CS(u_h, f_h, v_1, v_2, γ)
    Wenn h = h_max
        Löse direkt → return.
    Relaxiere v_1 mal A_h u_h = f_h.
    Bilde r_h ← f_h − A_h u_h.
    Bilde f_2h ← I_h^2h r_h.
    Initialisiere u_2h ← 0.
    Rufe γ mal CS(u_2h, f_2h, v_1, v_2, γ) auf.
    Korrigiere u_h ← u_h + I_2h^h u_2h
    Relaxiere v_2 mal A_h u_h = f_h.
end proc.
```

Abbildung 2.7: Der *Correction Scheme* Mehrgitter–Algorithmus.

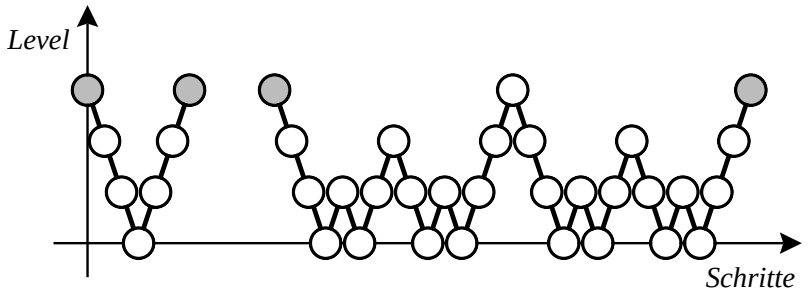


Abbildung 2.8: *Links* der V–Zyklus mit $\gamma = 1$, *rechts* der W–Zyklus mit $\gamma = 2$ im schematischen Ablauf.

einen sogenannten W–Zyklus. Für die meisten Probleme wird $\gamma = 1$ und somit der V–Zyklus gewählt.

Die Transportoperatoren für das CS angewandt auf das elliptische Modellproblem müssen so gewählt werden, daß der diskrete Laplace–Operator Δ_h direkt aus Δ erzeugt wird und zwei Operatoren Δ_h und Δ_{2h} die Galerkin–Approximation erfüllen. Somit sind die Prolongation und die Restriktion nach (2.10) zueinander kompatibel.

Es kann für die Prolongation die *bilineare Interpolation* verwendet werden und passend dazu als Restriktion die *gewichtete Restriktion*, so daß gilt:

$$I_{2h}^{h} = c\left(I_{h}^{2h}\right)^{T}, \qquad c \in \mathbb{R}.$$

Transportoperatoren werden wie in [Gri90] in einer eigenen Schreibweise

[5] Dieses Kriterium ist insbesondere für Abschnitt 3.1 relevant.

notiert. Die gewichtete Restriktion wird in der *Molekül*–Notation[6] als

$$I_h^{2h} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

definiert, die bilineare Interpolation analog dazu als Prolongation in der *Stempel*–Notation[7]

$$I_{2h}^h = \frac{1}{4} \rceil \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix} \lceil .$$

Weiterführende Betrachtungen der Transportoperatoren erfolgen in Abschnitt 3.1 und in [Bri87, Gri90, Gri94].

Das Full Approximation Storage Scheme — FAS. Ein Nachteil des CS ist, daß die Werte jeder Ebene nur Korrekturen sind. Ersetzt man die Grobgitterfunktion des CS durch

$$\hat{u}_{2h} = u_{2h} + \tilde{I}_h^{2h} u_h,$$

wobei u_{2h} noch immer die alte Grobgitterkorrektur darstellt, dann erhält man die neue Grobgittergleichung

$$A_{2h}\hat{u}_{2h} = f_{2h} + A_{2h}\tilde{I}_h^{2h} u_h.$$

Hierbei ist $\tilde{I}_h^{2h}$ nicht gleich dem von $F_h \to F_{2h}$ transportierendem I_h^{2h} (vergleiche Abbildung 2.6). Für das elliptische Modellproblem kann die einfache Injektion

$$\tilde{I}_h^{2h} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

angewandt werden. Zusätzlich muß auch die u–Korrektur angepaßt werden. Daraus ergibt sich der Algorithmus in Abbildung 2.9. Das FAS–Schema ist im Fall eines linearen diskreten Operators äquivalent zum CS–Algorithmus. Für die adaptive Verfeinerung aus Abschnitt 3.1 ist das FAS–Verfahren jedoch von großem Vorteil.

[6] Moleküle definieren analog der Stern–Notation einzelne Zeilen einer Operandenmatrix.
[7] Stempel definieren einzelne Spalten einer Operandenmatrix.

1. Relaxiere v_1 mal das Feingittersystem $A_h u_h = f_h$.
2. Bilde den Startwert $\bar{u}_{2h} \leftarrow \tilde{I}_h^{2h} u_h$.
3. Bilde das Residuum $r_h \leftarrow f_h - A_h u_h$.
4. Bilde $\hat{f}_{2h} \leftarrow I_h^{2h} r_h + A_{2h} \bar{u}_{2h}$.
5. Löse das grobe System $A_{2h} \hat{u}_{2h} = \hat{f}_{2h}$ exakt.
6. Korrigiere $u_h \leftarrow u_h + I_{2h}^h (\hat{u}_{2h} - \bar{u}_{2h})$.

Abbildung 2.9: Der *Full Approximation Storage* Zweigitter–Algorithmus.

Analog zum Mehrgitter–CS kann das Mehrgitter–FAS–Schema aus Abbildung 2.10 aufgebaut werden[8]. Das Residuum r_h wird jedoch nicht mehr explizit im feinen Gitter berechnet und dann transportiert sondern umgeformt in

$$I_h^{2h} r_h = I_h^{2h}(f_h - A_h u_h) = I_h^{2h} f_h - I_h^{2h}(A_h u_h).$$

Somit ist

$$\begin{aligned} \hat{f}_{2h} &= I_h^{2h} f_h - I_h^{2h}(A_h u_h) + A_{2h} \bar{u}_{2h}, \qquad \text{mit } f_{2h} = I_h^{2h} f_h \text{ ist} \\ \hat{f}_{2h} &= f_{2h} - I_h^{2h}(A_h u_h) + A_{2h} \bar{u}_{2h}. \end{aligned}$$

Diese Korrektur von f_{2h} wird bezeichnet als *τ–Korrektur*

$$\tau_h^{2h} = A_{2h} \bar{u}_{2h} - I_h^{2h}(A_h u_h). \tag{2.11}$$

Ohne τ–Korrektur ist das grobe Gleichungssystem gleich der Diskretisierung des kontinuierlichen Problems. Die Korrektur τ_h^{2h} kann wie in [Gri94] auch als „Fein–zu–Grob–Defektkorrektur“ gesehen werden. Die Defektdifferenz τ_h^{2h} zwischen zwei Gittern h und $2h$ wird auf der rechten Seite des groben Gitters korrigiert als

$$f_{2h} \leftarrow f_{2h} + \underbrace{r_2 h - I_h^{2h} r_h}_{=\tau_h^{2h}}.$$

Auch bei der Definition der virtuellen globalen Gitter in Abschnitt 3.1.2 wird der τ–Term verwendet. Durch Extrapolation läßt sich bei der Lösung linearer Probleme sogar die Ordnung der Diskretisierung im MG–Verfahren erhöhen. Über das Thema Extrapolation finden sich in der Mehrgitter–Literatur mehrere Artikel zum Stichwort „τ–Korrektur“.

[8] Wobei $\hat{f}$ und $\hat{u}$ als f und u notiert werden.

