Diplomarbeit

Der Block Lanczos Algorithmus über GF(2)

Olaf Groß

Nach einem Thema von Prof. J.Buchmann
FB 14— Informatik—der Universität des Saarlandes

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen verwendet habe.

Uchtelfangen, den 04.07.1994

Mein Dank gebührt Herrn Prof. Dr. Buchmann für die Vergabe dieses interessanten Themas.
Des weiteren bedanke ich mich bei Jörg Zayer für die motivierende Betreuung während der Ausarbeitung der Arbeit.
Insbesondere möchte ich mich auch bei Don Coppersmith für seine Bereitschaft bedanken, mir bei schwierigen Detailfragen zu dem von ihm erarbeiteten Algorithmus zu helfen.
Außerdem bedanke ich mich bei Jörg Zayer und Martin Schido für das Korrekturlesen der Arbeit.
Mein besonderer Dank gilt meinen Eltern, die mein Studium ermöglichten und mich nach besten Kräften unterstützt haben.

Vorwort

Diese Arbeit beschäftigt sich mit dem Lösen von großen dünnbesetzten Gleichungssystemen über $GF(2)$.

Generell teilt man Algorithmen zum Lösen von linearen Gleichungssystemen in zwei Klassen ein :

1. direkte Lösungsverfahren(z.B. Gauss-Elimination, Cholesky-Zerlegung, vgl. auch [15])
2. iterative Lösungsverfahren(z.B. Mehrgitterverfahren, konjugierte Gradienten, vgl. auch [6])

Aber welcher Unterschied könnte zwischen der Lösung großer und kleiner Gleichungssysteme bestehen? Die in der Linearen Algebra geläufigen Verfahren sind für jede Dimension — egal ob groß oder klein — anwendbar. Aber der benötigte Rechenaufwand steigt mit der Dimension des Gleichungssystems so stark an, daß man zur Lösung eines Systems mit 10.000, 100.000 oder einer Million Gleichungen und ebensovielen Variablen nach einem besseren Verfahren suchen muß. Diese Suche wird bestimmt durch die speziellen Eigenschaften der Matrizen, die diese Gleichungssysteme in der Praxis haben.
Ein spezielles Merkmal, daß unter anderem auch zu dem in der folgenden Arbeit behandelten Algorithmus führte, ist die Ausdünnung eines Gleichungssystems, d.h. die Mehrzahl der in dem Gleichungssystem auftretenden Koeffizienten sind Null.
Ein wichtiges praktisches Beispiel für das Auftreten von großen dünnen Gleichungssystemen ist der letzte Abschnitt bei den meisten Algorithmen zur Faktorisierung großer ganzer Zahlen (z.B quadratic sieve, number field sieve), in dem sehr große schwachbesetzte Matrizen der Dimension 100.000×100.000 und mehr auftreten.

Der hier beschriebene Algorithmus zur Lösung von großen, dünnen Gleichungssystemen ist eine spezielle Anwendung des Lanczos–Algorithmus (vgl. [9] und [10]) über $GF(2)$, dem Körper mit zwei Elementen. Er ist mit der structured Gauss Elimination vergleichbar, was die Laufzeit betrifft, stellt aber bedeutend geringere Speicheranforderungen (siehe hierzu auch [8]).

Meine Arbeit gliedert sich in drei große Abschnitte und einen Anhang :

Abschnitt I:

Dieser Abschnitt enthält lediglich das Kapitel 1.
Hier werden die benutzten Bezeichnungen, Definitionen und Sätze vorgestellt, die Arithmetik in $GF(2)$ dargestellt, sowie die Darstellung der Algorithmen durch Struktogramme erläutert.

Abschnitt II:

Dieser Abschnitt enthält die Kapitel 2 und 3 und beschäftigt sich mit den theoretischen Aspekten des Block Lanczos Algorithmus über $GF(2)$.
Im einzelnen enthalten die Kapitel :

Kapitel 2:

Hier erfolgt eine kurze Wiederholung des allgemeinen Lanczos–Algorithmus zum Lösen von linearen Gleichungssystemen und eine Beschreibung der in dieser speziellen Version vorgenommenen Änderungen. Außerdem gehe ich hier auf den Unterschied zwischen homogenen und inhomogenen Gleichungssystemen und auf die unterschiedliche Behandlung von symmetrischen und asymmetrischen Matrizen ein.

Kapitel 3:

Hier beschreibe ich die grundlegende Struktur des Algorithmus, erläutere die Ausführung der Berechnungen, und gebe einen kurzen Überblick über das Laufzeitverhalten des Algorithmus.

Abschnitt III:

Dieser Abschnitt enthält die Kapitel 4 bis 7 und beschreibt die Implementierung des Algorithmus und die Daten einiger Testläufe. Die einzelnen Kapitel behandeln dabei folgende Themen :

Kapitel 4:

In diesem Kapitel wird die Struktur der Implementierung des Block Lanczos Algorithmus beschrieben.

Kapitel 5:

In diesem Kapitel werden die zur Berechnung neu eingeführten Datentypen beschrieben und ihre Verwendung im Programm erläutert.

Kapitel 6:

Dieses Kapitel enthält die Beschreibung der Struktur von Algorithmen für einfache und spezielle Operationen auf den neu eingeführten Datentypen.

Kapitel 7:

In diesem Kapitel werden die Daten einiger Testläufe angeführt.

Anhang :

Der Anhang enthält eine Beschreibung des LOG-Files, d.h. eines Files, das einige Daten zum Programmablauf, wie z.B. Laufzeit des Progamms bzw. der einzelnen Teilabschnitte, benutzter Speicher usw. enthält, eine Beschreibung des Formates der abgespeicherten Matrizen und Vektoren und der Parameter des Programmaufrufs.

Inhaltsverzeichnis

Tabellenverzeichnis

Kapitel 1

Grundlagen

Im einführenden Kapitel werde ich die benutzten Bezeichnungen und Definitionen vorstellen, die Formulierung und Darstellung der Algorithmen erklären und die benutzte O-Notation besprechen. Außerdem erläutere ich die Grundoperationen in $GF(2)$ und ihre Darstellung im Computer.
Dieses Kapitel ist unterteilt in die folgenden Abschnitte :

- benutzte Bezeichnungen
- benutzte Definitionen
- Operationen in $GF(2)$
- Darstellung der Algorithmen

1.1 Bezeichnungen

Die Wortgröße in Bit der Maschine wird mit `WordSize` bezeichnet (diese ist z.B bei einem IBM-PC 16, bei einer SPARC-Station 32).

Der zugrundeliegende Körper ist $GF(2) \hat{=} \mathbb{Z}/2\mathbb{Z}$.

Dünnbesetzte Matrizen werden mit den Großbuchstaben A, B bezeichnet.
Die Größe einer Matrix A ist $N \times N$.
Die Größe einer Matrix B ist $N \times M$.
Die Anzahl Einträge $\neq 0$ in einer Matrix A oder B ist T.
Die durchschnittliche Anzahl Einträge $\neq 0$ pro Zeile einer Matrix A oder B wird mit D bezeichnet.
Die Koeffizienten einer Matrix A werden mit $a_{i,j}, 0 < i, j \leq N$ bezeichnet.

Die Transponierte einer Matrix A wird mit $A^{\mathbf{T}}$ bezeichnet.

Wenn eine Zeile einer Matrix A ein einzelnes Element $\neq 0$ enthält, so nennen wir dieses Element ein 'Singleton' .

Mit I_k bezeichnen wir die Identitätsmatrix aus $GF(2)^{k \times k}$.

Kleine Buchstaben i, j, k, l usw. werden, wenn nicht anders angegeben, Elemente aus $\mathbb{N}$ bezeichnen.

Eine kleine Matrix ist eine Matrix der Dimension $k \times l \quad k, l \leq$ `WordSize` über $GF(2)$. Griechische Buchstaben α, β usw. werden solche kleine Matrizen über $GF(2)$ bezeichnen.

1.2 Benutzte Definitionen

Zur Erläuterung von Standarddefinitionen und –sätzen aus der linearen Algebra verweise ich auf Lehrbücher zu diesem Thema, z.B auf [16] oder [5].

Definition 1.1 (O-Notation)
Für zwei Funktionen $f, g : \mathbb{N} \to \mathbb{R}$ sagen wir :

$$f \in O(g) \iff \exists_{n_0, c \in \mathbb{N}} \ \forall_{n \geq n_0} \ f(n) \leq c \cdot g(n)$$

$$f \in o(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Streng genommen, bezeichnen $O(g)$ und $o(g)$ Mengen von Funktionen. Es hat sich aber eingebürgert, die O-Notation zusammen mit dem Gleichheitszeichen zu benutzen anstatt mit den Zeichen $\in$ und $\subseteq$ für Elementbeziehung bzw. Enthaltensein.

Definition 1.2 (Multiplikation von Matrizen)
Sei $A = (a_{i,j})$ eine $k \times m$–Matrix, $B = (b_{i,j})$ eine $m \times l$–Matrix und $C = (c_{i,j})$ eine $k \times l$–Matrix über $GF(2)$.
Dann setzen wir $C = A \cdot B$, wenn für jede Komponente $c_{i,j}$ von C gilt :

$$c_{i,j} = \sum_{k=0}^{m} a_{i,k} \cdot b_{k,j}.$$

Definition 1.3 (Vektorblock)
Seien $u_i, 1 \leq i \leq k, k \leq$ `WordSize` Vektoren aus $GF(2)^N$.
Ein Vektorblock $\mathbf{u}$ ist dann eine $N \times k$–Matrix über $GF(2)$, in der jede Spalte i einem Vektor u_i entspricht, also $\mathbf{u} = (\, u_1 \quad u_2 \quad \ldots \quad u_k \,)$.

Zu näheren Erläuterungen zu Vektorblöcken vgl. Kap. 2.4.
Die Vektoren der Vektorblöcke sind in der folgenden Arbeit, wenn nicht anders angegeben, Vektoren aus $GF(2)^N$.

Definition 1.4 (Größe eines Vektorblocks)
Sei $\mathbf{u}$ *ein Vektorblock.*
Dann wird die Größe von $\mathbf{u}$ *definiert als die Anzahl* k *Spalten dieses Vektorblocks, also*

$$\mathtt{size}(\mathbf{u}) = k.$$

Definition 1.5 (Addition/Subtraktion von Vektorblöcken)
Seien $u_i, v_i, w_i, \quad 1 \leq i \leq k$ *Vektoren aus* $GF(2)^N$, $\mathbf{u}$, $\mathbf{v}$ *und* $\mathbf{w}$ *die von diesen Vektoren gebildeten Vektorblöcke, wobei* $\mathtt{size}(\mathbf{u}) = \mathtt{size}(\mathbf{v}) = \mathtt{size}(\mathbf{w}) = k$.
Man setzt $\mathbf{w} = \mathbf{u} \pm \mathbf{v}$, *wenn gilt :*

$$\underset{1 \leq i \leq k}{\forall} : w_i = u_i \pm v_i.$$

Definition 1.6 (Multiplikation von Vektorblöcken)
Seien $\mathbf{u}$ *und* $\mathbf{v}$ *Vektorblöcke, wobei* $\mathtt{size}(\mathbf{u}) = k, \mathtt{size}(\mathbf{v}) = l$. *Sei* α *eine* $k \times l$–*Matrix über* $GF(2)$.
Dann setzen wir $\alpha = \mathbf{u}^{\mathbf{T}} \cdot \mathbf{v}$, *wenn für jede Komponente* $\alpha_{i,j}$ *von* α *gilt :*

$$\alpha_{i,j} = \sum_{k=0}^{N} \mathbf{u}_{k,i} \cdot \mathbf{v}_{k,j}.$$

Die Arithmetik von Vektorblöcken ist, wie aus Definition 1.5 und 1.6 ersichtlich, mit der Arithmetik von Matrizen identisch, d.h. die Addition bzw. Subtraktion zweier Vektorblöcke entspricht der Addition bzw. Subtraktion der sie darstellenden Matrizen, ebenso die Multiplikation zweier Vektorblöcke.

Definition 1.7 (Orthogonalität)
Sei K *ein Körper, seien* $\mathbf{t}$, $\mathbf{u}$ *Vektoren aus* K^N.
Zwei Vektoren $\mathbf{t}$, $\mathbf{u}$ *sind orthogonal, wenn*

$$\mathbf{t}^{\mathbf{T}}\mathbf{u} = 0$$

Diese Definition von Orthogonalität wird analog auf Blöcke von Vektoren ausgeweitet.

Definition 1.8 (orthogonales Komplement)
Sei t *Vektor aus* $GF(2)^N$.
Das orthogonale Komplement $t^{\perp}$ *ist der Raum, der von den Vektoren aus* $GF(2)^N$, *die orthogonal zu* t *sind, aufgespannt wird.*

Definition 1.9
Sei $\mathbf{u}$ *ein Vektorblock, der aus den Vektoren* $u_1, \ldots, u_k \in GF(2)^N$ *besteht. Dann definieren wir mit*

$$\mathrm{Span}(\mathbf{u}) = \left\{ v \mid v \in GF(2)^N, \quad \underset{i_1,\ldots,i_k \in GF(2)}{\exists} \quad v = i_1 u_1 + \ldots + i_k u_k \right\}$$

den von dem Vektorblock $\mathbf{u}$ *aufgespannten Raum.*

Definition 1.10
Seien $\mathbf{t}$ *und* $\mathbf{u}$ *Vektorblöcke und* t *und* u *die von ihnen aufgespannten Unterräume. Dann bezeichnen wir mit*

$$t \oplus u$$

den von diesen beiden Unterräumen aufgespannten Raum.

Der von t und u aufgespannte Raum wird durch $t \oplus u$ bezeichnet, um ihn von der komponentenweise Summe zweier Vektoren $\mathbf{t} + \mathbf{u}$ zu unterscheiden. Mit einer Ausnahme (wird erwähnt) wird die Notation $t \oplus u$ nur verwendet, wenn der Schnitt der beiden Unterräume leer ist.

Definition 1.11
Seien $\mathbf{w}_i, 0 \leq i \leq n$ *Blöcke von Vektoren aus* $GF(2)^N$ *und* $w_i, 0 \leq i \leq n$ *die von diesen Vektorblöcken aufgespannten Unterräume. Dann setzen wir*

$$W_n = w_0 \oplus w_1 \oplus \cdots \oplus w_n$$

Diese abkürzende Schreibweise wird in vielen späteren Formeln verwendet.

Definition 1.12
Sei A *eine* $N \times N$*–Matrix, seien* $\mathbf{u}_0, \ldots, \mathbf{u}_n$ *Vektorblöcke und* U *der von* $\mathbf{u}_0, \ldots, \mathbf{u}_n$ *aufgespannte Unterraum.*
Dann definiert man

$$A(U) = A \cdot U = \mathrm{Span}(A \cdot \mathbf{u}_0) \oplus \ldots \oplus \mathrm{Span}(A \cdot \mathbf{u}_n).$$

Analog definiert man

$$A^{-1}(U) = A^{-1} \cdot U = \mathrm{Span}(A^{-1} \cdot \mathbf{u}_0) \oplus \ldots \oplus \mathrm{Span}(A^{-1} \cdot \mathbf{u}_n).$$

Dies sind lediglich abkürzende Schreibweisen, die zur Übersichtlichkeit beitragen.

Definition 1.13 (orthogonale Projektion)
Sei A *eine* $N \times N$*–Matrix über* $GF(2)$*, sei* V *eine* $N \times M$*–Matrix über* $GF(2)$*, deren Spaltenvektoren paarweise orthogonal zueinander sind. Dann ist die Matrix*

$$B = V^{\mathbf{T}} A V$$

eine orthogonale Projektion von A *auf den Raum, der von den Spaltenvektoren von* V *aufgespannt wird.*

1.3 Arithmetik in $GF(2)$

In der folgenden Tabelle werden die Grundoperationen in $GF(2) \hat{=} \mathbb{Z}/2\mathbb{Z}$ aufgeführt.

a	b	a+b	a−b	a·b	a/b
0	0	0	0	0	undef.
0	1	1	1	0	0
1	0	1	1	0	undef.
1	1	0	0	1	1

Tabelle 1.1: Operationen in $GF(2)$

Man sieht, daß Addition und Subtraktion in $GF(2)$ bei gleichem Wert der Operanden zu gleichen Ergebnissen führen.
Man kann ein Element aus $GF(2)$ durch ein Bit im Computer darstellen (da $GF(2) \hat{=} \{\text{FALSE}, \text{TRUE}\}$). Daraus und aus Tabelle 1.1 ergibt sich unmittelbar, daß die Operationen Addition/Subtraktion von Elementen aus $GF(2)$ durch die XOR-Operation und die Multiplikation von Elementen aus $GF(2)$ durch die AND-Operation simuliert werden können.

1.4 Darstellung der Algorithmen

Zur Darstellung der Algorithmen verwende ich eine modifizierte Form der Nassi-Shneidermann-Diagramme, die in [12] ausführlich beschrieben werden. In diesen Diagrammen werden Algorithmen als Aneinanderreihung sogenannter Strukturblöcke dargestellt. Jeder Strukturblock erhält eine fortlaufende Nummer. Bei der Beschreibung der Algorithmen wird diese Nummer bei Verweisen benutzt.
Der einfachste dieser Strukturblöcke ist der Anweisungsblock:

Anweisung;	(1)

Unter einer Anweisung versteht man dabei eine Zuweisung oder einen Prozeduraufruf. Für den Rest dieses Abschnittes verwende ich den Anweisungsblock stellvertretend für einen oder mehrere solcher Anweisungsblöcke.

Zur Darstellung von Schleifen stehen folgende Strukturblöcke zur Verfügung:

1. Zählschleifen

FOR $i \leftarrow$ Anfangswert TO Endwert		(1)
	Anweisung;	(2)

Es handelt sich hierbei um eine Zählschleife, in der aufwärts gezählt wird. Die Zählvariable i wird mit dem *Anfangswert* initialisiert und dann bis zum *Endwert* hochgezählt. Dabei sind *Anfangswert* und *Endwert* Konstanten oder Variablen. Ist der *Anfangswert* größer als der *Endwert*, so nennt man die Zählschleife **abweisend**.

FOR $i \leftarrow$ Anfangswert DOWNTO Endwert		(1)
	Anweisung;	(2)

Es handelt sich hierbei um eine Variante der obigen Zählschleife, in der abwärts gezählt wird. Wie oben wird die Zählvariable i dem *Anfangswert* initialisiert, dann aber bis zum *Endwert* runtergezählt. Dieses Konstrukt wird als abweisend bezeichnet, wenn der *Anfangswert* kleiner als der *Endwert* ist.
Diese beiden Zählschleifenkonstrukte werden nur dann verwendet, wenn innerhalb des Schleifenrumpfes weder die Zählvariable, der *Anfangswert* noch der *Endwert* (falls es sich dabei um Variablen handelt) verändert werden.

2. Schleifen, deren Fortsetzung bzw. Beendigung von einer oder mehreren Bedingungen abhängt. Man unterscheidet dabei zwischen WHILE-Schleifen:

WHILE Bedingung		(1)
	Anweisung;	(2)

und REPEAT-Schleifen :

	Anweisung;	(1)
UNTIL Bedingung		(2)

Der Unterschied zwischen diesen beiden Konstrukten besteht darin, daß der Rumpf der WHILE-Schleife gegebenenfalls überhaupt nicht ausgeführt wird (abweisende Schleife), da die Abbruchbedingung zu Beginn jedes Schleifendurchlaufes überprüft wird. während der Rumpf der UNTIL-Schleife mindestens einmal ausgeführt wird (nicht abweisende Schleife), da die Abbruchbedingung erst am Ende jedes Schleifendurchlaufes überprüft wird.

Als Kontrollstrukturen stehen uns zur Verfügung:

IF	Bedingung	(1)
THEN	Anweisung1;	(2)
ELSE	Anweisung2;	(3)

Dabei wird die Bedingung ausgewertet und liefert als Wert entweder TRUE oder FALSE. Liefert die Auswertung TRUE, so wird Anweisung1 ausgeführt; bei FALSE wird Anweisung2 ausgeführt. Danach wird jeweils die Abarbeitung der Konstrukte beim nächsten Konstrukt fortgesetzt. Ist der Anweisungsteil 2 leer, so kann der ELSE-Fall auch weggelassen werden.

IF	Bedingung	(1)
THEN	Anweisung;	(2)

Neben den bisher erläuterten Strukturblöcken werden die Anweisung RETURN(z) oder RETURN("Text") verwendet.

RETURN(z);	(1)

oder

RETURN("Text");	(1)

Mit deren Hilfe wird eine Funktion (oder eine Prozedur) vorzeitig beendet und $z \in \mathbb{Z}$, bzw. der Text "Text" als Ergebnis zurückgeliefert.

Zur Kennzeichnung von Kommentaren innerhalb der Algorithmen verwende ich die in der Programmiersprache "C" übliche Notation :

Anweisung /* Kommentar */	(1)

oder

/* Kommentar */	(1)
Anweisung	(2)

Kapitel 2

Strategien des Algorithmus

In diesem Kapitel beschreibe ich den ursprünglichen Lanczos Algorithmus und verschiedene Probleme, die bei seiner Anwendung auf das Lösen von großen, dünnbesetzten Gleichungssystemen über $GF(2)$ entstehen. Ferner beschreibe ich hier eine Technik, um den Zeit– und Platzbedarf des Algorithmus zu verringern.
Dieses Kapitel ist unterteilt in fünf Abschnitte, die die folgenden Themen behandeln :

- Wiederholung des ursprünglichen Lanczos Algorithmus
- Probleme mit symmetrischen Matrizen
- Probleme mit selbstorthogonalen Vektoren
- Packen von mehreren Vektoren zu einem Vektorblock
- Preprocessing

2.1 Wiederholung des ursprünglichen Lanczos-Algorithmus

Beschreibung :
Der Lanczos Algorithmus wurde zuerst eingeführt, um die Eigenwerte und Eigenvektoren einer Matrix über $\mathbb{R}$ zu bestimmen (siehe [9]), und nur kurze Zeit später mit nur geringen Änderungen angewendet, um (dünnbesetzte) lineare Gleichungssysteme über $\mathbb{R}$ zu lösen (vgl. [10]). In dieser Version gehört er zu den iterativen Methoden zur Lösung von linearen Gleichungssystemen (vgl. zu solchen Methoden auch [6]).

Grundidee :
Die Grundidee dieses Algorithmus ist es, bei jeder Iteration einen Vektor zu berechnen, der paarweise orthogonal (bzw. konjugiert) zu allen in vorhergegangenen Iterationen bestimmten Versionen ist, und mit diesen Vektoren eine orthogonale Matrix , die denselben Rang wie die Koeffizientenmatrix des zu lösenden Systems hat, aufzubauen.

