Algorithmen und Datenstrukturen II

–Parallele Algorithmen–

Prof. Dr. Andreas Frommer

Wintersemester 2005/06

Bergische Universität Wuppertal
Fachbereich Mathematik

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmen

Vorwort

Die Vorlesung „Parallele Algorithmen“ wird in Wuppertal jährlich angeboten. Ihr Ziel ist es, Studierenden der Mathematik, der Informationstechnologie und des internationalen Studienganges „Computer Simulation in Science“ in numerische Verfahren für Parallelrechner einzuführen. Im Vordergrund steht dabei das Lösen linearer Gleichungssysteme mit direkten und – in der Praxis wichtiger – modernen iterativen Verfahren. Dabei gehen wir Fragen nach der (parallelen) Effizienz und (ggfs.) der Konvergenz ebenso nach wie grundsätzliche Fragen nach der parallelen Implementierung und der Skalierbarkeit. Noch immer besitzt die Universität Wuppertal mit ALiCEnext den derzeit leistungsfähigsten Parallelrechner, der an einer deutschen Universität installiert ist, und der hier angebotene Kurs bereitet in Verbindung mit dem Parallelrechnerpraktikum auch auf entsprechende Diplomarbeiten und Master-Theses vor.

Die Vorlesung wird ab sofort in der Regel in englischer Spürache abgehalten. Das vorliegende Skript ist gedacht als Unterstützung für all diejenigen, die ergänzend eine deutschsprachige Version des Lehrinhaltes zur Verfügung haben wollen. Es ist keine Eins-zu-eins Umsetzung der Vorlesung.

Ich bin den Herren Sven Blumberg und Stephan Borovac zu besonderem Dank verpflichtet. Herr Blumberg hat im Wintersemester 2002/2003 mit großem persönlichem Einsatz aus eigener Initiative eine erste LaTeX-Version dieses Skriptes erstellt. Herr Borovac hat die Übungen zu der Veranstaltung mehrfach gestaltet und dabei insbesodere auch zahlreiche Verbesserungen in das Skript eingebracht. Alle noch verbleibenden Fehler – und das sind sicher nicht wenige – liegen nun in meiner Verantwortung.

Wuppertal, den 8. Februar 2006 Andreas Frommer

Kapitel 1

Parallelrechner

1.1 Architekturen

Ein **Parallelrechner** ist ein Rechner mit $p > 1$ Prozessoren, die *gleichzeitig* an einer Aufgabe arbeiten. Parallelrechner kann man auf zwei Arten klassifizieren:

1. nach Programmiermodell

   SIMD (single instruction multiple data):
   Eine zentrale Kontrolleinheit arbeitet ein Programm ab, jeder Prozessor arbeitet dabei im Takt dieselbe Anweisung mit individuellen Daten ab.

   SPMD (single program multiple data):
   Jeder Prozessor arbeitet dasselbe Programm mit verschiedenen Daten ab.

   MIMD (multiple instruction multiple data):
   Jeder Prozessor arbeitet ein individuelles Programm mit individuellen Daten ab.

   Der Übergang von SPMD nach MIMD ist fließend (z.B. durch Verzweigungen in SPMD-Programmen, die von der Prozessoridentifikation abhängen).

2. nach Speichermodell

   gemeinsamer Speicher: Alle Prozessoren greifen auf einen gemeinsamen Speicher zu (technisch nur bis $p \approx 100$ möglich, näheres später).

verteilter Speicher: Jeder Prozessor besitzt einen eigenen internen Speicher, Datenaustausch erfolgt durch Kommunikation.

SPMD stellt das am meisten verbreitete Modell dar, SIMD ist auch ein Modell für sogenannte Vektorrechner.

1.1.1 Beispiel

Die Unterschiede zwischen den einzelnen Programmiermodellen erkennt man am einfachen Beispiel der Minimumsfunktion.

Programm:

if $a < b$ **then**
  $b = a$
else
  $a = b$
end if

Takt	SIMD		SPMD	
	P_1	P_2	P_1	P_2
	$a = 3, b = 4$	$a = 4, b = 3$	$a = 3, b = 4$	$a = 4, b = 3$
takt 1	$a < b$=true	$a < b$=false	$a < b$=true	$a < b$=false
takt 2	$b = a$	–	$b = a$	$a = b$
takt 3	–	$a = b$	–	–

Beim SIMD-Modell entstehen in diesem Beispiel zwei Totzeiten, in denen die Prozessoren nicht arbeiten, also Rechenzeit verlorengeht.

Prinzipiell treten auch bei der SPMD-Architektur Totzeiten auf, nämlich dann, wenn die Prozessoren synchronisiert werden müssen (im Beispiel nicht erkennbar).

Im MIMD-Modell können auf verschiedenen Prozessoren voneinander unabhängige Prozesse laufen, beispielsweise eine Schleife auf P_1 und die Minimumsfunktion auf P_2.

Eine besondere Variante des SIMD-Modells bilden *Vektorprozessoren.* Dabei erreicht man Parallelität innerhalb einer einzelnen Operation.

1.1.2 Beispiel

Die Addition zweier Gleitpunktzahlen $x = 0.m_x \cdot 2^{e_x}, y = 0.m_y \cdot 2^{e_y}$ unterteilt sich in 4 Schritte

1. Exponentenabgleich: berechne $e_x - e_y$

2. Verschieben einer Mantisse: $y = 0.m'_y \cdot 2^{e_x}$
3. Addition der Mantisse: $m_z = m_x + m'_y$
4. Normalisierung des Ergebnisses z

Nehme z. B. $x = 0.1101 \cdot 2^3, y = 0.1001 \cdot 2^2$

1. $e_x - e_y > 0$
2. $y = 0.0100 \cdot 2^3$ (! bei 4-stelliger Mantisse)
3. $m_z = 0.1101 + 0.0100 = 1.0001$
4. $z = 1.0001 \cdot 2^3 \stackrel{!}{=} 0.1000 \cdot 2^4$

Bei Vektorprozessoren wird dieselbe Operation auf vielen Operanden dadurch "parallelisiert", dass die verschiedenen Schritte mit verschiedenen Operanden gleichzeitig durchgeführt werden.

	1. Takt	2. Takt	3. Takt	4. Takt	5. Takt	
1. Schritt	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4	x_5, y_5	
2. Schritt		x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4	$\ldots$
3. Schritt			x_1, y_1	x_2, y_2	x_3, y_3	
4. Schritt				x_1, y_1	x_2, y_2	

Beachte: Vektorprozessoren benötigen besonders schnelle Speicher $\rightarrow$ teuer

Die Unterschiede im Speichermodell kann man folgendermaßen illustrieren:

1. Gemeinsamer Speicher (Shared Memory): Bildlich kann man sich den gemeinsamen Speicher wie in Abbildung 1.1 vorstellen.

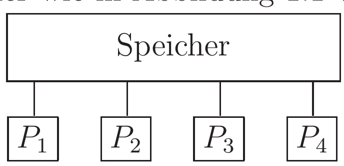


Abbildung 1.1: Gemeinsamer Speicher

Ein gemeinsamer Speicher bedeutet einen Mehraufwand an Verwaltung. So müssen Zugriffskonflikte vermieden werden, und es muss möglich sein, *private* Speicherbereiche zu allokieren.

Für Systeme mit gemeinsamen Speicher hat sich zur Programmierung der OpenMP-Standard durchgesetzt.

2. Verteilter Speicher (Distributed Memory): Wir betrachten zwei verschiedene Schemata

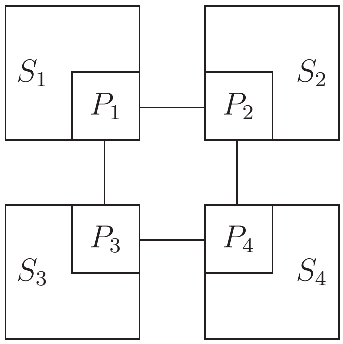


Abbildung 1.2: Verteilter Speicher: Kommunikationskanäle

   (a) Bei einer Architektur mit *Kommunikationskanälen* sind einige Prozessoren und ihr Speicher direkt mit anderen verbunden. Auf großen Systemen können technisch nicht alle Prozessoren direkt verbunden werden.

      Es entsteht der Nachteil, dass (in Abbildung 1.2) eine Kommunikation von P_1 nach P_4 nur über die Zwischenstationen P_2 bzw. P_3 möglich ist. Dies muss zum Teil vom Anwender explizit implementiert werden. Neuere Programmiersprachen/Betriebssysteme nehmen dies dem Programmierer aber auch ab.

   (b) Ein *Netzwerk mit Schaltern* bietet die Möglichkeit, durch Schalten direkte Verbindungen zu allen anderen Prozessoren herzustellen. Wie die Schaltungen geschaltet werden, wird durch einen Router entschieden, ohne dass der Programmierer darauf einen Einfluss hat.

      Der Clustercomputer ALiCEnext in Wuppertal besitzt ein solches Netzwerk mit Schaltern.

Bei Systemen mit verteiltem Speicher muss eine Programmiersprache das sogenannte **„Message Passing“** (Versenden/Empfangen von Nachrichten zwischen den Prozessoren) unterstützen.

Hier hat sich der Standard MPI (Message Passing Interface) durchgesetzt.

Statt der Speicherkonflikte beim gemeinsamen Speicher hat man nun Zeitverluste wegen der Kommunikation.

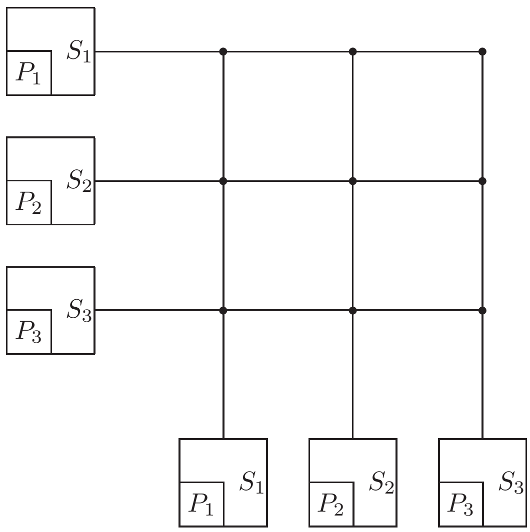


Abbildung 1.3: Verteilter Speicher: Netzwerk mit Schalter (Switch-Network) (physikalisch kommt jeder Prozessor natürlich nur einmal vor)

1.2 Architekturen mit festem Netzwerk

In diesem Abschnitt betrachten wir verschiedene (klassische) Arten der Anordnung von Prozessoren in einem Netzwerk. Im Allgemeinen ist nicht jeder Prozessor mit allen anderen verbunden, da „Nächste-Nachbar-Kommunikation" am schnellsten ist.

Es ist günstig das Netzwerk als einen topologischen Graphen aufzufassen, dessen Knoten die Prozessoren und dessen Kanten (bidirektionale) Kommunikationskanäle zwischen zwei Prozessoren darstellen.

1.2.1 Beispiel

1. Ring: p Prozessoren $P_0, ..., P_{p-1}$, in denen der Prozessor P_j mit den Prozessoren $P_{(j-1) \bmod p}$ und $P_{(j+1) \bmod p}$ verbunden ist.

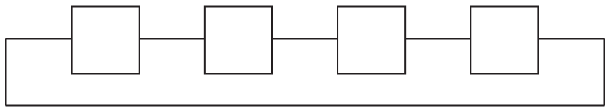


Abbildung 1.4: Ring

2. Binärbaum: $p = 2^{P+1} - 1$ Prozessoren, von denen eine *Wurzel* mit zwei *Söhnen* verbunden, $2^P - 2$ *innere Knoten* mit jeweils einem *Vater* und zwei Söhnen verbunden und 2^P *Blätter* mit je einem Vater verbunden sind.

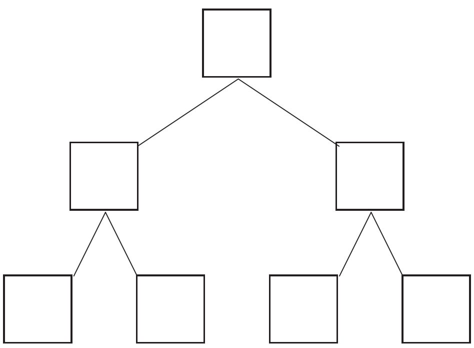


Abbildung 1.5: Binärbaum

3. Gitter: $p = N \cdot M$ Prozessoren, von denen bis auf die Randprozessoren alle vier Nachbarn besitzen

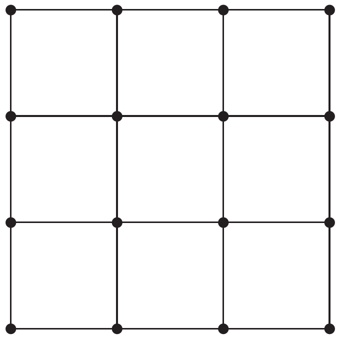


Abbildung 1.6: Gitter $(N = 4, M = 4)$

4. 2d-Torus: $p = N \cdot M$ Prozessoren, welche *alle* vier Nachbarn besitzen.

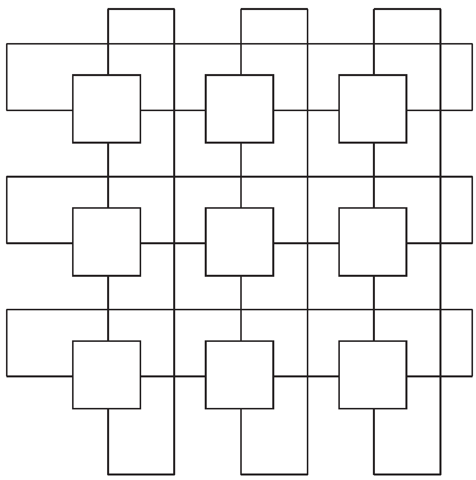


Abbildung 1.7: 2d-Torus $(N = 3, M = 3)$

5. N-dimensionaler Hypercube: Er besteht aus 2^N Prozessoren, die wir mit P_0,..., P_{2^N-1} durchnummerieren. Damit sind die Prozessoren durch eine N-stellige (Binär-) Zahl beschreibbar. Dabei gilt:

$$\begin{aligned} & \text{Zwei Prozessoren } P_i \text{ und } P_j \text{ sind verbunden} \\ :\Longleftrightarrow \quad & i,\ j \text{ unterscheiden sich um genau ein Bit.} \end{aligned}$$

Beispiel: Die Konstruktion des N-dimensionalen Hypercubes geht induktiv: Man zeichne zwei $(N-1)$-dimensionale Hypercubes mit den zu-

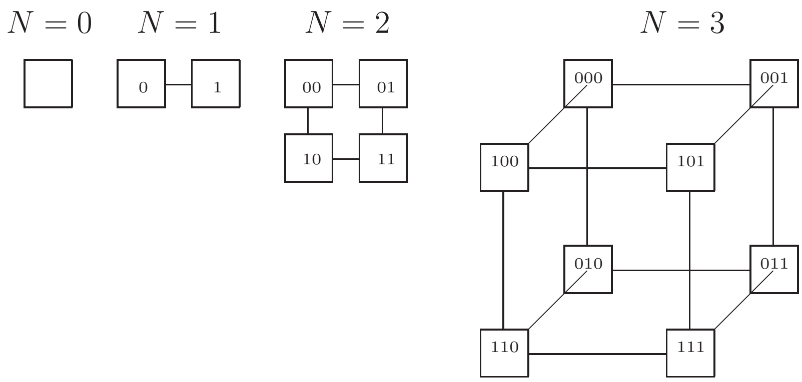


Abbildung 1.8: N-dimensionaler Hypercube

gehörigen Bitzahlen der Prozessoren. Man erweitert nun die Binärdarstellung des einen Hypercube mit einer führenden „0“ und die des anderen mit einer führenden „1“. Dann erstellt man die neuen Verbindungen zwischen den Prozessoren gemäß der obigen Regeln (siehe Abbildung 1.8).

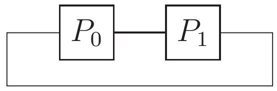
Man kann durch geschicktes Nummerieren zeigen, dass der 4×4-Torus und der 4-dimensionale Hypercube als Graphen topologisch äquivalent sind. Der Hypercube ist eine sehr „reiche“ Architektur, welche viele andere Architekturen enthält. Das wollen wir weiter untersuchen.

Bemerkung: Der Ring, der Binärbaum, der Torus und das Gitter sind skalierbare Architekturen, d.h. verhältnismäßig einfach zu erweitern. Die Anzahl der Kommunikationskanäle pro Prozessor ändert sich nämlich bei Erweiterung nicht. Beim Hypercube ist das nichtder Fall.

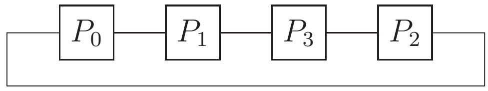
Beispiel: Jeder Hypercube enthält einen Ring.

Zum Verständnis: Die Zahlen stellen die Binärdarstellung der Indizes der Prozessoren dar, d.h. 10 stellt hier z.B. Prozessor P_2 dar.

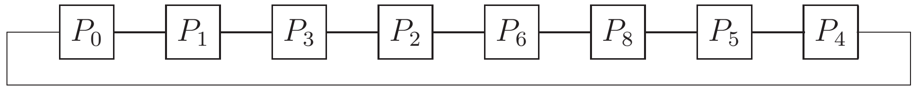
$N = 1$ 0-1



$N = 2$ 0-1 | 1-0
00-01 | 11-10



$N = 3$ 00-01-11-10 | 10-11-01-00
000-001-011-010 | 110-111-101-100



Prinzip: Nehme den $N-1$-dimensionalen Hypercube, spiegle diesen (genauer: die Binärdarstellung der Indizes) füge links eine führende 0, rechts eine führende 1 hinzu.

1.2.2 Definition
Wir definieren die beiden Rundungen für eine Zahl $x \in \mathbb{R}$ nach $\mathbb{Z}$ durch

1. $\lceil x \rceil := \{z \in \mathbb{Z} |\ z \geq x\}$ („**Rundung nach oben**“),
2. $\lfloor x \rfloor := \{z \in \mathbb{Z} |\ z \leq x\}$ („**Rundung nach unten**“).

1.2.3 Definition
Die Funktion *graycode* : $\mathbb{N}_0 \to \mathbb{N}_0$ ist definiert durch folgendes Programm

if $n = 0$ **then**
  $graycode(n) := 0$
else
  $N = \lfloor \log_2(n) \rfloor$ $\{2^N \leq n < 2^{N+1}\}$
  $\tilde{n} = 2^{N+1} - n - 1$ {Spiegelung}
  $graycode(n) = 2^N + graycode(\tilde{n})$
end if

Beispiel: Greifen wir noch einmal die Suche nach einem Ring im 3-dimensionalen Hypercube auf, so ergibt sich

n	0	1	2	3	4	5	6	7
graycode (binär)	000	001	011	010	110	111	101	100
graycode (dezimal)	0	1	3	2	6	7	5	4

Ordnet man die Zahlen $n = 0, \ldots, 7$ gemäß ihres *graycode* an, so unterscheiden sich benachbarte Zahlen um genau ein Bit. Wir weisen dies jetzt allgemein nach, wozu wir folgenden nicht-rekursiven Algorithmus für *graycode* benötigen:

$graycode = 0$
repeat
  $N = \lfloor \log_2(n) \rfloor$
  $n = 2^{N+1} - n - 1$
  $graycode = graycode + 2^N$
until $n = 0$

1.2.4 Satz
Sei $n \in \mathbb{N}_0$

1. Ist $0 \leq n < 2^N$, so ist auch $0 \leq \text{graycode}(n) < 2^N$.
2. $graycode: \mathbb{N}_0 \to \mathbb{N}_0$ ist bijektiv.
3. $graycode(n)$ und $graycode(n+1)$ unterscheiden sich genau um ein Bit für alle $n \in \mathbb{N}_0$.

Beweis:

1. Übungsaufgabe
2. Übungsaufgabe
3. Wir beweisen dies durch Induktion.

   $n = 0$: Es ist $graycode(0) = 0$ und $graycode(1) = 1$, der Unterschied ist genau ein Bit wie behauptet.

   Die Behauptung gelte für $n = 0, ..., 2^{N-1} - 1$ $(N \geq 2)$, wir untersuchen $n = 2^{N-1}, ..., 2^N - 1$. Es gilt

$$graycode(n) = 2^{N-1} + graycode(\underbrace{2^N - n - 1}_{=:\ \alpha < 2^{N-1}}).$$

(a) $n < 2^N - 1$. Dann ist $n+1 \leq 2^N - 1$, also gilt

$$graycode(n+1) = 2^{N-1} + graycode(\underbrace{2^N - n - 2}_{=:\ \beta < 2^{N-1}}).$$

Es ist also $\alpha = \beta + 1 < 2^{N-1}$ und damit unterscheiden sich nach Induktionsvoraussetzung $graycode(\alpha)$ und $graycode(\beta)$ um genau ein Bit. Damit unterscheiden sich auch $graycode(n)$ und $graycode(n+1)$ um genau ein Bit.

(b) $n = 2^N - 1$. Es folgt durch Anwenden des Algorithmus

$$\begin{aligned} graycode(n) &= 2^{N-1} \\ graycode(n+1) &= 2^N + graycode(\underbrace{2^{N+1} - (2^N + 1)}_{=2^N - 1 = n}) \\ &= 2^N + graycode(n) \\ &= 2^N + 2^{N-1} \end{aligned}$$

Damit unterscheiden sich $graycode(n)$ und $graycode(n+1)$ genau um das führende Bit.

$\square$

Bemerkung: Es gilt zusätzlich, dass sich

$$graycode(0) = 0 \text{ und } graycode(2^N - 1) = 2^{N-1}$$

ebenfalls um genau ein Bit unterscheiden. Dies liefert den Ringzusammenschluss, d.h. die Verbindung zwischen erstem und letztem Prozessor.

Mit Hilfe des graycode können wir darüberhinaus noch folgendes zeigen

1.2.5 Satz

1. Ein N-dimensionaler Hypercube enthält einen 2^N-Ring.

2. Ein N-dimensionaler Hypercube enthält jeden 2-dimensionalen $2^M \times 2^{N-M}$-Torus, für $M = 0, ..., N$.

3. Ein $2N$-dimensionaler Hypercube ist topologisch äquivalent zu einem N-dimensionalen $\underbrace{4 \times 4 \times \cdots \times 4}_{N\text{-mal}}$-Torus .

Beweis: Wir bezeichnen die Prozessoren des Hypercubes mit $P_0, ..., P_{2^N-1}$

1. Wir bezeichnen die Prozessoren des Ringes mit $Q_0, ..., Q_{2^N-1}$ und setzen
$$Q_i := P_{graycode(i)}.$$
Nach Satz 1.2.4 existiert zwischen Q_i und Q_{i+1} eine Kante für $i = 0, ..., 2^N - 2$, ebenso sind die Prozessoren Q_0 und Q_{2^N-1} verbunden (siehe vorherige Bemerkung).

2. Wir bezeichnen die Prozessoren des Torus mit $Q_{k,\ell}$, $k = 0, ..., 2^M - 1$, $\ell = 0, ..., 2^{N-M} - 1$ und setzen
$$Q_{k,\ell} := P_{graycode(\ell)\cdot 2^M + graycode(k)}.$$
Damit ist dann $Q_{k,\ell}$ mit $Q_{k\pm1,\ell}$ und $Q_{k,\ell\pm1}$ mit $k \pm 1 \mod 2^M$, $\ell \pm 1 \mod 2^{N-M}$ verbunden.

3. Übungsaufgabe

$\square$

1.3 Architekturen mit Switch-Netzwerk

Switch-Netzwerke können die Kommunikation eines Prozessors mit allen anderen gewährleisten. Sie repräsentieren keine feste Netzwerk-Architektur, da sie im allgemeinen rekonfigurierbar sind.

Ein 1-Switch-Netzwerk lässt sich schematisch folgendermaßen darstellen:

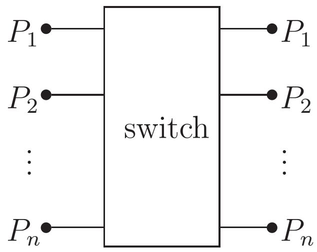


Abbildung 1.9: 1-Switch Netzwerk

Der Switch ist rekonfigurierbar (durch das Betriebssystem), er realisiert einige oder alle der $n!$ Permutationen auf der Menge $\{1, ..., n\}$.

Ein vollständiger Switch (auch Crossbar oder kurz Xbar genannt) realisiert *alle* Permutationen. Allerdings sind solche Xbars wegen technischer Hindernisse auf ca. 100 Ein-/Ausgänge begrenzt.

Man verwendet darum mehrstufige Netzwerke, die sich als Kombination von verschiedenen Xbars ergeben.

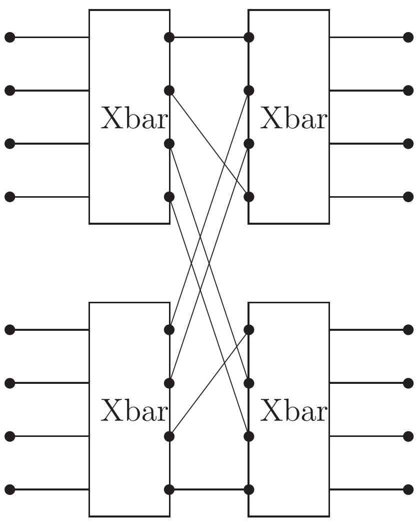


Abbildung 1.10: mehrstufiges Netzwerk

Wichtig sind hierbei hierarchische Realisierungen von vollständigen Netzwerken.

1.3.1 Beispiel
Der **Exchange-Switch**, d.h. der vollständige X-bar für zwei Ein-/Ausgänge, wird folgendermaßen realisiert

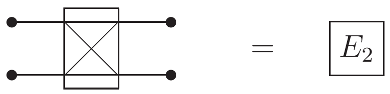


Abbildung 1.11: Exchange-Switch

1.3.2 Beispiel
Das **Benes-Netzwerk**, Rekursionsstufe 1 ist gegeben durch

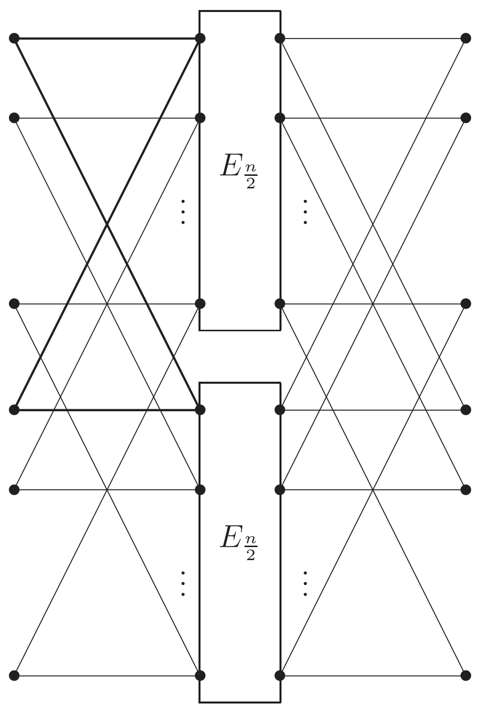


Abbildung 1.12: Benes-Netzwerk

Das Netzwerk bestehst aus $2 \cdot \frac{n}{2}$ E_2s und aus 2 $E_{\frac{n}{2}}$. (In der Abbildung ist ein E_2 durch dicke Linien hervorgehoben.) Dabei sei $E_{\frac{n}{2}}$ ein vollständiger Xbar auf $\frac{n}{2}$ Ein-/Ausgängen. Die Rekursion besteht nun darin, dass $E_{\frac{n}{2}}$ durch E_2s und zwei $E_{\frac{n}{4}}$ realisiert ist und so weiter.

Bemerkung: Auch unvollständige Netzwerke werden hierarchisch organisiert, z.B. Ω-Switches etc.

1.4 Aktuelle Klassifikationen und Hybride

In diesem Abschnitt werden einige kommerzielle Superrechner in Kategorien eingeteilt. Diese Klassifikation basiert auf Richtlinien der aktuellen Top500-Liste (http://www.top500.org).

Eingesetzt werden solche Rechner hauptsächlich in wissenschaftlichen Institutionen, beim Militär, bei Banken (zum schnellen Verarbeiten von großen Datenmengen, wie sie beim Erstellen von Kaufprofilen nötig sind) und bei High-Tech-Firmen, die schnelle Datenverarbeitungen z.B. für Echtzeit-Simulationen benötigen (vergleiche Kooperation von Daimler-Chrysler und der Uni Stuttgart).

1. Shared Memory SIMD

   Ein eher historisches Modell. Die ersten Vektorrechner fallen in diese Kategorie, d.h. Prozessoren, die sehr gut für Vektoroperationen optimiert sind. Beispiele sind

   - Cray um 1979 mit einer Rechenleistung von ca. 1 GFlops
   - Cyber205 um 1980 z.B. in Karlsruhe, Bochum

2. Distributed memory SIMD

   bestehend aus Prozessorfeldern, wie z.B.

   - Thinking-machines CM 2 mit bis zu $64k$ Prozessoren um 1990 (eine CM 2 mit $16k$ Prozessoren war damals auch in Wuppertal vorhanden und kann heute noch am Haspel besichtigt werden).
   - Thinking-machines CM 5 um 1992 (ausgestellt vor der Mathe-Cafeteria)
   - Maspar um 1990

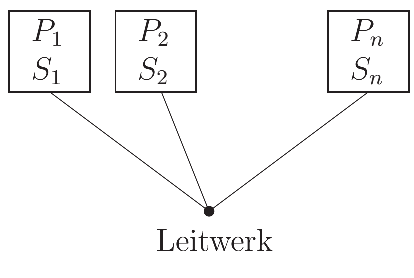


Abbildung 1.13: Schema für Distributed Memory SIMD

3. Shared Memory MIMD

   Im Wesentlichen sind dies Maschinen, die mit einigen (bis zu 100) mächtigen Prozessoren (meist Vektorprozessoren) ausgestattet sind und über ein Netzwerk mit dem shared memory verbunden werden.

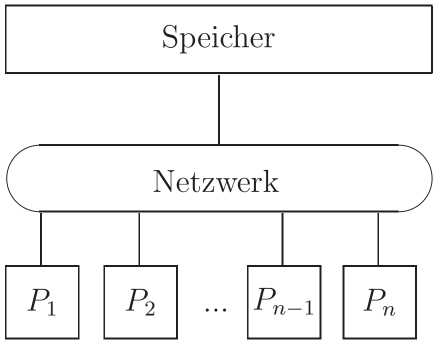


Abbildung 1.14: Schema für Shared Memory MIMD

4. Distributed Memory MIMD

   Vergleiche Abschnitt 1.3 (mit fester Topologie oder Netzwerk)

5. ccNUMA (cache coherant Non Uniform Memory Access)

   hybride Architektur, wobei die einzelnen Knoten, welche selbst Shared Memory MIMD sind, über ein Netzwerk verbunden sind.

6. Cluster (NOW: Networks of Workstations)

   besteht aus Standard-PCs oder Workstations verbunden über ein „Standard“-Netzwerk (Eigenbau).

   Die Cluster-Variante ist mittlerweile weit verbreitet (vgl. Top500-Liste), da sie nicht so kostenintensiv ist.

SM MIMD		
Compaq	GS	(1999, $p \leq 32$, 64 GFlop/s)
Cray	SV1EX	(2000, $p \leq 32$, 256 GFlop/s)
DM MIMD		
Cray	T3E	(1996, 3d-Torus, $p \leq 2000$, 3 TFlop/s)
Fujitsu	AP3000	(1996, 2d-Torus, $p \leq 1000$, 600 GFlop/s)
Fujitsu	VPP5000	(1999, voller Xbar, $p \leq 128$, 1 TFlop/s)
Hitachi	SR8000 G1	(2000, mehrdim. Xbar, $p \leq 512$, 7 TFlop/s)
Cray	X1	(2002, Xbar, $p = 16 - 64$, 812 GFlop/s)
NEC	SX-6	(2002, multi-stage Xbar, $p = 8 - 1024$, 8 TFlop/s)
C-DAC	PARAM Padma	(2003, switched Network, $p = 24$, 1 TFlop/s)
ccNUMA		
Compaq	Alpha Server	(1999, 4 Proz./Knoten, $p \leq 4000$, Baum, 1 TFlop/s)
SGI	Origin 3800	(2000, 4 Proz./Knoten, $p \leq 512$, Xbar/Hypercube, 717 GFlop/s)
HP	9000 supreme	(2000, 16 Proz./Knoten, $p \leq 64$, Xbar, 200 GFlop/s)
SUN	Fire 3800-15k	(2001, 4 Proz./Knoten, $p \leq 72$, Xbar, 200 GFlop/s)
IBM	eServer p 690	(2001, 16-32 Proz./Knoten, $p \leq 16384$, Ω Switch, 111 TFlop/s)

Beachte: Die meisten Top500-Top-Rechner sind nicht-kommerzielle Rechner und fallen deshalb nur bedingt in das obige Schema. Dies gilt insbesondere für die derzeit aktuellen Clusterrechner.

Links: www.top500.org
Aad van der Steen, Jack Dongarra: Overview of recent supercomputers, www.top500.org/ORSC/2003/overview.html

1.5 Zwei Supercomputer aus NRW

ALiCEnext (Universität Wuppertal)
Advanced Linux Cluster Engine - next generation

512 dual Opteron Prozessoren 1,7 GHz
$p = 1024$, Leistung $\leq$ 3.5 TFlops

Netzwerk: Hierarchisches Netzwerk über HP-Switches (Standard-Komponenten), Protokoll: GBit-Ethernet, spezielle Kommunikationssoftware Parastation IV

Vergleich Rechenleistung - Kommunikationsleistung:
Zeit für eine Gleitpunktoperation: $\dfrac{1}{2 \times 1.7GHz} \approx 0.3$ ns

Zeit zur Kommunikation von 64 bit (= 1 Gleitkommazahl):
$64 \times 10^{-9}s = 64$ ns

Latenz-Zeit zum Aufsetzen der Kommunikation:

$\sim 60\ \mu$ s = 60 000 ns
$\sim 20\ \mu$ s = 20 000 ns mit Parasation IV

JUMP (Jülich Multiprozessor)
IBM p690 Cluster

41 frames, jeder frame sind 32 Power 4x-Prozessoren, shared memory, 1.7 GHz
Verbindungsnetzwerk zwischen den frames: Ω-switch („confederation switch“), Bandbreite 1.4 GB/s, Latenzzeit 6.5 μs

Zeit zur Kommunikation von 64 bit (= 8 Byte): 8/1.4 ns $\approx$ 5 ns

1.6 Messgrößen

Gegeben sei im folgenden ein Parallelrechner mit p Prozessoren, eine zu lösende Aufgabe und ein zugehöriger Algorithmus (z.B. Gauß-Elimination für lineare Gleichungssysteme).

Wir wollen nun Größen einführen, die angeben, wie gut sich ein Problem parallelisieren lässt. Dazu bezeichne $t(p)$ die Ausführungszeit des gegebenen Algorithmus bei p Prozessoren.

1.6.1 Definition
Die Größe

$$S(p) := \frac{t(1)}{t(p)}$$

heißt **Speedup** des Algorithmus bezüglich sich selbst. Der Speedup gibt also den Zeitgewinn an, der sich gegenüber Ausführung auf einem Prozessor ergibt. Die Größe

$$E(p) := \frac{S(p)}{p}$$

heißt **Effizienz**. Je näher sie an 1 liegt desto besser parallelisiert der Algorithmus.

Mit t' bezeichnen wir die Ausführungszeit des schnellsten *bekannten* seriellen Algorithmus[1] zur Lösung der Aufgabe. Damit definieren wir

1.6.2 Definition
Die Größe

$$S'(p) := \frac{t'}{t(p)}$$

bezeichnet den **eigentlichen Speedup** und

$$E'(p) := \frac{S'(p)}{p}$$

die **eigentliche Effizienz**.

Diese Größen sind wichtig, weil sie das wirkliche Verhältnis widerspiegeln, das zwischen dem Lösen des Problems auf einem und mehreren Prozessoren besteht. Gut parallelisierende Algorithmen (also mit großem $E(p)$) können seriell ineffizient sein (also $t(1) > t'(p)$ und damit $E'(p)$ kleiner).

[1]d.i. der schnellste Algorithmus für einen Prozessor

1.6.3 Definition
Eine Operation sei die Ausführung von $+, -, *, /$. Dann heißt

$$r(p) := \frac{\text{Anzahl Operationen}}{t(p)}$$

die **Rechenleistung** und wird mit der Einheit Flops (**Fl**oatingpoint **Op**erationen pro **S**ekunde) gemessen.

Bei numerischen Anwendungen geht man davon aus, dass sämtliche Rechenzeit in die Operationen fließt, damit gilt dann

$$S(p) = \frac{r(p)}{r(1)},$$

wenn die Anzahl der Operationen unabhängig von p ist.

Verfeinerung: Meist sind die obigen Größen nicht nur für eine feste Problemgröße interessant (man kann sich leicht vorstellen, dass man keine Gewinne mehr erhält, wenn man p zu groß im Vergleich zur Problemgröße n wählt. Sei also n im folgenden eine charakteristische Größe für das zu lösenden Problem (zum Beispiel die Dimension der Matrix bei einem quadratischen linearen Gleichungssystem). Um die Abhängigkeit von n zu dokumentieren, notieren wir deshalb auch

$$r(p,n),\ t(p,n),\ S(p,n).$$

1.6.4 Definition
Für einen gegebenen Rechner mit p Prozessor und eine Aufgabe mit variabler Größe n bezeichnet

$$Sc(p) := \sup_{n \in \mathbb{N}} S(p,n)$$

den **Scaleup**. Tatsächlich muss man hierbei sich auf die n beschränken, für welche der Speicher zum Lösen der Aufgabe ausreicht.

Man erhält die Beziehung

$$S'(p,n) \leq S(p,n) \leq Sc(p) \leq p.$$

Abbildung 1.15 gibt den typischen Verlauf dieser Größen an.

1.6.5 Definition
Ein paralleler Algorithmus zur Lösung einer Aufgabe **skaliert**, falls

$$\liminf_{p \to \infty} \frac{Sc(p)}{p} > 0$$

ist, d.h. der Quotient nach unten durch eine positive Zahl beschränkt bleibt.

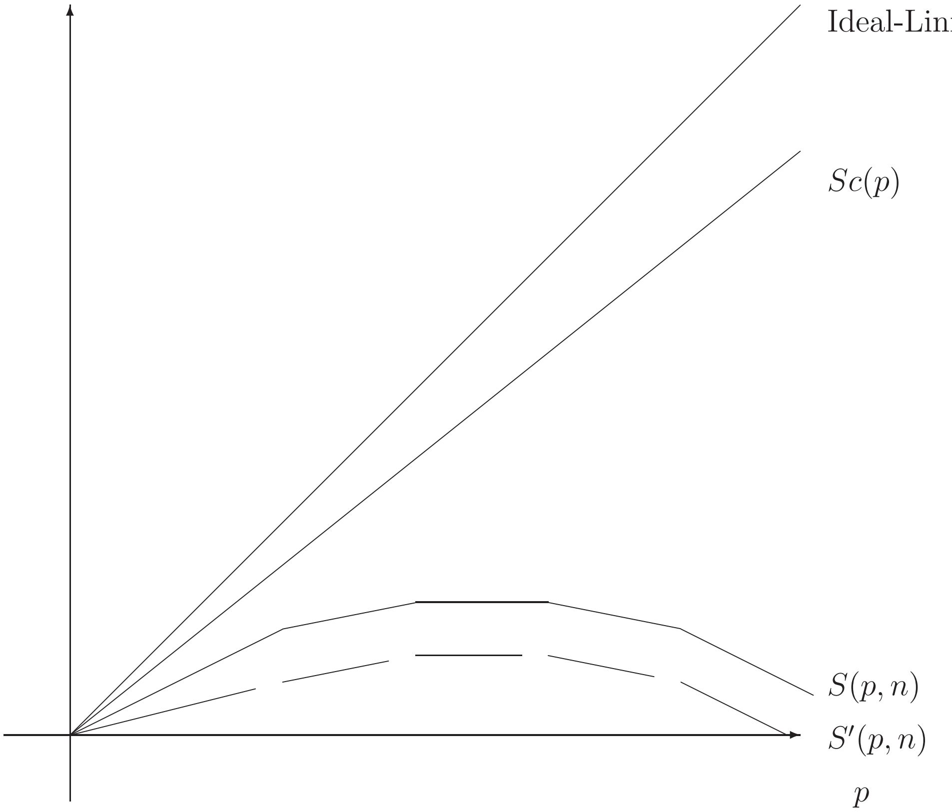


Abbildung 1.15: Scaleup, Speedup, eigentlicher Speedup

1.6.6 Satz (Amdahls Gesetz)
Es sei $\alpha \in [0,1]$ der Anteil der seriellen Gesamtrechenzeit des Teils eines Algorithmus, der nicht parallelisiert werden kann. Dann gilt für alle $p > 1$

$$S(p) \leq \frac{1}{\alpha}.$$

Beweis: Es sei $t(1)$ die serielle Zeit, dann gilt

$$t(p) \geq \alpha \cdot t(1).$$

Damit ergibt sich

$$S(p) = \frac{t(1)}{t(p)} \leq \frac{1}{\alpha}.$$

□

Beispiel: Ist $\alpha = 0,01 = 10^{-2}$, so ist $S(p) \leq 100$ für alle p.

Es lohnt sich also wohl kaum, mehr als 100 Prozessoren zur Problemlösung einzusetzen. (Je nach Problem bedeutet eine noch größere Zahl dann sogar eine Verringerung des Speedup.)

1.7 Kommunikationsanweisungen

Die meisten der vorhandenen Bibliotheken für parallele Algorithmen basieren auf dem „Message-Passing-Programmiermodell“. Parallel nebeneinander laufende Tasks tauschen Informationen durch `send`- und `receive`-Direktiven aus.

Um dies hinterher unabhängig von der gewählten Sprache darzustellen, führen wir Pseudocode-Bezeichnungen ein.

send(D) to P_i:	sende Datenmenge D an Prozessor P_i
receive(D) from P_i:	empfange Datenmenge D von Prozessor P_i
broadcast(D):	sende D an alle Prozessoren
fan-in$\leftarrow s$:	siehe Abschnitt 1.8

Kommunikationsanweisungen können prinzipiell *blockierend* oder *nicht-blockierend* implementiert werden. Bei blockierender Kommunikaton steht ein sendendes oder empfangendes Programm so lange still, bis der Datenaustausch beendet ist.

Im Gegensatz dazu werden bei *nicht-blockierenden* Kommunikationsanweisungen Sender und Empfänger lediglich initialisiert. Durch Tests kann zu einem späteren Zeitpunkt abgefragt werden, ob Daten versendet/empfangen wurden.

Wie die Kommunikationsanweisungen möglichst effizient ausgeführt werden, hängt von der jeweils verwendeten Architektur ab. Es kommt bei den `send`- und `receive`-Anweisungen besonders darauf an, wie nahe beieinander die Prozessoren liegen, welche die Daten austauschen.

1.8 Fan-in

Wir betrachten zunächst das Beispiel der Addition.

Gegeben sind die Zahlen $a_1, ..., a_{2^N} \in \mathbb{R}$.
Gesucht ist die Summe $s = \sum_{j=1}^{2^N} a_j$.

1.8.1 Algorithmus (Serielle Addition)
Die gewöhnliche serielle Addition, d.h. Addition auf einem Prozessor kann folgendermaßen erfolgen:

$s^{(0)}$:=0
for $i = 1$ to 2^N **do**
  $s^{(i)} = s^{(i-1)} + a_i$
end for
$\{(s = s^{(2^N)})\}$

Hierbei werden $2^N - 1$ Additionen hintereinander ausgeführt. Durch ungünstige Anordnung und Vorzeichen kann ein evtl. sehr großer numerischer Fehler entstehen („**Auslöschung**").

1.8.2 Algorithmus (Fan-in Summation)
Bei mehr als einem Prozessor kann man die Rechenlast möglichst gleichmäßig verteilen, indem man jeweils parallel Paare von Zahlen addiert.

$a_i^{(0)} = a_i,\ i = 1...2^N$ {Initialisierung}
for $k = 1$ to N **do**
  for $i = 1$ to 2^{N-k} **do**
    $a_i^{(k)} = a_{2i}^{(k-1)} + a_{2i-1}^{(k-1)}$
  end for
end for
$\{(s = a_1^{(N)})\}$

Die Schleife über i kann parallel durchgeführt werden. Abbildung 1.16 veranschaulicht das Vorgehen.

Für die Zahl der Additionen ergibt sich nach Algorithmus 1.8.2

$$\sum_{k=1}^{N} 2^{N-k} = \sum_{j=0}^{N-1} 2^j = 2^N - 1.$$

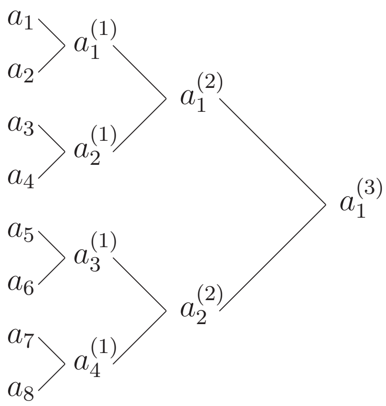


Abbildung 1.16: Schema für $N = 3$

Es bleibt also bei $2^N - 1$ Additionen, die auch bei der seriellen Addition nötig sind. Wird die Schleife über i mit $p = 2^N$ Prozessoren parallel ausgeführt, so wird ein Prozessor insgesamt N Additionen durchführen. Damit ergibt sich für den Speedup

$$S(p) = \frac{2^N - 1}{N} \approx \frac{2^N}{N} = \frac{p}{\log(p)} \Rightarrow E(p) = \frac{1}{\log(p).}$$

Wir formulieren jetzt einen parallen Algorithmus für die Fan-in-Summation im Detail. Dazu sei $p = 2^N$, wir bezeichnen die Prozessoren mit $P_1, ..., P_p$. Wähle für festes k (k sei dabei die Schleifenvariable des fan-in) unter den Prozessoren P_j verschiedene Prozessoren $P(k,i)$, $i = 1...2^{N-k}$, derart, dass $P(k,i)$ das Teilergebnis $a_i^{(k)}$ berechnet. Dies ergibt folgenden Algorithmus

1.8.3 Algorithmus (fan-in mit $p = 2^N$ Prozessoren)
Require: $a = a_i$ in Prozessor $P(0, i)$
  for $k = 0$ to $N - 1$ **do**
    for $i = 1$ to 2^{N-k} **do**
      if me=$P(k, i)$ **then**
        send(a) to $P(k + 1, \lceil \frac{i}{2} \rceil)$
      end if
    end for
    for $i = 1$ to 2^{N-k} **do**
      if me=$P(k + 1, i)$ **then**
        receive(b) from $P(k, 2i - 1)$
        receive(c) from $P(k, 2i)$
        $a = b + c$
      end if
    end for
  end for

Achtung:

- Dies ist der Pseudocode für einen Prozessor (SPMD-Modell).
- Eine Modifikation ist nötig, falls $n = q \cdot 2^N$, $q, N \in \mathbb{N}$, d.h. eine Addition auf weniger Prozessoren als Zahlen ausgeführt werden soll. Dann initialisiert man nämlich

$$a = \sum_{j=1}^{q} a_{(i-1)q+j} \text{ in } P(0, i).$$

1.8.4 Definition
Ein **deadlock** in einem parallelen Algorithmus liegt vor, wenn ein Prozessor auf den Empfang von Daten wartet, die niemals gesendet werden.

Beachte: Ein `send`-Befehl ist in der Regel auch dann erst beendet, wenn das zugehörige `receive` ausgeführt wurde, andernfalls könnte es zum Überschreiben von Daten kommen. Zusätzlich kommt noch die Aufgabe der Synchronisation hinzu!

Lassen wir nun das folgende Programm auf zwei Prozessoren ablaufen

```
if me=P_1 then
  send(a) to P_2
  receive(b) from P_2
end if
if me=P_2 then
  send(a) to P_1
  receive(b) from P_1
end if
```

so halten beide Programme vor dem `receive`-Befehl; denn P_1, P_2 können ihr `receive` erst nach Beenden des jeweiligen `send`-Befehls abarbeiten. Jetzt „hängen" aber beide Prozessoren im Sende-Zustand fest, weil sie gegenseitig auf das `receive` warten müssen!

Algorithmus 1.8.3 ist damit aber nur dann deadlock-frei, wenn alle $P(k+1, i)$ für $i = 1, ..., 2^{N-(k+1)}$ verschieden von den $P(k, i),\ i = 1, ..., 2^{N-k}$ sind, denn nur dann enden auch die `send`-Befehle der Prozessoren der Stufe $k + 1$.

Dies ist aber für $k = 0$ prinzipiell nicht möglich. Wir könnten den Pseudocode jetzt so modifizieren, dass geeignete Prozessoren zuerst die `receive`s ausführen. Darauf verzichten wir aber und wählen stattdessen eine geschickte Zuordnung $P_j \leftrightarrow P(k, i)$.

1.8.5 Algorithmus (Verbesserung)
Spare Kommunikation durch geeignete Prozessorwahl:
Die Prozessoren seien mit $P_0, ..., P_{2^N-1}$ nummeriert, setze dann

$$P(k, i) = P_{2^k(i-1)}.$$

Es ergibt sich

$$P(k+1, \lceil \frac{i}{2} \rceil) = P_{2^{k+1}(\lceil \frac{i}{2} \rceil - 1)} = \begin{cases} P_{2^k(i-2)} & i \text{ gerade} \\ P_{2^k(i-1)} & i \text{ ungerade.} \end{cases}$$

Also werden ein `send`- und ein `receive`-Befehl unnötig, weil bei dieser Belegung ein Prozessor an sich selber schicken würde. (Dies löst die deadlocks in Algorithmus 1.8.3 auf.)

Require: $a = \sum_{j=1}^{q} a_{(i-1)q+j}$ in Prozessor $P(0,i)$
  for $k = 0$ to $N-1$ **do**
    for $i = 1$ to 2^{N-k} **do**
      if me=$P_{2^k(i-1)}$ **then**
        if even(i) **then**
          send(a) to $P(k+1, \frac{i}{2}) = P_{2^k(i-2)}$
        end if
      end if
    end for
    for $i = 1$ to 2^{N-k-1} **do**
      if me=$P_{2^{k+1}(i-1)}$ **then**
        receive(c) from $P_{2^k(2i-1)}$
        $a = a + c$
      end if
    end for
  end for

1.8.6 Satz
Es sei die Prozessorzahl $p = 2^N$ und $n = q \cdot 2^N$, $q \in \mathbb{N}$, desweiteren seien $a_i \in \mathbb{R}$, $i = 1, ..., n$. Für den Algorithmus 1.7.5 zur Berechnung der Summe $\sum_{j=1}^{n} a_j$ gilt *bei Vernachlässigung von Kommunikationszeiten*

$$S = \frac{q \cdot 2^N - 1}{(q-1)+N}, \quad E = \frac{q - \frac{1}{2^N}}{(q-1)+N}.$$

Beweis: Es sei t_a die Zeit für eine Addition, dann ergibt sich für die serielle Zeit

$$t_1 = (n-1) \cdot t_a.$$

Für die parallele Ausführung werden die Zahlen in p Folgen der Länge q aufgeteilt. In Algorithmus 1.8.5 benötigt die Initalisierung von a eine Zeit von $(q-1) \cdot t_a$, hinzu kommen für jeden Prozessor maximal N weitere Additionen, also

$$t(p) = (q-1) \cdot t_a + N \cdot t_a.$$

Also

$$\begin{aligned} S(p) &= \frac{t(1)}{t(p)} = \frac{(n-1) \cdot t_a}{(q-1) \cdot t_a + N \cdot t_a} \\ &= \frac{n-1}{(q-1)+N} = \frac{q \cdot 2^N - 1}{(q-1)+N} \end{aligned}$$

$$E(p) \quad = \quad \frac{S(p)}{p} = \frac{S(p)}{2^N} = \frac{q - \frac{1}{2^N}}{(q-1)+N}$$

□

Beispiel: Ist z.B. $q = 1$, so ergibt sich damit

$$E = \frac{1 - \frac{1}{2^N}}{N} \cong \frac{1}{N} = \frac{1}{\log(p)}.$$

Betrachten wir den Fall $q \to \infty$, so ergibt sich $E \to 1$, was zwar wünschenswert aber eher unrealistisch ist.