```
proc FAS(u_h, f_h, v_1, v_2, γ)
    Wenn h = h_max
        Löse direkt → return.
    Relaxiere v_1 mal A_h u_h = f_h.
    Bilde den Startwert ū_2h ← Ĩ_h^2h u_h.
    Bilde τ_h^2h ← A_2h ū_2h − I_h^2h A_h u_h.
    Bilde f_2h ← I_h^2h f_h + τ_h^2h.
    Initialisiere u_2h ← ū_2h.
    Rufe γ mal FAS(u_2h, f_2h, v_1, v_2, γ) auf.
    Korrigiere u_h ← u_h + I_2h^h (u_2h − ū_2h)
    Relaxiere v_2 mal A_h u_h = f_h.
end proc.
```

Abbildung 2.10: Der *Full Approximation Storage* Mehrgitter–Algorithmus mit expliziter τ–Berechnung.

Andere Mehrgitterverfahren. Ein Abkömmling des FAS–Schemas ist das *HT–Schema* (das Hierarchische Transformationen–Schema) aus [Gri94]. Es hat die gleichen Eigenschaften wie das FAS–Schema, benutzt jedoch eine hierarchische Basis für die Darstellung von u. Dadurch ergeben sich algorithmische Vor– und Nachteile, jedoch keine Unterschiede in der Qualität der Lösung.

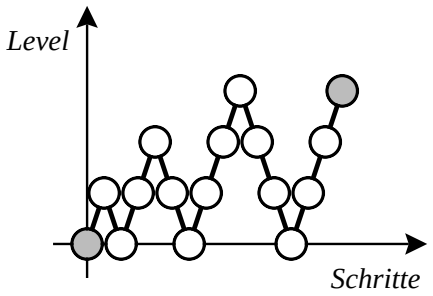


Abbildung 2.11: Der schematische Ablauf des *Full Multigrid* Zyklus FMG mit $\gamma = 1$.

Die Vielzahl von Varianten mit verschiedenen Zyklen wie aus Abbildung 2.8 wird hier nicht weiter betrachtet. Für die Implementierung wurde ein FAS–Verfahren als Grundlage verwendet, das mit einem dem *Full Multigrid* FMG aus Abbildung 2.11 ähnlichen Zyklus arbeitet. Das FMG–Verfahren oder im speziellen der FMV–Zyklus mit $\gamma = 1$ wird in [Bri87] erläutert. Eine Approximation von u_h wird mit einem V–Zyklus erzeugt und danach auf dem nächstfeineren Gitter als Startwert verwendet. FMG beginnt mit

dem gröbsten Gitter und arbeitet sich bis zur gewünschten Diskretisierung vor. Das Verfahren bietet den Vorteil, daß schon zu Beginn eines jeden V–Zyklus über $A_h u_h = f_h$ ein guter Startwert durch die Prolongation des vorhergehenden Ergebnisses u_{2h} erzeugt wurde.

Zusätzlich zur FAS–Methode und zum FMG–Zyklus bietet es eine adaptive Verfeinerung und eine adaptive Glättung. Dafür wird im folgenden Kapitel 3 näher auf die Basisfunktionen von u und die Transportoperatoren eingegangen.

Notation

Die Notation stammt überwiegend aus [McC89], [McC92] und [Rüd93b], wird in anderen Büchern jedoch ähnlich verwendet.

U, F	Funktionenräume
u, f	Funktionen
A	Kontinuierlicher Operator des Problems $Au = f$
h	Mehrdimensionale Gitterweite, verwendet im Index von diskretisierten Funktionen deren Funktionsräumen.
$2h$	Bezeichnet im allgemeinen eine gröbere Gitterweite. Bei *Standard–Coarsening* ist $2h = 2 \cdot h$.
h_x	Gitterweite in der x Richtung
Ω	Definitionsbereich aus einem n–dimensionalen euklidischen Raum, für die Modellprobleme ist $\Omega = [0,1]^2 \subset \mathbb{R}^2$.
Γ	Randbereich $\Gamma = \partial\Omega$
Ω_h, Γ_h	n–dimensionals Gitter und dessen Rand mit der Maschenweite h
U_h, F_h	Gitterräume mit Elementen u_h, f_h
u_h, f_h	Gitter, dessen Punkte $u_h(x,y)$ (zweidimensionaler Fall) mit i numeriert werden als $u_h(i)$
r_h	Residuum $r_h = f_h - A_h u_h$
A_h	Diskreter Operator des diskretisierten Problems $A_h u_h = f_h$ von $U_h \to F_h$ als Matrix
$a_{ij}, A_h[i,\ j]$	Das Element der i. Zeile in der j. Spalte der Matrix A_h
u^*, u_h^*	Exakte Lösung eines Problems als kontinuierliche Funktion u^* und als Gitter u_h^*
Θ_h	Das skalierte Residuum $\Theta_h = D_h^{-1}(f_h - A_h u_h)$ für das diskretisierte Problem
$K_{h,i}$	Basisfunktion eines Punktes i eines Gitters u_h $(K_{h,i} \in U)$
$\|\| \cdot \|\|_\infty$	Maximumsnorm
∇	Divergenzoperator, $\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}$
Δ	Laplace–Operator, in (2.4) definiert. Im zweidimensionalen Fall ist $\Delta = \nabla \cdot \nabla$
	Transportoperatoren (siehe auch Abbildung 2.6)
I_h^{2h}	Restriktion $F_h \to F_{2h}$
$\tilde{I}_h^{2h}$	Restriktion $U_h \to U_{2h}$
I_{2h}^h	Prolongation $U_{2h} \to U_h$
$\tilde{I}_{2h}^h$	Prolongation $F_{2h} \to F_h$
	Abkürzungen
PDE	Parital differential equation, partielle Differentialgleichung
MG	Multi–Grid, Mehrgitterverfahren (Abschnitt 2.2.2)
FMG	Full multigrid method (Seite 17)
FAS	Fully approximation scheme (Seite 15)
HT	Hierarchische Transformationen Schema (Seite 17)
FAMe	Fully adaptive multigrid (Seite 31)

Kapitel 3

Adaptive Verfahren

Es gibt in dieser Arbeit zwei adaptive Ansätze: eine *adaptive Verfeinerung* der Diskretisierung sowie die *Gauß'sche adaptive Relaxation* eines Gleichungssystems. In [Rüd93b, Rüd93a, McC89, McC92] werden adaptive Verfahren näher betrachtet. Dieser Abschnitt baut insbesondere auf [Rüd93b] auf.

3.1 Adaptive Verfeinerung der Diskretisierung

Die numerische Lösung eines Problems muß passend diskretisiert werden um eine gewünschte Genauigkeit zu erreichen. Gesucht ist die diskrete Lösung u_h mit der *kleinsten Anzahl* von Unbekannten, die die exakte Lösung u^* hinreichend approximiert (analog [PR93, Seite 5]).

In der Praxis gibt es natürlich noch zusätzliche Anforderungen, die oft durch die verwendeten Lösungsalgorithmen gegeben sind, wie etwa der Zwang einer *uniformen Diskretisierung* mit Punktabstand h. Mit dieser Zusatzanforderung gibt es in der Regel für jede gewünschte Genauigkeit ein h, so daß die uniform diskretisierte Lösung u_h die exakte Lösung u^* hinreichend genau approximiert. Existieren in einer gesuchten Funktion jedoch singuläre Stellen, so kann das Diskretisierungsgitter sehr fein werden und damit auch die Anzahl der Unbekannten sehr hoch, obwohl nur an wenigen Stellen eine feine Diskretisierung notwendig wäre.

Die Lösung des elliptischen Modellbeispiels (2.1) wird durch die folgenden explizit singulär vorgegebenen Randbedingungen auch singulär:

$$u(x,y) = \mathrm{Re}[(x+iy)^\alpha] = \cos(\alpha\Theta) r^\alpha, \qquad \alpha = 0.5,\ (x,y) \in \Gamma \tag{3.1}$$

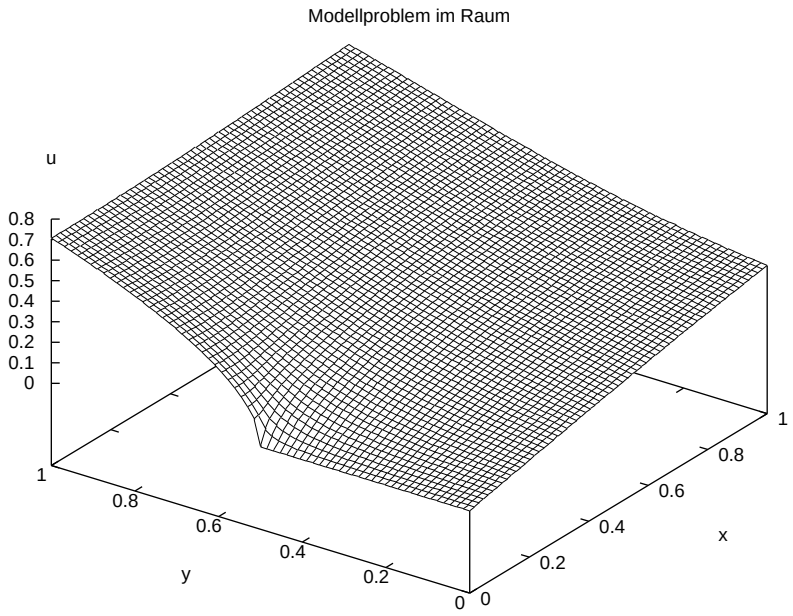


Abbildung 3.1: Lösung des elliptischen Modellproblems bei singulären Randbedingungen (3.1).

3.1.1 Basisfunktionen

Ein wichtiger Parameter ist der *Interpolationsoperator* I_h. Er wird durch die Basisfunktionen K definiert, die somit die approximierte Lösungfunktion bilden:

$$I_h = (K_{h,0} \quad K_{h,1} \quad \dots \quad K_{h,N}), \qquad I_h : U_h \to U \tag{3.2}$$

Oft wird bei Raum–Problemen mit linearen Interpolationen gearbeitet. Die dazu gehörenden Basisfunktionen sind die *Hut–Funktionen* 1. Ordnung, am Beispiel für den eindimensionalen Fall ohne uniforme Diskretisierung:

$$K_i(x) = \begin{cases} (x - x_{i-1})/(x_i - x_{i-1}) & x \in [x_{i-1}, x_i] \\ (x_{i+1} - x)/(x_{i+1} - x_i) & x \in (x_i, x_{i+1}] \\ 0 & \text{sonst.} \end{cases} \tag{3.3}$$

Hierbei ist x_i die Koordinate des Punktes i. Für die uniforme Diskretisierung gilt somit, daß $x_i = h \cdot i$. Konkret sind die Werte $u_h(i)$ also Koeffizienten von Hut–Funktionen. In mehrdimensionalen Raum–Problemen können analog dazu mehrdimensionale Hut–Funktionen als Basis verwendet werden.

Die Interpolation in einem Gitter

Die folgenden Umformungen sollten zeigen, daß die Interpolation in einem Gitter von den Basisfuntionen abhängig sind. Am Beispiel von Abbildung

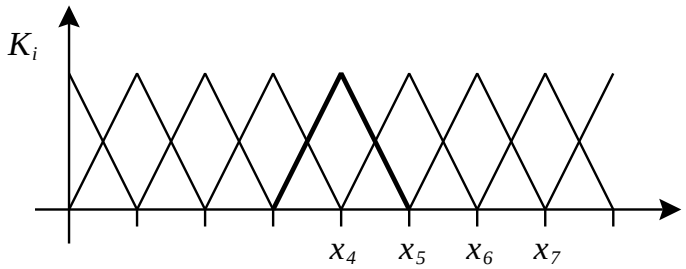


Abbildung 3.2: Funktionenbasis für $h = 1/8$ im eindimensionalen Fall bestehend aus Hut–Funktionen erster Ordnung.

3.2 wird die stetige Lösung u durch die Diskretisierung[1] $(u_0, \ldots, u_8)^t$ und den Hut–Funktionen K_i gebildet:

$$u(x) = \sum_{i=0}^{8} u_i \cdot K_i(x)$$

Angenommen, die Werte u_5 und u_7 werden aus ihren Nachbarpunkten *linear interpoliert*, so daß $u_5 = (u_4 + u_6)/2$ und $u_7 = (u_6 + u_8)/2$ ist. Die Summenformel kann darauf umgeformt werden auf

$$\begin{aligned} u(x) \quad = \quad & u_0 K_0(x) + \ldots + u_4 K_4(x) + \\ & u_4 K_5(x)/2 + u_6 K_5(x)/2 + u_6 K_6(x) + \\ & u_6 K_7(x)/2 + u_8 K_7(x)/2 + u_8 K_8(x). \end{aligned}$$

Daraus summieren sich für u_4, u_6 und u_8 die neuen Basisfunktionen

$$\begin{aligned} K_4' &= K_4 + K_5/2 \\ K_6' &= K_5/2 + K_6 + K_7/2 \\ K_8' &= K_7/2 + K_8. \end{aligned}$$

Die neuen Basisfunktionen für die Koeffizienten u_i mit $i = 0, \ldots, 4, 6, 8$ unterscheiden sind bei genauerer Betrachtung nur im Index von den Basisfunktionen für die nicht uniforme Diskretisierung in Abbildung 3.3.

Damit ein uniform diskretisiertes Gitter mit interpolierten Punkten zu einem nicht uniform diskretisierten Gitter äquivalent ist, muß die Interpolation durch die gewählten Basisfunktionen definiert sein.

[1] Da immer nur ein Gitter betrachtet wird, ist in diesem Abschnitt $u_i \hat{=} u_{h,i}$.

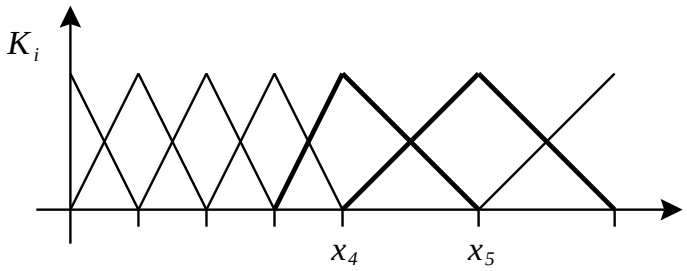


Abbildung 3.3: Nicht uniforme Funktionenbasis für sieben Punkte incl. Rand.

Die Prolongation

Die Prolongation I_{2h}^h von $2h$ nach h wird wie die Interpolation im Gitter durch die Basisfunktionen bestimmt. Eine diskrete grobe Lösung u_{2h} muß die gleiche kontinuierliche Funktion erzeugen wie die daraus Prolongierte feine Lösung u_h:

$$\begin{aligned} I_{2h}u_{2h} &= I_h u_h, \qquad u_h = I_{2h}^h u_{2h} \\ I_{2h}u_{2h} &= I_h I_{2h}^h u_{2h} \\ I_{2h} &= I_h I_{2h}^h \end{aligned}$$

Diese letzte Bedingung ist gleich dem Kompatibilitätskriterium der Trasportoperatoren (2.10). Wenn die Basisfunktionen die Interpolationsoperatoren I_h wie in (3.2) definieren, muß für jedes i die folgende Bedingung muß erfüllt sein, damit die Basisfunktionen mit der Prolongation konsistent sind. Die Summe der feinen Basisfunktionen

$$\sum_j K_{h,j} \cdot I_{2h}^h[j,i] = K_{2h,i} \tag{3.4}$$

muß in jedem Fall gleich der groben Basisfunktion $K_{2h,i}$ des Punktes i sein.

Die Kriterien für die kontinuierliche Interpolation I_h und die Prolongation I_{2h}^h werden für die folgenden virtuellen globalen Gitter verwendet.

3.1.2 Virtuelles globales Gitter

Um mit beliebig feinen Gittern rechnen zu können, kann man *virtuelle, globale Gitter* einsetzen [Rüd93a, LR97] und damit den Speicheraufwand klein halten. In einem solchen uniformen Gitter ist jeder Punkt verfügbar. Eine Teilmenge von Punkten kann jedoch virtuell sein. Das heißt, die Funktionswerte dieser Punkte werden aus ihren Nachbarn interpoliert. Derartige Punkte heißen *Geisterpunkte*. Die gekreuzten Punkte in Abbildung 3.4 stellen die Geisterpunkte u_5 und u_7 dar. Die Randpunkte sind grau hinterlegt

und werden nicht relaxiert. Andere Punkte werden *reale* Punkte oder lebende Punkte genannt.

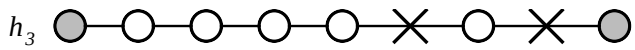


Abbildung 3.4: Das einfache virtuelle Gitter passend zur nicht uniformen Diskretisierung aus Abbildung 3.3 mit $h_3 = 1/8$ und zwei Geisterpunkten.

Der Punkt u_6 (analog auch u_8), diskretisiert mit $h_3 = 1/8$, hat, da seine Nachbarn u_5 und u_7 linear interpoliert werden, dieselbe Basisfunktion wie eine nicht uniforme Diskretisierung der verbleibenden Punkte. Dieses Verhalten wurde in Abschnitt 3.1.1 erläutert. Der mit $h_2 = 1/4$ diskretisierte Punkt an der gleichen Position auf dem gröberen Gitter hat ebenfalls die gleiche Basisfunktion. Folglich kann $u_{h_3,6}$ auch als Geist angesehen werden, der unmittelbar vom gröberen Gitter abstammt. Diese Vererbung als auch Interpolation im Gitter werden durch die Prolongation I_{2h}^{h} definiert und sind somit wiederum von den Basisfunktionen abhängig. u_8 wird in diesem Fall als Randpunkt und nicht als Geist gesehen. Abbildung 3.5 zeigt diese

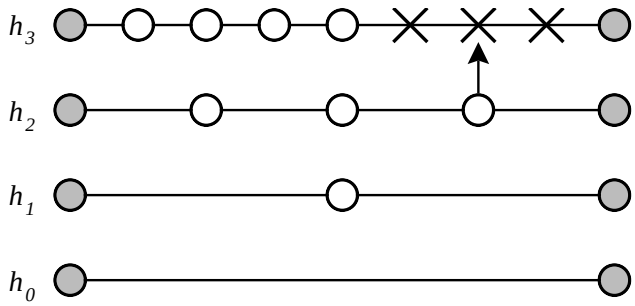


Abbildung 3.5: Die virtuelle Gitterhierarchie mit drei Geisterpunkten im feinsten Gitter, von denen einer aus einem gröberen Gitter übernommen wird.

zusätzliche Abhängigkeit und die so entstandene virtuelle Gitterhierarchie. Es existieren in dieser Hierarchie keine Geisterabstammungen zwischen den Gittern $u_{h_0} \dots u_{h_2}$.

Die Basisfunktionen der Lösung werden ähnlich dem Beispiel in Abschnitt 3.1.1 gebildet, jedoch wird nur K_6' und K_8' aus der groben Diskretisierung verwendet. Somit ist die Basis *äquivalent zu der einer nicht uniformen diskretisierung*. Die Festlegung der Diskretisierung auf $h = 2^{-n}$ ist nicht von Nachteil, da h trotzdem beliebig klein werden kann.

Multigridverfahren über virtuelle Gitterhierarchien

Auf einer wie in Abbildung 3.5 dargestellten Hierarchie kann unmittelbar ein Mehrgitter–Zyklus laufen [Rüd92]. Die einzelnen Schritte im Algorithmus bleiben unverändert — die Geister haben jedoch, wie am Beispiel des FAS–Schemas gezeigt, besondere Eigenschaften:

Die wichtigste Eigenschaft ist, daß *die Fein–zu–Grob–Defektkorrektur* an einem Punkt i gleich 0 ist,

$$\tau_h^{2h}(i) = 0,$$

wenn die *Galerkin Bedingung* (2.8) gilt und alle an $\tau_h^{2h}(i)$ beteiligten Punkte in u_h durch die Prolongation aus u_{2h} definiert sind:

$$u_h(k) = \left(I_{2h}^h u_{2h}\right)(k), \qquad \forall k \in \left\{k \mid (I_h^{2h} A_h)[i,k] \neq 0\right\}. \tag{3.5}$$

Die so definierten feinen *Geisterpunkte* müssen *konstant* den Wert der Prolongation von u_{2h} haben. Ein solcher Punkt kann entweder direkt durch die Prolongation bestimmt sein (ohne Interpolation) oder seine Interpolation im Gitter hängt nur von anderen Geisterpunkten ab. Durch eine geschickte Verfeinerungsstrategie kann und muß erreicht werden, daß, wenn ein betroffener Punkt ein Geist ist, er in jedem Fall einen konstanten Wert hat. Die später in Abschnitt 3.1.3 erläuterte Strategie erfüllt diese zusätzliche Bedingung. Abbildung 3.6 veranschaulicht die Bestimmung dieser geforderten Menge[2].

Beweisidee: Angenommen, das feinste Gitter u_h besteht zur Gänze aus Geistern, so ist $u_h = I_{2h}^h u_{2h}$ und somit

$$\begin{aligned} \tau_h^{2h} &= A_{2h}u_{2h} - I_h^{2h} A_h u_h \\ &= A_{2h}u_{2h} - I_h^{2h} A_h I_{2h}^h u_{2h}. \end{aligned}$$

Mit der Galerkin Bedingung $A_{2h} = I_h^{2h} A_h I_{2h}^h$ ergibt sich:

$$\begin{aligned} \tau_h^{2h} &= \left[I_h^{2h} A_h I_{2h}^h - I_h^{2h} A_h I_{2h}^h\right] u_{2h} \\ &= 0 \qquad \forall u_{2h}. \end{aligned}$$

Die Bedingung, daß Geister konstante Werte haben, läßt natürlich eine Relaxation nicht zu. Folglich läuft die Relaxation in den Mehrgitter–Algorithmen nicht mehr über alle Punkte, sondern nur mehr über die realen Punkte.

Man kann einen *Mehrgitter–Algorithmus*, der auf allen Ebenen mit Basisfunktionen einer Lösungsapproximation arbeitet, wie das FAS– oder HT–Schema ([Gri90, Gri94]), direkt verwenden.

[2] Für den Fall, daß I_h^{2h} ein „full weighting"– und A_h ein 5er–Stern–Operator ist gilt für das zweidimensionale elliptische Modellproblem, daß die Menge von feinen Geistern über dem Punkt i aus den 21 Punkte eines 5×5 Gitters ohne die vier Eckpunkte besteht.

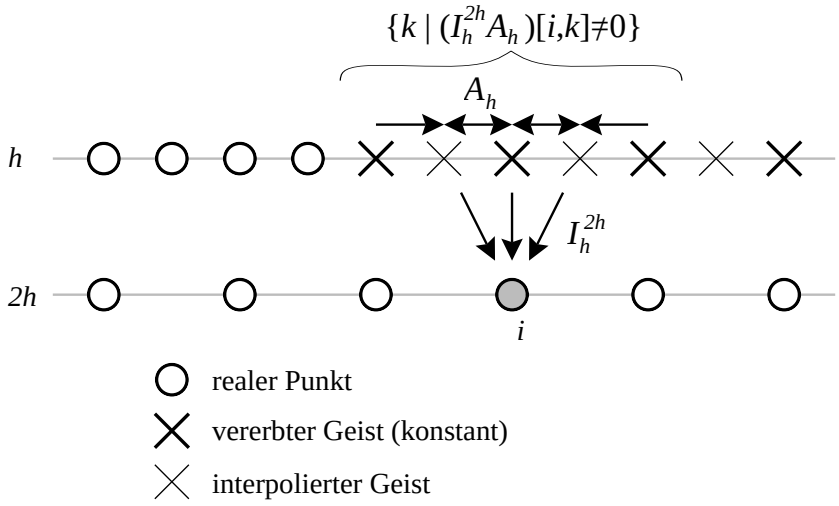


Abbildung 3.6: Die Bestimmung, welche Punkte aus u_h Geister mit konstantem Wert sein müssen, damit $\tau_h^{2h}(i) = 0$ ist.

Die Lösung läßt sich entweder durch die Auswertung *aller* Punkte auf der feinsten Ebene definieren, wobei auch alle Geister bestimmt werden müssen, oder als die Menge der Punkte mit den dazugehörenden Basisfunktionen, für die kein realer Punkt auf einem feineren Gitter existiert. Die Basisfunktionen an den Übergängen zwischen realen und Geister–Punkten müssen dann jedoch bestimmt werden.

Die Lösung des Problems mit einer virtuellen Gitterhierarchie ist äquivalent zu einer Lösung mit nicht uniformer Diskretisierung.

3.1.3 Adaptive Verfeinerung

Der Zwang einer uniformen Diskretisierung konnte mit virtuellen globalen Gittern abgelegt werden. Nun stellt sich aber noch die initiale Frage, wie man die eingangs geforderte Lösung u_h mit der *kleinsten Anzahl* von Unbekannten findet, die die exakte Lösung u^* hinreichend genau approximiert.

Der *adaptive Aufbau der Gitterhierarchie* sollte nun die Basisfunktionen finden, die maßgeblich an der Approximation der Lösung beteiligt sind. Der Ablauf zum Bestimmen der Lösung besteht wie in Abbildung 3.8 aus drei Schritten:

1. **Startgittererzeugung:** Um ein zuverlässiges Starten der Verfeinerungsstrategie zu gewährleisten, muß ein initiales Gitter erzwungen werden. Das initiale h ist von der Art des Problems und der Verfeinerung abhängig. Eventuelle Singularitäten müssen bereits beim Start diskretisiert werden.

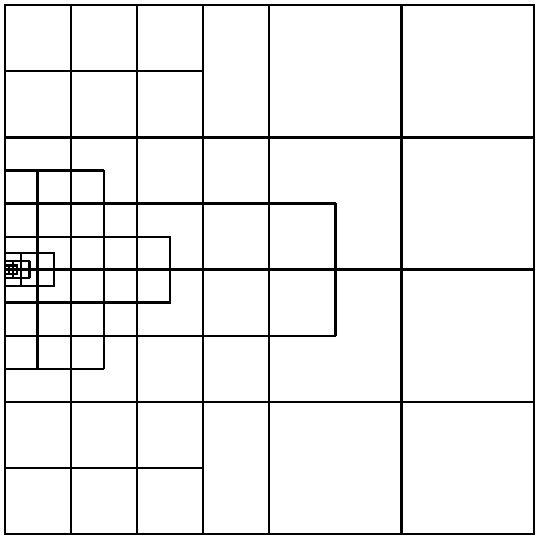


Abbildung 3.7: Adaptiv verfeinertes Gitter des singulären elliptischen Modellproblems in Abbildung 3.1.

2. **V–Zyklus:** Wie in Abschnitt 3.1.2 angerissen, kann ein herkömmlicher Mehrgitter–Algorithmus wie FAS– oder HT–Schema angewandt werden.

   Im *FAS–Algorithmus* gibt es folgendes zu beachten:

   - Das Relaxieren läuft nur über alle realen Punkte.
   - Die τ–Korrektur erfolgt nur bei denjenigen groben Punkten, die von einem realen, feineren Punkt beeinflußt werden (siehe Abschnitt 3.1.2).
   - Die rückwärtige u–Korrektur läuft nur zu den realen Punkten im feineren Gitter.

3. **Adaptive Verfeinerung:** An dieser Stelle wird aus einem Gitter u_h ein noch feineres Gitter $u_{h/2}$ erzeugt, bei dem die benötigten realen Punkte zu ermitteln sind. Um zu wissen, welcher Punkt nun real wird und welcher ein Geist bleibt, benötigt man ein *Kriterium*, das entscheidet, an welchen Stellen verfeinert wird.

Standard–Kriterium

Als Kriterium wurde in [LR97] vorgeschlagen, den Wert $\hat{u}_{h/2}(i)$ eines feinen Punktes i durch die Interpolation $I_h^{h/2}$ zu initialisieren und ihn einmal zu relaxieren:

$$u_{h/2} \leftarrow \text{relax}(\hat{u}_{h/2})$$

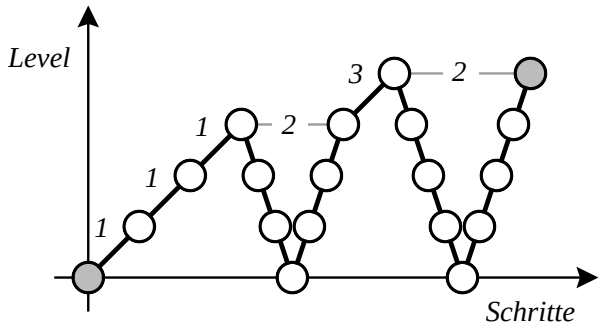


Abbildung 3.8: Die verschiedenen Schritte im Ablauf: Startlevel erreichen (1), V–Zyklus durchführen (2) und adaptiv verfeinern (3).

Liegt die Differenz $|u_{h/2}(i) - \hat{u}_{h/2}(i)|$ unter einem Schwellwert, so kann der Punkt i ein Geist bleiben. Dieses Kriterium ist äquivalent zu einer **Bewertung des skalierten Residuums** ([Rüd93b]) der Interpolation. Das *skalierte Residuum* Θ ist definiert als

$$\Theta = D^{-1}(f - Au) \tag{3.6}$$

und gleich der Differenz $u_{h/2} - \hat{u}_{h/2}$. Die Bewertung des skalierten Residuums funktioniert auch, wenn die grobe Lösung nicht exakt ist. Jedoch können Probleme zwischen der Interpolation und dem Diskretisierungsoperator auftreten. Wird zum Beispiel bilinear interpoliert, und für den Laplace–Operator Δ der diskrete 5er–Stern

$$\frac{1}{h^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}, \qquad h_x = h_y = h$$

verwendet (für $h_x \neq h_y$ in (2.6) definiert), so haben die feinen Punkte $u_{h/2}(x, y)$ an den geraden Stellen (3.7) über einem groben Punkt und an den doppelt interpolierten Stellen (3.8) zwischen vier groben Punkten folgende skalierte Residuen:

$$\Theta_{h/2}(x, y) = \frac{\Theta_h(x, y)}{2} + \frac{3h^2 f_{h/2}(x, y)}{4} \qquad \forall (x, y),\ x \equiv_h 0,\ y \equiv_h 0 \tag{3.7}$$

$$\Theta_{h/2}(x, y) = -\frac{h^2 f_{h/2}(x, y)}{4} \qquad \forall (x, y),\ x \equiv_h \tfrac{h}{2},\ y \equiv_h \tfrac{h}{2} \tag{3.8}$$

Man sieht, daß die skalierten Residuen an den doppelt interpolierten Werten (3.8) nur mehr von f abhängen und damit bei homogenen Problemen gleich 0 sind. An den geraden Punkten (3.7) läßt sich das skalierte Residuum von den gröberen Pendants zusammen mit f errechnen. Daraus resultiert

ein Unterschied in der Größenordnung der skalierten Residuen abhängig von der relativen Position der Feingitterpunkte.

Ein Effekt dieses Zusammenspiels ist, daß eine einfache Bewertung der skalierten Residuen der feinen Punkte bei *Standard–Verfeinerung* in x– und y–Richtung nicht funktioniert. Im Falle der *Semi–Verfeinerung* alternierend in der Richtung funktioniert das Verfahren bedingt. Um systematischen Fehlern im Zusammenspiel zwischen Diskretisierung und Interpolation aus dem Weg zu gehen, ist ein anderes Verfahren notwendig.

Bewertung am groben Punkt

Ein grober Punkt i in u_h wird während eines FAS–Mehrgitterzyklus von der Menge feiner Punkte

$$before(i) = \left\{ k \mid I_{h/2}^{h}[i,k] \neq 0 \right\} \tag{3.9}$$

durch die τ–Korrektur modifiziert und reicht die u–Korrektur an eine Menge von feineren Punkten

$$after(i) = \left\{ k \mid I_{h}^{h/2}[k,i] \neq 0 \right\} \tag{3.10}$$

zurück. Der erste Schritt, die τ–Korrektur wird durch die Restriktion $I_{h/2}^{h}$ bestimmt, der zweite Schritt, die u–Korrektur durch die Prolongation $I_{h}^{h/2}$. Aus (3.9) und (3.10) ergibt sich die Menge von *beteiligten Punkten*

$$coop_{h}^{h/2}(i) = \left\{ k \mid I_{h/2}^{h}[i,k] \neq 0 \right\} \cup \left\{ k \mid I_{h}^{h/2}[k,i] \neq 0 \right\}. \tag{3.11}$$

Im Modellproblem für den Raum ist bei Standard–Verfeinerung die feine Punktmenge (3.11) ein 3×3 Feld von Punkten über einem groben Punkt.

Der *Fehlerschätzer* verwendet nun den „restringierten Betrag des skalierten Residuums“

$$\Theta_{h/2}^{h} = I_{h/2}^{h} \left| D_{h/2}^{-1} \left(f_{h/2} - A_{h/2} \hat{u}_{h/2} \right) \right|, \quad \hat{u}_{h/2} = I_{h}^{h/2} u_h, \tag{3.12}$$

um für jeden Punkt i abzuschätzen, ob die Menge von Punkten $coop_{h}^{h/2}(i)$ um i verfeinert wird. Der Term (3.12) ist äquivalent zum „skalierten restringierten Betrag des Residuums“

$$\Theta_{h/2}^{h} = |a_{h/2}^{-1}| \left(I_{h/2}^{h} \left| f_{h/2} - A_{h/2} \hat{u}_{h/2} \right| \right), \quad \hat{u}_{h/2} = I_{h}^{h/2} u_h \tag{3.13}$$

der im Algorithmus in Abbildung 3.9 verwendet wird, wobei die Diagonalmatrix durch einen konstanten Werte $a_{h/2}$ definiert ist:

$$D_{h/2} = (a_{i,i}) \quad a_{i,i} = a_{h/2} \quad \forall i$$