Durchführung :
Man löst die Gleichung

$$Ay = b \tag{2.1}$$

wobei A eine symmetrische $N \times N$–Matrix, $b \in \mathbb{R}^N$ ein Vektor $\neq 0$ und $y \in \mathbb{R}^N$ der zu bestimmende Vektor ist.
Man entwickelt hierzu zwei Folgen von Vektoren v_n, w_n :

$$\begin{aligned} w_0 &= b \\ v_1 &= Aw_0 \\ v_{n+1} &= Aw_n \\ w_{n+1} &= v_{n+1} - \alpha_n w_n - \beta_{n-1} w_{n-1} \end{aligned} \tag{2.2}$$

α_n und β_{n-1} sind hier Skalare, die so gewählt werden, daß :

$$w_i^{\mathbf{T}} A w_j = 0, \quad \text{wenn} \quad i \neq j, \tag{2.3}$$

d.h. die w_i, $\quad 0 \leq i \leq n-2$ sind paarweise konjugiert.
Für α_n und β_{n-1} gilt :

$$\begin{aligned} \alpha_n &= (w_n^{\mathbf{T}} v_{n+1})^{-1} (v_{n+1}^{\mathbf{T}} v_{n+1}), \\ \beta_{n-1} &= (w_{n-1}^{\mathbf{T}} v_n)^{-1} (v_{n+1}^{\mathbf{T}} v_n) \end{aligned} \tag{2.4}$$

Mit dieser Wahl von α_n, β_{n-1} ist w_{n+1}, wie in Gleichung 2.3 gefordert, zu jedem $w_i, 0 \leq i \leq n$ konjugiert. Weiter berechnet man den Vektor y als :

$$y = \sum_{i=0}^{n-1} b_i w_i, \tag{2.5}$$

wobei die b_i's Skalare sind, die durch

$$b_i = (w_i^{\mathbf{T}} v_{i+1})^{-1} (w_i^{\mathbf{T}} b). \tag{2.6}$$

bestimmt werden. Der Algorithmus terminiert wenn w_n zu sich selbst konjugiert ist, d.h.

$$w_n^{\mathbf{T}} A w_n = 0. \tag{2.7}$$

Dies tritt wegen Orthogonalität der $w_i, 0 \leq i \leq n$ für ein $n \leq N$ ein (Bew. siehe [10]). Ist $w_n = 0$, so liefert der kumulierte Wert von y dann den gesuchten Lösungsvektor des obigen Gleichungssystems, ist $w_n \neq 0$, versagt der Algorithmus.

Die Speicheranforderungen sind minimal; bei der Entwicklung von w_{n+1} und v_{n+1} benötigt man lediglich die Vektoren v_n, w_n und w_{n-1} und die Matrix A. (vgl. hierzu [10]).

Mit der Forderung, daß die w_i paarweise orthogonal sind, d.h.

$$w_i^{\mathbf{T}} w_j = 0 \text{ für } \quad i \neq j, \tag{2.8}$$

wird eine andere Berechnung der α_n und β_{n-1} bewirkt, nämlich

$$\begin{aligned} \alpha_n &= (w_n^{\mathbf{T}} w_n)^{-1}(w_n^{\mathbf{T}} v_{n+1}), \\ \beta_{n-1} &= (w_{n-1}^{\mathbf{T}} w_{n-1})^{-1}(w_{n-1}^{\mathbf{T}} v_{n+1}) \end{aligned} \tag{2.9}$$

So berechnet, ist w_{n+1} orthogonal zu allen $w_i \quad 0 \leq i \leq n$. Daraus entwickelt man eine Matrix W, deren Spalten aus den Vektoren w_i bestehen, und eine tridiagonale Matrix T, die den Gleichungen

$$\begin{aligned} W^{\mathbf{T}} W &= I \\ AW &= WT \end{aligned} \tag{2.10}$$

genügen. Diese werden verwendet, um die Eigenwerte und Eigenvektoren von A zu berechnen (näheres z.B. in [9] oder [4]).

Bei der speziellen Anwendung dieses Algorithmus auf den endlichen Körper $GF(2)$ treten einige Probleme auf. Diese werden in den folgenden Abschnitten besprochen.

2.2 Symmetrische Matrizen

Beschreibung :
Der eigentliche Lanczos-Algorithmus arbeitet mit einer symmetrischen $N \times N$–Matrix A und will die inhomogene Gleichung

$$A\mathbf{y} = \mathbf{b} \tag{2.11}$$

lösen. Bei der Anwendung z.B. für die Faktorisierung großer ganzer Zahlen hat man eine asymmetrische $N \times M$–Matrix $\hat{B}$, und versucht, nichttriviale Lösungen des homogenen Gleichungssystems

$$\hat{B}\hat{\mathbf{z}} = 0 \tag{2.12}$$

zu finden. Hierbei kann die Matrix $\hat{B}$ durchaus auch rechteckig sein.

Durchführung :
Um dieses Problem zu lösen, wandelt man zuerst das homogene Gleichungssystem in ein inhomogenes Gleichungssystem um. Dazu trennt man $\hat{B}$ auf in die rechten $k \leq$ `WordSize` Spalten $\mathbf{b}$, und die Restmatrix B. Dann versucht man, Lösungen zu der Gleichung

$$B\mathbf{z} = \mathbf{b} \tag{2.13}$$

zu finden. Hat man dieses System gelöst, setzt man

$$\hat{\mathbf{z}} = \begin{pmatrix} \mathbf{z} \\ -I_k \end{pmatrix}. \tag{2.14}$$

Das so gewählte $\hat{\mathbf{z}}$ löst nun die Gleichung (2.13), denn es gilt :

$$\begin{aligned} \hat{B}\hat{\mathbf{z}} &= (B \mid \mathbf{b})\hat{\mathbf{z}} \\ &= (B \mid \mathbf{b})\begin{pmatrix} \mathbf{z} \\ -I_k \end{pmatrix} \\ &= B\mathbf{z} - \mathbf{b}I_k \\ &= \mathbf{b} - \mathbf{b} = 0. \end{aligned} \tag{2.15}$$

Wegen der notwendigen Symmetrie setzt man

$$A = B \cdot B^{\mathbf{T}} \tag{2.16}$$

also ist A symmetrisch (vgl. [16]). Man berechnet oder speichert jedoch nie explizit das Matrizenprodukt A, da diese Matrix meistens viel dichter als B besetzt ist. Stattdessen benötigt man A nur als "Black Box", da man A auf einen Vektor (oder Vektorblock) $\mathbf{w}$ anwenden muß. Um das Produkt $A \cdot \mathbf{w}$ zu bestimmen, bildet man zuerst $B^{\mathbf{T}} \cdot \mathbf{w}$ und multipliziert das Resultat dann mit B :

$$A \cdot \mathbf{w} = B \cdot (B^{\mathbf{T}} \cdot \mathbf{w}). \tag{2.17}$$

Ist die Gleichung 2.1 nun mit dem Lanczos Algorithmus gelöst, setzt man

$$\mathbf{z} = B^{\mathbf{T}}\mathbf{y}, \tag{2.18}$$

denn es gilt

$$B\mathbf{z} = BB^{\mathbf{T}}\mathbf{y} = A\mathbf{y} = \mathbf{b}. \tag{2.19}$$

Also hat man dadurch eine Lösung zu der Gleichung 2.13 gefunden.

2.3 Selbst-orthogonale Vektoren

Beschreibung :
Ein weiteres Problem des Lanczos Algorithmus in diesem Zusammenhang ist, daß in einem Vektorraum über einem kleinen endlichen Körper oft gilt :

$$\begin{aligned} \mathbf{w}_n^{\mathbf{T}} A \mathbf{w}_n &= 0 \text{ bzw.} \\ \mathbf{w}_n^{\mathbf{T}} \mathbf{w}_n &= 0, \text{ wobei } \mathbf{w}_n \neq 0. \end{aligned} \tag{2.20}$$

Dies verursacht natürlich Probleme, weil man sehr häufig durch diesen Term dividieren muß.

Bei der Anwendung des Algorithmus in $GF(2)$ kennt man nun die beiden folgenden Ansätze zur Lösung dieses Problems :

Einbetten :
In anderen Arbeiten (z.B. [3] oder [8]) versucht man dieses Problem dadurch zu umgehen, daß man $GF(2)$ in einen größeren Körper $GF(2^k)$ einbettet. Man erzeugt dabei zufällig eine Diagonalmatrix D mit Elementen aus $GF(2^k) - \{0\}$; anstatt nun $B\mathrm{z} = \mathrm{b}$ zu lösen, löst man

$$BD\mathrm{z} = \mathrm{b}. \tag{2.21}$$

Also wird der neue Wert von A

$$\tilde{A} = BD^2B^{\mathbf{T}}. \tag{2.22}$$

Dann ist die durchschnittliche Wahrscheinlichkeit, daß $w_i^{\mathbf{T}} w_i = 0$ für $w_i \neq 0$ etwa $\frac{1}{2^k}$ (vgl. [8]). Ist $2^k > N$, hat man gute Chancen, das Problem der Selbstorthogonalität nach wenigen zufälligen Versuchen zu umgehen. Aber der Aufwand, um die Arithmetik in dem endlichen Körper $GF(2^k)$ zu berechnen, ist sehr groß und verbraucht viel Zeit und Speicherplatz.

"Look Ahead"–Methode :
Im Falle des Lanczos Algorithmus über $\mathbb{R}$ wird ein ähnliches Problem durch Benutzung der "Look ahead Lanczos" Methode (ausführliche Beschreibung in [13]) gelöst. Diese Methode beachtet das Problem der Existenz von sehr kleinen inneren Produkten, da die Division durch sehr kleine Zahlen schlecht konditioniert ist, also größere numerische Fehler verursacht. Die Idee dieser "Look ahead Lanczos" Methode ist es, $\mathbf{w}_{n+1}$ aus mehreren Vektoren zu berechnen, also nicht nur aus $\mathbf{w}_n$, $\mathbf{v}_n$, $\mathbf{w}_{n-1}$, sondern auch aus den älteren Versionen $\mathbf{w}_{n-2}, \ldots, \mathbf{w}_{n-k}$, wobei k von der Anzahl der auftretenden Vektoren $\mathbf{w}_i$ abhängt, die numerische Instabilität verursachen können. Diese Idee lässt sich analog auch auf den Lanczos Algorithmus über $GF(2)$ übertragen (Beschreibung im folgenden Kapitel).

2.4 Packen von mehreren Vektoren

Motivation :
Der Lanczos Algorithmus über $GF(2)$ verlangt es, etwa N Matrix–Vektor Produkte $A \cdot w$ zu bilden (vgl. [1]). Jedes dieser Produkte erfordert T Indizierungsschritte, und jeder Indizierungsschritt arbeitet nur mit einem einzigen Bit. Also liegt es nahe, den Algorithmus so zu verändern, daß er etwa `WordSize` Vektor–Matrix Produkte zur gleichen Zeit berechnet.

Durchführung :
Dieses Ziel wird dadurch erreicht, daß man in einem Maschinenwort die Zeilen von `WordSize` Vektoren anstelle der Zeile von lediglich einem einzigen Vektor speichert. Dann kann jeder Indizierungsschritt mit `WordSize` Bits arbeiten. Man hat dadurch `WordSize` Vektoren zu einem Vektorblock zusammengefasst. An Stelle eines Matrix–Vektor Produktes wird jetzt ein Matrix–Vektorblock Produkt ausgeführt, d.h. es werden `WordSize` Matrix–Vektor Produkte in der gleichen Zeit berechnet. Diese Änderung erlaubt es nun $N/$`WordSize` Matrix–Vektor Produkte in der gleichen Zeit wie N Matrix–Vektor Produkte zu berechnen.
Durch Benutzung von Vektorblöcken anstelle von Vektoren ergibt sich auch, daß die oben erwähnten α und β als 'Skalar'produkte dieser Vektorblöcke keine Elemente aus $GF(2)$ mehr sind, sondern $k \times l$-Matrizen über $GF(2)$, wobei k und l die Anzahl der Vektoren in den an dem 'Skalar'produkt beteiligten Vektorblöcken sind; es gilt $k, l <$ `WordSize`. Die Spaltenvektoren dieser kleinen Matrizen werden dann mit einer analogen Methode zusammengefaßt.

Weitere Bezeichnungen :
Ein fett gedruckter Buchstabe, indiziert oder nicht, $\mathbf{w}$ oder $\mathbf{w}_n$, bezeichnet im folgenden einen Spaltenvektor aus $GF(2)^N$ oder einen Block von solchen Spaltenvektoren. Der entsprechend in Italic gesetzte Buchstabe w oder w_n bezeichnet im folgenden den Unterraum von $GF(2)^N$, der von diesen Vektoren aufgespannt wird, also $w = \mathrm{Span}(\mathbf{w})$.

Horizontale Konkatenation von Vektorblöcken wird in der folgenden Arbeit durch einen senkrechten Strich bezeichnet : $\mathbf{t} \mid \mathbf{u}$. Dies bedeutet : man packt die Vektorblöcke $\mathbf{t}$ und $\mathbf{u}$ zu einem einzigen Vektorblock zusammen. Dies verläuft analog zu obiger Methode (man kann umgekehrt auch sagen, ein Vektorblock sei die horizontale Konkatenation mehrerer Vektoren).

2.5 Preprocessing

Idee :
Die Idee des Preprocessing ist es, im homogenen Fall Zeilen und Spalten aus der Matrix zu löschen (und somit die Größe des Problems, den Speicher– und Zeitbedarf zu verringern), indem man ausnutzt, daß Teile der Lösung schon explizit durch die Struktur der Matrix gegeben sind.

Durchführung :
Man will ja das homogene Gleichungssystem

$$Bz = 0 \tag{2.23}$$

lösen.
Eine Zeile i von B enthalte ein einzelnes Element $\neq 0$, also ein 'Singleton' . Die Spalte j, die dieses 'Singleton' enthält, besitzt keine weiteren zur Lösung des Systems verwendbaren Informationen. Wenn $B\mathbf{z} = 0$ ist, dann muß das der Zeile i entsprechende Bit von $\mathbf{z}$ 0 sein. Da die Spalte j nur mit diesem Bit multipliziert wird, können die Zeile i und auch die Spalte j aus B gelöscht werden. Man speichert dabei nur die Nummer j der gelöschten Spalte, um die Lösung des ursprünglichen Systems wieder rekonstruieren zu können.
Man kann nun ebenso alle leeren Zeilen (d.h. Zeilen ohne Einträge $\neq 0$), und, mit analoger Argumentation, auch alle leeren Spalten aus der Matrix löschen. Man muß dabei wieder nur die Nummer j der gelöschten Spalten speichern, um die Lösung des ursprünglichen Systems wieder rekonstruieren zu können.
Dieser Prozess wird nun solange durchlaufen, bis die Matrix keine leeren Zeilen oder Spalten und keine Zeilen mit lediglich einem Eintrag $\neq 0$ enthält.

Dies wird normalerweise in einem der eigentlichen Lösung des Gleichungssystems vorangehenden Schritt getan, um die Berechnung zu beschleunigen und den Speicherplatzbedarf zu verringern. In Testfällen erwies es sich, daß man dabei ca. 10% der Zeilen und Spalten löschen kann. Dadurch verringern sich auch der Speicherbedarf und die Laufzeit entsprechend.

Für den hier vorgestellten Algorithmus ist es sogar notwendig, diese Löschungen vorzunehmen. Zur Begründung greife ich auf erst im folgenden Kapitel erfolgende Definitionen und Ausdrücken etwas vor :
Angenommen, Spalte j von B enthält zwei 'Singletons' , die den Zeilen i_1 und i_2 entsprechen, und angenommen, die Spalte j kann während des Preprocessing nicht gelöscht werden. Betrachtet man nun den Vektor $\mathbf{e}$ mit genau zwei Einsen an den Stellen i_1 und i_2 und sonst lauter Nullen. Man bemerkt, daß $B^{\mathbf{T}}\mathbf{e} = 0$. Also ist $\mathbf{e}$ orthogonal zu dem Bild von B. Dies bedeutet, wenn $\mathbf{e}$ jemals im Bereich von $\mathbf{w}_0, \ldots, \mathbf{w}_n$ berechnet wird, wird $\mathbf{e}$ ein Teil des Raumes r_n; dann wird die nächste Iteration, da $A\mathbf{e} = 0$ ist, die "Lösung" $\mathbf{e}$ zu $A\mathbf{y} = 0$ finden, die die triviale Lösung $\mathbf{z} = 0$ zu $B\mathbf{z} = 0$ liefert. Weiter wird $\mathbf{w}_{n+1}$ in seiner Größe vermindert, wobei man nur eine triviale Lösung gewonnen hat.

Beispiel 2.1
In einem Testlauf gab es 24 solcher Vektoren $\mathbf{e}$. Dies bedeutet, daß der Raum $\mathbf{r}_n$ zu groß für meine Routine wurde, und daß das Programm abbrach. Wenn ich das Programm mit einer kleineren Anfangsgröße von $\mathbf{w}_0$ (16 Spalten anstelle von 24), wieder laufen liess, terminierte es zwar, gab aber nur die triviale Lösung $\mathbf{z} = 0$ zu $B\mathbf{z} = 0$. Der Grund war, daß in den Lösungen zu

$$\begin{aligned} B\mathbf{z} &= 0, \text{ also} \\ A\mathbf{y} &= \tilde{\mathbf{b}} \text{ und } \mathrm{Span}(\tilde{\mathbf{b}}) \subseteq b \end{aligned} \tag{2.24}$$

(der Raum b war in diesem Fall irrelevant; es ergab sich $\tilde{\mathbf{b}} = 0 \subseteq b$) die Vektoren $\mathbf{y}$ alle Linearkombinationen des speziellen Vektors $\mathbf{e}$ waren, sodaß $\mathbf{z} = B^{\mathbf{T}}\mathbf{y} = 0$ galt. Als ich stattdessen in diesem speziellen Testfall die 'Singletons' durch Preprocessing entfernte, lief das Programm ohne Schwierigkeiten und lieferte 16 linear unabhängige Lösungen $\mathbf{z}$ der Gleichung 2.23.

Kapitel 3

Der Algorithmus

In diesem Kapitel beschreibe ich den in dieser Arbeit verwendeten Algorithmus. Dabei erläutere ich die Grundlagen und die theoretische Fundierung. Weiter beschreibe ich in diesem Kapitel, wie die Lösungen des Systems berechnet werden können, und gebe eine Analyse des Laufzeit– und Platzbedarfs des konkreten Algorithmus.
Dieses Kapitel ist unterteilt in fünf Abschnitte, in denen die folgenden Themen behandelt werden :

- Grundlagen des Algorithmus
- Erläuterungen zu den Grundlagen
- Terminieren und Konstruieren der Lösungen
- Zusammenfassung des Algorithmus und Reihenfolge der Berechnungen
- Laufzeit– und Platzbedarfanalyse

3.1 Grundlagen

Man will das lineare Gleichungssystem

$$A\mathbf{y} = \mathbf{b} \tag{3.1}$$

lösen, wobei A eine symmetrische $N \times N$–Matrix, $\mathbf{b} \in GF(2)^N$ ein Vektorblock $\neq 0$ und $\mathbf{y} \in GF(2)^N$ der zu bestimmende Vektorblock ist (zu Komplikationen mit asymmetrischen Matrizen vgl. Kap. 2.2).

Der hier behandelte spezielle Algorithmus entstand durch das Zusammenwirken dreier Ideen :

1. der Lanczos Algorithmus (vgl. Kap. 2.1)
2. Packen von Vektoren zu einem Vektorblock aus Effizienzgründen (siehe Kap. 2.4)
3. die "Look Ahead" Methode, um Problemen mit selbstorthogonalen Vektoren in $GF(2)^N$ begegnen zu können (vgl. Kap. 2.3)

Der Lanczos Algorithmus :
Wie in Kap. 2.1 schon erwähnt, ist es die Grundidee des Lanczos Algorithmus, bei jeder seiner Iterationen einen Vektor zu berechnen, der paarweise orthogonal zu allen in vorhergegangenen Iterationen bestimmten Versionen ist, und mit diesen Vektoren eine orthogonale Matrix von demselben Rang wie die Koeffizientenmatrix des zu lösenden Systems aufzubauen. Zu diesem Zweck werden, ähnlich wie im ursprünglichen Lanczos Algorithmus auch, die Vektorblöcke $\mathbf{w}_n$, $\mathbf{v}_{n+1}$ verwendet. Dabei wird $\mathbf{v}_{n+1}$ wieder durch Anwendung von A auf $\mathbf{w}_n$ erhalten, und $\mathbf{w}_{n+1}$ durch Subtraktion verschiedener Vektorblöcke von $\mathbf{v}_{n+1}$.

Packen von Vektoren :
Bei dem hier verwendeten Algorithmus arbeitet man aus Gründen der Effizienz (vgl. S.23) mit Vektorblöcken anstelle von Vektoren. Dies hat besseres Laufzeitverhalten und geringeren Speicherbedarf des Algorithmus zur Folge, führt jedoch auch zu Komplikationen, da das innere Produkt $\mathbf{u}^{\mathbf{T}}\mathbf{v}$ zweier Vektorblöcke $\mathbf{u}$ und $\mathbf{v}$, wobei $\mathtt{size}(\mathbf{u}) = k, \mathtt{size}(\mathbf{v}) = l$, jetzt eine $k \times l$–Matrix ergibt.

"Look Ahead" Methode :
Bei dem traditionellen Lanczos Algorithmus arbeitet man mit zwei Vektoren $\mathbf{w}_n$ und $\mathbf{v}_n$ und mit jeweils einer Version ihrer direkten Vorgänger.
Nun müssen wegen Komplikationen, die durch das Auftreten von verschwindenden inneren Produkten entstehen, mehrere Vektorblöcke verwendet werden : $\mathbf{w}_n$, $\mathbf{v}_n$, $\mathbf{x}_n$, $\mathbf{r}_n$, $\mathbf{d}_n$, $A^{\mathtt{inv}}(\mathbf{w}_n)$, $A^{\mathtt{inv}}(\mathbf{v}_n)$, $A^{\mathtt{inv}}(\mathbf{x}_n)$, $A^{\mathtt{inv}}(\mathbf{r}_n)$ und $A^{\mathtt{inv}}(\mathbf{d}_n)$.
Die Vektorblöcke $\mathbf{x}_n$ und $\mathbf{r}_n$ werden dabei eingeführt, um das Problem von selbstorthogonalen Vektoren zu übernehmen. Diese werden so aus $\mathbf{w}_n$ konstruiert, daß der Vektorblock $\mathbf{x}_n$ keine selbstorthogonalen Vektoren enthält, aber die $\mathbf{x}_i, 0 \leq i \leq n$ paarweise orthogonal sind. Damit erhält man die folgenden Bedingungen :

$$\begin{aligned} &\mathbf{x}_n \text{ ist regulär}, \\ &\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_i = 0 \quad 0 \leq i < n. \end{aligned} \tag{3.2}$$

Der Block $\mathbf{r}_n$ enthält die selbstorthogonalen Vektoren von $\mathbf{w}_n$. Er wird so konstruiert, daß immer

$$\begin{aligned} &\mathbf{r}_n^{\mathbf{T}}\mathbf{r}_n = 0, \\ &\mathbf{r}_n^{\mathbf{T}}\mathbf{x}_i = 0 \quad 0 \leq i < n. \end{aligned} \tag{3.3}$$

Die Blöcke $\mathbf{d}_{n-1}$ und $\mathbf{f}_{n-1}$ sind an der Berechnung von $\mathbf{w}_{n+1}$ aus $\mathbf{v}_n$ beteiligt. Sie werden so konstruiert, daß immer

$$\mathbf{w}_n^{\mathbf{T}}\mathbf{x}_i = 0 \quad 0 \leq i < n. \tag{3.4}$$

Im folgenden Abschnitt wird dabei auch eine Methode angegeben, die es ermöglicht, die explizite Berechnung und Speicherung des Vektorblocks $\mathbf{f}_{n-1}$ zu umgehen.
Die Blöcke $A^{\mathtt{inv}}(\mathbf{w}_n)$, $A^{\mathtt{inv}}(\mathbf{v}_n)$, $A^{\mathtt{inv}}(\mathbf{r}_n)$ und $A^{\mathtt{inv}}(\mathbf{d}_n)$ werden in Kap. 3.3 genauer beschrieben. Sie dienen zur Berechnung der Lösungen des Systems.