Praktisch ist nämlich $q \leq c$, wobei c eine feste Konstante ist, die von der Speichergröße eines Prozessors abhängt. Für alle $q \leq c$ ist aber stets $\lim\limits_{N\to\infty} E(q, N) = 0$, d.h. der Fan-in-Algortihmus skaliert *nicht*!

Bemerkung: Die Kommuniaktionszeiten zu vernachlässigen ist eine sehr grobe Vereinfachung. Sei t_k die Zeit für ein `send`/`receive`-Paar; wir setzen $t_k = \alpha t_a$, wobei t_a wiederum die Zeit für eine Addition sei. Typisch ist hier der Fall $\alpha \gg 1$. Es ergibt sich

$$T_p = (q-1)t_0 + \underbrace{Nt_0 + Nt_k}_{=Nt_0(1+\alpha)}$$

Also ist der Speedup wegen den Kommunikationszeiten noch viel kleiner!

Zahlenbeispiel: Sei $p = 2^N, \alpha = 10, n = 2^M$ mit $M \geq N$, dann folgt

$$t_p(N) = 2^{M-N-1}t_a + 11Nt_a.$$

Mittels einfacher Differentialrechnung erhält man, dass dieser Ausdruck bezüglich N minimal wird, wenn

$$N = M - 3,988... \cong N - 4$$

ist, d.h. eine Addition von 1.024 Zahlen erfolgt also am besten mit $10^{10-4} = 64$ Prozessoren.

1.8.7 Beispiel (im Hypercube)

Es seien die Prozessoren in einem Hypercube durch $P_0, ..., P_{2^k-1}$ nummeriert. Wähle wie in Algorithmus 1.7.5 die Prozessoren gemäß $P(k, i) = P_{2^k(i-1)}$, dann ergibt sich

$$\begin{aligned} P(k+1, i) = P(k, 2i-1) &= P_{2^{k+1}(i-1)} \\ P(k, 2i) = P_{2^k(2i-1)} \quad &= P_{2^{k+1}(i-1)+2^k}. \end{aligned}$$

Also unterscheiden sich $P(k+1, i)$ und $P(k, 2i)$ um genau ein Bit, d.h. sind im Hypercube direkt miteinander verbunden. Die Kommunikation beim Fan-in in Algorithmus 1.8.5 erfolgt also nur zwischen direkt verbundenen Prozessoren.

1.8.8 Bemerkung
Das Fan-in Prinzip ist allgemeiner formulierbar als für Addition. Es kann für jeden Ausdruck der Form

$$a_1 \circ a_2 \circ \ldots \circ a_n \quad (n = 2^N)$$

angewendet werden, wenn nur $\circ$ eine assoziative[2] Operation ist.

Andere Beispiele als die bisher betrachtete Summation sind:

$$\max_{i=1,\ldots,n} \{a_i\} \qquad (\text{d.h. } a \circ b = \max\{a, b\})$$

$$\min_{i=1,\ldots,n} \{a_i\} \qquad (\text{d.h. } a \circ b = \min\{a, b\})$$

$$\prod_{i=1}^{n} a_i \qquad (\text{d.h. } a \circ b = a \cdot b,\ \text{wobei } a \in \mathbb{R}^{n\times m}, b \in \mathbb{R}^{m\times q})$$

$$\sum_{i=1}^{n} a_i \qquad (\text{d.h. } a \circ b = a + b,\ \text{wobei } a, b \in \mathbb{R}^{n\times m})$$

Weiter Beispiele sind auch kgV oder ggT etc.

1.8.9 Definition
Die Notation

$$\textbf{fan-in}(a_i{,}s{,}P_i :\ i \in I)$$

bedeutet, dass die Prozessoren $P_i,\ i \in I$ ein fan-in zur Berechnung der Summe

$$s = \sum_{i\in I}^{n} a_i$$

durchführen.

Beachte: Es wird *nicht* angegeben, in welchem Prozessor das Endergebnis s vorliegt!

[2] d.h. für beliebige a, b, c gilt: $a \circ (b \circ c) = (a \circ b) \circ c)$. Damit ist die Aufteilung in „Pakete“ überhaupt erst möglich!

Kapitel 2

Direkte Löser für lineare Gleichungssysteme

2.1 Matrizenmultiplikation

Matrizenmultiplikation ist tatsächlich nicht nur eine (einfache) mathematische Operation, vielmehr kommt sie auch in Anwendungen häufiger vor.

2.1.1 Beispiel
Zur Berechnung von stochastischen Modellen werden häufig Übergangsmatrizen aufgestellt. Gibt man einen Startvektor $s^{(1)}$ (=Anfangsverteilung) vor, so erhält man durch Multiplikation mit der Übergangsmatrix A einen Vektor, der eine Folgeverteilung beschreibt. Analog erhält man für die n-te Verteilung

$$s^{(n)} = As^{(n-1)} = A(As^{(n-2)}) = ... = \underbrace{A(A(...A(A}_{n-\text{mal}}\, s^{(1)})...).$$

Die Übergangsmatrix in die n-te Verteilung ist also die Matrix

$$A^n.$$

Solche Modelle werden z.B. bei Telekommunikationsunternehmen eingesetzt.

Gegeben: Zwei Matrizen $A = (a_{ij}) \in \mathbb{R}^{n\times m}$ und $B = (b_{ij}) \in \mathbb{R}^{m\times q}$

Gesucht: Eine Matrix $C = (c_{ij}) = A \cdot B \in \mathbb{R}^{n\times q}$ mit

$$c_{ij} = \sum_{k=1}^{m} a_{ik}b_{kj}, \quad i = 1, ..., n, \;\; j = 1, ..., q.$$

2.1.2 Definition
Das Mengensystem $I_r,\ r = 1, ..., R$ heißt **Partition** der Menge M, falls

1. $I_r \neq \emptyset$ für alle r
2. $I_r \cap I_s = \emptyset$, $r \neq s$
3. $\bigcup\limits_{r=1}^{R} I_r = M$

Mittels Partitionen wollen wir nun die Matrizen in (möglichst gleich große) Blöcke unterteilen und diese Blöcke miteinander verrechnen. Sei dazu

$$\begin{array}{lll} I_r,\ r = 1, ..., R & \text{eine Partition von} & \{1, ..., n\} \\ K_s,\ s = 1, ..., S & \text{eine Partition von} & \{1, ..., m\} \\ J_t,\ t = 1, ..., T & \text{eine Partition von} & \{1, ..., q\} \end{array}$$

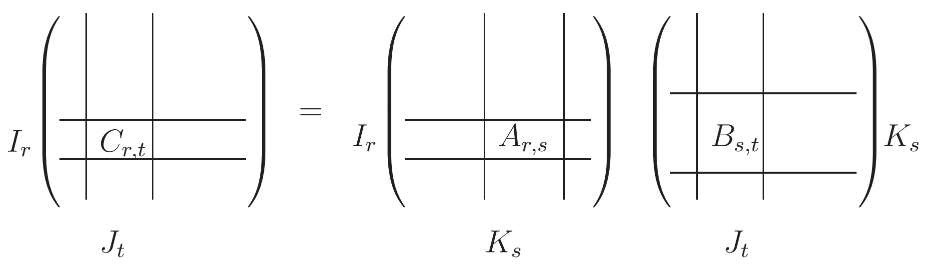
gewählt und es sei weiter

$$\begin{aligned} A_{rs} &:= (a_{ik})_{\substack{i \in I_r \\ k \in K_s}} \\ B_{st} &:= (b_{kj})_{\substack{k \in K_s \\ j \in J_t}} \\ C_{rt} &:= (c_{ij})_{\substack{i \in I_r \\ j \in J_t}}. \end{aligned}$$

Dann gilt

$$C_{rt} = \sum_{s=1}^{S} A_{rs} B_{st}.$$

Veranschaulichen kann man das folgendermaßen:



Der Parallelrechner besitze nun $p = R \cdot S \cdot T$ Prozessoren, die wir mit $P(r, s, t), r = 1, ..., n,\ s = 1, ..., S,\ t = 1, ..., T$ bezeichnen.

Prozessor $P(r, s, t)$ berechnet das Produkt $A_{rs} \cdot B_{st}$ und dann über Fan-in mit $P(r, \sigma, t), \sigma = 1, ..., S$ den Block C_{rt} von C.

2.1.3 Algorithmus

Der Algorithmus für $P(r,s,t)$ ergibt sich als

berechne $\widetilde{C}_{rt} = A_{rs} \cdot B_{st}$ **fan-in**($\widetilde{C}_{rt}$,C_{rt}, $P(r,\sigma,t)$: $\sigma = 1,...,S$)

Die Rechenlast im Algorithmus 2.1.3 ist damit gleichmäßig verteilt, wenn das Produkt

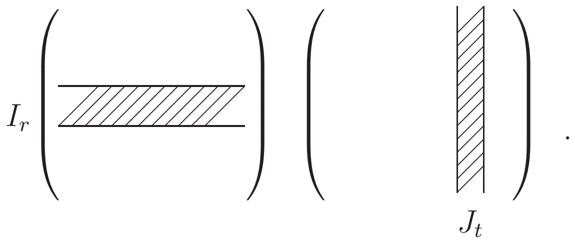
$$|I_r| \cdot |K_s| \cdot |J_t|$$

möglichst unabhängig von r, s und t ist; der Rechenaufwand zur Bestimmung von $\widetilde{C}_{r,t}$ ist rund $|I_r| \cdot |K_s| \cdot |J_t|$. Erreichbar ist

$$\begin{aligned} |\ |I_r| - |I_{r'}|\ | \leq &\ \ 1 \text{ für } r, r' = 1,...,R, \\ |\ |K_s| - |K_{s'}|\ | \leq &\ \ 1 \text{ für } s, s' = 1,...,S, \\ |\ |J_t| - |J_{t'}|\ | \leq &\ \ 1 \text{ für } t, t' = 1,...,T. \end{aligned}$$

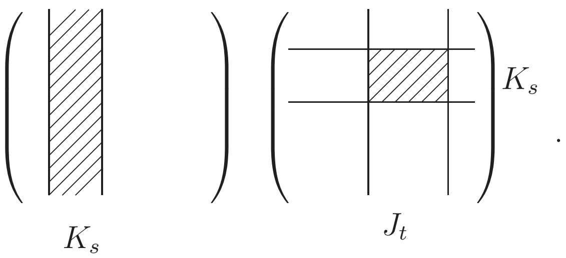
2.1.4 Bemerkung (Spezialfälle)

1. $S = 1$, also $K_1 = \{1, ..., m\}$



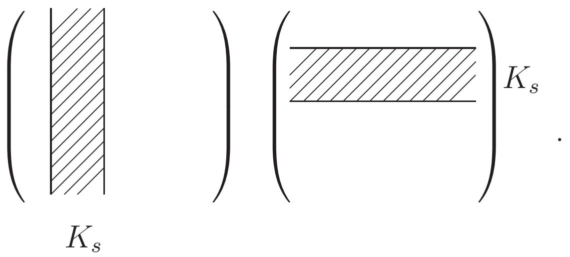
Es entfällt der fan-in, d.h. der Algorithmus ist kommunikationsfrei. Wir setzen aber voraus, dass die Matrix A hierbei in Blöcken von Zeilen und die Matrix B in Blöcken von Spalten auf die Prozessoren abgespeichert sei.

2. $R = 1$, also $I_1 = \{1, ..., n\}$. Also enthalten die Prozessoren volle Spalten von A und einen Teilblock von B,



Es sind T fan-ins sind nötig, an denen jeweils S Prozessoren beteiligt sind.

3. $R = 1$ und $T = 1$,



Jeder Prozessor berechnet ein Teilergebnis für jeden Eintrag der Ergebnismatrix. Es ist ein fan-in (mit Matrizen der Dimension $n \times q$) nötig, an dem alle Prozessoren beteiligt sind. Diesmal muss A in Blöcken von Spalten und B in Blöcken von Zeilen auf die Prozessoren verteilt sein.

4. $R = n, S = m$ und $T = q$

$$\begin{array}{cccc} r \rightarrow & \begin{pmatrix} \quad & \bullet & \quad \\ & & \\ & & \end{pmatrix} & \begin{pmatrix} \quad & & \quad \\ & & \\ \bullet & & \end{pmatrix} & \leftarrow s \\ & \begin{matrix} \uparrow \\ s \end{matrix} & \begin{matrix} \uparrow \\ t \end{matrix} & \end{array}$$

Es wird vorausgesetzt, dass $n \cdot m \cdot q$ Prozessoren vorhanden sind. Jeder Prozessor berechnet *ein* Produkt. Danach werden $n \cdot q$ fan-ins mit je m Prozessoren nötig. Jeder fan-in ergibt dann genau ein Element von C.

Diese Möglichkeit ist jedoch „illusorisch“, da die Prozessorzahl hierfür *enorm* wäre, ebenso die Kommunikation.

Bemerkung: Algorithmus 2.1.3 verschwendet Speicher! Auf einem seriellen Rechner benötigen die Matrizen A und B genau $n \cdot m + m \cdot q$ Speicherplätze für den jeweiligen Eintragstyp. In Algorithmus 2.1.2 hingegen ergibt sich:

Der Prozessor $P(r, s, t)$ enthält A_{rs}, B_{rt}, das bedeutet einen Speicherbedarf von

$$|I_r| \cdot |K_s| + |K_s| \cdot |J_t|.$$

Damit ergibt sich ein gesamter Speicherbedarf

$$\sum_{r=1}^{R} \sum_{s=1}^{S} \sum_{t=1}^{T} |I_r| \cdot |K_s| + |K_s| \cdot |J_t| \quad = \quad T \cdot n \cdot m + R \cdot m \cdot q.$$

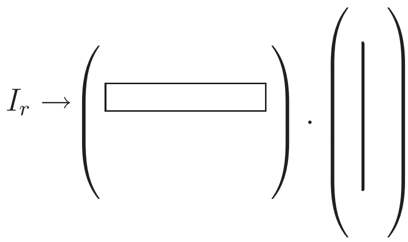
Also: Nur im Spezialfall 3. $(T = 1, R = 1)$ ergibt sich keine Speicherverschwendung!

2.2 Matrix-Vektor-Multiplikation

Die Matrix-Vektor-Multiplikation ist ein Spezialfall der in Abschnitt 2.1 behandelten Matrix-Matrix-Multiplikation mit $q = 1$, also $T = 1$.

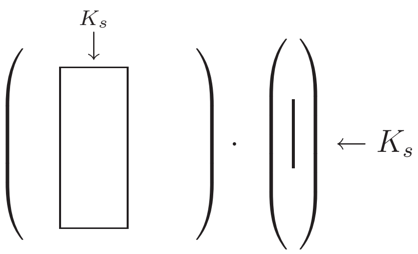
Insbesondere ergeben sich die beiden Sonderfälle:

1. $S = 1$:



   Der Vektor ist auf jedem Prozessor vorhanden. Es fällt keine Kommunikation an.

2. $R = 1$:



   Die Multiplikation erfolgt spaltenüberstreichend, es wird nur 1 fan-in benötigt und kein Speicher verschwendet.

2.3 Bandmatrizen

2.3.1 Definition
Eine Matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ heißt **Bandmatrix** mit halber Bandbreite b, falls gilt:

$$a_{ij} = 0 \text{ für } |i-j| > b.$$

Bandmatrizen haben damit also folgendes Aussehen

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,b+1} & & & & \\ \vdots & \ddots & & \ddots & & 0 & \\ a_{b+1,1} & & \ddots & & \ddots & & \\ & \ddots & & \ddots & & \ddots & \\ & & \ddots & & \ddots & & a_{n-b,n} \\ & 0 & & \ddots & & \ddots & \vdots \\ & & & & a_{n,n-b} & \cdots & a_{n,n} \end{pmatrix}.$$

Bei der sogenannten diagonalen Abspeicherung verwendet man folgendes Format

$$\begin{array}{cc} \widetilde{a}_{1,-b} & \cdots \\ & \ddots \end{array} \begin{pmatrix} & \widetilde{a}_{1,0} & \cdots & \widetilde{a}_{1,b} & & & \\ & & \ddots & & \ddots & & \\ \ddots & & & \ddots & & \ddots & \\ & \ddots & & & \ddots & & \ddots \\ & & \ddots & & & \ddots & \\ & & & \widetilde{a}_{n,-b} & \cdots & \widetilde{a}_{n,0} & \end{pmatrix} \begin{array}{cc} \ddots & \\ \cdots & \widetilde{a}_{n,b} \end{array}$$

d.h. die Zuordnung

$$\begin{aligned} \widetilde{a}_{i,j} &= a_{i,i+j} && \text{für } i = 1,...,n,\ j = -b,...,b,\ i+j \in \{1,...,n\} \\ \widetilde{a}_{i,j} &= 0 && \text{für } i = 1,...,n,\ j = -b,...,b,\ i+j \notin \{1,...,n\}. \end{aligned}$$

Damit besitzt das Feld $\widetilde{A} = (\widetilde{a}_{i,j})$ die Dimension $n \times (2b+1)$. Dadurch wird für $b << n$ sehr viel Speicherplatz eingespart, da die meisten Nullen aus A nicht mehr mit abgespeichert werden. Wir bezeichnen die Diagonalen von A als $s_j := \widetilde{a}_{*,j}$, das sind jetzt Spalten in $\widetilde{A}$.

2.3.2 Satz
Seien $A, B \in \mathbb{R}^{n\times n}$ Bandmatrizen mit der halben Bandbreite b_A bzw. b_B. Dann besitzt ihr Produkt $C = A \cdot B$ die halbe Bandbreite $b_C = b_A + b_B$. Sind $\widetilde{A}, \widetilde{B}$ und $\widetilde{C}$ die zugehörigen Felder bei diagonaler Abspeicherung, so gilt

$$\widetilde{c}_{i,j} = \sum_{k=\max\{-b_A, j-b_B\}}^{\min\{-b_A, j+b_B\}} \widetilde{a}_{i,k}\widetilde{b}_{k,j}, \quad i = 1, ..., n, \; j = -b, ..., b.$$

Beweis: Es gilt

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j} = \sum_{k=\max\{i-b_A, j-b_B, 1\}}^{\min\{i+b_A, j+b_B, n\}} a_{i,k}b_{k,j}$$

$$\begin{aligned}&(\text{da } a_{i,k} = 0 \text{ für } k < i - b_A, k > i + b_A,\\ &\quad b_{k,j} = 0 \text{ für } k < j - b_B, k > i + b_B).\end{aligned}$$

Also gilt

$$\begin{aligned} c_{i,j} = 0 \Longleftarrow \quad & \min\{i + b_A, j + b_B, n\} < \max\{i - b_A, j - b_B, 1\} \\ \Longleftarrow \quad & j + b_B < i - b_A \text{ oder } i + b_A < j - b_B \\ \Longleftrightarrow \quad & b_A + b_B < i - j \text{ oder } b_A + b_B < j - i \\ \Longleftrightarrow \quad & b_A + b_B < |i - j|. \end{aligned}$$

Also besitzt C die halbe Bandbreite $b_A + b_B$. Außerdem folgt

$$c_{i,j} = \sum_{k=\max\{i-b_A, j-b_B\}}^{\min\{i+b_A, j+b_B\}} a_{i,k}b_{k,j} = \sum_{k=\max\{i-b_A, j-b_B\}}^{\min\{i+b_A, j+b_B\}} \widetilde{a}_{i,k-i}\widetilde{b}_{k,i+j-k}.$$

Es folgt also

$$\begin{aligned} \widetilde{c}_{i,j} = c_{i,i+j} \quad &= \sum_{k=\max\{i-b_A, i+j-b_B\}}^{\min\{i+b_A, i+j+b_B\}} \widetilde{a}_{i,k-i}\widetilde{b}_{k,i+j-k} \\ &\overset{k-i\to k}{=} \sum_{k=\max\{-b_A, j-b_B\}}^{\min\{b_A, j+b_B\}} \widetilde{a}_{i,k}\widetilde{b}_{k+i,j-k}. \end{aligned}$$

□

Analog zu Algorithmus 2.1.3 ergeben sich folgende parallele Algorithmen:
Es sei

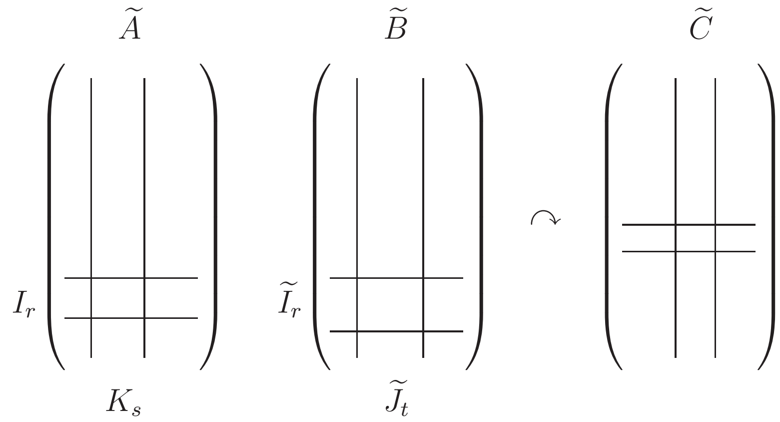
$$I_r, \; r = 1, ..., R \text{ eine Partition von} \qquad \{1, ..., n\}$$

$$
\begin{array}{ll}
J_t,\ t = 1, ..., T \text{ eine Partition von} & \{-b_A - b_B, ..., b_A + b_B\} \\
K_s,\ s = 1, ..., S \text{ eine Partition von} & \{-b_A, ..., b_A\}.
\end{array}
$$

Der Prozessor $P(r, s, t)$ berechnet

$$
\widetilde{c}_{i,j} \quad = \sum_{k=\max\{-b_A, j-b_B\} k\in K_s}^{\min\{b_A, j+b_B\}} \widetilde{a}_{i,k} \widetilde{b}_{k+i,j-k} \quad i \in I_r,\ j \in J_t
$$

und führt danach ein fan-in aus. Welche Daten werden benötigt?



Von $\widetilde{B}$ wird im allgemeinen der vergrößerte Block $(\widetilde{b}_{k,j})_{k\in\tilde{I}_r, j\in\tilde{J}_t}$ benötigt mit

$$
\begin{aligned}
\tilde{I}_r &= \{i + k,\ i \in I_r,\ k \in K_s\}, \\
\tilde{J}_t &= \{j - k,\ i \in I_r,\ k \in K_s\}.
\end{aligned}
$$

2.4 LU-Zerlegung

Gegeben ist eine Matrix $A \in \mathbb{R}^{n\times n}$, ein Vektor $b \in \mathbb{R}^n$. Zusätzlich sei A regulär. Wir betrachten das Gleichungssystem

$$Ax = b. \tag{2.4.1}$$

Gesucht ist eine Zerlegung $A = LU$ mit

$$L = \begin{pmatrix} 1 & & & & \\ \ell_{2,1} & \ddots & & 0 & \\ \vdots & \ddots & \ddots & & \\ \vdots & & \ddots & \ddots & \\ \ell_{n,1} & \dots & \dots & \ell_{n,n-1} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{1,1} & \dots & \dots & \dots & u_{1,n} \\ & \ddots & & & \vdots \\ & & \ddots & & \vdots \\ & 0 & & \ddots & \vdots \\ & & & & u_{n,n} \end{pmatrix}.$$

Damit ergibt sich dann die Lösung von 2.4.1 über $Ly = b$, $Ux = y$.
Beachte: Die Zerlegung $A = LU$ existiert nur in Spezialfällen („ohne Pivotsuche"). Wir betrachten zunächst nur diesen Spezialfall.

2.4.1 Algorithmus (Gauß-Elimination ohne Pivotsuche)

Setze $A^{(1)} = (a^{(1)}_{i,j}) := A$
for $k = 1, ..., n-1$ **do**
  for $i = k+1, ..., n$ **do**
    $\ell_{i,k} = a^{(k)}_{i,k} / a^{(k)}_{k,k}, \quad i = k+1, ..., n$
  end for
  for $j = k+1, ..., n$ **do**
    for $i = k+1, ..., n$ **do**
$$a^{(k+1)}_{i,j} = \begin{cases} a^{(k)}_{i,j} - \ell_{i,k} a^{(k)}_{k,j} & i \in \{k+1, ..., n\},\ j \in \{k, ..., n\} \\ \\ a^{(k)}_{i,j} & \text{sonst} \end{cases}$$
    end for
  end for
end for

Beachte: Es gilt
$$a^{(k+1)}_{i,k} = 0, \quad i = k+1, ..., n.$$

Mit

$$
L^{(k)} = \begin{pmatrix}
1 & & & & & & \\
 & \ddots & & & & & \\
 & & 1 & & & & \\
 & & -\ell_{k+1,k} & 1 & & & \\
 & & & & 1 & & \\
 & & \vdots & & & \ddots & \\
 & & -\ell_{n,k} & & & & 1
\end{pmatrix}
$$

(leere Einträge bedeuten hierbei Nullen) gilt:

$$A^{(k+1)} = L^{(k)} \cdot A^{(k)}.$$

Also ergibt sich

$$A^{(n)} = L^{(n-1)} \cdots L^{(1)} \cdot A,$$

und damit

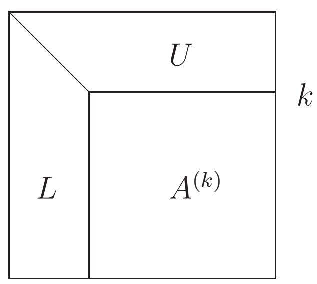
$$A = \underbrace{(L^{(1)})^{-1} \cdots (L^{(n-1)})^{-1}}_{=:L} \cdot \underbrace{A^{(n)}}_{=:U} .$$

Folgender Pseudocode setzt die Gauß-Elimination um.

2.4.2 Algorithmus (Serielle Gauß-Elimination, „kji"-Form)

```
for k = 1 to n − 1 do
  for s = k + 1 to n do
    ℓ_{s,k} := a_{s,k}/a_{k,k}
  end for
  for j = k + 1 to n do
    for i = k + 1 to n do
      a_{i,j} = a_{i,j} − ℓ_{i,k} a_{k,j}
    end for
  end for
end for
```

Für festes k sind also folgende Teile von L, U und $A^{(k)}$ berechnet:



Die Elemente von U und $A^{(k)}$ heißen $a_{i,j}$ in Algorithmus 2.4.2. Prinzipiell sollte man die Elemente von L auch in A abspeichern, also $a_{s,k} := a_{s,k}/a_{k,k}$ statt $\ell_{s,k} :=$ ..., was aber die Übersicht bei unseren Darstellungen verringern würde. Die Variable k heißt hierbei **Stufenindex** oder **Stufe**.

2.4.3 Definition
Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt auf die Prozssoren $P_1, ..., P_p$ **zyklisch nach Spalten** (**Zeilen**) abgespeichert, falls gilt:
Prozessor P_i enthält genau die Spalten (Zeilen) j von A mit $j \equiv i \mod p$.

Beispiel: Ist $n = 8$ und $p = 3$, so ergibt sich:

Prozessor	enthaltene Spalten (Zeilen)
P_1:	1, 4, 7
P_2:	2, 5, 8
P_3:	3, 6

Bei der Gauß-Elimination in der Stufe k sind die aktiven Spalten von A die Spalten $k + 1, ..., n$, bei zyklischer Abspeicherung in P_i also alle j mit $j \equiv i \mod p \wedge j > k$.

2.4.4 Lemma
Für $k \in \{1, ..., n\}$ und $i \in \{1, ..., p\}$ sei

$$M(k, i) := \{j|\ j \equiv i \mod p\ \wedge j > k\}.$$

Dann gilt:

$$|\ |M(k, i)| - |M(k, i')|\ | \leq 1,\ i, i' \in \{1, ..., p\}.$$

Beweis: Es seien k und i fest, es bezeichne $n = mp + \ell$ mit $m \geq 0$ und $0 \leq k < \ell$. Desweiteren sei $k = ap + b, a \geq 0$ und $0 \leq b < p$. Dann gibt es zwei Fälle zu unterscheiden:

1. $b \leq \ell$, dann gilt

$$|M(k, i)| = \begin{cases} m - a & \text{für } i \leq b \text{ oder } i > \ell \\ m - a + 1 & \text{für } b < i \leq \ell \end{cases}$$

2. $b > \ell$, dann gilt

$$|M(k, i)| = \begin{cases} m - a & \text{für } i \leq \ell \text{ oder } i > b \\ m - a + 1 & \text{für } \ell < i \leq b \end{cases}$$

Damit ist gezeigt, dass $|M(k,i)|$ bei festem k nur die beiden Werte $m-a$ und $m-a+1$ annimmt. □

Lemma 2.4.4 bedeutet, dass die Rechenlast bei der Gauß-Elimination bei zyklischer Abspeicherung in jeder Stufe (d.h. für jedes k) „ziemlich gut" verteilt ist.

Bezeichnung: Es bezeichne im Folgenden $mycolumns$ in Prozessor P_ℓ

$$mycolumns := \{j \mid P_\ell \text{ enthält Spalte } j\}.$$

Bei zyklischer Speicherung ist also

$$mycolumns = \{j \mid j \equiv \ell \mod p\}.$$

Wählen wir eine spaltenorientierte Abspeicherung, so beschreibt der folgende Pseudocode die LU-Zerlegung

2.4.5 Algorithmus

```
for k = 1 to n − 1 do
  if k ∈ mycolumns then
    for i = k + 1 to n do
      ℓ_{i,k} := a_{i,k}/a_{k,k}
    end for
    broadcast(ℓ_{k+1,k}, ..., ℓ_{n,k})
  else
    receive(ℓ_{k+1,k}, ..., ℓ_{n,k})
  end if
  for (j ≥ k + 1) and (j ∈ mycolumns) do
    for i = k + 1 to n do
      a_{i,j} := a_{i,j} − ℓ_{i,k} a_{k,j}
    end for
  end for
end for
```

Wie Abbildung 2.1 zeigt, entstehen dabei „Totzeiten" in allen Prozessoren, denn

1. die Berechnung der $\ell_{i,k}$ erfolgt in nur einem Prozessor
2. die Kommunikation verzögert Senden und Empfangen (In Abbildung 2.1 *zu klein* eingezeichnet).

Abhilfe: Verwenden einer „send-ahead"-Strategie.

Aktivitätszeiten für Algorithmus 2.4.5 ($p = 3, n = 9$):

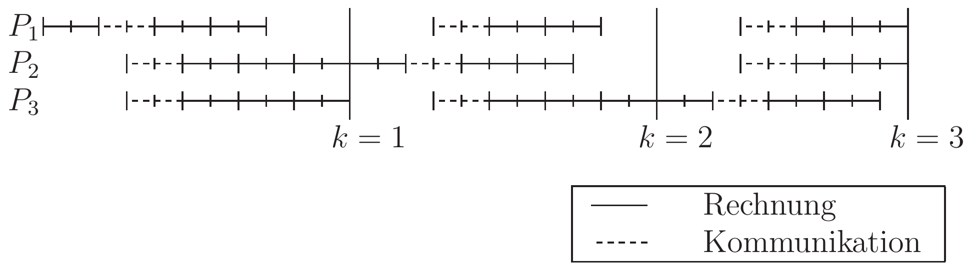


Abbildung 2.1: Gantt-Diagramm für Algorithmus 2.4.5

Berechne und versende zu kommunizierende Daten sobald wie möglich!

Beispiel: Der Prozessor P_2 kann $\ell_{3,2}, ..., \ell_{n,2}$ ausrechnen und versenden, sobald er (für $k = 1$) die 2. Spalte von A aufdatiert hat, also *bevor* er die Aufdatierung in der 1. Stufe der übrigen Spalten vornimmt (d.h. der Spalten $p + 2, 2p + 2, ...$).

Bemerkung: Um einen send-ahead zu ermöglichen muss das Senden von Daten unabhängig von der Empfangsbereitschaft des Empfängers möglich sein (nicht-blockierendes Senden, vergleiche MPI-Standard).

Aktivitätzeiten für Algorithmus 2.4.6 ($p = 3, n = 9$):

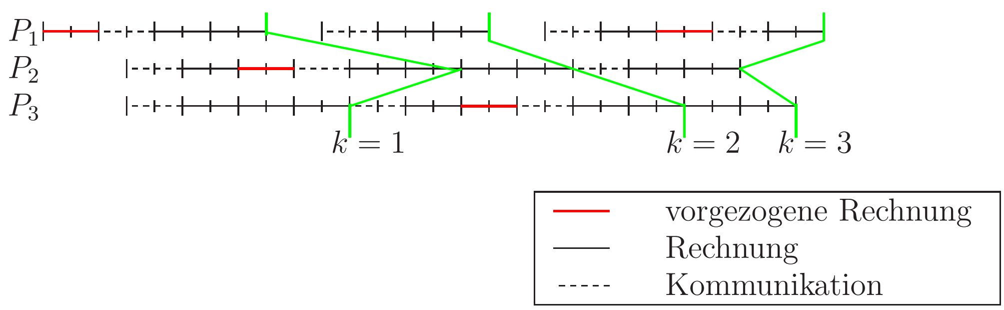


Abbildung 2.2: Gantt-Diagramm für Algorithmus 2.4.6

Die send-ahead-Strategie können wir folgendermaßen einarbeiten.

2.4.6 Algorithmus

```
if 1 ∈ mycolumns then
   for i = 2 to n do
      ℓ_{i,1} = a_{i,1}/a_{1,1}
   end for
   broadcast(ℓ_{2,1}, ..., ℓ_{n,1})
end if
for k = 1 to n − 1 do
   if k ∉ mycolumns then
      receive(ℓ_{k+1,k}, ..., ℓ_{n,k})
   end if
   for (j ≥ k + 1) and (j ∈ mycolumns) do
      for i = k + 1 to n do
         a_{i,j} = a_{i,j} − ℓ_{i,k} a_{k,j}
      end for
      if (j = k + 1) and (j ≠ n) then
         for s = k + 2 to n do
            ℓ_{s,k+1} = a_{s,k+1}/a_{k+1,k+1}
         end for
         broadcast(ℓ_{k+2,k+1}, ..., ℓ_{n,k+1})
      end if
   end for
end for
```

Es ist bekannt, dass man wegen der Stabilität und der Durchführbarkeit in der Regel zusätzlich *Pivotstrategien* verwenden muss. Gauß-Elimination mit *Spalten*pivotsuche wird im nächsten Abschnitt behandelt.

2.5 Pivot-Strategien

In diesem Abschnitt betrachten wir nur Spaltenpivotsuche (dies führt zu Zeilenvertauschungen, engl.: „row-pivoting").

LU-Zerlegung mit Spaltenpivotsuche und explizitem Vertauschen

2.5.1 Algorithmus

for $k = 1$ to $n - 1$ **do**
    bestimme s mit $|a_{s,k}| = \max\{|a_{i,k}|, i = k, ..., n\}$
    vertausche Zeilen s und k von A
    for $m = k + 1$ to n **do**
        $\ell_{m,k} = a_{m,k}/a_{k,k}$
    end for
    for $j = k + 1$ to n **do**
        for $i = k + 1$ to n **do**
            $a_{i,j} = a_{i,j} - \ell_{i,k}a_{k,j}$
        end for
    end for
end for

Beachte: Im seriellen Fall wird explizites Vertauschen gewöhnlich nicht durchgeführt! Vertauschen wäre ein Umspeicher-Prozeß, der Zeit kostet. Stattdessen wird implizites Vertauschen verwendet, d.h. der Zeilenindex i läuft statt von $k+1$ bis n über die Komponenten $ind(k+1), ..., ind(n)$ eines Indexvektors, der die Vertauschungen speichert.

Das implizite Verfahren kann dann folgendermaßen implementiert werden.

LU-Zerlegung mit Spaltenpivotsuche und implizitem Vertauschen.

2.5.2 Algorithmus

Require: $ind(i) = i, \quad i = 1, ..., n.$
  for $k = 1$ to $n - 1$ **do**
    bestimme s mit $|a_{ind(s),k}| = \max\{|a_{ind(i),k}|, i = k, ..., n\}$
    vertausche die Indizes $ind(s)$ und $ind(k)$
    for $m = k + 1$ to n **do**
      $\ell_{ind(m),k} = a_{ind(m),k}/a_{ind(k),k}$
    end for
    for $j = k + 1$ to n **do**
      for $i = k + 1$ to n **do**
        $a_{ind(i),j} = a_{ind(i),j} - \ell_{ind(i),k} a_{ind(k),j}$
      end for
    end for
  end for

Wir werden gleich sehen, dass die (implizite) Spaltenpivotsuche gut zu parallelisieren ist, wenn die Matrix spaltenzyklisch abgespeichert ist.

Gauß-Elimination mit Spaltenpivotsuche auf Parallelrechnern

Falls A zyklisch nach Spalten abgespeichert ist

- findet eine Pivotsuche nur in einem Prozessor statt; zur Vermeidung von Totzeiten ist ein send-ahead noch wichtiger als früher.
- Zeilenvertauschungen sind explizit oder implizit möglich

Die LU-Zerlegung für A (zyklisch nach Spalten abgespeichert) mit Spaltenpivotsuche, implizitem Vertauschen und send-ahead ist durch folgenden Algorithmus gegeben.

2.5.3 Algorithmus

Require: $ind(i) = i, \quad i = 1, ..., n.$
  if $1 \in mycolumns$ **then**
    bestimme s mit $|a_{s,1}| = \max\limits_{j=1,...,n} |a_{j,1}|$
    for $m = 1$ to n **do**
      if $m \neq s$ **then**
        $\ell_{ind(m),1} = a_{ind(m),1} / a_{s,1}$
      end if
      broadcast$(s, \ell_{ind(1),1}, ..., \ell_{ind(s-1),1}, \ell_{ind(s+1),1}, ..., \ell_{ind(n),1})$
    end for
  end if
  for $k = 1$ to $n - 1$ **do**
    if $k \notin mycolumns$ **then**
      receive$(s, \ell_{ind(k),k}, ..., \ell_{ind(s-1),k}, \ell_{ind(s+1),k}, ..., \ell_{ind(n),k})$
    end if
    vertausche $ind(k)$ und $ind(s)$
    for $(j \geq k + 1)$ and $(j \in mycolumns)$ **do**
      for $i = k + 1$ to n **do**
        $a_{ind(i),j} = a_{ind(i),j} - \ell_{ind(i),k} \cdot a_{ind(k),j}$
      end for
      if $(j = k + 1)$ and $(j \neq n)$ **then**
        bestimme s mit $|a_{ind(s),k+1}| = \max\limits_{j=k+1,...,n} |a_{ind(j),k+1}|$
        for $m = k + 1$ to n **do**
          if $m \neq s$ **then**
            $\ell_{ind(m),k+1} = a_{ind(m),k+1} / a_{ind(s),k+1}$
          end if
          broadcast$(s,\ell_{ind(k+1),k+1},...,\ell_{ind(s-1),k+1},\ell_{ind(s+1),k+1},...,\ell_{ind(n),k+1})$
        end for
      end if
    end for
  end for

Beachte: Ein $a_{j,k} \neq 0$ kann immer gefunden werden, da A regulär ist und somit in jeder Stufe k keine Nullzeilen bzw. -spalten auftreten.

Wäre A zyklisch nach Zeilen abgespeichert, so ergäbe sich:

- Spaltenpivotsuche beschäftigt alle Prozessoren (fan-in)
- send-ahead ist nicht möglich, da eine Synchronisation für den fan-in nötig ist

- implizites Vertauschen kann zu einer unausgewogenen Rechenlast führen
- explizites Vertauschen bedeutet Kommunikation

2.5.4 Beispiel $(n = 9, p = 3)$
Es ergibt sich als Zeilenverteilung

$$\begin{array}{ll} P_1: & 1{,}4{,}7 \\ P_2: & 2{,}5{,}8 \\ P_3: & 3{,}6{,}9. \end{array}$$

Würden sich nun als Pivotzeilen der Reihe nach

$$1, 4, 7, 2, 5, 8, 3, 6, 9$$

ergeben, so bedeutete dies bei implizitem Vertauschen, dass P_1 nach der Stufe $k = 3$ und P_2 nach der Stufe $k = 6$ arbeitslos sind.

2.5.5 Satz
Der Rechenaufwand bei der Gauß-Elimination beträgt

$$\begin{array}{rl} \frac{1}{3}n^3 + \mathcal{O}(n^2) & \text{Additionen} \\ \frac{1}{3}n^3 + \mathcal{O}(n^2) & \text{Multiplikationen} \\ \frac{1}{2}n^2 + \mathcal{O}(n) & \text{Divisionen} \end{array}$$

Beachte: Divisionen sind deutlich teuerer als Additionen oder Multiplikationen; sie fallen aber nicht so sehr ins Gewicht, da nur $\mathcal{O}(n^2)$ nötig sind.

2.6 Skalierende Varianten

2.6.1 Bemerkung
Obwohl die Rechenlast in den bisher behandelten Verfahren zur Gauß-Elimination gut verteilt ist, skalieren sie *nicht* (Details siehe Übung).

Erst seit Beginn der 90er Jahre sind skalierende parallele Verfahren zur Gauß-Elimination bekannt. Hierzu muss ein Speicherschema gefunden werden, welches die Kommunikation reduziert.

2.6.2 Definition
Sei $p = P \cdot Q$ und die p Prozessoren eines Parallelrechners seien (zumindest logisch) als ein $P \times Q$-Torus angeordnet. Wir bezeichnen die Prozessoren dann mit $P_{i,j}, i = 1, ..., P,\ j = 1, ..., Q$.

Die Matrix $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$ heißt **torus-zyklisch** abgespeichert (im engl.: „torus-wrap mapping“), falls gilt

$$P_{i,j} \text{ enthält } a_{k,\ell} \text{ genau dann, wenn } \begin{cases} k \equiv i \mod P \\ \ell \equiv j \mod Q \end{cases}.$$

Beispiel:$n = 5,\ P = 3$ und $Q = 2$.

$$\begin{array}{llll} P_{1,1}: & a_{1,1}, \quad a_{1,3}, \quad a_{1,5} & \qquad P_{1,2}: & a_{1,2}, \quad a_{1,4} \\ & a_{4,1}, \quad a_{4,3}, \quad a_{4,5} & & a_{4,2}, \quad a_{4,4} \\ \\ P_{2,1}: & a_{2,1}, \quad a_{2,3}, \quad a_{2,5} & \qquad P_{2,2}: & a_{2,2}, \quad a_{2,4} \\ & a_{5,1}, \quad a_{5,3}, \quad a_{5,5} & & a_{5,2}, \quad a_{5,4} \\ \\ P_{3,1}: & a_{3,1}, \quad a_{3,3}, \quad a_{3,5} & \qquad P_{3,2}: & a_{3,2}, \quad a_{3,4}. \end{array}$$

bzw., „aus Sicht der Matrix“

$$\begin{pmatrix} (1,1) & (1,2) & (1,1) & (1,2) & (1,1) \\ (2,1) & (2,2) & (2,1) & (2,2) & (2,1) \\ (3,1) & (3,2) & (3,1) & (3,2) & (3,1) \\ (1,1) & (1,2) & (1,1) & (1,2) & (1,1) \\ (2,1) & (2,2) & (2,1) & (2,2) & (2,1) \end{pmatrix}$$

Für $P_{i,j}$ sei

$$\begin{array}{lcl} mycolumns & = & \{\ell |\ \ell \equiv j \mod Q\} \\ myrows & = & \{k|\ k \equiv i \mod P\} \\ my\_z & = & i \\ my\_s & = & j. \end{array}$$

Die LU-Zerlegung von A ohne Pivotsuche für torus-zyklisch abgespeichertes A, ohne send-ahead ist gegeben durch:

2.6.3 Algorithmus

for $k = 1$ to $n-1$ **do**
  bestimme $sender\_z \in \{1, ..., P\}$, $sender\_z \equiv k \mod P$
  bestimme $sender\_s \in \{1, ..., Q\}$, $sender\_s \equiv k \mod Q$
  if $k \in mycolumns$ **then**
    if $k \in myrows$ **then**
      for $s = 1$ to P, $s \neq my\_z$ **do** {Pivot-Element an Rest der Spalte}
        send($a_{k,k}$) to P_{s,my_s}
      end for
    else
      receive($a_{k,k}$) from $P_{sender\_z,sender\_s}$
    end if
    for $i \in myrows$, $i \geq k+1$ **do** {Pivotspalten ausrechnen ...}
      $\ell_{i,k} = a_{i,k}/a_{k,k}$
    end for
    for $s = 1$ to Q, $s \neq my\_s$ **do** {... und versenden}
      send($\{\ell_{i,k}, i \in myrows, i \geq k+1\}$) to $P_{my\_z,s}$
    end for
  else
    receive($\{\ell_{i,k}, i \in myrows, i \geq k+1\}$) from $P_{my\_z,sender\_s}$
  end if
  if $k \in myrows$ **then**
    for $s = 1$ to P, $s \neq my\_z$ **do** {Pivotzeile versenden}
      send($\{a_{k,j}, j \in mycolumns, j \geq k+1\}$) to P_{s,my_s}
    end for
  else
    receive($\{a_{k,j}, j \in mycolumns, j \geq k+1\}$) from P_{sender_z,my_s}
  end if
  for $i \in myrows$, $i \geq k+1$ **do**
    for $j \in mycolumns$, $j \geq k+1$ **do**
      $a_{i,j} = a_{i,j} - \ell_{i,k}a_{k,j}$
    end for
  end for
end for

Verbesserung: Die Pivotzeilen ab $j \geq k$ zuerst versenden! Damit entfällt das Versenden des Pivotelements. Der Block

> **if** $k \in myrows$ **then**
>   **for** $s = 1$ to P, $s \neq my\_z$ **do**
>   send($a_{k,k}$) to P_{s,my_s}
>   **end for**
> **else**
>   receive($a_{k,k}$) from $P_{sender\_z,sender\_s}$
> **end if**

wird also ersetzt durch

> **if** $k \in myrows$ **then**
>   **for** $s = 1$ to Q, $s \neq my\_z$ **do**
>   send($\{a_{k,j}, j \in mycolumns, j \geq k\}$) to P_{s,my_s}
>   **end for**
> **else**
>   receive($\{a_{k,j}, j \in mycolumns, j \geq k\}$) from P_{sender_z,my_s}
> **end if**

Die letzte Abfrage 'if $k \in$ *myrows*' entfällt komplett.

2.6.4 Bemerkung

1. Für $P = 1$ ergibt sich der übliche, spaltenzyklisch orientierte Algorithmus.
2. Ist $p = P^2$, so skaliert der Algorithmus, siehe Übung.

2.7 Die BLAS

Die Abkürzung BLAS steht für **B**asic **L**inear **A**lgebra **S**ubprograms und ist eine Sammlung von FORTRAN-Unterprogrammen für Basisoperationen der linearen Algebra.

Namenskonventionen:

1. Buchstabe: Datentyp (S: single precision, D: double precision, C: complex, Z: double complex)

2.-6. Buchstabe: Beschreibung der Routine

Es existieren 3 Levels von BLAS: BLAS1-BLAS3.

1. BLAS1 enthält 15 Vektor-Vektor-Operationen (x, y seien Vektoren, α ein Skalar)

   - `_AXPY`: $y = \alpha x + y$
   - `_SCAL`: $x = \alpha x$
   - `_DOT`: $\text{dot} = x^T \cdot y$
   - `_NRM2`: $\text{nrm2} = \|x\|_2$
   - `_ASUM`: $\text{asum} = \|\text{Re}(x)\|_1 + \|\text{Im}(x)\|_1$
   - ...

2. BLAS2 enthält 25 Matrix-Vektor-Operationen (A sei eine $n \times m$-Matrix, x, y passende Vektoren, T eine $n \times n$-Dreiecksmatrix [obere oder untere])

   - `_GEMV`: $y = \alpha Ax + \beta y$
   - `_GBMV`: $y = \alpha Ax + \beta y$, A Bandmatrix
   - `_SYMV`: $y = \alpha Ax + \beta y$, A symmetrisch
   - `_TRSV`: $x = T^{-1}x$
   - `_TRMV`: $x = Tx$
   - `_GER`: $A = \alpha xy^T + A$
   - `_SYR`: $A = \alpha xy^T + A$, A symmetrisch
   - ...

3. BLAS3 enthält 9 Matrix-Matrix-Operationen (A, B, C, T seien passende Matrizen, T eine Dreiecksmatrix)

   - `_GEMM`: $C = \alpha AB + \beta C$

- `_SYMM`: $C = \alpha AB + \beta C$, A symmetrisch
- `_SYRK`: $C = \alpha AA^T + \beta C$, C symmetrisch
- `_TRMM`: $B = \alpha TB$
- `_TRSM`: $B = \alpha T^{-1}B$
- ...

Hintergrund:

1. Rechnerhersteller liefern optimierte (z.B. in Assembler programmierte), oder sich selbst optimierende Blas1-Blas3-Routinen, die von Benutzern verwendet werden können. Die Routinen optimieren im wesentlichen:
   - die eigentlichen Rechenoperationen
   - die Cachenutzung
2. Praktisch alle leistungsfähigen Prozessoren besitzen heute eine Speicher-Hierarchie wie in Abbildung 2.3:

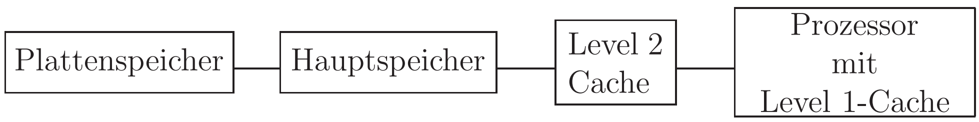


Abbildung 2.3: Speicher-Hierarchie, Standard-PC

Je mehr ein Algorithmus nur den relativ kleinen Cache-Speicher nutzt, desto schneller wird er. Optimal ist dies bei Matrix-Matrix Operationen, wenn die Matrizen in Blöcke unterteilt werden, die in den Cache-Speicher passen.

Im Wesentlichen laufen die Blas-Operationen dann besonders schnell, wenn das Verhältnis von Rechenoperationen zu Speicherzugriffen groß ist.

Es bezeichne

$$r := \frac{\#\text{Operationen}}{\#\text{Speicherzugriffe}}.$$

Dann ergibt sich, wenn $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times k}, C \in \mathbb{R}^{m\times k}$ und $x \in \mathbb{R}^n, y \in \mathbb{R}^m$

		#Op	#Zugriffe	r	
BLAS1	z.B. `_AXPY` $y = \alpha x + y$	$2n$	$3n$	$\frac{2}{3}$	
BLAS2	z.B. `_GEMV` $y = Ax$	$2mn$	$mn + n + 2m$	$\cong 2$	für $n = m$
BLAS3	z.B. `_GEMM` $C = AB$	$2mnk$	$mn + nk + 2mk$	$\frac{n}{2}$	für $n = m = k$

r ist also bei BLAS3-Operationen groß. Deshalb können die BLAS3 besonders effizient implementiert werden. Sie erreichen häufig über 90 % der theoretisch maximalen Rechenleistung.

Also sind gerade die BLAS3-Operationen durch die optimale Cache-Nutzung sehr schnell (häufig sogar bis zu 10mal schneller als von Hand programmierte).