```
proc AdaptiveMultigrid(u_h , δ_min)
    V–Cycle(u_h)
    create u_{h/2}
    P' ← RealPoints(u_h)
    while(P' ≠ ∅)
        pick p_h ∈ P'
        p_{h/2} ← u_{h/2}(p_h.getPos())
        P' ← P' \ p_h
        /* errechne analog zu (3.13): */
        Θ^h_{h/2} ← |a^{-1}_{h/2}| * ( I^h_{h/2} |f_{h/2} − A_{h/2} u_{h/2}| )
        if(Θ^h_{h/2}(p_h.getPos()) ≥ δ_min)
            /* verfeinere mit Hilfe von (3.11): */
            foreach(p ∈ coop(p_{h/2}))
                p.kind ← real
            end foreach
        end if
    end while
    if(RealPoints(u_{h/2}) ≠ ∅)
        AdaptiveMultigrid(u_{h/2}, δ_min)
    end if
end proc.
```

Abbildung 3.9: Der Algorithmus der adaptiven Verfeinerung.

Das zweite Verfahren, bei dem die Beträge der skalierten Residuen des feinen Gitters auf das grobe Gitter gebracht werden, ist in der Art der Verfeinerung äquivalent zum ursprünglichen Verfahren der Bewertung der skalierten Residuen der feinen Punkte. Es funktioniert jedoch unabhängig von der Verfeinerungstrategie und von Diskretisierungs–Operatoren.

3.2 Adaptive Relaxation

In [Rüd93b, Rüd93a] und sehr anschaulich in [PR93] wurde ein *adaptiver Glätter* bei einem Mehrgitterverfahren eingesetzt. Das dort beschriebene Verfahren heißt *Fully Adaptive Multigrid — FAMe*, da es auch eine adaptive Verfeinerung bietet.

Der adaptive Glätter „*Gauß'sche adaptive Relaxation*“ beruht auf der Annahme, daß das Residuum einer Approximation u in jedem Punkt eine Schranke unterschreiten muß, damit die Lösung hinreichend genau ist. Das Lemma 3.1.1 aus [Rüd93b] beschreibt, wie groß die Differenz der Energienorm des Fehlers *vor* und *nach* einem einzigen Relaxierungsschritt am Punkt

i ist[3], nämlich

$$||u - u^*||_E^2 - ||u' - u^*||_E^2 = a_{i,i}\Theta_i(u)^2. \tag{3.14}$$

Wobei $u'_i \leftarrow u_i + \Theta_i(x)$ ein einzelner Relaxierungsschritt am Punkt i ist. Das skalierte Residuum ist für einen Punkt analog zu (3.6) definiert als

$$\Theta_i = \Theta_i(u) = a_{i,i}^{-1} e_i^T (f - Au),$$

wobei e_i der i-te Einheitsvektor ist. Das zitierte Lemma aus (3.14) wird klar, wenn man bedenkt, daß das skalierte Residuum nichts anderes ist, als die Differenz eines Wertes u_i vor und nach einem Relaxierungsschritt: $\Theta_i = u'_i - u_i$.

Die Idee der adaptiven Relaxation ist, daß Werte mit einem Θ_i kleiner als Θ_{max} nicht relaxiert werden. Dadurch ergibt sich die Menge der Punkte, die relaxiert werden müssen, die *strikt aktive Menge*:

$$S(\Theta_{max}, u) \stackrel{\text{def}}{=} \{i \mid |\Theta_i(u)| \geq \Theta_{max}\}$$

Diese Menge zu berechnen ist gleich aufwendig, wie alle Punkte einmal zu relaxieren. Deshalb wird als *aktive Menge* $\tilde{S}$ verwendet, die folgende Bedingung erfüllen muß:

$$S(\Theta, u) \subseteq \tilde{S}(\Theta, u) \subseteq \{1, 2, \ldots, n\}.$$

Die aktive Menge kann zu Beginn einfach als die Menge aller Punkte $\{1, 2, \ldots, n\}$ gesetzt werden. Man kann mit etwas größerem Aufwand eine kleinere Menge finden, wie in [Löt96] ausführlicher beschrieben wird. Es können etwa nur aus Punkte in die aktive Menge aufgenommen werden, die sich im Lauf anderer Algorithmen verändert haben, wie auch Beispiel neu hinzugekommene Punkte, korrigierte Punkte nach einem V–Zyklus, etc. . . .

Der Algorithmus der sequentiellen adaptiven Relaxation

Der Algorithmus in Abbildung 3.10 wird mit einer initialen aktiven Menge $\tilde{S}$ gestartet. So lange noch ein Punkt in der Menge enthalten ist, wird einer ausgewählt und überprüft, ob sein skaliertes Residuum Θ_i über einer Schranke Θ_{max} liegt. Ist dies der Fall, so wird der Punkt relaxiert und die *von seinem Wert abhängigen* Nachbarpunkte kommen in die aktive Menge.

Nach [PR93, Seite 7] kann sich die aktive Menge bei dem Beispiel in Abbildung 3.11 folgendermaßen verändern: Liegt $|\Theta_i|$ eines Punktes unter dem maximal erlaubten Θ_{max}, so wird er aus der aktiven Menge genommen (rechter Weg). Wird er jedoch relaxiert, kann er nicht mehr in der aktiven

[3] Für die folgenden Definitionen wird die Notation aus [Rüd93b] ohne Diskretisierungsindex h und mit dem Vektorindex i verwendet, da keine kontinuierlichen Funktionen verwendet werden. $u_i \hat{=} u_h(i)$.