In jeder Iteration benötigt man dabei höchstens zwei Versionen jedes Vektorblocks (z.B. $\mathbf{v}_{n+1}$ und $\mathbf{v}_{n+2}$); es werden deswegen nur diese und keine älteren Versionen gespeichert.

3.2 Erläuterungen

Die Vektorblöcke $\mathbf{w}_{n+1}$ und $\mathbf{v}_{n+2}$:
Wie in Kap. 2.1 auch, setzt man

$$\mathbf{w}_0 = \mathbf{b}. \tag{3.5}$$

Man erhält $\mathbf{v}_{n+1}$ durch Anwendung von A auf $\mathbf{w}_n$:

$$\mathbf{v}_{n+1} = A\mathbf{w}_n \tag{3.6}$$

$\mathbf{w}_{n+1}$ schließlich berechnet sich aus $\mathbf{v}_{n+1}$, indem man einige andere Vektorblöcke subtrahiert :

$$\mathbf{w}_{n+1} = \mathbf{v}_{n+1} - \mathbf{x}_n\alpha_n - \mathbf{x}_{n-1}\beta_{n-1} - \mathbf{d}_{n-2}\gamma_{n-2}. \tag{3.7}$$

Dabei fordert man, daß :

$$\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{x}_i = 0, \qquad 0 \leq i \leq n, \tag{3.8}$$

um die Eigenschaft der Orthogonalität zu erhalten (die Blöcke $\mathbf{x}_n$, $\mathbf{d}_{n-2}$ und $\mathbf{f}_{n-2}$ werden später beschrieben).
Die Wahl von α_n und β_{n-1} wird durch die Forderung in Gleichung 3.8 bestimmt. Sie ist ähnlich zu der im ursprünglichen Lanczos Algorithmus, jedoch wird $\mathbf{x}_n$ anstelle von $\mathbf{w}_n$ benutzt :

$$\begin{aligned} \alpha_n &= (\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n)^{-1}(\mathbf{x}_n^{\mathbf{T}}\mathbf{v}_{n+1}), \\ \beta_{n-1} &= (\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1})^{-1}(\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{v}_{n+1}) \end{aligned}$$

Die Matrix γ_{n-2} wird definiert als :

$$\gamma_{n-2} = (\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2})^{-1}(\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1}). \tag{3.9}$$

Wegen der später beschriebenen speziellen Konstruktion von $\mathbf{f}_{n-2}$ und $\mathbf{d}_{n-2}$ ist $\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2} = I$. Weiter wird das innere Produkt $\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1}$ nicht direkt, sondern auf Umwegen über verschiedene proj-Operatoren bestimmt, um die explizite Berechnung und Speicherung des Vektorblocks $\mathbf{f}_{n-2}$ und seiner älteren Versionen zu sparen.

Die Vektorblöcke $\mathbf{r}_{n+1}$ und $\mathbf{x}_{n+1}$:
Die Blöcke $\mathbf{x}_n$ und $\mathbf{r}_n$ werden eingeführt, um das Problem der selbstorthogonalen Vektoren zu lösen. $\mathbf{x}_n$ genügt dabei der Forderung

$$\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n \quad \text{ist regulär, also invertierbar,} \tag{3.10}$$

und ist orthogonal zu all seinen vorhergegangenen Versionen :

$$\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_i = 0, \qquad 0 \leq i < n. \tag{3.11}$$

Wegen diesen Eigenschaften kann $\mathbf{x}_n$ den Vektorblock $\mathbf{w}_n$ in der Prozedur, die $\mathbf{w}_{n+1}$ aus $\mathbf{v}_{n+1}$ berechnet, ersetzen; hierbei werden dann alle Forderungen nach Orthogonalität bzw. Nichtorthogonalität bewahrt.
Zum Beweis der Eigenschaft $\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{x}_i = 0$, für $0 \leq i \leq n$ siehe [1], S. 9ff.

Der Block $\mathbf{r}_n$ ist orthogonal zu allen Blöcken, die bis jetzt aufgetreten sind; es muß gelten :

$$\begin{aligned} \mathbf{r}_n^{\mathbf{T}}\mathbf{w}_i &= 0, \quad 0 \leq i \leq n \\ \mathbf{r}_n^{\mathbf{T}}\mathbf{x}_i &= 0, \quad 0 \leq i \leq n \\ \mathbf{r}_n^{\mathbf{T}}\mathbf{r}_n &= 0 \end{aligned} \tag{3.12}$$

Also werden alle Probleme mit selbstorthogonalen Vektoren und verschwindenden inneren Produkten durch diese Eigenschaften in den Unterraum r_n verschoben.

Mit den in 3.12 gennanten Eigenschaften gilt für die von den Vektorblöcken aufgespannten Unterräume

$$v_0 \oplus \cdots \oplus v_n = w_0 \oplus \cdots \oplus w_n = x_0 \oplus \cdots \oplus x_n \oplus r_n \tag{3.13}$$

oder, um die im ersten Kapitel definierte Abkürzung zu benutzen,

$$V_n = W_n = X_n \oplus r_n. \tag{3.14}$$

Denn es gilt :

$$\begin{aligned} x_n \cap x_i &= \emptyset, \quad 0 \leq i < n \text{ wegen } \mathbf{x}_n^{\mathbf{T}}\mathbf{x}_i = 0, \quad 0 \leq i < n, \\ r_n \cap x_i &= \emptyset, \quad 0 \leq i \leq n \text{ wegen } \mathbf{r}_n^{\mathbf{T}}\mathbf{x}_i = 0, \quad 0 \leq i \leq n, \\ w_n \cap w_i &= \emptyset, \quad 0 \leq i < n \text{ wegen } \mathbf{w}_n^{\mathbf{T}}\mathbf{w}_i = 0, \quad 0 \leq i < n \end{aligned} \tag{3.15}$$

und $V_n = W_n$, da alle Vektorblöcke, die von $\mathbf{v}_n$ subtrahiert werden, um $\mathbf{w}_n$ zu erhalten, in X_n enthalten sind.

Man definiert r_n als den Unterraum von W_n, der orthogonal zu ganz W_n ist :

$$r_n = W_n \cap W_n^{\perp}. \tag{3.16}$$

x_n wird dann als ein beliebiges Komplement von r_n in $W_n \cap X_{n-1}^{\perp}$ definiert; es zeigt sich, daß jede so getroffene Wahl den Bedingungen für $\mathbf{x}_n$ genügt, denn wählt man Basen von x_n, r_n aus und bezeichnet sie als die Vektorblöcke $\mathbf{x}_n$, $\mathbf{r}_n$, dann gilt :

$$\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_i = 0 \quad , 0 \leq i < n, \tag{3.17}$$

da man ja weiß, daß $x_n \subseteq X_{n-1}^{\perp}$. Außerdem gilt wegen der Wahl von $\mathbf{x}_n$

$$r_n \oplus x_n = W_n \cap X_{n-1}^{\perp}. \tag{3.18}$$

Daraus folgt, daß $\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n$ regulär ist wegen 3.16 , wie gewünscht.

Die eigentliche Berechnung von (x_n, r_n) berücksichtigt nicht die früheren Versionen $x_0, x_1, \dots x_{n-1}$.
Es gilt

$$r_{n-1} \oplus w_n = W_n \cap X_{n-1}^{\perp}, \tag{3.19}$$

da $\mathbf{w}_n \perp \mathbf{r}_n$ und $\mathbf{r}_n \perp \mathbf{x}_{n-1}$. So ergibt sich r_n zu

$$r_n = (r_{n-1} \oplus w_n) \cap (r_{n-1} \oplus w_n)^{\perp}. \tag{3.20}$$

Also kann man in Analogie zu oben auch x_n als beliebiges Komplement zu r_n in $r_{n-1} \oplus w_n$ definieren. So erhält man die erwünschten Eigenschaften :

$$\begin{aligned}
r_{n-1} \oplus w_n &= x_n \oplus r_n \\
\mathbf{r}_n^{\mathbf{T}}\mathbf{x}_i &= 0, \quad 0 \leq i \leq n \\
\mathbf{r}_n^{\mathbf{T}}\mathbf{r}_n &= 0 \\
\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n & \quad \text{ist nicht singulär} \\
\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_i &= 0, \quad 0 \leq i \leq n
\end{aligned} \tag{3.21}$$

Will man nun durch

$$x_n \oplus r_n = r_{n-1} \oplus w_n \tag{3.22}$$

die Blöcke $\mathbf{x}_n$ und $\mathbf{r}_n$ tatsächlich berechnen, führt man eine Matrix τ_n ein mit :

$$\begin{aligned}
(\mathbf{x}_n \mid \mathbf{r}_n) &= (\mathbf{r}_{n-1} \mid \mathbf{w}_n)\tau_n \\
(\mathbf{r}_{n-1} \mid \mathbf{w}_n) &= (\mathbf{x}_n \mid \mathbf{r}_n){\tau_n}^{-1}
\end{aligned} \tag{3.23}$$

Dadurch verhindert man die Benutzung von früheren Versionen der $\mathbf{x}_i$ oder $\mathbf{w}_i$, die Aufspaltung wird dadurch effizienter.
Die Berechnung von τ_n wird in Kapitel 3.4 näher erläutert.

Die Vektorblöcke $\mathbf{d}_{n-2}$ und $\mathbf{f}_{n-2}$:
Der Raum d_{n-2} wird definiert als

$$d_{n-2} = X_{n-2} \cap \left(X_{n-2} \cap A^{-1}(X_{n-1})\right)^{\perp}, \tag{3.24}$$

und f_{n-2} ist ein Komplement von $X_{n-2} \cap A^{-1}X_{n-1}$ in X_{n-2}, erfüllt also folgende Bedingung :

$$f_{n-2} \oplus (X_{n-2} \cap A^{-1}(X_{n-1})) = X_{n-2} \tag{3.25}$$

Nun wählt man Basen von d_{n-2} und f_{n-2}, die die Vektorblöcke $\mathbf{d}_{n-2}$ und $\mathbf{f}_{n-2}$ bilden. Aufgrund der oben genannten Eigenschaften des inneren Produktes in X_{n-2} weiß man, daß d_{n-2} und f_{n-2} dieselbe Dimension haben, nicht größer sind als r_{n-1}, und außerdem, daß $\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2}$ nicht singulär ist, da $\mathbf{x}_{n-2}^{\mathbf{T}}\mathbf{x}_{n-2}$ regulär ist. Tatsächlich wird die Berechnung dieser beiden Vektorblöcke so arrangiert, daß

$$\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2} = I \tag{3.26}$$

gilt (vgl. Kapitel 3.3).
Diese Definitionen geben keine für die Praxis relevante Methode, um $\mathbf{f}_{n-2}$ oder $\mathbf{d}_{n-2}$ tatsächlich zu berechnen, da die Definitionen die großen Unterräume X_{n-2} und X_{n-1} enthalten. Diese großen Unterräume können nicht bei der Berechnung von $\mathbf{d}_{n-2}$ und $\mathbf{f}_{n-2}$ benutzt werden; sowohl Zeit– als auch Speicherverbrauch würden viel zu groß. Wie man den Vektorblock $\mathbf{d}_{n-2}$ effizient berechnen und die Berechnung des Vektorblocks $\mathbf{f}_{n_2}$ vermeiden kann, beschreiben die beiden folgenden Abschnitte.

Vermeidung der Berechnung von großen Unterräumen :
Um die Verwendung (und damit Speicherung) dieser großen Unterräume zu umgehen, definiert man zuerst einen sogenannten 'Projektions'operator.

Definition 3.1 ('Projektions'operator)
Seien $\mathbf{u}$, $\mathbf{w}_i, \mathbf{x}_i$, $\quad 0 \leq i \leq n$ *und* $\mathbf{r}_n$ *Vektorblöcke,* X_n *der von den* $\mathbf{x}_i$*,* W_n *der von den* $\mathbf{w}_i$ *und* r_n *der von* $\mathbf{r}_n$ *aufgespannte Raum.*
Ist bekannt, daß ein Vektorblock $\mathbf{u}$ *in* $W_n = r_n \oplus X_n$ *enthalten ist, definiert man*

$$\texttt{Proj}(\mathbf{r}_n; \mathbf{u})$$

als die Matrix, die $\mathbf{r}_n$ *auf die orthogonale Projektion von* $\mathbf{u}$ *auf* $\mathbf{r}_n$ *aus* X_n *abbildet. Also hat man die Eigenschaft*

$$\mathbf{u} - \mathbf{r}_n\texttt{Proj}(\mathbf{r}_n; \mathbf{u}) \subseteq X_n \tag{3.27}$$

Damit wird der Test, ob $\mathbf{u} \subseteq X_n$ ist durch den einfacheren Test, ob $\texttt{Proj}(\mathbf{r}_n; \mathbf{u}) = 0$ ist, ersetzt; denn gilt $\texttt{Proj}(\mathbf{r}_n; \mathbf{u}) = 0$, reduziert sich obige Gleichung zu $\mathbf{u} \subseteq X_n$.
Das Resultat diese Operators ist eine kleine Matrix. die $\texttt{size}(\mathbf{r}_n)$ Zeilen und $\texttt{size}(\mathbf{u})$ Spalten hat.

Zwei Teilmatrizen der bei der Berechnung von $\mathbf{x}_n$ und $\mathbf{r}_n$ eingeführten Matrix ${\tau_n}^{-1}$ sind :

$$\texttt{Proj}(\mathbf{r}_n; \mathbf{r}_{n-1}) \tag{3.28}$$

und

$$\texttt{Proj}(\mathbf{r}_n; \mathbf{w}_n) = \texttt{Proj}(\mathbf{r}_n; \mathbf{v}_n) = \texttt{Proj}(\mathbf{r}_n; A\mathbf{w}_{n-1}). \tag{3.29}$$

Die Gleichungen 3.28 und 3.29 gelten, weil $(\mathbf{x}_n \mid \mathbf{r}_n){\tau_n}^{-1} = (\mathbf{r}_{n-1} \mid \mathbf{w}_n)$, und außerdem $W_n = V_n$, da alle Terme, die von $\mathbf{v}_n$ subtrahiert werden, um $\mathbf{w}_n$ zu erhalten, wie bekannt in X_n enthalten sind.
Die Matrizen, die sich als Resultat dieser Operatoren ergeben, werden in der Praxis aus Gründen der Effizienz nur als Teilmatrizen von ${\tau_n}^{-1}$ berechnet.

Für einen beliebigen Vektorblock $\mathbf{u} \subseteq W_{n-1}$ gilt zusätzlich

$$\texttt{Proj}(\mathbf{r}_n; \mathbf{u}) = \texttt{Proj}(\mathbf{r}_n; \mathbf{r}_{n-1})\texttt{Proj}(\mathbf{r}_{n-1}; \mathbf{u}). \tag{3.30}$$

wegen Gleichung 3.27 und der Tatsache, daß $W_{n-1} \subseteq W_n$.
Man benützt nun τ_{n-1}, um die Projektionen $\texttt{Proj}(\mathbf{r}_n; A\mathbf{x}_{n-1})$ und $\texttt{Proj}(\mathbf{r}_n; A\mathbf{r}_{n-1})$ zu erhalten, denn es gilt

$$\begin{aligned}
\texttt{Proj}(\mathbf{r}_n; (A\mathbf{x}_{n-1} \mid A\mathbf{r}_{n-1})) &= \texttt{Proj}(\mathbf{r}_n; (A\mathbf{r}_{n-2} \mid A\mathbf{w}_{n-1}))\tau_{n-1} \\
&= \texttt{Proj}(\mathbf{r}_n; \mathbf{r}_{n-1})\texttt{Proj}(\mathbf{r}_{n-1}; (A\mathbf{r}_{n-2} \mid A\mathbf{w}_{n-1}))\tau_{n-1}
\end{aligned}$$

da $(\mathbf{x}_{n-1} \mid \mathbf{r}_{n-1}) = (\mathbf{r}_{n-2} \mid \mathbf{w}_{n-1})\tau_{n-1}$. Diese Methode, die Matrizen $\texttt{Proj}(\mathbf{r}_n; A\mathbf{x}_{n-1})$ und $\texttt{Proj}(\mathbf{r}_n; A\mathbf{r}_{n-1})$ aus $\texttt{Proj}(\mathbf{r}_{n-1}; A\mathbf{r}_{n-2})$, τ_{n-1} und τ_n zu berechnen, setzt nur Operationen mit kleinen Matrizen ein. Keine großen Vektorblöcke werden benutzt, der 'Projektions'operator wird nie explizit aus den Vektorblöcken berechnet.

Der Raum c_{n-2} :
Man definiert den Raum c_{n-2} (der niemals tatsächlich berechnet oder gespeichert wird), aus Gründen der Übersicht als :

$$C_{n-2} = X_{n-2} \cap A^{-1}(X_{n-1}). \tag{3.31}$$

Damit gilt

$$C_{n-2} \oplus f_{n-2} = X_{n-2} \text{ (vgl. Definition von } f_{n-2})$$

und

$$\begin{aligned}
C_{n-1} &= X_{n-1} \cap A^{-1}(X_n) \\
&= [X_{n-2} \oplus x_{n-1}] \cap A^{-1}(X_n) \\
&= [C_{n-2} \oplus f_{n-2} \oplus x_{n-1}] \cap A^{-1}(X_n) \\
&= C_{n-2} \oplus [f_{n-2} \oplus x_{n-1} \cap A^{-1}(X_n)]
\end{aligned} \tag{3.32}$$

denn es ist $C_{n-2} \subseteq A^{-1}(X_{n-1}) \subseteq A^{-1}(X_n)$.
Man definiert nun c_{n-1} als den neuen Teil von C_{n-1} :

$$C_{n-1} = C_{n-2} \oplus c_{n-1}$$

also ist

$$c_{n-1} \;=\; (f_{n-2} \oplus x_{n-1}) \cap A^{-1}(X_n). \tag{3.33}$$

Nach Multiplikation mit A gilt außerdem

$$Ac_{n-1} = (Af_{n-2} \oplus Ax_{n-1}) \cap (X_n). \tag{3.34}$$

Man wählt eine Basis von c_{n-1} und läßt diese den Vektorblock $\mathbf{c}_{n-1}$ bilden. Nun führt man eine Matrix σ_{n-1} ein, die durch folgende Eigenschaft definiert wird :

$$\mathbf{c}_{n-1} = (\mathbf{f}_{n-2} \mid \mathbf{x}_{n-1})\sigma_{n-1}. \tag{3.35}$$

Es ist $Ac_{n-1} \subseteq X_n$, und deswegen $\mathtt{Proj}(\mathbf{r}_n; A\mathbf{c}_{n-1}) = 0$; tatsächlich ist Ac_{n-1} der größte Unterraum von $Af_{n-2} \oplus Ax_{n-1}$, dessen Projektion auf $\mathbf{r}_n = 0$ ist (vgl. Definition von c_{n-1}).
Dies zeigt eine effiziente Methode, wie man σ_{n-1} berechnen kann; es ist die größte Matrix mit vollem Rang, die der Bedingung

$$\mathtt{Proj}(\mathbf{r}_n; (A\mathbf{f}_{n-2} \mid A\mathbf{x}_{n-1}))\sigma_{n-1} = 0. \tag{3.36}$$

genügt. Jeder Unterraum von $f_{n-2} \oplus x_{n-1}$, der komplementär zu c_{n-1} ist, kann als f_{n-1} benutzt werden. Mit $c_{n-1} \oplus f_{n-1} = f_{n-2} \oplus x_{n-1}$ gilt dann

$$f_{n-1} \oplus (X_{n-1} \cap A^{-1}(X_n)) = X_{n-1}. \tag{3.37}$$

Unter Verwendung von τ_n und σ_n kann man unter Benutzung der Gleichungen 3.35 – 3.37 $\mathtt{Proj}(\mathbf{r}_n; A\mathbf{f}_{n-1})$ aus $\mathtt{Proj}(\mathbf{r}_n; A\mathbf{f}_{n-2})$ und $\mathtt{Proj}(\mathbf{r}_n; A\mathbf{x}_{n-1})$ berechnen, ohne Vektorblöcke benutzen zu müssen.

Der Berechnung des Blockes $\mathbf{d}_{n-1}$ lässt sich also wie folgt vereinfachen :

$$\begin{aligned} d_{n-2} &= X_{n-2} \cap {C_{n-2}}^{\perp} \\ d_{n-1} &= (d_{n-2} \oplus x_{n-1}) \cap c_{n-1}^{\perp}. \end{aligned} \tag{3.38}$$

Denn es gilt, daß $C_{n-2} \subseteq X_{n-2}$ nach Gleichung 3.31, sodaß $\mathbf{x}_{n-1} \subseteq C_{n-2}^{\perp}$.

Man berechnet

$$\begin{aligned} \mathbf{c}_{n-1}^{\mathbf{T}}(\mathbf{d}_{n-2} \mid \mathbf{x}_{n-1}) &= \sigma_{n-1}^{\mathbf{T}} \begin{pmatrix} \mathbf{f}_{n-2}^{\mathbf{T}} \\ \mathbf{x}_{n-1}^{\mathbf{T}} \end{pmatrix} (\mathbf{d}_{n-2} \mid \mathbf{x}_{n-1}) \\ &= \sigma_{n-1}^{\mathbf{T}} \begin{pmatrix} \mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2} & 0 \\ 0 & \mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1} \end{pmatrix}. \end{aligned} \tag{3.39}$$

Gilt nun

$$\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{d}_{n-2} = I \tag{3.40}$$

so vereinfacht sich Gleichung 3.39 zu

$$\mathbf{c}_{n-1}^{\mathbf{T}}(\mathbf{d}_{n-2} \mid \mathbf{x}_{n-1}) = \sigma_{n-1}^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1} \end{pmatrix} \tag{3.41}$$

Sei ρ_{n-1} die größte Matrix mit vollem Rang, sodaß

$$\mathbf{c}_{n-1}^{\mathbf{T}}(\mathbf{d}_{n-2} \mid \mathbf{x}_{n-1})\rho_{n-1} = 0. \tag{3.42}$$

Dann gilt wegen Gleichung 3.38

$$\mathbf{d}_{n-1} = (\mathbf{d}_{n-2} \mid \mathbf{x}_{n-1})\rho_{n-1}. \tag{3.43}$$

Mit dieser Wahl ist $\mathbf{f}_{n-1}^{\mathbf{T}}\mathbf{d}_{n-1}$ regulär, und durch sorgfältige Wahl von ρ_{n-1} erreicht man, daß

$$\mathbf{f}_{n-1}^{\mathbf{T}}\mathbf{d}_{n-1} = I. \tag{3.44}$$

(vgl. S.43).