Bemerkung: Die BLAS-Bibliotheken sind FORTRAN-Bibliotheken. Matrizen werden also spaltenweise gespeichert. Im Gegensatz dazu werden in C Matrizen zeilenweise gespeichert. Verwendung der BLAS unter C erfolgt deshalb mit Hilfe von Wrapper-Bibliotheken oder durch Umstrukturierung der Programme.

2.7.1 Beispiel (Matrix-Matrix-Multiplikation und Addition)
Gegeben: $A, B, C \in \mathbb{R}^{n\times n}$, $\beta \in \mathbb{R}$

Gesucht: $C = AB + \beta C$
Zur Implementierung werden drei verschiedene Versionen betrachtet.

for $j = 1$ to n **do**
  for $i = 1$ to n **do**
    $c_{i,j} = \beta c_{i,j}$
  end for
end for
for $k = 1$ to n **do**
  for $j = 1$ to n **do**
    for $i = 1$ to n **do**
      $c_{i,j} = c_{i,j} + a_{i,k} b_{k,j}$
    end for
  end for
end for

Es treten $n^3 + n^2$ Multiplikationen, n^3 Additionen und $n^3 + n^2$ Speicherzugriffe auf. In der BLAS1-Version (single-precision) sieht das dann folgendermaßen aus

for $j = 1$ to n **do**
  call SSCAL$(c_{\bullet,j}, \beta)$ $\quad (c_{\bullet,j} = (c_{1,j}, ..., c_{n,j}))$
end for
for $k = 1$ to n **do**
  for $j = 1$ to n **do**
    call SAXPY$(c_{\bullet,j}, a_{\bullet,k}, b_{k,j})$
  end for
end for

Dies führt zu $n^2 + n$ Aufrufen und jedesmal werden Vektoren aus dem Hauptspeicher gelesen und geschrieben!

for $j = 1$ to n **do**
  call SSCAL($c_{\bullet,j}\beta$) $c_{\bullet,j} = (c_{1,j}, ..., c_{n,j})$
end for
for $k = 1$ to n **do**
  call SGER($C, a_{\bullet,k}, b_{k,\bullet}$)
end for

Die Blas2-Version führt zu $2n$ Aufrufen mit ensprechend weniger Zugriffen auf den Hauptspeicher.

call GEMM(...)

Hier reduziert sich alles auf einen Aufruf!

Die Blas3-Routinen werden in komplizierten Algorithmen, insbesondere bei Block-orientierten Verfahren eingesetzt.

2.7.2 Beispiel
Der LINPACK-Benchmark, welcher das Ranking der Top500-Liste festlegt, testet die Leistungsfähigkeit der für den jeweiligen Rechner besten parallelen Implementierung der Gauß-Elimination. In der Regel ist dies eine blockorientierte Version von Algorithmus 2.6.3 (block-toruszyklische Abspeicherung) mit Verwendung der Blas.

2.8 Cholesky-Zerlegung, geblockte Verfahren

Im Folgenden sei $A \in \mathbb{R}^{n\times n}$ stets symmetrisch und positiv definit.

2.8.1 Satz
Für A existiert die Cholesky-Zerlegung

$$A = L \cdot L^T$$

mit einer unteren Dreiecksmatrix

$$L = \begin{pmatrix} \ell_{1,1} & & \text{\Large 0} & \\ \vdots & \ddots & & \\ \vdots & & \ddots & \\ \ell_{n,1} & \dots & \dots & \ell_{n,n} \end{pmatrix}.$$

Die Matrix L ist durch die zusätzliche Bedingung $\ell_{j,j} > 0$, $j = 1, ..., n$, eindeutig bestimmt.

Beweis: s. Numerik I □

Bemerkung:

1. Weil A symmetrisch positiv definit ist, existiert tatsächlich die LU-Zerlegung *ohne* Pivotsuche!

2. Die Idee zum Beweis des obigen Satzes ist folgende: Es gibt eine eindeutige LU-Zerlegung von A in

$$A = LU.$$

Die Matrix $D = diag(U)$ ist regulär. Mit $U' = D^{-1}U$ ist die Zerlegung

$$A = LDU'$$

eindeutig mit $diag(L) = diag(U') = I$. Wegen

$$LDU' = A = A^T = (U')^T DL^T$$

folgt damit $L = U^T$, also

$$A = LDL^T.$$

Mit A ist auch $D = L^{-1}A(L^T)^{-1}$ positiv definit, d.h. D enthält nur positive Diagonalelemente. Damit existiert dann $D^{1/2}$ als Diagonalmatrix mit $D^{1/2} \cdot D^{1/2} = D$ und es gilt

$$\begin{aligned} A &= LD^{1/2}D^{1/2}L^T \\ &= LD^{1/2}(D^{1/2})^T L^T \\ &= (L') \cdot (L')^T. \end{aligned}$$

Berechnung von L:
Aus der Bedingung

$$
\begin{aligned}
A &= L \cdot L^T \\
&= \begin{pmatrix} \ell_{1,1} & & & 0 \\ \vdots & \ddots & & \\ \vdots & & \ddots & \\ \ell_{n,1} & \dots & \dots & \ell_{n,n} \end{pmatrix} \begin{pmatrix} \ell_{1,1} & \dots & \dots & \ell_{1,n} \\ & \ddots & & \vdots \\ & & \ddots & \vdots \\ 0 & & & \ell_{n,n} \end{pmatrix}
\end{aligned}
$$

folgt

$$
a_{i,j} = \sum_{k=1}^{\min\{i,j\}} \ell_{i,k} \cdot \ell_{j,k}.
$$

Wegen der Symmetrie von A betrachten wir nur die (i, j) mit $i \geq j$ (da $a_{i,j} = a_{j,i}$). Damit ist dann

$$
a_{i,j} = \sum_{k=1}^{j} \ell_{i,k} \cdot \ell_{j,k},
$$

bzw.

$$
\begin{aligned}
a_{1,1} &= \ell_{1,1}^2 && \Rightarrow \ell_{1,1} = \sqrt{a_{1,1}}, \\
a_{2,1} &= \ell_{2,1} \cdot \ell_{1,1} && \Rightarrow \ell_{2,1} = a_{1,1}/\ell_{1,1}, \\
&\vdots \\
a_{n,1} &= \ell_{n,1} \cdot \ell_{1,1} && \Rightarrow \ell_{n,1} = a_{n,1}/\ell_{1,1}, \\
a_{2,2} &= \ell_{2,1} \cdot \ell_{2,1} + \ell_{2,2}^2 && \Rightarrow \ell_{2,2} = \sqrt{a_{2,2} - \ell_{2,1}^2}, \\
&\vdots
\end{aligned}
$$

Allgemein gilt für $i = 1, ..., n$

$$
\begin{aligned}
\ell_{i,j} &= \left(a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k} \ell_{j,k} \right) / \ell_{j,j}, \quad j = 1, ..., i-1, \\
\ell_{i,i} &= \sqrt{a_{i,i} - \sum_{k=1}^{i-1} \ell_{i,k}^2}.
\end{aligned}
$$

Berechnet man L zeilenweise, ergibt sich die ijk-Form der Cholesky-Zerlegung

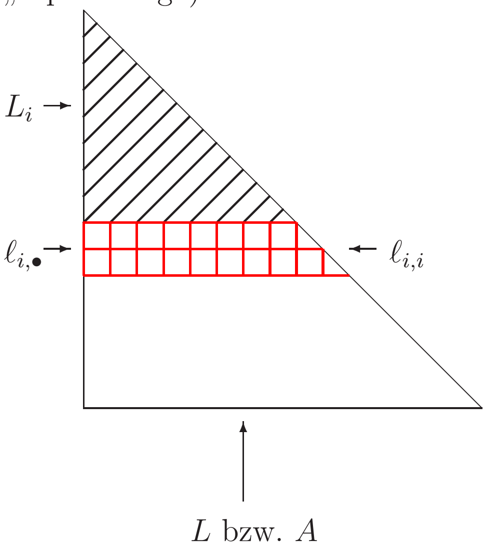
2.8.2 Algorithmus (ijk-Form der Cholesky-Zerlegung)

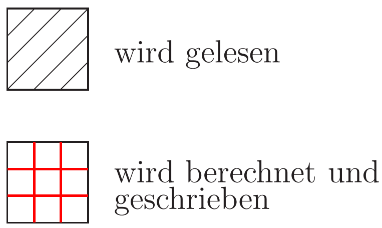
for $i = 1$ to n **do**
  for $j = 1$ to $i-1$ **do**
    $\ell_{i,j} = (a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k})/\ell_{j,j}$ `_DOT`
  end for
  $\ell_{i,i} = (a_{i,i} - \sum_{k=1}^{j-1} \ell_{i,k}^2)^{1/2}$ `_DOT`
end for

$\left.\right\}\ \ell_{i,\bullet} = a_{i,\bullet}(L_{\bullet,i}^T)^{-1}$ (`_TRSV`)

BLAS1 BLAS2

Skizze (Momentaufnahme von Algorithmus 2.8.2 für festes i, sogenanntes „top-looking“):





$$\ell_{i,\bullet} \cdot L_i^T = a_{i,\bullet}$$
$$\Longrightarrow \ell_{i,\bullet} = a_{i,\bullet} \cdot (L_i^T)^{-1}$$

(BLAS2-Routine `_TRSV`)

$$\ell_{i,i}^2 + \ell_{i,\bullet} \cdot \ell_{i,\bullet}^T = a_{i,i}$$
$$\Longrightarrow \ell_{i,i} = \sqrt{a_{i,i} - \ell_{i,\bullet}\ell_{i,\bullet}^T}$$

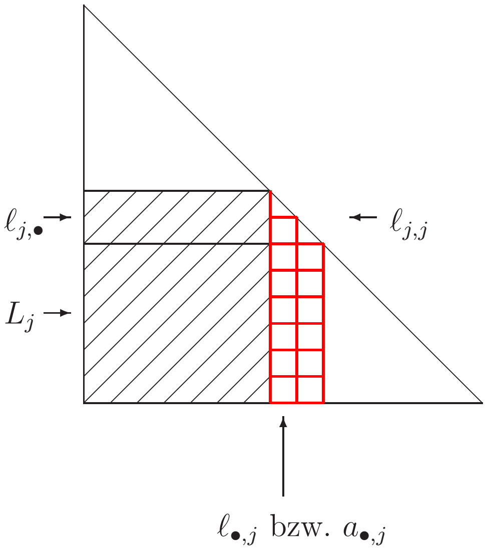
(BLAS1-Routine `_DOT`)

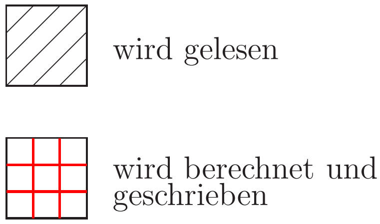
Weitere „Formen“ der Cholesky-Zerlegung erhält man durch andere Anordnung der Schleifenindizes. Wir starten mit der jik-Form, welche L spaltenorientiert berechnet.

2.8.3 Algorithmus (*jik*-Form der Cholesky-Zerlegung)

	Blas1	Blas2
for $j = 1$ to n **do**		
$\quad \ell_{j,j} = (a_{j,j} - \sum_{k=1}^{j-1} \ell_{j,k}^2)^{1/2}$	`_DDOT`	
$\quad$ **for** $i = j+1$ to n **do**		$\left.\right\}\ \ell_{\bullet,j} = \dfrac{1}{\ell_{j,j}} \cdot (a_{\bullet,j} - L_j \cdot \ell_{j,\bullet}^T)$
$\quad\quad \ell_{i,j} = (a_{i,j} - \sum_{k=1}^{j-1} \ell_{i,k}\ell_{j,k})/\ell_{j,j}$	`_DDOT`	(`_GEMV`)
$\quad$ **end for**		
end for		

Skizze (Momentaufnahme von Algorithmus 2.8.3 für festes j, sogenanntes „left-looking“):





$$\ell_{j,j}^2 + \ell_{j,\bullet} \cdot \ell_{j,\bullet}^T = a_{j,j}$$
$$\Longrightarrow \ell_{j,j} = \sqrt{a_{j,j} - \ell_{j,\bullet} \cdot \ell_{j,\bullet}^T}$$

(Blas1-Routine `_DOT`)

$$L_j \cdot \ell_{j,\bullet}^T + \ell_{\bullet,j} \cdot \ell_{j,j} = a_{\bullet,j}$$
$$\Longrightarrow \ell_{\bullet,j} = \frac{1}{\ell_{j,j}}(a_{\bullet,j} - L_j \cdot \ell_{j,\bullet}^T)$$

(Blas2-Routine `_GEMV`)

Die kji-Form der Cholesky-Zerlegung entspricht der kji-Form bei der Gauß-Elimination.

2.8.4 Algorithmus (kji-Form der Cholesky-Zerlegung)

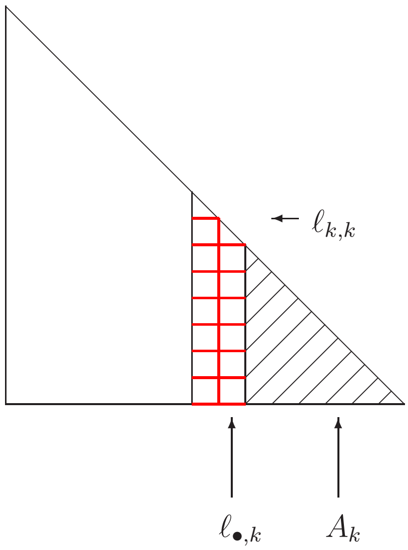
for $k = 1$ to n **do**
  $\ell_{k,k} = \sqrt{a_{k,k}}$
  for $s = k + 1$ to n **do**
    $\ell_{s,k} = a_{s,k}/\ell_{k,k}$
  end for
  for $j = k + 1$ to n **do**
    for $i = j$ to n **do**
      $a_{i,j} = a_{i,j} - \ell_{i,k} \cdot \ell_{j,k}$ } _AXPY
    end for
  end for
end for

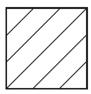
$\ell_{\bullet,k} = a_{\bullet,k}/\ell_{k,k}$
(_SCAL)

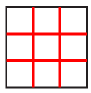
$A_k = A_k - \ell_{\bullet,k} \cdot \ell^T_{\bullet,k}$
(_SYR)

Blas1 Blas2

Skizze (Momentaufnahme von Algorithmus 2.8.4 für festes k, sogenanntes „right-looking“):




wird gelesen


wird berechnet und geschrieben

$\ell_{\bullet,k} = a_{\bullet,k}/\ell_{k,k}$

(Blas1-Routine _SCAL)

$A_k = A_k - \ell_{\bullet,k} \cdot \ell^T_{\bullet,k}$

(Blas2-Routine _SYR)

Zur Parallelisierung betrachten wir zunächst die kji-Form bei spaltenzyklisch abgespeicherten A.

2.8.5 Algorithmus (kji-Form der Cholesky-Zerlegung, parallel)

Require: A spaltenzyklisch abgespeichert
  for $k = 1$ to n **do**
    if me=$P(k)$ **then**
      $\ell_{k,k} = \sqrt{a_{k,k}}$
      for $s = k + 1$ to n **do**
        $\ell_{s,k} = a_{s,k}/\ell_{k,k}$
      end for
      broadcast($\ell_{k,k}, ..., \ell_{n,k}$)
    else
      receive($\ell_{k,k}, ..., \ell_{n,k}$)
    end if
    for $j = k + 1$ to n **do**
      if $j \in mycolumns$ **then**
        for $i = j$ to n **do**
          $a_{i,j} = a_{i,j} - \ell_{i,k}\ell_{j,k}$
        end for
      end if
    end for
  end for

Bemerkung: Eine send-ahead-Variante ist möglich.

Die _SYR-Routine der BLAS2 setzt voraus, dass die *gesamte* betroffene Matrix vorhanden ist (genauer: der obere oder untere Dreiecksanteil davon). Dies ist bei spaltenzyklischer Abspeicherung nicht der Fall. Deshalb kann Algorithmus 2.8.5 nur mit BLAS1 implementiert werden.

BLAS2-Routinen können bei der jik-Form (Algorithmus 2.8.3) und spaltenzyklischer Abspeicherung verwendet werden.

Wir formulieren diesen Algorithmus sofort BLAS2-orientiert. Vorsicht: ein Stern $\star$ steht jetzt für den Indexbereich, der auf dem jeweiligen Prozessor vorhanden ist, d.h. der Bereich

$$\{i : \ 1 \leq i \leq j \text{ und } i \in mycolumns\}.$$

2.8.6 Algorithmus (*jik*-Form der Cholesky-Zerlegung, parallel)

Require: A spaltenzyklisch abgespeichert
  for $j = 1$ to n **do**
    $\widetilde{s} = \ell_{j,\star} \cdot \ell_{j,\star}^T$
    fan-in$(\widetilde{s}, s, P_k,\ k = 1, ..., p)$ $\{$Ergebnis s in $P(j)\}$
    if $j \in mycolumns$ **then**
      $\ell_{j,j} = \sqrt{a_{j,j} - s}$
    end if
    $\tilde{\ell}_{j,\bullet} = L_{j,\star} \cdot \ell_{j,\star}^T)$ $\{$ `(_GEMV)`$\}$
    fan-in$(\tilde{\ell}_{j,\bullet}, \ell_{j,\bullet}, P_k,\ k = 1, \ldots, p)$ $\{$Ergebnis in $P(j)\}$
    if $j \in mycolumns$ **then**
      $\ell_{j,\bullet} = \frac{1}{\ell_{j,j}}(a_{j,\bullet} - \ell_{j,\bullet}^T)$ $\{$ `(_AXPY)`$\}$
    end if
  end for

Aber: Ein BLAS3-Einsatz ist bei spaltenorientierter Abspeicherung prinzipiell nicht möglich. Bei den nun folgenden geblockten Verfahren können die BLAS3 jedoch eingesetzt werden.

Block-Cholesky-Zerlegung: Berechne die übliche Cholesky-Zerlegung, verwende nun aber statt einzelner Elemente ganze Blöcke von A und L.

$$L = \begin{pmatrix} L_{1,1} & & & & 0 \\ \vdots & \ddots & & & \\ \vdots & & \ddots & & \\ \vdots & & & \ddots & \\ L_{m,1} & \dots & \dots & \dots & L_{m,m} \end{pmatrix},$$

$L_{i,i} \in \mathbb{R}^{n_i \times n_i}$, $i = 1, ..., n$ sind untere Dreiecksmatrizen und es gilt

$$\sum_{i=1}^{m} n_i = n$$

$$A = \begin{pmatrix} A_{1,1} & \dots & \dots & \dots & A_{1,m} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ A_{m,1} & \dots & \dots & \dots & A_{m,m} \end{pmatrix},$$

entsprechend mit $A_{i,i} = A_{i,i}^T$ und

$$A_{i,j} = A_{j,i}^T, \; i, j = 1, ..., m.$$

Die Bedingung

$$A = L \cdot L^T$$

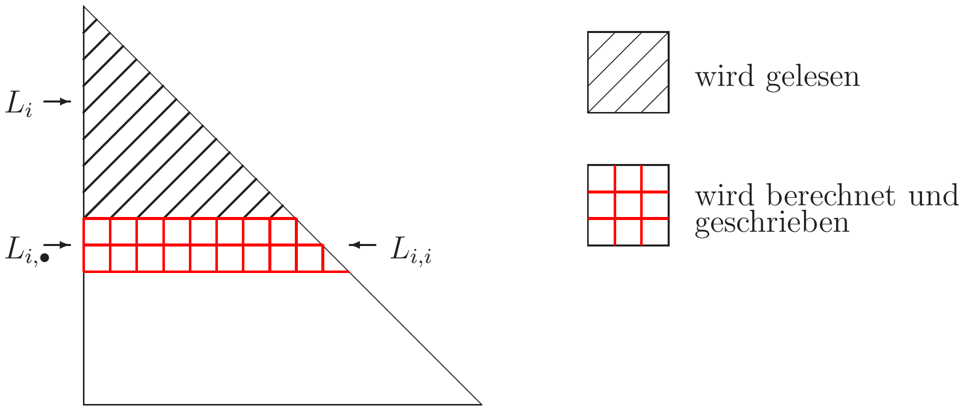
liefert nun wiederum für jedes $i = 1, ..., m$:

$$\begin{aligned} A_{i,j} &= \sum_{k=1}^{j} L_{i,k} L_{j,k}^T, \quad j = 1, ..., i-1, \\ A_{i,i} &= \sum_{k=1}^{i} L_{i,k} L_{i,k}^T. \end{aligned}$$

Also ergibt sich

$$\begin{aligned} L_{i,j} &= \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k}^T \right) \left(L_{j,j}^T \right)^{-1} \\ L_{i,i} \cdot L_{i,i}^T &= A_{i,i} - \sum_{k=1}^{i} L_{i,k} L_{i,k}^T \quad \text{(„kleine“ Cholesky-Zerlegung).} \end{aligned}$$

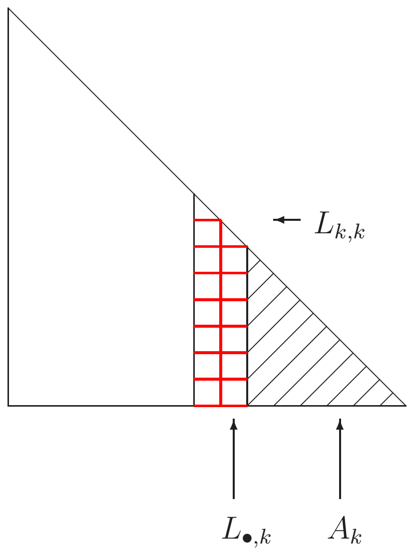
Skizze (Momentaufnahme für Block-„top-looking“-Variante):

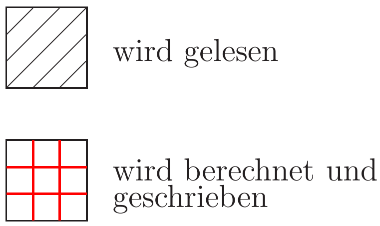


$$
\begin{aligned}
L_{i,\bullet} &= A_{i,\bullet} \cdot (L_i^T)^{-1} \\
L_{i,i} \cdot L_{i,i}^T &= A_{i,i} - L_{i,\bullet} \cdot L_{i,\bullet}^T
\end{aligned}
$$

Dabei können die kleine Cholesky-Zerlegung $L_{i,i} \cdot L_{i,i}^T$ mit BLAS1- und BLAS2-Routinen und $A_{i,i} - L_{i,\bullet} \cdot L_{i,\bullet}^T$ (`_GEMM`) sowie $A_{i,\bullet} \cdot (L_i^T)^{-1}$ (`_TRSM`) mit BLAS3-Routinen berechnet werden.

Skizze (Momentaufnahme für Block-„right-looking“-Variante):





$$
\begin{aligned}
L_{k,k} \cdot L_{k,k}^T &= A_{k,k} \\
L_{\bullet,k} &= A_{\bullet,k} \cdot (L_{k,k}^T)^{-1} \\
A_k &= A_k - L_{\bullet,k} \cdot L_{\bullet,k}^T
\end{aligned}
$$

Wieder können die kleine Cholesky-Zerlegung $L_{k,k} \cdot L_{k,k}^T$ mit BLAS1- und BLAS2-Routinen und $A_k - L_{\bullet,k} \cdot L_{\bullet,k}^T$ (`_GEMM`) sowie $A_{\bullet,k} \cdot (L_{k,k}^T)^{-1}$ (`_TRSM`) mit BLAS3-Routinen berechnet werden.

Ist A zyklisch nach Spalten*blöcken* $A_{\bullet,1}, ..., A_{\bullet,m}$ abgespeichert, so kann die kji-Form der Cholesky-Zerlegung folgendermaßen implementiert werden.

2.8.7 Algorithmus (kji-Form, blockorientiert)

Require: A blockspaltenzyklisch abgespeichert
  for $k = 1$ to m **do**
    if me=$P(k)$ **then**
      berechne Cholesky-Zerlegung $A_{k,k} = L_{k,k} \cdot L_{k,k}^T$ } BLAS1, BLAS 2
      $L_{\bullet,k} = A_{\bullet,k} \cdot (L_{k,k}^T)^{-1}$ } BLAS3
      broadcast($L_{\bullet,k}$)
    else
      receive($L_{\bullet,k}$)
    end if
    for $j = k+1$ to m **do**
      if $j \in myblockcolumns$ **then**
        $A_{\bullet,j} = A_{\bullet,j} - L_{\bullet,k} \cdot L_{j,k}^T$ } BLAS3
      end if
    end for
  end for

Vorteile

- jeder Prozessor kann BLAS3-Routinen verwenden (dies bedeutet im typischen Fall eine Beschleunigung um einen Faktor 2 bis 5!)
- es werden weniger (nämlich m) Kommunikationen angestoßen

Nachteil

- Die Rechenlast ist weniger ausgeglichen verteilt. Jedoch ist eine send-ahead-Variante möglich.

Auch für die anderen „Formen" der Cholesky-Zerlegung können entsprechende parallele Varianten bei blockspaltenzyklischer Abspeicherung formuliert werden.

2.8.8 Satz
Der Rechenaufwand für die Cholesky-Zerlegung beträgt

$$\begin{aligned} &\frac{1}{6}n^3 + \mathcal{O}(n^2) \qquad \text{Additionen} \\ &\frac{1}{6}n^3 + \mathcal{O}(n^2) \qquad \text{Multiplikationen} \end{aligned}$$

$$\begin{aligned} \frac{1}{2}n^2 + \mathcal{O}(n) &\qquad \text{Divisionen} \\ n &\qquad \text{Wurzeln} \end{aligned}$$

Beweis: Abzählen. □

2.9 Gestaffelte Systeme

Gegeben: $L = (\ell_{i,j}) \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^n, \quad \ell_{i,j} \begin{cases} = 0 & i < j \\ \neq 0 & i = j = 1, ..., n \end{cases}$

Gesucht: Das eindeutige $x \in \mathbb{R}^n$ mit $Lx = b$.

Wir untersuchen solche unteren Dreieckssysteme stellvertretend für die beiden Arten von Dreieckssystemen

$$\begin{aligned} Ly &= b \\ Ux &= y \end{aligned}$$

wie sie bei der Lösung von $Ax = b$ nach erfolgter LU-Faktorisierung $A = LU$ zu lösen sind.

Wir beschreiben zuerst zwei serielle Varianten

2.9.1 Algorithmus (*ij*-Form für gestaffelte Systeme)

Require: $s_i = 0,\ i = 1, ..., n$
  for $i = 1$ to n **do**
    for $j = 1$ to $i - 1$ **do**
      $s_i = s_i + \ell_{i,j} x_j$
    end for
    $x_i = (b_i - s_i)/\ell_{i,i}$
  end for

$\left.\begin{array}{l} \textbf{for } j = 1 \text{ to } i-1 \textbf{ do} \\ \quad s_i = s_i + \ell_{i,j} x_j \\ \textbf{end for} \end{array}\right\}$ `(_DOT)`

In der ji-Form ergibt das

2.9.2 Algorithmus (*ji*-Form für gestaffelte Systeme)

Require: $s_i = 0,\ i = 1, ..., n$
  for $j = 1$ to n **do**
    $x_j = (b_j - s_j)/\ell_{j,j}$
    for $i = j + 1$ to n **do**
      $s_i = s_i + \ell_{i,j} x_j$
    end for
  end for

$\left.\begin{array}{l} \textbf{for } i = j+1 \text{ to } n \textbf{ do} \\ \quad s_i = s_i + \ell_{i,j} x_j \\ \textbf{end for} \end{array}\right\}$ `(_AXPY)`

Die ij-Form kann bei spaltenzyklischem Speicherschema folgendermaßen implementiert werden

2.9.3 Algorithmus (ij-Form für gestaffelte Systeme)

Require: L spaltenzyklisch abgespeichert
  for $i = 1$ to n **do**
    $\widetilde{s} = 0$
    for $j = 1$ to $i - 1$ and $j \in mycolumns$ **do**
      $\widetilde{s} = \widetilde{s} + \ell_{i,j} \cdot x_j$
    end for (_DOT)
    fan-in$(\widetilde{s}, s_i, P_k, k = 1, ..., p)$ $\{$Ergebnis s_i in $P(i)\}$
    if $i \in mycolumns$ **then**
      $x_i = (b_i - s_i)/\ell_{i,i}$
    end if
  end for

Bemerkung:

Da ein fan-in pro verteilte _DOT-Operation nötig ist, ergibt sich eine sehr hohe Kommunikation relativ zur Rechnung.

Ist L zeilenzyklisch abgespeichert, so kann die ji-Form folgendermaßen implementiert werden.

2.9.4 Algorithmus (ji-Form für gestaffelte Systeme)

Require: L zeilenzyklisch abgespeichert
  for $j = 1$ to n **do**
    if $j \in myrows$ **then**
      $x_j = (b_j - s_j)/\ell_{j,j}$
      if $j < n$ **then**
        broadcast(x_j)
      end if
    else
      if $j < n$ **then**
        receive(x_j) from $P(j)$
      end if
      for $(i \geq j + 1)$ and $i \in myrows$ **do**
        $s_i = s_i + \ell_{ij} x_j$
      end for (_AXPY)
    end if
  end for

Bemerkung:

1. In 2.9.4 kann zusätzlich noch eine send-ahead-Strategie angewendet werden im Gegensatz zu 2.9.3, da dort eine Synchronisation wegen der fan-ins erforderlich ist.

2. 2.9.4 ist im allgemeinen deutlich schneller als 2.9.3, setzt aber eine zeilenweise Abspeicherung von L voraus!

2.9.5 Bemerkung
Der Rechenaufwand in 2.9.3 und 2.9.4 beträgt

$$\begin{array}{rl} \dfrac{n^2}{2} + \mathcal{O}(n) & \text{Additionen} \\ \dfrac{n^2}{2} + \mathcal{O}(n) & \text{Multiplikationen} \\ n & \text{Divisionen} \end{array}$$

Zum Vergleich: Die LU-Zerlegung benötigt

$$\begin{array}{rl} \dfrac{1}{3}n^3 + \mathcal{O}(n^2) & \text{Additionen} \\ \dfrac{1}{3}n^3 + \mathcal{O}(n^2) & \text{Multiplikationen} \\ \dfrac{1}{2}n^2 + \mathcal{O}(n) & \text{Divisionen} \end{array}$$

Die ji-Form kann bei spaltenzyklischer Abspeicherung mit Hilfe eines Pipeline-Schema parallelisiert werden.

Dazu führt ein Prozessor nacheinander die Berechnung einzelner „Segmente“ statt ganzer Spalten durch. Jedes Segment wird nach erfolgter Berechnung bereits weiterkommuniziert.

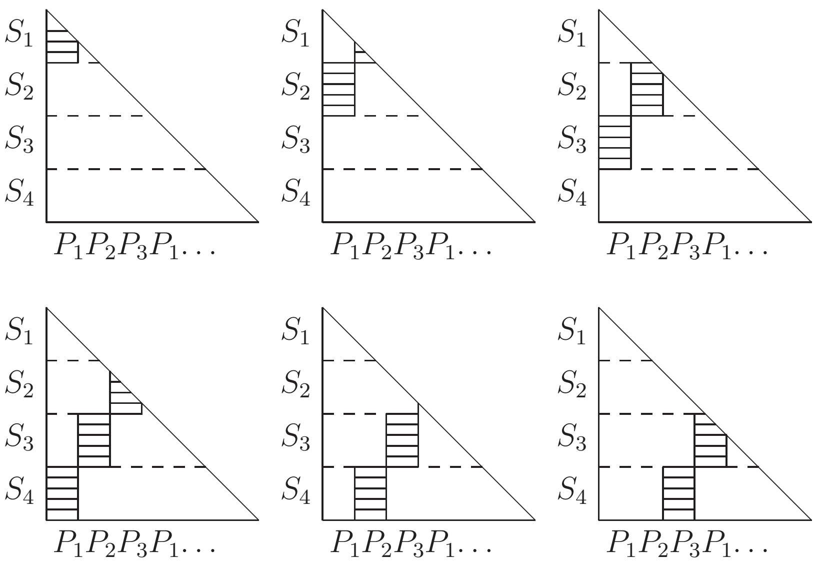
Es sei σ die Segmentlänge, $n = \sigma{\cdot}N,\ N \in \mathbb{N}$. Wir bezeichnen die Segmente mit

$$S_r = \{(r-1)\sigma + 1, ..., r\sigma\}, \quad r = 1, ..., N$$

2.9.6 Algorithmus (*ji*-Form für gestaffelte Systeme, Pipeline)

Require: L spaltenzyklisch abgespeichert, $s_i = 0,\ i = 0, ..., n$
  for $j \in mycolumns$ **do**
    {aktuelles oberes Segment}
    receive$(s_i, i \in S_{\lceil \frac{j}{\sigma} \rceil},\ i \geq j)$ from $P(j-1)$
    $x_j = (b_j - s_j)/\ell_{j,j}$
    for $(i \in S_{\lceil \frac{j}{\sigma} \rceil})$ and $(i > j)$ **do**
      $s_i = s_i + \ell_{i,j} x_j$ (_AXPY)
    end for
    send$(s_i, i \in S_{\lceil \frac{j}{\sigma} \rceil},\ i > j)$ to $P(j+1)$
    {übrige Segmente}
    for $r = \lceil \frac{j}{\sigma} \rceil + 1$ to N **do**
      receive$(s_i, i \in S_r)$ from $P(j-1)$
      for $i \in S_r$ **do**
        $s_i = s_i + \ell_{i,j} x_j$ (_AXPY)
      end for
      send$(s_i, i \in S_r)$ to $P(j+1)$
    end for
  end for

Die nachfolgende Abbildung zeigt für den Fall $p = 3$, welche Bereiche von L jeweils gleichzeitig bearbeitet werden.



Bemerkung:

1. Für $j = 1$ fallen `receive`- und für $j = n$ `send`-Ausdrücke weg. Für $j = \sigma r,\ r \in \mathbb{N}$, würde beim „obersten“ Segment die leere Menge an $P(j+1)$ gesendet werden, dieses `send` fällt weg (das zugehörige receive ist nicht vorhanden).

2. Eine Ring-Topologie ist hier ausreichend.

3. σ ist ein anpassbarer Parameter

4. Experimente zeigen: Algorithmus 2.9.6 ist bis doppelt so schnell wie Algorithmus 2.9.4!

2.10 Integralgleichungen und vollbesetzte lineare Gleichungssysteme

Modellierungen von einigen physikalischen Vorgängen führen auf sogenannte Fredholmsche Integralgleichungen zweiter Art

$$x(s) = \int_a^b k(s,t)x(t)\ dt + b(s), \tag{2.10.1}$$

wobei die Funktion k, der sogenannte Kern, mit

$$k : [a,b] \times [a,b] \to \mathbb{R}$$

stetig sei.

Wann besitzt nun die Gleichung (2.10.1) eine Lösung? Die Antwort gibt die Funktionalanalysis, durch Betrachtung von

$$\begin{aligned} K : \ \mathcal{C}([a,b]) &\to \mathcal{C}([a,b]) \\ x &\mapsto \int_a^b k(s,t)x(t)\ dt \end{aligned}$$

als *Operator* auf dem *Banachraum* $(\mathcal{C}([a,b]), \|\cdot\|_\infty)$. Man kann nun allgemeiner folgenden Begriff einführen.

2.10.1 Definition
Es sei $T : A \to B$ ein linearer Oparator zwischen zwei Banachräumen A und B. Der Operator T heißt **kompakt** genau dann, wenn das Bild der Einheitskugel von A unter T relativ kompakt in B ist.

Damit kann man nun folgendes zeigen

2.10.2 Satz (Fredholmsche Alternative)
Es sei $T : \ A \to B$ kompakt. Dann besitzt die Gleichung

$$(I - T)x = b \tag{2.10.2}$$

entweder für jedes $b \in B$ eine eindeutige Lösung *oder* sie besitzt für $b = 0$ unendlich viele Lösungen und damit für beliebiges b stets keine oder gleich unendlich viele Lösungen.

Beweis: Siehe Funktionalanalsysis. □

Der obige Operator K ist z.B. kompakt, wenn

$$\int_a^b |k(s,t)|\,dt < \infty$$

und

$$\lim_{\zeta\to s}\int_a^b |k(\zeta,t)-k(s,t)|\,dt = 0 \quad \text{für alle } s\in[a,b] \text{ gilt.}$$

Zur tatsächlichen numerischen Lösung von (2.10.1) gibt es verschiedene Verfahren, wie z.B. Quadraturformeln

$$\int_a^b f(t)\,dt \approx \sum_{j=1}^n w_j\cdot f(t_j)$$

mit Gewichtung $w_j\in\mathbb{R}$ und einer Zerlegung des Intervalls mit Knoten $a\le t_1\le t_2\le ...\le t_n\le b$, z.B. mit Trapez-Regel oder Gauß-Quadratur.

Damit wird aber die ursprüngliche Gleichung (2.10.1) *semi-diskretisiert* durch

$$x(s)\approx\sum_{j=1}^n w_j k(s,t_j)x(t_j)+b(s). \tag{2.10.3}$$

Wähle jetzt noch diskrete Werte für s, nämlich $s=t_i$:

$$x(t_i)\approx\sum_{j=1}^n w_j k(t_i,t_j)x(t_j)+b(t_i)\quad i=1,...,n$$

also ergibt sich ein Gleichungssystem

$$Ax=b,\quad A\in\mathbb{R}^{n\times n},\ b\in\mathbb{R}^n$$

mit

$$\begin{aligned} x_i &= x(t_i)\\ b_i &= b(t_i)\\ A &= (a_{i,j})_{i,j} \end{aligned}$$

wobei

$$a_{i,j}=\begin{cases}1-w_iK(t_i,t_i) & i=j\\ -w_jK(t_i,t_j) & i\neq j.\end{cases}$$

Dies ist ein im Allgemeinen vollbesetztes lineares Gleichungssystem.

2.11 Tridiagonalsysteme

Eine **Tridiagonalmatrix** besitzt die Form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_1 & \ddots & \ddots & & 0 \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix}.$$

Unser Ziel ist es, das lineare Gleichungssystem

$$Ax = d$$

mit einer Tridiagonalmatrix A parallel zu lösen. Dazu sei $N' = \lceil \log_2(n) \rceil$, wir setzen

$$\begin{aligned} a_i^{(0)} &= a_i, \quad i = 2, ..., n, \quad a_1^{(0)} = 0 \\ b_i^{(0)} &= b_i, \quad i = 1, ..., n, \\ c_i^{(0)} &= c_i, \quad i = 1, ..., n-1, \quad c_n^{(0)} = 0 \\ d_i^{(0)} &= d_i, \quad i = 1, ..., n, \end{aligned}$$

und desweiteren

$$a_i^{(k)} = c_i^{(k)} = d_i^{(k)} = 0, \quad b_i^{(k)} = 1 \text{ für } i \in \mathbb{Z} \setminus \{1, ..., n\},\ k = 0, ..., N'.$$

Es sei $x = (x_1, ..., x_n)^T$ die Lösung von $Ax = d$ und $x_i = 0$ für $i \in \mathbb{Z} \setminus \{1, ..., n\}$, dann gilt

2.11.1 Satz
Für $k = 1, ..., N'$ seien $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}$ rekursiv definiert durch

$$\text{(2.11.1)} \qquad \begin{cases} a_i^{(k)} = \alpha_i^{(k-1)} a_{i-2^{k-1}}^{(k-1)} \\ b_i^{(k)} = \alpha_i^{(k-1)} c_{i-2^{k-1}}^{(k-1)} + b_i^{(k-1)} + \gamma_i^{(k-1)} a_{i+2^{k-1}}^{(k-1)} \\ c_i^{(k)} = \gamma_i^{(k-1)} c_{i+2^{k-1}}^{(k-1)} \\ d_i^{(k)} = \alpha_i^{(k-1)} d_{i-2^{k-1}}^{(k-1)} + d_i^{(k-1)} + \gamma_i^{(k-1)} d_{i+2^{k-1}}^{(k-1)} \end{cases}$$

mit

$$\alpha_i^{(k-1)} = -\frac{a_i^{(k-1)}}{b_{i-2^{k-1}}^{(k-1)}}$$

$$\gamma_i^{(k-1)} = -\frac{c_i^{(k-1)}}{b_{i+2^{k-1}}^{(k-1)}}$$

dann gilt

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = d_i^{(k)}, \quad i \in \mathbb{Z} \tag{2.11.2}$$

Beweis: Die Gleichung (2.11.1) ist trivialerweise für $i \in \mathbb{Z} \setminus \{1, ..., n\}$, $k = 0, ..., N'$ erfüllt. Für $k = 0$ ist (2.11.1) das Ausgangssystem, also auch erfüllt.

$k - 1 \to k$: Wir schreiben die Gleichung $i - 2^{k+1}, i$ und $i + 2^{k+1}$ auf:

$$\begin{array}{ccccccccc}
a_{i-2^{k-1}}^{(k-1)} x_{i-2^k} & + & b_{i-2^{k-1}}^{(k-1)} x_{i-2^{k-1}} & + & c_{i-2^{k-1}}^{(k-1)} x_i & & & & = d_{i-2^{k-1}}^{(k-1)} \\
 & & a_i^{(k-1)} x_{i-2^{k-1}} & + & b_i^{(k-1)} x_i & + & c_i^{(k-1)} x_{i+2^{k-1}} & & = d_i^{(k-1)} \\
 & & & & a_{i+2k-1}^{(k-1)} x_i & + & b_{i+2k-1}^{(k-1)} x_{i+2k-1} & + \; c_{i+2k-1}^{(k-1)} x_{i+2^k} & = d_{i+2^{k-1}}^{(k-1)}
\end{array}$$

Multiplikation der ersten Zeile mit $\alpha_i^{(k-1)}$, der dritten Zeile mit $\gamma_i^{(k-1)}$ und Addition ergibt

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = d_i^{(k)}.$$

□

Bemerkung:

1. Tatsächlich gilt sogar

$$\begin{cases} a_i^{(k)} = \alpha_i^{(k)} = 0 & i = 1, ..., 2^k \\ c_i^{(k)} = \gamma_i^{(k)} = 0 & i = n - 2^k + 1, ..., n \end{cases}. \tag{2.11.3}$$

2. Die $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}$ können bei festem k parallel berechnet werden.

3. Für $k = N'$ ist $a_i^{(k)} = 0 = c_i^{(k)}$, $i = 1, ..., n$. Also gilt dann

$$x_i = d_i^{(N')}/b_i^{(N')}, \quad i = 1, ..., n$$

2.11.2 Algorithmus (Hockney und Golub, rekursives Verdoppeln)

for $k = 1$ to N' **do**
  for $i = 1$ to n **do**
    berechne $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}$
  end for
end for
for $i = 1$ to n **do**
  $x_i := d_i^{(N')}/b_i^{(N')}$
end for

Abbildung 2.4 stellt Algorithmus 2.11.2 schematisch dar ($n = 8$).

Das Verfahren des rekursiven Verdoppelns erfordert nur $N' + 1$ parallele Schritte, hat aber einen Gesamtaufwand von $\mathcal{O}(nN') = \mathcal{O}(n \log(n))$.

Der Gesamtrechenaufwand kann auf $\mathcal{O}(n)$ reduziert werden, wenn man nur bestimmte Koeffizienten $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}$ berechnet. Die Komponenten der Lösung gewinnt man dann rekursiv Der nächste Algorithmus formuliert das resultierende Verfahren der zyklischen Reduktion im Detail.

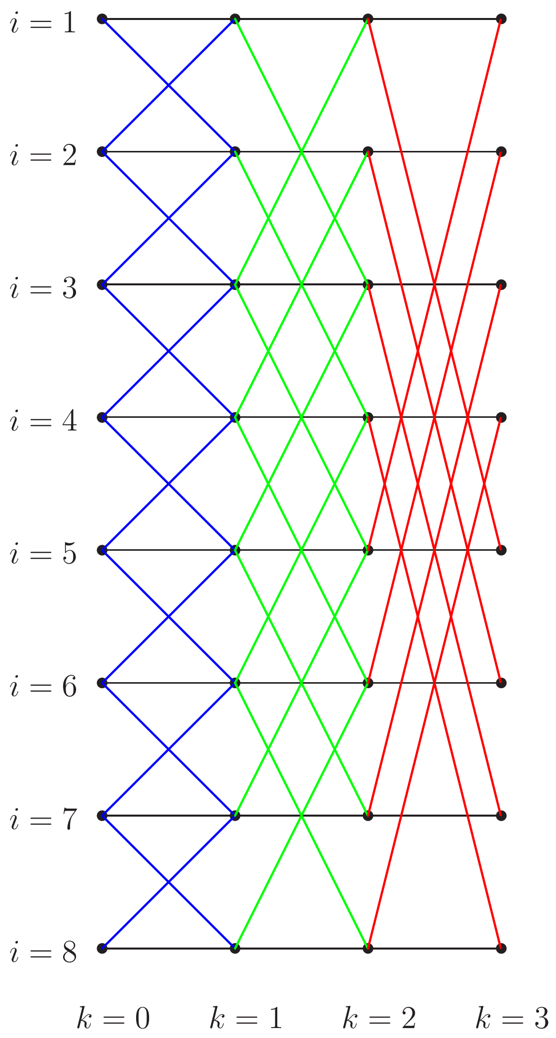


Abbildung 2.4: rekursives Verdoppeln für $n = 8$

2.11.3 Algorithmus (Hockney und Golub, zyklische Reduktion)

for $k = 1$ to $\widetilde{N}$ **do**
  for $i = 1$ to n und $i = 2^k(2^k)n^1$ **do**
    berechne $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}$
  end for
end for
for $k = \widetilde{N}$ to 1 **do**
  for $i = 1$ to n und $i = 2^k(2^k)n$ **do**
    $x_i := d_i^{(N')}/b_i^{(N')}$
  end for
end for

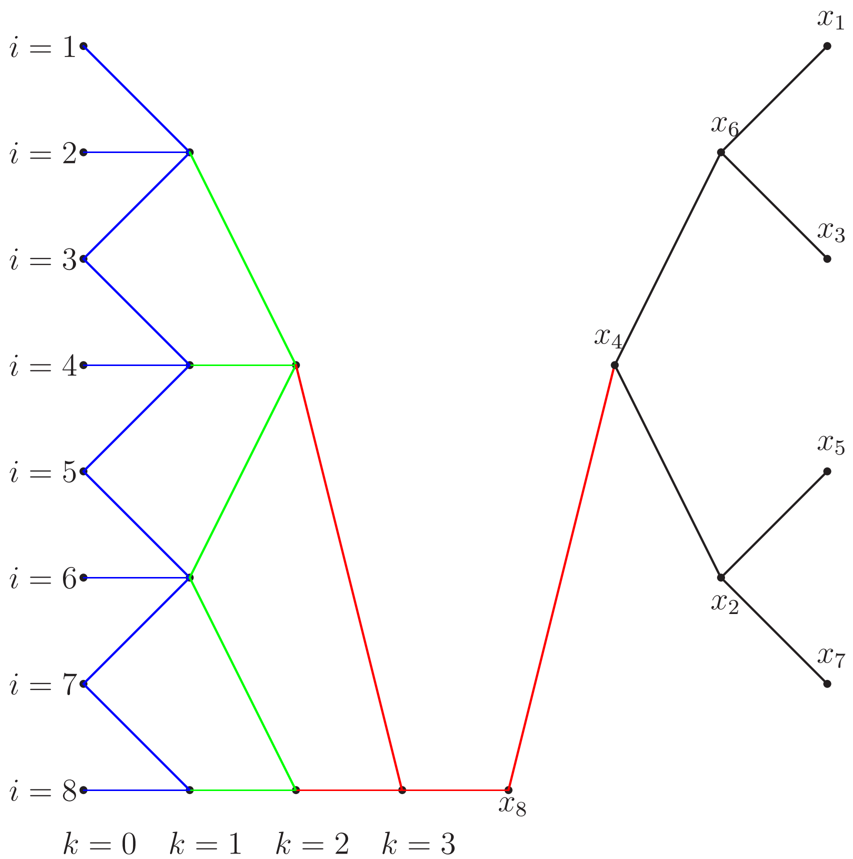


Abbildung 2.5: zyklische Reduktion für $n = 8$

2.11.4 Satz
Der Rechenaufwand für das Verfahren von Hockney und Golub, zyklische Reduktion, beträgt:

$$\begin{aligned} &3n + \mathcal{O}(\log_2(n)) && \text{Divisionen} \\ &8n + \mathcal{O}(\log_2(n)) && \text{Multiplikationen} \\ &6n + \mathcal{O}(\log_2(n)) && \text{Additionen.} \end{aligned}$$

Beweis: Nachzählen. □

Bemerkung:

1. Bei der Variante „rekursives Verdoppeln" ist der Aufwand umgefähr um den Faktor $\log_2(n)$ höher.

2. Aufwand bei der Gauß-Elimination waren $2n$ Divisionen, $3n$ Additionen, $3n$ Multiplikationen.

Durchführbarkeit: Die Verfahren sind nur Durchführbarkeit, wenn alle $b_i^{(k)} \neq 0$ sind. Dies ist z.B. für H-Matrizen der Fall (man kann sogar zeigen, dass eine tridiagonale, symmetrische, positiv definite Matrix eine H-Matrix ist).

Kap3Kap

Kapitel 3

Dünn besetzte Matrizen und Graph-Partitionierung

3.1 Dünn besetzte Gleichungssysteme

Dünn besetzte Gleichungssysteme besitzen in Anwendungen eine wichtige Rolle. Eine Matrix $A \in \mathbb{R}^{n \times n}$ heißt dabei **dünn besetzt**, wenn A sehr viele Nullen enthält.

Konsequenz:

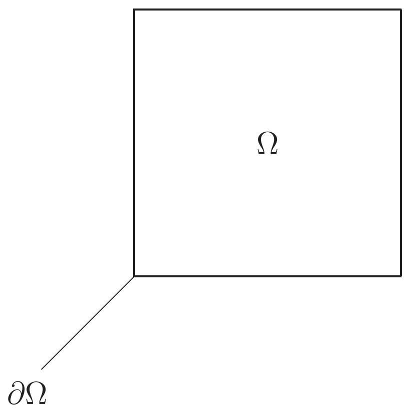
- Die Gauß-Elimination ist sehr aufwendig im Vergleich zu der Zahl der Nicht-Nullen, denn im Allgemeinen sind die Matrizen L und U nicht mehr dünn (oder „weniger dünn") besetzt, dies könnte dann sogar zu einem Speicherproblem führen.

- Abhilfe kann man dadurch schaffen, dass statt der Gauß-Elimination Iterationsverfahren verwendet werden. Die wesentlichen Operationen sind dann viele (abhängig von der Konvergenzgeschwindigkeit) Matrix-Vektoroperationen mit A oder Modifikationen von A.

Wir stellen zunächst zwei Modellprobleme vor, bei denen dünn besetzte Matrizen auftauchen.

3.2 Modellproblem I: Laplace-Gleichung, Finite Differenzen

Es sei $\Omega := (0,1) \times (0,1) \subseteq \mathbb{R}^2$.



Die *Laplace-Gleichung* mit **Dirichlet-Randbedingung** ist definiert durch das elliptische Randwertproblem:

$$\begin{array}{rcll} -\Delta u(x,y) & = & f(x,y), & (x,y) \in \Omega \\ u(x,y) & = & g(x,y), & (x,y) \in \partial\Omega \end{array}, \tag{3.2.1}$$

wobei $f:\ \Omega \to \mathbb{R},\ g:\ \partial\Omega \to \mathbb{R}$ zwei gegebene Funktionen sind und Δ der *Laplace-Operator*, der durch

$$\Delta u := \frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y)$$

definiert ist.

Für $f = 0$ beschreibt die Lösung u der partiellen Differentialgleichung z.B. die Fläche einer Seifenblase in einem Drahtrahmen, welcher durch g beschrieben ist.