```
proc SequentialAdaptiveRelax( Θ_max, u, S̃ )
    /* Θ_max ... maximal erlaubtes skaliertes Residuum */
    /* S̃ ... Aktive Menge der zu relaxierenden Knoten */
    while(S̃ ≠ ∅)
        pick node i ∈ S̃
        S̃ ← S̃ \ {i}
        Θ_i ← 1/a_{i,i} ( f_i − Σ_{j≠i} a_{i,j} u_j ) − u_i
        if(|Θ_i| > Θ_max)
            u_i ← u_i + Θ_i
            S̃ ← S̃ ∪ {j | j ≠ i ∨ a_{i,j} ≠ 0}
        end if
    end while
end proc.
```

Abbildung 3.10: Die sequentielle adaptive Relaxation (aus [Rüd93b, Rüd93a]).

Menge sein, da sein neues Θ_i gleich 0 ist[4]. In diesem Fall kann sich das Θ der benachbarten Punkte geändert haben. Deshalb werden sie in die aktive Menge $\tilde{S}$ eingetragen (linker Weg).

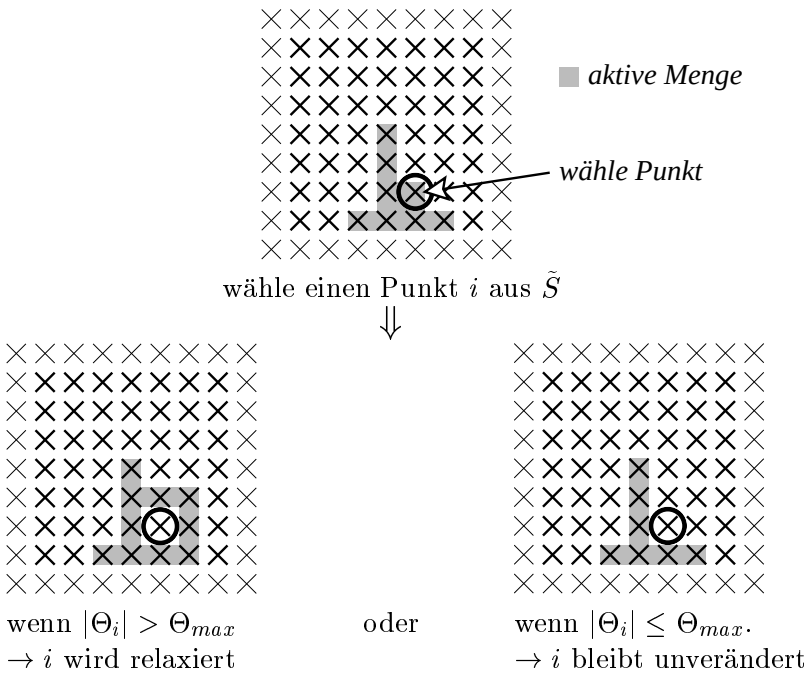
Im Vergleich zur herkömmlichen Gauß–Seidel– oder Jacobi–Relaxation funktioniert dieses Verfahren mit der *notwendigen* Anzahl von Relaxationen, um ein Problem zu approximieren. Als Vorglätter in einem V–Zyklus funktioniert dieser Algorithmus nur bedingt, da als Abbruchkriterium der gesamte Fehler klein gehalten werden muß. Der Vorglätter soll und kann aber nur die hochfrequenten Anteile des Fehlers effizient eliminieren.

Sortierte aktive Menge

Die aktive Menge ist abstrakt definiert. Es gibt im Algorithmus 3.10 keine strikte Reihenfolge, in der die Punkte aus der Menge gewählt werden sollen. So kommt es zu Problemen, wenn zwei Punkte p_1 und p_2 immer vor allen anderen Punkten aus der Menge gewählt werden und sie sich gegenseitig nach der jeweiligen Relaxation wieder in die Menge eintragen. Man muß also sicherstellen, daß nach Art der Gauß–Seidel Relaxation alle Punkte aus der Menge abgearbeitet werden, bevor die wiedereingetragenen Punkte gewählt werden.

Es gibt verschiedene Arten, dieses Verhalten zu Implementieren. Eine einfache ist es, einen *FIFO–Puffer* (first in—first out, auch „Queue“) als

[4]Das gilt nur bei ungewichteten Verfahren. Ist $\omega \neq 1$, so muß der Punkt i abermals in die aktive Menge aufgenommen werden, wenn $|\Theta_i \cdot (1-\omega)| > \Theta_{max}$ ist.



(Graphik übernommen aus [PR93])

Abbildung 3.11: Veränderung der aktiven Menge je nach Θ_i.

aktive Menge zu verwenden. Ein neueingetragener Punkt wird erst nach allen bereits in der Schlange vorhandenen Punkten verwendet. Die konkrete Implementierung wird in 5.3 ab Seite 65 besprochen.

Für *parabolische Probleme* wie dem Modellproblem für die Zeit aus Kapitel 4 gibt es zusätzliche Informationen über die Abhängigkeiten eines Punktes. Darauf wird im Abschnitt 4.2 detailliert eingegangen.

Kapitel 4

Zeitabhängige parabolische Probleme

Bisher wurde nur das zweidimensionale Raum–Problem (2.1) untersucht. Probleme in Raum und Zeit wurden noch nicht näher betrachtet. Das Zeitabhängige Modellproblem ist wie in (2.2) definiert als

$$u_t - a u_{xx} = f.$$

Für die folgenden Betrachtungen wird $a = 1$ konstant gewählt und $\Omega = [0,1]^2$ als der zweidimensionale Raum $x \times t$ verwendet. Die Zeit t als auch der eindimensionale Raum x sind beschränkt und werden hier auf das Intervall $[0,1]$ normiert. Der Definitionsbereich Ω, der Rand Γ und die Diskretisierung werden in Abbildung 4.1 dargestellt. Für die Tests des Modellproblems wird ein größeres Zeitintervall $0 \leq t \leq T = 8$ verwendet, weshalb die Zeit mit $1/8$ skaliert werden muß. Der Randbereich $\Gamma = \partial\Omega$ wird mit der Randfunktion g und Dirichlet–Bedingungen definiert als

$$u(x,t) = g(x,t), \qquad (x,t) \in \Gamma.$$

Um auch im Raum$\times$Zeit Modellproblem eine Singularität zu erzeugen, wurde g definiert als

$$g(x,t) = \begin{cases} 0 & : \quad x = 0 \\ 1 & : \quad x = 1 \\ 0 & : \quad sonst. \end{cases}$$

In Abbildung 4.2 ist die Lösung über den gesamten Definitionsbereich aufgetragen. Das Beispiel zeigt die zeitabhängige Wärmeverteilung eines Stabes. Am Anfang ist er gleichmäßig kalt, nur an der rechten Seite gibt es eine konstante Wärmequelle, die mit der Zeit einen gleichmäßigen Wärmeverlauf im Stab erzeugt.

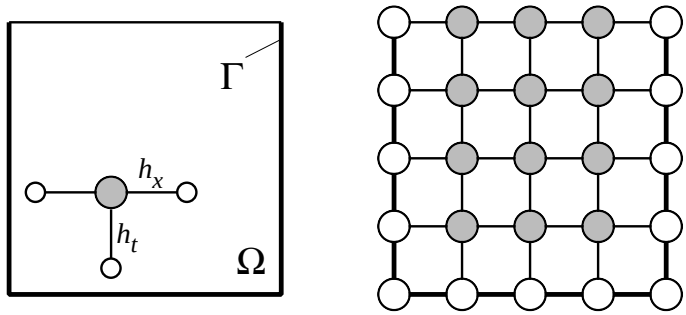


Abbildung 4.1: *Links* der Definitionsbereich Ω, der Rand Γ und ein mit h diskretisierter Punkt des parabolischen Raum×Zeit Modellproblems. *Rechts* dasselbe gleichmäßig diskretisierte Problem mit grauen inneren Punkten und weißen Randpunkten. Die Zeit ist jeweils von unten nach oben aufgetragen.

4.1 Herkömmliche Lösungsansätze

Zeitabhängige Probleme werden meist als eine Reihe von untereinander abhängigen Problemen gesehen. Die einfachste Methode, die Zeitkomponente zu diskretisieren, ist die sogenannte *„Rückwärts–Euler–Methode“* oder *implizite* Euler–Methode. Dabei wird $u_t^{(t)}$ approximiert als

$$u_t^{(t)} \approx \frac{u^{(t)} - u^{(t-\Delta t)}}{\Delta t}, \qquad \lim_{\Delta t \to 0} u_t^{(t)}(x) = u_t(x,t).$$

Dadurch entsteht das in der Zeit diskretisierte Gleichungssystem

$$\begin{aligned} \frac{u^{(t)} - u^{(t-\Delta t)}}{\Delta t} - u_{xx}^{(t)} &= f^{(t)}, \qquad u^{(t-\Delta t)} \text{ ist gegeben} \\ &\Downarrow \\ \frac{u^{(t)}}{\Delta t} - u_{xx}^{(t)} &= g^{(t)} = f^{(t)} + \frac{u^{(t-\Delta t)}}{\Delta t}. \end{aligned} \tag{4.1}$$

Dieses Gleichungssystem wird dann in der Raumkoordinate x diskretisiert und kann Schritt für Schritt mit herkömmlichen Mehrgitterverfahren gelöst werden. So entsteht eine Kette von diskreten elliptischen linearen PDEs. Ein Mehrgitterzyklus bearbeitet immer nur einen Zeitschritt.

Für die Zeitdiskretisierung stehen noch andere Methoden, teils von höherer Ordnung, zur Verfügung. Die *Backward Differentiation Formula* erster Ordnung BDF1 ist äquivalent zur beschriebenen impliziten Euler–Methode. Eine Methode zweiter Ordnung ist die *Crank–Nicolson* oder Trapezoid–Methode und auch die BDF zweiter Ordnung BDF2, die jedoch drei Zeitpunkte für eine Diskretisierung benötigt.

Der immense Nachteil dieser Schrittverfahren ist, daß zu jedem Zeitschritt das gesamte Problem gelöst werden muß. Eine nicht uniforme Zeitschrittweite entschärft das Problem teilweise. In Verbindung mit einem

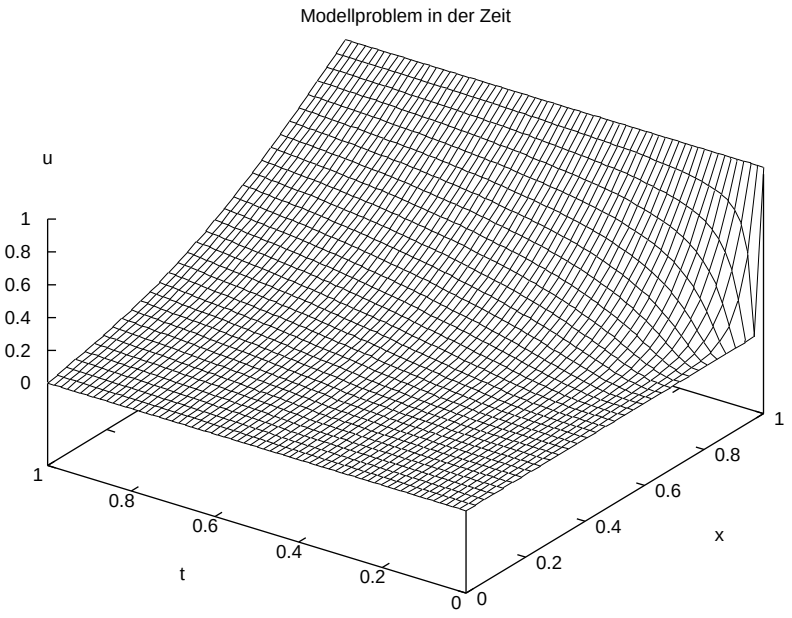


Abbildung 4.2: Lösung des zeitabhängigen singulären parabolischen Modellproblems.

Schrittweitenschätzer ergibt sich eine sehr große Effizienzsteigerung. Bei nicht homogenen Problemen wie zum Beispiel der Berechnung von Wettermodellen gibt es jedoch meist zu jedem Zeitschritt eine Stelle mit lokalen Veränderungen, so daß bei solchen Verfahren ein kleiner Zeitschritt für das ganze Gebiet erforderlich ist.

Partielle Zeitschritte. S. McCormick schlägt in [McC92, ab Seite 40] für ein Teilgebiet kleinere Zeitschritte vor. Der Rand des nicht über dem gesamten Gebiet vollzogenen Zeitschrittes wird durch *Sklavenpunkte* erzeugt. Der Vergleich zur herkömmlichen partiellen Verfeinerung in Abschnitt 3.1 ist berechtigt. Diese Idee wurde nun teilweise aufgegriffen und mit dem herkömmlichen MG–Zyklus vereint.

4.2 Eingitter–Verfahren in Raum und Zeit

Die unmittelbare Anwendung eines Mehrgitterverfahrens auf ein Problem aus Raum×Zeit bedarf zuerst einer gründlichen Untersuchung der iterativen Eingitter–Verfahren.

Gleichungssystem. Es muß ein einziges Gleichungssystem geschaffen werden — die Zerlegung in mehrere einzelne elliptische PDEs ist obsolet. Das gesamte Problem soll als ganzes diskretisiert werden, wobei die Diskretisierung h für das Modellproblem aus h_x und h_t besteht[1]. Die in der Gleichung (4.1) verwendete implizite Methode BDF1 für die Zeit kann mit einer herkömmlichen Raumdiskretisierung der zentralen Differenzen kombiniert werden. Das Modellproblem $u_t(x,t) - u_{xx}(x,t) = f(x,t)$ wird diskretisiert in

$$\frac{u_h(x,t) - u_h(x,t-h_t)}{h_t} - \frac{u_h(x-h_x,t) - 2u_h(x,t) + u_h(x+h_x,t)}{h_x^2} = f_h(x,t). \tag{4.2}$$

Diese Gleichung ist wie (2.6) als Sternoperator darstellbar[2]:

$$\begin{bmatrix} 0 & 0 & 0 \\ -h_x^{-2} & 2h_x^{-2} + h_y^{-1} & -h_x^{-2} \\ 0 & -h_y^{-1} & 0 \end{bmatrix} \tag{4.3}$$

Der Wert λ, definiert als $\lambda_h = h_t/h_x^2$, formt den Operator auf die etwas einfachere, in [HV95] verwendete Form

$$\frac{1}{h_t} \begin{bmatrix} 0 & 0 & 0 \\ -\lambda_h & 2\lambda_h + 1 & -\lambda_h \\ 0 & -1 & 0 \end{bmatrix} \tag{4.4}$$

um. Die Operatoren für die anfangs erwähnten Diskretisierungsverfahren höherer Ordnung lauten demnach für Crank–Nocolson

$$\frac{1}{h_t} \begin{bmatrix} 0 & 0 & 0 \\ -\lambda_h/2 & \lambda_h + 1 & -\lambda_h/2 \\ -\lambda_h/2 & \lambda_h + 1 & -\lambda_h/2 \end{bmatrix} \tag{4.5}$$

und für die Rückwärts–Differenzierungsformel BDF2 ergibt sich der größere Fünferstern

$$\frac{1}{h_t} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -\lambda_h & 2\lambda_h + 2/3 & -\lambda_h & 0 \\ 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 1/2 + 2/3 & 0 & 0 \end{bmatrix}, \tag{4.6}$$

der natürlich nicht in der ersten Punktmenge nahe dem Rand angewandt werden kann.

[1] Der Artikel [HV95] untersucht die Diskretisierung von Raum×Zeit–Problemen sehr detailliert.

[2] Bei Raum–Zeit Sternoperatoren wird x von links nach rechts aufgetragen, t von unten nach oben.

Iterative Löser. Das so definierte Gleichungssystem hat nun einige spezielle Eigenschaften. So sind in einem elliptischen Raum×Raum–Problem bei der Diskretisierung mittels zentraler Differenzen die vier Punkte mit Abstand h_x beziehungsweise h_y von einem Wert $u_h(x, y)$ abhängig. Ändert sich dieser Wert, so muß das Residuum der vier nächsten Nachbarn neu berechnet werden. Die adaptive Relaxation aus Abschnitt 3.2 verwendet die *Nachbarn des Punktes i*, definiert durch

$$\{j | j \neq i \vee a_{i,j} \neq 0\},$$

indem sie in die aktive Menge der zu relaxierenden Punkte eingetragen, sozusagen *„aktiviert“*, werden.

Die Nachbarn sind bei Zeitdiskretisierungen nicht in jeder Richtung zu finden. Vielmehr wird davon ausgegangen, daß eine Approximation von u_t nur aus der Vergangenheit und dem aktuellen u besteht. Durch diese Annahme ist ein Punkt nie von seinen zeitlich nachfolgenden Punkte abhängig. Also kann von einer anderen Strategie für die aktive Menge ausgegangen werden. Ein Punkt zum Zeitpunkt t_1 kann immer *vor* einem Punkt zu einem späteren Zeitpunkt $t_2 > t_1$ relaxiert werden, da er nur von einem Punkt mit $t_3 \leq t_1$ in die aktive Menge zurückgebracht werden kann. Daraus folgt, daß die Wahl von Elementen aus der aktiven Menge zeitlich sortiert erfolgen kann.

Der parabolische Charakter des Problems erlaubt hier also eine Optimierung der Glättung. Verwendet man den adaptiven Glätter zur Lösung, so arbeitet dieser zeitsortierte Glätter Zeitschritt für Zeitschritt einzeln durch. Er entspricht dadurch in etwa dem herkömmlichen Zeitschrittverfahren ohne Mehrgittermethode.

Als Implementierung wurde eine *priorisierte Warteschlange* wie aus Abbildung 4.3 verwendet, in der jeder Zeitpunkt eine Priorität darstellt. Anfangs werden alle Punkte mit ihren jeweiligen Prioritäten, den Zeitpunkten, in die Warteschlange eingefügt. Die Abarbeitung der Punkte erfolgt zeitlich sortiert. Da kein Punkt durch einen Punkt eines späteren Zeitpunktes aktiviert werden kann, leert sich die aktive Menge in Zeitrichtung. Der Effekt ist ein zeilenweises Vorgehen über das *Raum* × *Zeit*–Problem. Jede Zeitstufe wird adaptiv relaxiert, bevor die Punkte der nächsten Zeitstufe bearbeitet werden.

Die Relaxation funktioniert damit uneingeschränkt auch für parabolische Probleme. In Zusammenhang mit einem Mehrgitterverfahren sind jedoch noch andere Effekte zu beachten.

4.3 Mehrgitterverfahren in Raum und Zeit

Das durch die gewählte Diskretisierung entstandene Gleichungssystem soll mit einem Mehrgitterverfahren gelöst werden. Alle Schritte des implemen-

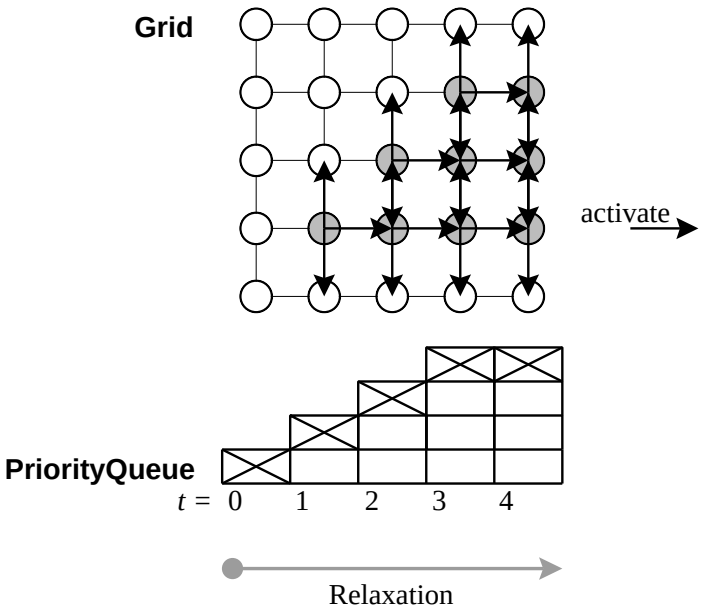


Abbildung 4.3: Schemenhafte Darstellung der priorisierten Warteschlange und der Wiedereintragung von realen Punkten in die aktive Mange (*activate*) für ein 5×5 Gitter und neun realen (grauen) Punkten. Die Zeit ist von links nach rechts aufgetragen.

tierten modifizierten FAMe–Algorithmus müssen auf die Tauglichkeit bei parabolischen Problemen untersucht werden.

4.3.1 Die Glättung im Mehrgitterzyklus

Die Glättung an sich funktioniert wie im Abschnitt 4.2 besprochen ohne besondere Einschränkungen. Im Mehrgitter–Zyklus muß das *Verhalten des Vorglätters* neu untersucht werden. In der Abbildung 4.4 wird die Konvergenz des Residuums in der Maximumsnorm $||f - Au||_\infty$ in bezug zur Anzahl der V–Zyklen veranschaulicht. Dabei wird das nicht adaptive FAS–Schema mit v_1 Vor– und v_2 Nachglättungsschritten verwendet, wobei $v_1 + v_2$ für den Vergleich konstant gewählt wurde.

Die mittlere Konvergenzrate des *elliptischen Modellproblems* ist im Fall von $v_1 = 1$ und $v_2 = 3$ mit 0.055 am besten. Das maximale Residuum wird dabei ab dem fünften Zyklus zum kleinsten im Vergleich. Für das *parabolische Modellproblem* ist die Vorglättung weniger von Bedeutung. Eine reine Nachglättung ($v_1 = 0$, $v_2 = 4$) hat die kleinste mittlere Kovergenzrate (0.167) und nach jedem Schritt das kleinste Residuum zur Folge. Auf die schlechte MG–Konvergenz des parabolischen Problems wird später eingegangen.

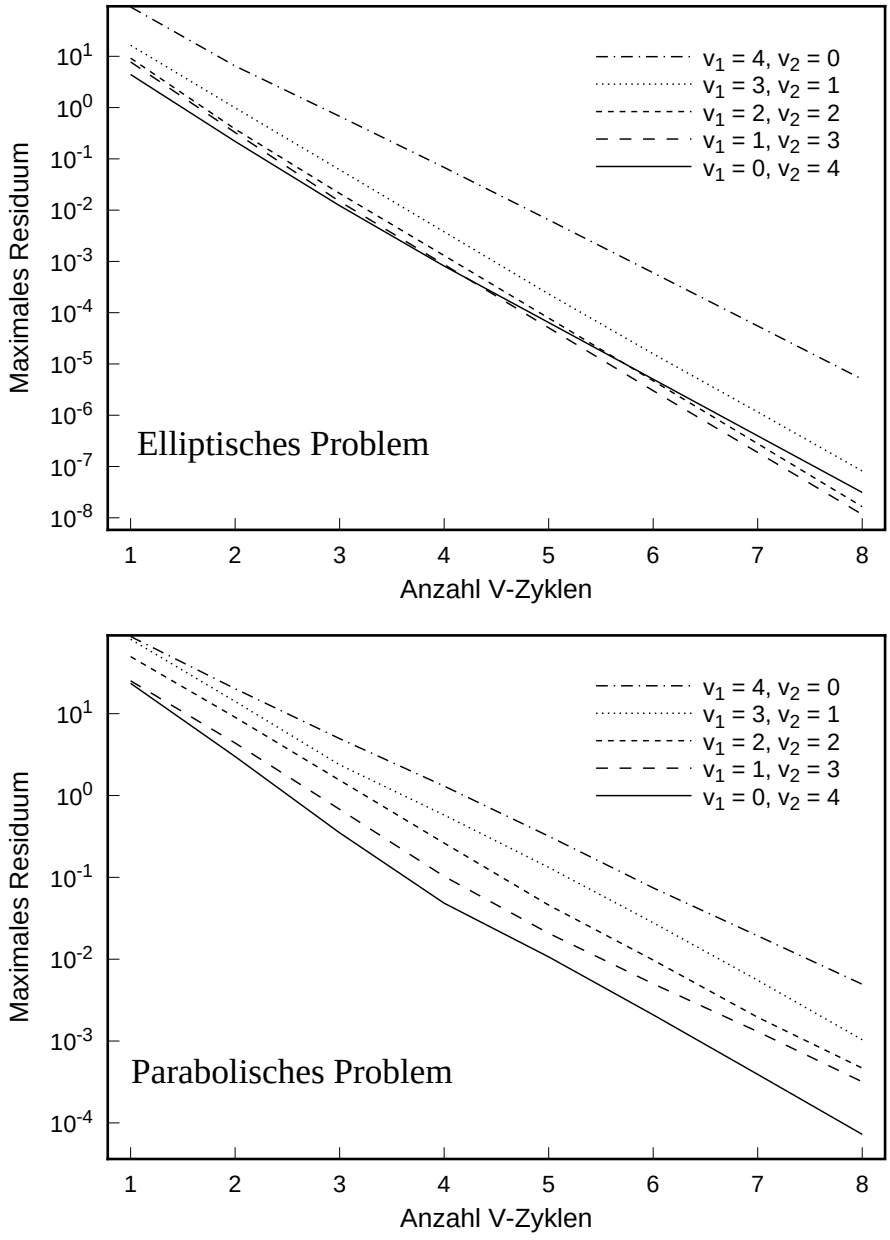


Abbildung 4.4: Das Verhältnis von ungewichteter Nach– zu Vorglättung in einem FAS–Verfahren am Beispiel der Modellprobleme und $v_1 + v_2 = 4$.

Adaptive Nachglättung. Die adaptiven Glättungsverfahren müssen im Kontext eines MG–Verfahrens für parabolische Probleme ebenfalls neu bewertet werden. Dafür wurde als Maß für die Effizienz die mittlere Anzahl der durchgeführten Relaxationen je Punkt der Gitterhierarchie in Relation zur erreichten Genauigkeit gesetzt. Die erreichte Genauigkeit ist wiederum das maximale Residuum.

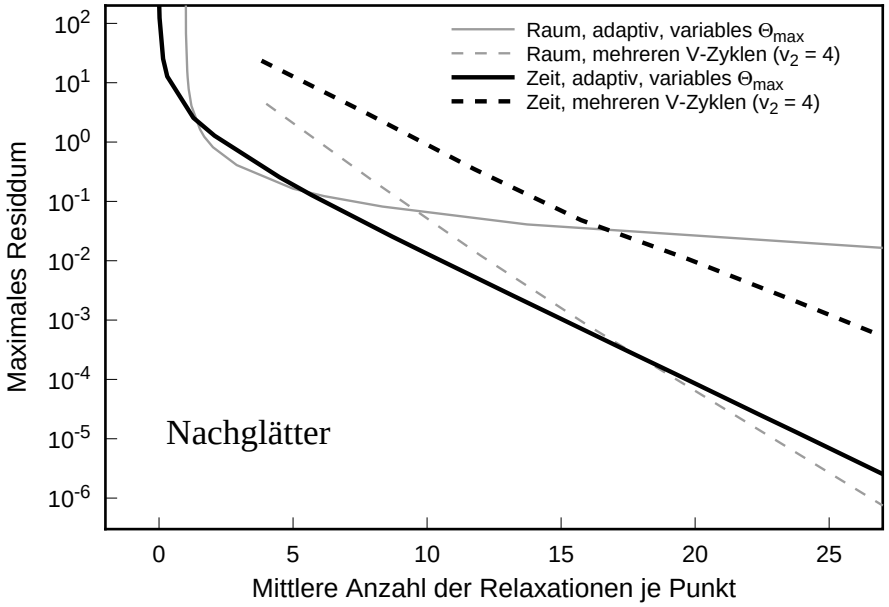


Abbildung 4.5: Die durch den Nachglätter beeinflußte Konvergenz in bezug auf die mittlere Anzahl der Relaxationen je Punkt ($v_1 = 0$). Das adaptive Verfahren benötigt immer nur einen V–Zyklus, um das gewünschte Θ_{max} zu unterschreiten.

Abbildung 4.5 zeigt für das schwarz dargestellte Zeitproblem und dem grau dargestellten Raumproblem das maximale Residuum in bezug auf die Anzahl der durchgeführten Relaxationen. Die durchgezogenen Linien zeigen die Verwendung eines adaptiven Nachglätters mit verschieden gewähltem Θ_{max}, die gestrichelten Linien die Anwendung der herkömmlichen Nachglättung mit v_2 Iterationen. Für das Zeitproblem ist der adaptive Nachglätter eine durchgängig bessere Wahl. Die Genauigkeit der Lösung $|u_h - u_h^*|$ läßt sich mit dem adaptiven Glätter beliebig steigern, wobei die Effizienz immer über dem herkömmlichen Glätter liegt[3]. Dieses Verhalten legt die Vermutung nahe, daß das gewählte Verfahren sich wie ein *direkter Löser* für parabolische Probleme der Art des Modellproblems verhält.

[3]Die Genauigkeit wurde bis auf ein skaliertes Residuum von $\Theta_{max} = 10^{-20}$ getestet. Ab dieser Genauigkeit verändert sich nichts mehr. Jede 10er Potenz benötigte nur wenige zusätzliche Relaxationen je Punkt.

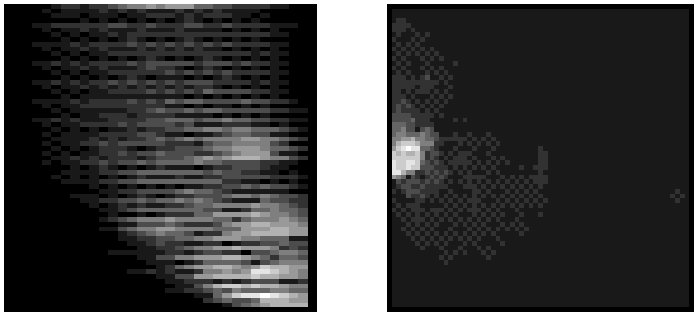


Abbildung 4.6: *Links* die Anzahl der adaptiven Relaxationen am feinsten Gitter des parabolischen Problems ohne Vorglätter und einem gewählten Θ_{max} von 0.0005. *Rechts* das gleiche Bild für das elliptische Problem mit $\Theta_{max} = 0.00008$. Die hellste Stelle ist jeweils das Maximum mit zehn Relaxationen je Punkt in der Nähe der Singularität.

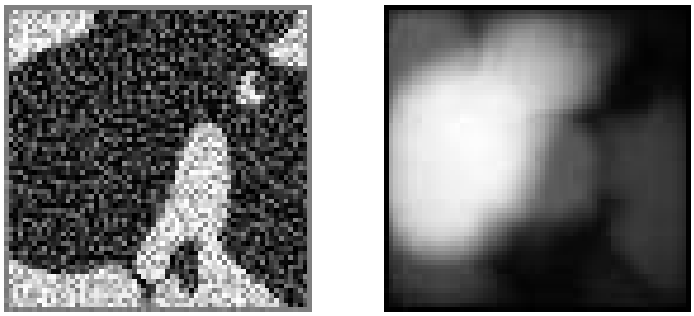


Abbildung 4.7: *Links* das Residuum des elliptischen Problems nach einem V–Zyklus mit adaptivem Nachglätter und $\Theta_{max} = 5 \cdot 10^{-6}$ (der graue Rand hat den Wert 0). *Rechts* die Anzahl der Relaxationen je Punkt auf dem feinsten Gitter (Weiß $\hat{=}$ 318).

Im elliptischen Fall funktioniert die adaptive Nachglättung bis zur Genauigkeit von $\Theta_{max} = 10^{-6}$ ebenfalls effizienter als der herkömmliche Glätter. Die Konvergenzrate ist anfangs extrem hoch, sinkt jedoch sehr stark ab und kann auch nicht durch mehrere V–Zyklen mitsamt herkömmlicher Vorglättung verbessert werden. Ab einem gewissen Punkt konvergiert das Verfahren kaum mehr. Einen Hinweis liefert Abbildung 4.6. Im parabolischen Fall werden viele Punkte gar nicht relaxiert. Im elliptischen Fall werden fast alle Punkte mindestens einmal relaxiert. Bei Semi–Verfeinerung ist dies zwar nicht der Fall, der Effekt bleibt jedoch. Abbildung 4.7 zeigt den nicht mehr effizienten Fall des elliptischen Problems mit $\Theta_{max} < 10^{-5}$. Die Anzahl der Relaxationen je Punkt ist nicht mehr an der Singularität sondern in einem großen Bereich darum verteilt. Dieses Ausstrahlen der Singularität auf das gesamte Gebiet Ω könnte das Problem sein. Ein Test mit einem nicht singulären Problem bestätigte aber ebenfalls das schlechte Verhalten.

4.3.2 Die Transportoperatoren

Die vier Gruppen von Transportoperatoren I_h^{2h}, I_{2h}^h, $\tilde{I}_h^{2h}$ und $\tilde{I}_{2h}^h$ aus Abbildung 2.6 müssen neu definiert werden. Drei davon werden im FAS–Algorithmus wie in Abbildung 2.9 zusammenfassend für folgende Schritte benötigt.

- I_h^{2h} für die Restriktion der rechten Seite f_h,
- I_{2h}^h für die Prolongation der Korrektur $u_{2h} - \bar{u}_{2h}$ und für die Initialisierung eines feinen Gitters u_h sowie
- $\tilde{I}_h^{2h}$ für die Initialisierung der groben linken Seite $\bar{u}_{2h}$.

Die vierte Gruppe von Transportoperatoren $\tilde{I}_{2h}^h$ wird nie verwendet, da die Initialisierung von f_h immer durch eine explizite Berechnung erfolgt.

Erste Bedingung. Wie in Abschnitt 2.2.1 gezeigt, kann der diskrete Grobgitteroperator A_{2h} entweder durch die Galerkin–Approximation oder durch eine explizite Diskretisierung des kontinuierlichen Operators A erzeugt werden. Gleichung (2.10) gibt an, wie beide Bedingungen durch die Wahl der Operatoren zueinander *kompatibel* gemacht werden können, sodaß alle A_h immer eine wie in Abschnitt 4.2 geforderte gleichartige Struktur aufweisen.

Zweite Bedingung. In Abschnitt 3.1.1 wurde der Zusammenhang zwischen der *Prolongation* und der *Basisfunktion* der Diskretisierung aufgezeigt. Die Prolongation ergibt sich aus der gewählten Basis. Abbildung 4.8 zeigt die auf der Hutfunktion erster Ordnung in x und y basierende Basisfunktion eines Punktes. Für das parabolische Problem wurde wie in [HV95] im *Raum* eine Hutfunktion K erster Ordnung gewählt, in der *Zeit* jedoch die stückweise konstante Funktion $\hat{K}$ mit einem Träger in der Zukunft:

$$\hat{K}_i(x) = \begin{cases} 1 & x \in [x_i, x_{i+1}) \\ 0 & \text{sonst.} \end{cases} \tag{4.7}$$