So hat man $\mathbf{d}_{n-1}$ und $\mathtt{Proj}(\mathbf{r}_n; A\mathbf{f}_{n-1})$ berechnet, ohne die großen Unterräume X_n und C_n zu benutzen. Der Vektorblock $\mathbf{f}_{n-1}$ wurde überhaupt nicht berechnet. Tatsächlich wird $\mathbf{f}_{n-1}$ auch nur einmal benutzt, nämlich zur Berechnung von $\mathbf{w}_n$ in dem inneren Produkt

$$\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1}. \tag{3.45}$$

Dieses wird nun einfacher berechnet als

$$\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1} = (\mathtt{Proj}(\mathbf{r}_{n-1}; A\mathbf{f}_{n-2}))^{\mathbf{T}}(\mathbf{r}_{n-1}^{\mathbf{T}}\mathbf{w}_n), \tag{3.46}$$

dem Produkt von zwei kleinen Matrizen. Man kann so verfahren, da

$$A\mathbf{f}_{n-2} = \mathbf{r}_{n-1}\mathtt{Proj}(\mathbf{r}_{n-1}; a\mathbf{f}_{n-2}) + \mathbf{x}_{n-1}\kappa_n \tag{3.47}$$

für eine Matrix κ_n, sodaß

$$\begin{aligned} \mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1} &= \mathbf{f}_{n-2}^{\mathbf{T}}A\mathbf{w}_n = (A\mathbf{f}_{n-2})^{\mathbf{T}}\mathbf{w}_n \\ &= (\mathtt{Proj}(\mathbf{r}_{n-1}; a\mathbf{f}_{n-2}))^{\mathbf{T}}(\mathbf{r}_{n-1}^{\mathbf{T}}\mathbf{w}_n) + \kappa_n^{\mathbf{T}}\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{w}_n. \end{aligned} \tag{3.48}$$

Der zweite Term verschwindet dabei wegen $\mathbf{w}_n^{\mathbf{T}}\mathbf{x}_{n-1} = 0$.

Während der praktischen Berechnung werden weder die Vektorblöcke $\mathbf{f}_n$ und $\mathbf{c}_n$, noch die großen Unterräume W_n, X_n und C_n benutzt. Dies konnte durch die Einführung des Projektionsoperators, seiner Berechnung nur durch Operationen mit Matrizen und die Einführung von zusätzlichen Matrizen, die relativ schnell und effizient ohne Benutzung von viel Speicherplatz berechenbar sind, vermieden werden.

3.3 Berechnung der Lösungen

Definition 3.2 ('Inversions'operator)
Sei $\mathbf{u}$ *ein Vektorblock. Dann ist* $A^{\mathtt{inv}}(\mathbf{u})$ *ein Vektorblock, der durch folgende Eigenschaft bestimmt wird :*

$$\mathrm{Span}(\mathbf{u} - A \cdot A^{\mathtt{inv}}(\mathbf{u})) \subseteq b. \tag{3.49}$$

Berechnung des Operators:
Um $A^{\mathtt{inv}}(\mathbf{u})$ für die auftretenden Vektorblöcke zu bestimmen, setzt man

$$\begin{aligned} A^{\mathtt{inv}}(\mathbf{v}_0) &= 0 \\ A^{\mathtt{inv}}(\mathbf{v}_{n+1}) &= \mathbf{w}_n \qquad n \geq, 0 \end{aligned} \tag{3.50}$$

um die in Gleichung 3.49 geforderte Eigenschaft zu erhalten, Wenn nun in folgenden Schritten die Blöcke $\mathbf{x}_n$,$\mathbf{w}_n$,$\mathbf{r}_n$ oder $\mathbf{d}_n$ berechnet werden, werden auf die gleiche Art auch $A^{\mathtt{inv}}(\mathbf{x}_n)$, $A^{\mathtt{inv}}(\mathbf{w}_n)$, $A^{\mathtt{inv}}(\mathbf{r}_n)$, $A^{\mathtt{inv}}(\mathbf{d}_n)$ bestimmt :

$$\begin{aligned} A^{\mathtt{inv}}(\mathbf{w}_{n+1}) &= A^{\mathtt{inv}}(\mathbf{v}_{n+1}) - A^{\mathtt{inv}}(\mathbf{x}_n)\alpha_n - A^{\mathtt{inv}}(\mathbf{x}_{n-1})\beta_{n-1} \\ &\quad - A^{\mathtt{inv}}(\mathbf{d}_{n-2})\gamma_{n-2} \\ (A^{\mathtt{inv}}(\mathbf{x}_{n+1}) \mid A^{\mathtt{inv}}(\mathbf{r}_{n+1})) &= (A^{\mathtt{inv}}(\mathbf{r}_n) \mid A^{\mathtt{inv}}(\mathbf{w}_{n+1}))\tau_{n+1} \\ A^{\mathtt{inv}}(\mathbf{d}_n) &= (A^{\mathtt{inv}}(\mathbf{d}_{n-1}) \mid A^{\mathtt{inv}}(\mathbf{x}_n))\rho_n. \end{aligned} \tag{3.51}$$

Diese Vektorblöcke sind wichtig, um die Lösungen des Gleichungssystems zu berechnen.

Idee :
Die Eigenschaft der Orthogonalität bedingt es, daß gegen Ende des Algorithmus die Vektoren in $\mathbf{r}_n$ und $\mathbf{w}_n$ linear abhängig sind (vgl. [10] oder [1]). Dies ist auch das einzige Mal, daß $r_n \cap w_n \neq \emptyset$.
In diesem Falle kann man eine Matrix $\psi_n \neq 0$ berechnen, für die gilt :

$$\mathbf{r}_n \psi_n = 0. \tag{3.52}$$

Durch Anwendung von ψ_n auf $A^{\mathtt{inv}}(\mathbf{r}_n)$ erhält man nun wegen Gleichung 3.49 Lösungen des Systems 3.1.

Durchführung :
Hat man eine nichttriviale Linearkombination ψ_n der Vektoren in $\mathbf{r}_n$ gefunden, für die Gleichung 3.52 gilt, so ergibt dieselbe Linearkombination von $A^{\mathtt{inv}}(\mathbf{r}_n)$ einen Vektor $\mathbf{y}$ mit

$$\mathbf{y} = A^{\mathtt{inv}}(\mathbf{r}_n)\psi_n \tag{3.53}$$

sodaß

$$\mathrm{Span}(A\mathbf{y}) \subseteq b \tag{3.54}$$

wegen

$$\mathrm{Span}(\mathbf{r}_n - A \cdot A^{\mathtt{inv}}(\mathbf{r}_n)) \subseteq b. \tag{3.55}$$

Außerdem ist $\mathbf{y} \neq 0$, denn : $\mathbf{y}$ ist eine nichttriviale Linearkombination von $A^{\mathtt{inv}}(\mathbf{w}_n)$ und $A^{\mathtt{inv}}(\mathbf{r}_{n-1})$ wegen

$$(A^{\mathtt{inv}}(\mathbf{x}_n) \mid A^{\mathtt{inv}}(\mathbf{r}_n)) = (A^{\mathtt{inv}}(\mathbf{r}_{n-1}) \mid A^{\mathtt{inv}}(\mathbf{w}_n))\tau_n. \tag{3.56}$$

Weiterhin wird ja $A^{\mathtt{inv}}(\mathbf{w}_n)$ aus $A^{\mathtt{inv}}(\mathbf{v}_n) = \mathbf{w}_{n-1}$ berechnet, und die Ausführungen auf Seite 35 ergeben, daß die Vektorblöcke $\mathbf{w}_0, \ldots, \mathbf{w}_{n-1}$ linear unabhängig sind. Weiter folgt aus der Formel 3.56, daß $A^{\mathtt{inv}}(\mathbf{r}_{n-1}) \subseteq W_{n-1}$. Daraus ergibt sich, daß $\mathbf{y} \neq 0$.

Wegen obigen Ausführungen kann man sicher sein, Lösungen mit $A\mathbf{y} \subseteq \mathbf{b}$ zu finden; tatsächlich findet man auch bis zu `WordSize` linear unabhängige Lösungen. Hierbei ist es allerdings möglich, daß diese Lösungen $\mathbf{y}$ von der Form $A\mathbf{y} = 0$ sind, also nichttriviale Linearkombinationen der Spaltenvektoren in A bilden. In diesem Fall war A singulär.

Nun betrachtet man den Fall, daß A nicht quadratisch, also

$$A = BB^{\mathbf{T}}. \tag{3.57}$$

Ist nun

$$A\mathbf{y} = \tilde{\mathbf{b}} \text{ und } \tilde{b} \subseteq b \tag{3.58}$$

sieht man, daß

$$B(B^{\mathbf{T}}\mathbf{y}) = \tilde{\mathbf{b}} \tag{3.59}$$

und

$$\tilde{\mathbf{b}} = \mathbf{b}\lambda \tag{3.60}$$

für eine bestimmte kleine Matrix λ, und

$$\hat{B}\begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix} = (B \mid \mathbf{b})\begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix} = \tilde{\mathbf{b}} - \tilde{\mathbf{b}} = 0. \tag{3.61}$$

im Falle einer ursprünglich homogenen Gleichung.
Definiert man nun

$$\tilde{\mathbf{z}} = \begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix}, \tag{3.62}$$

hat man damit eine Lösung zu dem ursprünglichen homogenen Gleichungssystem

$$\hat{B}\hat{\mathbf{z}} = 0 \tag{3.63}$$

gefunden.

Terminieren :
Nachdem mit Hilfe der Matrix τ_n durch

$$(\mathbf{x}_n \mid \mathbf{r}_n) = (\mathbf{r}_{n-1} \mid \mathbf{w}_n)\tau_n \tag{3.64}$$

der Vektorblock $\mathbf{r}_n$ berechnet wurde, testet man, ob die Vektoren in $\mathbf{r}_n$ linear abhängig sind. Da man weiß, daß die Vektoren in $\mathbf{r}_{n-1}$ linear unabhängig sind, weiß man aus Formel 3.64 auch, daß die lineare Abhängigkeit allein von den Vektoren in $\mathbf{w}_n$ hervorgerufen wurde, eben wegen Gleichung 3.64.
Aus der Forderung, daß das innere Produkt $\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n$ regulär ist, kann man schließen, daß die Vektoren in $\mathbf{x}_n$ linear unabhängig sind. Außerdem gilt, daß auch die Vektoren in $\mathbf{v}_{n+1}$ ebenfalls linear abhängig sind, da dieser Vektorblock schon vorher durch Anwendung von A auf $\mathbf{w}_n$ berechnet wurde.
Aus diesen Gründen löscht man nach Entdeckung einer linearen Abhängigkeit redundante Vektoren aus $\mathbf{w}_n$ und $\mathbf{v}_{n+1}$ und berechnet den Vektorblock $\mathbf{r}_n$ mit Gleichung 3.64 neu. Ebenso löscht man dieselben Vektoren auch aus $A^{\mathtt{inv}}(\mathbf{w}_n)$ und $A^{\mathtt{inv}}(\mathbf{v}_{n+1})$, um die Eigenschaft aus Gleichung 3.49 zu erhalten. Nun sind die Vektoren in $\mathbf{r}_n$, $\mathbf{w}_n$ und $\mathbf{v}_{n+1}$ wieder linear unabhängig, und es ist $r_{n-1} \cap w_n = \emptyset$.

Das Löschen von Vektoren aus $\mathbf{w}_n$ bewirkt, daß in den folgenden Iterationen die Größe von $\mathbf{w}_{n\prime}$ für $n\prime \geq n$ geringer ist (und bleibt) als die von $\mathbf{w}_n$.
Hat man nun die Größe von $\mathbf{w}_n$ auf 0 reduziert, wobei $n \approx N$ wegen der Forderung der Orthogonalität an verschiedene Vektorblöcke, hält der Algorithmus. Auf Grund dieser vorher schon oft dargestellten Forderung wird der Algorithmus in jedem Fall für ein $n \leq N$ diesen Punkt erreichen.

3.4 Zusammenfassung des Algorithmus

In dieser Zusammenfassung des Algorithmus wird eine genauere Darstellung der tatsächlichen Berechnung von Vektorblöcken und Matrizen in der Praxis gegeben(zu der Darstellung der einzelnen Operationen vergleiche Kap.6 auf S.60 ff.).

Initialisierung
Setze

$$A^{\mathtt{inv}}(\mathbf{v}_0) = 0. \tag{3.65}$$

Jeder Vektorblock mit negativem Index($\mathbf{x}_{-1}$) ist leer(Dimension $N \times 0$), und jede Matrix mit negativen Index(γ_{-1}) hat Dimension 0×0 (dies wird bei der Speicherplatzverwaltung ausgeführt und ist nur wichtig für die ersten Iterationen).

Schritt 1 :

$$\mathbf{v}_0 = b. \tag{3.66}$$

Hier wird ein Vektorblock kopiert.

Für $n = -1, 0, 1, \ldots$ bis zum Ende(etwa $n = N/\texttt{WordSize}$) führt man die folgenden Schritte aus.
Dieses Ende ist erreicht, wenn $\mathbf{w}_{n+1}$ leer ist.

Schritt 2 :

$$\alpha_n = \left(\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n\right)^{-1}(\mathbf{x}_n^{\mathbf{T}}\mathbf{v}_{n+1}), \tag{3.67}$$

Ywobei $\xi_n^{-1} = \left(\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n\right)^{-1}$ in einer vorangegangenen Iteration berechnet wurde.
Dieser Schritt führt das innere Produkt zweier Vektorblöcke

$$\mathbf{x}_n^{\mathbf{T}}\mathbf{v}_{n+1} \tag{3.68}$$

ein. Außerdem werden hier zwei kleine Matrizen miteinander multipliziert.

Schritt 3 :

$$\beta_{n-1} = \left(\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1}\right)^{-1}(\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{v}_{n+1}), \tag{3.69}$$

wobei wieder $(\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1})^{-1}$ vorher berechnet wurde. Der größere Berechnungsschritt hier ist die Berechnung des inneren Produktes

$$\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{v}_{n+1}. \tag{3.70}$$

Weiter werden wie im letzten Schritt auch zwei kleine Matrizen miteinander multipliziert.

Schritt 4 :

$$\gamma_{n-2} = (\texttt{Proj}(\mathbf{r}_{n-1}; A\mathbf{f}_{n-2}))^{\mathbf{T}}(\mathbf{r}_{n-1}^{\mathbf{T}}\mathbf{w}_n). \tag{3.71}$$

Dies ist lediglich das Produkt von zwei kleinen Matrizen, die beide schon in einer vorangegangenen Iteration berechnet wurden, und das Transponieren des Resultates. Damit läßt sich γ_{n-2} wesentlich effizienter berechnen als durch Einführung des Vektorblocks $\mathbf{f}_{n-2}$ und Berechnung des inneren Produktes $\mathbf{f}_{n-2}^{\mathbf{T}}\mathbf{v}_{n+1}$.

Schritt 5 :

$$\mathbf{w}_{n+1} = \mathbf{v}_{n+1} - \mathbf{x}_n\alpha_n - \mathbf{x}_{n-1}\beta_{n-1} - \mathbf{d}_{n-2}\gamma_{n-2}. \tag{3.72}$$

Dieser Schritt führt drei Produkte von der Form Vektorblock mal kleine Matrix und drei Subtraktionen von Vektorblöcken aus.

Schritt 6 :

$$A^{\mathtt{inv}}(\mathbf{w}_{n+1}) = A^{\mathtt{inv}}(\mathbf{v}_{n+1}) - A^{\mathtt{inv}}(\mathbf{x}_n)\alpha_n - A^{\mathtt{inv}}(\mathbf{x}_{n-1})\beta_{n-1} - A^{\mathtt{inv}}(\mathbf{d}_{n-2})\gamma_{n-2}. \tag{3.73}$$

Wie im letzten Schritt führt man hier drei Produkte von der Form Vektorblock mal kleine Matrix und drei Subtraktionen von Vektorblöcken aus.
Dieser Schritt wird ausgeführt, um die Eigenschaften des $A^{\mathtt{inv}}$–Operators während den Berechnungen zu garantieren.

Schritt 7 :

$$\mathbf{v}_{n+2} = A\mathbf{w}_{n+1}. \tag{3.74}$$

Dies ist die Anwendung der schwachbesetzten Matrix A auf einen Vektorblock (von einer Größe von weniger als `WordSize`). Wie in Kap. 2.2 diskutiert, bedeutet dies im asymmetrischen Fall die Anwendung von $B^{\mathbf{T}}$, gefolgt von der Anwendung von B.

Schritt 8 :

$$A^{\mathtt{inv}}(\mathbf{v}_{n+2}) = \mathbf{w}_{n+1}. \tag{3.75}$$

Hier wird ein Vektorblock kopiert. Dieser Schritt ist wesentlich, um die Eigenschaften des $A^{\mathtt{inv}}$–Operators während den folgenden Berechnungen zu garantieren.

Schritt 9 :
Bilde die beiden inneren Produkte

$$\begin{gathered}\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{r}_n \\ \mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{w}_{n+1}.\end{gathered} \tag{3.76}$$

Dies sind die inneren Produkte zweier Vektorblöcke. Sie werden im nächsten Schritt bzw. in Schritt (4) der folgenden Iteration benutzt.

Schritt 10 :

$$\eta_{n+1} = (\mathbf{r}_n \mid \mathbf{w}_{n+1})^{\mathbf{T}} (\mathbf{r}_n \mid \mathbf{w}_{n+1}) = \begin{pmatrix} 0 & (\mathbf{w}_{n+1}^{\mathbf{T}} \mathbf{r}_n)^{\mathbf{T}} \\ (\mathbf{w}_{n+1}^{\mathbf{T}} \mathbf{r}_n) & (\mathbf{w}_{n+1}^{\mathbf{T}} \mathbf{w}_{n+1}) \end{pmatrix} \tag{3.77}$$

Hier wird eine im vorigen Schritt berechnete kleine Matrix transponiert und das Resultat mit ebenfalls im vorigen Schritt berechneten kleinen Matrizen zusammengesetzt.
Die Matrix η_{n+1} ist eine Hilfsmatrix, die zur Berechnung der Matrix τ_{n+1} und damit mittelbar zur Berechnung von $\mathbf{x}_{n+1}$ und $\mathbf{r}_{n+1}$ eingeführt wurde. η_{n+1} ist eine symmetrische Matrix mit $\mathtt{size}(\mathbf{r}_n) + \mathtt{size}(\mathbf{w}_{n+1})$ Zeilen und Spalten. Außerdem wird durch $rang(\eta_{n+1})$ die Zahl derjenigen Vektoren von $(\mathbf{r}_n \mid \mathbf{w}_{n+1})$ angegeben, die nicht zu sich selbst orthogonal sind.

Schritt 11 :
Berechne die Matrix τ_{n+1}, mit

$$\eta_{n+1}\tau_{n+1} = (\ \delta_{n+1} \quad 0\ ) \tag{3.78}$$

wobei δ_{n+1} eine Matrix mit $rang(\eta_{n+1})$ linear unabhängigen Spaltenvektoren ist. Weiterhin muß gelten, daß τ_{n+1} regulär ist. Dies sind lediglich Operationen auf einer kleinen Matrix. Man berechnet zuerst eine Basis des Lösungsraums der homogenen Gleichung $\eta_{n+1}x = 0$, lässt diese die rechten Spalten von τ_{n+1} bilden, und berechnet dann eine Basis aus Einheitsvektoren des von den Spaltenvektoren von η_{n+1} aufgespannten Raumes, die die linken Spalten von τ_{n+1} bilden.
τ_{n+1} ist regulär und hat dieselbe Anzahl Spalten und Zeilen wie η_{n+1}. Deswegen gilt

$$\tau_{n+1}^{\mathbf{T}}\eta_{n+1}\tau_{n+1} = \begin{pmatrix} \xi_{n+1} & 0 \\ 0 & 0 \end{pmatrix} \tag{3.79}$$

wobei

$$\xi_{n+1} = \mathbf{x}_{n+1}^{\mathbf{T}}\mathbf{x}_{n+1}. \tag{3.80}$$

Beachte : $\mathbf{x}_{n+1}$ wird erst im nächsten Schritt bestimmt. Mit Gleichung 3.80 kann man lediglich die richtige Berechnung von τ_{n+1} und $\mathbf{x}_{n+1}$ verifizieren.

Schritt 12 :
Unter Benutzung von τ_{n+1} berechne die Räume $\mathbf{x}_{n+1}, \mathbf{r}_{n+1}$ durch

$$(\mathbf{x}_{n+1} \mid \mathbf{r}_{n+1}) = (\mathbf{r}_n \mid \mathbf{w}_{n+1})\tau_{n+1}. \tag{3.81}$$

Hier haben wir zwei Vektorblöcke miteinander konkateniert, einen Vektorblock mit einer kleinen Matrix multipliziert, und den resultierenden Vektorblock in zwei kleinere Vektorblöcke aufgespalten.
Durch diesen Schritt wird die maximale Größe der auftretenden Vektorblöcke bestimmt. Es gilt :

$$\mathtt{size}(\mathbf{x}_{n+1}) + \mathtt{size}(\mathbf{r}_{n+1}) \;=\; \mathtt{size}(\mathbf{r}_n) + \mathtt{size}(\mathbf{w}_{n+1})$$

$$
\begin{aligned}
&= \; rang(\tau_{n+1}) \\
&\leq \; \mathtt{size}(\mathbf{r}_n) + \mathtt{size}(\mathbf{w}_0)
\end{aligned} \tag{3.82}
$$

Wenn aber jemals im Laufe der Berechnung gilt

$$\mathtt{size}(\mathbf{r}_n) + \mathtt{size}(\mathbf{w}_{n+1}) > \mathtt{WordSize}, \tag{3.83}$$

dann hält der Algorithmus.

Schritt 13 :
Berechne die Räume $A^{\mathtt{inv}}(\mathbf{x}_{n+1}), A^{\mathtt{inv}}(\mathbf{r}_{n+1})$ durch

$$(A^{\mathtt{inv}}(\mathbf{x}_{n+1}) \mid A^{\mathtt{inv}}(\mathbf{r}_{n+1})) = (A^{\mathtt{inv}}(\mathbf{r}_n) \mid A^{\mathtt{inv}}(\mathbf{w}_{n+1}))\tau_{n+1}. \tag{3.84}$$

Hier haben wir wieder wie im vorigen Schritt zwei Vektorblöcke konkateniert, einen Vektorblock mit einer kleinen Matrix multipliziert, und den resultierenden Vektorblock in zwei kleinere Blöcke aufgespalten.
Dieser Schritt wird ausgeführt, um die Eigenschaften des $A^{\mathtt{inv}}$–Operators während den Berechnungen zu garantieren. Weiterhin ist dieser Schritt wesentlich, um später eine Lösung des Gleichungssystems berechnen zu können.

Schritt 14 :
Wir berechnen das Inverse

$$(\mathbf{x}_{n+1}^{\mathbf{T}}\mathbf{x}_{n+1})^{-1} = \xi_{n+1}^{-1}, \tag{3.85}$$

indem wir das innere Produkt zweier Vektorblöcke bilden und die entstehende kleine Matrix invertieren. Das Ergebnis dieser Operationen wird in Schritt (2) und (3) der beiden nächsten Iterationen benötigt.

Schritt 15 :
Bilde τ_{n+1}^{-1}; dies ist das Inverse einer kleinen Matrix.
Nun berechnet man

$$\mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{r}_n), \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{w}_{n+1}) = \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{w}_n). \tag{3.86}$$

(vgl. auch S.31) durch Aufspalten von τ_{n+1}^{-1}.