Auch für „einfache“ Funktionen f und g kann man u im Allgemeinen nicht in geschlossener Form angeben. Deshalb wollen wir nun die numerische Behandlung der Laplace-Gleichung mit *finiten Differenzen* angehen:

Für $N \in \mathbb{N}$, setzen wir $h := \frac{1}{N+1}$. Dann ist

$$\Omega_h := \{(ih, jh),\ i, j = 0, ..., N+1\}$$

ein **äquidistantes Gitter**[1] auf Ω.

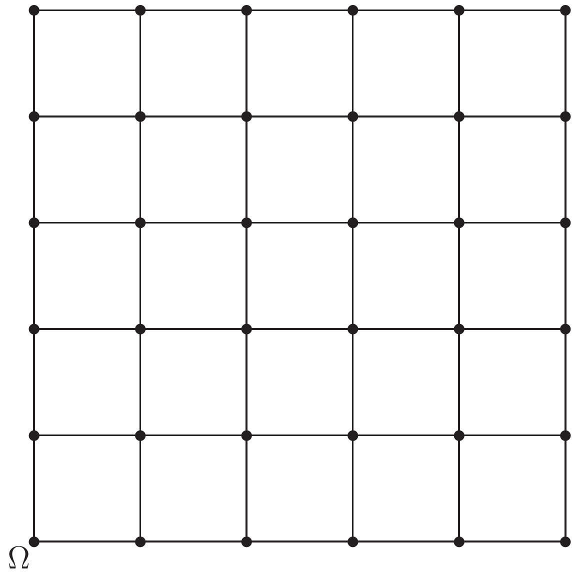


Abbildung 3.1: Äquidistantes Gitter auf $\overline{\Omega} = [0,1] \times [0,1],\ N = 4,\ h = \frac{1}{5}$

Wir setzen nun weiter

$$\begin{aligned} u_{i,j} &:= u(ih, jh), \quad i, j = 1, ..., n \\ f_{i,j} &:= f(ih, jh), \quad i, j = 1, ..., n \\ g_{i,j} &:= g(ih, jh), \quad \text{für Punkte } (ih, jh) \text{ auf dem Rand.} \end{aligned}$$

Wir wollen die *kontinuierliche* Gleichung 3.2.1 in eine *diskrete* Gleichung, d.h. ein endliches Gleichungssystem in *endlich vielen* Unbekannten überführen.

Dazu beachten wir, dass analog zum Differenzenquotienten als Näherung für die zweite Ableitung

$$\begin{aligned} -\frac{\partial^2 u}{\partial x^2}(ih, jh) &\approx \frac{1}{h^2}(-u_{i-1,j} + 2u_{i,j} - u_{i+1,j}) \\ -\frac{\partial^2 u}{\partial y^2}(ih, jh) &\approx \frac{1}{h^2}(-u_{i,j-1} + 2u_{i,j} - u_{i,j+1}), \quad i, j = 1, ..., N \end{aligned}$$

[1]Ein Gitter heißt äquidistant, wenn die horizontalen und vertikalen Abstände zwischen je zwei Gitterpunkten gleich sind und unserem Fall also h

gilt. Durch Einsetzen in die Laplace-Gleichung erhält man dann die *diskrete Laplace-Gleichung*, wobei u_{ij} nur noch eine Näherung für $u(ih, jh)$ ist. Die diskrete Laplace-Gleichung ergibt sich als

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j} \quad i, j = 1, ..., N. \tag{3.2.2}$$

In (3.2.2) sind dabei unbekannt

$$u_{i,j} \quad i, j = 1, ..., N,$$

wegen der Randwertbeziehung sind bekannt

$$\begin{aligned} u_{0,j} = g_{0,j}, u_{N+1,j} = g_{N+1,j}, &\quad j = 1, ..., N \\ u_{i,0} = g_{i,0}, u_{i,N+1} = g_{i,N+1}, &\quad i = 1, ..., N. \end{aligned}$$

Damit ist (3.2.2) ein lineares Gleichungssystem mit $n = N^2$ Gleichungen für n Unbekannte $u_{i,j}$ für $i, j = 1, ..., N$. Die zugehörige Matrix heißt *diskrete Laplace-Matrix*.

Um die diskrete Laplace-Matrix explizit darzustellen, muss man sich für eine lineare Anordnung der Unbekannten $u_{i,j}$ entscheiden z.B. die lexikographische Anordnung bezüglich des Doppelindex (i, j). Einfacher ist es aber, die „geometrische Herkunft“ der Variablen nicht zu ignorieren und die Matrix einfach über ihre Wirkung auf jeden Gitterpunkt als „5-Punkte-Stern“ (s. Abbildung 3.2), darzustellen.

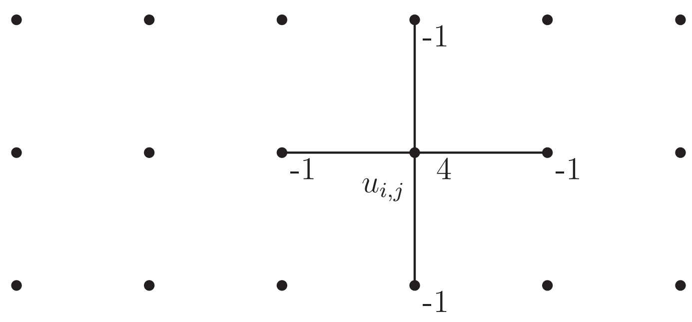


Abbildung 3.2: „5-Punkte-Stern“

3.3 Modellproblem II: Laplace-Gleichung, Finite Elemente

Es sei $\Omega \subseteq \mathbb{R}^2$ ein beliebiges Gebiet.

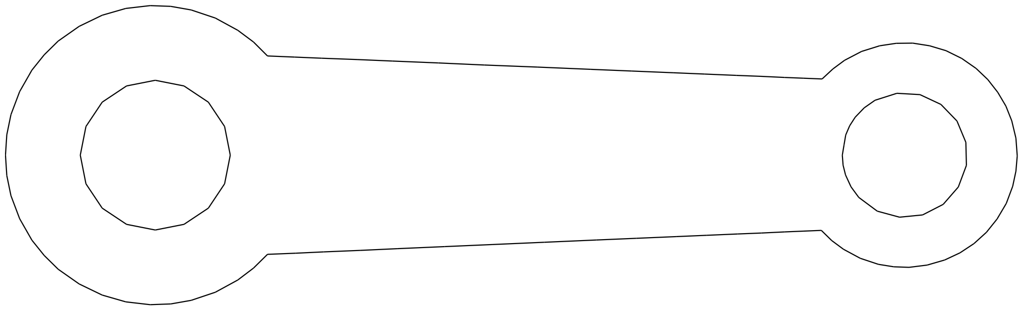


Abbildung 3.3: $\Omega \subseteq \mathbb{R}^2$: Pleuelstange

Wir betrachten wieder die Laplace-Gleichung mit Dirichlet-Randbedingung (vergleiche 3.2)

$$\begin{aligned} -\Delta u(x,y) &= f(x,y), \quad (x,y) \in \Omega \\ u(x,y) &= g(x,y), \quad (x,y) \in \partial\Omega. \end{aligned} \tag{3.3.1}$$

Wenn Ω krumm berandet ist, ist das Verfahren der finiten Differenzen aus dem vorherigen Kapitel nur schlecht einsetzbar. Man kann z.B. mit einem äquidistanten Gitter nicht unbedingt erreichen, dass die Randpunkte des Gitters auf den Rand auf Ω liegen.

Alternative: Finite Elemente

1. Trianguliere Ω (z.B. wie in Abbildung 3.4). Die entstehenden Dreiecke nennen wir Elemente und die Eckpunkte eines Elements Knoten.

2. Verwende jetzt das Skalarprodukt

   $$\langle u, v \rangle := \int_\Omega u \cdot v \; dx \; dy$$

   auf einem geeigneten Hilbertraum H. Es gilt

   $$-\Delta u = f \Rightarrow \langle -\Delta u, v \rangle = \langle f, v \rangle \text{ für alle } v \in H \tag{3.3.2}$$

   Eine Funktion u, welche (3.3.2) erfüllt, heißt *schwache Lösung* (der partiellen Differentialgleichung).

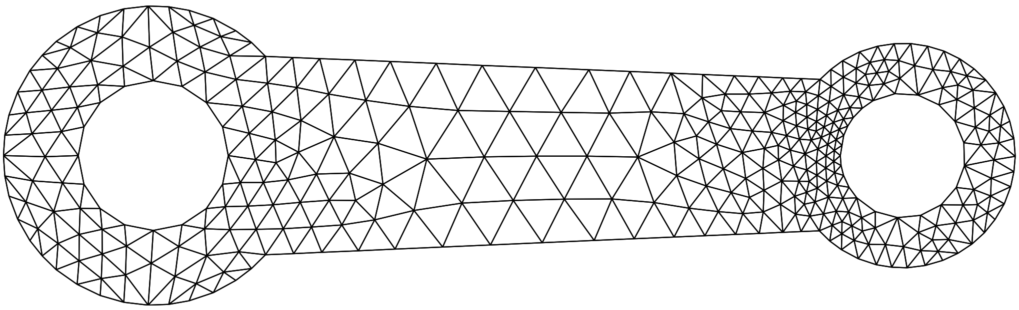


Abbildung 3.4: Triangulierung von Ω um Verformung des Pleuels bei laufendem Motor zu berechnen

Unter Verwendung der Definition des Skalarprodukts und der Greenschen Integralformel ergibt sich

$$\int_\Omega -\Delta u \cdot v \; dx \; dy = \int_\Omega f \cdot v \; dx \; dy$$

$$(3.3.3) \overset{\text{S. v. Green}}{\Longrightarrow} \int_\Omega \nabla u \cdot \nabla v \; dx \; dy - \oint_{\partial\Omega} \frac{\partial u}{\partial n} \cdot v \; ds = \int_\Omega f \cdot v \; dx \; dy$$

3. Verwende nun einen *endlich*-dimensionalen Teilraum von H, welcher von den *lokalen Formfunktionen* φ_i aufgespannt wird. Dabei ist

$$\varphi_i(x) := \begin{cases} 1, & \text{falls } (x,y) = \text{ Knoten } i \\ 0, & \text{falls } (x,y) \text{ ein anderer Knoten oder in einem Element} \\ & \text{enthalten ist, welches nicht den Knoten } i \text{ enthält} \end{cases}$$

und stückweise linear ansonsten. Der Graph einer solchen Funktion ist mit einem „Zeltdach“ vergleichbar (vgl. Abbildung 3.5). Damit sind aber die Gradienten $\nabla\varphi_i$ elementweise konstant. Sie sind von 0 nur auf jenen Elementen, welche den Knoten i enthalten, und 0 auf allen anderen Elementen.

4. Mit den lokalen Formfunktionen erreichen wir eine Diskretisierung, indem wir für die Lösung u näherungsweise ansetzen:

$$u(x,y) = \sum_{j=1}^{n} c_j \varphi_j(x,y).$$

Ist j ein Knoten „am Rand“ $\partial\Omega$, so ist $c_j = g(x_j, y_j)$ zu nehmen, die anderen Koeffizienten c_j sind zunächst noch unbekannt. Ohne

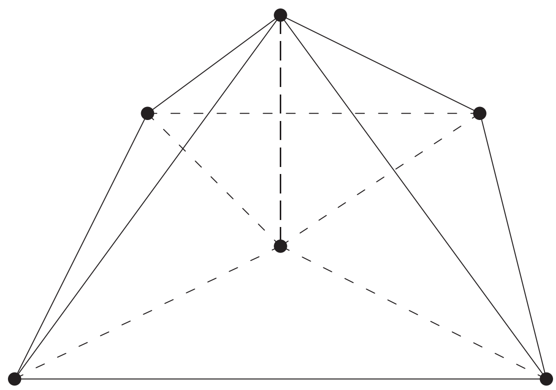


Abbildung 3.5: Eine lokale Formfunktion

Beschränkung der Allgemeinheit können wir nun bei der Nummerierung der Knoten davon ausgehen, dass die Randknoten die Nummern $m+1, ..., n$ tragen. Wir haben also

$$c_j = g(x_j, y_j), \quad j = m+1, ..., n.$$

Erfülle nun die Gleichung (3.3.3) für alle Funktionen $v = \varphi_i(x, y)$ $i = 1, ..., m$, d.h. für alle lokalen Formfunktionen, deren zugrundeliegender Knoten *nicht* auf dem Rand liegt.

Damit wird aber nun Gleichung (3.3.3) zu einem linearen Gleichungssystem

$$\sum_{j=1}^{m} a_{i,j} c_j = b_i, \quad i = 1, ..., m, \tag{3.3.4}$$

wobei die Koeffizienten $a_{i,j}$ bestimmt sind durch

$$\begin{aligned} a_{i,j} &:= \int_\Omega \nabla\varphi_i \nabla\varphi_j \; dx \; dy \\ &= \begin{cases} \int_{e_{i,j}} \nabla\varphi_i \nabla\varphi_j \; dx \; dy & \text{falls } i, j \text{ benachbart} \\ 0 & \text{sonst} \end{cases} . \end{aligned}$$

$e_{i,j}$ bezeichne hierbei die Vereinigung

(a) der beiden Elemente, welche i und j enthalten, falls $i \neq j$,

(b) aller Elemente, welche i enthalten, falls $i = j$.

Die letzte Gleichheit resultiert daraus, dass außerhalb dieser Elemente die Gradienten der lokalen Formfunktionen verschwinden.

Bemerkung: In (3.3.3) gilt speziell für die Formfunktionen

$$\oint_{\partial\Omega} \frac{\partial \varphi_i}{\partial n} \cdot \varphi_j \, ds \;=\; 0, \quad j = 1, ..., m$$

da φ_j auf $\partial\Omega$ verschwindet.

Wir erhalten schließlich noch

$$\begin{aligned} b_i \;:=\; & \int_\Omega f \cdot \varphi_i \, dx\, dy \\ & - \sum_{j=m+1}^{n} c_j \cdot \left(\int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \, dx\, dy + \oint_{\partial\Omega} \frac{\partial \varphi_j}{\partial n} \varphi_i \, dx\, dy \right), \end{aligned}$$

wobei auch hier wieder die Randintegrale im letzten Term verschwinden.

5. Resultat ist nun, dass (3.3.3) ein dünn besetztes lineares Gleichungssystem ergibt. Tatsächlich kann man nun zusätzlich Aussagen über die Approximationsgüte machen etc. Dies ist aber nicht Gegenstand dieser Vorlesung.

Zusammenfassend erhalten wir also

$$\begin{aligned} c_j &= g(x_j, y_j), \quad j = m+1, ..., n \quad \text{(Randknoten)} \\ a_{i,j} &= \begin{cases} \int_{e_{i,j}} \nabla\varphi_i \nabla\varphi_j \, dx\, dy & \text{falls } i, j \text{ benachbart} \\ 0 & \text{sonst} \end{cases} \\ & i, j = 1, ..., m \\ b_i &= \int_\Omega f \cdot \varphi_i \, dx\, dy - \sum_{j=m+1}^{n} c_j \int_\Omega \nabla\varphi_j \nabla\varphi_i \, dx\, dy, \\ & i = 1, ..., m \end{aligned}$$

und das zugehörige lineare Gleichungssystem

$$Ac = b. \tag{3.3.5}$$

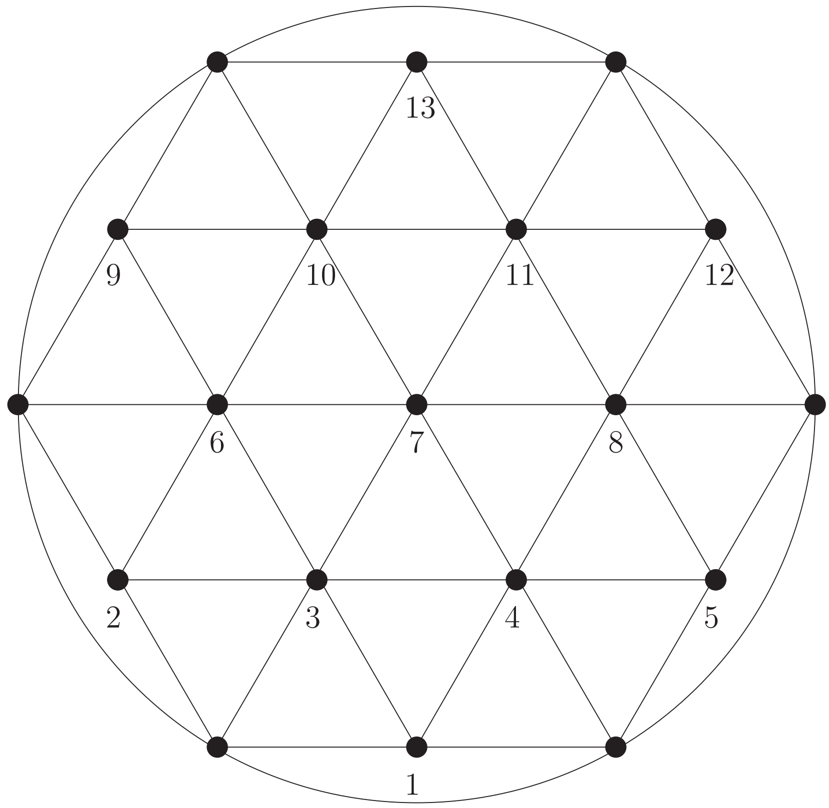


Abbildung 3.6: Triangulierung eines Kreises

Zum Abschluss dieses Abschnitts betrachten wir die Triangulierung eines Kreises aus Abbildung 3.6.

Wir approximieren den Kreis durch ein Sechseck, welches wir anschließend triangulieren. Es ergeben sich 13 Knoten, die nicht auf dem Rand liegen und 24 Elemente. Damit können wir die Matrix A aus Gleichung (3.3.5) bestimmen. Dabei ist die Anzahl der von 0 verschiedenen Elemente in der Zeile i die um 1 erhöhte Anzahl der echten Nachbarn des Knotens i. Genauer ergibt sich

Knoten mit 2 Nachbarn	1,2,5,9,12,13	$\Rightarrow$ 3 Nicht-Nullen
Knoten mit 5 Nachbarn	3,4,6,10,11	$\Rightarrow$ 6 Nicht-Nullen
Knoten mit 6 Nachbarn	7	$\Rightarrow$ 7 Nicht-Nullen

Für das Besetztheitsmuster der Matrix A ergibt sich

$$
\begin{array}{cc}
 & \begin{array}{ccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{array} &
\begin{pmatrix}
* &   & * & * &   &   &   &   &   &   &   &   &   \\
  & * & * &   &   & * &   &   &   &   &   &   &   \\
* & * & * & * &   & * & * &   &   &   &   &   &   \\
* &   & * & * & * &   & * & * &   &   &   &   &   \\
  &   &   & * & * &   &   & * &   &   &   &   &   \\
  & * & * &   &   & * & * &   & * & * &   &   &   \\
  &   & * & * &   & * & * & * &   & * & * &   &   \\
  &   &   & * & * &   & * & * &   &   & * & * &   \\
  &   &   &   &   & * &   &   & * & * &   &   &   \\
  &   &   &   &   & * & * &   & * & * & * &   & * \\
  &   &   &   &   &   & * & * &   & * & * & * & * \\
  &   &   &   &   &   &   &   & * &   & * & * &   \\
  &   &   &   &   &   &   &   &   & * & * &   & *
\end{pmatrix}
\end{array}
$$

Eine so entstehende Matrix ist stets symmetrisch und positiv definit (wenn der zugehörige partielle Differentialoperator selbstadjungiert und elliptisch ist).

Es ist bekannt, dass man einer symmmetrischen Matrix (wenn man sie als Adjazenzmatrix auffasst), einen ungerichteten Graphen zuordnen kann. Wir haben gerade eben umgekehrt einem ungerichteten Graphen wieder eine Matrix zugeordnet. Dieses Phänomen wollen wir genauer untersuchen, dazu benötigen wir aber einige weitere Grundlagen.

3.4 Grundlagen aus der Graphentheorie

Wir wollen zunächst an die grundlegenden Begriffe erinnern.

3.4.1 Definition
Ein Graph $G = (V, E)$ ist ein Paar bestehend aus einer endlichen Menge $V = \{v_1, ..., v_n\}$[2], den **Knoten** (engl.: vertices), und einer Menge $E \subseteq V \times V$ den **Kanten** (engl.: edges).

Bemerkung: Wir betrachten ausschließlich ungerichtete Graphen.

3.4.2 Beispiel
Einfache Beispiele von Graphen haben wir als Topologien für Parallelrechner bereits im Abschnitt 1.2 kennengelernt.

1. Für $V = \{1, 2, 3, 4\}$ und $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}\}$[3] ergibt sich der Graph (ein „Viereck“) aus Abbildung 3.7

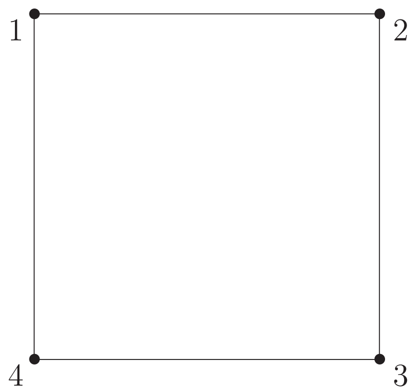


Abbildung 3.7: Einfacher Graph

2. Ein für uns interessanteres Beispiel ist ein Graph bestehend aus den Knoten und Kanten einer Triangulierung.

3.4.3 Definition
Es sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix; der zu A gehörende Graph $G(A) = (V, E)$ ist gegeben durch

$$\begin{aligned} V &:= \{1, 2, ..., n\} \\ E &:= \{\{i, j\} \in V \times V : \ a_{i,j} \neq 0\} \end{aligned}$$

[2] wir werden immer $V = \{1, ..., n\}$ betrachten

[3] Zur Erinnerung: Man verwendet die Notation $\{1, 2\}$ um deutlich zu machen, dass die Kante zwischen den Knoten 1 und 2 *nicht* gerichtet ist

Bemerkung: E ist wohldefiniert, denn weil A symmetrisch ist, gilt $a_{i,j} \neq 0 \iff a_{j,i} \neq 0$.

3.4.4 Beispiel

1. Es sei A eine Tridiagonalmatrix, d.h. eine Matrix der Form

$$\begin{pmatrix} * & * & & & & \\ * & * & * & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & * \\ & & & & * & * \end{pmatrix}$$

   dann ist der zu A gehörige Graph eine Kette,

$G(A)$

Abbildung 3.8: Graph einer Tridiagonalmatrix

2. Ist A die diskrete Laplacematrix aus Abschnitt 3.2, so ergibt sich als Graph ein Gitter gemäß Abbildung 3.9.

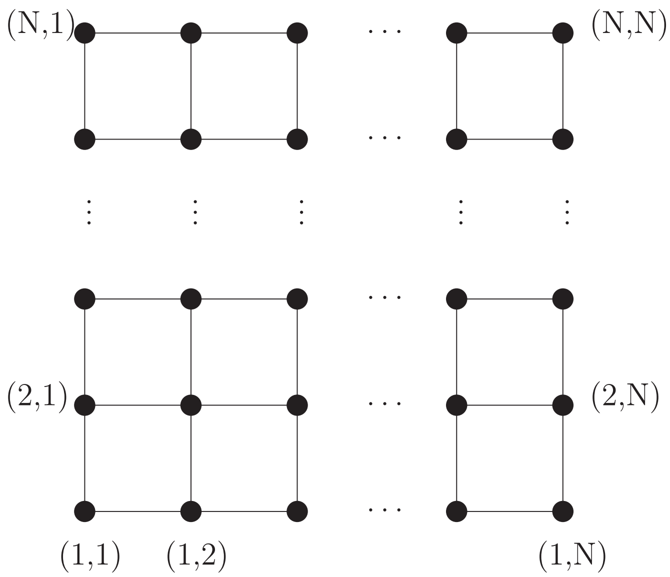


Abbildung 3.9: Graph der diskreten Laplacematrix

3. Ein Graph heißt **eben** oder **planar**, wenn man ihn in die Ebene zeichnen kann, ohne dass sich zwei Kanten überschneiden.

   Ist A die Matrix für die Gleichung 3.3.5 aus der Finite-Elemente Diskretisierung, dann ist $G(A)$ die Triangulierung und der Graph planar. (Bei anderen Wahlen für die lokalen Formfunktionen können auch nichtplanare Graphen entstehen.)

3.4.5 Definition
Es sei $G = (V, E)$ ein Graph. Der **Grad** $d(v)$ eines Knotens $v \in V$ ist die Anzahl der mit v **inzidenten** Kanten, d.h.

$$d(v) := |\{e \in E : \ e = \{v, w\}, \ w \in V, \ w \neq v\}|.$$

(Eventuell vorkommende Schleifen $\{v, v\} \in E$ werden also nicht berücksichtigt.)

3.4.6 Definition
Es sei $G = (V, E)$ ein Graph mit $V = \{1, ..., n\}$. Die zu G gehörige **Laplace-Matrix** $L(G) = (\ell_{i,j})$ ist definiert durch

$$\ell_{i,j} := \begin{cases} -1 & i \neq j \text{ und } \{i, j\} \in E, \\ d(i) & i = j, \\ 0 & \text{sonst.} \end{cases}$$

3.4.7 Satz
Die Laplace-Matrix $L(G)$ besitzt folgende Eigenschaften:

1. $L(G)$ ist symmetrisch,
2. $L(G)$ ist positiv semidefinit,
3. $L(G)$ besitzt den Eigenwert 0 mit Eigenvektor $(1, 1, ..., 1)^T$.

Beweis:

1. Trivial, da G ungerichtet ist.
2. Zu zeigen ist dazu, dass alle Eigenwerte von $L(G)$ nicht negativ sind. Ist λ ein Eigenwert von $L(G)$ so folgt nach dem Satz von Gerschgorin

$$\lambda \in \bigcup_{j=1}^{n} C_j, \quad C_j = \left\{ z \in \mathbb{C} : \ |z - \ell_{j,j}| \leq \sum_{j=1, \ j \neq i}^{n} |\ell_{i,j}| \right\}.$$

Weil L symmetrisch ist, gilt

$$\lambda \in \bigcup_{j=1}^{n} (C_j \cap \mathbb{R})$$

und wegen

$$C_j \cap \mathbb{R} = [0, 2d_j],$$

ergibt sich sogar

$$\lambda \in [0, \max_{j=1,\dots,n} \{d_j\}].$$

3. Einfaches Nachrechnen zeigt

$$L(G) \cdot (1, 1, ..., 1)^T = 0.$$

□

3.4.8 Definition
Ein Graph $G = (V, E)$ heißt **zusammenhängend**, falls für jede Wahl von zwei Knoten $v, w \in V$, $v \neq w$ ein Kantenzug $(v = v_0, v_1, ..., v_\ell = w)$ existiert. (Kantenzug bedeutet $\{v_i, v_{i+1}\} \in E, i = 0, ..., \ell - 1$.)

Falls G nicht zusammenhängend ist, so besteht G aus mehreren **Zusammenhangskomponenten**, den maximalen zusammenhängenden Teilgraphen.

3.4.9 Definition
Ein Graph $G' = (V', E')$ heißt **Teilgraph** von $G = (V, E)$ genau dann, wenn $V' \subseteq V, E' \subseteq E$ und $E' \subseteq V' \times V'$ gilt.

3.4.10 Definition
Für $n \geq 2$ heißt eine symmetrische Matrix $A \in \mathbb{R}^{n \times n}$ **irreduzibel**, falls der zugehörige Graph $G(A)$ zusammenhängend ist. Andernfalls heißt A **reduzibel**.

Falls A reduzibel ist, so existieren Mengen V_1, V_2 mit $V_1 \neq \emptyset$, $V_2 \neq \emptyset$, $V_1 \cup V_2 = \{1, ..., n\}$, so dass gilt:

$$PAP^T = \begin{pmatrix} \boxed{} & 0 \\ 0 & \boxed{} \end{pmatrix}.$$

Dabei ist P eine Permutationsmatrix, welche die Knoten von V_1 an den Anfang und die Knoten von V_2 an das Ende permutiert.

Für A reduzibel zerfällt also das ursprüngliche Gleichungssystem in zwei *unabhängige* Gleichungssysteme kleinerer Größe!

Eine Verallgemeinerung für nicht symmetrische Matrizen ist die folgende Definition, die wir nur der Vollständigkeit halber angeben, denn wir setzen in diesem Kapitel A immer als symmetrisch voraus.

3.4.11 Definition
Für $n \geq 2$ heißt eine Matrix $A \in \mathbb{R}^{n \times n}$ **reduzibel**, falls es eine Permutationsmatrix P gibt mit

$$PAP^T = \left(\begin{array}{c|c} & 0 \\ \hline & \end{array}\right).$$

Andernfalls heißt A **irreduzibel**.

3.4.12 Definition
Es sei $G = (V, E)$ ein zusammenhängender Graph. Dann heißt $S \subseteq E$ ein **Schnitt** für G, falls der Graph $G' = (V, E \setminus S)$ nicht zusammenhängend ist.

3.4.13 Definition
Es sei $G = (V, E)$ und $V' \subseteq V$. Der von V' **induzierte Teilgraph** $G|_{V'} := V', E')$ besitzt die Kanten

$$E' := \{e \in E : \ e = \{i, j\}, \ i, j \in V'\}.$$

3.4.14 Definition
Es sei $G = (V, E)$ ein Graph und $V_1, ..., V_p$ eine Partition von V. Dann ist der **Quotientengraph** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ gegeben durch

$$\begin{array}{rcl} \mathcal{V} & := & \{1, ..., p\} \\ \mathcal{E} & := & \{\{\ell, m\} \subseteq \mathcal{V} \times \mathcal{V} : \ \exists\, i \in V_\ell, j \in V_m \text{ mit } \{i, j\} \in E\}. \end{array}$$

Beispiel: Für das 4×4-Gitter aus Abbildung 3.10 ist die Menge $S = \{\{5,9\},\{5,10\},\{6,10\},\{6,11\}\{7,11\},\{7,12\},\{8,12\}\}$ ein Schnitt. Der Quotientengraph bezüglich $V_1 = \{1,...,8\}$ und $V_2 = \{9,...16\}$ ist eine Kette der Länge 2.

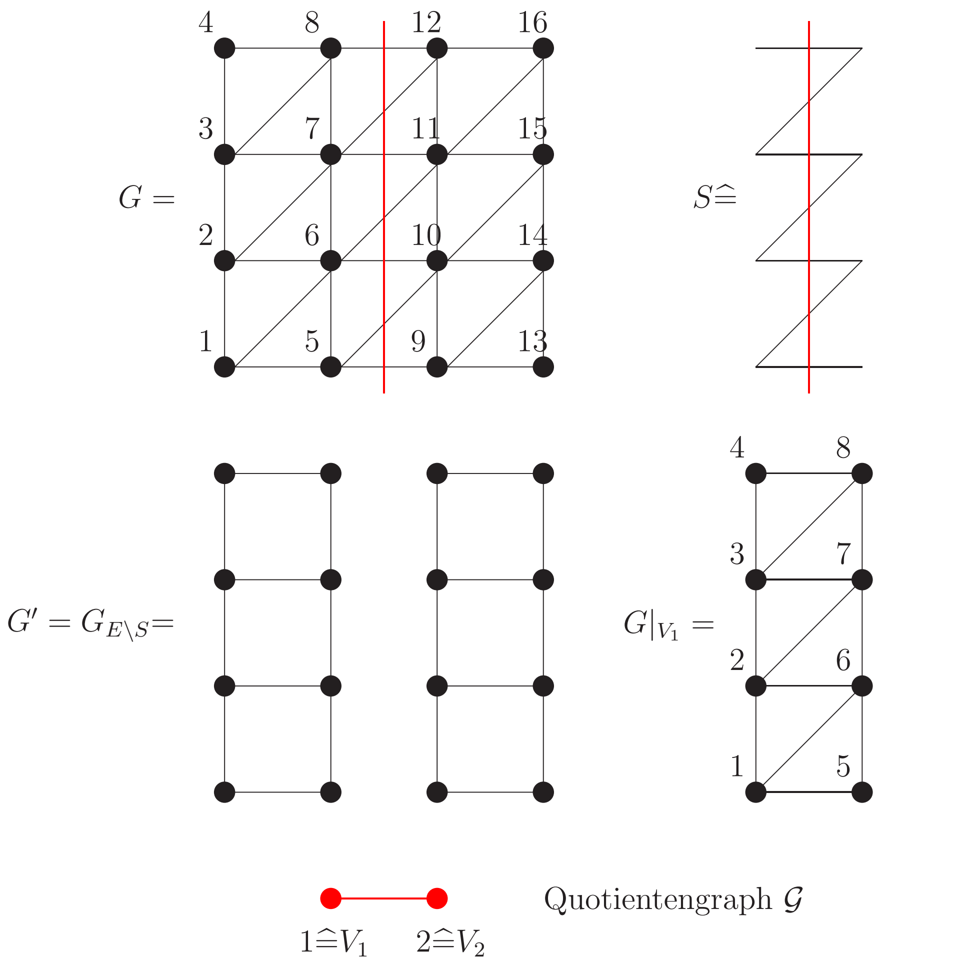


Abbildung 3.10: Beispiel für Schnitt, Teil- und Quotientengraphen

3.5 Parallele Matrix-Vektor-Multiplikation

Gegeben ist eine dünn besetzte, symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ mit

$$V = V_1 \cup V_2 \cup \ldots \cup V_p \text{ Partition von } V = \{1, ..., n\}.$$

Der Prozessor P_ℓ kenne $x_i,\ i \in V_\ell$, und berechne $y_i = (Ax)_i, i \in V_\ell$. Dann werden einige x_j aus „Nachbarprozessoren" benötigt, nämlich von genau den Nachbarn im Quotientengraphen $\mathcal{G}$ bezüglich $V_1, ..., V_p$ des Graphen $G = G(A)$.

3.5.1 Algorithmus (Parallele Matrix-Vektor-Multiplikation)

if me=P_ℓ **then**
  for all m mit $\{m, \ell\} \in \mathcal{E}$ **do**
    send($x_i :\ \exists\, i \in V_\ell,\ j \in V_m$ mit $\{j, i\} \in E$) to P_m
  end for
  for all m mit $\{\ell, m\} \in \mathcal{E}$ **do**
    receive($x_j :\ \exists\, i \in V_\ell,\ j \in V_m$ mit $\{i, j\} \in E$) from P_m
  end for
  for $i \in V_\ell$ **do**
    $y_i = \sum\limits_{\{i,j\}\in E} a_{i,j} x_j$
  end for
end if

Es sei $S := \{\{i, j\} \in E :\ i, j$ liegen nicht im selben $V_\ell\}$. $S \subseteq E$ ist der von der Partition $V_1, ..., V_p$ *induzierte* Schnitt. $(V, E \setminus S)$ besitzt mindestens p Zusammenhangskomponenten.

$|S|$ nennen wir ab jetzt die **Schnittgröße**.

In Algorithmus 3.5.1 ergibt sich als globaler Aufwand:

$$\begin{aligned} \text{Kommunikation:}\quad & 2 \cdot |S| \\ \text{Rechnung:}\quad & |E|. \end{aligned}$$

Ziel: Gleiche Verteilung der Rechenlast und geringe Kommunikation.

Rechenlast für Prozessor P_ℓ:

$$\sum_{i\in V_\ell} d(i) \text{ Additionen}$$
$$\sum_{i\in V_\ell} d(i) + 1 \text{ Multiplikationen.}$$

Die Rechenlast ist also „im Wesentlichen“ proportional zu $\sum\limits_{i \in V_\ell} d(i)$. Wir betrachten deshalb die

Aufgabe „Graph-Partitionierung 1“:

Gegeben: Graph $G = (V, E)$ und eine Partitionsgröße p

Gesucht: Partition $V = V_1 \cup ... \cup V_p$ mit

(P.1) $\sum\limits_{i \in V_\ell} d(i)$ ist gleich groß für $\ell = 1, ..., p$

(P.2) $|S|$ ist minimal für den induzierten Schnitt S

Anmerkung: Diese Aufgabe ist NP-vollständig, d.h. man wird wahrscheinlich nie einen „realistischen“ Algorithmus zur exakten Lösung finden. Darum verwendet man Heuristiken, von denen wir einige im Folgenden diskutieren wollen.

Häufig ist $d(i)$ fast gleich groß für alle i. Dann vereinfacht sich die Aufgabe „Graph-Partitionierung 1“ zu

Aufgabe „Graph-Partitionierung 2“:

Gegeben: Graph $G = (V, E)$ und eine Partitionsgröße p

Gesucht: Partition $V = V_1 \cup ... \cup V_p$ mit

(P.1’) $|V_\ell|$ ist gleich groß für $\ell = 1, ..., p$

(P.2) $|S|$ ist minimal für den induzierten Schnitt S

Beide Aufgaben sind eher prototypisch zu verstehen. Sie können gar nicht gelöst werden, wenn $\sum_{i=1}^n d(i)$ bzw. n nicht durch p teilbar sind ....

Auch diese Aufgabe ist noch NP-vollständig.

3.6 Graph-Partitionierung: Spektrale Bisektion

Die spektrale Bisektion ist ein Verfahren zur approximativen Lösung der Aufgabe „Graph-Partitionierung 2“ aus Abschnitt 3.5, wobei (P.1') exakt (für $p = 2$ und n gerade), (P.2) approximativ gelöst wird.

Gegegeben ist eine symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ mit zugehöriger Laplace-Matrix $L = L(G(A))$.

Wir benötigen zwei Resultate aus der linearen Algebra, nämlich den Satz von Perron-Frobenius und den Cauchyschen Trennungssatz

3.6.1 Definition
Es seien $A, B \in \mathbb{R}^{n\times n}$. Dann definieren wir

$$A < (\leq) B :\Longleftrightarrow a_{i,j} < (\leq) b_{i,j} \quad \forall\ i,j = 1,...,n.$$

Die Relationen $\leq, <$ sind also komponentenweise zu verstehen. (Damit ist $\leq$ nur noch eine Halbordnung.)

3.6.2 Definition
Der **Spektralradius** $\rho(A)$ einer Matrix $A \in \mathbb{R}^{n\times n}$ ist definiert als

$$\rho(A) := \max\{|\lambda| : \ \lambda \text{ ist Eigenwert von } A\}.$$

3.6.3 Lemma
Es sei $A \geq 0$ und $s > \rho(A)$. Dann existiert

$$B := (s \cdot I - A)^{-1}$$

und es gilt

$$B \geq 0.$$

Beweis: Die Matrix $sI - A$ besitzt die Eigenwerte

$$s - \lambda_i, \quad \lambda_i \text{ Eigenwert von } A.$$

Es ist $\mathrm{Re}(s - \lambda_i) > 0$, da $|\lambda_i| \leq \rho(A) < s$. Also ist ist die Matrix $sI - A$ regulär.

Es gilt

$$\left(I - \frac{1}{s}A\right)\left(I + \frac{1}{s}A + \ldots + \left(\frac{1}{s}A\right)^n\right) = I - \left(\frac{1}{s}A\right)^{n+1}$$

$$\Longleftrightarrow \qquad I + \frac{1}{s}A + \ldots \left(\frac{1}{s}A\right)^n = \left(I - \left(\frac{1}{s}A\right)^{n+1}\right) \cdot \left(I - \frac{1}{s}A\right)^{-1}.$$

Es ist aber $\rho\left(\frac{1}{s}A\right) = \frac{1}{s} \cdot \rho(A) < 1$, so dass $\lim\limits_{n\to\infty} (\frac{1}{s}A)^n = 0$ gilt (s. Numerik I und LA II). Also konvergiert die Neumannsche Reihe und es gilt

$$\sum_{j=0}^{\infty} \left(\frac{1}{s}A\right)^j = \left(I - \frac{1}{s}A\right)^{-1}.$$

Damit ergibt sich

$$(sI - A)^{-1} = \left(s \cdot \left(I - \frac{1}{s}A\right)\right)^{-1} = \frac{1}{s}\sum_{j=0}^{\infty} \underbrace{\left(\frac{1}{s}A\right)^j}_{\geq 0} \geq 0.$$

□

3.6.4 Satz (Perron-Frobenius)
Es sei $A \in \mathbb{R}^{n\times n}$ irreduzibel und $A \geq 0$. Dann gilt

1. $\rho(A)$ ist einfacher Eigenwert von A.
2. Es existiert ein positiver Eigenvektor $u > 0$ zum Eigenwert $\rho(A)$,

$$Au = \rho(A)u,$$

(der wegen 1. bis auf Multiplikation mit Skalaren eindeutig ist).

Beweis: Siehe Numerik oder „Varga: Matrix Iterative Analysis“. □

3.6.5 Definition
Sei $A \in \mathbb{R}^{n\times n}$ und $M := \{i_1, ..., i_m\} \subseteq \{1, ..., n\}$. Dann ist die zu M gehörende **Haupt-Untermatrix** $A(M, M) \in \mathbb{R}^{m\times m}$ definiert durch

$$(A(M, M))_{\ell,m} := a_{i_\ell,i_m}$$

(Hauptuntermatrizen entstehen also durch Streichen der Spalten und Zeilen, welche in M nicht vorkommen).

3.6.6 Definition
Es sei $A \in \mathbb{R}^{n \times n}$ symmetrisch mit den Eigenwerten $\lambda_n \leq \lambda_{n-1} \leq ... \leq \lambda_1$ (mehrfache Eigenwerte werden auch mehrfach gezählt). Dann ist die **Signatur** von A gegeben durch

$$\begin{aligned} p &= |\{i : \lambda_i > 0\}| \\ q &= |\{j : \lambda_j < 0\}| \end{aligned}$$

3.6.7 Satz (Cauchyscher Trennungssatz)
Es sei $A \in \mathbb{R}^{n \times n}$ symmetrisch, $M \subseteq \{1, ..., n\}$ mit $|M| = n - 1$. Setze $A' = A(M, M)$ und es bezeichne $\lambda_n \leq ... \leq \lambda_1$ die Eigenwerte von A bzw. $\lambda'_{n-1} \leq ... \leq \lambda'_1$ die Eigenwerte von A'. Dann gilt

$$\lambda_n \leq \lambda'_{n-1} \leq \lambda_{n-1} \leq ... \leq \lambda'_1 \leq \lambda_1.$$

Beweis: Siehe „Parlett: The Symmetric Eigenvalue Problem“. □

3.6.8 Korollar
Es sei $A \in \mathbb{R}^{n \times n}$ symmetrisch, $M \subseteq \{1, ..., n\}$ mit $|M| = m$. Desweiteren sei $A' = A(M, M)$ und es seien (p, q) bzw. (p', q') die Signaturen von A bzw. A'. Dann gilt

$$p' \leq p, \quad q' \leq q, \quad (\text{kurz: } (p', q') \leq (p, q)).$$

Beweis: Für $m = n - 1$ folgt die Behauptung direkt aus Satz 3.6.7. Für $m < n - 1$ folgt mit Mengen

$$M_0 \supseteq M_1 \supseteq M_2 \supseteq ... \supseteq M_{n-m} = M,$$

wobei $|M_i| = n - i$ sei, dann induktiv für die jeweiligen Signaturen

$$(p, q) = (p_0, q_0) \geq (p_1, q_1) \geq (p_2, q_2) \geq ... \geq (p_{n-m}, q_{n-m}) = (p', q').$$

□

3.6.9 Korollar
Es sei $A \in \mathbb{R}^{n \times n}$ symmetrisch. Dann besitzt jede Hauptuntermatrix von $\lambda_s I - A$ für $s = 1, ..., n$ höchstens $s - 1$ negative Eigenwerte.

Beweis: Die Matrix $\lambda_s I - A$ hat höchstens $s - 1$ negative Eigenwerte, der Rest folgt dann aus Korollar 3.6.8. □

Nach diesen Vorbereitungen formulieren wir nun das zentrale Resultat, auf welchem die spektrale Bisektion als Graph-Partitionierungsverfahren aufbaut.

3.6.10 Satz
Es sei $A \in \mathbb{R}^{n \times n}$ nichtnegativ, irreduzibel, symmetrisch und besitze die Eigenwerte $\lambda_n \leq ... \leq \lambda_1$. Es sei $v \in \mathbb{R}^n$ ein Vektor mit $Av \geq \lambda_2 v$ und $M := \{i \in \{1, ..., n\} : v_i \geq 0\}$. Dann ist $M \neq \emptyset$ und der Graph $G(A(M, M))$ ist zusammenhängend.

Beweis: Angenommen, es gälte $M = \emptyset$, d.h. $v < 0$. Dann gilt für $z = -v$

$$z > 0, \quad Az \leq \lambda_2 z. \tag{3.6.1}$$

Nach dem Satz von Perron-Frobenius existiert ein $u > 0$, so dass $Au = \lambda_1 u$. Damit folgt aber

$$u^T Az = \lambda_1 u^T z > \lambda_2 u^T z \quad (\text{da } \lambda_1 > \lambda_2).$$

Wegen (3.6.1) folgt aber auch

$$u^T Az \leq \lambda_2 u^T z.$$

Also ist $M \neq \emptyset$.

Im Fall $M = \{1, ..., n\}$ ist der Satz richtig. Ab jetzt sei also

$$\emptyset \neq M \subsetneq \{1, ..., n\}.$$

Angenommen der Graph $G(A(M, M))$ besäße $r \geq 2$ Zusammenhangskomponenten. Nach Umordnung der Zeilen und Spalten können wir ohne Einschränkung annehmen, dass $M = \{1, ..., m\}, \ m < n$ und damit

$$A = \begin{pmatrix} A_{1,1} & & & 0 & A_{1,r+1} \\ & \ddots & & & \vdots \\ & & \ddots & & \vdots \\ & 0 & & \ddots & \vdots \\ & & & A_{r,r} & A_{r,r+1} \\ A_{1,r+1}^T & \dots & \dots & \dots \ A_{r,r+1}^T & A_{r+1,r+1} \end{pmatrix}$$

gilt, wobei $A_{j,j} \in \mathbb{R}^{n_j \times n_j}, \ \sum_{j=1}^{r} n_j = m$ ist. Wir partitionieren v entsprechend, d.h. $v = (v_1, v_2, ..., v_r, v_{r+1}), \ v_j \in \mathbb{R}^{n_j}$. Dann gilt

$$v_j \geq 0, \quad j = 1, ..., r, \ v_{r+1} < 0.$$

Nach Voraussetzung gilt $(\lambda_2 I - A_{j,j})v_j - A_{j,r+1}v_{r+1} \leq 0$, also

$$(\lambda_2 I - A_{j,j})v_j \leq \underbrace{A_{j,r+1}}_{\geq 0}\underbrace{v_{r+1}}_{<0} \leq 0, \quad j = 1, ..., r. \tag{3.6.2}$$

Nach Korollar 3.6.9 hat die Hauptuntermatrix

$$\begin{bmatrix} \lambda_2 I - A_{1,1} & & \\ & \ddots & \\ & & \lambda_2 I - A_{r,r} \end{bmatrix} =: A'$$

von $\lambda_2 I - A$ höchstens einen negativen Eigenwert. Die Eigenwerte von A' sind die Eigenwert der Matrizen $\lambda_2 I - A_{j,j}, \quad j = 1, ..., r$. Es existiert also ein $1 \leq j_0 \leq r$, so dass $\lambda_2 I - A_{j_0,j_0}$ nur nicht-negative Eigenwerte besitzt. Ohne Einschränkung sei $j_0 = 1$ (andernfalls müssen in A noch einmal die Zeilen und Spalten passend vertauscht werden). Wir unterscheiden zwei Fälle:

1. Alle Eigenwerte von $\lambda_2 I - A_{1,1}$ sind positiv, dann ist $\rho(A_{1,1}) < \lambda_2$ und damit nach Lemma 3.6.3

   $$(\lambda_2 I - A_{1,1})^{-1} \geq 0.$$

   Aus (3.6.2) folgt somit für $j = 1$

   $$v_1 \leq \underbrace{(\lambda_2 I - A_{1,1})^{-1} A_{1,r+1}}_{\geq 0} \cdot \underbrace{v_{r+1}}_{<0} \leq 0,$$

   also $v_1 = 0$ und damit $A_{1,r+1}v_{r+1} = 0$. Da $v_{r+1} < 0$ ist und $A_{1,r+1} \geq 0$, folgt sogar $A_{1,r+1} = 0$. Dies steht aber im Widerspruch zur Irreduzibilität von A!

2. Mindestens ein Eigenwert von $\lambda_2 I - A_{1,1}$ verschwindet, also $\rho(A_{1,1}) = \lambda_2$. Da $A_{1,1}$ irreduzibel ist, ist λ_2 einfacher Eigenwert und es existiert nach dem Satz von Perron-Frobenius ein u mit

   $$A_{1,1}u = \lambda_2 u \Longleftrightarrow u^T(\lambda_2 I - A_{1,1}) = 0 \text{ beachte: } A_{1,1} = A_{1,1}^T).$$

   Damit gilt dann

   $$\underbrace{u^T}_{>0}\underbrace{(\lambda_2 I - A_{1,1})v_1}_{\leq 0, \text{ nach } (3.6.2)} = 0,$$

also $(\lambda_2 I - A_{1,1})v_1 = 0$. Mit (3.6.2) folgt

$$0 \leq A_{1,r+1}v_{r+1} \leq 0,$$

also $A_{1,r+1} = 0$. Dies ist wiederum ein Widerspruch zur Irreduzibiltät von A.

Unsere Annahme, $G(A(M,M))$ besitze $r \geq 2$ Zusammenhangskomponenten war also falsch. □

Bemerkung: Die Aussage des Satzes bleibt auch richtig, wenn man

$$M := \{i : \ v_i \leq 0\}$$

setzt. In den meisten Fällen wird sie auch richtig sein, wenn man

$$M := \{i : \ v_i < 0\}$$

setzt, das Komplement von $\{i : \ v_i \geq 0\}$ in $\{1, ..., n\}$.

Wie findet man einen Vektor v wie in Satz 3.6.10? Sei dazu $u > 0$ Eigenvektor zum Eigenwert λ_1 (solch ein Vektor existiert nach Perron-Frobenius), w ein Eigenvektor zum Eigenwert λ_2 und

$$v_\alpha = \alpha u + w, \quad \alpha \geq 0. \tag{3.6.3}$$

Jedes v_α erfüllt damit

$$\begin{aligned} Av_\alpha &= \underbrace{\alpha\lambda_1 u}_{\geq 0} \pm \lambda_2 w \\ &\geq \alpha\lambda_2 u \pm \lambda_2 w \\ &= \lambda_2(\alpha u \pm w) \\ &= \lambda_2 v_\alpha. \end{aligned}$$

3.6.11 Korollar
Für $\alpha > 0$ sei

$$\begin{aligned} V_1 &:= \{i : \ w_i \geq -\alpha u_i\} \\ V_1' &:= \{i : \ w_i \leq +\alpha u_i\}. \end{aligned}$$

Dann sind die beiden Graphen $G(A)|_{V_1}$ und $G(A)|_{V_1'}$ zusammenhängend.

Wir wenden unsere Resultate jetzt auf die Graph-Partitionierungsaufgabe 2 an.

Dazu sei A irreduzibel, also $G(A)$ zusammenhängend und $L = L(G(A))$. Für $\sigma > \max\limits_{i=1,\dots,n} d(i)$ ist dann

$$B_\sigma := \sigma I - L \geq 0$$

und irreduzibel.

Es seien $0 = \lambda_1 < \lambda_2 \leq \dots \leq \lambda_n$ die Eigenwerte von L, dann sind

$$\sigma - \lambda_1 > \sigma - \lambda_2 \geq \dots \geq \sigma - \lambda_n$$

die Eigenwerte von B_σ. Zu

- $\sigma - \lambda_1$ gehört ein Eigenvektor $e = (1, 1, \dots, 1)^T$
- $\sigma - \lambda_2$ gehört ein Eigenvektor f mit $Lf = \lambda_2 f$, der sogenannte **Fiedler-Vektor**.