Mit den beiden Bedingungen können die in [HV95] vorgeschlagenen folgenden Transportoperatoren verwendet werden.

Die Prolongation I_{2h}^h kann jeweils für die Verfeinerung im Raum und in der Zeit getrennt angegeben werden. Die Prolongation wird in der Stempel–Notation für *Raum*, *Zeit* und *Raum*$\times$*Zeit* definiert durch

$$\frac{1}{2}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{und} \quad \frac{1}{2}\begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

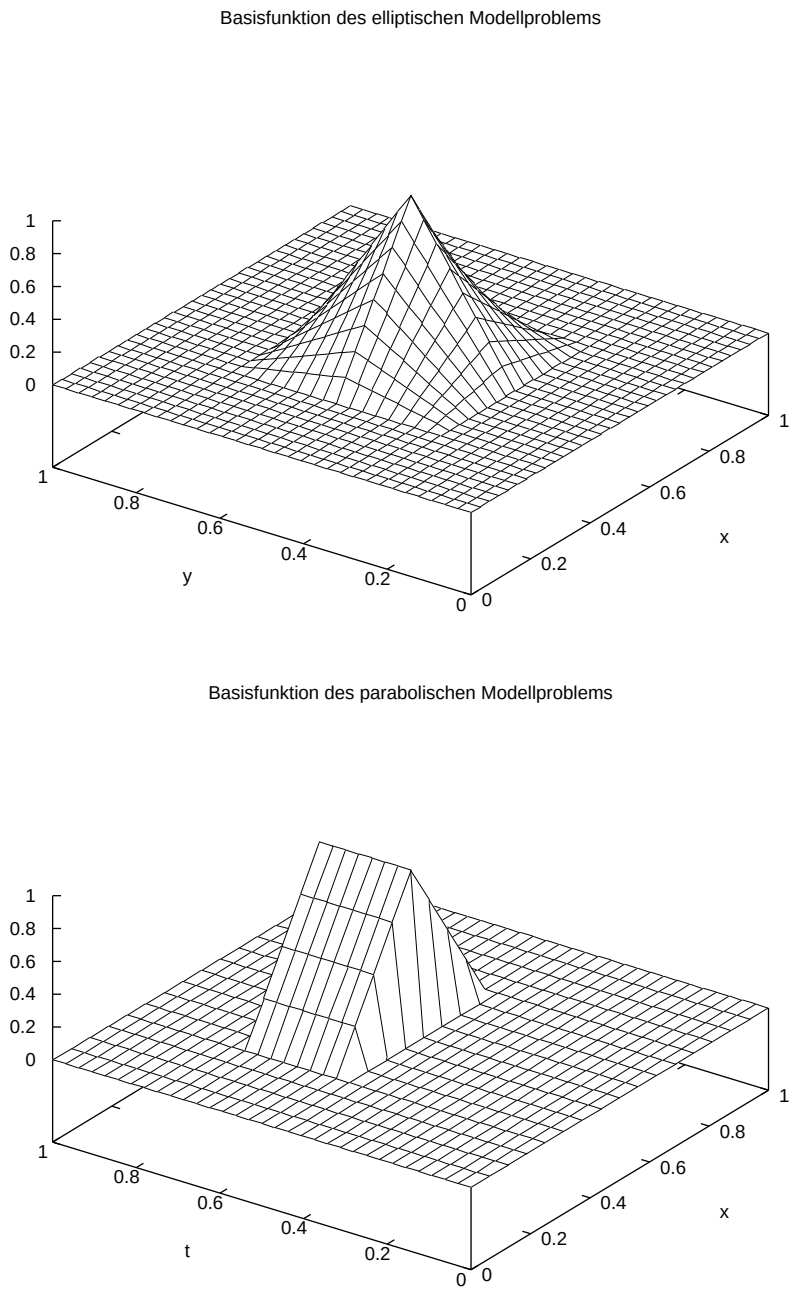


Abbildung 4.8: Basisfunktionen des Punktes $(0.5, 0.5)$ für das elliptische Raum– und das parabolische Raum×Zeit–Modellproblem mit $h_x = h_y = 1/4$ beziehungsweise $h_t = 1/4$.

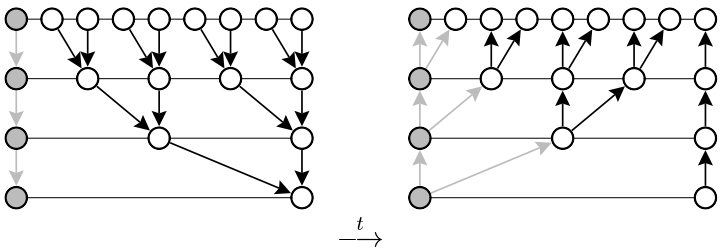


Abbildung 4.9: *Links* die Restriktion, *rechts* die Prolongation der Zeit über mehrere Gitterebenen eines endlichen Zeitabschnittes. Die unveränderlichen Randwerte sind grau hinterlegt.

Für die reine Raum–Verfeinerung ist der Operator identisch zur herkömmlichen Prolongation. In der Zeit ist die Prolongation asymmetrisch und trägt die Korrektur nur in die Zukunft. Abbildung 4.9 zeigt das zeitliche Verhalten der Prolongation als auch der Restriktion.

Es ist leicht erkennbar, daß die Basisfunktionen mit der Prolongation konstistent sind. Die Summe der feinen Basisfunktionen

$$\sum_j K_{h,j} \cdot I_{2h}^h[j,i] = K_{2h,i}$$

ist in jedem Fall gleich der groben Basisfunktion $K_{2h,i}$ des Punktes i (siehe (3.4) in Abschnitt 3.1.1).

Die Prolongation der rechten Seite, $\tilde{I}_{2h}^h$, wird im verwendeten FAMe–Algorithmus nicht verwendet.

Die Restriktion der rechten Seite, I_h^{2h}, kann als Gegenstück zur Prolongation ebenfalls getrennt für die unterschiedlichen Verfeinerungsvarianten in der Stern–Notation definiert werden als

$$\frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \frac{1}{2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{und} \quad \frac{1}{8}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

Für die Restriktion der linken Seite, $\tilde{I}_h^{2h}$, wird wie im elliptischen Modellproblem ebenfalls die direkte Injektion

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

verwendet.

Adaptive Verfeinerung. Für die adaptive Verfeinerung, wie in Abschnitt 3.1.3 beschrieben, benötigt man die von einem groben Punkt in beiderlei Dimensionen abhängigen Feingitterpunkte. Diese Punkte bilden die Menge von feinen Punkten, die bei einer Verfeinerung als real betrachtet werden. Abbildung 4.9 zeigt, welche feinen Punkte durch die *Restriktion* auf einen groben einwirken und welche feinen Punkte durch die *Prolongation* von einem groben Punkt modifiziert werden können.

Die abhängigen Punkte aus den Gleichungen (3.9) und (3.10) sind bei Semi–Verfeinerung in der Zeit jeweils der Punkt vor, zu und nach dem zu verfeinernden Zeitpunkt. In Kombination mit der Verfeinerung im Raum bekommt man bei der Standard–Verfeinerung wiederum ein feines 3×3 Gitter zu einem groben Punkt. Die Menge der kooperierenden Punkte (3.11) ist also gleich der des elliptischen Raum–Problems. Die *Bewertung* des Verfeinerungskriteriums (3.12) erfolgt nun jedoch nicht mehr aufgrund aller kooperierender Punkte. Nur mehr die durch die Restriktion in (3.9) definierten kooperierenden Punkte spielen hierbei eine Rolle.

4.3.3 Konvergenz

Die Konvergenz des Mehrgitterverfahrens ist wie in [HV95] genauer betrachtet sehr stark von $\lambda = \Delta t/\Delta x^2$ abhängig (siehe auch Abschnitt 4.2). λ kann als der *Grad der Anisotropie* gesehen werden. Zusammenfassend läßt sich feststellen, daß die verscheidenen λ der Gitterhierarchie eines Multigrid–Verfahrens dessen Konvergenz bestimmen. Ist $\lambda < 2^{-4}$ oder $\lambda > 2^2$, so divergiert das System oder es konvergiert nur extrem langsam. Auch die bestmögliche Konvergenzrate für ein $\lambda \approx 1$ ist bei Verwendung von Standard–Glättern nicht besonders gut. Das naive Verfahren der Standard–Verfeinerung in Raum$\times$Zeit kann so also nicht funktionieren, da λ sich in jedem Verfeinerungsschritt verändert.

Um über die gesamte Gitterhierarchie eine bestmögliche Konvergenz zu gewährleisten muß das λ jedes Gitters im gut konvergierenden Bereich zwischen 2^0 und 2^2 liegen (λ–Kriterium). Da die Gitterhierarchie im FMG–Algorithmus vom gröbsten Gitter weg beliebig aufgebaut werden kann, wird in der Implementierung sichergestellt, daß alle erzeugten feineren Gitter dem λ–Kriterium genügen. Dadurch ergibt sich dann eine Verfeinerung der Diskretisierung wie in Abbildung 4.10.

Nachteilig ist, daß die Zeit–Dimension dabei doppelt so oft verfeinert wird wie die Raum–Dimension. Für das Modellproblem wurde für eine verbesserte Veranschaulichung die Zeit mit 1/8 skaliert. Dadurch ergibt sich ein $\Delta t = h_t/8$ und somit ein anderes λ. Abbildung 4.11 zeigt die Verfei-

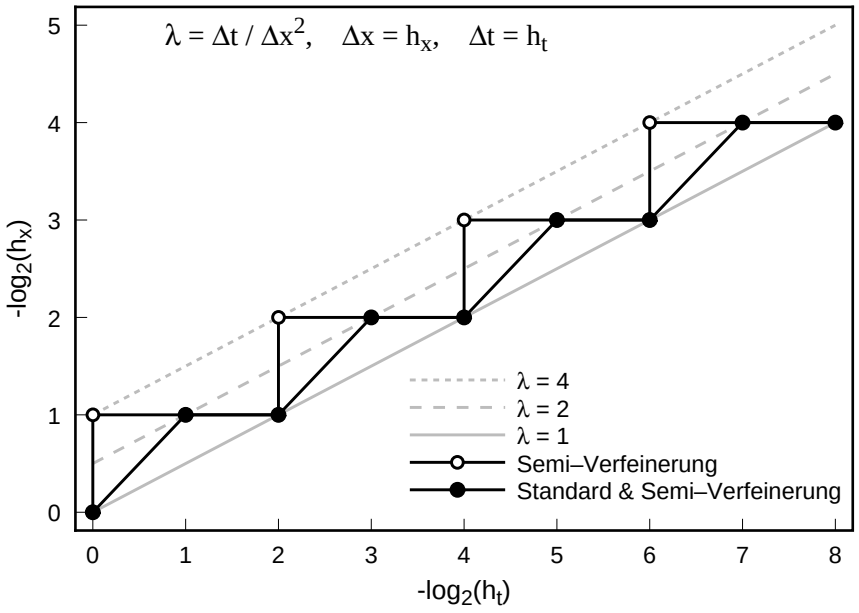


Abbildung 4.10: Die aus dem gewünschten λ resultierenden Verfeinerungsschritte bei einer reinen Semi–Verfeinerung oder bei gemischter Standard– und Semi–Verfeinerung. Das gröbste Gitter ist immer die 1×1 Diskretisierung.

nerungsstrategie für das Zeit–Skalierte Modellproblem. In der „Standard & Semi“ Variante wird ausschließlich die $x \times t$ Standard–Verfeinerung und die t Semi–Verfeinerung verwendet. Eine zusätzliches x Semi–Verfeinerung würde die Gitterhierarchie früher in den Bereich zwischen $\lambda = 2^0$ und $\lambda = 2^2$ bringen.

Gründe. Das schlechte Konvergenzverhalten kann anhand Gleichung (4.4) erläutert werden. Bei kleinen und großen λ entstehen stark anisotrope Gleichungssysteme, die mit Relaxation nicht gut gelöst werden können. Gewisse hochfrequente Anteile werden nicht geglättet, da punktweise Relaxation nur in Richtung der stärksten Kopplung gut glättet (aus [HV95]). Ein Lösungsansatz ist die Wahl eines angemessenen Glätters. Es ist zu bemerken, daß der parabolische Charakter des Problems der Grund für die schlechte Konvergenz ist.

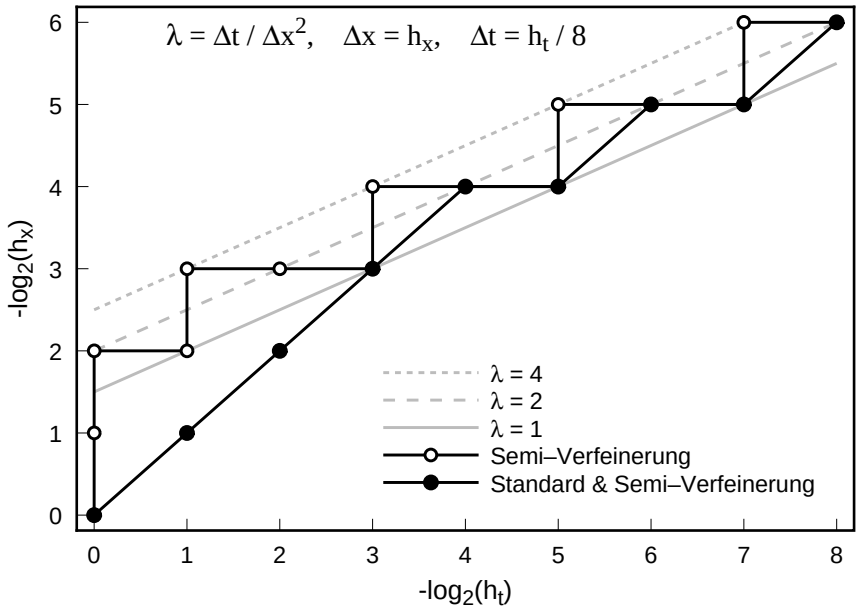


Abbildung 4.11: Die Gitterbildung analog zu Abbildung 4.10, jedoch mit einer Zeitskala $\Delta t = h_t/8$.

4.3.4 Adaptive Verfeinerung

Mit den oben gewählten Transportoperatoren und der Bestimmung der Kooperierenden Menge funktioniert die adaptive Verfeinerung gleich wie für das Raum–Problem beschrieben. Es gibt keine Einschränkungen. Für höherdimensionale Probleme wie einem Raum^3×Zeit–Problem wird vermutet, daß alle Algorithmen analog funktionieren. Abbildung 4.12 zeigt die adaptive Verfeinerung des parabolischen Modellproblems bei reiner Semi–Verfeinerung und bei Standard & Semi–Verfeinerung. Meist ist die reine Semi–Verfeinerung präziser in der Schätzung, das heißt, daß weniger reale Punkte angelegt werden. Dagegen gibt es mehr Gitter in der Hierarchie und somit auch mehr Inter–Gitter–Transport. λ ist bei reiner Semi–Verfeinerung weniger gut kontrollierbar, es schwankt jedoch nur unwesentlich mehr.

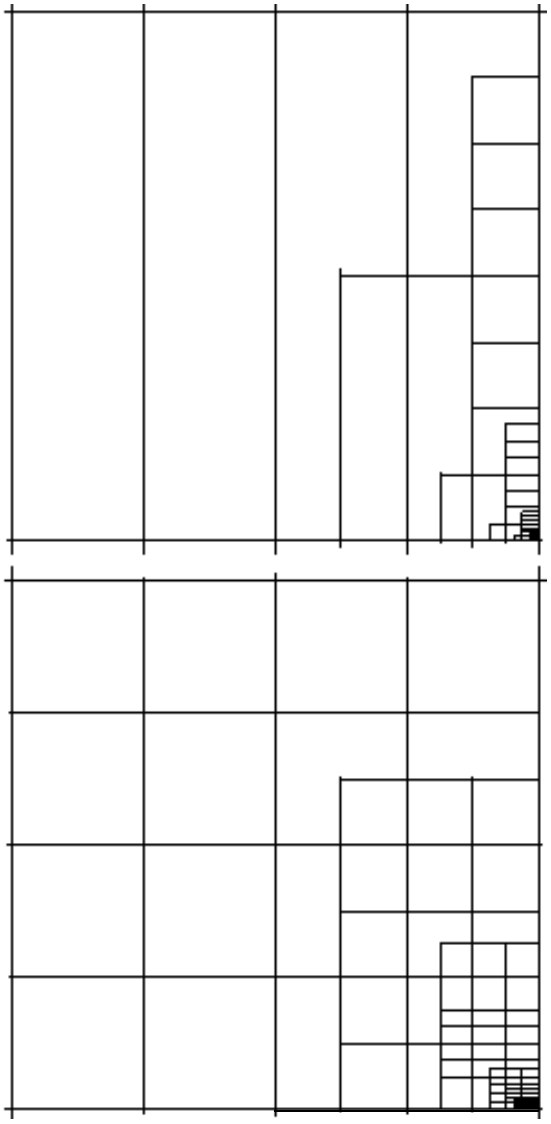


Abbildung 4.12: Adaptiv verfeinertes Gitter des Zeit–Problems. *Oben* bei reiner Semi–Verfeinerung, *unten* bei alternierender Standard & Semi–Verfeinerung.

Kapitel 5

Datenstrukturen und Implementierung

Dieses Kapitel gliedert die Implementierung in einzelne Komponenten und beschreibt die zugrundeliegenden Datenstrukturen, aber auch konkret die Programmierung. Einzelne C++–Klassen werden im Anhang A mit den dazugehörenden Dateien kurz beschrieben.

Es gibt drei große unabhängige Teile. Erstens die Speicherung von Punkten und die dazugehörenden Datenstrukturen (5.1), zweitens die Punkte, welche die Mehrgitter–Operatoren und das Problem an sich präsentieren (5.2), und drittens das eigentliche adaptive Mehrgitterverfahren (5.4) mit der Implementierung der adaptiven Relaxation (5.3).

5.1 Speicherung der dünnbesetzten Gitter

Die adaptiv verfeinerten Gitter sind üblicherweise nur dünn besetzt. Das heißt, daß die Anzahl der Punkte n auf einem Gitter

$$n_h \ll (1 + 1/h_x) * (1 + 1/h_y) \qquad \text{ist.}$$

In der Regel wird n_h ab einem gewissen h nicht mehr wachsen, wenn die Singularität fraktalen Charakter hat. Im Beispiel in Abbildung 3.1 (Seite 22) ist bei jeder Diskretisierung immer eine gleichförmige Menge von Knoten um die Singularität vorhanden, die laut Fehlerschätzer noch zu verfeinern ist. Deren Anzahl ist ab einem gewissen h in etwa konstant. Somit wächst n_h ab einer gewissen Diskretisierung nicht mehr, die Anzahl möglicher Punkte jedoch quadratisch.

Es gibt in der Literatur eine große Anzahl von beschriebenen Datenstrukturen zur Darstellung dünnbesetzter Gitter. Viele davon beruhen auf *Baum–Strukturen*. Andere Ansätze wie [GZ97] basieren auf *Hashtabellen*.

5.1.1 Zugriff über Bäume

In der Informatik sind Bäume eine sehr effiziente und einfache Art, Daten zu speichern. Ein Datenzugriff beginnt an der Wurzel und geht Schritt für Schritt in Richtung Blatt. Bei jedem Schritt muß mit einer *Verzweigungsfunktion* entschieden werden, in welchen der Äste die Suche verzweigt. Im einfachsten Fall entscheidet ein einzelnes Bit die Wahl des Pfades in einem binären Baum, so daß bei einer Null in den linken und bei einer Eins in den rechten Ast verzweigt wird.

Diese Verzweigungsfunktion kann aber auch jede andere Zugriffsmethode sein. In [Löt96] wird die Auswahl des Astes über ein quadratisches Array getroffen. Der benötigte Index errechnet sich auf jeder Ebene aus der Koordinate des gesuchten Punktes. Ein Zugriff auf Punkt $P_{x,y}$ mit achtbittigen Indizes $x_{0\ldots7}$ und $y_{0\ldots7}$ über zwei 4×4 Ebenen kann folgendermaßen aussehen:

$$tree \overset{i(x_{0\ldots3},y_{0\ldots3})}{\longrightarrow} subtree \overset{i(x_{4\ldots7},y_{4\ldots7})}{\longrightarrow} P_{x,y}$$

Daraus kann man folgern, daß Bäume eine Kette von Zugriffsfunktionen haben, bei der jede Zugriffsfunktion einen Teil der Positionsinformation verwendet. Die mittlere *Zugriffsgeschwindigkeit* auf Objekte eines Baumes wird durch die mittlere Anzahl der Verzweigungsfunktionen und Zeit zur Durchführung einer Verzweigung definiert, die in der Regel eine Größe der Ordnung $\mathcal{O}(1)$ hat. Dadurch errechnet sich bei einer *mittleren Pfadlänge* von p die Zugriffskomplexität $\mathcal{O}(\lceil p \rceil)$.

5.1.2 Hashtabellen im allgemeinen

Hashtabellen sind eine besondere Art von endlichen Tabellen, beschrieben in Informatik–Werken wie [Knu73, Kapitel 6.4] oder ausführlicher in [OW96, Kapitel 4], woraus folgende Definitionen großteils stammen.

Jedem Datensatz oder Objekt ist ein eindeutiger *Schlüssel* k zugeordnet. Die Menge aller möglichen Schlüssel ist $\mathcal{K}$, von der jedoch meist nur eine Teilmenge K aktiv ist. Die Datensätze werden in einem linearen Feld der Größe m mit Indizes $0, \ldots, m-1$ gespeichert, der *Hashtabelle*. Durch die *Hashfunktion* $h : \mathcal{K} \to \{0, \ldots, m-1\}$ kann jedem Schlüssel k ein Index $h(k)$, die *Hashadresse* oder der *Hashindex*, zugewiesen werden. Mit diesem Index kann ein Objekt in der Hashtabelle plaziert oder referenziert werden. Die verwendeten Schlüssel K sind in der Regel nur eine sehr kleine Teilmenge von $\mathcal{K}$, da wie in der Einleitung von Abschnitt 5.1 erwähnt nur wenige Punkte K aus der Menge aller Punkte $\mathcal{K}$ verwendet werden. Es gilt im allgemeinen

$$|K| < m \ll |\mathcal{K}|.$$

Schlüssel, deren Hashadressen identisch sind ($h(k) = h(k')$) heißen *Synonyme*. Befinden sich zwei derartige Schlüssel in der aktiven Menge K, so ergibt

sich eine *Adreßkollision*. Bei diesen Kollisionen muß eine Sonderbehandlung erfolgen. Ein Hashverfahren muß daher zwei Bedingungen erfüllen. Erstens muß die Hashfunktion derart gewählt werden, daß nur wenige Kollisionen auftreten, und zweitens müssen diese möglichst effizient behandelt werden.

Die Hashfunktion ist eine surjektive Abbildung vom Objektschlüssel k zu dessen Hashadresse $h(k)$. Sie erfolgt üblicherweise nicht direkt vom Schlüsselraum $\mathcal{K}$ zu einem Hashindex j sondern über einen mit der Funktion h' generierten Index aus den natürlichen Zahlen, $n \in \mathbb{N}$. Aus dieser Zahl n wird dann mittels einer Hashfunktion h'' die endgültige Adresse errechnet. Die Index–Funktion h' muß die Eigenschaften einer Hashfunktion haben, womit $h(k) = h''(h'(k))$ ist, also:

$$k \in \mathcal{K} \quad \xrightarrow{h'} \quad i \in \mathbb{N} \quad \xrightarrow{h''} \quad j \in \{0, \ldots, m-1\}.$$

Diese Vorgehensweise ist von Vorteil, da der Anwendungsprogrammierer aus einem Schlüssel k nur mehr eine natürliche Zahl i, den *Objektindex* mit Hilfe von h' errechnen muß (primäres Hashing). In der Implementierung müssen Eigenschaften der Hashtabelle wie deren Größe in den zu speichernden Objekten nicht beachtet werden, da ja der Objektindex i von der Hashtabelle benutzt wird, um die endgültige Hashadresse j zu finden. Das Hashtabellen–Objekt kümmert sich nur mehr um das sekundäre Hashing h'' in Abhängigkeit zur Tabellengröße (sekundäres Hashing).

Die Hashfunktion h'' kann nun unabhängig vom Objekt gewählt werden. Bekannt sind viele Verfahren wie die *Divisions–Rest–Methode*, die *multiplikative Methode* und andere. In der Implementierung bietet die Divisions–Rest–Methode bei einer Wahl von m als Primzahl ein sehr gutes Verhalten als Hashfunktion und auch bezüglich der Geschwindigkeit.

$$h''(i) = i \bmod m, \qquad m \text{ ist prim.}$$

Wenn die Größe m der Hashtabelle sehr viel kleiner als der mögliche Wertebereich von i und außerdem m prim ist, kann man von einer Gleichverteilung der zugewiesenen Indizes j in der Hashtabelle ausgehen. In der Praxis werden i und m als 32– oder 64–Bit Maschinenwörter dargestellt. Dadurch ist die Anzahl aller durch h möglichen Synonyme zu einer Hashadresse nicht gleich. Bei 32 Bit gibt es für $2^{32} \bmod m$ Hashadressen $\lceil 2^{32}/m \rceil$ Synonyme, für die restlichen Adressen nur $\lfloor 2^{32}/m \rfloor$, eines weniger. Diese Tatsache erzeugt jedoch keine ernsthaften Randeffekte.

Die sekundäre Hashfunktion h'' kann mit der Divisions–Rest–Methode zufriedenstellend als Bibliotheksfunktion implementiert werden. Die primäre Hashfunktion h' muß ebenfalls möglichst gleichverteilend und kollisionsarm arbeiten und für jede Klasse von Objekten neu implementiert werden.

Kollisionsbehandlung. Kollisionen können auftreten, wenn für zwei Schlüssel k und k' deren Hashadressen $h(k)$ und $h(k')$ gleich sind. Diese Kollisionen treten bei gleichverteiltem j mit der Wahrscheinlichkeit n/m, dem *Belegungsfaktor* α, auf, wobei n die Anzahl der Elemente $|K|$ in der Hashtabelle ist. Eine guter Kompromiß zwischen Speicherverbrauch und Geschwindigkeit ist $n/m < 0.5$ (siehe auch [Knu73, OW96]).

Tritt eine Kollision auf, muß sie aufgelöst oder behandelt werden. Es gibt Auflösungsstrategien die nur einen Eintrag je Tabellenplatz benötigen wie das *mehrfache Hashing*. Alternativ dazu können alle Synonyme gemeinsam als *Überlauf* an einer Stelle der Hashtabelle gespeichert werden. Dafür können wiederum verschiedene Techniken verwendet werden. Da in einem Tabellenplatz bei vorsichtiger Handhabe meist weniger als zwei Objekte gespeichert werden ($n/m < 0.5$), kann auch eine sehr kleine Liste mit linearem Zugriff (Array) oder eine verkettete Liste zu diesem Zweck verwendet werden.

Der Objektindex $i \in \mathbb{N}$ wird bestimmt durch die Abbildung h' aus dem Schlüsselraum $\mathcal{K}$ in die natürlichen Zahlen. Die Funktion, die den Objektindex erzeugt, kann auch surjektiv sein und wird in [GZ97] als „binary hash key“ bezeichnet. Im Idealfall beschreibt er aber die Aufzählung aller möglichen Objekte aus dem Objektraum. Dadurch ist die Gleichverteilung sichergestellt, da bei zufälliger Objektwahl jeder mögliche Index i die gleiche Wahrscheinlichkeit $|\mathcal{K}|/|K|$ hat. Den Index i eines Objekts zu generieren stellt oft ein Problem dar. Probleme ergeben sich zum Beispiel, wenn mehrdimensional beschriebene Objekten mit der endlichen Menge darstellbarer Indize (zum Beispiel 2^{32}) aufgezählt werden müssen.

Die Zugriffsgeschwindigkeit. Der Aufwand für einen Zugriff auf einen Hashtabelleneintrag ist bestimmt durch $h''(h'(k))$ und sollte somit $\mathcal{O}(1) * \mathcal{O}(1)$ sein. Der Aufwand für die Suche eines Objektes in einem Tabelleneintrag hängt von der verwendeten Kollisionsbehandlung ab und ist für die lineare Suche in einem Array gleich $\mathcal{O}(\lceil (n/m)/2 \rceil)$. Dadurch ergibt sich eine mittlere Zugriffs–Komplexität von

$$\mathcal{O}(\lceil (n/m)/2 \rceil).$$

Die effektive Zugriffsgeschwindigkeit hängt also nur mehr von der konstanten Zeit, die die Indexberechnung benötigt und vom Füllgrad der Hashtabelle ab.

Hashtabellen sind komplexer zu implementieren, dafür jedoch flexibler und im Zugriff oft schneller als Baumstrukturen oder ähnliche Datenstrukturen. In der Implementierung werden deshalb dünnbesetzte virtuelle Gitter von Punkten in Hashtabellen gespeichert.

5.1.3 Objektindex h' von Koordinaten

Eines der ersten Probleme bei der Speicherung von Punkten in Hashtabellen ist die *Erzeugung eines Objektindex*. Ein Objekt ist in diesem Fall ein Punkt, dessen Schlüssel eine mehrdimensionale Koordinate ist. Die Objektindizierung muß also als primäres Hashverfahren h' jeder Koordinate eine natürliche Zahl zuweisen können.

Zahlenmodell. Computer können nicht mit reellen $\mathbb{R}$ sondern nur mit natürlichen $\mathbb{N}$ und rationalen $\mathbb{Q}$ Zahlen arbeiten. Daraus folgt, daß alle darstellbaren Werte eines Computers abzählbar sind. Das gilt auch uneingeschränkt für mehrdimensionale Daten. Eine *Koordinate* besteht im Rechner aus je einem Wert für jede Dimension. Der Datentyp dieser Werte ist in der C++–Implementierung die Klasse `Dimension`. Diese Klasse ist eine Fixkommazahl und muß mindestens den Wertebereich $[0, 1]$ definieren, da für die Modellprobleme $\Omega = [0, 1] \times [0, 1]$ ist. Der Wert wird durch eine ganze Zahl *val* beschrieben:

$$x \leftarrow val * 2^{-bits+2}$$

Wobei *bits* die Anzahl der verfügbaren Bits von *val* ist. Ist die interne Repräsentation von *val* ein 32–Bit Wort, so können hiermit Werte bis zu 2^{-30} dargestellt werden. Der von $val * 2^{-bits}$ definierte Bereich ist $[0, 1)$. Da auch die Eins dargestellt werden muß, wird der Wertebereich auf Kosten einer Nachkommastelle auf $[0, 2)$ erhöht. Bei Iterationen wie in der folgenden Schleife wird auch $x = 2$ benötigt. Da x bei $x = 2 \equiv_2 0$ auf die 0 überläuft, wird der Darstellungsbereich auf $[0, 4)$ ausgedehnt. Deshalb ist der Exponent um zwei Bits auf $-bits + 2$ verschoben.