Jetzt wird überprüft, ob die Vektoren von $\mathbf{r}_{n+1}$ linear unabhängig sind. Sind sie unabhängig, ist dies schnell geschehen (in allen bisherigen Tests war in ca. 70% aller Fälle nur höchstens ein Vektor in $\mathbf{r}_{n+1}$ enthalten). Sind die Vektoren in $\mathbf{r}_{n+1}$ dagegen linear abhängig, gilt

$$\mathbf{r}_{n+1}\psi_{n+1} = 0 \quad \text{für eine Matrix } \psi_{n+1} \neq 0; \tag{3.87}$$

der Test dauert wesentlich länger. In diesem Falle aber lässt sich die Größe von $\mathbf{w}_{n+1}$ reduzieren, also passiert dies höchstens `WordSize` Mal.

Wenn die Vektoren in $\mathbf{r}_{n+1}$ linear abhängig sind, löscht man redundante Vektoren aus $\mathbf{w}_{n+1}$ und $\mathbf{v}_{n+2}$.

Schritt 16 :
Hat man durch eine lineare Abhängigkeit in $\mathbf{r}_{n+1}$ eine Matrix $\psi_{n+1} \neq 0$ im vorigen Schritt gefunden, dann berechnet man

$$\mathbf{y} = A^{\mathtt{inv}}(\mathbf{r}_{n+1})\psi_{n+1}, \tag{3.88}$$

was das Produkt eines Vektorblocks mit einer kleinen Matrix ist.
Es gilt

$$A\mathbf{y} = \mathbf{b}\lambda \subseteq b. \tag{3.89}$$

War nun die ursprüngliche Gleichung homogen, so berechne

$$\hat{\mathbf{z}} = \begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix} \tag{3.90}$$

als Lösung der homogenen Gleichung

$$\hat{B}\hat{\mathbf{z}} = 0. \tag{3.91}$$

Hierbei wird das Transponierte der schwachbesetzten Matrix mit einem Vektorblock multipliziert und das Resultat mit einer kleinen Matrix vertikal konkateniert. Dies kann aber allerhöchstens `WordSize`–mal auftreten.

Die Berechnungsschritte (9)–(15) werden nun mit den neuen Werten von $\mathbf{w}_{n+1}$ und $\mathbf{v}_{n+2}$ wiederholt; nun hat man aber die Größe von $\mathbf{w}_{n+1}$ reduziert, und die Vektoren in $\mathbf{r}_{n+1}$ sind wieder linear unabhängig.
Der Operator

$$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{w}_n) \tag{3.92}$$

muß mit dem neuen Wert von $\mathbf{r}_{n+1}$ und dem alten Wert von $\mathbf{w}_n$ neu berechnet werden; es gilt nun

$$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{w}_n) \neq \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{w}_{n+1}), \tag{3.93}$$

da der linke Wert eine geringere Größe hat.

Schritt 17 :

$$\begin{aligned}
\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_{n-1}) &= \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{r}_n)\mathtt{Proj}(\mathbf{r}_n; A\mathbf{r}_{n-1}) \\
(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_n)) &= (\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{w}_n))\tau_n \\
\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) &= \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{r}_n)\mathtt{Proj}(\mathbf{r}_n; A\mathbf{f}_{n-1}) \qquad (3.94)
\end{aligned}$$

Dies sind lediglich Operationen auf kleinen Matrizen, welche z.T. schon in vorangegangenen Schritten berechnet wurden; keine Vektorblöcke wurden benutzt. Dies dient vor allem dazu, die Berechnung des Vektorblocks $\mathbf{f}_n$ unnötig zu machen, und zur effizienten Berechnung von $\mathbf{d}_n$ in Schritt (21).

Schritt 18 :
Als nächstes berechne eine größte Matrix mit vollem Rang σ_n, sodaß

$$(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n))\sigma_n = 0. \qquad (3.95)$$

Dies sind ebenfalls nur Operationen auf kleinen Matrizen; nämlich Konkatenation zweier kleiner Matrizen und Berechnen einer Basis des Lösungsraums der homogenen Gleichung 3.95.

Nun füllt man σ_n zu einer Basis von $GF(2)^k$ auf und bildet aus den hinzugefügten Spalten eine Matrix ω_n. Hier gilt :

$$k = \mathit{Spaltenzahl}(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1})) + \mathit{Spaltenzahl}(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n)) \qquad (3.96)$$

Schritt 19 :
Berechne

$$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_n) = (\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n))\omega_n \qquad (3.97)$$

Dies ist die Konkatenation und Multiplikation zweier kleiner Matrizen, die beide schon berechnet wurden. Das Resultat wird in Schritt (17) der nächsten und in Schritt (4) der übernächsten Iteration verwendet.

Schritt 20 :
Berechne die Matrix ρ_n mit

$$\sigma_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n \end{pmatrix} \rho_n = 0, \omega_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n \end{pmatrix} \rho_n = I. \qquad (3.98)$$

Also gilt

$$\rho_n = \begin{pmatrix} I & 0 \\ 0 & \xi_n^{-1} \end{pmatrix} \begin{pmatrix} \sigma_n^{\mathbf{T}} \\ \omega_n^{\mathbf{T}} \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ I \end{pmatrix}. \tag{3.99}$$

Durch diese bestimmte Wahl von ρ_n wird garantiert, daß immer gilt :

$$\mathbf{f}_n^{\mathbf{T}} \mathbf{d}_n = I. \tag{3.100}$$

Hier wird ebenfalls nur auf kleinen Matrizen operiert. So wurde ξ_n^{-1} in Schritt (14) der vorangegangenen Iteration berechnet, und σ_n und ω_n wurden eben in Schritt (18) berechnet. Also transponiert und invertiert man eine kleine Matrix, berechnet zwei vertikale Konkatenationen kleiner Matrizen und multipliziert die Ergebnisse.

Schritt 21 :
Berechne

$$\mathbf{d}_n = (\mathbf{d}_{n-1} \mid \mathbf{x}_n)\rho_n. \tag{3.101}$$

Dies ist die Konkatenation zweier Vektorblöcke und die Multiplikation des Ergebnisses mit einer kleinen Matrix; $\mathbf{d}_{n-2}$ wird zur Berechnung von $\mathbf{w}_{n+1}$ verwendet.

Schritt 22 :
Berechne

$$A^{\mathtt{inv}}(\mathbf{d}_n) = (A^{\mathtt{inv}}(\mathbf{d}_{n-1}) \mid A^{\mathtt{inv}}(\mathbf{x}_n))\rho_n. \tag{3.102}$$

Dies ist ebenfalls wie im vorigen Schritt die Konkatenation zweier Vektorblöcke und die Multiplikation des Resultates mit einer kleinen Matrix; $A^{\mathtt{inv}}(\mathbf{d}_{n-2})$ wird zur Berechnung von $A^{\mathtt{inv}}(\mathbf{w}_{n+1})$ verwendet.
Dieser Schritt ist wesentlich, um die Eigenschaften des $A^{\mathtt{inv}}$–Operators während den folgenden Berechnungen zu garantieren.

Schritt 23 :
Ist $\mathbf{w}_{n+1}$ nicht leer, erhöhe n um 1 und gehe zu Schritt 2.
Ist $\mathbf{w}_{n+1}$ leer, stop.
Wegen den orthogonalen Bedingungen an die Vektorblöcke (vgl. Kap. 3.2)wird die Größe von $\mathbf{w}_{n+1}$ in Schritt (15) von Zeit zu Zeit verringert. $\mathbf{w}_{n+1}$ wird deswegen leer für $n \approx N/\mathtt{WordSize}$.

3.5 Laufzeit– und Platzbedarfanalyse

In diesem Abschnitt gebe ich eine Zusammenfassung der Laufzeitanalye und des Speicherplatzbedarfs. Die einzelnen Schrittangaben beziehen sich auf im vorigen Abschnitt besprochene Schritte. Die Angaben von Kosten von einzelnen Routinen werden bei der Besprechung dieser Routinen in Kapitel 6 auf Seite 60ff. genauer besprochen und hergeleitet. Hier werden nur die zusammengefassten Zeiten verwendet.

Initialisierung
Das Setzen eines Vektorblocks braucht Zeit

$$c_0 = c_{CV} \in O(N).$$

Schritt 1 :
Das Kopieren eines Vektorblocks braucht Zeit

$$c_1 = c_{CV} \in O(N).$$

Schritt 2 :
Das Multiplizieren zweier kleiner Matrizen hat Laufzeit $c_{MM} \in O(\texttt{WordSize}^2)$. Das innere Produkt zweier Vektorblöcke läuft in Zeit $c_{MV} = O(N \cdot \texttt{WordSize})$. Also hat dieser Schritt insgesamt Laufzeit

$$c_2 = c_{MM} + c_{MV} \in O(N \cdot \texttt{WordSize}).$$

Schritt 3 :
Das Multiplizieren zweier kleiner Matrizen hat Laufzeit $c_{MM} \in O(\texttt{WordSize}^2)$. Das innere Produkt zweier Vektorblöcke läuft in Zeit $c_{MV} = O(N \cdot \texttt{WordSize})$. Also hat dieser Schritt insgesamt Laufzeit

$$c_3 = c_{MM} + c_{MV} \in O(N \cdot \texttt{WordSize}).$$

Schritt 4 :
Das Multiplizieren zweier kleiner Matrizen hat Laufzeit $c_{MM} \in O(\texttt{WordSize}^2)$. Also hat dieser Schritt insgesamt Laufzeit

$$c_4 = c_{MM} \in O(\texttt{WordSize}^2).$$

Schritt 5 :
Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Das Subtrahieren zweier Vektorblöcke benötigt Zeit $c_{SV} \in O(N)$. Zusammen braucht Schritt (5) also Zeit

$$c_5 = 3 \cdot (c_{VM} + c_{SV}) \in O(N \cdot \texttt{WordSize}).$$

Schritt 6 :
Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Das Subtrahieren zweier Vektorblöcke benötigt Zeit $c_{SV} \in O(N)$. Zusammen braucht Schritt (6) also, wie Schritt (5), Zeit

$$c_6 = 3 \cdot (c_{VM} + c_{SV}) \in O(N \cdot \texttt{WordSize}).$$

Schritt 7 :
Die Anwendung der dünnbesetzten Matrix braucht, wenn sie symmetrisch ist, Zeit

$$c_7 = c_{D\ddot{U}_1} \in O(N+T),$$

wenn sie asymmetrisch ist, Zeit

$$c_7 = c_{D\ddot{U}_2} \in O(N+T).$$

Schritt 8 :
Das Kopieren eines Vektorblocks braucht Zeit

$$c_8 = c_{CV} \in O(N).$$

Schritt 9 :
Das innere Produkt zweier Vektorblöcke läßt sich in Zeit $c_{MV} = O(N \cdot \texttt{WordSize})$ berechnen. Also gilt :

$$c_9 = 2 \cdot c_{MV} \in O(N \cdot \texttt{WordSize}).$$

Schritt 10 :
Alle notwendigen Operationen wurden in einer Funktion zusammengefasst. Man hat also

$$c_{10} = c_\eta \in O(\texttt{WordSize}^2).$$

Schritt 11 :
Alle notwendigen Operationen wurden in einer Funktion zusammengefasst. Man hat also

$$c_{11} = c_\tau \in O(\texttt{WordSize}^3).$$

Schritt 12 :
Das Konkatenieren zweier Vektorblöcke braucht Zeit $c_{KV} \in O(N)$. Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Das Dekonkatenieren zweier Vektorblöcke braucht Zeit $c_{DV} \in O(N)$. Also hat man insgesamt Laufzeit

$$c_{12} = c_{KV} + c_{MV} + c_{DV} \in O(N \cdot \texttt{WordSize}).$$

Schritt 13 :
Das Konkatenieren zweier Vektorblöcke braucht Zeit $c_{KV} \in O(N)$. Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Das Dekonkatenieren zweier Vektorblöcke braucht Zeit $c_{DV} \in O(N)$. Also hat man, wie in Schritt (12) auch, insgesamt Laufzeit

$$c_{13} = c_{KV} + c_{MV} + c_{DV} \in O(N \cdot \texttt{WordSize}).$$

Schritt 14 :
Das innere Produkt zweier Vektorblöcke läuft in Zeit $c_{MV} = O(N \cdot \texttt{WordSize})$. Das Invertieren einer kleinen Matrix braucht Zeit $c_{I_1} = O(\texttt{WordSize}^2)$. Also hat man insgesamt

$$c_{14} = c_{MV} + c_{I_1} \in O(N \cdot \texttt{WordSize}).$$

Schritt 15 :
Der Algorithmus zur Berechnung der Projektionen aus einer kleinen Matrix benötigt Laufzeit $c_{PO} \in O(\texttt{WordSize})$.
Das Invertieren einer kleinen Matrix braucht Zeit $c_{I_1} = O(\texttt{WordSize}^2)$. Die Überprüfung eines Vektorblocks auf lineare Abhängigkeit (und ggf. die Berechnung von ψ_{n+1}) hat Laufzeit $c_{VD} \in O(N \cdot \texttt{WordSize})$. Das Löschen von überflüssigen Vektoren braucht Zeit $c_{L\ddot{U}} \in O(N \cdot \texttt{WordSize})$. Also hat man insgesamt höchstens eine Laufzeit von

$$c_{15} = c_{po} + c_{I_1} + c_{L\ddot{U}} + c_{VD} \in O(N \cdot \texttt{WordSize}).$$

Schritt 16 :
Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Das Berechnen des Resultates im asymmetrischen Fall hat Laufzeit $c_{AR} \in O(N + T)$. Also hat man im symmetrischen Fall Laufzeit

$$c_{16} = c_{VM} \in O(N \cdot \texttt{WordSize}),$$

im asymmetrischen Fall jedoch

$$c_{16} = c_{VM} + c_{AR} \in O(N + T).$$

Dieser Schritt wird allerdings höchstens `WordSize`–mal ausgeführt.

Schritt 17 :
Das Multiplizieren zweier kleiner Matrizen hat Laufzeit $c_{MM} \in O(\texttt{WordSize}^2)$. Die Konkatenation zweier kleiner Matrizen läßt sich in Zeit $c_{KM} = O(\texttt{WordSize})$ berechnen. Die Dekonkatenation zweier kleiner Matrizen läßt sich ebenfalls in Zeit $c_{DM} = O(\texttt{WordSize})$. Also hat diese Schritt zusammen Laufzeit

$$c_{17} = 3 \cdot c_{MM} + c_{DM} + c_{KM} \in O(\texttt{WordSize}^2).$$

Schritt 18 :
Das Berechnen der Matrix σ_n braucht Zeit $c_\sigma = O(\texttt{WordSize}^3)$; die Berechnung der Matrix ω_n braucht Zeit $c_\omega = O(\texttt{WordSize}^3)$; also insgesamt hat dieser Schritt Laufzeit

$$c_{18} = c_\sigma + c_\omega \in O(\texttt{WordSize}^3).$$

Schritt 19 :
Die Multiplikation zweier kleiner Matrizen hat Laufzeit $c_{MM} \in O(\texttt{WordSize}^2)$. Also hat dieser Schritt Laufzeit

$$c_{19} = c_{MM} \in O(\texttt{WordSize}^2).$$

Schritt 20 :
Diese Berechnungen wurden in einer Funktion realisiert. Man hat also

$$c_{20} = c_\rho \in O(\texttt{WordSize}^2).$$

Schritt 21 :
Das Konkatenieren zweier Vektorblöcke braucht Zeit $c_{KV} \in O(N)$. Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Dieser Schritt hat also Laufzeit

$$c_{21} = c_{KV} + c_{VM} \in O(N \cdot \texttt{WordSize}).$$

Schritt 22 :
Das Konkatenieren zweier Vektorblöcke braucht Zeit $c_{KV} \in O(N)$. Das Produkt Vektorblock mal kleine Matrix benötigt $c_{VM} \in O(N \cdot \texttt{WordSize})$ Operationen. Dieser Schritt hat also ebenfalls Laufzeit

$$c_{22} = c_{KV} + c_{VM} \in O(N \cdot \texttt{WordSize}).$$

Schritt 23 :
Der Test, ob ein Vektorblock leer ist, sowie das Inkrementieren eines Zählers benötigen jeweils Zeit $O(1)$. Also gilt

$$c_{23} \in O(1).$$

Zusammenfassung :
Die Schleife Schritt (9)–Schritt (16) wird höchstens `WordSize`–mal mehr durchlaufen als die Schleife Schritt (2)–Schritt (23), da man höchstens `WordSize` Vektoren in $\mathbf{w}_0$ hat, und die Größe von $\mathbf{w}_{n+1}$ folglich höchstens so oft verringert werden kann.
Die Schleife Schritt (2)–Schritt (23) wird $n \approx N/\texttt{WordSize}$–mal durchlaufen (vgl. z.B 3.3).
Man hat also insgesamt

$$\begin{aligned} c_{BL} &= c_0 + c_1 + N \cdot (c_2 + c_3 + \ldots + c_{15} + c_{17} + c_{18} + \ldots + c_{23}) \\ &\quad + \texttt{WordSize} \cdot (c_9 + c_{10} + \ldots + c_{16}) \end{aligned} \tag{3.103}$$

Wegen den Ausführungen zur Laufzeit von Schritt (7) hat man also sowohl im symmetrischen als auch im asymmetrischen Fall eine Gesamtlaufzeit des Algorithmus von

$$O(N \cdot (N + T)).$$

Platzbedarf :
Die Angaben des Speicherplatzbedarfs der einzelnen Datenstrukturen wird bei der Besprechung dieser Strukturen in Kapitel 5 auf Seite 55ff. genauer besprochen und hergeleitet.
Man muß die Vektorblöcke $\mathbf{w}$, $\mathbf{v}$, $\mathbf{x}$, $\mathbf{r}$, $\mathbf{d}$, $A^{\mathtt{inv}}(\mathbf{w})$, $A^{\mathtt{inv}}(\mathbf{v})$, $A^{\mathtt{inv}}(\mathbf{x})$, $A^{\mathtt{inv}}(\mathbf{r})$ und $A^{\mathtt{inv}}(\mathbf{d})$ speichern. Von jedem dieser Vektorblöcke braucht man zwei Versionen. Zusätzlich braucht man noch einen Vektorblock zur Speicherung des Resultates und einen Block zur Speicherung der zu bestimmenden Lösungen. Jeder Vektorblock braucht $S_V \in O(N)$ Speicherplatz, und man muss insgesamt 22 Vektorblöcke speichern. Die große dünnbesetzte Matrix braucht Platz $S_M \in O(N+T)$.
Weiter braucht man noch 14 kleine Matrizen. Jede kleine Matrix verbraucht $S_S \in O(\mathtt{WordSize})$ Platz.

Kapitel 4

Struktur der Implementierung

In diesem Kapitel beschäftige ich mich mit den Struktogrammen zum Hauptprogramm und den Struktogrammen, die den Algorithmus zum Lösen des Gleichungssystems beschreiben.
Dieses Kapitel ist unterteilt in drei Abschnitte :

- Struktur des Hauptprogramms
- Struktur des Algorithmus zum Lösen des Gleichungssystems
- Bemerkungen zur Implementierung

4.1 Struktur des Hauptprogramms

Das Hauptprogramm erledigt notwendige Arbeiten wie z.B. Einlesen einer Matrix, Speicherhandling oder Handling des LOG-Files. Dies sind im Einzelnen :

1. Abfrage der Kommandoparameter und Setzen der entsprechenden globalen Flags
2. Einlesen einer Matrix bzw. Erzeugen einer zufälligen Matrix
3. Reservierung des zusätzlich benötigten Speicherplatzes
4. Initialisierung der globalen Variablen
5. Preprocessing und Lösen des Gleichungssystems
6. Abspeichern der Matrix und des Resultates, wenn erwünscht
7. Ausgabe von eventuellen Fehlermeldungen,bzw. Speichern des LOG-Files

8. Freigabe des benötigten Speicherplatzes, Aufräumarbeiten

Das Struktogramm des Hauptprogramms ist :

Abfrage der Kommand Line Parameter;		(1)
Initialisierung der globalen Variablen;		(2)
IF File mit Matrix angegeben		(3)
THEN	lese Matrix ein;	(4)
ELSE	erzeuge zufällige Matrix;	(5)
Reservierung des benötigten Speicherplatzes;		(6)
Preprocessing;		(7)
Lösen des Gleichungssystems;		(8)
IF Matrix soll abgespeichert werden		(9)
THEN	speichere Matrix und Resultat ab;	(10)
Freigabe des benutzten Speichers;		(11)
Speichern des LOG-Files;		(12)

Zur Beschreibung der Kommandoparameter vergleiche Kapitel C im Anhang.
Zum Erzeugen einer zufälligen Matrix vergleiche Kapitel 6.1.1.
Zum Format der einzulesenden Matrix vergleiche Kapitel B im Anhang.
Zum Format des LOG–Files vergleiche Kapitel A im Anhang.

4.2 Struktur des Lösungsalgorithmus

In diesem Abschnitt folgt das Struktogramm des implementierten Algorithmus, wie er in Kapitel 3.4 beschrieben wurde. Dieses ist zur übersichtlicheren Darstellung in einen Steuerblock und drei Unterblöcke aufgeteilt. Die Aufteilung ist dabei analog zu derjenigen in Kapitel 3.2.

Steuerblock
Hier steht das Struktogramm des Steuerblocks.

Initialisierung der Vektorblöcke mit 0; $\mathbf{v}_0 \leftarrow \mathbf{b}$;	(1)
$A^{\mathtt{inv}}(\mathbf{v}_0) \leftarrow 0$; $n \leftarrow -1$;	(2)
Block 1 : Berechnung von $\mathbf{w}_{n+1}$, $\mathbf{v}_{n+2}$, $A^{\mathtt{inv}}(\mathbf{w}_{n+1})$ und $A^{\mathtt{inv}}(\mathbf{v}_{n+2})$;	(3)
Block 2 : Berechnung von $\mathbf{r}_{n+1}$, $\mathbf{x}_{n+1}$, $A^{\mathtt{inv}}(\mathbf{r}_{n+1})$ und $A^{\mathtt{inv}}(\mathbf{x}_{n+1})$;	(4)
Block 3 : Berechnung einer Lösung;	(5)
UNTIL Die Vektoren in $\mathbf{r}_{n+1}$ sind linear unabhängig	(6)
Block 4 : Berechnung der 'Projektions'operatoren und der Vektorblöcke $\mathbf{d}_n$ und $A^{\mathtt{inv}}(\mathbf{d}_n)$;	(7)
IF $\mathbf{w}_{n+1}$ ist nicht leer	(8)
THEN inkrementiere n;	(9)
UNTIL $\mathbf{w}_{n+1}$ ist leer	(10)

Der Aufbau der Unterblöcke 1–4 wird in den folgenden Teilabschnitten dargestellt.

Block 1

In diesem Block werden die Schritte (2) – (8) des Algorithmus beschrieben. Dies sind diejenigen Schritte, die sich mit der Berechnung von $\mathbf{w}_{n+1}$ und $\mathbf{v}_{n+2}$ und der dazugehörigen Vektorblöcke $A^{\mathtt{inv}}(\mathbf{w}_{n+1})$ und $A^{\mathtt{inv}}(\mathbf{v}_{n+2})$ beschäftigen.