Bei Anwendung von Satz 3.6.10 auf $\sigma I - L$ ist in (3.6.3) also $w = f$, $u = (1, 1, \dots, 1)^T$.

Jetzt sei α in (3.6.3) so gewählt, dass $V_1 = \{i : \ (f + \alpha e)_i \geq 0\} = \{i : f_i + \alpha \geq 0\}$ möglichst genau $\frac{n}{2}$ Elemente besitzt.

3.6.12 Definition
Es sei $(\beta_1, \dots, \beta_n)$ eine endliche Folge aus $\mathbb{R}$. Ein **Median** dieser Folge ist ein Element β_i mit der Eigenschaft

$$|\{j : \ \beta_j \leq \beta_i\}| = \left\lceil \frac{n}{2} \right\rceil .$$

3.6.13 Algorithmus (Spektrale Bisketion)

Require: Zusammenhängender Graph $G = (V, E)$, z.B. $L(G(A))$ mit irreduziblem A
Bestimme einen Fiedler-Vektor f von $L(G)$
Bestimme Median F von $f = (f_1, f_2, \dots, f_n)$
falls $F < 0$
  setze $V_1 = \{i : \ f_i \geq F\}$, $V_2 = \{i : \ f_i < F\}$
falls $F \geq 0$
  setze $V_1 = \{i : \ f_i \leq F\}$, $V_2 = \{i : \ f_i > F\}$

Bemerkung:

1. Auf Grund von Satz 3.6.10 ist $G|_{V_1}$ zusammenhängend. Wird der Median F von mehreren Komponenten von f angenommen, so ist $|V_1| > |V_2|$. Sollen die beiden Mengen gleich groß sein, muss man einige dieser Element von V_1 nach V_2 transferieren.

2. Der Fiedler-Vektor f muss negative Elemente besitzen, da f Eigenvektor zu λ_2 ist und $e = (1, 1, ..., 1)^T$ Eigenvektor zu λ_1, aber zwei Eigenvektoren zu verschiedenen Eigenwerten senkrecht aufeinander stehen, d.h.
$$\langle e, f\rangle = \sum_{j=1}^{n} f_j = 0.$$
3. Algorithmus 3.6.13 löst weder die Aufgabe „Graphpartitionierung 1“ noch „Graphpartitionierung 2“. Heuristisch kann man aber erwarten, dass er kleine Schnittgrößen erzeugt, denn $G|_{V_1}$ ist zusammenhängend.

Die spektrale Bisektion kann zusätzlich durch eine formale Beschreibung der „Graphpartitionierungs-Aufgabe 2“ motiviert werden:

Gegeben: Es sei $G = (V, E)$ zusammenhängend mit $|V| = n$
$$\begin{aligned} f : \mathbb{Z}^n &\rightarrow \mathbb{N} \\ x = (x_1, ..., x_n) &\mapsto \frac{1}{4} \sum_{(i,j)\in E} (x_i - x_j)^2 \end{aligned}$$

Gesucht: $x \in \mathbb{Z}^n$, welches $f(x)$ unter den Nebenbedingung

1. $x_i \neq 0,\ i = 1, ..., n$
2. $\langle x, x\rangle = n$ (aus 1. und 2. folgt $x_i \in \{\pm 1\}$)
3. $\langle x, e\rangle = 0$ und $e = (1, ..., 1)$ ($\Longleftrightarrow$ x hat gleich viele positive wie negative Komponenten).

minimiert.

Setzt man dann $V_1 = \{i :\ x_i = +1\}, V_2 = \{i :\ x_i = -1\}$ (wobei x Lösung der obigen Minimierungsaufgabe ist), so ist $|V_1| = |V_2|$ und $f(x)$ genau die Schnittgröße, die also minimal ist.

Die gegebene Aufgabe ist *diskret* ($x \in \mathbb{Z}^n$) und deshalb schwierig zu lösen. Deshalb geht man zu einer einfacheren, kontinuierlichen Aufgabenstellung über. Dazu verwenden wir zunächst

3.6.14 Lemma

$x \in \mathbb{Z}^n$ löst die obige Miminierungsaufgabe genau dann, wenn es auch die Funktion $\ell(x) = \frac{1}{4}x^T Lx$ mit $L = L(G)$ unter den selben Nebenbedingungen löst, d.h. x löst

$$(3.6.4) \qquad \left\{ \begin{array}{rcl} \multicolumn{3}{l}{\text{minimiere } \ell(x) = \frac{1}{4}x^T Lx :} \\ x &\in& \mathbb{Z}^n \\ x_i &\neq& 0,\ i = 1, \ldots, n \\ \langle x, x\rangle &=& n \\ \langle x, e\rangle &=& 0. \end{array} \right.$$

Beweis: Weil aus den Nebenbedingungen $x_i = \pm 1$ folgt, gilt

$$\begin{aligned} f(x) &= \frac{1}{4}\sum_{(i,j)\in E}(x_i - x_j)^2 = \frac{1}{4}\left(\sum_{(i,j)\in E}(x_i^2 + x_j^2) - 2\sum_{(i,j)\in E} x_i x_j\right) \\ &= \frac{1}{4}\Big(\underbrace{\sum_{(i,j)\in E} 2}_{=2|E|} - 2\sum_{(i,j)\in E} x_i x_j\Big) \\ &= \frac{1}{4}\Big(\sum_{j=1}^{n} d(j) - 2\sum_{(i,j)\in E} x_i x_j\Big) \\ &= \frac{1}{4} x^T L x. \end{aligned}$$

□

Als aus (3.6.4) abgeleitete vereinfachte (kontinuierliche) Aufgabenstellung betrachten wir deshalb

$$(3.6.5) \qquad \left\{ \begin{array}{lcl} \text{minimiere } \ell(x), x \in \mathbb{R}^n : & & \\ \langle x, x\rangle & = & n \\ \langle x, e\rangle & = & 0. \end{array} \right.$$

Der nächste Satz zeigt, dass (3.6.5) genau durch die Fiedler-Vektoren von $L(G)$ gelöst wird.

3.6.15 Satz
Ein x löst (3.6.5) genau dann, wenn x Eigenvektor zum Eigenwert λ_2 von $L = L(G)$ ist, wobei $0 = \lambda_1 < \lambda_2 \leq ... \leq \lambda_n$ die Eigenwerte von L sind.

Beweis: Es sei z_j ein Eigenvektor zum Eigenwert λ_j mit $\|z_j\| = 1$. Da L symmetrisch ist, kann man $\langle z_i, z_j\rangle = 0$ für $i \neq j$ erreichen. Da G zusammenhängend ist, wissen wir auch

$$z_1 = \frac{1}{\sqrt{n}} e = \frac{1}{\sqrt{n}}(1, 1, ..., 1)^T.$$

Sei nun

$$x = \sum_{j=1}^{n} \alpha_j z_j, \quad \|x\|_2^2 = n \Longleftrightarrow \sum_{j=1}^{n} \alpha_j^2 = n.$$

Es ist

$$\langle x, e\rangle = \left\langle x, \frac{1}{\sqrt{n}} z_1 \right\rangle = \frac{1}{\sqrt{n}}\alpha_1 = 0 \Longleftrightarrow \alpha_1 = 0.$$

Wir suchen also ein x mit $x = \sum_{j=2}^{n} \alpha_j z_j$, $\sum_{j=2}^{n} \alpha_j^2 = n$, welches $\frac{1}{4} x^T L x$ minimiert. Für solche x ist

$$\begin{aligned} Lx &= \sum_{j=2}^{n} \alpha_j \lambda_j z_j \\ x^T L x &= \sum_{j=2}^{n} \alpha_j^2 \lambda_j \end{aligned}$$

und weiter

$$\sum_{j=2}^{n} \alpha_j^2 \lambda_j \geq \sum_{j=2}^{n} \alpha_j^2 \lambda_2 = n\lambda_2.$$

Der Wert $n\lambda_2$ wird genau dann angenommen, wenn $\alpha_j = 0$ für alle j mit $\lambda_j > \lambda_2$ (λ_2 muss kein einfacher Eigenwert sein), d.h. genau dann, wenn x Eigenvektor zum Eigenwert λ_2 ist. $\square$

Die spektrale Bisektion nach Algorithmus 3.6.13 kann man nun so auffassen, dass sie x aus Satz 3.6.15 in geeignete Weise auf $\{\pm 1\}^n$ projiziert, und somit eine nährungsweise Lösung der *diskreten* Optimierungsaufgabe bestimmt.

3.7 Raumfüllende Kurven

Die Anwendung von *raumfüllenden Kurven* auf das Partitionierungsproblem ist noch relativ neu (seit Ende der 90er).

Wir wollen uns zuerst mit den Grundlagen befassen.

3.7.1 Definition
Eine stetige, *surjektive* Abbildung

$$\begin{aligned} \varphi : \ [0,1] &\rightarrow [0,1] \times [0,1] \\ t &\mapsto (\varphi_x(t), \varphi_y(t)) \end{aligned}$$

heißt **raumfüllende Kurve**.

Zunächst einmal ist es wegen Kardinalitätsbetrachtungen klar, dass es bijektive Abbildungen zwischen $[0,1]$ und $[0,1] \times [0,1]$ gibt. Man kann zeigen, dass solche *bijektiven* Abbildungen nie stetig sind. Bei raumfüllenden Kurven verzichtet man auf die Injektivität zu Gunsten der Stetigkeit. Trotzdem ist nicht ohne weiteres klar, dass es überhaupt raumfüllende Kurven gibt. Wir behandeln im Folgenden nur eine von vielen Möglichkeiten, nämlich die *Hilbertkurve.*

Die Hilbertkurve

Ein rekursives Konstruktionsprinzip für die Hilbertkurve bestimmt für $\ell = 0, 1, ...$ Polygonzüge φ_ℓ, welche Punkte (x, y) der Form

$$\text{(l)} \qquad \begin{cases} x = \dfrac{2k+1}{2^{\ell+1}} \\ y = \dfrac{2m+1}{2^{\ell+1}} \end{cases}, \quad k, m = 0, ..., 2^\ell - 1$$

verbinden. Dies sind im ℓ-ten Schritt die Mittelpunkte der $2^{2\cdot\ell}$ Quadrate, die entstehen, wenn man jedes Quadrate des $(\ell - 1)$-ten Schrittes in vier gleich große achsenparallele Quadrate der Seitenlänge $1/2^\ell$ unterteilt.

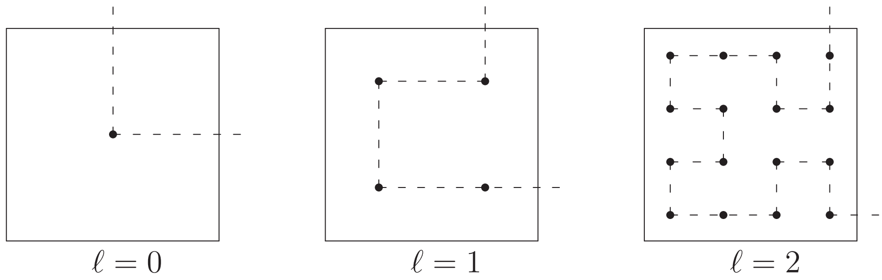
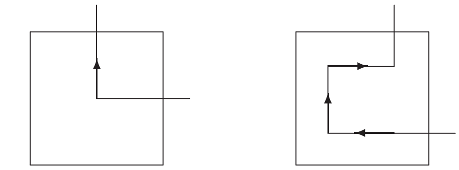


Abbildung 3.11: Die ersten drei Polygonzüge für die Hilbertkurve

Der Polygonzug φ verbindet also diese Mittelpunkte. Der Polygonzug $\varphi_{\ell+1}$ verbindet die Mittelpunkte von Quadraten der halben Seitenlänge. Wie das geschieht, hängt davon ab, wie φ_ℓ verläuft. Wir unterscheiden deshalb verschiedene „Typen“. Es treten 12 verschiedene Typen auf, wobei sich immer zwei wegen der Symmetrie entsprechen (nur der Durchlaufssinn ändert sich). Wir nummerieren die Typen von 0 bis 5 durch und markieren den Durchlaufssinn mit einer führenden 0 bzw. 1. Es gibt also die Typen 00 bis 05 und 10 bis 15.

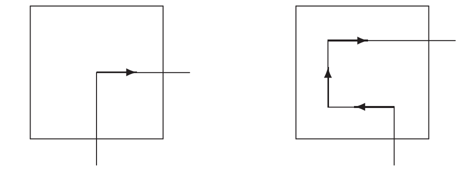
Typ φ_ℓ $\varphi_{\ell+1}$ **Tabelle**

00



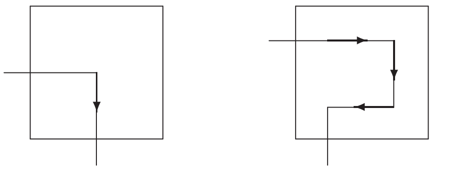
type	14	00	01	13
xbit	+	−	−	+
ybit	−	−	+	+

01



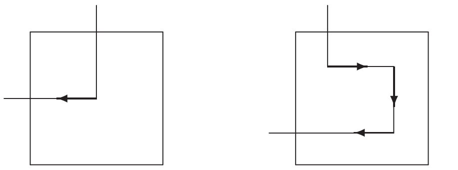
type	12	00	01	04
xbit	+	−	−	+
ybit	−	−	+	+

02



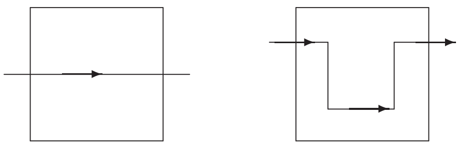
type	04	02	03	11
xbit	−	+	+	−
ybit	+	+	−	−

03



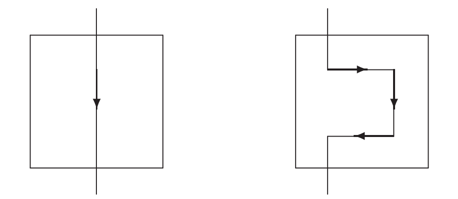
type	10	02	03	14
xbit	−	+	+	−
ybit	−	+	−	+

04



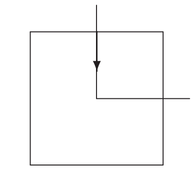
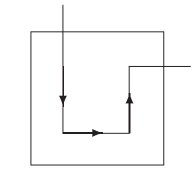
type	02	10	13	01
xbit	−	−	+	+
ybit	+	−	−	+

05

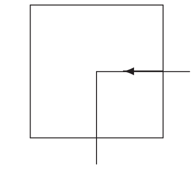
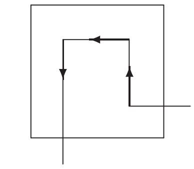


type	10	02	03	11
xbit	−	+	+	−
ybit	+	+	−	−

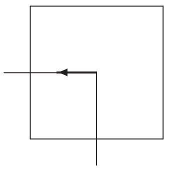
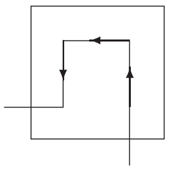
Typ	φ_ℓ	$\varphi_{\ell+1}$	**Tabelle**

10  

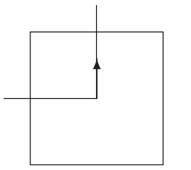
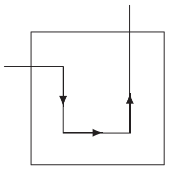
type	05	10	13	01
xbit	−	−	+	+
ybit	+	−	−	+

11  

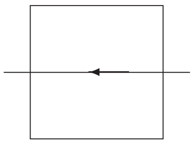
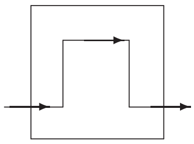
type	00	12	11	05
xbit	+	+	−	−
ybit	−	+	+	−

12  

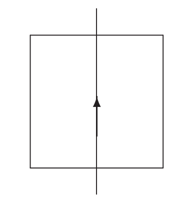
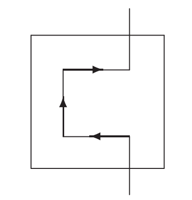
type	15	12	11	03
xbit	+	+	−	−
ybit	−	+	+	−

13  

type	02	10	13	15
xbit	−	−	+	+
ybit	+	−	−	+

14  

type	00	12	11	03
xbit	+	+	−	−
ybit	−	+	+	−

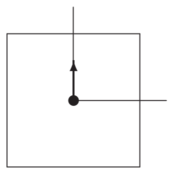
15  

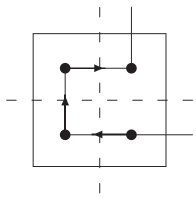
type	12	00	01	13
xbit	+	−	−	+
ybit	−	−	+	+

Wir erläutern die Tabelle am Beispiel des ersten Typs 00:

Typ φ_ℓ $\varphi_{\ell+1}$ **Tabelle**

00





type	14	00	01	13
xbit	+	−	−	+
ybit	−	−	+	+

Das Quadrat wird in vier kleinere Quadrate unterteilt, d.h. aus einem Mittelpunkt M in der Rekursionsstufe ℓ werden vier neue $m_1, ..., m_4$. Diese Mittelpunkte können wir eindeutig dadurch bestimmen, indem wir ihre Position bezüglich M angeben. Diese Position gibt (xbit,ybit) an und zwar ist für m_j

$$\begin{aligned} \text{xbit} &= \begin{cases} +, & \text{falls } x(M) > x(m_j) \\ -, & \text{falls } x(M) < x(m_j) \end{cases} \\ \text{ybit} &= \begin{cases} +, & \text{falls } y(M) > y(m_j) \\ -, & \text{falls } y(M) < y(m_j) \end{cases}. \end{aligned}$$

Genauer bedeuten $+, -$ hierbei eine Verschiebung um $+2^{-(\ell+1)}$ bzw. $-2^{-(\ell+1)}$ der entsprechenden Koordinate von m_j gegenüber M. Die Größe type gibt für das neue Quadrat des Mittelpunkts m_j den Typ an. Die Reihenfolge der Spalten der Tabelle ist die Reihenfolge, in welcher $\varphi_{\ell+1}$ die neuen Mittelpunkte durchläuft.

Wie bestimmt man für ein gegebenes t die Werte $\varphi_x(t), \varphi_y(t)$ rechnerisch? Es sei t im 4er-System gegeben, d.h.

$$t = 0.q_1q_2...., \quad q_i \in \{0, 1, 2, 3\}.$$

Wir bestimmen die Dualdarstellung

$$\begin{aligned} \varphi_x(t) &= 0.x_1x_2x_3... \quad x_i \in \{0, 1\} \\ \varphi_y(t) &= 0.y_1y_2y_3... \quad y_i \in \{0, 1\} \end{aligned}$$

unter Zuhilfenahme der bereits in Tabellenform aufgeschriebenen Daten. Setzen wir $Typen := \{00, 01, ..., 05, 10, ...15\}$, so können wir type, xbit und ybit auch als Abbildungen auffassen, nämlich

$$\text{type}: \; Typen \times \{0, 1, 2, 3\} \;\; \rightarrow \;\; Typen$$

$$\begin{aligned}\text{xbit}: \ \mathit{Typen} \times \{0,1,2,3\} &\rightarrow \{-1,1\}\\ \text{ybit}: \ \mathit{Typen} \times \{0,1,2,3\} &\rightarrow \{-1,1\}.\end{aligned}$$

Damit erhalten wir folgenden Algorithmus zur Berechnung von φ_x, φ_y.

3.7.2 Algorithmus

Require: currenttype=00, φ_x=0.5, φ_y=0.5, $t = 0.q_1q_2...$ mit $q_i \in \{0,1,2,3\}$
  for $i = 1, 2, ...$ **do**
    $\varphi_x = \varphi_x$+xbit(currenttype,q_i) $\cdot 2^{-(i+1)}$
    $\varphi_y = \varphi_y$+ybit(currenttype,q_i) $\cdot 2^{-(i+1)}$
    currenttype=type(currenttype,q_i)
  end for

Bemerkung: Man kann die Initialisierung von currenttype in Algorithmus 3.7.2 selbstverständlich ändern. Alle entstehenden Kurven sind jedoch nicht geschlossen. Möchte man eine geschlossene raumfüllende Kurve bestimmen, muss man statt mit der Rekursionsstufe 0 mit der Rekursionsstufe 1 initialisieren, siehe Übung.

Bemerkung: Die Zahldarstellung im Vierersystem ist nicht eindeutig, da für $q_k < 3$ gilt

$$0.q_1q_2 \ldots q_k\bar{3} = 0.q_1q_2 \ldots q_{k-1}(q_k + 1)\bar{0}.$$

Man kann zeigen (wir tun das jetzt aber nicht), dass Algorithmus 3.7.2 unabhängig von der Zahldarstellung ist. Ebesno verzichten ir darauf zu zeigen, dass φ_x und φ_y tatsächlich stetig sind. Klar ist, dass $\varphi(t) \in [0,1]^2$ gilt.

Wir geben jetzt noch einen Algorithmus an der zu gegebenen $x, y \in [0,1]$ ein t bestimmt mit $\varphi_x(t) = x, \varphi_y(t) = y$. Den durch den Algorithmus bestimmten Wert t bezeichnen wir mit $\psi(x,y)$.

Wir betrachten zunächst

$$\text{tqit}: \ \{-1,1\} \times \{-1,1\} \times \mathit{Typen} \rightarrow \{0,1,2,3\}$$

als Umkehrabbildung von xbit und ybit in dem Sinne, dass

$$\text{tqit}(\text{xbit}(\text{type}, q), \text{ybit}(\text{type}, q), \text{type}) = q.$$

Im Fall des Typs 00 mit

type	14	00	01	13
xbit	+	−	−	+
ybit	−	−	+	+

ist also

$$\begin{aligned} \text{tqit}(+1,-1,00) &= 0 \\ \text{tqit}(+1,+1,00) &= 3. \end{aligned}$$

Die Werte von tqit sind also aus den vorherigen Tabellen ablesbar. Z.B. ergibt sich

tqit$(\cdot,\cdot,00)$:

xbit \ ybit	-1	+1
-1	1	2
+1	0	3

usw. Damit können wir aber als „Umkehrung" zu Algorithmus 3.7.2 den folgenden Algorithmus formulieren. Er bestimmt zu $x, y \in [0,1]$ die Darstellung $\psi(x,y) = t = 0.q_1q_2...$ im Vierersystem, so dass $x = \varphi_x(t),\ y = \varphi_y(t)$.

3.7.3 Algorithmus

Require: $x \in [0,1],\ y \in [0,1]$
  $x = x - 0.5$, $y = y - 0.5$, currenttype = 00;
  for $i = 1, 2, ...$ **do**
    if $x \leq 0$ **then**
      xbit=-1, $x = x + 2^{-(i+1)}$
    else
      xbit=1, $x = x - 2^{-(i+1)}$
    end if
    if $y \leq 0$ **then**
      ybit=-1, $y = y + 2^{-(i+1)}$
    else
      ybit=1, $y = y - 2^{-(i+1)}$
    end if
    q_i=tqit(xbit,ybit,currenttype)
    currenttype = type(currenttype,q_i)
  end for

Bemerkung: Die Fälle $x = 0$, $y = 0$ kann man im obigen Algorithmus wahlweise und an sogar indivisuell an jeder Stelle dem anderen Fall ($x < 0$, $y < 0$) zuschlagen, ohne dass der Algorihmus inkorrekt würde. Dies zeigt, dass φ nicht injektiv ist.

Jetzt wenden wir raumfüllende Kurven auf die Aufgabe der Graph-Partitionierung an.

Gegeben: Ein Graph $G = (V, E)$. Zusätzlich besitze jeder Knoten i eine $x-$ und $y-$Koordinate (x_i, y_i). Ohne Beschränkung seien $x_i, y_i \in (0, 1)$. Bezüglich dieser Koordinaten seien die Nachbarschaften im Graphen auch geometrisch lokal, d.h. Knoten, welche über eine Kante verbunden sind, besitzen Koordinaten, die sich um wenig unterscheiden. (Dies sind typische Voraussetzungen, die von Finiten-Elemente-Graphen erfüllt werden.)

Gesucht: Partitionierung von G in p Teile mit möglichst gleich großer Knotenzahl und geringer Schnittgröße

Heuristik: [4] Wähle eine raumfüllende Kurve, wie z.B. die Hilbertkurve. Bestimme zu jedem Knoten den Kurvenparameter. Über die Kurvenparameter sind die Knoten linear geordnet. Teile bezüglich dieser Anordnung in aufeinanderfolgende Blöcke auf.

Die Blöcke bestehen so meist aus geometrisch benachbarten Punkten, weshalb die Schnittgröße klein werden sollte.

Der zugehörige Parameterbereich entspricht einem „zweidimensionalen“ Bereich in $(0, 1) \times (0, 1)$.

3.7.4 Algorithmus

for all Knoten $i \in V$ **do**
  bestimme $t_i = \psi(x_i, y_i)$
end for
sortiere die t_i zu $t_{ind(1)} < t_{ind(2)} < ... < t_{ind(n)}$
bestimme $n_i = \lceil i \cdot \frac{n}{p} \rceil, \; i = 1, ..., p-1$
Setze

$$\begin{aligned} V_1 &= \{ind(1), ..., ind(n_1)\} \\ V_2 &= \{ind(n_1+1), ..., ind(n_2)\} \\ &\vdots \\ V_p &= \{ind(n_{p-1}+1), ..., ind(n)\} \end{aligned}$$

3.7.5 Satz
Der Aufwand von Algorithmus 3.7.4 beträgt

$$\mathcal{O}(n \log(n)) + \mathcal{O}(nd),$$

[4]Heuristik: Lösungsverfahren, welches zwar motiviert werden kann, dessen Korrektheit jedoch nicht bewiesen ist.

wobei d die Anzahl der qits ist, mit denen $t = \psi(x, y)$ in Algorithmus 3.7.4 bestimmt wird (d.h. die Genauigkeit).

Beweis: Als Aufwand ergibt sich

Berechnung von t_i:	$\mathcal{O}(nd)$	
Sortieren:	$\mathcal{O}(n\log(n))$	(z.B. Quicksort)
Rest:	$\mathcal{O}(n)$	(Überschreiben, etc.)

□

Bemerkung: Die Größe d ist üblicherweise ziemlich klein, da die t_i nur so genau bekannt sein müssen, dass man sie unterscheiden kann. Bei der Partitionierung in k Teile genügen ungefähr $2 \cdot \log_4(k)$ qits, wenn die t_i auch gleichmäßig auf $(0, 1)$ verteilt sind.

Der Aufwand von $\mathcal{O}(n\log(n))$ kann noch weiter auf $\mathcal{O}(nd)$ reduziert werden. Dazu verwenden wir Radixsort zum Radix[5] 4. Es bezeichne $quit(t, k)$ das k-te qit q_k von $t = 0.q_1q_2...$ Wir verwendeen 5 Queues Q, Q_0, Q_1, Q_2, Q_3. Queues stellen hierbei die optimale Datenstruktur dar (beachte Speicherbedarf). Damit erhalten wir folgenden Algorithmus.

3.7.6 Algorithmus

Füge alle t in Q ein
for $k = d$ bis 1 **do** {die Stellen werden rückwärts durchlaufen}
  while Q nicht leer **do**
    entferne nächstes Element aus Q
    bestimme $q = qit(t, k)$
    füge t in Q_q ein
  end while
  Übertrage Q_0 in Q
  Übertrage Q_1 in Q
  Übertrage Q_2 in Q
  Übertrage Q_3 in Q
end for

3.7.7 Beispiel

1. Betrachten wir $Q = \{0.20, 0.22, 0.31, 0.00, 0.01, 0.32, 0.33, 0.11\}$ Dann ergibt Algorithmus 3.7.6 für $d = 2$ folgende Schritte

[5] d.h. Basis

Q	0.20,0.22,0.31, 0.00, 0.01, 0.32, 0.33, 0.11
Q_0	
Q_1	
Q_2	
Q_3	
Q	
Q_0	0.20, 0.00
Q_1	0.31, 0.01, 0.11
Q_2	0.22,0.32
Q_3	0.33
Q	0.20, 0.00, 0.31, 0.01, 0.11, 0.22, 0.32, 0.33
Q_0	
Q_1	
Q_2	
Q_3	
Q	
Q_0	0.00, 0.01
Q_1	0.11
Q_2	0.20, 0.22
Q_3	0.31, 0.32, 0.33
Q	0.00, 0.01, 0.11, 0.20, 0.22, 0.31, 0.32, 0.33

2. Mittels MATLAB kann man die Partitionierung visualisieren (siehe Übung).

3.7.8 Bemerkung
In Simulationsrechnung mit Finiten Elementen kann es nötig werden während der Rechnung das Finite-Elemente-Gitter verfeinern zu müssen. Eine Partitionierung für das neue Gitter kann man dann folgendermaßen gewinnen:

1. Verteilung der zusätzlichen Knoten unter Beibehaltung der Partitionierung für die alten Knoten. Dann wird die Schnittgröße im Allgemeinen relativ groß.

2. neue Verteilung aller Gitterknoten. Dies bedeutet einen einmaligen, im Allgemeinen sehr hohen, Kommunikationsaufwand.

Raumfüllende Kurven sind im 2. Fall erfahrungsgemäß günstig: Die neue Verteilung ergibt sich dann durch „Verschiebung" des Intervalls für den Parameter t, das jedem Prozessor zugeordnet ist. Es ist dann nur Kommunikation zwischen „benachbarten" Parameterintervallen (=Prozessoren) nötig.

3.8 Der Kernigan-Lin Algorithmus

Der Kernigan-Lin Algorithmus verkleinert die Schnittgröße für eine gegebene Partition $V = V_1 \cup V_2$ eines Graphen $G = (V, E)$. Dabei können die Kanten $e \in E$ mit einer Gewichtsfunktion

$$w : E \to \mathbb{R}^+$$

gewichtet sein. Bei uns liegt in der Regel Einheitsgewichtung vor, d.h. $w(e) = 1$ für alle $e \in E$.

3.8.1 Definition
Es sei $V = V_1 \cup V_2$ eine Partition eines Graphen $G = (V, E)$ und $v \in V$. Dann setzen wir

$$\mathrm{diff}(v) := \mathrm{diff}(v, V_1, V_2) = \sum_{e \in E,\ e=\{v,x\},\ x \in V_2} w(e) - \sum_{e \in E,\ e=\{v,x\},\ x \in V_1} w(e).$$

Interpretation: Wat $v \in V_1$ und ändern wir die Partition $V_1 \cup V_2$ in $V = V_1' \cup V_2'$ mit $V_1' = V_1 \setminus \{v\}$ und $V_2' = V_2 \cup \{v\}$, so gilt:

$$\mathrm{diff}(v, V_1', V_2') = -\mathrm{diff}(v, V_1, V_2)$$

$$\mathrm{diff}(x, V_1', V_2') = \begin{cases} \mathrm{diff}(x, V_1, V_2) - 2w(\{x, v\}) & \text{falls } x \in V_2, \{x, v\} \in E, \\ \mathrm{diff}(x, V_1, V_2) + 2w(\{x, v\}) & \text{falls } x \in V_1, \{x, v\} \in E, \\ \mathrm{diff}(x, V_1, V_2) & \text{sonst.} \end{cases}$$

Die (gewichtete) Schnittgröße

$$s(V_1, V_2) = \sum_{e \in E,\ e=\{x,y\},\ x \in V_1,\ y \in V_2} w(e)$$

ändert sich gemäß

$$s(V_1', V_2') = s(V_1, V_2) - \mathrm{diff}(v).$$

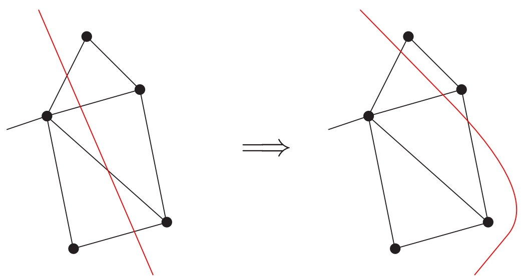


Abbildung 3.12: diff ändert sich

Bemerkung: Es gilt

$$\operatorname{diff}(x, V_1', V_2') \neq \operatorname{diff}(x, V_1, V_2)$$

nur, wenn entweder $x = v$ oder es gibt eine Kante $\{x, v\} \in E$.

3.8.2 Definition
Es sei $V = V_1 \cup V_2$ eine Partition eines Graphen $G = (V, E)$, es sei weiter $v_1 \in V_1, v_2 \in V_2$. Dann setzen wir

$$\operatorname{gain}(v_1, v_2, V_1, V_2) := \operatorname{diff}(v_1) + \operatorname{diff}(v_2) - 2w(\{v_1, v_2\}),$$

wobei $w(\{v_1, v_2\}) = 0$ gelte, wenn $\{v_1, v_2\} \notin E$.

Interpretation: Ändern wir die Partition $V_1 \cup V_2$ in $V = V_1' \cup V_2'$ mit $V_1' = (V_1 \setminus \{v_1\}) \cup \{v_2\}$ und $V_2' = (V_2 \setminus \{v_2\}) \cup \{v_1\}$, so ändert sich die (gewichtete) Schnittgröße gemäß

$$s(V_1', V_2') = s(V_1, V_2) - \operatorname{gain}(v_1, v_2).$$

3.8.3 Lemma
Es sei $V = V_1 \cup V_2$ eine Partition eines Graphen $G = (V, E)$, weiter seien $v_1 \in V_1, v_2 \in V_2$ und $V_1' = (V_1 \setminus \{v_1\}) \cup \{v_2\}, V_2' = (V_2 \setminus \{v_2\}) \cup \{v_1\}$. Dann gilt für alle x, die Nachbarn von v_1 oder v_2 sind

$$\text{(3.8.1)}\ \operatorname{diff}(x, V_1', V_2') = \operatorname{diff}(x, V_1, V_2) + \begin{cases} 2 \cdot (w(\{x, v_1\}) - w(\{x, v_2\})), & x \in V_1 \\ 2 \cdot (w(\{x, v_2\}) - w(\{x, v_1\})), & x \in V_2. \end{cases}$$

Für alle anderen x ändert sich $\operatorname{diff}(x, V_1, V_2)$ nicht.

Beweis: Übung. □

Der Algorithmus von Kernigan-Lin versucht, $s(V_1, V_2)$ zu minimieren, indem geeignete Knoten vertauscht werden.

3.8.4 Algorithmus

Require: $V = V_1 \cup V_2$ Partition von V in $G = (V, E)$
  $s = s(V_1, V_2)$
  repeat
    bestimme diff(x) für alle Knoten x
    $k_0 = s$
    for $i = 1$ bis $\min\{|V_1|, |V_2|\}$ **do**
      bestimme $a_i \in V_1 \setminus \{a_1, ..., a_{i-1}\}, b_i \in V_2 \setminus \{b_1, ..., b_{i-1}\}$ so dass gain(a_i, b_i) maximal ist
      for all Nachbarn x von a_i und b_i **do**
        datiere diff(x) auf gemäß Gleichung (3.8.1)
      end for
      $k_i = k_{i-1} - \text{gain}(a_i, b_i)$ {k_i ist die Schnittgröße, wenn alle a_j, $1 \le j \le i$ nach V_2 und alle b_j, $1 \le j \le i$ nach V_1 gebracht würden.}
    end for
    bestimme j mit k_j ist minimal in $\{k_0, \ldots, k_{\min\{|V_1|,|V_2|\}}\}$.
    vertausche die ersten j Paare:

$$\begin{aligned} V_1 &= (V_1 \setminus \{a_1, ..., a_j\}) \cup \{b_1, ..., b_j\} \\ V_2 &= (V_2 \setminus \{b_1, ..., b_j\}) \cup \{a_1, ..., a_j\} \end{aligned}$$

    $s = k_j$
  until $s = k_0$ {Dann wurde keine Verbesserung mehr erreicht}

Der Algorithmus kann unter Verwendung geeigneter Datenstrukturen (und Bucket-Sort) so implementiert werden, dass eine (Repeat-) Iteration einen Aufwand von $\mathcal{O}(|E|)$ besitzt.

3.8.5 Beispiel (eine (Repeat-) Iteration!)

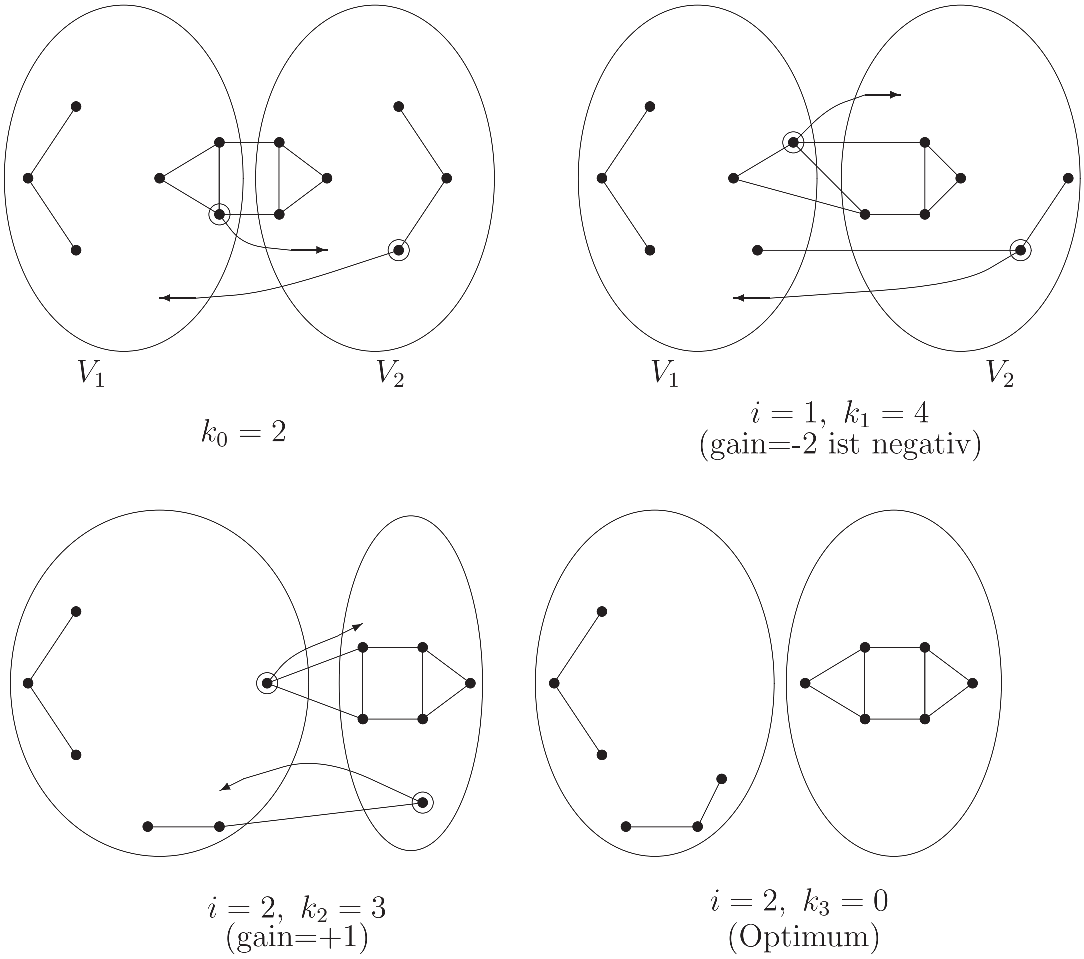


Abbildung 3.13: Beispiel zu Algorithmus 3.8.4

Bemerkung: Es gibt keine Garantie, dass Algorithmus 3.8.4 eine minimale Schnittgröße findet, aber er verschlechtert sie jedenfalls nicht.

Kapitel 4

Iterationsverfahren

Gegeben: Ein lineares Gleichungssystem

$$Ax = b$$

mit einer regulären Matrix $A \in \mathbb{R}^{n \times n}$.

Gesucht: Lösung $x \in \mathbb{R}^n$.

Vorgehen: Iterativ, d.h. man bestimmt eine Folge $x^{(k)}$ von Näherungslösungen mit $\lim\limits_{k \to \infty} x^{(k)} = A^{-1}b$.

Bemerkung: Iterative Verfahren werden aus mehreren Gründen benutzt:

1. Der Aufwand kann wesentlich geringer sein als beim expliziten Lösen.
2. Meist sind in Anwendungen nicht die exakten Lösungen wichtig, da das Gleichungssystem
   (a) Modellfehler oder
   (b) Diskretisierungsfehler

   aufweist. Man kann die Iteration also häufig relativ früh abbrechen.

4.1 Das Jacobi-Verfahren

4.1.1 Definition
Es sei $A = (a_{i,j}) \in \mathbb{R}^{n\times n}$. Dann sind D, L und U gegeben durch

$$D = \begin{pmatrix} a_{1,1} & & & & \\ & \ddots & & 0 & \\ & & \ddots & & \\ & 0 & & \ddots & \\ & & & & a_{n,n} \end{pmatrix} \qquad \text{(Diagonalteil)}$$

$$L = \begin{pmatrix} 0 & & & & \\ -a_{2,1} & \ddots & & 0 & \\ \vdots & & \ddots & & \\ \vdots & & & \ddots & \\ -a_{n,1} & \dots & \dots & -a_{n,n-1} & 0 \end{pmatrix} \qquad \begin{array}{l}\text{(negativer)}\\ \text{linker unterer Dreiecksteil}\end{array}$$

$$U = \begin{pmatrix} 0 & -a_{1,2} & \dots & \dots & -a_{1,n} \\ & \ddots & & & \vdots \\ & & \ddots & & \vdots \\ & 0 & & \ddots & -a_{n-1,n} \\ & & & & 0 \end{pmatrix} \qquad \begin{array}{l}\text{(negativer)}\\ \text{rechter oberer Dreiecksteil}\end{array}$$

Also ist $A = D - L - U$.

Wir setzen ab jetzt voraus, dass D regulär ist.

4.1.2 Definition
Das **Jacobi-Verfahren** (Gesamtschrittverfahren) zur Lösung von $Ax = b$ ist gegeben durch

$$x^{(k+1)} = D^{-1}\{(L+U)x^{(k)} + b\}, \quad k = 0, 1, ... \tag{4.1.1}$$

wobei $x^{(0)}$ ein beliebiger Startwert ist.

Es sei $e^{(k)} = x^{(k)} - x^*$ mit $x^* = A^{-1}b$. Dann folgt aus $Ax^* = b$

$$Dx^* = (L+U)x^* + b, \quad x^* = D^{-1}\{(L+U)x^* + b\}$$

und damit
$$e^{(k+1)} = D^{-1}(L+U)e^{(k)}, \quad k = 0, 1, ...$$

4.1.3 Satz
Es sei $B \in \mathbb{R}^{n\times n}$, $y^{(k)} = B^k y^{(0)},\ k = 1, 2, ...$ Dann gilt

$$\lim_{k\to\infty} B^k y^{(0)} = 0 \text{ für jede Wahl von } y^{(0)} \Longleftrightarrow \rho(B) < 1.$$

Beweis: Bekannt aus der Numerik. □

Beachte: Es gilt, $\rho(B) < 1$ dann und nur dann, wenn es eine Norm $\|\cdot\|$ auf dem $\mathbb{R}^n$ und ein $\alpha \in [0, 1)$ gibt, so dass $\|By\| \leq \alpha\|y\|$ für alle y gilt.

4.1.4 Definition
Es sei $\|\cdot\|$ eine Norm im $\mathbb{R}^n$. Dann ist die zugehörige **Operatornorm** $\|\cdot\|$ auf dem $\mathbb{R}^{n\times n}$ gegeben durch

$$\|A\| := \max_{\|x\|=1} \|Ax\|, \quad (A \in \mathbb{R}^{n\times n},\ x \in \mathbb{R}^n).$$

Insbesondere gilt also
$$\|Ax\| \leq \|A\| \cdot \|x\|.$$

4.1.5 Beispiel

1. Setzen wir $\|x\| = \|x\|_2 = \left(\sum\limits_{j=1}^{n} |x_j|^2\right)^{1/2}$, so ergibt sich $\|A\|_2 = (\rho(A^T A))^{1/2}$ (**Spektralnorm**).

2. Ist $\|x\| = \|x\|_\infty = \max\limits_{j=1,...,n} |x_j|$, so folgt $\|A\|_\infty = \max\limits_{i=1,...,n} \sum\limits_{j=1}^{n} |a_{i,j}|$ (**Zeilensummennorm**).

3. Setzen wir $\|x\| = \|x\|_1 = \sum\limits_{j=1}^{n} |x_j|$, so ist $\|A\|_1 = \max\limits_{j=1,...,n} \sum\limits_{i=1}^{n} |a_{i,j}|$ (**Spaltensummennorm**).

4. Ist $u \in \mathbb{R}^n$ mit $u > 0$ (d.h. $u_j > 0,\ j = 1, ..., n$) fest, so definiere

$$\|x\| = \|x\|_u = \max_{j=1,...,n} \left|\frac{x_j}{u_j}\right|$$

damit ergibt sich

$$\|A\|_u = \max_{i=1,...,n} \left(\frac{1}{u_i} \cdot \sum_{j=1}^{n} |a_{i,j}| \cdot u_j\right).$$

4.1.6 Satz
Das Jacobi-Verfahren 4.1.1 konvergiert (für jeden Startwert $x^{(0)}$ gegen $x^* = A^{-1}b$) genau dann, wenn $\rho(D^{-1}(L+U)) < 1$ gilt.

Beweis: Klar nach Satz 4.1.3. □

Aus der Matrix-Formulierung des Jacobi-Verfahrens ist

$$x^{(k+1)} = D^{-1}[(L+U)x^{(k)} + b]$$

ergibt sich komponentenweise

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, ..., n,\ k = 0, 1, ... \quad .$$

Speziell beim Modellproblem I aus Abschnitt 3.2

$$\begin{aligned} \Delta u &= f \quad \text{in } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned}$$

ergibt sich

$$\text{(4.1.2)} \qquad \begin{aligned} u_{i,j}^{(k+1)} = & \frac{1}{4}\left(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + h^2 f_{i,j}\right) \\ & i, j = 1, ..., N\ , \\ & k = 0, 1, ...\ , \end{aligned}$$

wobei $u_{0,j}^{(k)}, u_{N+1,j}^{(k)}, u_{i,0}^{(k)}, u_{i,N+1}^{(k)}$ für $i, j = 1, ..., N,\ k = 1, 2, ...$ mit den vorgegebenen Randwerten zu initialisieren sind.

Für eine dünn besetzte Matrix $A \in \mathbb{R}^{n\times n}$ geschieht die Parallelisierung des Jacobi-Verfahren im Wesentlichen genau so wie die Matrix-Vektor-Multiplikation, siehe Abschnitt 3.5. Wir setzen A als symmetrisch voraus, $G = G(A) = (V, E)$ sei der zugehörige Graph, $V = V_1 \cup V_2 \cup ... \cup V_p$ eine Partition der Knotenmenge und $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ der Quotientengraph.

4.1.7 Algorithmus (Jacobi-Verfahren)

for $k = 0, 1, ...$ **do**
  if me $= P_\ell$ **then**
    for all m mit $\{m, \ell\} \in \mathcal{E}$ **do**
      send$\Big(x_i^{(k)}, \exists i \in V_\ell, j \in V_m \text{ mit } \{i, j\} \in E\Big)$ to P_m
    end for
    for all m mit $\{\ell, m\} \in \mathcal{E}$ **do**
      receive$\Big(x_j^{(k)}, \exists i \in V_\ell, j \in V_m \text{ mit } \{i, j\} \in E\Big)$ from P_m
    end for
    for all $i \in V_\ell$ **do**
      $$x_i^{(k+1)} = \frac{1}{a_{i,i}} \cdot \left(b_i - \sum_{\substack{j=1 \\ \{i,j\} \in E}}^{n} a_{i,j} x_j^{(k)} \right)$$
    end for
  end if
end for

Bemerkung:

1. Natürlich muss man noch zusätzlich Abbruchkriterien einbauen (z.B. die Bestimmung der Größe des Residuums). Diese Abbruchkriterien können nicht in *einem* Prozessor entschieden werden, d.h. es wird zusätzliche Kommunikation notwendig.

2. Das Jacobi-Verfahren konvergiert nicht immer. Häufig ist die Konvergenz zudem sehr langsam. Für das Modellproblem I konvergiert das Jacobi-Verfahren z. B. immer langsamer mit wachsendem N.

4.2 Das Gauß-Seidel-Verfahren

Das Jacobi-Verfahren war gegeben durch

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{i,j} x_j^{(k)} \right) \qquad i = 1, ..., n,\ k = 0, 1, ...$$

Für festes i liegen aber bereits die $x_j^{(k+1)}$, $j = 1, ..., i-1$, vor. Die Idee ist, diese neuen Werte zu verwenden!

4.2.1 Definition
Das **Gauß-Seidel-Verfahren** (Einzelschrittverfahren) zur Lösung von $Ax = b$ berechnet die Iterierte $x^{(k+1)}$ gemäß

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, ..., n. \tag{4.2.1}$$

Beachte: Im obigen Verfahren hängt das Resultat im Gegensatz zum Jacobi-Verfahren nun aber von der Nummerierung der Komponenten ab! Mehr noch: Es gibt Beispiele, bei denen das Gauß-Seidel-Verfahren für eine gegebene Nummerierung konvergiert, für eine andere aber *nicht*!

Es sei $A = D - L - U$ wie in Abschnitt 4.1. Dann ist eine kompakte Formulierung des Gauß-Seidel-Verfahrens

$$\begin{cases} & Dx^{(k+1)} = b + Lx^{(k+1)} + Ux^{(k)} \\ \Longleftrightarrow & (D-L)x^{(k+1)} = b + Ux^{(k)} \\ \Longleftrightarrow & x^{(k+1)} = (D-L)^{-1}(b + Ux^{(k)}) \end{cases} \tag{4.2.2}$$

Folge: Das Gauß-Seidel-Verfahren konvergiert für jeden Startwert genau dann, wenn

$$\rho((D-L)^{-1}U) < 1.$$

Ziel: Zeige die Konvergenz des Gauß-Seidel-Verfahrens, falls A *symmetrisch positiv definit* ist.

4.2.2 Definition

1. Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt **spd** (symmetrisch positiv definit), wenn A symmetrisch und positiv definit ist, d.h. wenn $A = A^T$ und

$$\langle x, Ax \rangle = x^T A x > 0 \quad \forall\, x \neq 0. \tag{4.2.3}$$

2. Ist $A \in \mathbb{R}^{n\times n}$ spd, so ist die zugehörige **Energienorm** $\|\cdot\|_A$ auf dem $\mathbb{R}^n$ gegeben durch

$$\|x\|_A := (\langle x, Ax\rangle)^{1/2} = \left(\sum_{j=1}^{n}\sum_{i=1}^{n} a_{i,j}x_i x_j\right)^{1/2}.$$

Einige Tatsachen über spd-Matrizen

Für $A \in \mathbb{R}^{n\times n}$ spd gilt:

1. Ist λ Eigenwert von A, so ist $\lambda > 0$.
2. Ist P eine Permutationsmatrix, so ist auch PAP^T spd.
3. Es gibt eine Zerlegung
$$A = Q\Lambda Q^T,$$
wobei $\Lambda = \operatorname{diag}(\lambda_1, \lambda_2, ..., \lambda_n), Q = [q_1, q_2, ..., q_n]$. Dabei sind die $\{q_i\}$ ein *orthonormiertes* System von Eigenvektoren und die λ_i Eigenwerte von A, wobei q_i ein Eigenvektor zum Eigenwert λ_i ist.
4. Es sei $B \in \mathbb{R}^{n\times n}$. Dann ist B^TAB symmetrisch und positiv semidefinit (d.h. in (4.2.3) gilt ≥ 0).