```
Dimension delta = 1.0;
for(Dimension x=0.0; x<=1.0; x+=delta) {
    ...
}
```

Mit den Zahlen der Klasse `Dimension` läßt sich wie mit den herkömmlichen Fließkommaklassen `float` und `double` arbeiten. Die Verarbeitungsgeschwindigkeit der hautsächlich verwendeten Operationen ist höher als bei Fließkommaarithmetik. Zudem ist der Speicherverbrauch in der Regel ein Maschinenwort.

Koordinaten. Die Position eines Punktes wird als mehrdimensionaler Wert in der Klasse `Coordinate` gespeichert. Eine Koordinate enthällt für jede Dimension einen Wert vom Typ `Dimension`. Die wichtigste Operation auf den enthaltenen Werten ist die Erzeugung einer Hashadresse oder besser gesagt eines Objektindex für die Verwendung in der Hashtabelle.

Erzeugung des Objektindex. Ein Objektindex ist vom Typ `Hashvalue`, der ein vorzeichenloses Maschinenwort darstellt. Ein eindimensionaler Wert x_0 kann in einem Wort gespeichert werden und könnte eins zu eins als Objektindex dienen. Mehrdimensionale Werte $x_i, 0 \leq i \leq d$ können allein aus Platzmangel nicht injektiv auf ein Maschinenwort abgebildet werden.

Zusätzlich erschwerend ist, daß die Koordinaten nicht zufällig sondern in geometrischen Strukturen auftreten. Bei Mehrgitterverfahren sind die Punkte in gleichmäßigen Gittern positioniert. Dadurch gibt es, wenn man aus zwei 32–Bit Werten mit `XOR`–Operation einen Wert erzeugt, keine Gleichverteilung und in der Konsequenz sehr viele Kollisionen in der Hashtabelle.

Für die Erzeugung des Objektindex $\text{hash}(x, y)$ in Form eines 32–Bit `Hashvalue` **aus zwei 16–Bit Werten** (je eine `Dimension`) wird eine Verschachtelung der Bits $x_{0\ldots15}$ und $y_{0\ldots15}$ vorgeschlagen:

$$h'(x_{0\ldots15}, y_{0\ldots15}) = x_{15}y_{15}x_{14}y_{14}\ldots x_0y_0$$

Dadurch ist der Objektindex eine injektive Abbildung aller möglichen Koordinaten in einen Objektindex. Das führt zu dem Effekt, daß durch die Modulo–Operation kollidierende Objekte der Hashtabelle keinen geometrischen Bezug zueinander besitzen. Konkret senkt dies die Kollisionsrate von gleichmäßigen Punktanordnungen.

Der *Zugriff auf alle Objekte* in einem Hash, wie er bei der Relaxation aller Punkte eines Gitters stattfindet, geschieht über eine direkte Auflistung der Hashtabelle. Deren Reihenfolge ergibt dadurch auch die Reihenfolge der punktuellen Relaxation. Erzeugt der Zugriff eine *musterartige Reihung* der Punkte, so kann es zu Randeffekten wie punktuell sehr hohen Fehlern kommen. Um solche systematischen Fehler auszuschließen, muß die *Reihung* der Punkte *quasi zufällig* erfolgen. Ein negatives Beispiel in diesem Sinn stellt folgende injektive Aufzählung dar, da eine Aufzählung der Punkte in der Hashtabelle Streifenartige Muster erzeugt:

$$h'(x_{0\ldots15}, y_{0\ldots15}) = x + (y \ll 16) = y_{15}\ldots y_0x_{15}\ldots x_0$$

Empirisch wurde für die Erzeugung des Objektindex **aus zwei 32–Bit Werten** folgende sehr zufriedenstellende Funktion gefunden:

$$h'(x_{0\ldots31}, y_{0\ldots31}) = (29 * (x \gg 8)) \oplus (7 * (y \gg 8)) \oplus x \oplus (y \gg 16) \oplus (y \ll 16)$$

Andere Hashverfahren. In [GZ97] wurde der Ansatz, Hashtabellen zur Speicherung von Gittern zu benutzen, ebenfalls verfolgt. Dort wird auf eine implizite Aufteilung der Punktmenge auf mehrere Prozessoren durch den Objektindex (dort „binary hash key“) Wert gelegt. Um die Inter–Prozessor–Kommunikation gering zu halten, muß die gewählte injektive *Auffädelung* einen geometrischen Bezug haben. Dafür wird die *Hilbertsche raumfüllende*

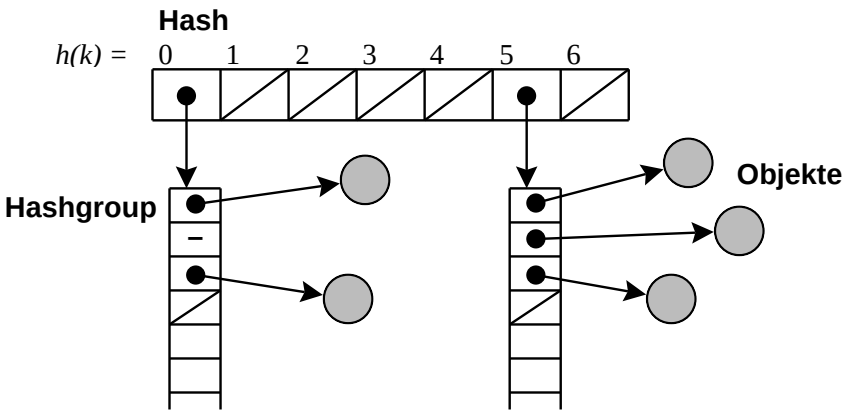


Abbildung 5.1: Aufbau der Hashtabelle *Hash* und der Überlaufspeicher *Hashgroup*.

Kurve („Hilberts space filling curve“) in von h abhängigen Auflösungen verwendet. Dieses Gebilde ist eine fraktale Linie, die jeden Punkt einer Fläche beziehungsweise eines Raumes berührt. Abschnitte auf dieser Linie werden den Prozessoren zugewiesen. Die Position eines Punktes auf dieser Linie ist der Objektindex $h'(k)$. Für eine gewünschte *Parallelisierung* der Algorithmen bietet dieses Verfahren Vorteile.

5.1.4 Implementierung der Hashtabelle

Die Klasse `Hash` implementiert die sekundäre Hashfunktion h'' und die Speicherung der einzelnen Objekte. In der objektorientierten Programmierung wird fast ausschließlich mit Referenzen von Objekten gearbeitet, deshalb werden keine Kopien der Objekte im Hash gespeichert sondern nur Referenzen auf diese. Die Hashtabelle kennt den Typ eines Objektes nicht, da bei Zugriffen jedoch der Objektschlüssel k verglichen werden muß, gibt es eine virtuelle Methode `equals`, der zwei Objektreferenzen zum Vergleich übergeben werden. Diese Methode muß den Objekttyp kennen und den Vergleich durchführen.

Die Hashfunktion h'' ist die beschriebene Divisions–Rest–Methode und zur Kollisionsbehandlung wird ein Array für alle Synonyme, eine *Hashgroup* verwendet. Die Klasse `Hashgroup` bietet ein dynamisch wachsendes Array mit der empirisch ermittelten Anfangsgröße von sechs Elementen. Die Suche in der Hashgroup ist linear und läuft bis zum markierten Ende des Arrays. In einer Hashgroup müssen Kollisionen in der primären Hashfunktion h' abgefangen werden, da ein gleicher Objektindex i noch keine Gleichheit der Schlüssel bedeutet. Deshalb wird bei einem Zugriff der Schlüssel k eines Objektes mittels der virtuellen Methode `Hash::equals` verglichen.

Die Abbildung 5.1 zeigt ein Beispiel, in dem eine Hashtabelle der Größe

$m = 7$ fünf Objekte speichert. Bei einem Zugriff wird der Objektindex i übergeben und eine Hashgroup gewählt. In dieser Gruppe wird anschließend der Schlüssel eines jeden Objekts mit dem gesuchten verglichen.

5.1.5 Implementierung des Gitters

Die Gitterklasse `Grid` implementiert die abstrakte Basisklasse `Hash`. Die Schlüsselmenge $\mathcal{K}$ der Objekte sind die Koordinaten `Coordinate`

$$\mathcal{K} = \{(x, y) \mid (x, y) \in [0, 1] \times [0, 1]\}$$

Die aktive Menge K ist bei vollbesetzen Gittern die Menge der Koordinaten

$$K = \{(x, y) \mid (x, y) \in [0, 1] \times [0, 1],\ x = i \cdot h_x,\ y = j \cdot h_y,\ i, j \in \mathbb{N}\}$$

oder bei einem teilweise besetzen Gitter eine Teilmenge davon.

Die Operation `put(Point)` ermittelt den Schlüssel des übergebenen Punktes, dessen Koordinate k, und errechnet daraus den Objektindex $h'(k)$ vom Typ `Hashvalue`. Diese Berechnung wird durch automatische C++–Typumwandlungen implizit durchgeführt. Da ein Punkt die Koordinate zur Basisklasse hat und der *Hashvalue–Cast* automatisch von der Koordinate durchgeführt wird, ruft der Compiler für einen Punkt automatisch als Cast die Hashvalue–Berechnung auf. In den Funktionen der Klasse `Hash` kann als Parameter für einen Hashvalue somit immer ein Punkt oder eine Koordinate übergeben werden.

Operationen: Die Gitterklasse `Grid` bietet folgende Operationen:

- Mehrdimensionale Zugriffe
- Linearer Zugriff
- Die Verwaltung der Gitterweite über die `getH*()` Operationen
- Hinzufügen und Entfernen von Punkten
- Die Verwaltung und Erzeugung der Gitterhierarchie

Mehrdimensionaler Zugriff: Die Klammer–Operatoren `(Coordinate)` und `(Dimension x, Dimension y)` sind überladen und liefern einen vorhandenen Punkt an der gesuchten Stelle. Da es sich um ein *virtuelles globales Gitter* handelt, wird bei einem negativen Zugriff automatisch ein Geisterpunkt an der gewünschten Stelle erzeugt. Dieses Verhalten läßt sich durch einen optionalen zusätzlichen Parameter steuern. Die Deklarationen der Zugriffopeatoren