$\alpha_n \leftarrow (\mathbf{x}_n^{\mathbf{T}}\mathbf{x}_n)^{-1}(\mathbf{x}_n^{\mathbf{T}}\mathbf{v}_{n+1})$;	(1)
$\beta_{n-1} \leftarrow (\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{x}_{n-1})^{-1}(\mathbf{x}_{n-1}^{\mathbf{T}}\mathbf{v}_{n+1})$;	(2)
$\gamma_{n-2} \leftarrow (\mathtt{Proj}(\mathbf{r}_{n-1}; A\mathbf{f}_{n-2}))^{\mathbf{T}}(\mathbf{r}_{n-1}^{\mathbf{T}}\mathbf{w}_n)$;	(3)
$\mathbf{w}_{n+1} \leftarrow \mathbf{v}_{n+1} - \mathbf{x}_n\alpha_n - \mathbf{x}_{n-1}\beta_{n-1} - \mathbf{d}_{n-2}\gamma_{n-2}$;	(4)
$A^{\mathtt{inv}}(\mathbf{w}_{n+1}) \leftarrow A^{\mathtt{inv}}(\mathbf{v}_{n+1}) - A^{\mathtt{inv}}(\mathbf{x}_n)\alpha_n - A^{\mathtt{inv}}(\mathbf{x}_{n-1})\beta_{n-1}$ $-A^{\mathtt{inv}}(\mathbf{d}_{n-2})\gamma_{n-2}$;	(5)
$\mathbf{v}_{n+2} \leftarrow A\mathbf{w}_{n+1}$;	(6)
$A^{\mathtt{inv}}(\mathbf{v}_{n+2}) \leftarrow \mathbf{w}_{n+1}$;	(7)

Block 2

In diesem Block werden die Schritte (9) – (15) des Algorithmus beschrieben. In diesen Schritten erfolgt die Berechnung der Vektorblöcke $\mathbf{r}_{n+1}$ und $\mathbf{x}_{n+1}$ und der dazugehörenden $A^{\mathtt{inv}}(\mathbf{r}_{n+1})$ bzw. $A^{\mathtt{inv}}(\mathbf{x}_{n+1})$–Blöcke. Weiter wird hier der Test auf lineare Unabhängigkeit von $\mathbf{r}_{n+1}$ vorgenommen.

berechne $\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{r}_n$ und $\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{w}_{n+1}$;		(1)
$\eta_{n+1} \leftarrow (\mathbf{r}_n \mid \mathbf{w}_{n+1})^{\mathbf{T}}(\mathbf{r}_n \mid \mathbf{w}_{n+1})$;		(2)
berechne τ_{n+1} mit $\tau_{n+1}^{\mathbf{T}}\eta_{n+1}\tau_{n+1} = \begin{pmatrix} \xi_{n+1} & 0 \\ 0 & 0 \end{pmatrix}$;		(3)
$(\mathbf{x}_{n+1} \mid \mathbf{r}_{n+1}) \leftarrow (\mathbf{r}_n \mid \mathbf{w}_{n+1})\tau_{n+1}$;		(4)
$(A^{\mathtt{inv}}(\mathbf{x}_{n+1}) \mid A^{\mathtt{inv}}(\mathbf{r}_{n+1})) \leftarrow (A^{\mathtt{inv}}(\mathbf{r}_n) \mid A^{\mathtt{inv}}(\mathbf{w}_{n+1}))\tau_{n+1}$;		(5)
$\xi_{n+1}^{-1} \leftarrow (\mathbf{x}_{n+1}^{\mathbf{T}}\mathbf{x}_{n+1})^{-1}$;		(6)
berechne τ_{n+1}^{-1}; berechne $\mathtt{Proj}(\mathbf{r}_{n+1};\mathbf{r}_n)$ und $\mathtt{Proj}(\mathbf{r}_{n+1};A\mathbf{w}_n)$ als Untermatrizen von τ_{n+1}^{-1}		(7)
IF Vektoren in $\mathbf{r}_{n+1}$ sind linear abhängig		(8)
THEN	berechne ψ_{n+1} mit $\mathbf{r}_{n+1}\psi_{n+1} = 0$ und $\psi_{n+1} \neq 0$;	(9)
	lösche die redundanten Vektoren aus $\mathbf{v}_{n+2}$ und $\mathbf{w}_{n+1}$;	(10)

Dieser Block wird im allgemeinen nur einmal (in der Schleife (5) des vorangegangenen Struktogramms) durchlaufen. Nur in wenigen Fällen sind die Vektoren in $\mathbf{r}_{n+1}$ linear abhängig.

Block 3

In diesem Block wird Schritt (16) des Algorithmus dargestellt. Hier werden die Lösungen des Systems berechnet.

IF $\psi_{n+1} \neq 0$			(1)
THEN	berechne $\mathbf{y} \leftarrow A^{\mathtt{inv}}(\mathbf{r}_{n+1})\psi_{n+1}$ als Lösung des Gleichungssystems;		(2)
	IF ursprüngliches Gleichungssystem homogen		(3)
	THEN	$\hat{\mathbf{z}} \leftarrow \begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix}$;	(4)

Block 4
In diesem Block werden die Schritte (17)–(22) des Algorithmus dargestellt. Hier werden die Vektorblöcke $\mathbf{d}_n$ und $A^{\mathtt{inv}}(\mathbf{d}_n)$ und die 'Projektions'operatoren durch Operationen auf kleinen Matrizen berechnet.

$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_{n-1}) \leftarrow \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{r}_n)\mathtt{Proj}(\mathbf{r}_n; A\mathbf{r}_{n-1});$	(1)
$\iota_{n+1} \leftarrow (\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{w}_n));$ $(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{r}_n)) \leftarrow \iota_{n+1}\tau_n;$	(2)
$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) \leftarrow \mathtt{Proj}(\mathbf{r}_{n+1}; \mathbf{r}_n)\mathtt{Proj}(\mathbf{r}_n; A\mathbf{f}_{n-1});$	(3)
berechne eine größte Matrix mit vollem Rang σ_n, sodaß $(\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n))\sigma_n = 0;$	(4)
fülle σ_n zu einer Basis von $GF(2)^k$ auf; lasse die übrigen Spalten dieser Basis eine Matrix ω_n bilden;	(5)
$\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_n) \leftarrow (\mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{f}_{n-1}) \mid \mathtt{Proj}(\mathbf{r}_{n+1}; A\mathbf{x}_n))\omega_n;$	(6)
berechne ρ_n mit $\sigma_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \mathbf{x}_n^{\mathbf{T}} bfx_n \end{pmatrix} \rho_n = 0, \omega_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \mathbf{x}_n^{\mathbf{T}} bfx_n \end{pmatrix} \rho_n = I;$	(7)
$\mathbf{d}_n \leftarrow (\mathbf{d}_{n-1} \mid \mathbf{x}_n)\rho_n;$	(8)
$A^{\mathtt{inv}}(\mathbf{d}_n) \leftarrow (A^{\mathtt{inv}}(\mathbf{d}_{n-1}) \mid A^{\mathtt{inv}}(\mathbf{x}_n))\rho_n;$	(9)

4.3 Bemerkungen zur Implementierung

Das eigentliche Lösen des Gleichungssystems wurde folgendermaßen realisiert : Die Operationen auf den neu eingeführten Datentypen wurden als separate Funktionen geschrieben. Der Block Lanczos Algorithmus wurde darauf mit Hilfe dieser Funktionen formuliert. Dies hat den Vorteil, daß bei einer eventuell später erfolgenden Parallelisierung des Algorithmus lediglich die Operationen auf den Datentypen parallelisiert werden müssen, der Hauptteil des Algorithmus aber kann übernommen werden. Weiter sind spätere evtl. notwendige Änderungen am Algorithmus durch z.B effizientere Methoden, Teilprobleme zu lösen, leichter zu implementieren und auszutesten.
Nachteile dieser Methode sind ein etwas erhöhter Speicherplatzbedarf und erhöhter praktische Laufzeiten durch Parameterübergabe und Funktionsaufrufe.

Alle Teile des Algorithmus wurden in ANSI–C (vgl. [7]) geschrieben, um die Portierung auf verschiedenen Rechnerarchitekturen zu erleichtern. Das Programm selbst wurde zuerst auf einem IBM PC 386 implementiert und ausgetestet, dann auf die SPARC-Stations portiert.

Kapitel 5

Datenstrukturen

In diesem Kapitel werden die für den Algorithmus neu definierten Datenstrukturen und ihre Verwendung erklärt.
Diese sind im einzelnen :

- Eine Datenstruktur für die große dünnbesetzte Matrix
- Eine Datenstruktur für einen Vektorblock
- Eine Datenstruktur für eine kleine dichte Matrix

5.1 Die große dünnbesetzte Matrix

Sei A eine $N \times M$- Matrix mit insgesamt T Einträgen $\neq 0$. Dann errechnet man die durchschnittliche Anzahl D an Einträgen $\neq 0$ pro Zeile durch

$$D = T/N$$

Die Matrix A wird als eine Liste von N Listen gespeichert, wobei in jeder dieser N Listen die Einträge $\neq 0$ einer Zeile abgespeichert werden.
Um Speicherplatz zu sparen, werden dabei die linearen Listen durch den in C möglichen Mechanismus der dynamischen Felder simuliert. Dies führt zu einer Reduktion des Speicherbedarfs um etwa die Hälfte. Die Verwendung von dynamischen Feldern anstelle von linearen Listen ergab sogar geringe Vorteile im Laufzeitverhalten. Die Matrix A wird aufgebaut(Einlesen vom externen Speicher oder zufällige Erzeugung) und nur einmal durch Preprocessing verändert (vgl. 2.5, S. 24). Alle weiteren Anwendungen der Matrix sind Matrix-Vektor-Multiplikationen, bei denen die Matrix nicht verändert wird. Die geringen Zeitersparnisse ergeben sich daraus, daß weniger Pointer-Operationen bei der Verwendung von dynamischen Feldern vorgenommen werden müssen.

Im folgenden wird die Datenstruktur für die Matrix A angegeben.
Jene zerfällt in zwei Teile :

1. Die Struktur für die Verwaltung der Einträge $\neq 0$ einer Zeile
2. Die Struktur für die Verwaltung der Zeilen

Bei den folgenden Definitionen wurden folgende Typen definiert, um die Portierung von 16-Bit– auf 32-Bit Computer zu erleichtern :

```
typedef unsigned int  WORD;
typedef unsigned long DWORD;
```

Zuerst wird die Struktur für die Verwaltung der Einträge $\neq 0$ einer Zeile angeführt :

```
typedef struct d_row
{
   unsigned size;
   DWORD *col;
}  ROW;
```

Enthält eine Zeile der Matrix d Einträge $\neq 0$, so wird der benötigte Speicherplatz für diese Spalte durch

```
   col = ( DWORD * )calloc( d, sizeof( DWORD ) );
   size = D;
```

reserviert. In dem Eintrag col$[i]$ des dynamischen Feldes, $0 \leq i < d$, wird die Position des i-ten Eintrags $\neq 0$ dieser Zeile gespeichert. Die Komponente `size` gibt dabei die gesamte Anzahl der Einträge $\neq 0$ und damit die relative Größe des dynamischen Feldes an.

Dann wird die Struktur zur Verwaltung der Zeilen der Matrix angegeben :

```
typedef struct d_sparse_matrix
{
   DWORD total, rows, columns;
   ROW *row;
}  SPARSE_MATRIX;
```

Die Matrix enthält, wie oben erwähnt, N Zeilen und M Spalten (wobei gilt : total $= T$, rows $= N$ und columns $= M$). Den Speicherplatz für die Verwaltung dieser N Zeilen erhält man wie folgt :

```
row = ( ROW * )calloc( N, sizeof( ROW ) );
rows = N;
columns = M;
total = T;
```

In dem Eintrag row$[i]$ dieses Feldes, $0 \leq i < N$, wird die i-te Zeile der Matrix A gespeichert. Die Komponente rows gibt die Anzahl Zeilen der Matrix an (und damit die relative Größe des dynamischen Feldes), die Komponente columns gibt die Anzahl Spalten der Matrix an.

Ist A eine $N \times M$- Matrix mit insgesamt T Einträgen $\neq 0$ und durchschnittlich D Einträgen $\neq 0$ pro Zeile, so läßt sich der zur Speicherung von A benötigte Speicherplatz S in Bytes folgendermaßen berechnen :

$$\begin{aligned} S_M &= \text{Größe der Verwaltungsstruktur für die Zeilen} + \\ &\quad N \cdot \text{ Größe der Verwaltungsstruktur einer Zeile} + T \\ &= \text{Größe der Verwaltungsstruktur für die Zeilen} + \\ &\quad N \cdot \text{ Größe der Verwaltungsstruktur einer Zeile} + N \cdot D \end{aligned}$$

Beispiel 5.1
Ist A eine 100.000×100.000 Matrix mit insgesamt 2.000.000 Einträgen $\neq 0$, so belegt diese Matrix

$$\begin{aligned} S_M &= 12 + 8 \cdot 100.000 + 4 \cdot 2.000.000 \\ &= 8.800.012 \text{ Bytes, also ca. 8.4 MBytes.} \end{aligned}$$

unter der Voraussetzung, daß ein Zeiger, gleichgültig welchen Typs, genau 4 Bytes und ein Element des Typs DWORD wie ein Element des Typs WORD ebenfalls genau 4 Bytes verbrauchen.

5.2 Ein Vektorblock

Wie in Abschnitt 2.4 erläutert, werden in einem Vektorblock maximal WordSize Vektoren aus $GF(2)^N$ zusammengefasst, wobei WordSize ja die Wortgröße in Bit der jeweiligen Maschine ist.
Ein Vektorblock wird als Feld von N Komponenten gespeichert, wobei jeder Eintrag $i, 0 \leq i < N$, die i-ten Komponenten der in diesem Block zusammengefassten Vektoren aus $GF(2)^N$ enthält. Die Verwendung eines (dynamischen) Feldes ist der einer linearen Liste vorzuziehen, da die Anzahl Komponenten eines Vektorblocks während

der Berechnung nicht verändert wird, sondern nur deren Inhalt. Somit ergibt sich wie bei der Speicherung der großen Matrix auch sowohl eine Speicherplatzersparnis als auch ein geringer Laufzeitvorteil.
Die Struktur eines solchen Vektorblockes besteht aus :

```
typedef struct d_vector
{
   unsigned size;
   WORD *array;
}  VECTOR;
```

Den Speicherplatz für einen Vektorblock erhält man folgendermaßen :

```
   array = ( WORD * )calloc( N, sizeof( WORD ) );
   size = k;
```

Jeder Eintrag in array$[i]$, $0 \leq i < N$, enthält, wie oben schon gesagt, die i-ten Komponenten der Vektoren, die zu diesem Vektorblock zusammengefasst wurden. Hier gibt `size` = `k` $\leq$ `WordSize` die Anzahl Vektoren aus $GF(2)^N$ an, die in diesem Vektorblock enthalten sind.

Sei `w` die Wortgröße der Maschine in Byte.
Ist $\mathbf{v}$ ein Vektorblock, der aus k Vektoren aus $GF(2)^N$ besteht, $0 < k \leq$ `WordSize`, so berechnet sich der zur Speicherung von $\mathbf{v}$ benötigte Platz S in Bytes wie folgt :

$$S_V = \text{Größe der Verwaltungsstruktur} + \mathtt{w} \cdot N.$$

Beispiel 5.2
Ist $N = 100.000$ und $\mathtt{w} = 4$, so belegt ein Vektorblock aus $GF(2)^N$

$$\begin{aligned} S_V &= 8 + 4 \cdot 100.000. \\ &= 400.008 \text{ Bytes, also ca. 390 KBytes.} \end{aligned}$$

5.3 Eine kleine dichte Matrix

Eine kleine Matrix α ist eine $k \times l$-Matrix mit $0 < k, l \leq$ `WordSize`.
Sie wird als ein Feld von `WordSize` Komponenten gespeichert, wobei jede Komponente des Feldes eine Zeile der kleinen Matrix enthält. Die Reservierung von `WordSize` Komponenten wird dadurch notwendig, daß einerseits sich die Dimensionen der in dem Algorithmus verwendeten kleinen Matrizen sehr oft ändern, also die Verwendung von linearen Listen zu größerem Zeitaufwand bei der Speicherverwaltung der Maschine führen würden, und andererseits aus dem gleichen Grund eine Verwendung

von dynamischen Feldern unzweckmäßig ist. Weiterhin ist der im Laufe der Berechnung nicht benutzte und damit 'unnötig' reservierte Speicherplatz in Relation zu z.B. dem zur Speicherung der großen Matrix benötigten Platz vernachlässigbar klein (in der Praxis ca. 1–2 KBytes).
Die Struktur einer solchen kleinen Matrix ist :

```
typedef struct d_small_matrix
{
   unsigned rows, columns;
   WORD *matrix;
}  SMALL_MATRIX;
```

Den Speicherplatz für eine solche Matrix erhält man durch

```
   matrix = ( WORD * )calloc( WordSize, sizeof( WORD ) );
   rows = k;
   columns = l;
```

Der Wert `matrix`$[i]$ gibt hierbei die i-te Zeile der Matrix α an.

Sei w die Wortgröße der Maschine in Byte.
Ist α eine solche kleine Matrix, so berechnet sich der zur Speicherung von α benötigte Platz S in Bytes zu :

$$S_S = \text{Größe der Verwaltungsstruktur} + w \cdot \texttt{WordSize}$$

Beispiel 5.3
Ist `WordSize` $= 32$ und `w` $= 4$, so belegt ein kleine Matrix

$$\begin{aligned} S_S &= 12 + 4 \cdot 32 \\ &= 140 \text{ Bytes.} \end{aligned}$$

Die Größe einer kleinen Matrix ist auf einer bestimmten Rechnerarchitektur konstant, kann jedoch bei unterschiedlichen Architekturen auch unterschiedlich viel Platz belegen (abhängig von `WordSize`).

Kapitel 6

Operationen auf den Datenstrukturen

In diesem Kapitel werden die einfachen Operationen auf den im vorigen Kapitel eingeführten Datenstrukturen erläutert. Diese werden in vielen Schritten des Algorithmus mit wechselnden Parametern verwendet.
Die Beschreibung einer solchen Operation ist gegliedert in :

- Beschreibung der auszuführenden Operation
- notwendige Bedingungen vor der Ausführung
- Algorithmus
- zusätzliche Bedingungen nach der Ausführung
- Benutzte Resourcen oder Laufzeitverhalten

Bei der Laufzeitangabe wird davon ausgegangen, daß die Indizierungsoperationen in konstanter Zeit ausgeführt werden können. Auf sie wird in den folgenden Kapiteln nicht mehr weiter eingegangen. Die angegebenen Laufzeiten sind, wenn nicht ausdrücklich anders bezeichnet, alle worst-case Laufzeiten.
Grundvoraussetzung für das korrekte Funktionieren all dieser Algorithmen ist, daß der notwendige Speicherplatz für die vorkommenden Matrizen und Vektorblöcke schon reserviert ist. Den Algorithmen selbst wird nur ein Zeiger auf die Verwaltungsstrukturen der jeweiligen Matrizen oder Vektorblöcke übergeben.
Falls nicht anders angegeben, enthalten alle Vektorblöcke $\leq$ `WordSize` Vektoren aus $GF(2)^N$, und die kleinen Matrizen sind $k \times l$–Matrizen über $GF(2)$ mit $0 \leq k, l <$ `WordSize`.

Der Rest dieses Kapitels ist gegliedert in fünf Abschnitte :

Abschnitt 1 : Hier werden die Operationen mit den großen, dünnbesetzten Matrizen beschrieben.

Abschnitt 2 : Hier werden alle einfachen Operationen, die Vektorblöcke als Operanden erhalten, erläutert.

Abschnitt 3 : Hier werden alle einfachen Operationen, die kleine Matrizen als Operanden erhalten, erläutert.

Abschnitt 4 : Hier werden alle einfachen gemischten Operationen erläutert.

Abschnitt 5 : In diesem Abschnitt schließlich werden alle speziellen Operationen erläutert.

Die Algorithmen selbst werden mit Hilfe des in Kapitel 1 erläuterten Struktogramms notiert und beschrieben. In den Abschnitten 2–4 dieses Kapitels werden die Algorithmen auf einer sehr einfachen Stufe beschrieben, die sich in der Praxis ohne größere Änderungen implementieren läßt. Die Algorithmen in Abschnitt 5 dagegen werden auf einer abstrakten Stufe, teilweise unter Verwendung (nur durch Nennung) von Algorithmen der Abschnitte 2–4, erläutert.

6.1 Operationen auf schwachbesetzten Matrizen

Dieser Abschnitt beschreibt die Operationen auf der großen dünnbesetzten Matrix. Dieses sind im einzelnen :

- Zufälliges Erzeugen einer quadratischen schwachbesetzten Matrix
- Preprocessing

6.1.1 Erzeugen einer zufälligen Matrix

Eine zufällige Matrix wird folgendermaßen erzeugt :
Es werden die gewünschte Anzahl Zeilen in der Matrix und die durchschnittliche Anzahl D Einträge $\neq 0$ pro Zeile abgefragt. Die tatsächliche Anzahl Einträge $\neq 0$ pro Zeile variiert zwischen 1 und dem Doppelten der erwünschten durchschnittlichen Anzahl Einträge $\neq 0$ pro Zeile $2D$.
Dies führt zu folgendem Struktogrammm :

Abfrage der gewünschten Anzahl Zeilen N und Spalten M;	(1)
Abfrage der durchschnittlichen Anzahl Einträge $\neq 0$ D pro Zeile;	(2)
Reservierung des Speicherplatzes für N Zeilen	(3)
FOR $i \leftarrow 1$ TO $N/2D$	(4)
FOR $j \leftarrow 1$ TO $2D$	(5)
reserviere Speicher für j Spalteneinträge $\neq 0$ in Zeile $i \cdot 2D + j$;	(6)
die Positionen dieser j Einträge $\neq 0$ werden zufällig gesetzt;	(7)

Der Algorithmus hat Laufzeit $O(\ N\ )$; zum zufälligen Setzen von D Werten braucht man D Zuweisungen; die innere Schleife wird $2D$–mal, und die äußere Schleife wird $N/(\ 2D\ )$–mal durchlaufen, das Reservieren von Speicher benutzt die in Kap. 5 erwähnten Systemfunktionen und kostet Zeit c_S. Man hat also insgesamt

$$c_{EM} = N/2D \cdot (\ 2D + \sum_{j=0}^{2D} j\ ) + 3 + c_S$$

Operationen.

6.1.2 Preprocessing

Zu Preprocessing vergleiche auch die Ausführungen in Kapitel 2.5 auf Seite 24 ff.