   Falls B regulär ist, ist B^TAB symmetrisch und positiv definit.
5. Für die Diagonalelemente von A gilt $a_{i,i} > 0,\ i = 1, 2, ..., n$.

4.2.3 Lemma

Es sei $A \in \mathbb{R}^{n\times n}$ spd. Dann gibt es eine spd-Matrix $A^{1/2} \in \mathbb{R}^{n\times n}$ mit

$$A^{1/2}\cdot A^{1/2} = A.$$

Beweis: Es sei $A = Q\Lambda Q^T$ wie oben, d.h. $Q^TQ = QQ^T = I$ und $\Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_n)$. Setze

$$A^{1/2} := Q\Lambda^{1/2}Q^T \quad \Lambda^{1/2} = diag(\lambda_1^{1/2}, \lambda_2^{1/2}, ..., \lambda_n^{1/2}),$$

dann gilt

$$\begin{aligned} A^{1/2}\cdot A^{1/2} &= (Q\Lambda^{1/2}Q^T)\cdot(Q\Lambda^{1/2}Q^T) \\ &= Q\Lambda^{1/2}\underbrace{Q^TQ}_{=I}\Lambda^{1/2}Q^T \\ &= Q\Lambda^{1/2}\Lambda^{1/2}Q^T \\ &= Q\Lambda Q^T \\ &= A \end{aligned}$$

$\square$

4.2.4 Korollar
Sei $A \in \mathbb{R}^{n\times n}$ spd. Dann gilt

1. $\|x\|_A = \|A^{1/2}x\|_2$
2. Für $B \in \mathbb{R}^{n\times n}$ folgt
$$\|B\|_A^2 = \rho(A^{-1/2}B^TABA^{-1/2}),$$
wobei $A^{-1/2} := (A^{1/2})^{-1}$.

Beweis:

1. Es gilt
$$\begin{aligned}\|x\|_A^2 &= \langle x, Ax\rangle = x^TAx \\ &= x^TA^{1/2}A^{1/2}x = x^T(A^{1/2})^TA^{1/2}x \\ &= \|A^{1/2}x\|_2^2.\end{aligned}$$
2. Es ist
$$\begin{aligned}\|B\|_A &= \max_{x\neq 0}\frac{\|Bx\|_A}{\|x\|_A} = \max_{x\neq 0}\frac{\|A^{1/2}Bx\|_2}{\|A^{1/2}x\|_2} \\ &= \max_{y\neq 0}\frac{\|A^{1/2}BA^{-1/2}y\|_2}{\|y\|_2} \\ &= \|A^{1/2}BA^{-1/2}\|_2 \\ &= \rho\left[(A^{1/2}BA^{-1/2})^T(A^{1/2}BA^{-1/2})\right]^{1/2} \\ &= \rho(A^{-1/2}BABA^{-1/2})^{1/2}.\end{aligned}$$

$\square$

4.2.5 Definition
Sei $A \in \mathbb{R}^{n\times n}$ mit $A = P - Q$, $P, Q \in \mathbb{R}^{n\times n}$. Dann heißt die Zerlegung $A = P - Q$ **P-regulär**, falls $P^T + Q$ positiv definit ist, d.h. es gilt
$$\left\langle x, (P^T + Q)x\right\rangle > 0 \quad \forall\, x \neq 0.$$

4.2.6 Satz
Es sei $A \in \mathbb{R}^{n\times n}$ eine spd-Matrix und die Zerlegung $A = P - Q$ sei P-regulär. Dann folgt
$$\|P^{-1}Q\|_A < 1, \text{ also auch } \rho(P^{-1}Q) < 1.$$

Beweis: Es gilt

$$\begin{aligned} P^{-1}Q &= P^{-1}(P-A) = I - P^{-1}A \\ \|P^{-1}Q\|_A^2 &= \|A^{1/2}P^{-1}QA^{-1/2}\|_2^2 \\ &= \|A^{1/2}(I-P^{-1}A)A^{1/2}\|_2^2 \\ &= \rho(\underbrace{A^{-1/2}(I-P^{-1}A)^T A(I-P^{-1}A)A^{-1/2}}_{=:B}). \end{aligned}$$

B ist symmetrisch und positiv semidefinit. Also sind alle Eigenwerte von B nicht negativ. Betrachte nun $I-B$:

$$\begin{aligned} I-B &= I-[I-A^{1/2}P^{-T}AP^{-1}A^{1/2}-A^{1/2}P^{-T}A^{1/2}-A^{1/2}P^{-1}A^{1/2}] \\ &= A^{1/2}P^{-T}[-A+P+P^T]P^{-1}A^{1/2} \\ &= A^{1/2}P^{-T}[P^T+Q]P^{-1}A^{1/2}. \end{aligned}$$

Mit P^T+Q ist auch $I-B$ positiv definit. Also liegen alle Eigenwert von $I-B$ in $(0,+\infty)$. Insgesamt folgt $\text{spek}(B) \subseteq [0,1)$ und damit $\rho(B) < 1$. □

4.2.7 Korollar
Sei $A \in \mathbb{R}^{n\times n}$ spd. Dann konvergiert das Gauß-Seidel-Verfahren, d.h. es gilt

$$\rho((D-L)^{-1}U) < 1.$$

Beweis: Da A spd-Matrix ist, folgt $a_{i,i} > 0, L = U^T$. Setze $P = D-L$ und $Q = U = L^T$. Dann ist P regulär und es gilt

$$P^T + Q = D^T - L^T + L^T = D^T = D,$$

also ist P^T+Q spd-Matrix. Also gilt nach Satz 4.2.6

$$\rho((D-L)^{-1}U) < 1.$$

□

Beispiel Modellproblem I:
Zunächst muss eine Reihenfolge für die Unbekannten $u_{i,j}$ festgelegt werden. Der Vektor der Unbekannten ist also wie immer beim MP I mit dem 2-dimensionalen Index (i,j) versehen. Eine zeilenweise (= lexikographische) Anordnung ergibt sich dann folgendermaßen

$$(i,j) \lesssim (i',j') \Longleftrightarrow i < i' \text{ oder } (i = i' \text{ und } j < j').$$

Mit dieser Anordnung lautet dann ein Schritt des Gauß-Seidel-Verfahrens

4.2.8 Algorithmus (Gauß-Seidel für MP I, zeilenweise Anordnung)

for $k = 0, 1, \ldots$ **do**
  for $i = 1$ to N **do**
    for $j = 1$ to N **do**
      $u_{i,j}^{(k+1)} = \frac{1}{4}\left(h^2 f_{i,j} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right)$
    end for
  end for
end for

Dieses Verfahren ist aber schlecht parallelisierbar (man kann nur „Diagonalen" der $u_{i,j}$ parallel aufdatieren).

Aufgabe: Finde eine Anordnung der Unbekannten, so dass eine einfache parallele Ausführung des Gauß-Seidel-Verfahrens möglich ist!

Formal: Sei P Permutationsmatrix, wende das Gauß-Seidel-Verfahren auf

$$PAP^T x = Pb$$

an. Ist A spd, dann ist auch PAP^T spd, also konvergiert das Gauß-Seidel-Verfahren (*aber:* $\rho((D-L)^{-1}U)$ kann von der Permutationsmatrix P abhängen und damit auch die Konvergenzgeschwindigkeit).

Beispiel: Die **Red-Black (Schachbrett)-Anordnung** für Modellproblem I

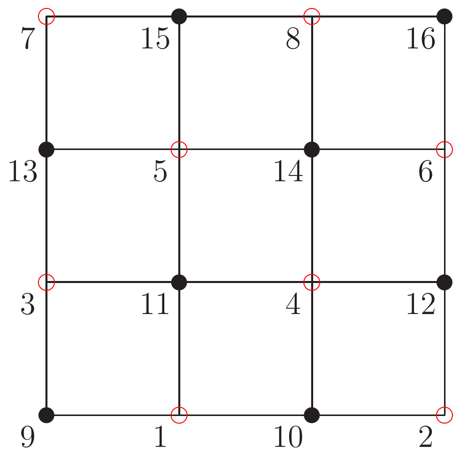


Abbildung 4.1: Red-Black-Anordnung für Modellproblem I

Red-Black-Anordnung:

Nummeriere zuerst alle roten, dann alle schwarzen Gitterpunkte. Damit koppeln im Gleichungssystem für das Modellproblem I rote Gitterpunkte nur noch mit schwarzen und umgekehrt. Mit dieser Anordnung ergibt sich ein Schritt des Gauß-Seidel-Verfahrens wie folgt:

4.2.9 Algorithmus (Gauß-Seidel für MP I, Red-Black-Anordnung)

for $k = 0, 1, \ldots$ **do**
  for all (i, j) mit (ih, jh) ist roter Punkt **do**
    $u_{i,j}^{(k+1)} = \frac{1}{4}\left(h^2 f_{i,j} + u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right)$
  end for
  for all (i, j) mit (ih, jh) ist schwarzer Punkt **do**
    $u_{i,j}^{(k+1)} = \frac{1}{4}\left(h^2 f_{i,j} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k+1)} + u_{i,j+1}^{(k+1)}\right)$
  end for
end for

4.2.10 Bemerkung
Für das Modellproblem I kann man zeigen, dass

$$\underbrace{\rho((D-L)^{-1}U)}_{\text{Gauß-Seidel}} = \underbrace{\rho(D^{-1}(L+U))^2}_{\text{Jacobi}}$$

sowohl bei zeilenweiser wie auch bei Red-Black-Anordung gilt.

4.2.11 Definition
Für ein konvergentes Iterationsverfahren der Gestalt

$$x^{(k+1)} = Hx^{(k)} + c, \quad H \in \mathbb{R}^{n \times n}, c \in \mathbb{R}^n, \rho(H) < 1 \tag{4.2.4}$$

heißt $-\log(\rho(H))$ die **asymptotische Konvergenzgeschwindigkeit**.

Interpretation: Es ist bekannt, dass für jede Operaternorm

$$\rho(H) = \lim_{k\to\infty} \left(\|H^k\|\right)^{\frac{1}{k}}$$

gilt. Aus (4.2.4) folgt mit

$$x^* = Hx^* + c \text{ und } e^{(k)} = x^{(k)} - x^*,$$

dass auch

$$e^{(k)} = He^{(k-1)} = H^2 e^{(k-2)} = \ldots = H^k e^{(0)}$$

und damit

$$\|e^{(k)}\| \leq \underbrace{\|H^k\|}_{\approx \rho^k} \cdot \|e^{(0)}\|$$

gilt. Damit ist dann

$$\frac{\|e^{(k)}\|}{\|e^{(0)}\|} \leq \epsilon, \text{ falls } \rho^k \leq \epsilon \Longleftrightarrow k \geq \frac{\log(\epsilon)}{\log(\rho)}.$$

Bemerkung: 4.2.10 kann dann für das Modellproblem I so interpretiert werden, dass das Gauß-Seidel-Verfahren doppelt so schnell konvergiert wie das Jacobi-Verfahren.

Wir haben also eine Beschleunigung um den Faktor 2 erreicht, d.h. benötigt das Jacobi-Verfahren 2000 Schritte zum Unterschreiten einer vorgegebenen Fehlerschranke, so genügen mit dem Gauß-Seidel-Verfahren wohl lediglich 1000 Iterationsschritte.

Wir wollen nun ein weiteres Verfahren einführen, welches die Konvergenzgeschwindigkeit substanziell erhöht!

Beschleunigung des Gauß-Seidel-Verfahrens durch „*Überrelaxation*“

$$x_i^{(k+1)} = \frac{\omega}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} \qquad i = 1, ..., n, \quad k = 0, 1, ... \tag{4.2.5}$$

Dies ist also eine Art Mittelwertbildung mit den Gewichten ω und $(1-\omega)$.
Bemerkung: Die Mittelwertbildung geht hierbei sogar rekursiv ein, denn zur Berechnung der $x_j^{(k+1)}$ in der ersten Summe wurden ebenfalls bereits Gewichte benutzt.

Ist nun $A = D - L - U$ eine Zerlegung einer Matrix $A \in \mathbb{R}^{n \times n}$, so kann man die Iterationsmatrix dieses Verfahrens bestimmen durch

$$\begin{aligned}
& \frac{1}{\omega} D x^{(k+1)} = b + L x^{(k+1)} + U x^{(k)} + \frac{1-\omega}{\omega} D x^{(k)} \\
\Longleftrightarrow \quad & \left(\frac{1}{\omega} D - L \right) x^{(k+1)} = b + \left(U + \frac{1-\omega}{\omega} D \right) x^{(k)} \\
\Longleftrightarrow \quad & x^{(k+1)} = \left(\frac{1}{\omega} D - L \right)^{-1} \left(b + \left(U + \frac{1-\omega}{\omega} D \right) x^{(k)} \right)
\end{aligned}$$

4.2.12 Definition
Für $\omega \neq 0$ heißt die Iteration (4.2.5) **SOR-Iteration** (Sucessive OverRelaxation). Zu ihr gehört die Zerlegung

$$A = \underbrace{\left(\frac{1}{\omega} D - L \right)}_{=:P} - \underbrace{\left(U + \frac{1-\omega}{\omega} D \right)}_{=:Q}$$

mit
$$x^{(k+1)} = P^{-1}(Qx^{(k)} + b).$$

4.2.13 Satz
Es sei $A \in \mathbb{R}^{n\times n}$ eine spd-Matrix und $\omega \in (0,2)$. Dann gilt

$$\rho(H_\omega) < 1 \text{ mit } H_\omega = \left(\frac{1}{\omega}D - L\right)^{-1}\left(U + \frac{1-\omega}{\omega}D\right).$$

Beweis: Folgt aus Satz 4.2.6, denn

$$P^T + Q = \left(\frac{1}{\omega}D - U\right) + \left(U + \frac{1-\omega}{\omega}D\right) = \frac{2-\omega}{\omega}D$$

ist positiv definit für $\omega \in (0,2)$. □

Man kann außerdem zeigen, dass für $\omega \notin (0,2)$ auch $\rho(H_\omega) \geq 1$ gilt (d.h. es gibt dann Startwerte, für die das Verfahren nicht konvergiert).

Was ist die beste Wahl für ω?

4.2.14 Satz
Für das Modellproblem I gilt bei zeilenweiser oder Red-Black-Anordnung

$$\rho(H_\omega) \text{ ist minimal für } \omega = \omega_0 = \frac{2}{1+\sqrt{1-\beta^2}}, \beta = \rho(D^{-1}(L+U)).$$

Dabei ist dann $\rho(H_{\omega_0}) = \omega_0 - 1$.

Beweis: Siehe Numerik. □

Bemerkung: Der Satz ist auf Matrizen übertragbar, für welche es eine Red-Black-Anordnung gibt!

4.2.15 Bemerkung
Für das Modellproblem I ergibt sich

Jacobi:	$\rho(D^{-1}(L+U)) = \cos(\pi h) = 1 - \frac{1}{2}\pi^2h^2 + \mathcal{O}(h^4)$
Gauß-Seidel:	$\rho((D-L)^{-1}U) = \cos^2(\pi h) = 1 - \pi^2h^2 + \mathcal{O}(h^4)$
SOR:	$\rho(H_{\omega_0}) = \frac{2}{1+\pi h+\mathcal{O}(h^2)} - 1 = 1 - 2\pi h + \mathcal{O}(h^2)$.

Für die asymptotische Konvergenzgeschwindigkeit erhalten wir damit (mit $\log(1+x) = x + \mathcal{O}(x^2)$):

$$\begin{aligned} \text{Jacobi:} \quad & \pi^2h^2 + \mathcal{O}(h^4) \\ \text{Gauß-Seidel:} \quad & \frac{1}{2}\pi^2h^2 + \mathcal{O}(h^4) \\ \text{SOR:} \quad & 2\pi h + \mathcal{O}(h^2). \end{aligned}$$

Das SOR-Verfahren erreicht also eine Verbesserung im Exponenten bei der Abhängigkeit von h. Dies ist wesentlich mehr als der Faktor 2 von Gauß-Seidel gegenüber Jacobi.

Für das Modellproblem I lautet die k-te Iteration des SOR-Verfahrens bei Red-Black-Anordnung

for all rote Punkte (ih, jh) **do**

$$u_{i,j}^{(k+1)} = \frac{\omega}{4}\left(u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)}\right) + (1-\omega)u_{i,j}^{(k)}$$

end for

for all schwarze Punkte (ih, jh) **do**

$$u_{i,j}^{(k+1)} = \frac{\omega}{4}\left(u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k+1)} + u_{i,j+1}^{(k+1)}\right) + (1-\omega)u_{i,j}^{(k)}$$

end for

Damit kann man nun das parallele SOR-Verfahren formulieren: Jeder Prozessor bearbeitet nur seine Gitterpunkte aus einer Menge *myregion*.

4.2.16 Algorithmus (SOR für MP I, Red-Black, parallel)

for $k = 0, 1, \ldots$ **do**
  tausche schwarze Punkte aus
  for $(ih, jh) \in myregion$ **do**
    if (ih, jh) is red **then**
      update $u_{i,j}$
    end if
  end for
  tausche rote Punkte aus
  for $(ih, jh) \in myregion$ **do**
    if (ih, jh) is black **then**
      update $u_{i,j}$
    end if
  end for
end for

Bemerkung:

1. k_{stop} ist die maximale Anzahl der Iterationsschritte.

2. *myregion* ist das „Teilgebiet", welches einem Prozessor zugeordnet ist.

3. $u_{i,j}^{(k)}$ kann und sollte immer sofort mit $u_{i,j}^{(k+1)}$ übeschrieben worden (spart Speicherplatz). Dieses Überschreiben entspricht dem obigen Aufruf von update.

4. Die Färbung der Knoten in der Red-Black-Anordnung kann folgendermaßen geschehen

$$(ih, jh) \text{ is black} \iff \begin{array}{l}(i \equiv 0 \mod 2 \text{ und } j \equiv 0 \mod 2) \text{ oder} \\ (i \equiv 1 \mod 2 \text{ und } j \equiv 1 \mod 2)\end{array}$$

$$(ih, jh) \text{ is red} \iff \begin{array}{l}(i \equiv 1 \mod 2 \text{ und } j \equiv 0 \mod 2) \text{ oder} \\ (i \equiv 0 \mod 2 \text{ und } j \equiv 1 \mod 2).\end{array}$$

5. Kommunikationszeiten entstehen durch das Austauschen der Punkte. Allerdings müssen tatsächlich nicht alle Punkte getauscht werden. Prinzipiell müssen dieselben Variablen ausgetauscht werden wie bei der Matrix-Vektor-Multiplikation mit der Matrix A, jedoch diesmal in zwei Etappen (nach Farben).

Beispiel:

Wir betrachten die ein- und zweidimensionale Aufteilung des Gitters aus Modellproblem I und die dabei auftretenden Kommunikationen. In Abbildung 4.2 ist für die eindimensionale Aufteilung eingetragen, welche Variablen von `myregion` aus versendet werden müssen
Dabei werden im ersten Schritt nur die roten Punkte (rote Pfeile) und dann im zweiten Teilschritt die schwarzen versendet. In jedem Teilschritt muss an beide benachbarten Prozessoren, den oberen und den unteren, gesendet werden.

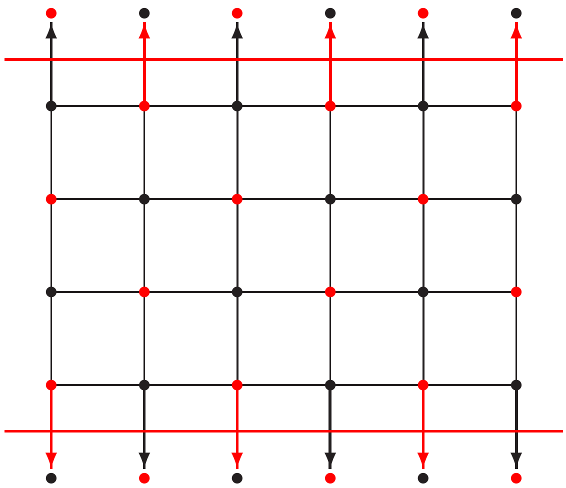


Abbildung 4.2: Eindimensionale Aufteilung bei Red-Black: Kommunikation

Bei der zweidimensionalen Aufteilung ergibt sich für das Versenden das Schema aus Abbildung 4.3.

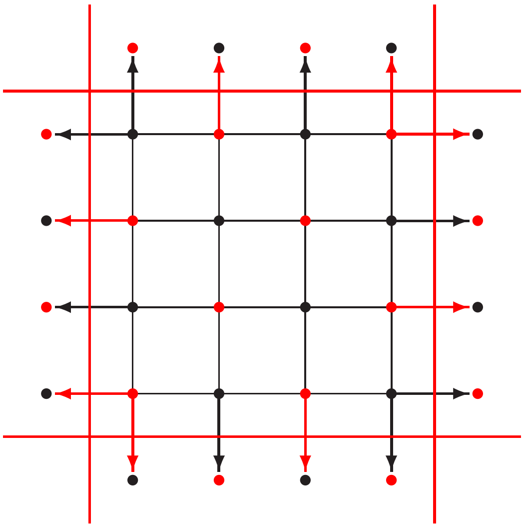


Abbildung 4.3: Zweidimensionale Aufteilung bei Red-Black: Kommunikation

Wieder werden rote und schwarze Punkte in getrennten Phasen kommuniziert. In jeder Phase wird an vier benachbarte Prozessoren (oben, unten, links, rechts) gesendet,
Man beachte dabei, dass der Algorithmus zur eindimensionalen Aufteilung *nicht skaliert* (siehe Übung)! Hingegen müssen bei der zweidimensionalen Aufteilung pro Schritt acht Kommunikationen in vier Richtungen angestoßen werden. Bei der eindimensionalen sind dies nur vier. (Durch geschicktes Zusammenfassen kann man hier sogar erreichen, dass nur zwei Kommunikationen von *vollständigen* Zeilen notwendig werden).

Jetzt wollen wir die Idee der Red-Black-Einteilung auf Matrizen mit komplexeren Besetztheitsstrukturen übertragen.

4.2.17 Definition
Es sei $G = (V, E)$ ein Graph. Eine Partition

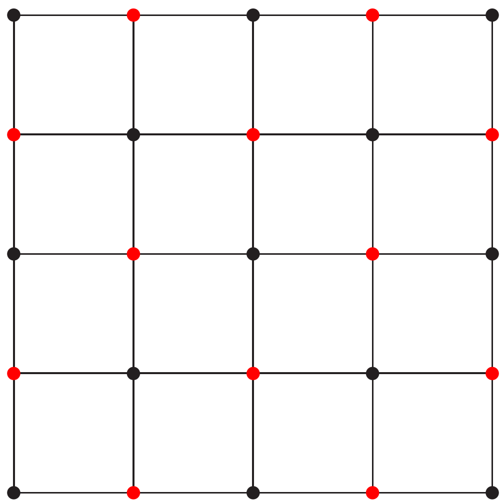
$$V = \bigcup_{j=1}^{c} V_j$$

heißt eine **Knotenfärbung** von G, falls

$$i, j \in V_\ell \text{ und } i \neq j \Rightarrow \{i, j\} \notin E \quad \ell = 1, ..., n.$$

In V_ℓ gibt es also keine benachbarten Knoten.

4.2.18 Beispiel
Es sei A die Matrix des Modellproblems I und $G = G(A)$.



$c = 2,\ V_1 = \{\bullet\},\ V_2 = \{\bullet\}$

Red-Black-Färbung

Abbildung 4.4: Red-Black-Färbung: Modellproblem I

Sei A die Matrix des Modellproblem II und $G = G(A)$.

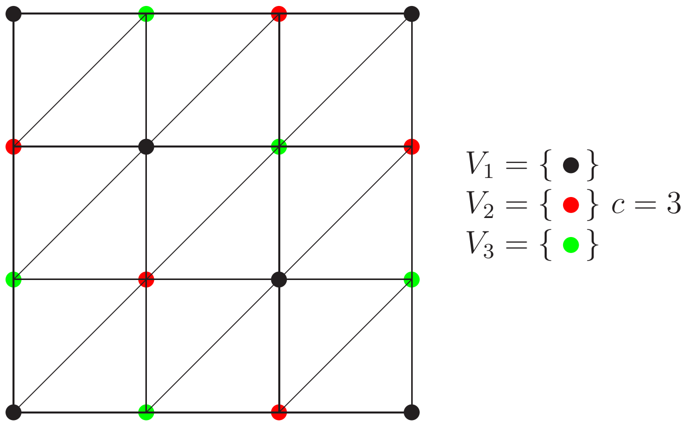


Abbildung 4.5: Red-Black-Green-Färbung: Modellproblem II

Die Idee ist es nun, die Matrix A gemäß den Farben zu permutieren. Sei also $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix,

$$G = G(A) = (V, E) \text{ und } V = \bigcup_{j=1}^{c} V_j$$

eine Knotenfärbung von V. Sei P eine Permutationsmatrix, welche A „nach Farben sortiert", d.h.

$$Px = \begin{pmatrix} X_1 \\ \vdots \\ X_c \end{pmatrix} \qquad \text{wobei } X_j = \{x_\ell |\ \ell \in V_j\}.$$

Damit besitzt dann die Matrix PAP^T die folgende Gestalt

$$PAP^T = \begin{pmatrix} \boxed{{}_0\diagdown^0} & \cdots & \cdots & \boxed{} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \boxed{} & \cdots & \cdots & \boxed{{}_0\diagdown^0} \end{pmatrix} = \begin{pmatrix} D_1 & A_{1,2} & \ldots & A_{1,c} \\ A_{2,1} & D_2 & & \vdots \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \vdots & & & A_{c,c-1} \\ A_{c,1} & \ldots & A_{c-1,c} & D_c \end{pmatrix},$$

mit Diagonalmatrizen $D_j,\ j = 1, ..., c$. Nun wenden wir das Gauß-Seidel-Verfahren auf

$$PAP^T x = Pb$$

an. Dies ergibt:

4.2.19 Algorithmus (c-Farben Gauß-Seidel, seriell)

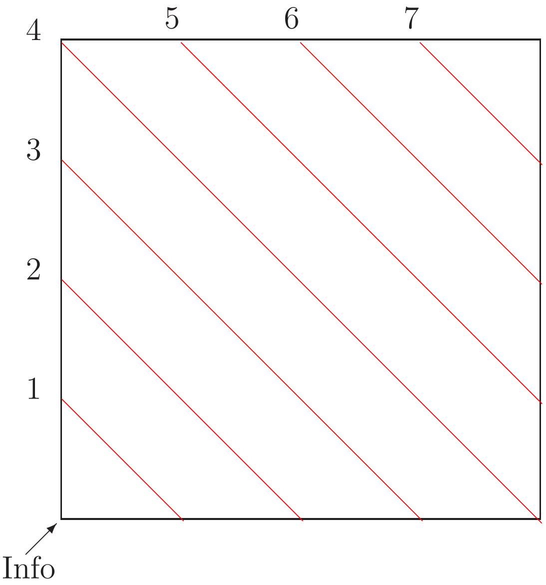
for $k = 1$ to k_{stop} **do**
  for $i = 1$ to c **do**
    for $j \in V_i$ **do**
      update x_j } parallele Ausführung möglich
    end for
  end for
end for

Bei der parallelen Variante bezeichne *myregion* wieder die Komponenten der Iterierten, welche einem bestimmten Prozessor zugeordnet sind.

4.2.20 Algorithmus (c-Farben Gauß-Seidel, parallel)

for $k = 1$ to k_{stop} **do**
  for $i = 1$ to c **do**
    for $j \in V_i \cap myregion$ **do**
      update x_j
    end for
    tausche die benötigten Variablen der Farbe i aus
  end for
end for

Der Gewinn des Entkoppelns durch eine Färbung wird in der Regel durch „gewisse" Geschwindigkeitseinbußen bezahlt werden. Heuristisch ist dies dadurch zu erklären, dass bei einer Anordnung nach Färbung in einem SOR-Schritt keinerlei Information zwischen Punkten gleicher Farbe ausgetauscht wird. Im Modellproblem I ist dagegen bei zeilenweise Abspeicherung neue Information bereits nach *einem* Schritt vom ersten Punkt bis zum letzten weitergetragen worden (in umgekehrter Richtung allerdings nicht ...)! Die asymptotische Konvergenzgeschwindigkeit $\rho((D - L^{-1})U)$ im Gauß-Seidel- und SOR-Verfahren kann (muss aber nicht) ebenfalls von der Anordnung abhängen.



bei Red-Black-Nummerierung

Wir fassen zusammen: Das parallele SOR-Verfahren besitzt in der Regel eine langsamere Informationsweitergabe. Gleichzeitig ist es häufig schwierig, tatsächlich eine konkrete Einfärbung zu finden.

Neue Idee: Wähle einen Kompromiss zwischen zeilenweiser Anordnung und Red-Black-Anordnung!

- Innerhalb des Gitters *eines* Prozessors verwende möglichst zeilenweise Anordnung (schnelle Informationsweitergabe innerhalb eines Prozessors).
- Am Rand Entkoppeln durch Einfärben.

Für eine parallele Ausführbarkeit genügt es, wenn die Knoten einer Farbe entkoppelt sind von den Knoten derselben Farbe in *anderen Prozessoren*! Eine formale Beschreibung verwendet *Blockfärbungen.*

4.2.21 Definition
Gegeben sei ein Graph $G = (V, E)$ mit Partition der Knoten

$$V = \bigcup_{j=1}^{p} V_j.$$

Eine **Block-Färbung** mit c Farben ist eine Aufteilung

$$V_i = \bigcup_{j=1}^{c} V_{i,j} \text{ mit } V_{i,j} \cap V_{i,\ell} = \emptyset \text{ falls, } j \neq \ell \quad i = 1, ..., p$$

so dass für $j = 1, ..., c$ gilt

$$k \in V_{i,j},\ m \in V_{\ell,j} \text{ mit } i \neq \ell \Longrightarrow (k,m) \notin E.$$

Bemerkung:

1. Wir verlangen explizit *nicht*, dass alle $V_{i,j} \neq \emptyset$.
2. Die letzte Bedingung bedeutet, dass ein Knoten mit Knoten derselben Farbe in anderen Prozessoren nicht verbunden ist!

Damit ergibt sich als zugehöriger paralleler Algorithmus:

4.2.22 Algorithmus (Gauß-Seidel für Blockfärbung, parallel)

Require: $myregion = V_i$ in Prozessor P_i
  if $me = P_i$ **then**
    for $k = 1$ to k_{stop} **do**
      for $j = 1$ to c **do**
        for $\ell \in V_{i,j}$ **do**
          update x_ℓ
        end for
        tausche die benötigten Variablen der Farbe j aus
      end for
    end for
  end if

Beispiel: Für Modellproblem I und II kann man die in Abbildung 4.6 eingezeichnete Blockfärbung mit 4 Farben verwenden.

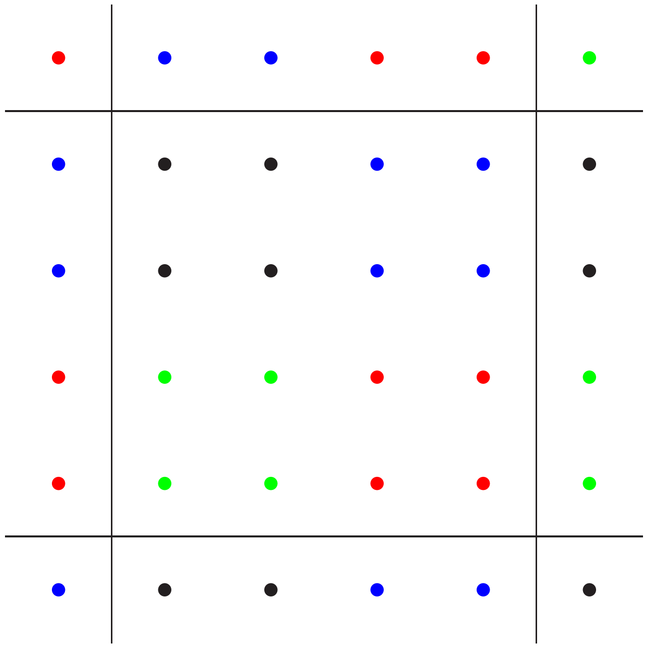


Abbildung 4.6: Block-Färbung: Modellproblem I

Wie Abbildung 4.7 zeigt, kann man beispielsweise zunächst die schwarzen Variablen berechnen, dann die grünen und *dann* erst die gesamte linke Spalte zum linken Nachbarn senden. Waren bei der Red-Black-Anordnung noch 8 Kommunikationen notwendig, so sind es hier so nur noch 4.

Vorteil: Bei zweidimensionalen Diskretisierungen mit geeigneter Partition genügen immer vier Farben (egal wie komplex deren Stern ist), wenn jedes *myregion* nur „breit genug“ ist.

Eine Farbe kann innerhalb eines Prozessors z.B. zeilenweise durchlaufen werden. Damit werden Informationen schneller weitergegeben als bei der normalen (Nicht-Block-) Färbung.

Zum Abschluss wollen wir das SOR-Verfahren noch weiter modifizieren. Auch wenn die Matrix A symmetrisch ist, ist nach Zerlegung die Iterationsmatrix $(D-L)^{-1}U$ i.Allg. nicht symmetrisch! Um dies zu vermeiden, betrachten wir nun das sogenannte SSOR-Verfahren, das *symmetric SOR-Verfahren*.

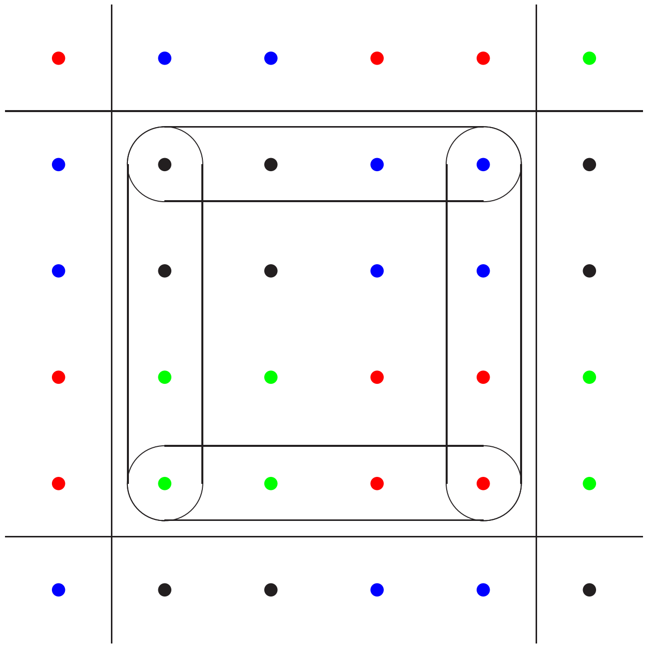


Abbildung 4.7: Block-Färbung: Modellproblem I (Fortsetzung)

4.2.23 Definition
Das Iterationsverfahren

$$forward: \; x_i^{(k+\frac{1}{2})} = \frac{\omega}{a_{i,i}}\Big(b_i - \underbrace{\sum_{j=1}^{i-1} a_{i,j}x_j^{(k+\frac{1}{2})}}_{(1)} - \sum_{j=i+1}^{n} a_{i,j}x_j^{(k)}\Big) + (1-\omega)x_i^{(k)}$$

$$i = 1, ..., n$$

$$backward: \; x_i^{(k+1)} = \frac{\omega}{a_{i,i}}\Big(b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+\frac{1}{2})} - \underbrace{\sum_{j=i+1}^{n} a_{i,j}x_j^{(k+1)}}_{(2)}\Big) + (1-\omega)x_i^{(k+\frac{1}{2})}$$

$$i = n, n-1, ..., 1$$

heißt *SSOR-Verfahren.*

Das SSOR-Verfahren hat nur einen geringen Mehraufwand gegenüber dem SOR-Verfahren, auch wenn zunächst scheinbar doppelt soviele Operationen pro Schritt nötig werden. Der Term (1) aus dem $forward$-Schritt kann im *backward*-Schritt verwendet werden. Gleichzeitig kann der Term (2) aus dem *backward*-Schritt im $forward$-Schritt des nächsten Schrittes verwendet werden. Der Mehraufwand gegenüber SOR reduziert sich also auf den ersten forward-Schritt und den letzten backward-Schritt.

Das SSOR-Verfahren kann in der folgenden Matrix-Form notiert werden

$$\begin{aligned} x^{(k+\frac{1}{2})} &= \left(\frac{1}{\omega}D - L\right)^{-1} \left(\left(U + \frac{1-\omega}{\omega}D\right) x^{(k)} + b\right), \\ x^{(k+1)} &= \left(\frac{1}{\omega}D - U\right)^{-1} \left(\left(L + \frac{1-\omega}{\omega}D\right) x^{(k+\frac{1}{2})} + b\right). \end{aligned}$$

Als eine geschlossene Form für $x^{(k+1)}$ ergibt sich

$$\begin{aligned} x^{(k+1)} = & \underbrace{\left(\frac{1}{\omega}D - U\right)^{-1} \left(L + \frac{1-\omega}{\omega}D\right) \left(\frac{1}{\omega}D - L\right)^{-1} \left(U + \frac{1-\omega}{\omega}D\right)}_{=:H} x^{(k)} \\ & + \left[\left(\frac{1}{\omega}D - U\right)^{-1} + \left(\frac{1}{\omega}D - U\right)^{-1} \left(L + \frac{1-\omega}{\omega}D\right) \left(\frac{1}{\omega}D - L\right)^{-1}\right] b. \end{aligned}$$

4.2.24 Lemma
Zur SSOR-Iteration gehört die Zerlegung $A = P - Q$ mit

$$\begin{aligned} P &= \left(\frac{1}{\omega}D - L\right) \frac{\omega}{2-\omega} D^{-1} \left(\frac{1}{\omega}D - U\right) \\ Q &= \left(\frac{1-\omega}{\omega}D + L\right) \frac{\omega}{2-\omega} D^{-1} \left(\frac{1-\omega}{\omega}D + U\right), \end{aligned}$$

d.h. es gilt $A = P - Q$ und $H = P^{-1}Q$.

Beweis:

1. Durch Ausmultiplizieren und Zusammenfassen ergibt sich
$$A = P - Q = D - L - U.$$

2. Es ist
$$P^{-1} = \left(\frac{1}{\omega}D - U\right)^{-1} \frac{2-\omega}{\omega} D \left(\frac{1}{\omega}D - L\right)^{-1}$$
und
$$\begin{aligned} & \left(\frac{1}{\omega}D - U\right)^{-1} + \left(\frac{1}{\omega}D - U\right)^{-1} \left(L + \frac{1-\omega}{\omega}D\right) \left(\frac{1}{\omega}D - U\right)^{-1} \\ = & \left(\frac{1}{\omega}D - U\right)^{-1} \frac{2-\omega}{\omega} D \left(\frac{1}{\omega} - L\right)^{-1}. \end{aligned}$$

3. Es ist

$$\begin{aligned}
&\left(L+\frac{1-\omega}{\omega}D\right)\left(\frac{1}{\omega}D-L\right)^{-1}\\
= \ & D\left(D^{-1}L+\frac{1-\omega}{\omega}I\right)\left(\frac{1}{\omega}I-D^{-1}L\right)^{-1}D^{-1}\\
= \ & D\left[\left(\frac{1}{\omega}I-D^{-1}L\right)\cdot\left(D^{-1}L+\frac{1-\omega}{\omega}I\right)\right]D^{-1}.
\end{aligned}$$

Also:

$$\begin{aligned}
H &= \left(\frac{1}{\omega}D-U\right)^{-1}D\left(\frac{1}{\omega}I-D^{-1}L\right)^{-1}\left(D^{-1}L+\frac{1-\omega}{\omega}I\right)\\
&\qquad D^{-1}\left(U+\frac{1-\omega}{\omega}D\right)\\
&= \left(\frac{1}{\omega}D-U\right)^{-1}D\left(\frac{1}{\omega}D-L\right)^{-1}\left(L+\frac{1-\omega}{\omega}D\right)\\
&\qquad D^{-1}\left(U+\frac{1-\omega}{\omega}D\right)\\
&= P^{-1}\frac{\omega}{\omega-2}\left(L+\frac{1-\omega}{\omega}D\right)D^{-1}\left(U+\frac{1-\omega}{\omega}D\right)\\
&= P^{-1}Q
\end{aligned}$$

□

4.2.25 Satz
Sei A spd. Für $\omega \in (0,2)$ gilt

$$\rho(H) < 1,$$

d.h. das SSOR-Verfahren konvergiert.

Beweis: Wir zeigen, dass die Iterationsmatrix $P^T + Q$ des SSOR-Verfahrens spd, ist, dann folgt nach Satz 4.2.6 die Behauptung. Es ist $A = A^T$ und damit folgt $U = L^T$, also auch $P = P^T$ und

$$P^T + Q = P + Q$$

mit

$$P = \left(\frac{1}{\omega}D-L\right)\frac{\omega}{2-\omega}D^{-1}\left(\frac{1}{\omega}D-U\right)$$

$$Q = \left(\frac{1-\omega}{\omega}D + L\right)\frac{\omega}{2-\omega}D^{-1}\left(\frac{1-\omega}{\omega}D + U\right).$$

Hier ist P spd, weil P von der Form $T^T D' T$ mit T regulär ist (siehe **Tatsachen** direkt vor Lemma 4.2.3). Außerdem ist Q spd für $\omega \neq 1$ und positiv semidefinit für $\omega = 1$. Also ist $P + Q$ für $\omega \in (0, 2)$ spd. $\square$

4.2.26 Bemerkung
Man kann zeigen, dass für $\omega \notin (0, 2)$ das SSOR-Verfahren divergiert. Für das Modellproblem I wird SOR und SSOR bei optimaler Wahl für ω gleich schnell. Der Fortschritt gegenüber dem Gauß-Seidel-Verfahren ist enorm. Ein Vergleich der Laufzeiten bei Implementierung in MATLAB und der nötigen Iterationsanzahl in Abhängigkeit von der Gittergröße N zeigt folgende Tabelle

	SOR		J		GS	
N	k	time	k	time	k	time
10	25	0	216	0,05	98	0,44
20	49	0,17	621	0,71	274	4,22
40	99	0,66	1690	4,89	710	42,95
80	200	3,4	3959	34.6	2076	486,09

(Hier wurde beim SOR-Verfahren stets ω_{opt} benutzt.)

Der Vorteil von SSOR gegenüber SOR liegt darin, dass P in $A = P - Q$ mit $H = P^{-1}Q$ spd ist. Dies benötigt man bei einer Verwendung von P als Präkonditionierer, siehe später.

4.3 Das Verfahren der konjugierten Gradienten (conjugate gradients = CG)

Dieses Verfahren ist Grundlage vieler, moderner, paralleler Iterationsverfahren. Charakteristisch ist, dass jeder Iterationsschritt im Wesentlichen eine Multiplikation mit der Matrix A benötigt. Entwickelt wurde es 1952 von Stiefel und Hestenes, dennoch dauerte es bis 1977 bis man die Mächtigkeit des Verfahrens wirklich eingesehen hatte.

Wir beginnen zunächst einmal damit, das Verfahren herzuleiten und schließen dann eine Konvergenzanalyse an. Man beachte, dass für den Aufwand die Matrix-Vektormultiplikationen entscheidend sind. Vom übrigen Aufwand sind die benötigten Innenprodukte auf parallelen Architekturen nicht vorteilhaft, da fan-ins nötig werden. Das CG-Verfahren ist parameterfrei, d.h. man muss nicht, wie z.B. beim SOR, bestimmte Parameter wie ω_{opt} berechnen.

Ausgangsituation ist wieder

$$Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad A \text{ spd.} \tag{4.3.1}$$

4.3.1 Lemma
Es sei

$$\begin{aligned} f: \mathbb{R}^n &\to \mathbb{R} \\ x &\mapsto \frac{1}{2}\langle x, Ax\rangle - \langle x, b\rangle . \end{aligned}$$

Dann ist x eine Lösung von $Ax = b$ genau dann, wenn x (das eindeutige, globale) Minimum von f ist.

Beweis: Zu jedem $\alpha \in \mathbb{R}$ gibt es ein $\beta > 0$, so dass

$$\|x\| > \beta \Rightarrow f(x) > \alpha$$

(Übung). Also besitzt f ein globales Minimum, welches auch lokales Minimum ist. Weiterhin gilt

$$\nabla f(x) = \mathrm{grad} f(x) = (Ax - b)^T.$$

Es gibt also genau einen stationären Punkt von $f(x)$, nämlich

$$x^* = A^{-1}b.$$

Die Hesse-Matrix von f ist nun

$$H_f(x) = A.$$

Da H_f positiv definit ist, ist x^* das eindeutige globale Minimum! □

4.3.2 Definition
Zu $r \in \mathbb{R}^n$ und $A \in \mathbb{R}^{n\times n}$ definieren wir den *Krylov-Unterraum* der Stufe m als[1]

$$\begin{aligned} K_m(A,r) &:= span\{r, Ar, A^2r, ..., A^{m-1}r\} \\ &= \{y \in \mathbb{R}^n :\ y = \sum_{j=0}^{m-1} \alpha_j A^j r\} \\ &= \{y \in \mathbb{R}^n :\ y = p_{m-1}(A)r, \quad p_{m-1} \in \Pi_{m-1}\}, \end{aligned}$$

wobei Π_{m-1} der Vektorraum der Polynome vom Höchstgrad $m-1$ ist, d.h.

$$\Pi_{m-1} := \{p :\ p(t) = \sum_{j=0}^{m-1} a_j t^j\},$$

wobei wir für ein Polynom p den Grad von p durch

$$deg(p) = \min_k \{k :\ p \in \Pi_k\}$$

definieren.

4.3.3 Lemma
Zu $r \in \mathbb{R}^n$ existiert ein $m^* \in \mathbb{N}_0$, $m^* \leq n$ mit

$$\begin{aligned} \dim(K_m(A,r)) &= m, \quad m = 1, ..., m^* \\ K_{m^*}(A,r) &= K_{m^*+1}(A,r) = K_{m^*+2}(A,r) = ... \end{aligned}$$

Beweis: Es ist klar, dass

$$\dim(K_m(A,r)) \leq \dim(K_{m+1}(A,r)) \leq \dim(K_m(A,r)) + 1 \leq n+1$$

gilt, da

$$K_{m+1}(A,r) = K_m(A,r) + \langle A^m r \rangle .$$

Es sei m^* der kleinste Index für den $\dim(K_m(A,r)) = \dim(K_{m+1}(A,r))$ und damit $K_m(A,r) = K_{m+1}(A,r)$ und damit $A^{m^*}r \in K_{m^*}(A,r)$ gilt. (Es ist $m^* \leq n$.) Trivialerweise gilt dann aber

$$\dim(K_m(A,r)) = m, \quad m = 1, ..., m^*$$

und

$$A^{m^*} r = q(A)r, \quad q \in \Pi_{m^*-1}. \tag{4.3.2}$$

[1] $span\{v_1, ..., v_n\}$ ist der Unterraum, der von den Vektoren $v_1, ..., v_n$ aufgespannt wird.

Dies bedeutet
$$K_{m^*}(A,r) = K_{m^*+1}(A,r),$$
und für $m > m^*$ gilt dann außerdem
$$A^m r = A^{m-m^*} \cdot \underbrace{A^{m^*} r}_{\in K_{m^*}(A,r)} \in K_m(A,r),$$
also ergibt sich für diese m
$$K_{m+1}(A,r) = K_m(A,r) + \langle A^m, r\rangle = K_m(A,r).$$
□

4.3.4 Lemma
Sei $x^{(0)} \in \mathbb{R}^n$, $r^{(0)} = b - Ax^{(0)}$. Dann erfüllt die Lösung $x^* = A^{-1}b$ von (4.3.1) die Beziehung
$$x^* \in x^{(0)} + K_{m^*}(A, r^{(0)}),$$
d.h. es gilt
$$x^* = x^{(0)} + q_{m^*-1}(A)r^{(0)}, \quad q_{m^*-1} \in \Pi_{m^*-1}.$$

Beweis: Nach dem Beweis von Lemma 4.3.3 gilt

(4.3.3) $$A^{m^*} r^{(0)} = q(A)r^{(0)} \quad q \in \Pi_{m^*-1},\ \alpha_{m^*-1} \neq 0.$$

Sei $q(t) = \alpha_0 + \alpha_1 t + ... + \alpha_{m^*-1}t^{m^*-1}$. Dann ist $\alpha_0 \neq 0$, denn anderfalls wäre (nach Multiplikation von (4.3.3) mit A^{-1})
$$A^{m^*-1}r^{(0)} = (\alpha_1 I + \alpha_2 A + ... + \alpha_{m^*-1}A^{m^*-2})r^{(0)},$$
im Widerspruch zur Minimalität von m^*.
Multiplikation von (4.3.3) mit A^{-1} liefert deshalb
$$\begin{aligned} A^{m^*-1}r^{(0)} &= \alpha_0 A^{-1}r^{(0)} + \alpha_1 r^{(0)} + ... + \alpha_{m^*-1}A^{m^*-2}r^{(0)} \\ \iff \underbrace{A^{-1}r^{(0)}}_{=A^{-1}b-x^{(0)}} &= \underbrace{\frac{1}{\alpha_0}\left(A^{m^*-1} - \alpha_{m^*-1}A^{m^*-2} - ... - \alpha_1 I\right) r^{(0)}}_{q_{m^*-1}(A)r^{(0)}}. \end{aligned}$$
Damit folgt aber
$$A^{-1}b = x^{(0)} + q_{m^*-1}(A)r^{(0)}.$$
□

Das CG-Verfahren bestimmt nun die Iterierten $x^{(k)}$ so, dass
$$x^{(k)} \in x^{(0)} + K_k(A, r^{(0)})$$
und

(4.3.4) $$x^{(k)} \text{ minimiert } f(x) \text{ unter allen } x \in x^{(0)} + K_k(A, r^{(0)}).$$

Zwischenüberlegung
Wie löst man Minimierungsaufgaben der Gestalt (4.3.4)?

4.3.5 Lemma
Seien $v^{(1)}, ..., v^{(m)}$ eine Basis eines Unterraums V von $\mathbb{R}^n$ und

$$V_m = [v^{(1)}|....|v^{(m)}] \in \mathbb{R}^{n\times m}.$$

Es sei weiter

$$x = x^{(0)} + \sum_{j=1}^{m} \alpha_j v^{(j)} = x^{(0)} + V_m\alpha, \quad \alpha = (\alpha_1, ..., \alpha_m) \in \mathbb{R}^m.$$

Dann gilt

$$\text{(4.3.5)} \qquad x^{(k)} \text{ minimiert } f(x) \text{ unter allen } x \in x^{(0)} + V.$$

genau dann, wenn

$$\text{(4.3.6)} \qquad (V_m^T A V_m)\alpha = V_m^T r^{(0)}, \quad r^{(0)} = b - Ax^{(0)}.$$

Beweis: Für $x = x^{(0)} + y$ gilt

$$\begin{aligned} f(x^{(0)}+y) = f(x) &= \frac{1}{2}\left\langle x^{(0)}+y, A(x^{(0)}+y)\right\rangle - \left\langle x^{(0)}+y, b\right\rangle \\ &= \underbrace{\frac{1}{2}\left\langle x^{(0)}, Ax^{(0)}\right\rangle - \left\langle x^{(0)}, b\right\rangle}_{\text{unabhängig von } y} \\ &\quad +\frac{1}{2}\left\langle y, Ay\right\rangle + \left\langle x^{(0)}, Ay\right\rangle - \left\langle y, b\right\rangle \\ &= \frac{1}{2}\left\langle x^{(0)}, Ax^{(0)}\right\rangle - \left\langle x^{(0)}, b\right\rangle + \frac{1}{2}\left\langle y, Ay\right\rangle \\ &\quad -\left\langle y, r^{(0)}\right\rangle. \end{aligned}$$

Die Minimierung von f in (4.3.5) ist damit äquivalent zur Minimierung von

$$g(y) = \frac{1}{2}\left\langle y, Ay\right\rangle - \left\langle y, r^{(0)}\right\rangle.$$

Für $y = V\alpha$ gilt

$$g(y) \;=\; \frac{1}{2}\left\langle V_m\alpha, AV_m\alpha\right\rangle - \left\langle V_m\alpha, r^{(0)}\right\rangle$$

$$= \frac{1}{2}\left\langle \alpha, \underbrace{V_m^T A V_m}_{\in\mathbb{R}^{m\times m},\ \mathrm{spd}} \alpha \right\rangle - \left\langle \alpha, \underbrace{V_m^T r^{(0)}}_{\in\mathbb{R}^{m\times m}} \right\rangle.$$

Diese Minimierungs-Aufgabe im $\mathbb{R}^m$ besitzt nach Lemma 4.3.1 die Lösung α^* mit

$$(V_m^T A V_m)\alpha^* = V_m^T r^{(0)}.$$

□

Allerdings ist schon jetzt klar, dass dies nur ein konzeptioneller Ansatz sein kann, denn

$$(V^T A V)\alpha^* = V^T r^{(0)}$$

kann durchaus ein sehr großes System sein, das nur schwer zu lösen ist. Im Falle des CG-Verfahrens mit (4.3.4) ist

$$v^{(1)} = r^{(0)},\ v^{(2)} = Ar^{(0)}, ..., v^{(m)} = A^{m-1}r^{(0)}$$

außerdem *keine* gute (numerische) Basis für den Unterraum $K_m(A, r^{(0)})$. Man nimmt dazu besser eine *Orthogonalbasis* gemäß folgendem Algorithmus.