```
Point *operator ()(Dimension x, Dimension y,
                   PointKind createWhat =GHOST);
Point *operator ()(Coordinate& c,
                   PointKind createWhat =GHOST);
```

sehen als Standardverhalten vor, daß ein Geist erzeugt wird. Sollte kein Punkt erzeugt werden, kann der Parameter `NOTHING` mit angegeben werden. Im normalen Fall gilt aber die Annahme, daß *immer* ein Punkt zurückgeliefert wird.

Linearer Zugriff: Der eckige Klammer–Operator `[unsigned int]` wird zum linearen Zugriff auf alle im Gitter vorhandenen Punkte angeboten. Zusammen mit der Funktion `getCount()` kann eine einfache Schleife *effizient* alle Punkte durcharbeiten:

```
Grid& g = ...;
for(unsigned int i=0; i < g.getCount(); i++) {
    Point *one = g[i];
    ...
}
```

Im Hintergrund arbeitet ein spezieller Mechanismus, der ein Array mit Punktreferenzen konsistent hält. Gibt es einen zu großen Zuwachs an Punkten, so kann sich die Reihenfolge der Punkte im Array ändern. Während der Abarbeitung einer Schleife über alle Indizes kann das zu störenden Effekten führen. Mit einer (auskommentierten) Annahme `assert(...)` in `Grid.h` kann eine strenge Überprüfung auf Veränderungen erzwungen werden.

Entfernen von Punkten: Die `remove(Coordinate)` Methode entfernt aus dem Gitter einen Punkt, ohne den Punkt selbst zu löschen.

Hinzufügen von Punkten: Die `put(Point)` Methode fügt einen neuen Punkt in die Gitterstruktur ein. Diese Methode wird jedoch fast ausschließlich intern verwendet, da alle Punkte automatisch von den Zugriffsoperatoren erzeugt werden können.

Erzeugen einer Gitterhierarche: Jedes Gitter kennt das nächst gröbere oder feinere Gitter in einer *Multigrid–Hierarchie*. Dadurch kann jeder Punkt über das ihm bekannte Gitter auf ein feineres oder gröberes Gitter zugreifen. Die Diskretisierung des nächst feineren Gitters wird vom aktuellen Gitter errechnet. Im Abschnitt 4.3.3 wird auf die besonderen Wahl einer gewünschten Diskretisierung eingegangen.

5.1.6 Resümee

Die Wahl eines virtuellen Gitters, das beliebigen Zugriff auf alle Koordinaten erlaubt und *bei Bedarf* Punkte erzeugt, ist für die Programmierung von unschätzbarem Vorteil. Als passende Art der Speicherung hat sich die Hashtabelle in Speicherverbrauch, Zugriffsgeschwindigkeit und Flexibilität bewährt. Sie bleibt für den Anwender jedoch komplett vom virtuellen Gitter `Grid` verborgen und kann dadurch jederzeit mit einem anderen Verfahren ausgetauscht werden.

Im virtuellen globalen Gitter benötigt ein realer Punkt deutlich mehr Speicher als in einem voll existierenden Gitter. Die eher auf Geschwindigkeit optimierte Implementierung benötigt für jeden Punkt zehn Pointer, vier Fließkommawerte und mindestens neun Bit für die Verwaltung von Zuständen. In einer 32–Bit Umgebung sind das mindestens 76 Bytes, in einer 64–Bit Umgebung mindestens 120 Bytes. Es wird also bis zu 15 mal mehr Speicher benötigt als bei einer Speicherung in Arrays. Ist ein virtuelles Gitter jedoch nur zu einem fünfzehntel gefüllt, benötigt es weniger Speicher. Besonders bei singulären Problemen kann man mit virtuellen globalen Gittern sehr feine Diskretisierungen erreichen. In der Tetstumgebung wurde auch mit $h = 2^{-20}$ bei wenigen Megabytes Speicher gearbeitet.

5.2 Punkte

Die Klasse `Point` beschreibt das zu lösende Problem und implementiert dafür folgende Funktionalitäten eines Punktes:

- Speichern einer diskreten Gleichung.
- Lösen der gespeicherten diskreten Gleichung.
- Definition der Randbedingungen.
- Zugriff auf benachbarte Punkte.
- Definition der Transportoperatoren.
- Verwaltung eines Geisterpunktes.

Diese funktionellen Gruppen beinhalten jeweils mehrere Methoden und Funktionen. Die folgenden Abschnitte gehen auf die einzelnen Punkte detailliert ein.

5.2.1 Speichern einer diskreten Gleichung

Die Kernaufgabe eines Punktes ist, eine Diskretisierung des zu lösenden Problems an einer Stelle zu behandeln. Die Gleichung eines Punktes i besteht aus

- der diskreten Lösung $u_h(i)$,
- der rechten Seite $f_h(i)$,
- einem Operator A_h und
- den benachbarten Punkten $\{j \mid a_{i,j} \neq 0\}$.

Da alle Punkte zu einem h in einem Gitter u_h gespeichert werden, wird h im Gitter gespeichert. Der Punkt hält nur eine Referenz auf das Gitter. Der Operator A_h wird im Programm abhängig von h explizit kodiert (mehr dazu im Abschnitt 5.2.2) und die benachbarten Punkte sind über das Gitter u_h verfügbar (siehe auch Abschnitt 5.2.4). Was bleibt, sind die Werte $u_h(i)$ und $f_h(i)$ sowie die Position des Punktes, um seine Nachbarn zu finden. Die Position wird wie im Abschnitt 5.1 beschrieben in der Vaterklasse `Coordinate` gespeichert.

In der Klassendefinition wird zusätzlich zu den Werten u und f eines Punktes der initiale Wert u_{init} von u gespeichert. Er wird bei der Abarbeitung eines FAS–Zyklus benötigt, um die Differenz $u - u_{init}$ im feineren Gitter zu korrigieren.

5.2.2 Lösen der diskreten Gleichung

Die Lösung der diskreten Gleichung ist analog zu (2.7) definiert als

$$u_i = a_{ii}^{-1} \left(f_i - \sum_{j,\ j \neq i} a_{ij} u_j \right)$$

und kann in der Regel für alle inneren Punkte gleichförmig dargestellt werden. Zum Beispiel für die bei Raum–Problemen verwendete 5–er Diskretisierung und $h = h_x = h_y$ ist

$$\begin{aligned} u_h(x,y) &= \frac{1}{4}(u_h(x-h,y) + u_h(x+h,y) \\ &\quad + u_h(x,y-h) + u_h(x,y+h) - h^2 f_h(x,y)). \end{aligned} \tag{5.1}$$

Die *Lösung des Gleichungssystems* wird für die Relaxation als auch die Berechnung des skalierten Residuums verwendet. In der Implementierung berechnet die Funktion

```
value Point::calcNewValue() {
    ...
}
```

eine Lösung u', speichert sie jedoch nicht. Das problemunabhängige Relaxations–Verfahren `Point::relax` kümmert sich um die eigentliche Relaxation.

Die *Berechnung des Residuums* funktioniert analog zur Lösung einer Gleichung in der Funktion `Point::calcNewRes`. Da häufig auf das Residuum eines Punktes zugegriffen wird, wird der Wert bei Nachfrage berechnet und in der Variable `res` *konsistent* gespeichert. Das heißt, daß bei einer Modifikation einer Komponente der Gleichung das Residuum für ungültig erklärt wird. Diese Methode ist schneller als eine permanente Berechnung des abgefragten Residuums, benötigt jedoch auch etwas Speicher für jeden Punkt.

Als letzte gleichungsabhängige Funktion wird der Faktor $a_{i,i}^{-1}$ in einer Hilfsfunktion `Point::scaleResidualFactor` berechnet. Da diese Funktion zum skalieren eines Residuums für jeden Punkt eines Gitters u_h gleich ist, wurde sie als Klassenfunktion `static` mit Parameter h deklariert.

5.2.3 Definition der Randbedingungen

Um die Randpunkte zu definieren, muß für ein Problem in der Funktion `Point::isBorder` für einen Punkt entschieden werden, ob er eine Randbedingung diskretisiert. Die einfache Kontrolle der Position genügt für die meisten Probleme. Der Wert u des Randes wird durch die Methode `Point::initBorder` definiert.

Die rechte Seite f des Problems $Au = f$ wird für jeden Punkt bei Bedarf durch die Methode `Point::initF` berechnet.

5.2.4 Zugriff auf benachbarte Punkte

Der Zugriff auf jeden anderen Punkt des Gitters kann über den Aufruf

```
Point *other = (*grid)(x,y);
```

geschehen. Dabei werden automatisch Geisterpunkte angelegt, wenn noch kein Punkt an der Stelle (x, y) existiert. Für die im zweidimensionalen Fall oft verwendeten acht benachbarten Punkte ist eine *direkte Refenzierung über Pointer* um einiges schneller als die Suche über die Klasse `Grid`, benötigt jedoch zusätzlichen Speicher.

Lazy Evaluation: Es wird im gesamten Quelltext Wert darauf gelegt, daß Berechnungen so spät wie möglich durchgeführt werden. Wird auf ein Datum zugegriffen, so wird mit einer `need*` Operation zuvor sichergestellt, daß es existiert. Bei Bedarf wird es dabei zu diesem Zeitpunkt berechnet. Das gilt für das Residuum, die Initialisierung eines Punktes sowie für Zugriffe auf die Nachbarpunkte.

Die benachbarten Punkte werden sehr oft benötigt und als Pointer zur Verfügung gestellt. Vor einem Zugriff wird der Bedarf mit einem `need*Neighbours` Aufruf gemeldet. Danach ist sichergestellt, daß die benötigten Nachbar–Referenzen existieren. Wird ein Nachbar gefunden oder

gegebenenfalls erzeugt, so trägt sich der Punkt auch als Nachbar des gefundenen Punktes ein. Dadurch wird jede bidirektionale Verbindung zwischen zwei Punkten nur einmal erstellt und die Anzahl der Zugriffe über das Gitter halbiert sich.

Die Nachbarn werden auch benachrichtigt, falls sich der Wert u eines Punktes verändert hat. Es wird dann ein eventuell berechnetes Residuum für ungültig erklärt und ein Geist wird sich, falls er aus dem veränderten Punkt interpoliert ist, für ungültig erklären und wiederum seine Nachbarn benachrichtigen.

Die Funktionen `migrate*ToReal` zum Migrieren einer Menge von Geistern zu realen Punkten sind auch Teil der Nachbarverwaltung und dort in allen benötigten Varianten implementiert.

5.2.5 Definition der Transportoperatoren

Unabhängig von den Randbedingungen und der Diskretisierung des Operators aus Abschnitt 5.2.2 und 5.2.3 müssen bei Raum– als auch bei Zeit–Problemen die Transportoperatoren *Prolongation* $I_h^{h/2}$ und *Restriktion* $I_{h/2}^h$ sowie $\tilde{I}_{h/2}^h$ definiert werden.

Die Prolongation $I_h^{h/2}$ wird bei zwei Schritten eines adaptiven FAS–Zyklus benötigt. Erstens bei der Korrektur von u am Ende eines V–Zyklus und zweitens bei der Interpolation eines Geisterpunktes. Die beiden Funktionen implementieren die Prolonagtion getrennt in `Point::correctU_fromCoarser` und `Point::initByInterpolation`, um Optimierungen des Compilers wie das „inlining“ zu ermöglichen.

Die Restriktion $I_{h/2}^h$ wird verwendet, um f, $f - Au$ und $|f - Au|$ gewichtet auf ein gröberes Gitter zu bringen. Da sich die drei Funktionen `Point::calcRestricted*` nur im Namen und in der Zugriffsfunktion auf die Werte unterscheiden, wird eine Art „Template“ als Makro definiert — wiederum, um dem Compiler die Möglichkeit zur Optimierung zu geben.

Die bei der adaptiven Verfeinerung benötigte Menge der sogenannten „kooperierenden Punkte“ aus Gleichung (3.11) wird auch in diesem Teil erwähnt, da sie vom Typ des Problems abhängig ist und aus der Art der Prolongation (3.9) als auch der Restriktion (3.10) bestimmt wird. Die Funktionen `Point::needAllNeighbours` und `Point::migrateAllNeighboursToReal` bestimmen nun genau diese Menge. Mehr dazu in Abschnitt 3.1.3.

5.2.6 Verwaltung eines Geisterpunktes

Ein Geisterpunkt i unterscheidet sich von einem realen Punkt dadurch, daß sein Wert u_i immer aus benachbarten oder gröberen Punkten interpoliert

wird und werden kann. Geister können jederzeit initialisiert und deren Residuum kann jederzeit berechnet werden. In der Implementierung wird ein Geist jedoch mitsamt seinen Daten im Speicher gehalten. Dadurch wird der Zugriff auf seinen Wert u_i und auf das Residuum effizienter, da es nur so oft wie nötig berechnet wird. Die Invalidierung der Werte muß dagegen sehr sorgfältig gehandhabt werden (siehe auch Abschnitt 5.2.4).

5.2.7 Aufteilung des Source–Code

Die einzelnen Funktionen der Klasse `Point` sind auf verscheidene Dateien verteilt. Eine Übersicht der Quelltexte aller Klassen ist im Anhang A aufgelistet.

Problemspezifisch: Diese Funktionen aus Abschnitt 5.2.2 und 5.2.3 werden in den Dateien `PointProblemSpace.cc` und `PointProblemTime.cc` implementiert. Jeweils eine davon wird durch ein Makro gesteuert compiliert. Aus dem Header `Point.h`:

```
value calcNewValue();
value calcNewRes();
static value scaleResidualFactor(Coordinate& h);

int isBorder();
void initBorder(value &setU);
void initF(value &setF);
```

Nachbarschaftsverwaltung: Die Suchfunktionen und die Operationen zum migrieren eines Geists in einen realen Punkt aus Abschnitt 5.2.4 sind in `PointNeighbours.cc` implementiert.

Transportoperatoren: Die Funktionen aus Abschnitt 5.2.5 werden in den Dateien `PointOpsSpace.cc` und `PointOpsTime.cc` implementiert. Wiederum wird jeweils eine davon durch ein Makro gesteuert compiliert. Aus dem Header `Point.h`:

```
void initByInterpolation(value& setU);
void correctU_fromCoarser();

value calcRestrictedF(Point *p);
value calcRestrictedResidual(Point *p);
value calcRestrictedAbsolutResidual(Point *p);

void needAllNeighbours();
void migrateAllNeighboursToReal();
```

Veraltung und Relaxation: Die rein punktspezifischen Operationen wie die Steuerung der Initialisierung, die Verwaltung des Residuums und die Zustandskontrolle aus Abschnitt 5.2.6 sind in `Point.cc` implementiert.

5.3 Implementierung der Relaxation

Die Relaxations–Funktionen für die normale Relaxation (`Point::relax`) und für die adaptive Relaxation (`Point::relaxAdaptive`) sind in der Datei `Point.cc` als statische Funktionen implementiert. Abschnitt 3.2 ab Seite 33 befaßt sich mit den Algorithmen an sich.

5.3.1 Relaxation

Die herkömmliche Relaxation relaxiert jeden Punkt i (`g[i]`) aus dem Gitter u_h (`g`) insgesamt n mal. Dabei wird für eine gewichtete Punkt–Relaxation ein ω (`weight`) als Parameter mit angegeben. Der Quelltext:

```
void Point::relax(Grid& g, int n, double weight) {
    for(; n>0; n--)
        for(unsigned int i=0; i < g.getCount(); i++)
            g[i]->relax(weight);
}
```

Die Reihenfolge der Relaxation verläuft wie in Abschnitt 5.1.3 beschrieben quasi zufällig über das Gitter und ist durch den linearen Zugriff (Abschnitt 5.1.5) auf die Elemente des Gitters definiert.

5.3.2 Adaptive Relaxation

Der Algorithmus zur adaptiven Relaxation ist in Abbilung 3.10 dargestellt und analog dazu implementiert. Die aktive Mange ist normalerweise vom Typ `Queue`, bei parabolischen Problemen kann sie auch vom Typ `PriorityQueue` sein, wobei dann alle zu relaxierenden Punkte zeitsortiert abgearbeitet werden. Zusätzlich wird der Parameter Θ_{max} (`maxsres`) übergeben, der den maximal erlaubten Wert des skalierten Residuums angibt. Die Funktion besteht aus folgendem Grundgerüst:

```
void Point::relaxAdaptive(Grid& g, value maxsres,
                          double weight)
{
    // PUT ANY NODE INTO THE ACTIVE SET
    Queue activeSet();
    for(unsigned int i=0; i < g.getCount(); i++)
        g[i]->putIntoQueue(activeSet);

    // RELAX ACTIVE SET UNTIL IT IS EMPTY
    Point *one;
    while((one = (Point*) activeSet.get()) != nil) {
        one->setInQueue(FALSE);
        if(one->relax(weight, maxsres)) {
            // => NODE RELAXED =>
            // PUT DEPENDING NODES INTO THE ACTIVE SET
            ...
        }
    }
}
```

Geschwindigkeit. Bei Messungen mit „Profiling–Software“ zeigt sich, daß der Anteil der Relaxation mit ca. 40% in dieser Implementierung bis über 95% in nicht adaptiven Mehrgitter–Implementierungen sehr groß und damit der Hauptansatzpunkt zur Optimierung ist.

Die reine Relaxationsgeschwindigkeit ist sehr stark vom Operator abhängig. Da in der Implementierung weniger Wert auf Effizienz als auf eine klar strukturierte Implementierung gelegt wurde, ist die Berechnung einer punktuelle Relaxation (Absatz 5.2.2) nicht optimiert. So wird beim Laplace–Modellproblem bei jeder Relaxation der Ausdruck