Diese Ausführungen führen zu folgendem Algorithmus :

/* erster Teil : lösche leere Spalte */	(1)
FOR $i \leftarrow 1$ TO N	(2)
WHILE Spalteneinträge in Zeile i	(3)
markiere Spalteneintrag j in Zeile i	(4)
lösche nichtmarkierte Spalteneinträge;	(5)

/* zweiter Teil : lösche 'Singletons' */	(1)
FOR $i \leftarrow 1$ TO N	(2)
IF keine Spalteneinträge in Zeile i	(3)
THEN lösche Zeile i;	(4)
IF nur ein Spalteneintrag in Zeile i	(5)
THEN lösche Zeile i;	(6)
/* lösche entsprechende Spalte durch Durchmustern der Matrix */;	(7)
FOR $j \leftarrow 1$ TO N	(8)
WHILE Spalteneinträge in Zeile j	(9)
IF Spalteneintrag i in Zeile j	(10)
THEN lösche Spalteneintrag i;	(11)
UNTIL keine singletons mehr in der Matrix	(12)

Im ersten Teil des Algorithmus müssen N Zeilen durchsucht werden. In jeder Zeile j sind g_j Spalteneinträge, also werden in der ersten FOR–Schleife g_j Zuweisungsoperationen vorgenommen. Da nun

$$\sum_{j=1}^{N} g_j = T,$$

und man höchstens M nichtmarkierte Spalteneinträge löschen muss, hat man zum Löschen der leeren Spalten höchstens

$$c_{LL} = M + \sum_{j=1}^{N} g_j$$

Operationen, also Laufzeit $O(\,M + T\,)$. Die äußere FOR-Schleife im zweiten Teil des Algorithmus wird N–mal durchlaufen. In dieser Schleife hat man zwei Vergleichsoperationen. Ist eine leere Zeile gefunden, hat man zusätzlich eine Zuweisungsoperation. Ist ein Singleton gefunden, braucht man noch eine Zuweisungsoperation, und die Matrix wird einmal durchsucht.
Es müssen N Zeilen durchsucht werden (auch wenn kein Eintrag in den Zeilen vorhanden ist!). In jeder Zeile j sind g_j Spalteneinträge, also werden in der inneren Schleife g_j Vergleichs– und höchstens g_j Zuweisungsoperationen vorgenommen. Es gilt :

$$\sum_{j=1}^{N} g_j = T.$$

Dieses Durchsuchen zum Finden und Löschen aller Spalteneinträge einer beliebigen Spalte benötigt also Zeit $O(\ N+T\ )$. Insgesamt hat man höchstens

$$c_P = c_{LL} + N \cdot (\ 4 + 2 \cdot \sum_{j=1}^{N} g_j\ ) \text{ Operationen,}$$

also benötigt ein Durchgang zum Löschen der Singletons $O(\ N(\ N+T\ )\ )$.
Durch das Löschen von Spalteneinträgen in allen Zeilen können nun Singletons in schon durchmusterten Zeilen auftreten (vgl. 2.5). Dies bedingt, daß im schlechtesten Falle (z.B. bei einer rechten oberen Dreiecksmatrix) N Durchgänge zum Löschen von Singletons vorgenommen werden müssen, im besten Fall jedoch nur ein Durchgang (z.B. bei einer linken unteren Dreiecksmatrix).
Also hat man im besten Falle eine Gesamtlaufzeit des Algorithmus von $O(M+N(N+T\ )\ )$, im schlechtesten Falle jedoch eine Gesamtlaufzeit von $O(\ M+N^2 \cdot (\ N+T\ )\ )$.

6.2 Einfache Operationen mit Vektorblöcken

Dieser Abschnitt beschreibt alle einfachen Operationen auf Vektorblöcken. Dieses sind im einzelnen :

- Subtraktion von Vektorblöcken
- Konkatenation von Vektorblöcken
- Multiplikation von Vektorblöcken
- Feststellen der Größe eines Vektorblocks
- Kopieren eines Vektorblocks
- Test, ob ein Vektorblock leer ist
- Aufspalten eines Vektorblock in zwei Vektorblöcke

6.2.1 Subtraktion von Vektorblöcken

Seien $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke. Es soll die Operation

$$\mathbf{w} = \mathbf{u} - \mathbf{v}$$

ausgeführt werden.
Es gilt $0 \leq \mathbf{u}.\text{size}, \mathbf{v}.\text{size} < \texttt{WordSize}$.
Diese Operation erledigt der folgende Algorithmus :

FOR $i \leftarrow 0$ TO N	(1)
$\mathbf{w}.array[i] \leftarrow (\ \mathbf{u}.array[i] \quad \text{XOR} \quad \mathbf{v}.array[i]\ );$	(2)
$\mathbf{w}.\text{size} \leftarrow \max(\ \mathbf{u}.\text{size}, \mathbf{v}.\text{size}\ );$	(3)

Nach Ausführung gilt $\mathbf{w}.\text{size} = \max(\ \mathbf{u}.\text{size}, \mathbf{v}.\text{size}\ )$.
Der Algorithmus benötigt Laufzeit $O(\ N\ )$; es werden N Zuweisungs– und N XOR–Operationen durchgeführt, außerdem noch ein Vergleich und eine Zuweisung außerhalb der Schleife. Man hat also insgesamt

$$c_{SV} = 2 \cdot (\ N + 1\ )$$

elementare Operationen.

6.2.2 Konkatenation von Vektorblöcken

Seien $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke. Es soll die Operation

$$\mathbf{w} = \mathbf{u} \mid \mathbf{v}$$

ausgeführt werden.
Es muß gelten, daß $\mathbf{u}.\text{size} + \mathbf{v}.\text{size} \leq$ `WordSize` ist.
Diese Operation erledigt der folgende Algorithmus :

FOR $i \leftarrow 0$ TO N	(1)
$s \leftarrow (\ \mathbf{u}.array[i] \cdot 2^{\mathbf{v}.size}\ );$	(2)
$\mathbf{w}.array[i] \leftarrow (\ s \quad \text{OR} \quad \mathbf{v}.array[i]\ );$	(3)

Nach Ausführung gilt $\mathbf{w}.size = \mathbf{u}.size + \mathbf{v}.size$.
Der Algorithmus benötigt Laufzeit $O(\ N\ )$; es werden $2 \cdot N$ Zuweisuns–, und N Bitverschiebe– und OR–Operationen durchgeführt. (Multiplikation mit $2^i, 0 \leq i <$ `WordSize`, kann durch eine Bitverschiebeoperation berechnet werden). Man hat also insgesamt

$$c_{KV} = 4 \cdot N + 1$$

elementare Operationen ausgeführt.

6.2.3 Dekonkatenation eines Vektorblocks

Seien $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke, sei s eine ganze Zahl mit $0 \leq s <$ `WordSize`. Der Vektorblock $\mathbf{u}$ soll in zwei Vektorblöcke $\mathbf{v}$ und $\mathbf{w}$ zerlegt werden, wobei

$$\begin{aligned} \mathbf{v}.size &= s, \\ \mathbf{w}.size &= \mathbf{u}.size - s, \quad \text{und} \\ (\,\mathbf{v} \mid \mathbf{w}\,) &= \mathbf{u}. \end{aligned}$$

Vor Ausführung des Algorithmus muß $s \leq \mathbf{u}.size$ gelten.

$\mathbf{v}.size \leftarrow s$; $\mathbf{w}.size \leftarrow (\,\mathbf{u}.size - s\,)$;	(1)
/* berechne Bitmaske m, bei der nur die rechten $\mathbf{w}.size$ Bits gesetzt sind */ $m \leftarrow (\ \text{NOT } 0\,)/(\,2^{\mathbf{w}.size}\,)$;	(2)
FOR $i \leftarrow 0$ TO N	(3)
$\mathbf{w}.array[i] \leftarrow (\,\mathbf{u}.array[i]\ \ \text{AND}\ \ m\,)$; $\mathbf{v}.array[i] \leftarrow (\,\mathbf{u}.array[i]/(\,2^{\mathbf{w}.size}\,)\,)$;	(4)

Nach Ausführung gilt $\mathbf{v}.size = s$, $\mathbf{w}.size = \mathbf{u}.size - s$ und $(\,\mathbf{v} \mid \mathbf{w}\,) = \mathbf{u}$.
Der Algorithmus benötigt Laufzeit $O(\,N\,)$. In der Hauptschleife werden insgesamt $2 \cdot N$ Zuweisungs– und N Bitverschiebungs– und AND–Operationen verwendet; außerhalb dieser Schleife werden noch 3 Zuweisungs–, eine Subtraktions–, eine NOT– und eine Bitverschiebeoperation benutzt, denn das Berechnen dieser Bitmaske benötigt eine NOT– und eine Bitverschiebeoperation, die Berechnung der Division durch $2^i, 0 \leq i <$ `WordSize` kann ebenfalls durch eine Bitverschiebeoperation erledigt werden. Man hat insgesamt

$$c_{DV} = 4 \cdot N + 6$$

elementare Operationen ausgeführt.

6.2.4 Multiplikation von Vektorblöcken

Seien $\mathbf{u}$ und $\mathbf{v}$ Vektorblöcke, α eine kleine Matrix. Es soll die Operation

$$\alpha = \mathbf{u}^{\mathbf{T}}\mathbf{v}$$

ausgeführt werden.
Diese Operation wird von dem folgenden Algorithmus ausgeführt :

setze α auf 0;	(1)
FOR $j \leftarrow 0$ TO $\mathbf{u}.size$	(2)
$\quad s \leftarrow 0;$	(3)
$\quad$ FOR $i \leftarrow 0$ TO N	(4)
$\quad\quad$ IF j-tes Bit von $\mathbf{u}.array[i]$ gesetzt	(5)
$\quad\quad$ THEN $s \leftarrow (\ s \quad \text{XOR} \quad \mathbf{v}.array[i]\ );$	(6)
$\quad \alpha.matrix[j] \leftarrow s;$	(7)
$\alpha.rows \leftarrow \mathbf{u}.size;$ $\alpha.columns \leftarrow \mathbf{v}.size$	(8)

Nach Ausführung gilt $\alpha.rows = \mathbf{u}.size$ und $\alpha.columns = \mathbf{v}.size$.
Der Algorithmus benötigt Laufzeit $O(\,\texttt{WordSize}\cdot N\,)$. In der inneren Schleife werden insgesamt N BitTest und höchstens N Zuweisungs– und XOR–Operationen verwendet, jeder Vergleich führt eine BitTest–Operation (die man z.B. mit einer AND-Operation und einem Vergleich berechnen kann) aus. Diese Schleife wird in der äußeren Schleife $\mathbf{u}.size \leq \texttt{WordSize}$–mal durchlaufen. Zusätzlich hat man in der äußeren Schleife insgesamt noch $2\cdot\mathbf{u}.size$ Zuweisungsoperationen. Das Löschen einer kleinen Matrix benötigt außerdem `WordSize` Zuweisungsoperationen. Also hat man höchstens

$$c_{MV} = \texttt{WordSize}\cdot(\,4\cdot N + 1\,)$$

Operationen.

6.2.5 Größe eines Vektorblocks

Sei $\mathbf{v}$ ein Vektorblock.
Es soll die Größe von $\mathbf{v}$ ermittelt werden, d.h. die Anzahl Vektoren aus $GF(2)^N$, die in $\mathbf{v}$ enthalten sind.

RETURN($\mathbf{v}$.size);	(1)

Der Algorithmus benötigt Laufzeit $O(\,1\,)$.

6.2.6 Kopieren eines Vektorblocks

Seien **u** und **v** Vektorblöcke.
Es soll **u** auf **v** kopiert werden.

FOR $i \leftarrow 0$ TO N		(1)
	$\mathbf{v}.array[i] \leftarrow \mathbf{u}.array[i]$;	(2)
$\mathbf{v}.size \leftarrow \mathbf{u}.size$;		(3)

Nach Ausführung gilt $\mathbf{u} = \mathbf{v}$.
Der Algorithmus benötigt Laufzeit $O(\ N\ )$; es werden $c_{CV} = N + 1$ Zuweisungsoperationen durchgeführt.

6.2.7 Testen eines Vektorblocks

Sei **v** ein Vektorblock.
Es soll getestet werden, ob **v** leer ist, d.h. $\mathbf{v}.size = 0$.

IF $\mathbf{v}.size = 0$		(1)
THEN	RETURN(" Vektor ist leer ");	(2)
ELSE	RETURN(" Vektor ist nicht leer ");	(3)

Der Algorithmus benötigt Laufzeit $O(\ 1\ )$.

6.3 Einfache Operationen mit kleinen Matrizen

Dieser Abschnitt beschreibt alle einfachen Operationen auf kleinen Matrizen. Dieses sind im einzelnen :

- Subtraktion von kleinen Matrizen
- Konkatenation von kleinen Matrizen
- Dekonkatenation von kleinen Matrizen
- Multiplikation von kleinen Matrizen

- Feststellen der Dimension einer kleinen Matrix
- Kopieren einer kleinen Matrix
- Transponieren einer kleinen Matrix
- Gauss-Elimination auf kleinen Matrizen und ihre Anwendung auf das Invertieren einer kleinen Matrix

6.3.1 Subtraktion von kleinen Matrizen

Seien α, β und γ kleine Matrizen. Es soll die Operation

$$\gamma = \alpha - \beta$$

ausgeführt werden.
Es muß gelten, daß $\alpha.columns = \beta.columns$ und $\alpha.rows = \beta.rows$ ist.
Dies erledigt der folgende Algorithmus :

FOR $i \leftarrow 1$ TO `WordSize`	(1)
$\gamma.matrix[i] \leftarrow (\ \alpha.matrix[i]$ XOR $\beta.matrix[i]\ )$;	(2)
$\gamma.columns \leftarrow \alpha.columns$; $\gamma.rows \leftarrow \alpha.rows$;	(3)

Nach Ausführung gilt $\alpha.rows = \gamma.rows$ und $\alpha.columns = \gamma.columns$.
Der Algorithmus benötigt Laufzeit $O($ `WordSize` $)$.
In der Schleife werden zusammen `WordSize` Zuweisungs– und XOR–Operationen, außerhalb der Schleife noch zusätzlich 2 Zuweisungsoperationen ausgeführt, also insgesamt

$$c_{SM} = 2 \cdot (\ \texttt{WordSize} + 1\ )$$

elementare Operationen.

6.3.2 Konkatenation von kleinen Matrizen

Seien α, β und γ kleine Matrizen. Es soll die Operation

$$\gamma = (\ \alpha \mid \beta\ )$$

ausgeführt werden.
Es muß gelten : $\alpha.columns + \beta.columns \leq$ `WordSize`.

FOR $i \leftarrow 1$ TO `WordSize`	(1)
$s \leftarrow (\alpha.matrix[i] \cdot 2^{\beta.columns})$; $\gamma.matrix[i] \leftarrow (s \quad \text{OR} \quad \beta.matrix[i])$;	(2)
$\gamma.columns \leftarrow (\alpha.columns + \beta.columns)$; $\gamma.rows \leftarrow \max(\alpha.rows, \beta.rows)$;	(3)

Es gilt $\gamma.columns = \alpha.columns + \beta.columns$ und $\gamma.rows = \max(\alpha.rows, \beta.rows)$. Der Algorithmus benötigt Laufzeit $O($ `WordSize` $)$.
In der Schleife werden $2 \cdot$ `WordSize` Zuweisuns–, und `WordSize` Bitverschiebungs– und OR–Operationen durchgeführt (die Multiplikation mit $2^{\beta.columns}$ kann durch eine Bitverschiebeoperation erledigt werden), außerhalb der Schleife werden noch 2 Zuweisungs–, eine Vergleichs– und eine Additionsoperation durchgeführt, also insgesamt $c_{KM} = 4 \cdot ($ `WordSize` $+ 1)$ elementare Operationen.

6.3.3 Dekonkatenation einer kleinen Matrix

Seien α, β und γ kleine Matrizen, sei s eine ganze Zahl mit $0 \leq s <$ `WordSize`. Die kleine Matrix α soll in zwei kleine Matrizen β und γ zerlegt werden, wobei

$$\begin{aligned} \beta.columns &= s, \\ \gamma.columns &= \alpha.columns - s, \quad \text{und} \\ (\beta \mid \gamma) &= \alpha. \end{aligned}$$

Vor Ausführung des Algorithmus muß $s \leq \alpha.columns$ gelten.

$\beta.columns \leftarrow s$; $\gamma.columns \leftarrow \alpha.columns - s$;	(1)
$\beta.rows \leftarrow \alpha.rows$; $\gamma.rows \leftarrow \alpha.rows$;	(2)
/* berechne Bitmaske m, bei der nur die rechten $\alpha.columns$ Bits gesetzt sind */ $m \leftarrow (\text{ NOT } 0)/(2^{\alpha.columns})$;	(3)
FOR $i \leftarrow 0$ TO $\alpha.rows$	(4)
$\gamma.matrix[i] \leftarrow (\alpha.matrix[i] \quad \text{AND} \quad m)$; $\beta.matrix[i] \leftarrow (\alpha.matrix[i]/(2^{\gamma.columns}))$;	(5)

Nach Ausführung gilt $\beta.columns = s$, $\gamma.columns = \alpha.columns - s$, $\alpha.rows = \beta.rows = \gamma.rows$ und und $(\beta \mid \gamma) = \alpha$.

Der Algorithmus benötigt Laufzeit $O($ `WordSize` $)$. In der Hauptschleife werden insgesamt $2 \cdot$ `WordSize` Zuweisungs– und `WordSize` Bitverschiebungs– und AND–Operationen verwendet; außerhalb dieser Schleife werden noch 5 Zuweisungs–, eine Subtraktions–, eine NOT– und eine Bitverschiebeoperation benutzt, denn das Berechnen dieser Bitmaske benötigt eine NOT– und eine Bitverschiebeoperation, die Berechnung der Division durch $2^i, 0 \leq i <$ `WordSize` kann ebenfalls durch eine Bitverschiebeoperation erledigt werden. Man hat insgesamt

$$c_{DM} = 4 \cdot \texttt{WordSize} + 8$$

elementare Operationen ausgeführt.

6.3.4 Multiplikation von kleinen Matrizen

Seien α, β und γ kleine Matrizen. Es soll die Operation

$$\gamma = \alpha \cdot \beta$$

ausgeführt werden.
Es muß vor Ausführung gelten, daß $\alpha.columns = \beta.rows$ ist.

FOR $i \leftarrow 1$ TO $\alpha.rows$			(1)
	$s \leftarrow 0$;		(2)
	FOR $j \leftarrow 1$ TO $\alpha.columns$		(3)
		IF j-tes Bit von $\alpha.matrix[i]$ gesetzt	(4)
		THEN $s \leftarrow ($ s XOR $\beta.matrix[j]$ $)$;	(5)
	$\gamma.matrix[i] \leftarrow s$;		(6)
$\gamma.rows \leftarrow \alpha.rows$; $\gamma.columns \leftarrow \beta.columns$;			(7)

Nach Ausführung gilt $\gamma.rows = \alpha.rows$ und $\gamma.columns = \beta.columns$.
Der Algorithmus benötigt Laufzeit $O($ $\texttt{WordSize}^2$ $)$. In der inneren Schleife werden zusammen $\alpha.columns$ BitTest– und höchstens $\alpha.columns$ Zuweisungs– und XOR-operationen ausgeführt. Diese Schleife wird $\alpha.rows$ mal in der äußeren Schleife ausgeführt, mit zusätzlichen $2 \cdot \alpha.rows$ Zuweisungsoperationen. Schließlich werden außerhalb der beiden Schleifen noch 2 Zuweisungsoperationen ausgeführt.
Also hat man insgesamt höchstens

$$c_{MM} = \alpha.rows \cdot (2 + 3 \cdot \alpha.columns) + 2$$

elementare Operationen.

6.3.5 Dimensionen einer kleinen Matrix

Sei α eine kleine Matrix, *modus* eine ganze Zahl.
Es soll ihre Anzahl Zeilen bzw. Anzahl Spalten ermittelt werden.
modus muß entweder einen symbolischen Wert für Zeilen oder einen symbolischen Wert für Spalten enthalten.

IF *modus* enthält den symbolischen Wert für Zeilen		(1)
THEN	RETURN($\alpha.rows$);	(2)
ELSE	RETURN($\alpha.columns$);	(3)

Der Algorithmus benötigt Laufzeit $O(1)$.

6.3.6 Kopieren einer kleinen Matrix

Seien α und β kleine Matrizen. Setze $k = \alpha.rows$.
Es soll α auf β kopiert werden.

FOR $i \leftarrow 0$ TO k		(1)
	$\beta.matrix[i] \leftarrow \alpha.matrix[i]$;	(2)
$\beta.rows \leftarrow \alpha.rows$; $\beta.columns \leftarrow \alpha.columns$;		(3)

Nach Ausführung gilt $\alpha = \beta$.
Der Algorithmus benötigt Laufzeit $O($ `WordSize` $)$; es werden $k + 2$ Zuweisungsoperationen vorgenommen, und es gilt immer $0 \leq k \leq$ `WordSize`. Man hat also insgesamt

$$c_{CM} = k + 2$$

elementare Operationen.

6.3.7 Transponieren einer kleinen Matrix

Seien α und β kleine Matrizen. Setze $k = \alpha.rows$ und $l = \alpha.columns$. Es soll die Operation

$$\beta = \alpha^{\mathbf{T}}$$

ausgeführt werden.

```
FOR  i ← 0 TO l                                                   (1)
    s ← 0;                                                        (2)
    /* isoliere i–te Spalte von α */                              (3)
    FOR  j ← 0 TO k                                               (4)
        bit ← ( isoliere i–tes Bit von α.matrix[j] );             (5)
        t ← ( bit · 2^j );                                        (6)
        s ← ( s  OR  t );
    /* die i–te Spalte von α wird zur i–ten Zeile von β */        (7)
    β.matrix[i] ← s;                                              (8)
β.columns ← α.rows;                                               (9)
β.rows ← α.columns;
```

Nach Ausführung gilt $\beta.columns = \alpha.rows$ und $\beta.rows = \alpha.columns$.
Der Algorithmus benötigt Laufzeit $O(\texttt{WordSize}^2)$.
Das Isolieren eines Bits benötigt höchstens eine Bitverschiebe– und eine AND–Operation. Also werden in der inneren Schleife zusammen $3k$ Zuweisungs–, k OR– und höchstens $2k$ Bitverschiebe– und k AND– Operationen ausgeführt. Diese Schleife wird in der äußeren Schleife l–mal durchlaufen. Zusätzlich werden hier zusammen $2l$ Zuweisungsoperationen, und außerhalb der Schleifen noch 2 Zuweisungsoperationen ausgeführt, also insgesamt

$$c_{TM} = l \cdot (7k + 2) + 2$$

elementare Operationen. Natürlich gilt auch hier wieder $0 \leq k, l \leq \texttt{WordSize}$.

6.3.8 Gauss-Elimination auf kleinen Matrizen

In diesem Abschnitt werden die folgenden Teilalgorithmen beschrieben :

- Invertieren einer kleinen Matrix
- Spaltenvertauschung
- Rückwärtssubstitution

Invertieren einer kleinen Matrix :
Seien α und β kleine Matrizen. Es soll die Operation

$$\beta = \alpha^{-1}$$

ausgeführt werden. Sei $k = \alpha.rows$.
Vor Ausführung muß gelten, daß α regulär ist.