4.3.6 Algorithmus (Lanczos-Verfahren)

$\beta_1 = \|r^{(0)}\|$
$v^{(1)} = r^{(0)}/\beta_1, \quad v^{(0)} = 0$
for $k = 1, 2, ..., m^* - 1$ **do**

$$\begin{aligned} q^{(k)} &= Av^{(k)} \\ \alpha_k &= \left\langle q^{(k)}, v^{(k)} \right\rangle \\ (*) \quad \widetilde{v}^{(k+1)} &= q^{(k)} - \alpha_k v^{(k)} - \beta_k v^{(k-1)} \\ \beta_{k+1} &= \|\widetilde{v}^{(k+1)}\| \\ v^{(k+1)} &= \widetilde{v}^{(k+1)}/\beta_{k+1} \end{aligned}$$

end for

Bemerkung: Beachtlich ist hier, dass dieser Algorithmus erheblich besser als das bekannte Gram-Schmidt-Verfahren ist, da zur Neuberechnung des nächsten orthogonalen Basisvektors in $(*)$ nur gegen die beiden letzten Vektoren statt gegen *alle* bisher berechneten Vektoren orthogonalisiert werden muss. Dies liegt daran, dass wir bestimmte Vektoren, nämlich Potenzen $A^j v,\ j = 0, ..., m-1$ betrachten, und dass A symmetrisch ist.

4.3.7 Lemma
Für $i, j = 1, 2, ..., m^*$ gilt

1. $\left\langle v^{(i)}, v^{(j)} \right\rangle = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$

2. $\{v^{(1)}, ..., v^{(m)}\}$ ist eine (Orthogonal-) Basis von $K_m(A, r^{(0)})$ für $m = 1, ..., m^*$

3. Mit $V_m = [v^{(1)}|...|v^{(m)}]$ gilt

$$AV_m = V_{m+1}T_{m+1,m},$$

wobei

$$T_{m+1,m} = \begin{pmatrix} \alpha_1 & \beta_2 & & & 0 \\ \beta_2 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_m \\ & & & \ddots & \alpha_m \\ 0 & & & & \beta_{m+1} \end{pmatrix} \in \mathbb{R}^{(m+1)\times m}.$$

Beweis:

1. Wir zeigen die Behauptung per Induktion über k

$$\begin{cases} \left\langle v^{(k)}, v^{(k)} \right\rangle = 1 & \text{trivial} \\ \left\langle v^{(k)}, v^{(j)} \right\rangle = 0, & j = 1, .., k-1. \end{cases} \tag{4.3.7}$$

Für $k = 1$ ist (4.3.7) richtig. Gelte nun also die Gleichung (4.3.7) bis zu einem festem k, dann gilt

$$\begin{aligned} \left\langle v^{(k+1)}, v^{(k)} \right\rangle &= \frac{1}{\beta_{k+1}} \left\langle \widetilde{v}^{(k+1)}, v^{(k)} \right\rangle \\ &= \frac{1}{\beta_{k+1}} \left\langle Av^{(k)} - \alpha_k v^{(k)} - \beta_k v^{(k-1)}, v^{(k)} \right\rangle \\ &\overset{\text{(IA)}}{=} \frac{1}{\beta_{k+1}} \left\langle Av^{(k)} - \alpha v^{(k)}, v^{(k)} \right\rangle \\ &= \frac{1}{\beta_{k+1}} \left(\left\langle Av^{(k)}, v^{(k)} \right\rangle - \alpha_k \right) = 0. \end{aligned}$$

Weiterhin gilt

$$\begin{aligned} \left\langle v^{(k+1)}, v^{(k-1)} \right\rangle &= \frac{1}{\beta_{k+1}} \left\langle \widetilde{v}^{(k+1)}, v^{(k-1)} \right\rangle \\ &= \frac{1}{\beta_{k+1}} \left\langle Av^{(k)} - \alpha_k v^{(k)} - \beta_k v^{(k-1)}, v^{(k-1)} \right\rangle \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{\beta_{k+1}} \left\langle Av^{(k)} - \beta_k v^{(k-1)}, v^{(k-1)} \right\rangle \\
&= \frac{1}{\beta_{k+1}} \left(\left\langle v^{(k)}, Av^{(k-1)} \right\rangle - \beta_k \right) \quad \text{(da } A \text{ symm.)} \\
&= \frac{1}{\beta_{k+1}} \left(\left\langle v^{(k)}, \widetilde{v}^{(k)} + \alpha_{k-1} v^{(k-1)} - \beta_{k-1} v^{(k-2)} \right\rangle - \beta_k \right) \\
&= \frac{1}{\beta_{k+1}} \left(\left\langle v^{(k)}, \widetilde{v}^{(k)} \right\rangle - \beta_k \right) = 0.
\end{aligned}
$$

Schließlich gilt für $j < k-1$

$$
\begin{aligned}
\left\langle v^{(k+1)}, v^{(j)} \right\rangle &= \frac{1}{\beta_{k+1}} \left\langle \widetilde{v}^{(k+1)}, v^{(j)} \right\rangle \\
&= \frac{1}{\beta_{k+1}} \left\langle Av^{(k)} - \alpha_k v^{(k)} - \beta_k v^{(k-1)}, v^{(j)} \right\rangle \\
&= \frac{1}{\beta_{k+1}} \left\langle Av^{(k)}, v^{(j)} \right\rangle \\
&= \frac{1}{\beta_{k+1}} \left\langle v^{(k)}, \underbrace{Av^{(j)}}_{\in K_{j+1} \subseteq K_{k-1}} \right\rangle = 0.
\end{aligned}
$$

2. Die Orthogonalität folgt aus 1. und damit auch, dass $\{v_1, ..., v_m\}$ Basis ist.

3. Aus Zeile (*) von Algorithmus 4.3.6 folgt

$$
\begin{aligned}
\beta_{k+1} v^{(k+1)} &= Av^{(k)} - \alpha v^{(k)} - \beta_k v^{(k+1)} \\
\iff \quad Av^{(k)} &= \beta_{k+1} v^{(k+1)} + \alpha_k v^{(k)} + \beta_k^{(k-1)}
\end{aligned}
$$

Dies ist genau die k-te Spalte von 3.

□

Sei nun $v^{(1)}, ..., v^{(m)}$ die mit Algorihmus 4.3.6 Orthonormalbasis (*Lanczos-Basis*) für $K_m(A, r^{(0)})$, $V_m = [v_1|...|v_m]$. Dann gilt nach Lemma 4.3.7 Teil 3

$$
AV_m = V_{m+1} T_{m+1,m}
$$

und damit

$$
V_m^T A V_m = (V_m^T V_{m-n}) T_{m+1,m} = \left(\begin{array}{c|c} I_m & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} \end{array} \right) \bar{T}_{m+1,m} = T_m.
$$

Dabei ist

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix} \in \mathbb{R}^{m\times m}, \text{ tridiagonal.}$$

In (4.3.6) gilt außerdem dann

$$V_m^T r^{(0)} = \beta_1 e_1.$$

4.3.8 Algorithmus (CG-Verfahren)

Wähle $x^{(0)}$, setze $r^{(0)} = b - Ax^{(0)}, \beta_1 = \|r^{(0)}\|, v^{(1)} = r^{(0)}/\beta_1$
for $k = 1, 2, ...$ **do**
bestimme α_k aus Lanczos-Verfahren
(4.3.8) setze $x^{(k)} = x^{(0)} + V_k(T_k)^{-1}\beta_1 e_1$ [Lösung von (4.3.4)]
bestimme $\beta_{k+1},\ v^{(k+1)}$ aus Lanczos-Verfahren
end for

Die $x^{(k)}$ sind dann die Iterierten des CG-Verfahrens, sie erfüllen Gleichung (4.3.6) und damit (4.3.5).

Für die Umsetzung in einen praktikablen Algorithmus muss eine geeignetere „Aufdatierungsvorschrift“ für die $x^{(k)}$ gefunden werden, denn $x^{(k)}$ soll aus wenigen „früheren“ Iterierten berechenbar sein. Dazu:

$$T_m = V_m^T A V_m \quad \text{ist positiv definit (da } A \text{ spd).}$$

Also existiert die (wurzelfreie) Cholesky-Zerlegung von T_m, d.h.

$$T_m = L_m \cdot D_m \cdot L_m^T$$

mit

$$D_m = \operatorname{diag}(\delta_1, ..., \delta_m), \quad L_m = \begin{pmatrix} 1 & & & \\ \zeta_2 & \ddots & & \\ & \ddots & \ddots & \\ & & \zeta_m & 1 \end{pmatrix}.$$

Durch Gleichsetzen erhält man

$$L_m D_m L_m^T = \begin{pmatrix} 1 & & & \\ \zeta_2 & \ddots & & \\ & \ddots & \ddots & \\ & & \zeta_m & 1 \end{pmatrix} \cdot \begin{pmatrix} \delta_1 & \delta_1\zeta_2 & & 0 \\ & \ddots & \ddots & \\ & & \ddots & \delta_{m-1}\zeta_m \\ & & & \delta_m \end{pmatrix}$$

$$
= \begin{pmatrix}
\delta_1 & \delta_1\zeta_2 & & & \\
\delta_1\zeta_2 & \delta_2 + \delta_1\zeta_2^2 & \ddots & & \\
 & \ddots & \ddots & \ddots & \\
 & & \ddots & \ddots & \delta_{m-1}\zeta_m \\
 & & & \delta_{m-1}\zeta_m & \delta_m + \delta_{m-1}\zeta_m^2
\end{pmatrix}
$$

$$
= \begin{pmatrix}
\alpha_1 & \beta_2 & & & \\
\beta_2 & \ddots & \ddots & & \\
 & \ddots & \ddots & \ddots & \\
 & & \ddots & \ddots & \beta_m \\
 & & & \beta_m & \alpha_m
\end{pmatrix}
$$

also

$$
(4.3.9) \qquad \left\{ \begin{array}{l} \delta_1 = \alpha_1 \\ \left.\begin{array}{l} \zeta_k = \beta_k/\delta_{k-1} \\ \delta_k = \alpha_k - \delta_{k-1}\zeta_k^2 \end{array}\right\} \; k = 2, 3, ..., m. \end{array}\right.
$$

Damit erhalten wir aus (4.3.8)

$$
\begin{aligned}
x^{(k)} &= x^{(0)} V_k (L_k D_k L_k^T)^{-1} \beta_1 e_1 \\
&= x^{(0)} + \underbrace{\left(V_k (L_k^T)^{-1}\right)}_{=:W_k} \cdot \underbrace{\left(D_k^{-1} L_k^{-1} \beta_1 e_1\right)}_{=:z^{(k)}}.
\end{aligned}
$$

Beachte nun

$$
\begin{array}{ccc}
W_{k+1} L_{k+1}^T = V_{k+1} & = & [V_k | v_{k+1}] \\
\| & & \\
W_{k+1} \left(\begin{array}{ccc|c} & & & 0 \\ & L_k^T & & \vdots \\ & & & 0 \\ & & & \zeta_{k+1} \\ \hline 0 & \dots & 0 & 1 \end{array} \right). & &
\end{array}
$$

Also gilt

$$W_{k+1} = [W_k | w_{k+1}] \text{ mit } \zeta_{k+1} w_k + w_{k+1} = v_{k+1}.$$

Außerdem gilt

$$L_{k+1} D_{k+1} z^{(k+1)} \quad = \quad \beta_1 e_1.$$

Setze

$$(z^{(k+1)})_{k+1} \quad = \quad \mu_{k+1},$$

woraus sich

$$\left(\begin{array}{cccc|c} & & & & 0 \\ & & L_k & & \vdots \\ & & & & 0 \\ & & & & 0 \\ \hline 0 & \dots & 0 & \zeta_{k+1} & 1 \end{array} \right) \cdot \left(\begin{array}{c|c} D_k & 0 \\ \hline 0 & \delta_{k+1} \end{array} \right) z^{(k+1)} \quad = \quad \beta_1 e_1$$

ergibt. Es folgt

$$z^{(k+1)} = \begin{pmatrix} z^{(k)} \\ \mu_{k+1} \end{pmatrix}.$$

Dabei gilt

$$\zeta_{k+1} \delta_k \mu_k + \mu_{k+1} \delta_{k+1} = 0$$

und somit

$$\begin{aligned} \mu_{k+1} &= \frac{1}{\delta_{k+1}} \left(-\delta_k \zeta_{k+1} \mu_k \right) \\ &= -\frac{\beta_{k+1}}{\delta_{k+1}} \mu_k. \end{aligned}$$

Fassen wir nun alles zusammen, so ergibt sich damit

$$\begin{aligned} x^{(k+1)} &= x^{(0)} + W_{k+1} z^{(k+1)} \\ &= x^{(0)} + [W_k | w_{k+1}] \begin{pmatrix} z^{(k)} \\ \mu_{k+1} \end{pmatrix} \\ &= \underbrace{x^{(0)} + W_k z^{(k)}}_{=x^{(k)}} + \mu_{k+1} w_{k+1} \\ &= x^{(k)} + \mu_{k+1} w_{k+1}. \end{aligned}$$

Hiermit erhalten wir aus Algorithmus 4.3.8 mit den richtigen Initialisierungen (für $k = 1$) den folgenden Algorithmus

4.3.9 Algorithmus (CG-Verfahren, Variante I)

Wähle $x^{(0)}$, setze $r^{(0)} = b - Ax^{(0)}, \beta_1 = \|r^{(0)}\|, v_1 = r^{(0)}/\beta_1$
setze $w^{(0)} = 0, \delta_0 = 1, \mu_0 = 1, v^{(0)} = 0$
for $k = 1, 2, ..$ **do**

$$\begin{array}{rrcl}
 & q^{(k)} & = & Av^{(k)} \\
 & \alpha_k & = & \left\langle q^{(k)}, v^{(k)} \right\rangle \\
 & \zeta_k & = & \begin{cases} 0 & \text{falls} k = 1 \\ \beta_k/\delta_{k-1} & \text{falls } k > 1 \end{cases} \\
 & \delta_k & = & \alpha_k - \delta_{k-1}\zeta_k^2 \\
 & w^{(k)} & = & v^{(k)} - \zeta_k w^{(k-1)} \\
 & \mu_k & = & -\dfrac{\beta_k}{\delta_k}\mu_{k-1} \\
 & x^{(k)} & = & x^{(k-1)} + \mu_k w^{(k)} \\
(4.3.10) & \widetilde{v}^{(k+1)} & = & q^{(k)} - \alpha_k v^{(k)} - \beta_k v^{(k-1)} \\
 & \beta_{k+1} & = & \|\widetilde{v}^{(k+1)}\| \\
 & v^{(k+1)} & = & \widetilde{v}^{(k+1)}/\beta_{k+1}
\end{array}$$

end for

4.3.10 Bemerkung
Eine Implementierung dieses Verfahrens benötigt nur fünf Vektoren (einen für x, einen für w, zwei für die $v's$, sowie einen Hilfsvektor für w bzw. $\widetilde{v}$).

Wir bringen jetzt in zwei weiteren Schritten das CG-Verfahren auf eine Standard-Gestalt.

1. Schritt
Wir eliminieren zunächst $v^{(k-1)}$ in Gleichung (4.3.10) (ein Vektor weniger wird benötigt). Verwende

$$V_k = [v^{(1)}|...|v^{(k)}],\ W_k = [w^{(1)}|...|w^{(k)}].$$

Aus Lemma 4.3.7 folgt

$$AV_k = V_{k+1}\begin{pmatrix} & & T_k & \\ 0 & \dots & 0 & \beta_{k+1} \end{pmatrix} = \begin{pmatrix} & & L_k D_k L_k^T & \\ 0 & \dots & 0 & \beta_{k+1} \end{pmatrix}.$$

Mit $V_k = W_k L_k^T$ folgt nun

$$\begin{aligned}
AW_k L_k^T &= V_{k+1}\begin{pmatrix} & & L_k D_k L_k^T & \\ 0 & \dots & 0 & \beta_{k+1} \end{pmatrix} \\
\Rightarrow \quad AW_k &= V_{k+1}\begin{pmatrix} & & L_k D_k & \\ 0 & \dots & 0 & \beta_{k+1} \end{pmatrix}
\end{aligned}$$

denn

$$L_k \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \beta_{k+1} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \beta_{k+1} \end{pmatrix},$$

also

$$\begin{pmatrix} 0 \\ \vdots \\ 0 \\ \beta_{k+1} \end{pmatrix} = L_k^{-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \beta_{k+1} \end{pmatrix},$$

und damit

$$(0, ..., 0, \beta_{k+1}) = (0, ..., 0, \beta_{k+1})(L_k^T)^{-1}.$$

Dies bedeutet für die k-te Spalte

$$\begin{array}{rrcl} & Aw^{(k)} & = & \beta_{k+1} v^{(k+1)} + \delta_k v^{(k)} \\ \iff & v^{(k+1)} & = & \dfrac{1}{\beta_{k+1}} \cdot (Aw^{(k)} - \delta_k v^{(k)}) \\ \iff & \widetilde{v}^{(k+1)} & = & Aw^{(k)} - \delta_k v^{(k)}. \end{array}$$

Dies kann man statt Gleichung(4.3.10) verwenden. Allerdings benötigt man immer noch $q = Av^{(k)}$ zur Bestimmung von α_k. Auf α_k kann aber ganz verzichtet werden, denn δ_k (einzige Stelle, wo bisher α_k noch benötigt wird) kann alternativ auch folgendermaßen berechnet werden:

$$\begin{array}{rrcl} & L_k D_k L_k^T = T_k = V_k^T A V_k & = & V_k^T A W_k L_k^T \\ & & = & V_k^T (A W_k) L_k^T \; . \\ \Rightarrow & L_k D_k & = & V_k^T (A W_k) \end{array}$$

Vergleichen wir nun das Element an der Stelle (k, k) in beiden Matrizen, so ergibt sich

$$\delta_k = \left\langle v^{(k)}, Aw^{(k)} \right\rangle .$$

Als Ergebnis des ersten Schrittes erhalten wir nun

4.3.11 Algorithmus (CG-Verfahren, Variante II)

Initialisierung wie in Algorithmus 4.3.9
for $k = 1, 2, \ldots$ **do**

$$\begin{array}{lrcl}
 & \zeta_k & = & \beta_k/\delta_{k-1} \\
(4.3.11a) & w^{(k)} & = & \zeta_k w^{(k-1)} \\
 & q & = & Aw^{(k)} \\
 & \delta_k & = & \langle v^{(k)}, q \rangle \\
(4.3.11b) & \mu_k & = & -\dfrac{\beta_k}{\delta_k}\mu_{k-1} \\
(4.3.12) & x^{(k)} & = & x^{(k-1)} + \mu_k w^{(k)} \\
 & \widetilde{v}_{k+1} & = & q - \delta_k v^{(k)} \\
 & \beta_{k+1} & = & \|w_{k+1}\|_2 \\
 & v^{(k+1)} & = & \dfrac{1}{\beta_{k+1}}\widetilde{v}_{k+1}
\end{array}$$

end for

2. Schritt

Einbeziehung des Residuums $r^{(k)} = b - Ax^{(k)}$ (mit $\|r^{(k)}\|$ hat man ein vernünftiges Abbruchkriterium zur Hand):

$$\begin{aligned}
r^{(k)} = b - Ax^{(k)} \quad &= \quad b - A(x^{(0)} + V_k T_k^{-1}\begin{pmatrix} \beta_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}) \\
&= \quad (b - Ax^{(0)}) - AV_k T_k^{-1}\begin{pmatrix} \beta_k \\ 0 \\ \vdots \\ 0 \end{pmatrix} \\
&\overset{(4.3.7)}{=} \quad r^{(0)} - V_{k+1} T_{k+1,k} T_k^{-1}\begin{pmatrix} \beta_1 \\ 0 \vdots \\ 0 \end{pmatrix} \\
&= \quad r^{(0)} - V_{k+1}\begin{pmatrix} & & T_k & \\ 0 & \ldots & 0 & \beta_{k+1} \end{pmatrix} T_k^{-1}\begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}
\end{aligned}$$

$$= \quad r^{(0)} - V_{k+1} \begin{pmatrix} I_k \\ \begin{pmatrix} 0 & \dots & 0 & \beta_{k+1} \end{pmatrix} T_k^{-1} \end{pmatrix} \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$= \quad r^{(0)} - V_{k+1} \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \\ \begin{pmatrix} 0, ..., 0, \beta_{k+1} \end{pmatrix} T_k^{-1} \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \end{pmatrix}$$

$$= \quad r^{(0)} - \beta_1 v^{(1)} - \begin{pmatrix} 0, ..., 0, \beta_{k+1} \end{pmatrix} T_k^{-1} \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} v^{(k+1)}$$

$$= \quad \underbrace{-\begin{pmatrix} 0, ..., 0, \beta_{k+1} \end{pmatrix} T_k^{-1} \begin{pmatrix} \beta_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{*} v^{(k+1)}.$$

Also ist $r^{(k)}$ ein skalares Vielfaches von $v^{(k+1)}$! Wie erhält man den Faktor $*$ einfach?

Aus Gleichung (4.3.11) folgt:

$$r^{(k)} = b - Ax^{(k)} = b - Ax^{(k-1)} - \mu_k Aw^{(k)} = r^{(k-1)} - \mu_k Aw^{(k)}. \tag{4.3.12}$$

Für $v^{(k+1)}$ gilt

$$\begin{aligned} \beta_{k+1} v^{(k+1)} &= Aw^{(k)} - \delta_k v^{(k)} \\ \Rightarrow \quad -\mu_k \beta_{k+1} v^{(k+1)} &= \mu_k \delta_k v^{(k)} - \mu_k Aw^{(k)} \end{aligned}$$

Aus Algorithmus 4.3.9 wissen wir nun

$$\mu_k = -\frac{\beta_k}{\delta_k} \mu_{k-1},$$

also

$$\mu_k \delta_k = -\mu_{k-1} \beta_k$$

und damit

$$-\mu_k\beta_{k+1}v^{(k+1)} = -\mu_{k-1}\beta_k v^{(k)} - \mu_k A w^{(k)}.$$

Der Vergleich mit Gleichung 4.3.12 ergibt also:

$$r^{(k)} = (-\mu_k\beta_{k+1})v^{(k+1)},$$

und wegen $\|v^{(k+1)}\| = 1$, insbesondere

$$|\mu_k\beta_{k+1}| = \|r^{(k)}\|. \tag{4.3.13}$$

Wir stellen deshalb Algorithmus 4.3.11 auf die Größen

$$\begin{array}{rclc} r^{(k)} & = & (-\mu_k\beta_{k+1})v^{(k+1)} & (r^{(k)} \text{ statt } v^{(k+1)}) \\ p^{(k)} & = & (-\mu_{k-1}\beta_k)w^{(k)} & (p^{(k)} \text{ statt } w^{(k)}) \end{array}$$

um und erhalten so die Aufdatierungsvorschriften

$$\begin{aligned} &(4.3.14) & p^{(k)} &= r^{(k-1)} - \zeta_k \cdot \frac{-\mu_{k-1}\beta_k}{-\mu_{k-2}\beta_{k-1}} p^{(k-1)} && (\text{aus } (4.3.11\text{a})) \\ &(4.3.15) & x^{(k)} &= x^{(k-1)} + \mu_k \frac{-1}{\mu_{k-1}\beta_k} p^{(k)} && (\text{aus } (4.3.11)) \\ &(4.3.16) & r^{(k)} &= r^{(k-1)} - \mu_k \frac{-1}{\mu_{k-1}\beta_k} A p^{(k)} && (\text{aus } (4.3.15)). \end{aligned}$$

Für die vorkommenden Koeffizienten gilt

$$\begin{aligned} \zeta_k \cdot \frac{-\mu_{k-1}\beta_k}{-\mu_{k-2}\beta_{k-1}} &= \frac{\beta_k}{\delta_{k-1}} \frac{\mu_{k-1}\beta_k}{\mu_{k-2}\beta_{k-1}} \\ &\overset{(4.3.11b)}{=} \frac{\beta_k}{\frac{-\beta_{k-1}}{\mu_{k-1}\mu_{k-2}}\mu_{k-2}\beta_{k-1}} \frac{\mu_{k-1}\beta_k}{\mu_{k-2}\beta_{k-1}} \\ &= \frac{\beta_k^2\mu_{k-1}^2}{\beta_{k-1}^2\mu_{k-2}^2} \overset{(4.3.13)}{=} \frac{\left\langle r^{(k-1)}, r^{(k-1)}\right\rangle}{\left\langle r^{(k-2)}, r^{(k-2)}\right\rangle} =: \epsilon_{k-1} \end{aligned} \tag{4.3.17}$$

und

$$\begin{aligned} -\frac{\mu_k}{\mu_{k-1}\beta_k} &\overset{(4.3.11b)}{=} \frac{1}{\delta_k} = \frac{1}{\langle v^{(k)}, Aw^{(k)}\rangle} \\ &= \frac{(\mu_{k-1}\beta_k)^2}{\langle r^{(k-1)}, Ap^{(k)}\rangle} \\ &= \frac{\left\langle r^{(k-1)}, r^{(k-1)}\right\rangle}{\left\langle r^{(k-1)}, Ap^{(k)}\right\rangle} =: \gamma_{k-1}. \end{aligned} \tag{4.3.18}$$

Verwendet man noch

$$\gamma_{k-1} = \frac{\left\langle r^{(k-1)}, r^{(k-1)} \right\rangle}{\langle p^{(k-1)}, Ap^{(k)} \rangle} \tag{4.3.19}$$

(siehe Lemma 4.3.13, unten), so erhält man die Standardform des CG-Verfahrens.

4.3.12 Algorithmus (CG-Verfahren, Standardform)

wähle $x^{(0)}$
setze $r^{(0)} = b - Ax^{(0)},\ p^{(0)} = r^{(0)}$
for $k = 1, 2, ...$ **do**

$$\gamma_{k-1} = \frac{\left\langle r^{(k-1)}, r^{(k-1)} \right\rangle}{\langle p^{(k-1)}, Ap^{(k-1)} \rangle}$$
$$x^{(k)} = x^{(k-1)} + \gamma_{k-1} p^{(k-1)}$$
$$r^{(k)} = r^{(k-1)} - \gamma_{k-1} \cdot A \cdot p^{(k-1)}$$
$$\epsilon_k = \frac{\left\langle r^{(k)}, r^{(k)} \right\rangle}{\langle r^{(k-1)}, r^{(k-1)} \rangle}$$
$$p^{(k)} = r^{(k)} + \epsilon_k p^{(k-1)}$$

end for

4.3.13 Lemma
Es ist

$$\gamma_{k-1} = \frac{\left\langle r^{(k-1)}, r^{(k-1)} \right\rangle}{\langle r^{(k-1)}, Ap^{(k)} \rangle} = \frac{\left\langle r^{(k-1)}, r^{(k-1)} \right\rangle}{\langle p^{(k-1)}, Ap^{(k-1)} \rangle}.$$

Beweis: Aus Gleichung (4.3.14) folgt per Induktion

$$p^{(k)} \in K_k(A, r^{(0)}).$$

Die $r^{(k)}$ sind skalare Vielfache der Lanczos-Vektoren $v^{(k+1)}$ und $K_k(A, r^{(0)})$ wird von den $v^{(1)}, ..., v^{(k)}$ aufgespannt. Da $\left\langle v^{(k+1)}, v^{(j)} \right\rangle = 0,\ j = 1, ..., k$ gilt

$$\left\langle v^{(k+1)}, p^{(k)} \right\rangle = \left\langle r^{(k)}, p^{(k)} \right\rangle = 0$$

und damit folgt aus (4.3.16) und (4.3.18):

$$\begin{aligned}
0 = \left\langle r^{(k)}, p^{(k)} \right\rangle &= \left\langle r^{(k-1)} - \gamma_{k-1} Ap^{(k)}, p^{(k)} \right\rangle \\
\Rightarrow \qquad \gamma_{k-1} &= \frac{\left\langle r^{(k-1)}, p^{(k)} \right\rangle}{\langle p^{(k)}, Ap^{(k)} \rangle} \\
&\overset{(4.3.14)}{=} \frac{\left\langle r^{(k-1)}, r^{(k-1)} - \epsilon_{k-1} p^{(k-1)} \right\rangle}{\langle p^{(k)}, Ap^{(k)} \rangle} \\
&= \frac{\left\langle r^{(k-1)}, r^{(k-1)} \right\rangle}{\langle p^{(k)}, Ap^{(k)} \rangle}.
\end{aligned}$$

□

4.4 Analyse des CG-Verfahrens

Die Bezeichnungen seien wie im Kapitel 4.3; insbesondere sei die Matrix A symmetrisch und positiv definit (spd).

4.4.1 Satz
Es seien $x^{(k)}$ die Iterierten des CG-Verfahrens. Dann gilt

$$x^{(m^*)} = x^* = A^{-1}b,$$

d.h. das CG-Verfahren bricht nach $m^* \leq n$ Schritten mit der Lösung x^* ab.

Beweis: Nach Lemma 4.3.4 ist

$$x^* \in x^{(0)} + K_{m^*}(A, r^{(0)}).$$

Nach Konstruktion des CG-Verfahrens gilt nun:
$x^{(m^*)}$ minimiert $f(x) = \frac{1}{2}\langle x, Ax\rangle - \langle x, b\rangle$ auf $x^{(0)} + K_{m^*}(A, r^{(0)})$, also gilt nach Lemma 4.3.1 $x^{(m^*)} = x^*$. □

4.4.2 Korollar
Es sei m die Anzahl der *verschiedenen* Eigenwerte von A. Dann gilt

$$x^{(m)} = x^* = A^{-1}b.$$

Beweis: Es seien $\lambda_1, ..., \lambda_m$ die Eigenwerte von A; es bezeichne

$$p(t) = \prod_{j=1}^{m}(t - \lambda_j).$$

Da A symmetrisch ist, gilt $p(A) = 0$, also ist

$$p(A)r^{(0)} = 0 \quad \forall\ r^{(0)}$$

und damit $m^* \leq m$. □

4.4.3 Beispiel
Es sei

$$E = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

und

$$A = I + E.$$

In diesem Fall konvergiert das CG-Verfahren nach spätestens *zwei* Schritten.
Beweis: Die Eigenwerte von E ergeben sich zu

$$\lambda_1^E = ... = \lambda_{n-1}^E = 0,\ \lambda_n^E = n.$$

Damit sind die Eigenwerte von A

$$\lambda_1^A = ... = \lambda_{n-1}^A = 1,\ \lambda_n^A = n+1.$$

□

4.4.4 Bemerkung

1. Wegen numerischen Ungenauigkeiten ergibt sich praktisch *nie*

$$x^{(m^*)} = A^{-1}b,$$

aber $\|r^{(m^*)}\|$ ist klein.

2. Der Satz 4.4.1 und Korollar 4.4.2 sind für die Praxis als Konvergenzsresultat noch nicht ausreichend: für sehr große n ist auch m^* im allgemeinen groß.

Eine detaillierte Analyse des CG-Verfahrens verwendet die Tschebyscheff-Polynome.

4.4.5 Definition
Die *Tschebyscheff-Polynome* $T_m(t)$ sind induktiv definiert durch

$$\begin{aligned} T_0(t) &= 1 \\ T_1(t) &= t \\ T_{m+1}(t) &= 2tT_m(t) - T_{m-1}(t), \quad m = 1,2,... \end{aligned}$$

Damit ist insbesondere

$$deg(T_m) = m.$$

Bemerkung: Die Tschebyscheff-Polynome sind ein wichtiges Hilfsmittel in der Numerik bei Interpolation und Approximation, sie bilden Orthogonalpolynome im gewichteten L^2-Raum

$$L^2\left([-1,1], \frac{1}{1-x^2}\right).$$

Einige der wichtigsten Eigenschaften der Tschebyscheff-Polynome zeigt das folgende Lemma.

4.4.6 Lemma
Die Tschebyscheff-Polynome erfüllen

1. $T_m(t) = \cos(m \cdot \arccos(t)), \quad t \in [-1,1]$,
2. $T_m(t) = (\mathrm{sign}(t))^m \cdot \cosh(m \cdot \mathrm{Arcosh}(|t|)), \quad t \in \mathbb{R} \setminus [-1,1]$,
3. $T_m(t) = \frac{1}{2}\left((t+\sqrt{t^2-1})^m + (t+\sqrt{t^2-1})^{-m}\right), \quad t \in \mathbb{R}$.
4. Es gilt
   - $|T_m(t)| \leq 1$ für $t \in [-1,1]$,
   - $|T_m(t)| > 1$ für $t \in \mathbb{R} \setminus [-1,1]$,
   - $T_m(t) = 0 \quad \Longleftrightarrow \quad t = \tau_j = \cos\left(\frac{2j+1}{m}\frac{\pi}{2}\right) \in (-1,1)$, $j = 0, ..., m-1$,
   - $T_m(\sigma_j) = (-1)^j$ für $\sigma_j = \cos\left(\frac{j\pi}{m}\right), \quad j = 0, ..., m$.

Beweis:

1. Nach den Additionstheoremen gilt
$$\begin{aligned} \cos((m+1)x) &= \cos(x)\cos(mx) - \sin(x)\sin(mx) \\ \cos((m-1)x) &= \cos(x)\cos(mx) + \sin(x)\sin(mx). \end{aligned}$$
Durch Umstellen ergibt sich daraus
$$\cos((m+1)x) = 2 \cdot \cos(x)\cos(mx) - \cos((m-1)x).$$
Setze $x = \arccos(t)$ mit $t \in [-1,1]$, dann ergibt sich genau die Rekursionformel der T_m, dies liefert die Behauptung.

2. Folgt analog zu 1.

3. Wir betrachten nur den Fall $t > 1$, die anderen Fälle ($t \in [-1,1], t < -1$) gehen ähnlich. Wir verwenden $\sinh(t) = \frac{1}{2}(e^t - e^{-t})$ und $\cosh(t) = \frac{1}{2}(e^t + e^{-t})$ mit
$$\cosh^2(t) - 1 = \sinh^2(t),$$
und
$$\begin{aligned} \cosh(t) + \sinh(t) &= \frac{1}{2}(e^t + e^{-t}) + \frac{1}{2}(e^t - e^{-t}) \\ &= e^t. \end{aligned}$$

Setzen wir jetzt $t = \cosh(x)$ in 2. ein, so ergibt sich

$$\begin{aligned}
T_m(t) &= \cosh(m\mathrm{Arcosh}(t)) \\
&= \cosh(m\mathrm{Arcosh}(\cosh(x))) \\
&= \cosh(mx) \\
&= \frac{1}{2}\left[e^{mx} + e^{-mx}\right] \\
&= \frac{1}{2}\left[(e^x)^m + (e^x)^{-m}\ \right] \\
&= \frac{1}{2}\left[(\cosh(x) + \sinh(x)\ )^m + (\cosh(x) + \sinh(x)\ )^{-m}\right] \\
&= \frac{1}{2}\left[\left(\cosh(x) + \sqrt{\cosh^2(x) - 1}\ \right)^m\right. \\
&\qquad\qquad\qquad \left. + \left(\cosh(x) + \sqrt{\cosh^2(x) - 1}\ \right)^{-m}\right] \\
&= \frac{1}{2}\left[\left(t + \sqrt{t^2 - 1}\ \right)^m + \left(t + \sqrt{t^2 - 1}\ \right)^{-m}\right].
\end{aligned}$$

Also folgt die Behauptung.

4. Die ersten beiden Abschätzungen folgen einfach aus den Eigenschaften von cos bzw. cosh. Weiter gilt

$$\begin{aligned}
T_m\left(\cos\left(\frac{i\pi}{m}\right)\right) &= \cos\left(m \cdot \arccos\left(\cos\left(\frac{i\pi}{m}\right)\right)\right) \\
&= \cos\left(m \cdot \frac{i\pi}{m}\right) \\
&= \cos(i \cdot \pi) \\
&= (-1)^i,
\end{aligned}$$

sowie

$$\begin{aligned}
T_m\left(\cos\left(\frac{(2i+1)\pi}{2m}\right)\right) &= \cos\left(m \cdot \arccos\left(\cos\left(\frac{(2i+1)\pi}{2m}\right)\right)\right) \\
&= \cos\left(m \cdot \frac{(2i+1)\pi}{2m}\right) \\
&= \cos\left(\frac{(2i+1)\pi}{2}\right) \\
&= 0 \quad \forall\ i = 0, ..., m-1\ m \geq 1.
\end{aligned}$$

□

Wir haben nun die Tschebyscheff-Polynome auf dem Intervall $[-1, 1]$ eingeführt. Man kann sie auf ein allgemeineres Intervall $[a, b]$ übertragen.

4.4.7 Definition
Es sei $a < b$, dann ist

$$T_m^{[a,b]}(t) := T_m\left(\frac{2t - (b + a)}{b - a}\right)$$

das *Tschebyscheff-Polynom bezüglich* $[a, b]$.

Bemerkung:

1. Die Abbildung
$$t \longmapsto \frac{2t - (b + a)}{b - a}$$
ist eine lineare Transformation des Intervalls $[a, b]$ auf das Intervall $[-1, 1]$.

2. Nach Lemma 4.4.6 ergeben sich unter anderem folgende Eigenschaften

$$\begin{array}{llcll}
(4.4.1) & |T_m^{[a,b]}(t)| & \leq & 1 & t \in [a, b] \\
 & |T_m^{[a,b]}(t)| & \geq & 1 & t \in \mathbb{R} \setminus [a, b] \\
(4.4.2) & T_m^{[a,b]}(\sigma_j) & = & (-1)^j & \sigma_j = \dfrac{a + b}{2} + \dfrac{b - a}{2} \cos\left(\dfrac{j\pi}{m}\right), \\
 & & & & j = 0, ..., m.
\end{array}$$

4.4.8 Satz
Es sei $0 < a < b$ und

$$p_m(t) := \frac{1}{c_m} T_m^{[a,b]}(t) \text{ mit } c_m := T_m^{[a,b]}(0). \tag{4.4.3}$$

Dann gilt

$$\max_{t \in [a,b]} |p_m(t)| = \min_{p \in \bar{\Pi}_m} \max_{t \in [a,b]} |p(t)| = \left|\frac{1}{c_m}\right| =: d_m,$$

wobei

$$\overline{\Pi}_m := \{p \in \Pi_m |\ p(0) = 1\}.$$

D.h. p_m besitzt minimale L_∞-Norm auf $[a, b]$ unter allen Polynomen in $\bar{\Pi}_m$.

Beweis: Angenommen es gäbe ein $q \in \overline{\Pi}_m$ mit

$$\max_{t\in[a,b]} |q(t)| < d_m,$$

(dann folgt $p_m \neq q$). Setze

$$r(t) := p_m(t) - q(t).$$

Da nun

$$p_m(\sigma_j) = (-1)^j, \quad j = 0, ..., m$$

gilt, wechselt r auf $[a, b]$ genau $m + 1$-mal das Vorzeichen, besitzt also auf $[a, b]$ mindestens m Nullstellen. Da aber auch $p_m, q \in \Pi_m^0$ und $0 \notin [a, b]$ gilt, ist

$$r(0) = p_m(0) - q(0) = 1 - 1 = 0,$$

also besitzt r mindestens $m+1$ Nullstellen, womit nach dem Fundamentalsatz der Algebra $r \equiv 0$ gilt, Widerspruch! □

4.4.9 Satz
Für die Koeffizienten c_m aus Gleichung (4.4.3) gilt

$$c_m = (-1)^m \frac{1 + c^{2m}}{2c^m} \text{ mit } c = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}, \ \kappa = \frac{b}{a}. \tag{4.4.4}$$

Beweis: Nach Lemma 4.4.6 und Definition 4.4.7 gilt (da $b + a > b - a$ und damit $\frac{b+a}{b-a} > 1$ ist)

$$\begin{aligned} c_m &= T_m^{[a,b]}(0) = T_m\Big(\underbrace{-\frac{(b+a)}{b-a}}_{=:t_0}\Big) \\ &= \frac{1}{2}\left[\left(t_0 + \sqrt{t_0^2 - 1}\right)^m + \left(t_0 + \sqrt{t_0^2 - 1}\right)^{-m}\right] \\ &= \frac{1}{2}\left[\frac{\left(t_0 + \sqrt{t_0^2 - 1}\right)^{2m} + 1}{\left(t_0 + \sqrt{t_0^2 - 1}\right)^m}\right]. \end{aligned}$$

Zu zeigen bleibt noch

$$-c = t_0 + \sqrt{t_0^2 - 1}.$$

Es gilt aber

$$t_0 + \sqrt{t_0^2 - 1} = -\frac{(b+a)}{b-a} + \sqrt{\left(-\frac{(b+a)}{b-a}\right)^2 - 1}$$

$$
\begin{aligned}
&= -\frac{(b+a)}{b-a} + \sqrt{\frac{(b+a)^2-(b-a)^2}{(b-a)^2}} \\
&= -\frac{(b+a)}{b-a} + \sqrt{\frac{4ab}{(b-a)^2}} \\
&= -\frac{(b+a)-2\sqrt{ab}}{b-a} \\
&= -\frac{(\sqrt{b}-\sqrt{a})^2}{b-a} \\
&= -\frac{(\sqrt{b}-\sqrt{a})^2}{(\sqrt{b}-\sqrt{a})(\sqrt{b}+\sqrt{a})} \\
&= -\frac{(\sqrt{b}-\sqrt{a})}{(\sqrt{b}+\sqrt{a})} \\
&= -\frac{\sqrt{a}(\frac{\sqrt{b}}{\sqrt{a}}-1)}{\sqrt{a}(\frac{\sqrt{b}}{\sqrt{a}}+1)} \\
&= -\frac{(\frac{\sqrt{b}}{\sqrt{a}}-1)}{(\frac{\sqrt{b}}{\sqrt{a}}+1)} \\
&= -\frac{(\sqrt{\kappa}-1)}{(\sqrt{\kappa}+1)} \\
&= -c.
\end{aligned}
$$

□

Damit kommen wir zur Analyse des CG-Verfahrens.

4.4.10 Satz
Für die Iterierten des CG-Verfahrens gilt in der A-Norm die Abschätzung

$$\|x^{(m)} - x^*\|_A \leq \frac{2c^m}{1+c^{2m}} \|x^{(0)} - x^*\|_A \leq 2c^m \|x^{(0)} - x^*\|_A.$$

Beweis: Wir starten zunächst eine Vorüberlegung (beachte, dass A spd ist)

$$
\begin{aligned}
\|x^{(m)} - x^*\|_A^2 &\stackrel{\text{Def}}{=} \left\langle x^{(m)} - x^*, A(x^{(m)} - x^*)\right\rangle \\
&= \left\langle x^{(m)}, A(x^{(m)} - x^*)\right\rangle - \left\langle x^*, A(x^{(m)} - x^*)\right\rangle \\
&= \left\langle x^{(m)}, Ax^{(m)}\right\rangle - \left\langle x^{(m)}, Ax^*\right\rangle - \left\langle x^*, Ax^{(m)}\right\rangle + \langle x^*, Ax^*\rangle \\
&= \left\langle x^{(m)}, Ax^{(m)}\right\rangle - 2\left\langle x^{(m)}, Ax^*\right\rangle + \langle x^*, Ax^*\rangle \\
&= \left\langle x^{(m)}, Ax^{(m)}\right\rangle - 2\left\langle x^{(m)}, b\right\rangle + \langle x^*, b\rangle
\end{aligned}
$$

$$= \quad 2f(x^{(m)}) + \langle x^*, b \rangle \,.$$

$x^{(m)}$ minimiert $f(x)$ auf $x^{(0)} + K_m(A, r^{(0)})$, also minimiert $x^{(m)}$ auch $\|x^{(m)} - A^*\|_A$.

Wir wählen nun ein beliebiges $y \in x^{(0)} + K_m(a, r^{(0)})$. Dann existiert ein Polynom $p \in \Pi_{m-1}$ mit

$$y = x^{(0)} + p(A)r^{(0)}.$$

Also folgt

$$\begin{aligned} y - x^* &= x^{(0)} + p(A)r^{(0)} - x^* \\ &= x^{(0)} + p(A)(b - Ax^{(0)}) - x^* \\ &= x^{(0)} + p(A)(Ax^* - Ax^{(0)}) - x^* \\ &= x^{(0)} - p(A)(Ax^{(0)} - Ax^*) - x^* \\ &= \underbrace{(I - p(A) \cdot A)}_{=:q(A)}(x^{(0)} - x^*). \end{aligned}$$

Es ist $q \in \bar{\Pi}_m$.

Andererseits gibt es für jedes $q_0 \in \bar{\Pi}_m$ ein $p_0 \in \Pi_{m-1}$ mit

$$q_0(t) = 1 - p_0(t) \cdot t.$$

Wähle nun y so, dass $q(t) = p_m(t)$ mit dem skalierten Tschebyscheff-Polynom p_m. Weil $x^{(m)}$ den Wert $\|x^{(m)} - x^*\|_A$ auf dem Krylov-Unterraum $K_m(A, r^{(0)})$ minimiert, gilt

$$\begin{aligned} \|x^{(m)} - x^*\|_A &\leq \|y - x^*\| \\ &= \|p_m(A)(x^{(0)} - x^*)\|_A \\ &\leq \|p_m(A)\|_A \cdot \|(x^{(0)} - x^*)\|_A. \end{aligned}$$

Es seien $0 < \lambda_1 \leq \lambda_2 ... \leq \lambda_n$ die Eigenwerte von A. Dann ergibt sich

$$\|p_m(A)\|_A = \|p_m(A)\|_2 = \max_{j=1,\dots,n} |p_m(\lambda_j)| \leq \max_{t \in [\lambda_1, \lambda_n]} |p_m(t)| = d_m = \frac{2c^m}{1 + c^{2m}}.$$

Also folgt

$$\|x^{(m)} - x^*\|_A \leq \frac{2c^m}{1 + c^{2m}} \cdot \|(x^{(0)} - x^*)\|_A \leq 2c^m \cdot \|(x^{(0)} - x^*)\|_A.$$

Beachte: Wegen $c < 1$ gilt $\lim\limits_{m \to \infty} c^m = 0$. □

Satz 4.4.10 kann zum Vergleich mit „herkömmlichen“ Iterationsverfahren herangezogen werden, nämlich:

cg: $\|x^{(k)} - x^*\|_A \leq 2c^k \cdot \|x^{(0)} - x^*\|_A$
SOR: $\|x^{(k)} - x^*\| \lesssim (\rho(H))^k \cdot \|x^{(0)} - x^*\|, \quad \|\cdot\|$ geeignete Norm.

Bei der Beurteilung der Konvergenzgeschwindigkeit entspricht der Faktor $(\rho(H))^k$ dem $2 \cdot c^k$, oder nach dem Ziehen der k-ten Wurzel

$$\rho(H) \text{ entspricht } \underbrace{\sqrt[k]{2}}_{\approx 1} \cdot c$$

und damit

$$\rho(H) \text{ entspricht } c.$$

Modellproblem I

Kehren wir zurück zu Modellproblem I, so ergab sich beim optimalen ω

$$\rho(H) = 1 - 2\pi h + \mathcal{O}(h^2).$$

Für das CG-Verfahren ist

$$\begin{aligned} \lambda_{\min} &= 4 - 4\cos(\pi h) \\ \lambda_{\max} &= 4 + 4\cos(\pi h) = 4 - 4\cos(\pi h N). \end{aligned}$$

Also gilt

$$\begin{aligned} \kappa &= \frac{\lambda_{\max}}{\lambda_{\min}} \\ &= \frac{4 + 4\cos(\pi h)}{4 - 4\cos(\pi h)} \\ &= \frac{4 + 4 - 2\pi^2 h^2 + \mathcal{O}(h^4)}{4 - 4 + 2\pi^2 h^2 + \mathcal{O}(h^4)}. \end{aligned}$$

Für den Kehrwert erhalten wir somit

$$\begin{aligned} \tfrac{1}{\kappa} &= \tfrac{1}{4}\pi^2 h^2 + \mathcal{O}(h^4) \\ \Rightarrow \quad \tfrac{1}{\sqrt{\kappa}} &= \tfrac{1}{2}\pi h + \mathcal{O}(h^2). \end{aligned}$$

Desweiteren (wobei wir $\frac{1}{1-x} = 1 + x + \mathcal{O}(x^2)$ benutzen) folgt

$$c = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

$$
\begin{aligned}
&= \frac{1-\frac{1}{\sqrt{\kappa}}}{1+\frac{1}{\sqrt{\kappa}}} \\
&= 1-\frac{2}{\sqrt{\kappa}}+\mathcal{O}\left(\frac{1}{\kappa}\right)
\end{aligned}
$$

und damit folgt

$$
\begin{aligned}
c &= 1-2\left(\frac{1}{2}\pi h+\mathcal{O}(h^2)\right)+\mathcal{O}(h^2)) \\
&= 1-\pi h+\mathcal{O}(h^2).
\end{aligned}
$$

Also besitzt das CG-Verfahren *die gleiche* Ordnung (nämlich h^1) wie das SOR-Verfahren mit ω_{opt}. Hier muss aber kein ω_{opt} bestimmt werden!

4.4.11 Definition

Es sei A spd, λ_1 der kleinste sowie λ_n der größte Eigenwert von A. Dann heißt

$$\operatorname{cond}(A)=\kappa(A)=\frac{\lambda_n}{\lambda_1}$$

die *Kondition* von A.

Die Kondition κ bestimmt (über $c=\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$) die Konvergenzgeschwindigkeit des CG-Verfahrens gemäß Satz 4.4.10.
Als Verbesserungsmöglichkeit für das CG-Verfahren bietet sich deshalb an: Verkleinere die Kondition

$$\kappa=\frac{b}{a}$$

durch ein Transformation

$$
\begin{aligned}
A &\to SAS^T \\
Ax=b &\to SAS^T(S^{-T}x)=Sb.
\end{aligned}
$$

Dies liefert das *präkonditionierte CG-Verfahren.*

4.5 Präkonditioniertes CG-Verfahren

Idee: Löse statt $Ax = b$ (implizit) das äquivalente System

$$
\begin{aligned}
\widehat{A}\widehat{x} &= \widehat{b} \qquad \text{mit} \\
\widehat{A} &= SAS^T\,, S \text{ regulär } (\Rightarrow \widehat{A} \text{ spd}) \\
\widehat{x} &= S^{-T}x \\
\widehat{b} &= Sb\,,
\end{aligned}
\tag{4.5.1}
$$

wobei $\mathrm{cond}(\widehat{A}) \ll \mathrm{cond}(A)$ sein soll.