```
needFourNeighbours();
value dhx = getHx(), dhy = getHy();
value dx2 = dhx*dhx, dy2 = dhy*dhy;
return (dy2*(nl->getU()+nr->getU())
        + dx2*(nt->getU()+nb->getU())
        - dx2*dy2*f) / (2 * (dx2 + dy2));
```

ausgewertet, wobei die Umwandlung der Diskretisierung h_x und h_y von Fix– in Fließkommawerte jeweils eine `double`–Operation benötigt. Im häufigen Fall, daß $h_x = h_y$ gilt, wird die vereinfachte Gleichung gelöst. Die etwas einfachere zeitabhängige Berechnung von `Point::calcNewValue` benötigt im Assembler–Code des Testsystems[1] fünf Additionen, vier Divisionen und vier Multiplikationen mit `double` Werten. Die mehr als 50 restlichen Assemblerbefehle sind Speicherzugriffe, Konsistenzprüfungen und eventuelle Funktionsaufrufe. Die gemessene Geschwindigkeit beim Lösen des parabolischen Modellproblems ist ca. 125000 Relaxationen je Sekunde. Bei ausschließlich raumabhängigen Problemen gibt es für die vier benötigten Nachbarn mehr Verwaltungsaufwand und daraus resultierende 105000 Relaxationen je Sekunde für das elliptische Modellproblem. Im Vergleich dazu berechnet ein nicht adaptives C–Programm in einer gleichen Umgebung ca. 470000 Relaxationen je Sekunde.

Der Geschwindigkeitsfaktor 4.5 ist in Anbetracht der sehr übersichtlichen Implementierung und der vollen Adaptivität erträglich. Eine optimierte, weniger flexible Implementierung könnte die Effizienz der Berechnung steigern. Ein großes Handicap der durchgefürten Implementierung sind jedoch die sehr flexiblen Datenstrukturen und die damit zusammenhängende nicht uniforme Speicherbenützung der Algorithmen.

[1] Das Testsystem ist ein Intel „Pentium“ Rechner mit 90 MHz CPU–Takt, 64 MByte Speicher und dem gcc 2.5.8 als Compiler. In der höchsten Optimierungsstufe wurden die Optionen „-O2 -ffast-math -fomit-frame-pointer -m486 -fnonnull-objects“ verwendet.

5.4 Implementierung des Mehrgitterverfahrens

Das Mehrgitterverfahren *FAS* und das volladaptive Mehrgitterverfahren *FAMe*, wie in Kapitel 2 beschrieben, werden in `mgm.cc` implementiert. Die dazu notwendigen Operatoren wurden im Abschnitt 5.2 ab Seite 60 vorgestellt. Zusammenfassend sind das

- die *Relaxation* eines Gitters (Abschnitt 5.3),
- die *Prolongation* für die u–Korrektur (Abschnitt 5.2.5),
- die *Restriktion* für die τ–Korrektur und den Fehlerschätzer (Abschnitt 5.2.5) sowie
- die Definition der *Verfeinerungs–Patches* $coop_h^{h/2}(i)$ aus Abschnitt 3.1.3, Seite 30.

Aus diesen Operationen und einigen in `Point.cc` definierten statischen Hilfsfunktionen, die einen Operator jeweils auf eine Menge von Punkten anwenden, wurde nun in der Funktion `solveFas` des FAS–Verfahren und in der Funktion `solveAdaptiveFas` der FAMe–Algorithmus implementiert.

5.4.1 Das Full Approximation Storage Scheme FAS

Die einzelnen Arbeitsschritte des FAS–Verfahrens sind in der Implementierung einzelne Aufrufe der statischen Hilfsfunktionen. Diese Hilfsfunktionen führen in der Regel eine gleichnamige Operation auf jedem Punkt des übergebenen Gitters aus. Die folgenden Zeilen spiegeln den original Quelltext wider:

```
void solveFas(Grid& g, RelaxHow relaxPre, RelaxHow relaxPost) {
```

- Die Vorglättung mit v_1 Durchgängen oder einer gegebenen Schranke für den adaptiven Glätter (entnommen aus dem Parameter `relaxPre`) wird durchgeführt. Hier wird die normale Relaxation bevorzugt.
  `Point::relax(g, relaxPre)`, beziehungsweise
  `Point::relaxAdaptive(g, relaxPre)`
- Im feinsten Gitter wird der Vorgang abgebrochen.
  ```
  if(!g.hasCoarserLevel())
     return;
  ```
- Das gröbere Gitter wird durch Restriktion $I_{h/2}^h$ von f und Injektion $\tilde{I}_{h/2}^h$ von u initialiert.
  `Point::initCoarserGrid(g);`

- Die τ–Korrektur am gröberen Gitter wird durchgeführt.
  `Point::correctF_inCoarser(g);`
- Das gröbere Problem wird rekursiv mit demselben Verfahren gelöst.
  `solveFas(g.getCoarser(), relaxPre, relaxPost);`
- Die u–Korrektur am feineren Gitter wird durchgeführt.
  `Point::correctU_fromCoarser(g);`
- Die Nachglättung mit v_2 Durchgängen oder einer gegebenen Schranke für den adaptiven Glätter (entnommen aus dem Parameter `relaxPost`) wird durchgeführt. Hier ist die adaptive Relaxation oft vorzuziehen.
  `Point::relax(g, relaxPost)`, beziehungsweise
  `Point::relaxAdaptive(g, relaxPost)`

}

5.4.2 Die Fully Adaptive Multigrid Method FAMe

Ähnlich wie das FAS–Schema ist die Implementierung der FAMe–Methode eine direkte Umsetzung des symbolischen Algorithmus in Abbildung 3.9 auf Seite 31. Der Funktionsaufruf ist umfangreicher und definiert als

```
void solveAdaptiveFas(Grid& g, Dimension maxLevel,
                      value diffMin, RelaxHow relaxPre,
                      RelaxHow relaxPost, unsigned int cnt);
```

wobei das Gitter u (als `g`), die feinste erlaubte Diskretisierung h_{fein} (`maxLevel`), die Unterschranke des restringierten skalierten Residuumsbetrags δ_{min} für die adaptive Verfeinerung (`diffMin`), die Art der Pre– und Post–Relaxation (`relaxPre`, `relaxPost`) und die Anzahl der V–Zyklen (`cnt`, Schritt 2 in Abbildung 3.8) je Arbeitsgang übergeben werden. Die vollständige Implementierung ist am Ende des Kapitels in Abbildung 5.2 abgedruckt.

Der wichtigste Punkt ist die Berechung des *Verfeinerungskriteriums* aus Abschnitt 3.1.3 in der Variable `scaledRestrRes`. Es wird aus dem nur aus Geistern bestehenden feinen Gitter $u_{h/2}$ der Betrag der Residuen einer Punktmenge restringiert und mit einem für $h/2$ konstantem Skalierungsfaktor multipliziert. Die Methode `Point::migrateAllNeighboursToReal` bestimmt letztendlich wie in (3.11) definiert, welche Punkte verfeinert werden.

Startgitteraufbau. Schritt 1 in Abbildung 3.8, der *Startgitteraufbau*, wird nicht von der FAMe–Methode durchgeführt, sondern muß im Vorfeld geschehen. In der Startfunktion `main` wird das *initiale Gitter* aufgebaut, mit realen Punkten belegt und zur Lösung dem FAMe–Algorithmus übergeben. Die Wahl eines kleinen h_{start} ist von Vorteil, da dadurch Singularitäten

durch die diskretisierte Lösung von der adaptive Verfeinerung wahrgenommen werden.

Das gröbste Gitter `base` hat immer die Diskretisierung $h_x = h_y = 1$. Dadurch ergeben sich in den beiden Modellproblemen zwar keine Unbekannten, es ist jedoch die Ausgangsbasis für die gesteuerte Verfeinerung, wie es in Abschnitt 4.3.3 benötigt wird. Die Verfeinerung ist wie im Kapitel 4 erläutert, nicht immer frei wählbar. Beim elliptischen Modellproblem kann zwischen Standard– und Semi–Verfeinerung gewählt werden, so daß $h_x/2 \leq h_y \leq 2h_x$ gilt. Beim parabolischen Modellproblem muß man, um eine gewisse Konvergenz des Verfahrens zu erreichen, das Verhältnis $\lambda = h_t/h_x^2$ einhalten. Das Erzeugen der feineren Gitter ist deshalb nur durch die Methode `Grid::needFiner` möglich. Diese Methode kennt alle Anforderungen an die Gitterhierarchie und hält sie beim Verfeinern ein. Um zu prüfen, ob ein gewünschter Start–Level h_{start} erreicht ist, dient die Methode `Grid::reachedLevel`. Das elliptische Modellproblem entscheidet positiv, wenn $h_x \leq h_{start} \wedge h_y \leq h_{start}$ wahr ist, das parabolische bei $h_x \leq h_{start} \vee h_t \leq h_{start}$. Das Anlegen des Startgitters u_{start} ist nun folgendermaßen definiert:

```
Grid base(Coordinate::one, 17);
Grid *start = &base;
while(!start->reachedLevel(hStart)) {
    start = &start->needFiner(TRUE);
}
```

```
void solveAdaptiveFas(Grid& g, Dimension maxLevel, value diffMin,
                      RelaxHow relaxPre, RelaxHow relaxPost,
                      unsigned int cnt)
{
    // FAS: mit cnt V-Zyklen
    unsigned int i;
    for(i=0; i<cnt; i++)
        solveFas(g, relaxPre, relaxPost);

    // ABBRUCH im feinsten erlaubten Gitter
    if(g.reachedLevel(maxLevel))
        return;

    Grid& g_f = g.needFiner(FALSE);
    value scaleResFactor = Point::scaleResidualFactor(g_f.getH());

    int migrated = 0;
    for(i=0; i < g.getCount(); i++) {
        // ZUGRIFF auf einen inneren, realen Punkt:
        Point *one = g[i];
        one->needInit();
        if(!one->isInnerReal())
            continue;

        // ERZEUGE Maximale Punktmenge von feinen Geistern:
        Coordinate oneCoord(*one);
        Point *finer = g_f(oneCoord);
        finer->needAllNeighbours();

        // TESTE an jedem groben Punkt das Verfeinerungskriterium:
        value scaledRestrRes =
                one->calcRestrictedAbsolutResidual(finer)
                * scaleResFactor;
        if(fabs(scaledRestrRes) <= diffMin)
            continue;

        // MIGRIERE feinen Punkt mit allen Nachbarn zu realem Punkt
        finer->migrateAllNeighboursToReal();
        migrated++;
    }

    if(migrated > 0)
        solveAdaptiveFas(g_f, maxLevel, diffMin,
                         relaxPre, relaxPost, cnt);
}
```

Abbildung 5.2: Die Implementierung der volladaptiven Mehrgittermethode *FAMe* aus dem Quelltext `mgm.cc`.

Anhang A

Klassenindex

Die Implementiert umfaßt ein Programm zur Lösung der Modellprobleme im Raum für $x \times y$ und in der Zeit für $x \times t$. Dieser Anhang beschreibt die verwendeten Klassen kurz und Gliedert sie in die einzelnen Quelltexte auf.

Coordinate (5.1.3, Seite 55)
Speichert mehrdimensionale Koordinaten–Angaben und errechnet einen Objektindex vom Typ `Hashvalue`.

- `Coordinate.h` ... Deklaration
- `Coordinate.cc` ... Implementierung

Dimension (5.1.3, Seite 55)
Klasse zum speichern von eindimensionalen Werten. Die Implementierung benützt Fixkommaarithmetik mit ganzen Zahlen, deren Wortbreite in `Dimension.h` einstellbar ist.

- `Dimension.h` ... Deklaration mit Inline–Implementierungen
- `Dimension.cc` ... Implementierung

Grid (5.1.5, Seite 58)
Die Klasse `Grid` dient zur Speicherung der dünn besetzten Gitter und implementiert dafür die abstrakten Funktionen der Basisklasse `Hash`.

- `Grid.h` ... Deklaration
- `Grid.cc` ... Implementierung
- `GridPrinters.cc` ... Implementierung der Ausgabefunktionen

Hash (5.1.4, Seite 57)
Diese Klasse verwaltet eine Hashtabelle, deren Einträge Objekte vom Typ *Hashgroup* sind. Die Hashadresse eines Objektes wird aus dem *Objektindex* modulo der Tabellengröße berechnet. Die Tabellegröße wird bei der Initialisierung immer auf die nächst größere Primzahl gesetzt.

Die Klasse *Hash* kümmert sich um das *primäre* Zugriffsverfahren, die Klasse *Hashgroup* um das *sekundäre*.

- `Hash.h` ... Deklaration
- `Hash.cc` ... Implementierung

Hashgroup (5.1.4, Seite 57)
Eine *Hashgroup* ist die Gruppe aller Objekte mit gleicher Hashtabellen–Position. Diese Gruppen sind in der Regel sehr klein und werden als Array mit lineare Suche implementiert.

- `Hashgroup.h` ... Deklaration
- `Hashgroup.cc` ... Implementierung

Hashvalue (5.1.3, Seite 56)
Hashvalue i st der Typ eines Objektindex, durch den eine Position in der Hashtabelle errechnet wird und ist als `unsigned int` definiert.

- `Hashvalue.h` ... Deklaration

Point (5.2, Seite 60)
Ein sogenannter Punkt oder Node ist eine von der Gitterweite abhängige Diskretisierung des Problems und kann mehrere in `PointTypes.h` definierte Zustände haben.

- `Point.h` ... Deklaration mit vielen Inline–Implementierungen
- `PointTypes.h` ... Deklaration von `PointType` und `PointKind` sowie verschiedener Hilfstypen.
- `Point.cc` ... Implementierung der *Verwaltungs–Operationen* Destruktion sowie verschiedene Initialisierungen. Auch Frontends zu den Mehrgitter–Operationen τ*–Korrektur* und *Relaxation* sind enthalten. Zusätzlich werden alle statischen Hilfsfunktionen definiert.
- `PointNeighbours.cc` ... Implementiert die Nachbarschafts–Verwaltung. Durch diese Funktionen ist ein sehr schneller Zugriff auf die benachbarten Punkte gegeben. Insbesondere wird eine Modifikation von u_i auf diesem Weg bekanntgegeben.
- `PointOpsSpace.cc PointOpsTime.cc` ... Implementiert die Mehrgitter–Operationen *Prolongation* und *Restriktion*.
- `PointProblemSpace.cc, PointProblemTime.cc` ... Implementiert die *Randbedingungen*, das Backend für die Berechnung einer *Relaxation* und für Testzwecke eventuell auch die Berechnung des echten Fehlers.

PriorityQueue. PriorityQueue ist eine *priorisierte Warteschlange* mit einer konstanten Anzahl von Prioritäten. Analog der *Queue* definiert sie die Operationen `put()` und `get()`.

- `PriorityQueue.h` ... Deklaration mit Inline–Implementierungen

Queue. Diese Warteschlange definiert einen FIFO (first in, first out) Puffer mit dynamischer Größe und den Operationen `put()` und `get()`.

- `Queue.h` ... Deklaration mit Inline–Implementierungen
- `Queue.cc` ... Implementierung

Andere Quelltexte. Folgende Quelltexte stellen die eigentlichen Anwendungen zur Verfügung bzw. dienen derer Erstellung:

- `Std.h` ... Definition des wichtigen Datentyp `Boolean` und des als Null–Referenz verwendeten Wertes `nil`.
- `Makefile` ... Das Makefile bietet verschiedene Steuermöglichkeiten zur Compilierung an. Das wichtigste Target ist `everything`, womit sechs Programme erzeugt werden: Die `base*` Programme zum Plot der Basisfunktion und die `mgm*` Programme zum lösen der Modellprobleme mit Standard– und Semi–Verfeinerung jeweils für Raum und Zeit.
  Die Optionen `CC`, `CFLAGS` und `LDFLAGS` können für eine gewählte Plattform modifiziert werden.
- `base.cc` ... Ein Programm zur Darstellung der Basisfunktionen im `gnuplot` Format. Die beiden Parameter für den Aufruf sind h_{start}^{-1} und h_{plot}^{-1}.
  h_{start} ist die Diskretisierung eines Punktes und h_{plot} die Ausgabegenauigkeit des Plots.
- `mgm.cc` ... (5.4, Seite 68) Das eigentliche Hauptprogramm. Es enthält das Mehrgitterverfahen nach dem *„Full Approximation Scheme“* FAS in `solveFas()` und das Verfahren zur adaptiven Verfeinerung `solveAdaptiveFas()`.
  Die Aufrufparameter werden in Anhang B näher betrachtet.

Anhang B

Die Aufrufparameter von mgm

Die Aufrufparameter für die verschiedenen Varianten des `mgm` Programmes sind folgendermaßen angeordnet:

```
./mgm [size] [count] [print] [maxsr] [startsz] [relaxPre] [relaxPost]
```

Die einzelnen Parameter haben dabei folgende Bedeutung:

size bestimmt die feinste zulässige Diskretisierung ($size = h^{-1}$).

count ist die Anzahl der V–Zyklen im FAS.

print gibt an, was wie auf der Standard–Ausgabe erzeugt werden soll:

- 1 ... die Lösung u_h des feinsten Gitters.
- 2 ... das Residuum $f_h - A_h u_h$ des feinsten Gitters.
- 3 ... der berechnete Fehler $u_h - u_h^*$ des feinsten Gitters.
- 4 ... die Punkt–Struktur des feinsten Gitters.
- 5 ... die Lösung u über die hierarchische Gitterstruktur.
- 6 ... das Residuum $f - Au$ über die hierarchische Gitterstruktur.
- 8 ... die rechte Seite f des feinsten Gitters.
- 9 ... die Struktur des feinsten Gitters im TEX–Format.
- 20 ... die Anzahl der Relaxationen je Punkt des feinsten Gitters.

Alle Ausgaben werden im (x y z)–Format Zeilenweise ausgegeben, sodaß das Programm `gnuplot` die Daten direkt einlesen kann. Die Optionen 6 und 8 erzeugen eine nicht uniforme Menge von Punkten, da alle realen Punkte der Gitterhierarchie übereinander geschrieben werden. Die Option 9 erzeugt keine Plot–Informationen sondern einen Aufsicht auf die Gitterstruktur im TEX–Format.

maxsr ist das Kriterium δ_{min} für die adaptive Verfeinerung (3.1.3, ab Seite 27).

startsz gibt die Diskretisierung des Startgitters an ($startsz = h_{start}^{-1}$).

relaxPre gibt das obere Limit Θ_{max} des skalierten Residuums für die adaptive Relaxation *vor* einem V–Zyklus an (siehe 3.2, Seite 31). Bei einem negativen Wert wird für die Vorglättung die normale Relaxation verwendet ($v_1 = -relaxPre$).

relaxPost bestimmt analog zu *relaxPre* die Relaxation *nach* einem V–Zyklus, die Nachglättung.

Literaturverzeichnis

[Bra77] A. Brandt. Multi–level adaptive solutions to boundary–value problems. *Math. Comp.*, 31:333–390, 1977.

[Bri87] W. L. Briggs. *A Multigrid Tutorial*. SIAM Books, Philadelphia, 1987.

[BSMM95] I. Bronstein, K. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, zweite edition, 1995.

[Gri90] M. Griebel. *Zur Lösung von Finite–Differenzen– und Finite–Element–Gleichungen mittels der Hierarchischen–Transformations–Mehrgitter–Methode*. PhD thesis, Institut für Informatik, TU München, 1990.

[Gri94] M. Griebel. Multilevel algorithms considered as iterative methods on semidefinite systems. *SIAM J. Sci. Stat. Comput.*, 15:547–565, 1994.

[GZ97] M. Griebel and G. W. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing. In E. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Proceedings of ParCo '97*, pages 589–599. Elsevier, 1997.

[HV95] G. Horton and S. Vandewalle. A space time multigrid method for parabolic partial differential equations. *SIAM J. Sci. Comput.*, 16:848–864, 1995.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison–Wesley, 1973.

[Löt96] H. Lötzbeyer. Parallele adaptive Mehrgitterverfahren. Diplomarbeit, Institut für Informatik, TU München, Dezember 1996.

[LR97] H. Lötzbeyer and U. Rüde. Patch-adaptive multilevel iteration. *BIT*, 37:739–758, 1997.

[McC89] S. F. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*, volume 6 of *Frontiers in Applied Mathematics*. SIAM Books, Philadelphia, 1989.

[McC92] S. F. McCormick. *Multilevel Projection Methods for Partial Differential Equations*, volume 62 of *CBMS–NSF*. SIAM Books, Philadelphia, 1992.

[OW96] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum, Akademischer Verlag, dritte edition, 1996.

[PR93] C. Pflaum and U. Rüde. Gauß' adaptive relaxation for the multilevel solution of partial differential equations on sparse grids. Technical Report TUM-I9327, SFB 342/13/93 A, Institut für Informatik, TU München, September 1993.

[Rüd88] U. Rüde. *Zur numerischen Behandlung von Singularitäten in elliptischen partiellen Differentialgleichungen*. PhD thesis, Institut für Informatik, TU München, 1988.

[Rüd92] U. Rüde. On the V–cycle of the fully adaptive multigrid method. Technical Report TUM-I9215, Institut für Informatik, TU München, May 1992.

[Rüd93a] U. Rüde. Fully adaptive multigrid methods. *SIAM J. Numer. Anal.*, 30:230–248, 1993.

[Rüd93b] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 1993.

Online–Literatur: Die aufgelistete Literatur ist teilweise unter `http://www.mgnet.org/mgnet/papers` online verfügbar.