Achtung: Wenn A dünn besetzt war, so kann SAS^T nach Ausmultiplizieren dicht besetzt sein! Darum werden wir die Matrix $\hat{S} = SAS^T$ nicht *explizit* berechnen.

Jede spd-Matrix M^{-1} besitzt eine Zerlegung $M^{-1} = S^TS$, z.B. die Cholesky-Zerlegung oder $S^T = M^{-\frac{1}{2}} = S$.

4.5.1 Definition
Seien $A \in \mathbb{R}^{n\times n}$, $M \in \mathbb{R}^{n\times n}$, A, M spd. Dann heißt M **Präkonditionierer** für A, falls $\mathrm{cond}(SAS^T) < \mathrm{cond}(A)$ für ein S mit $M^{-1} = S^TS$ gilt.

Ziel: Das CG-Verfahren für (4.5.1) soll so formulieret werden, dass

- die Iterierten bezüglich des ursprünglichen Systems $Ax = b$ gegeben sind,
- $\widehat{A} = SAS^T$ nicht explizit berechnet wird,
- statt S, S^T nur $M^{-1} = S^TS$ vorkommt (möglich?).

Das CG-Verfahren für (4.5.1) lautet (mit Index k um 1 verschoben im Vergleich zu Abschnitt 4.4).

Gegeben: $\widehat{x}_0,\ \widehat{r}_0 = \widehat{b} - \widehat{A}\widehat{x}_0,\ \widehat{p}_0 = \widehat{r}_0,\ \widehat{q}_0 = \widehat{A}\widehat{p}_0$

für $k = 0, 1, \ldots, m_0 - 1$

$$\widehat{x}_{k+1} = \widehat{x}_k + \widehat{\alpha}_k \widehat{p}_k \text{ mit } \widehat{\alpha}_k = \frac{\|\widehat{r}_k\|_2^2}{\langle \widehat{q}_k, \widehat{p}_k \rangle} \tag{4.5.2}$$

$$\widehat{r}_{k+1} = \widehat{r}_k - \widehat{\alpha}_k \widehat{q}_k \tag{4.5.3}$$

$$\widehat{p}_{k+1} = \widehat{r}_{k+1} + \widehat{\beta}_k \widehat{p}_k \text{ mit } \widehat{\beta}_k = \frac{\|\widehat{r}_{k+1}\|_2^2}{\|\widehat{r}_k\|_2^2} \tag{4.5.4}$$

$$\widehat{q}_{k+1} = \widehat{A}\widehat{p}_{k+1} \tag{4.5.5}$$

Rücktransformation in das ursprüngliche System:

$$\begin{aligned} \widehat{x}_k &= S^{-T} x_k \iff x_k = S^T \widehat{x}_k \\ \Rightarrow \underbrace{b - Ax_k}_{=:r_k} &= S^{-1} \underbrace{(Sb - SAS^T S^{-T} x_k)}_{\widehat{b} - \widehat{A}\widehat{x}_k} = S^{-1} \widehat{r}_k \end{aligned}$$

Multipliziere also (4.5.2), (4.5.4) mit S^T, (4.5.3), (4.5.5) mit S^{-1} und setze

$$\left\{ \begin{aligned} x_k &= S^T \widehat{x}_k, \\ r_k &= S^{-1} \widehat{r}_k \, (= b - Ax_k), \\ p_k &= S^T \widehat{p}_k, \\ q_k &= S^{-1} \widehat{q}_k. \end{aligned} \right. \tag{4.5.6}$$

Mit Hilfe von (4.5.6) formulieren wir die die Gleichungen (4.5.2) bis (4.5.5) um:

$$\begin{aligned} \widehat{\alpha}_k &= \frac{< \widehat{r}_k, \widehat{r}_k >}{< \widehat{q}_k, \widehat{p}_k >} \\ &= \frac{< Sr_k, Sr_k >}{< Sq_k, S^{-T} p_k >} \\ &= \frac{< r_k, S^T S r_k >}{< q_k, S^T S^{-T} p_k >} \\ &= \frac{< r_k, M^{-1} r_k >}{< q_k, p_k >} \\ &= \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} \\ \widehat{\beta}_k &= \frac{\|\widehat{r}_{k+1}\|_2^2}{\|\widehat{r}_k\|_2^2} \end{aligned}$$

$$
\begin{aligned}
&= \frac{< Sr_{k+1}, Sr_{k+1} >}{< Sr_k, Sr_k >} \\
&= \frac{< r_{k+1}, S^T Sr_{k+1} >}{< r_k, S^T Sr_k >} \\
&= \frac{< r_{k+1}, M^{-1} r_{k+1} >}{< r_k, M^{-1} r_k >} \\
&= \frac{< r_{k+1}, \widetilde{r}_{k+1} >}{< r_k, \widetilde{r}_k >}.
\end{aligned}
$$

Damit ergibt sich dann

$$
\begin{aligned}
\widehat{x}_{k+1} &= \widehat{x}_k + \widehat{\alpha}_k \widehat{p}_k \\
S^{-T} x_{k+1} &= S^{-T} x_k + \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} S^{-T} p_k \\
x_{k+1} &= x_k + \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} p_k \qquad \text{(4.5.2')}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{r}_{k+1} &= \widehat{r}_k - \widehat{\alpha}_k \widehat{q}_k \\
Sr_{k+1} &= Sr_k - \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} Sq_k \\
r_{k+1} &= r_k - \frac{< r_k, \widetilde{r}_k >}{< Sq_k, S^{-T} p >} q_k \qquad \text{(4.5.3')}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{p}_{k+1} &= \widehat{r}_{k+1} + \widehat{\beta}_k \widehat{p}_k \\
S^{-T} p_{k+1} &= Sr_{k+1} + \frac{< r_{k+1}, \widetilde{r}_{k+1} >}{< r_k, \widetilde{r}_k >} S^{-T} p_k \\
p_{k+1} &= S^T S r_{k+1} + \frac{< r_{k+1}, \widetilde{r}_{k+1} >}{< r_k, \widetilde{r}_k >} p_k \\
p_{k+1} &= \widetilde{r}_{k+1} + \frac{< r_{k+1}, \widetilde{r}_{k+1} >}{< r_k, \widetilde{r}_k >} p_k \qquad \text{(4.5.4')}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{q}_{k+1} &= \widehat{A} \widehat{p}_{k+1} \\
Sq_{k+1} &= (SAS^T)(S^{-T} p_{k+1}) \\
q_{k+1} &= Ap_{k+1}. \qquad \text{(4.5.5')}
\end{aligned}
$$

Dies liefert folgenden Algorithmus

4.5.2 Algorithmus (Präkonditioniertes CG-Verfahren)

wähle $x^{(0)}$ setze $r^{(0)} = b - Ax^{(0)}, p^{(0)} = r^{(0)}$, löse $M\hat{r}^{(0)} = r^{(0)}$

for $k = 0, 1, ..., m_0 - 1$ **do**

$$x_{k+1} = x_k + \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} p_k$$

$$r_{k+1} = r_k - \frac{< r_k, \widetilde{r}_k >}{< q_k, p_k >} q_k$$

löse $M\widetilde{r}_{k+1} = r_{k+1}$

$$p_{k+1} = \widetilde{r}_{k+1} + \frac{< r_{k+1}, \widetilde{r}_{k+1} >}{< r_k, \widetilde{r}_k >} p_k$$

$$q_{k+1} = Ap_{k+1}$$

end for

Bemerkung: Die Kunst ist nun, den Präkonditionierer so zu finden, dass die Gleichung $M\hat{r}^{(k+1)} = r^{(k+1)}$ leicht zu lösen ist und gleichzeitig $\text{cond}(SAS^T)$ klein wird.

4.6 Präkonditionierer

Allgemeine Situation:

Gegeben:	$A \in \mathbb{R}^{n\times n}$ spd
Gesucht:	$M \in \mathbb{R}^{n\times n}$ spd, so dass $M\tilde{r} = r$ einfach lösbar *und* $\operatorname{cond}(M^{-1}A) < \operatorname{cond}(A)$.

4.6.1 Bemerkung
Aus Lemma 4.6.6 (s.u.) folgt, dass $M^{-1}A$ dieselben (reellen und positiven) Eigenwerte hat wie SAS^T für jedes S mit $M^{-1} = S^TS$. In diesem Sinne ist für die nicht-symmetrische Matrix $M^{-1}A$ die Größe $\operatorname{cond}(M^{-1}A)$ immer noch erklärt als λ_n/λ_1.

4.6.2 Beispiel (Diagonale Präkonditionierung)
$M = \operatorname{diag}(A)$ ist spd und $M\tilde{r} = r$ ist sehr einfach zu lösen.

4.6.3 Beispiel
Bei Modellproblem I lässt die diagonale Präkonditionierung die Kondition unverändert. Denn:
Ist A die Matrix, des Modellproblems I, so gilt

$$M = \operatorname{diag}(A) = \begin{pmatrix} 4 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 4 \end{pmatrix}.$$

Also gilt für die Kondition von $M^{-1}A$

$$\operatorname{cond}(M^{-1}A) = \frac{\frac{1}{4}\lambda_{\max}}{\frac{1}{4}\lambda_{\min}} = \frac{\lambda_{\max}}{\lambda_{\min}} = \operatorname{cond}(A).$$

4.6.4 Beispiel (Präkond. mit abgebrochener Reihenentwicklung)
Sei

$$A = P - Q \ \text{ mit } \rho(P^{-1}Q) < 1.$$

Dann gilt

$$A^{-1} = (P-Q)^{-1} = (I - P^{-1}Q)^{-1}P^{-1} = \sum_{\nu=0}^{\infty}(P^{-1}Q)^{\nu}P^{-1}.$$

Nehme für ein festes m

$$M^{-1} = M_m^{-1} = \sum_{\nu=0}^{m-1} (P^{-1}Q)^\nu P^{-1}. \tag{4.6.1}$$

Je größer m, desto besser approximiert M_m^{-1} die Matrix A^{-1}.

Es stellen sich die folgenden Fragen:

- Wann ist M_m spd? (siehe Lemma 4.6.7).
- Wie löst man

  $$M\tilde{r} = r \Longleftrightarrow \tilde{r} = \sum_{\nu=0}^{m-1} (P^{-1}Q)^\nu P^{-1} r \tag{4.6.2}$$

  (siehe nächstes Lemma)

4.6.5 Lemma
Sei $r_0 = 0$ und für $\nu = 0, 1, \ldots, m-1$ sei

$$P r_{\nu+1} = Q r_\nu + r. \tag{4.6.3}$$

Dann gilt $r_m = \tilde{r}$ mit $\tilde{r}$ aus (4.6.2).

Beweis: Aus (4.6.3) folgt

$$\begin{aligned}
r_\nu &= P^{-1}Qr_{\nu-1} + P^{-1}r \\
&= P^{-1}Q(P^{-1}Qr_{\nu-2} + P^{-1}r) + P^{-1}r \\
&= P^{-1}Q(P^{-1}Q(P^{-1}Qr_{\nu-3} + P^{-1}r) + P^{-1}r) + P^{-1}r \\
&= \ldots \\
&= \sum_{\nu=0}^{m} (P^{-1}Q)^\nu \underbrace{r_0}_{=0} + \sum_{\nu=0}^{m-1} (P^{-1}Q)^\nu P^{-1} r \\
&= \sum_{\nu=0}^{m-1} (P^{-1}Q)^\nu P^{-1} r \\
&= \tilde{r}.
\end{aligned}$$

□

Lösen von $M_m\tilde{r} = r$ bedeutet also, m Iterationsschritte der einfachen Iteration (4.6.3) auszuführen.

4.6.6 Lemma
$A \in \mathbb{R}^{n\times n}$ spd, $B \in \mathbb{R}^{n\times n}$ symmetrisch. Dann gilt

a) Alle Eigenwerte von AB und BA sind reell.

b) Ist B spd (positiv semidefinit), so sind alle Eigenwerte von AB und BA positiv (nichtnegativ).

c) Sind alle Eigenwerte von AB oder BA positiv, so ist B spd.

Beweis: Mit spek(A) bezeichnen wir die Menge aller Eigenwerte von A (das *Spektrum*). Für einen Eigenvektor z zum Eigenwert λ von AB ist $A^{-1/2}z$ Eigenvektor von $A^{1/2}BA^{1/2}$ zum Eigenwert λ. Also ist $\text{spek}(AB) = \text{spek}(A^{1/2}BA^{1/2})$, wobei die letzte Matrix symmetrisch ist.

Es gilt die folgende zentrale Umformung

$$\left\{ \begin{array}{rcl} (BA)z = \lambda z & \iff & (BA^{1/2})(A^{1/2}z) = \lambda z \\ & \iff & \underbrace{(A^{1/2}BA^{1/2})}_{\text{sym.}}(A^{1/2}z) = \lambda(A^{1/2}z) \\ & \iff & A^{1/2}B(A^{1/2})^T w = \lambda w. \end{array} \right. \tag{4.6.4}$$

a) Nach (4.6.4) ist jeder Eigenwert λ von BA auch Eigenwert einer symmetrischen Matrix und somit reell. Die Behauptung für AB zeigt man analog.

b) Mit B ist auch $A^{1/2}B(A^{1/2})^T$ spd (positiv semidefinit) Matrix. Nach (4.6.4) ist damit jeder Eigenwert λ von BA positiv (nicht negativ). Die Behauptung für AB folgt analog.

c) Weil $\text{spek}(BA) \in \mathbb{R}^+$ ist, folgt $\text{spek}(A^{1/2}BA^{1/2}) \in \mathbb{R}^+$, also ist $A^{1/2}BA^{1/2}$ spd. Damit ist aber auch

$$A^{-1/2}(A^{1/2}BA^{1/2})A^{-1/2} = B$$

spd und damit folgt die Behauptung für BA. Für AB geht der Beweis analog.

□

4.6.7 Lemma
$A \in \mathbb{R}^{n\times n}$ sei spd, $A = P - Q$ mit $P = P^T$. Dann gilt für M_m aus 4.6.1

a) Ist m ungerade und P spd, so ist M_m spd.

b) Ist m gerade und $P + Q$ spd, so ist M_m spd.

Beweis: Wir zeigen jeweils, dass M_m^{-1} spd ist. Wir bezeichnen $H = P^{-1}Q$. Wegen $P = P^T, Q = Q^T$ folgt $H^T = QP^{-1}$. Durch geeignetes Zusammenfassen erhält man so

$$H^\nu P^{-1} = \begin{cases} (H^\mu P^{-1})^T Q (H^\mu P^{-1}) & \text{falls } \nu = 2\mu + 1 \\ H^\mu P^{-1} (H^\mu)^T & \text{falls } \nu = 2\mu. \end{cases}$$

In jedem Fall ist also $H^\nu P^{-1}$ symmetrisch und damit auch

$$M_m^{-1} = \sum_{\nu=0}^{m-1} H^\nu P^{-1}.$$

Wir müssen noch zeigen, dass die Eigenwerte von M_m^{-1} alle positiv sind. Nach Lemma 4.6.6 sind alle Eigenwerte von

$$H = I - P^{-1}A = (= P^{-1}Q)$$

reell da P^{-1} symmetrisch und A spd ist.

zu a): Alle Eigenwerte von $T_m := \sum_{\nu=0}^{m-1} H^\nu$ sind gegeben durch

$$\sum_{\nu=0}^{m-1} \lambda^\nu = \begin{cases} \frac{1-\lambda^m}{1-\lambda} & \text{falls } \lambda \neq 1, \\ m & \text{falls } \lambda = 1. \end{cases} , \tag{4.6.5}$$

wobei λ Eigenwert von H ist. Für m ungerade ist $(1 - \lambda^m)/(1 - \lambda)$ positiv für alle $\lambda \neq 1$. T_m hat also nur positive Eigenwerte. Ist außerdem P spd, so folgt aus $T_m = M_m^{-1}P$ nach Lemma 4.6.6 c), dass M_m^{-1} spd ist.

zu b): Nun sei m gerade. Es gilt $P + PH = P + Q$ und so

$$\begin{aligned} M_m^{-1} &= \sum_{\nu=0}^{m-1} H^\nu P^{-1} \\ &= H^0 P^{-1} + H^1 P^{-1} + ... + H^{m-1} P^{-1} \\ &= P^{-1}(P + PH + PH^2 + \ldots + PH^{m-1}) P^{-1} \\ &= P^{-1}((P + PH) + (P + PH)H^2 + \ldots + (P + PH)H^{m-2}) P^{-1} \\ &= P^{-1}(P + Q)(I + H^2 + \ldots + H^{m-2}) P^{-1}. \end{aligned}$$

Hieraus erhalten wir

$$PM_m^{-1}P = (P + Q)S_m \quad \text{mit } S_m = I + H^2 + \ldots + H^{m-2}.$$

Die Matrix S_m besitzt die Eigenwerte $\sum_{\mu=0}^{m/2-1} \lambda^{2\mu}$, λ Eigenwert von H. Also sind alle Eigenwerte von S_m positiv. Wie bei a) folgt mit Lemma 4.6.6 c) mit $S_m = (P + Q)^{-1}(PM_m^{-1}P)$ zunächst, dass $PM_m^{-1}P$ spd ist, wenn $P + Q$ spd ist, was nach Voraussetzung der Fall ist . Mit $PM_m^{-1}P$ ist auch M_m^{-1} spd. □

4.6.8 Lemma
$A \in \mathbb{R}^{n\times n}$ sei spd, $A = P - Q$ mit $P = P^T$ und $H = P^{-1}Q$, $\rho(H) < 1$, $M_m^{-1} = \sum\limits_{\nu=0}^{m-1} H^\nu P^{-1}$. Dann gilt

a) H besitzt nur reelle Eigenwerte $\lambda_1 \leq \lambda_2 \ldots \leq \lambda_n$

b) $M_m^{-1}A$ besitzt nur positive Eigenwerte und

$$\operatorname{cond}(M_m^{-1}A) = \frac{\lambda_{\max}(M_m^{-1}A)}{\lambda_{\min}(M_m^{-1}A)} = \begin{cases} \frac{1-\lambda_1^m}{1-\lambda_n^m} & \text{falls } \lambda_1 \geq 0 \text{ oder } m \\ & \text{ungerade} \\ \frac{1-\delta^m}{1-\lambda_n^m} & \text{falls } m \text{ gerade und } \lambda_1 < 0, \\ & \mid \lambda_n \mid \geq \mid \lambda_1 \mid \\ \frac{1-\delta^m}{1-\lambda_1^m} & \text{falls } m \text{ gerade und } \lambda_1 < 0, \\ & \mid \lambda_n \mid < \mid \lambda_1 \mid \end{cases}$$

wobei $\delta = \min\limits_{i=1}^{n} \mid \lambda_i \mid$

Beweis:

a) Es ist $H = I - P^{-1}A$ und $P^{-1}A$ hat nach Lemma 4.6.5 nur reelle Eigenwerte. Wegen $\rho(H) < 1$ gilt außerdem:

$$-1 < \lambda_1 \leq \lambda_n < 1.$$

b) Es gilt

$$\begin{aligned} M_m^{-1}A &= \left(\sum_{\nu=0}^{m-1} H^\nu P^{-1}\right) \underbrace{A}_{=P-Q} \\ &= \sum_{\nu=0}^{m-1} H^\nu (I - H) = I - H^m \\ \Rightarrow \quad \operatorname{spek}(M_m^{-1}A) &= \{1 - \lambda_i^m,\ i = 1, \ldots, n\}. \end{aligned}$$

$$\begin{aligned} \text{Setze } \lambda &= \min_i \mid 1 - \lambda_i^m \mid = \min_i (1 - \lambda_i^m) \quad \text{da } |\lambda_i| < 1 \\ \Lambda &= \max_i \mid 1 - \lambda_i^m \mid = \max_i (1 - \lambda_i^m) \end{aligned}$$

Falls m ungerade ist, gilt

$$\lambda_i^m \leq \ldots \leq \lambda_n^m,$$

also

$$\lambda = 1 - \lambda_n^m,\ \Lambda = 1 - \lambda_1^m.$$

Falls $\lambda_1 \geq 0$ ist, gilt $\lambda_i \geq 0$ für alle i , also

$$\lambda = 1 - \lambda_n^m\ \Lambda = 1 - \lambda_1^m,\ .$$

Dies liefert die 1. Zeile von b).

Falls m gerade ist und $\lambda_1 < 0$, folgt

$$\lambda = \min\{1 - \lambda_n^m, 1 - \lambda_1^m\},\ \Lambda = 1 - \delta^m, .$$

Dies ergibt die 2. und 3. Zeile in b).

□

4.6.9 Satz (m-Schritt-SSOR-Präkonditionierung:)
Die Matrix $A \in \mathbb{R}^{n\times n}$ sei spd und $A = P - Q$ sei die SSOR-Zerlegung

$$\begin{aligned} P &= \frac{1}{\omega(2-\omega)}(D - \omega L)D^{-1}(D - \omega L^T), \\ Q &= \frac{1}{\omega(2-\omega)}((1-\omega)D + \omega L)D^{-1}((1-\omega)D + \omega L^T) \end{aligned}$$

mit $\omega \in (0, 2), A = D - L - L^T$ wie üblich. Wieder sei

$$H = P^{-1}Q,\ M_m^{-1} = \sum_{r=0}^{m-1} H^r P^{-1}.$$

Dann gilt

a) M_m ist spd für alle m

b) H besitzt nichtnegative reelle Eigenwerte $0 \leq \lambda_1 \leq \ldots \leq \lambda_n$ und es ist

$$\mathrm{cond}(M_m^{-1}A) = \frac{1 - \lambda_1^m}{1 - \lambda_n^m},\ m = 1, 2, \ldots.$$

Überdies fällt $\mathrm{cond}(M_m^{-1}A)$ streng monoton in m, sofern $\lambda_1 < \lambda_n$.

Beweis:

a) Nach Satz 4.2.24 aus dem Abschnitt über SSOR ist P spd und $P^T + Q = P + Q = (P + Q)^T$ spd. Mit L 4.6.7 ergibt sich deshalb a).

b) P ist spd $\Rightarrow P^{-1}$ ist spd und Q ist positiv semidefinit für alle $\omega \in (0,2)$. Mit Lemma 4.6.6 b) folgt so: $\mathrm{spek}(P^{-1}Q) \subseteq [0,+\infty)$. Nach Lemma 4.6.8 b) gilt also

$$\mathrm{cond}(M_m^{-1}A) = \frac{1-\lambda_1^m}{1-\lambda_n^m},\ m = 1,2,\ldots.$$

Zur Monotonie: Für $a,b,c,d > 0$ gilt[2]

$$cb < ad \Rightarrow \frac{a+c}{b+d} < \frac{a}{b}.$$

Damit gilt im Fall $\lambda_1 < \lambda_n$

$$\begin{aligned} \frac{1-\lambda_1^{m+1}}{1-\lambda_n^{m+1}} &= \frac{1-\lambda_1}{1-\lambda_n} \cdot \frac{(\overbrace{1+\ldots+\lambda_1^{m-1}}^{=a} + \overbrace{\lambda_1^m}^{=c})}{(\underbrace{1+\ldots+\lambda_n^{m-1}}_{=b} + \underbrace{\lambda_n^m}_{=d})} \\ &< \frac{1-\lambda_1}{1-\lambda_n} \cdot \frac{1+\ldots+\lambda_1^{m-1}}{1+\ldots+\lambda_n^{m-1}} = \frac{1-\lambda_1^m}{1-\lambda_n^m}. \end{aligned}$$

□

[2] denn: $cb < ad \Rightarrow ab + cb < ab + ad \Rightarrow \frac{a+c}{b+d} < \frac{a}{b}$

4.7 Unvollständige Dreieckszerlegung

Die unvollständige Dreieckszerlegung führt auf in der Praxis sehr erfolgreiche Präkonditionierer. Wir betrachten zunächst einmal den SSOR-Präkonditionierer speziell für $\omega = 1$. Diesen nennt man auch SGS[3]-Präkonditionierer

$$A = \underbrace{(D-L)D^{-1}(D-U)}_{=:M} - LD^{-1}U.$$

Dann ist

$$M\hat{r} = r$$

über

$$\begin{aligned}(D-L)r_1 &= r\\ r_2 &= Dr_1\\ (D-U)\hat{r} &= r_2\end{aligned}$$

zu lösen. Zusätzlich gilt noch $U = L^T$, da A spd ist.
Idee: Bestimme eine andere Zerlegung von A

$$A = (D' - L')(D')^{-1}(D' - U') - R$$

mit D' diagonal, L' strikte untere und U' strikte obere Dreiecksmatrix. Die Berechnung von D', L', U' „darf" nun Arbeit kosten, dafür soll aber $(D' - L')(D')^{-1}(D' - U')$ die Matrix A dann besser approximieren.

Die Matrix A wird zunächst *nicht* als symmetrisch vorausgesetzt. Es ergibt sich

$$\begin{aligned}A &= (D' - L')(D')^{-1}(D' - U') - R\\ &= \underbrace{(I - L'')}_{=:L}\underbrace{(D' - U')}_{=:U} - R, \quad L'' = L'(D')^{-1}\\ &= LU - R \qquad (4.7.1)\end{aligned}$$

mir einer unteren Dreiecksmatrix L, $I = diag(L)$ und einer oberen Dreiecksmatrix U.

Beachte: Ab jetzt bezeichnen L und U die Matrizen aus Gleichung (4.7.1).

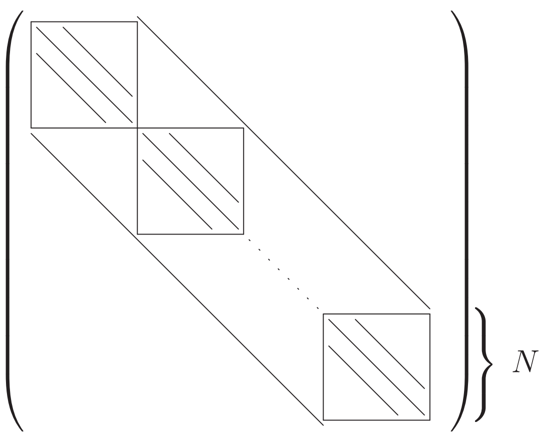
4.7.1 Bemerkung
Ist $R = 0$ in (4.7.1), so ist $A = LU$. Dann ist LU idealer Präkonditionierer, da $(LU)^{-1}A = I$. Die Berechnung von LU ist dann aber die Durchführung des

[3] symmetrischen Gauß-Seidel

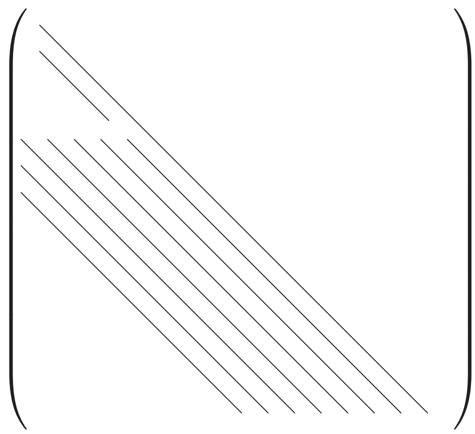
Gauß-Algorithmus. Damit würde aber das präkonditionierte CG-Verfahren zwar nach einem Schritt konvergieren, aber man kann dann das lineare Gleichungssystem gleich über den Gauß-Algorithmus lösen. L und U enthalten im Allgemeinen wesentlich mehr Nicht-Nullen als A. Deshalb ist die Berechnung sehr teuer und sogar eventuell mit Speicherproblemen verbunden, weshalb man gerade Iterationsverfahren verwendet,

4.7.2 Beispiel
Es sei A die Matrix des Modellproblems I. Dann besitzt A fünf Nicht-Nullen pro Zeile. Bei zeilenweiser Anordnung ergäbe sich dann für $A = L \cdot U$

$A =$



$L =$



Da (im Wesentlichen) $U = L^T$ gilt, ergeben sich N Nicht-Nullen pro Zeile in L und U.

Die Idee bei der unvollständigen Faktorisierung ist es nun, Besetztheitsmuster für L und U vorzuschreiben. Damit ist dann aber im Allgemeinen auch $R \neq 0$.

4.7.3 Definition
Es sei $n \in \mathbb{N}$. Eine Menge

$$E \subseteq \{(i,j) \in \{1,...,n\}^2 |\ i,j \in \{1,...,n\}\}$$

heißt **Muster**, falls

$$(i,i) \in E \quad \text{für } i = 1,...,n.$$

4.7.4 Definition
Es sei $A \in \mathbb{R}^{n \times n}$. Eine **unvollständige LU-Zerlegung** (**ILU**[4]) von A zum Muster E ist eine Zerlegung von A der Form

$$A = L \cdot U - R, \tag{4.7.2}$$

wenn für $L = (\ell_{i,j}),\ U = (u_{i,j}) \in \mathbb{R}^{n \times n}$ die Bedingungen

$$\begin{cases} \ell_{i,i} = 1 & i = 1,...,n, \\ \ell_{i,j} = 0 & i < j \quad (L \text{ untere Dreiecksmatrix}) \\ u_{i,j} = 0 & i > j \quad (U \text{ obere Dreiecksmatrix}) \end{cases} \tag{4.7.3}$$

und

$$\begin{cases} \ell_{i,j} = 0 & (i,j) \notin E, \\ u_{i,j} = 0 & (i,j) \notin E, \end{cases} \tag{4.7.4}$$

gelten.

Eine typische Wahl für E ist

$$E := \{(i,j) |\ a_{i,j} \neq 0\}.$$

(L, U benötigen dann genausoviel Speicher wie A.) Definition 4.7.4 legt eine ILU noch nicht vollständig fest. Von den $2n^2$ Parametern (Einträge von L und U) werden festgelegt:

$$\begin{array}{ll} \text{Elemente von } L\text{:} & \underbrace{n + \frac{n(n-1)}{2}}_{(4.7.3)} + \underbrace{\frac{n(n-1)}{2} - |E \cap \{(i,j)|\ i > j\}|}_{(4.7.4)} \\ \text{Elemente von } U\text{:} & \underbrace{\frac{n(n-1)}{2}}_{(4.7.3)} + \frac{n(n+1)}{2} - |E \cap \{(i,j)|\ i \leq j\}| \\ \hline \text{zusammen:} & 2n^2 - |E|. \end{array}$$

Man kann also genau die $|E|$ vorhandenen Nicht-Null-Elemente noch festlegen.

[4]incomplete LU-factorization

4.7.5 Definition
Die **direkte ILU-Zerlegung** der Matrix $A \in \mathbb{R}^{n\times n}$ bezüglich des Musters E ist durch (4.7.3), (4.7.4) und

$$\begin{aligned} a_{i,j} = (L\cdot U)_{i,j} &= \sum_{k=1}^{n} \ell_{i,k}u_{k,j} \\ &= \sum_{\substack{k=1 \\ (i,k)\in E \wedge (k,j)\in E}}^{\min\{i,j\}} \ell_{i,k}u_{k,j} \qquad \text{für } (i,j)\in E \end{aligned} \tag{4.7.5}$$

festgelegt. Dies ist äquivalent dazu, dass $r_{i,j} = 0$ für $(i,j) \in E$ gilt.

Gleichung (4.7.5) stellt $|E|$ zusätzliche Bedingungen an die Einträge von L und U.

Wie bei der ijk-Form der Gauß-Elimination kann man aus (4.7.5) die $\ell_{i,k}, u_{k,j}$ sukzessive berechnen.

4.7.6 Algorithmus (Direkte ILU-Zerlegung)

for $i = 1, ..., n$ **do**
  for $j = i, ..., n$ **do**
    if $(i,j) \in E$ **then**
      $u_{i,j} = a_{i,j} - \sum_{\substack{k=1 \\ (i,k)\in E\wedge(k,j)\in E}}^{i-1} \ell_{i,k}u_{k,j}$
    end if
  end for
  for $s = i+1, ..., n$ **do**
    if $(i,s) \in E$ **then**
      $\ell_{s,i} = \frac{1}{u_{i,i}} \left(a_{s,i} - \sum_{\substack{k=1 \\ (s,k)\in E\wedge(k,i)\in E}}^{i-1} \ell_{s,k}u_{k,i} \right)$
    end if
  end for
end for

Beachte die formale Verwandtschaft zur Gauß-Elimination. Das Verfahren verwendet aber *keine* „Pivot“-Strategie. Wann existiert also eine ILU?

4.7.7 Satz
Sei $A \in \mathbb{R}^{n\times n}$ eine H-Matrix, d.h. es gibt ein $v > 0$[5] mit

$$|a_{i,i}|v_i > \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{i,j}|v_j, \quad i = 1, ..., n.$$

Dann existiert die direkte ILU-Zerlegung

$$A = LU - R$$

zu *jedem* Muster E.

Beweis: Die einzig problematische Stelle ist die Aufdatierung der $\ell_{j,i}$, d.h. zu zeigen ist, dass $u_{i,i} \neq 0$ für alle i gilt. Dies zeigen wir mittels Induktion über n.

$n = 1$: Für $n = 1$ ist

$$a_{1,1} = \underbrace{1}_{L} \cdot \underbrace{a_{1,1}}_{U} - \underbrace{0}_{R}$$

die direkte ILU-Zerlegung . Wegen $a_{1,1} = u_{1,1}$ und weil

$$|a_{1,1}|v > 0 \iff a_{1,1} \neq 0$$

ist, folgt auch $u_{1,1} \neq 0$.

$n-1 \to n$: Sei $A \in \mathbb{R}^{n\times n}$ eine H-Matrix mit zugehörigen Vektor v. Wegen $|a_{1,1}|v_1 > 0$ gilt $a_{1,1} \neq 0$. Entsprechend Algorithmus 4.7.6 setzen wir

$$\begin{aligned} u_{1,j} &= \begin{cases} a_{1,j} & (1,j) \in E \\ 0 & \text{sonst} \end{cases} \\ \ell_{i,1} &= \begin{cases} \frac{a_{i,1}}{a_{1,1}} & (i,j) \in E \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

Damit sind die Gleichungen (4.7.3), (4.7.4) und (4.7.5), für alle Indexpaare $(1, j)$ und $(i, 1)$ erfüllt. Wir setzen nun

$$\begin{aligned} \bar{a}_{i,j} &= a_{i,j} - \ell_{i,1}a_{1,j}, \quad i, j = 2, ..., n \\ \bar{A} &= (\bar{a}_{i,j})_{i,j=2,...,n}. \end{aligned}$$

[5] zur Erinnerung: $v > 0 \iff v_i > 0, \quad i = 1, ..., n$

$\bar{A}$ ist wiederum eine H-Matrix, denn es gilt mit dem Vektor $\bar{v} = (v_2, ..., v_n) > 0$ für $i = 2, ..., n$

$$\begin{aligned}
\sum_{\substack{j=2\\j\neq i}}^{n} |\bar{a}_{i,j}|v_j &= \sum_{\substack{j=2\\j\neq i}}^{n} |a_{i,j} - \ell_{i,1}u_{1,j}|v_j \\
&\leq \sum_{\substack{j=2\\j\neq i}}^{n} |a_{i,j}|v_j + \sum_{\substack{j=2\\j\neq i}}^{n} |\ell_{i,1}u_{1j}|v_j \\
&< -|a_{i,1}|v_1 + |a_{i,i}|v_i + \sum_{\substack{j=2\\j\neq i}}^{n} |\ell_{i,1}u_{1j}|v_j \\
&\leq -|a_{i,1}|v_1 + |a_{i,i}|v_i + \sum_{\substack{j=2\\j\neq i}}^{n} \left|\frac{a_{i,1}}{a_{1,1}}u_{1j}\right| v_j \\
&= -|a_{i,1}|v_1 + |a_{i,i}|v_i + \left|\frac{a_{i,1}}{a_{1,1}}\right| \cdot \sum_{\substack{j=2\\j\neq i}}^{n} |u_{1j}|\, v_j \\
&\leq -|a_{i,1}|v_1 + |a_{i,i}|v_i + \left|\frac{a_{i,1}}{a_{1,1}}\right| \cdot (|a_{1,1}v_1 - |a_{1,i}|v_i) \\
&= |a_{i,i}|v_i - \left|\frac{a_{i,1}}{a_{1,1}}a_{1,i}\right| \cdot v_i \\
&\leq |a_{i,i}|v_i - |\ell_{i,1}u_{1,i}| \cdot v_i \\
&\leq |a_{i,i} - \ell_{i,1}u_{1,i}| \cdot v_i \\
&= |\bar{a}_{i,i}| \cdot v_i.
\end{aligned}$$

Setze nun $\bar{E} = E \cap \{(i,j) \mid 2 \leq i, j \leq n\}$. Nach Induktionsannahme existiert die direkte ILU

$$\bar{A} = \bar{L}\bar{U} - \bar{R}$$

von $\bar{A}$ bezüglich $\bar{E}$. Setze

$$L = \begin{pmatrix} 1 & \\ \ell_{2,1} & \\ \vdots & \bar{L} \\ \ell_{n,1} & \end{pmatrix}, \; U = \begin{pmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ & & & \\ & & \bar{U} & \\ & & & \end{pmatrix}.$$

Dann ist $A = LU - R$ die direkte ILU von A.

□

4.7.8 Bemerkung
Die Existenz der ILU ist allerdings noch kein Garant dafür, dass sie ein guter Präkonditionierer ist. Nach Beispiel 4.7.2 wäre es zumindest günstig, wenn in $A = LU - R$ auch

$$\rho((LU)^{-1}R) < 1$$

gilt. Dies kann man erreichen, wenn A zusätzlichen Bedingungen unterliegt.

4.7.9 Definition
Eine Matrix $A = (a_{i,j}) \in \mathbb{R}^{n\times n}$ heißt **M-Matrix**, falls

1. $a_{i,j} \le 0$ für $i \neq j$,
2. es existiert ein Vektor $u > 0 \in \mathbb{R}^n$ mit $Au > 0$.

4.7.10 Satz
Sei $A \in \mathbb{R}^{n\times n}$ eine M-Matrix. Dann existiert die direkte ILU-Zerlegung zu jedem Muster E. Speziell für die Muster E mit

$$\{(i,j):\ a_{i,j} \neq 0\} \subseteq E$$

gilt zusätzlich sogar

$$\rho((LU)^{-1}R) < 1.$$

Beweis: Übungsaufgabe. □

4.7.11 Definition
Wir definieren **die ILU von** $A \in \mathbb{R}^{n\times n}$ als die direkte ILU von A zum Muster

$$E = E(A) = \{(i,j):\ a_{i,j} \neq 0\}.$$

Sei jetzt A zusätzlich spd,

$$A = LU - R$$

die ILU von A, $D = \operatorname{diag}(U), U' = D^{-1}U$. Also gilt

$$\begin{aligned} A &= LDU' - R \\ A^T = A &= LDU' - R = (U')^T D^T L^T - R^T. \end{aligned}$$

Hierbei ist damit auch $(U')^T D^T L^T - R^T$ die ILU von A. Damit gilt $U' = L^T$, also liefert

$$A = LDL^T - R$$

eine **unvollständige wurzelfreie Cholesky-Zerlegung**. Mit dem Übergang $D = D^{1/2}D^{1/2},\ L' = LD^{1/2}$ erhält man schließlich die übliche **unvollständige Cholesky-Zerlegung** (IC[6])

$$A = (L')(L')^T - R.$$

Wie bei der ILU kann man die Matrix L' (die wir im folgenden wieder lediglich mit L bezeichnen wollen) über die ijk-Form berechnen. Dabei kann statt $E(A)$ ein beliebiges symmetrisches Muster E verwendet werden. Ein Muster E heißt dabei **symmetrisch**, falls stets

$$(i,j) \in E \Rightarrow (j,i) \in E$$

gilt.

4.7.12 Algorithmus (Direkte unvollständige Cholesky-Zerlegung)

A sei spd, E ein symmetrisches Muster
for $i = 1, ..., n$ **do**
$$\ell_{i,i} = \sqrt{a_{i,i} - \sum_{\substack{k=1\\(i,k)\in E}}^{i-1} \ell_{i,k}^2}$$
for $j = i+1, ..., n$ **do**
if $(i,j) \in E$ **then**
$$\ell_{j,i} = \frac{1}{\ell_{i,i}}\left(a_{j,i} - \sum_{\substack{k=1\\(i,k)\in E\wedge(j,k)\in E}}^{i-1} \ell_{j,k}\ell_{i,k}\right)$$
end if
end for
end for

Bezeichnung: Unvollständige Cholesky-Zerlegung bedeutet direkte IC zum Muster $E(A)$.

Beachte: Auch wenn A spd ist, bedeutet das nicht *notwendig*, dass die IC existiert! Selbst wenn sie existiert, ist $A = LL^T - R$ nicht unbedingt P-regulär, und LL^T braucht kein „guter“ Präkonditionierer sein.

Dies führt zu der Frage nach anderen ILU- (IC-) Zerlegungen.

Motivation: Bei vielen Diskretisierungen von partiellen Differentialgleichungen ist $e = (1, ..., 1)$ eine gute Approximation für die Eigenvektoren von A zu kleinen Eigenwerten („glatte Moden“).

[6]Incomplete Cholesky Factorisation

Darum ist es wünschenswert, dass die Bedingung

$$Ae = (LU)e \quad \text{in } A = LU - R$$

gilt. (Für die präkonditionierte Matrix $(LU)^{-1}A$ werden dann die kleinen Eigenwerte in eine kleine Umgebung der 1 abgebildet. Damit wird auch $\mathrm{cond}((LU)^{-1}A)$ wesentlich kleiner.)

Neben den Gleichungen (4.7.3) und (4.7.4) fordert man deshalb bei der **modifizierten ILU Zerlegung** (MILU)

$$a_{i,j} = \sum_{k=1}^{n} \ell_{i,k} u_{k,j} \quad \text{für } (i,j) \in E,\ i \neq j \tag{4.7.6a}$$

$$r_{i,i} = -\sum_{\substack{j=1 \\ j\neq i}}^{n} r_{i,j} \quad i = 1, ..., n \tag{4.7.6b}$$

$$(\Longleftrightarrow (Ae)_i = (LUe)_i).$$

Wie konstruiert man ein Verfahren hierzu? Betrachten wir dazu zuerst die kji-Form der direkten ILU.

4.7.13 Algorithmus (Direkte ILU zum Muster E, kji-Form)

for $k = 1, ..., n$ **do**
  for $j = k + 1, ..., n$ **do**
    if $(j, k) \in E$ **then**
      $\ell_{j,k} = a_{j,k}/a_{k,k}$
    end if
  end for
  for $i = k + 1, ..., n$ **do**
    for $j = k + 1, ..., n$ und $(i, j) \in E$ und $(k, j) \in E$ **do**
      $a_{i,j} = a_{i,j} - \ell_{i,k} a_{k,j}$ (∗)
    end for
  end for
end for

Hieraus bekommt man die MILU durch geeignete Modifikation von (∗).

4.7.14 Algorithmus (MILU (Modifikation von (*) in Algorithmus 4.7.13)

for $j = k+1, ..., n$ und $(k, j) \in E$ **do**
  if $(i, j) \in E$ **then**
    $a_{i,j} = a_{i,j} - \ell_{i,k} a_{k,j}$
  else
    $a_{i,i} = a_{i,i} - \ell_{i,k} a_{k,j}$ („Diagonale Kompensation")
  end if
end for

4.7.15 Beispiel

Wir betrachten die ILU-Zerlegung bei einem beliebigem 5-Punkte-Stern:

Das Gleichungssystem sei gegeben durch:

$$a_{\nu,\mu} u_{\nu-1,\mu} + b_{\nu,\mu} u_{\nu,\mu-1} + c_{\nu,\mu} u_{\nu,\mu} + d_{\nu,\mu} u_{\nu+1,\mu} + e_{\nu,\mu} u_{\nu,\mu+1} = f_{\nu,\mu} \quad \nu, \mu = 1, ..., N.$$

Zusätzlich seien die Randwerte

$$u_{0,\mu}, u_{N+1,\mu}, u_{\nu,0}, u_{\nu,N+1}$$

bekannt (im Modellproblem I gilt also: $a_{\nu,\mu} = b_{\nu,\mu} = d_{\nu,\mu} = e_{\nu,\mu} = -1$ und $c_{\nu,\mu} = 4$). Es sei $W = (w_{i,j}) \in \mathbb{C}^{N^2 \times N^2}$ die zugehörige Matrix. Statt der Indizes i und j schreiben wir jetzt Paare (ν, μ), wobei $i = (\nu - 1)N + \mu$ gelte (d.h. wir betrachten eine lexikographische Anordnung). Es gilt dann

$$w_{i,j} = w_{(\nu,\mu),(\rho,\sigma)} = \begin{cases} a_{\nu,\mu} & \rho = \nu - 1, \sigma = \mu \\ b_{\nu,\mu} & \rho = \nu, \sigma = \mu - 1 \\ c_{\nu,\mu} & \rho = \nu, \sigma = \mu \\ d_{\nu,\mu} & \rho = \nu + 1, \sigma = \mu \\ e_{\nu,\mu} & \rho = \nu, \sigma = \mu + 1 \\ 0 & \text{sonst} \end{cases}.$$

Das zu W gehörige Muster $E(W)$ ergibt sich dann (alle $a_{\nu,\mu}, b_{\nu,\mu}, c_{\nu,\mu}, d_{\nu,\mu}, e_{\nu,\mu}$ werden als nichtverschwindend angenommen) als

$$E(W) = \{((\nu, \mu), (\rho, \sigma)) \in \{1, ..., N\}^2 \times \{1, ..., N\}^2 : \ |\nu - \rho| + |\mu - \sigma| \leq 1\}.$$

Im Folgenden benutzen wir $E = E(W)$ und setzen W als symmetrisch voraus (damit folgt dann $a_{\nu,\mu} = d_{\nu-1,\mu}, b_{\nu,\mu} = e_{\nu,\mu-1}$ und $c_{\nu,\mu} \in \mathbb{R}$), um die zugehörige unvollständige direkte Cholesky-Zerlegung nach Algorithmus 4.7.12 zu bestimmen. Die Matrix $L = (\ell_{((\nu,\mu),(\rho,\sigma))})$ besitzt also die Struktur

$$\ell_{((\nu,\mu),(\rho,\sigma))} = \begin{cases} \ell a_{\nu,\mu} & \rho = \nu - 1, \sigma = \mu \\ \ell b_{\nu,\mu} & \rho = \nu, \sigma = \mu - 1 \\ \ell c_{\nu,\mu} & \rho = \nu, \sigma = \mu \\ 0 & \text{sonst.} \end{cases}$$

Nach Algorithmus 4.7.12 erhalten wir so

$$\left.\begin{array}{l}\textbf{for } \nu = 1,...,N \textbf{ do}\\ \quad \textbf{for } \mu = 1,...,N \textbf{ do}\end{array}\right\} \widehat{=} \text{ Schleife über } i$$

$$(4.7.7) \qquad \begin{aligned}(\ell c)_{\nu,\mu} &= \sqrt{c_{\nu,\mu} - \sum_{\substack{(\rho,\sigma)=(1,1)\\((\nu,\mu),(\rho,\sigma))\in E}}^{(\nu,\mu-1)} |\ell_{(\nu,\mu),(\rho,\sigma)}|^2}\\ &= \sqrt{c_{\nu,\mu} - |\ell a_{\nu,\mu}|^2 - |\ell b_{\nu,\mu}|^2}\end{aligned}$$

{Die Schleife über j mit „**if** $(i,j) \in E$ **then** reduziert sich auf die Berechnung von $\ell a_{\nu+1,\mu}$ und $\ell b_{\nu,\mu+1}$}

$$\begin{aligned}(4.7.8)\ \ell_{(\nu+1,\mu),(\nu,\mu)} &= \ell a_{\nu+1,\mu}\\ &= \frac{1}{(\ell c)_{\nu,\mu}}\left(a_{\nu+1,\mu} - \sum_{\substack{(\rho,\sigma)=(1,1)\\((\nu,\mu),(\rho,\sigma))\in E}}^{(\nu,\mu-1)} \ell_{(\nu+1,\mu),(\rho,\sigma)}\ell_{(\nu,\mu),(\rho,\sigma)}\right)\\ &= \frac{1}{(\ell c)_{\nu,\mu}} a_{\nu+1,\mu}\\ (4.7.9)\ \ell_{(\nu,\mu+1),(\nu,\mu)} &= \ell b_{\nu,\mu+1} = \frac{1}{(\ell c)_{\nu,\mu}} b_{\nu,\mu+1} \quad \text{(analog).}\end{aligned}$$

end for
end for

Die Summe bei $\ell a_{\nu+1,\mu}$ ist leer, denn

$$\begin{aligned} &((\nu+1,\mu),(\rho,\sigma)) \in E(W) \wedge ((\nu,\mu),(\rho,\sigma)) \in E(W)\\ \Longleftrightarrow\ & |\rho - (\nu+1)| + |\sigma - \mu| \leq 1 \wedge |\rho - \nu| + |\sigma - \mu| \leq 1\\ \Longleftrightarrow\ & \sigma = \mu \wedge (\rho = \nu \vee \rho = \nu + 1),\end{aligned}$$

was wegen der Obergrenze im Indexbereich aber nie eintritt.
Durch Einsetzen von (4.7.8), (4.7.9) in (4.7.7) erhalten wir

$$(4.7.10) \qquad (\ell c)_{\nu,\mu} = \sqrt{c_{\nu,\mu} - \left|\frac{a_{\nu,\mu}}{(\ell c)_{\nu-1,\mu}}\right|^2 - \left|\frac{b_{\nu,\mu}}{(\ell c)_{\nu,\mu-1}}\right|^2} \quad \nu,\mu = 1,...,N.$$

Setzen wir schließlich $D = \operatorname{diag}((\ell c)_{\nu,\mu})$ und $\widetilde{L} = LD$, so erhalten wir für die unvollständige Cholesky-Zerlegung die Darstellung

$$LL^T = LDD^{-2}DL^T \ = \ (LD)D^{-2}(DL^T)$$

$$= \quad \widetilde{L}^T \widetilde{L}^T,$$

wobei $\widetilde{L}$ dieselbe Struktur hat wie L und

$$(4.7.11) \quad \begin{cases} \begin{rcases} (\widetilde{\ell a})_{\nu,\mu} = a_{\nu,\mu} \\ (\widetilde{\ell b})_{\nu,\mu} = b_{\nu,\mu} \end{rcases} \quad \nu, \mu = 1, ..., N, \\ (\widetilde{\ell c})_{\nu,\mu} = (\ell c)^2_{\nu,\mu} \text{ mit (nach (4.7.10)):} \\ \qquad (\widetilde{\ell c})_{\nu,\mu} = c_{\nu,\mu} - \left| \dfrac{a_{\nu,\mu}}{(\widetilde{\ell c})_{\nu-1,\mu}} \right|^2 - \left| \dfrac{b_{\nu,\mu}}{(\widetilde{\ell c})_{\nu,\mu-1}} \right|^2, \quad \nu, \mu = 1, ..., N, \\ T = \operatorname{diag}(t_{\nu,\mu}) \text{ mit } t_{\nu,\mu} = \frac{1}{(\widetilde{\ell c})_{\nu,\mu}}, \quad \nu, \mu = 1, ..., N. \end{cases}$$

Literaturverzeichnis

[1] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[2] Andreas Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern*. Vieweg Verlag, 1990.

[3] Walter Huber. *Paralleles Rechnes*. Oldenbourg, München,)1997.

[4] James Ortega. *Introduction to the Vector and Parallel Solution of Linear Systems*. Plenum Press, New York, 1988.

[5] L. Ridgeway Scott, Terry Clark, and Babak Bagheri. *Scientific Parallel Computing*. Princeton University Press, Princeton, USA, 2005.