Der verwendete Algorithmus ist ein für diese Datenstruktur umgeschriebener Gauss-Algorithmus mit vollständiger Pivotierung (vgl. [14] oder [16]). Dieser wurde ausgewählt, da er für Aufgaben dieser Größenordnung ($k <$ `WordSize`) den geringsten tatsächlichen Zeitverbrauch und Speicherplatzbedarf hat.

setze β auf Einheitsmatrix; (1)
erzeuge h als Kopie von α;

FOR $i \leftarrow 0$ TO k (2)
- /* suche Pivotelement */ (3)
- FOR $j \leftarrow i$ TO k (4)
  - IF i-tes Bit in Zeile j gesetzt (5)
  - THEN $pivot \leftarrow j$; (6)
- IF keine Zeile gefunden, in der das i–te Bit gesetzt ist (7)
- THEN RETURN(Matrix ist singulär); (8)
- ELSE IF $pivot \neq i$ (9)
  - THEN /* Zeilenvertauschung : Dreieckstausch */ (10)
    tausche $h.matrix[pivot]$ und $h.matrix[i]$;
  - /* Elimination von Spalte i */ (11)
  - FOR $j \leftarrow 0$ TO k (12)
    - IF $j \neq i$ und i–tes Bit in der j–ten Zeile gesetzt (13)
    - THEN /* eliminiere i-tes Bit in Zeile j in h */ (14)
      $h.matrix[j] \leftarrow (h.matrix[j]$ XOR $h.matrix[i])$;
      /* führe analoge Operation mit β aus */ (15)
      $\beta.matrix[j] \leftarrow (\beta.matrix[j]$ XOR $\beta.matrix[i])$;

Der Algorithmus benötigt Laufzeit $O($ `WordSize`^{2}). Das Kopieren einer kleinen Matrix läuft in Zeit $O($ `WordSize`), ebenso das Setzen einer kleinen Matrix auf die

Einheitsmatrix. Die erste innere Schleife wird k–mal durchlaufen, bei jedem Durchlauf werden k BitTest– und höchstens k Zuweisungsoperationen verwendet. Die zweite innere Schleife wird ebenfalls k–mal durchlaufen; hier hat man k Vergleichs–, AND– und BitTestoperationen und höchstens $2 \cdot k$ Zuweisungs–, und XOR–Operationen.
Die äußere Schleife wird k–mal durchlaufen, hier werden zusätzlich zu den beiden inneren Schleifen bei jedem Durchlauf eine Vergleichs– und höchstens eine Rückgabe, eine Vergleichsoperation und eine Zeilenvertauschung vorgenommen. (die Zeilenvertauschung benutzt die Dreiecksvertauschung und damit 3 Zuweisungsoperationen). Insgesamt hat man also

$$c_{IM_1} = k \cdot (2 \cdot k + 7 \cdot k + 6) + 2 \cdot c_{CM}$$

elementare Operationen.

Die theoretische Verbesserung gegenüber dem Gauss–Algorithmus z.B über $\mathbb{Z}$ ist dadurch begründet, daß man bei Zeilenoperationen wie Addition oder Vertauschen zweier Zeilen die ganze Zeile simultan verarbeiten kann.
Dieser Algorithmus kann unter Verwendung einer Arithmetik für lange ganze Zahlen auch auf das Problem der Invertierung von(fast)beliebig großen (regulären) dichten $N \times N$–Matrizen über $GF(2)$ ausgedehnt werden.

Durch Weglassen von Schritt (13) kann man eine kleine $k \times l$ Matrix α, $k \leq l$ (β wird natürlich ebenfalls nicht verwendet) mit

$$c_{IM_2} = k \cdot (2 \cdot k + 5 \cdot k + 6) + c_{CM}$$

elementaren Operationen auf lineare Abhängigkeit testen, bzw. den Rang einer kleinen Matrix ermitteln (vgl. dazu auch [16] und [15]).

Eine leicht veränderte Version des obigen Algorithmus wird im Kapitel 6.5 öfter zum Lösen von kleinen homogenen linearen Gleichungssystemen verwendet. Dazu braucht man noch die Mechanismen der Rückwärtssubstitution bzw. der Spaltenvertauschung, die beide im folgenden beschrieben werden.

Mit nur geringen Veränderungen kann man mit diesen Mechanismen auch beliebige Gleichungssysteme in $GF(2)$ lösen. Dies wird z.B. verwendet, um die Lösungen des überbestimmten Gleichungssystems 3.60 (vgl. S. 36) zu bestimmen.

Spaltenvertauschung :
Der folgende Teilalgorithmus ersetzt Schritt (7) in obigem Algorithmus. Er wird ausgeführt, wenn eine kleine Matrix, deren Spaltenvektoren linear abhängig sind, auf rechte obere Dreiecksform gebracht werden soll (vgl. auch [15] oder [5]).

/* suche Zeile $\neq 0$ in der Matrix */ $j \leftarrow i$;			(1)
WHILE $h.matrix[j] = 0$ und $j < k$			(2)
	inkrementiere j um 1;		(3)
IF Zeile $h.matrix[j] \neq 0$ gefunden			(4)
THEN	tausche Zeile j mit Zeile i;		(5)
	/* suche Spalte mit Wert $\neq 0$; durch den Zeilentausch weiß man, daß eine solche Spalte existiert */ $j \leftarrow i+1$;		(6)
	WHILE j–tes Bit in Zeile i nicht gesetzt		(7)
		inkrementiere j um 1;	(8)
	/* tausche Spalte i mit Spalte j, Spaltentausch bedeutet Umnummerierung der Variablen */		(9)
	FOR $l \leftarrow 0$ TO k		(10)
		tausche Wert von Bit i und Bit j in Zeile $h.matrix[l]$;	(11)
ELSE	/* Gestalt einer oberen Dreiecksmatrix erreicht */ RETURN(i);		(12)

Die Laufzeit dieses Teilalgorithmus ist $O($ `WordSize` $)$.
In der ersten Schleife werden höchstens $2 \cdot k$ Vergleiche und k Additionen vorgenommen. Das Suchen einer Spalte mit Wert $\neq 0$ in der zweiten Schleife benötigt höchstens k BitTest– und Additionsoperationen. Der eigentliche Spaltentausch in der dritten Schleife schließlich benötigt $2 \cdot k$ NOT–, und jeweils k OR– und AND–Operationen. (Das Vertauschen zweier Bits benötigt 2 NOT–, eine AND– und eine OR–Operation). Außerdem hat man noch höchstens 5 Zuweisungen und einen Vergleich, also insgesamt höchstens

$$c_{STM} = 6 + 3 \cdot k + 3 \cdot k + 4 \cdot k = 6 + 10 \cdot k$$

elementare Operationen.

Rückwärtssubstitution

Dieser Teilalgorithmus wird dann ausgeführt, wenn eine Matrix bereits rechte obere Dreiecksgestalt hat, und man die Lösungen des homogenen Gleichungssystems berechnen will (vgl. auch [15]). Mit ihm kann man einen Lösungsvektor des Systems bestimmen.

```
FOR j ← k DOWNTO Rang der Matrix                                  (1)
    bestimme j–tes Bit des Lösungsvektors zufällig;               (2)
FOR j ← Rang der Matrix DOWNTO 1                                  (3)
    l ← k;                                                        (4)
    bit ← 0;
    WHILE l > j                                                   (5)
        IF l–tes Bit des Lösungsvektors gesetzt                   (6)
        THEN IF l–tes Bit in Zeile j gesetzt                      (7)
             THEN IF bit = 0                                      (8)
                  THEN bit ← 1;                                   (9)
                  ELSE bit ← 0;                                   (10)
        dekrementiere l um 1;                                     (11)
    j –tes Bit des Lösungsvektors ← bit;                          (12)
```

Der Algorithmus hat Laufzeit $O(\,\texttt{WordSize}^2\,)$.
In der ersten Schleife hat man höchstens k-Zuweisungsoperationen.
Die innere WHILE-Schleife wird höchstens k–mal durchlaufen. In ihr werden bei jedem Durchlauf höchstens 3 Vergleiche, 2 Zuweisungen und eine Addition ausgeführt. Die äußere Schleife wird ebenfalls höchstens k–mal durchlaufen; hier werden die innere Schleife, 3 Zuweisungen, eine OR– und eine Bitverschiebeoperation durchgeführt. Die Anzahl aller bestimmten Bits des Lösungsvektors ist hierbei k. Man hat also insgesamt höchstens

$$c_{RM} = k \cdot (\,1 + 6k + 5\,)$$

elementare Operationen.

Aus diesen Ausführungen folgt direkt, daß man mit Hilfe des in diesem Abschnitt beschriebenen Gauss-Algorithmus einen Lösungsvektor eines kleinen homogenen Gleichungssystems in $GF(2)$ in Laufzeit $O(\,\texttt{WordSize}^2\,)$ mit höchstens

$$c_{GM} = c_{IM} + kc_{STM} + c_{RM}$$

elementaren Operationen bestimmen kann.

6.4 Gemischte Operationen

Dieser Abschnitt beschreibt alle einfachen Operationen mit gemischten Operanden. Es sind im einzelnen :

- Multiplikation eines Vektorblocks mit einer kleinen Matrix
- Multiplikation der großen dünnen Matrix mit einem Vektorblock, wenn jene symmetrisch bzw. asymmetrisch ist

6.4.1 Multiplikation eines Vektorblocks mit einer kleinen Matrix

Sein $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke, sei α eine kleine Matrix. Es soll die Operation

$$\mathbf{w} = \mathbf{v} \cdot \alpha$$

ausgeführt werden.
Setze $k = \alpha.rows$. Vor Ausführung des Algorithmus muß gelten, daß $\alpha.rows = \mathbf{v}.size$ ist.

FOR $i \leftarrow 0$ TO N				(1)
	$s \leftarrow 0;$			(2)
	FOR $j \leftarrow 0$ TO k			(3)
		IF j–tes Bit in $\mathbf{v}.array[i]$ gesetzt		(4)
		THEN	$s \leftarrow (\ s$ XOR $\alpha.matrix[j]\ );$	(5)
	$\mathbf{w}.array[i] \leftarrow s;$			(6)

Nach Ausführung des Algorithmus ist $\mathbf{w}.size = \alpha.columns$.
Der Algorithmus benötigt Laufzeit $O(N \cdot \texttt{WordSize})$; die innere Schleife führt zusammen k BitTest– und höchstens k AND– und Zuweisungsoperationen aus. Die äußere Schleife wird N–mal durchlaufen, hierbei werden die innere Schleife und zusätzlich 2 Zuweisungsoperationen durchgeführt; also hat man insgesamt

$$c_{VM} = N \cdot (\ 2 + 5 \cdot k\ )$$

elementare Operationen.

6.4.2 Multiplikation der großen dünnen Matrix mit einem Vektorblock

Sei A die große, dünne Matrix, und seien $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke. Es soll die Operation

$$\mathbf{w} = A\mathbf{v}$$

ausgeführt werden.
Ist A symmetrisch, geschieht dies durch folgenden Algorithmus :

FOR $i \leftarrow 0$ TO N (1)

- $s \leftarrow 0;$ (2)
  $g_i \leftarrow A.row[i].size;$
- FOR $j \leftarrow 0$ TO g_i (3)
  - $h \leftarrow A.row[i].column[j];$ (4)
    $s \leftarrow (\ s$ XOR $\mathbf{v}.array[h]\ );$
- $\mathbf{w}.array[i] \leftarrow s;$ (5)

Nach Beendigung des Algorithmus ist $\mathbf{w}.size = \mathbf{v}.size$.
Dieser Algorithmus benötigt Laufzeit $O(\ N + T\ )$. Die äußere Schleife wird N–mal durchlaufen, in jedem Durchlauf werden 3 Zuweisungsoperationen und die innere Schleife durchlaufen. In der inneren Schleife hat man 2 Zuweisungs– und eine XOR–Operation.
Nun ist g_i die Anzahl der Einträge $\neq 0$ in Zeile i, also gilt :

$$\sum_{i=0}^{N} g_i = T$$

Also hat man insgesamt

$$c_{D\ddot{U}_1} = 3 \cdot N + 2 \cdot (\ N + T\ )$$

elementare Operationen.

Ist A dagegen asymmetrisch, setzt man $B = A \cdot A^{\mathbf{T}}$ und berechnet

$$\mathbf{w} = B\mathbf{v}.$$

Hierbei wird jedoch die Matrix B niemals explizit ausgerechnet oder sogar gespeichert, da sie u.U. viel dichter als A ist. Vielmehr wird zuerst die Operation $A^{\mathbf{T}}\mathbf{v}$ ausgeführt, dann wird die Matrix A mit dem Ergebnis dieser Operation multipliziert.
Dies geschieht durch folgenden Algorithmus :

/* erster Teil : berechne $\mathbf{h} = A^{\mathbf{T}}\mathbf{v}$ */ initialisiere Hilfsvektorblock $\mathbf{h}$;	(1)
FOR $i \leftarrow 0$ TO N	(2)
$s \leftarrow 0$; $g \leftarrow A.row[i].size$;	(3)
FOR $j \leftarrow 0$ TO g	(4)
$f \leftarrow A.row[i].column[j]$;	(5)
$\mathbf{h}.array[f] \leftarrow (\ \mathbf{h}.array[f]$ XOR $\mathbf{v}.array[i]\ )$;	(6)
/* zweiter Teil : berechne $\mathbf{w} = A\mathbf{h}$ */	(7)
FOR $i \leftarrow 0$ TO N	(8)
$s \leftarrow 0$; $g \leftarrow A.row[i].size$;	(9)
FOR $j \leftarrow 0$ TO g	(10)
$f \leftarrow A.row[i].column[j]$;	(11)
$s \leftarrow (\ s$ XOR $\mathbf{h}.array[f]\ )$;	(12)
$\mathbf{w}.array[i] \leftarrow s$;	(13)

Nach Beendigung des Algorithmus ist wieder $\mathbf{w}.size = \mathbf{v}.size$.
Dieser Algorithmus benötigt ebenfalls Laufzeit $O(\ N + T\ )$. Die äußere Schleife wird N–mal durchlaufen, in jedem Durchlauf werden 2 Zuweisungsoperationen und die innere Schleife durchlaufen. In der inneren Schleife hat man 2 Zuweisungs– und eine XOR–Operation.
Nun ist g_i die Anzahl der Einträge $\neq 0$ in Zeile i, also gilt wieder :

$$\sum_{i=0}^{N} g_i = T$$

Also hat man insgesamt

$$2 \cdot N + 2 \cdot (\ N + T\ )$$

elementare Operationen. Der zweite Teil des Algorithmus benötigt, wie oben gesehen, Laufzeit $O(\ N + T\ )$; das Initialisieren eines Vektorblocks schließlich läuft in konstanter Zeit und besteht im wesentlichen aus der Reservierung eines hinreichend großen Speicherblockes. Man hat also

$$c_{D\ddot{U}_2} = c_{D\ddot{U}_1} + 2 \cdot N + 2 \cdot (\ N + T\ )$$

elementare Operationen.

6.5 Spezielle Operationen

Dieser Abschnitt enthält die Beschreibung derjenigen Algorithmen, die spezielle Operationen auf den Datentypen ausführen. Spezielle Operationen sind solche, die nur an einer Stelle des Block Lanczos Algorithmus vorkommen. Diese Operationen wurden jeweils in eigenen Funktionen implementiert.

Diese Operationen sind im einzelnen :

- Berechnung von η_{n+1}
- Berechnung von τ_{n+1}
- Test auf lineare Abhängigkeit von $\mathbf{r}_{n+1}$ und Berechnung von ψ_{n+1}
- Test von $\mathbf{r}_{n+1}$ auf lineare Abhängigkeit
- Berechnung von ψ_{n+1}
- Berechnung des Resultats im asymmetrischen Fall
- Berechnung von σ_n
- Berechnung von ω_n
- Berechnung von ρ_n
- Löschen von überflüssigen Vektoren aus den Vektorblöcken
- Berechnung von proj()–Matrizen durch Aufspalten

6.5.1 Berechnung von η_{n+1}

Seien α, β und η_{n+1} kleine Matrizen, $\mathbf{w}_{n+1}$ und $\mathbf{r}_n$ Vektorblöcke. Es soll die kleine Matrix η_{n+1} berechnet werden mit

$$\eta_{n+1} = \begin{pmatrix} 0 & \alpha^{\mathbf{T}} \\ \alpha & \beta \end{pmatrix},$$

wobei gilt :

$$\begin{aligned} \alpha &= \mathbf{w}_{n+1}^{\mathbf{T}} \mathbf{r}_n, \\ \beta &= \mathbf{w}_{n+1}^{\mathbf{T}} \mathbf{w}_{n+1} \end{aligned}$$

also ist β symmetrisch (vergleiche auch S. 40), und $\alpha.rows = \beta.rows$.
Setze $k = \eta_{n+1}.rows$, $\delta = \alpha^{\mathbf{T}}, d = \delta.rows = \alpha.columns$.

berechne $\delta \leftarrow \alpha^{\mathbf{T}}$; /* da $\delta = \alpha^{\mathbf{T}}$, ist $\delta.rows = \alpha.columns$ und $\delta.columns = \alpha.rows$ */	(1)
$\eta_{n+1}.columns \leftarrow (\ \alpha.columns + \beta.columns\ )$; $\eta_{n+1}.rows \leftarrow (\ \alpha.columns + \beta.rows\ )$;	(2)
FOR $i \leftarrow 0$ TO d	(3)
$\eta_{n+1}.matrix[i] \leftarrow \delta.matrix[i]$;	(4)
FOR $i \leftarrow d$ TO k	(5)
$s \leftarrow (\ \alpha matrix[i-d] \cdot 2^{\beta.columns}\ )$; $eta_{n+1}.matrix[i] \leftarrow (\ s$ XOR $\beta.matrix[i-d]\ )$;	(6)

Nach Ende des Algorithmus gilt :
$\eta_{n+1}.columns = \alpha.columns + \beta.columns$ und $\eta_{n+1}.rows = \alpha.columns + \beta.rows$. Weiterhin ist η_{n+1} symmetrisch, da $\mathbf{w}_{n+1}^{\mathbf{T}}\mathbf{w}_{n+1}$ symmetrisch ist.
Der Algorithmus benötigt Laufzeit $O(\ \texttt{WordSize}^2\ )$; denn das Transponieren einer $k\times l$–Matrix benötigt Zeit $O(\texttt{WordSize}^2)$, und die beiden Schleifen werden zusammen $k \leq$ `WordSize`– mal durchlaufen; In der ersten Schleife hat man d Zuweisungen, in der zweiten Schleife hat man $2 \cdot (\ k-d\ )$ Zuweisungen und Subtraktionen und $k-d$ Bitverschiebe– und XOR–Operationen. Zusätzlich hat man noch 2 Zuweisungsoperationen, also insgesamt

$$c_\eta = c_{TM} + 2 + d + 6 \cdot (\ k-d\ )$$

elementare Operationen.

6.5.2 Berechnung von τ_{n+1}

Seien η_{n+1} und τ_{n+1} kleine Matrizen. Setze $k = \eta_{n+1}.rows$.
Es soll τ_{n+1} berechnet werden mit

$$\tau_{n+1}^{\mathbf{T}}\eta_{n+1}\tau_{n+1} = \begin{pmatrix} \delta & 0 \\ 0 & 0 \end{pmatrix}.$$

und δ ist reguläre Matrix vom selben Rang wie η_{n+1}.
Die bestimmende Gleichung für τ_{n+1} ist :

$$\eta_{n+1}\tau_{n+1} = (\ \nu \quad 0\ ).$$

wobei ν, die linken rang(η_{n+1}) linear unabhängigen Spaltenvektoren des Produkts darstellt, und die rechten Spalten des Matrizenproduktes Null sind. Ausserdem soll gelten : τ_{n+1} ist regulär.

$r \leftarrow$ rang(η_{n+1});				(1)
IF $r = k$				(2)
THEN	setze τ_{n+1} auf Einheitsmatrix;			(3)
ELSE	berechne eine Basis γ des Lösungsraums der Gleichung $\eta_{n+1} \cdot x = 0$ mit Gauss-Elimination;			(4)
	/* berechne eine Basis δ aus Einheitsvektoren des von η_{n+1} aufgespannten Raumes und füge diese zu γ hinzu */ setze δ auf leer;			(5)
		setze e_i auf i–ten Einheitsvektor in $GF(2)^k$;		(6)
		IF sind die Vektoren in ($e_i \mid \delta \mid \gamma$) linear unabhängig		(7)
		THEN	füge e_i zu δ hinzu;	(8)
		$i \leftarrow (i + 1)$;		(9)
	UNTIL ($\delta \mid \gamma$) ist regulär			(10)
	setze $\tau_{n+1} \leftarrow (\delta \mid \gamma)$;			(11)

Nach Ende des Algorithmus gilt : τ_{n+1} ist Basis von $GF(2)^k$. Weiterhin sind die obigen Gleichungen erfüllt, denn :
Es gilt $\eta_{n+1}\gamma = 0$ und $\nu = \eta_{n+1}\delta$ besteht aus rang(η_{n+1}) linear unabhängigen Spaltenvektoren.
Der Algorithmus benötigt Laufzeit $O($ `WordSize`^{3} $)$.
Die Berechnung des Ranges von η_{n+1} geschieht in $O($ `WordSize`^{2} $)$ (vgl. 73, oder [16]) durch Gauss-Elimination, ebenso das Berechnen eines Lösungsvektors dieses homogenen Systems. Eine Basis des Lösungsraums des Systems besteht hierbei aus $k - r$ Vektoren.
Die innere Schleife höchstens k–mal durchlaufen. Dabei erfolgen in jedem Durchlauf $k + 1$ Zuweisungen; das Hinzufügen eines Vektors zu einer Matrix benötigt k Zuweisungs–, OR– und Bitverschiebeoperationen, das Testen einer Matrix auf lineare Unabhängigkeit ebenfalls mit Gauss-Elimination Zeit $O($ `WordSize`^{2} $)$.
Also hat man insgesamt höchstens

$$c_\tau = c_{IM_2} + 1 + (k - r) \cdot c_{GM} + k \cdot (4k + 1 + c_{IM_2})$$

elementare Operationen.

6.5.3 Test auf lineare Abhängigkeit von $\mathbf{r}_{n+1}$ und Berechnung von ψ_{n+1}

Sei $\mathbf{r}_{n+1}$ ein Vektorblock, sei ψ_{n+1} eine kleine Matrix.
Sei k die Anzahl der Vektoren in $\mathbf{r}_{n+1}$. Dies führt zu folgendem Algorithmus :

IF $k \leq 1$		(1)
THEN	RETURN(" $\mathbf{r}_{n+1}$ linear unabhängig ");	(2)
ELSE	teste $\mathbf{r}_{n+1}$ auf lineare Abhängigkeit;	(3)
	berechne ψ_{n+1};	(4)

Die Gesamtlaufzeit des Algorithmus ist im besten Fall $O(\ 1\ )$, im schlechtesten Fall jedoch $O(\ N \cdot \texttt{WordSize}\ )$; man hat höchstens in diesem Fall

$$c_{VD} = c_{LA} + c_{\psi} + 1$$

elementare Operationen. Der Test auf lineare Abhängigkeit benötigt Zeit $O(\ N \cdot \texttt{WordSize}\ )$, die Berechnung von ψ_{n+1} braucht Zeit $O(\ \texttt{WordSize}^2\ )$, was beides in den folgenden Abschnitten noch ausführlicher dargestellt wird.

In den Abschnitten

1. Test von $\mathbf{r}_{n+1}$ auf lineare Abhängigkeit
2. Berechnung von ψ_{n+1}

werden die jeweiligen Teilalgorithmen beschrieben. Diese drei Teilalgorithmen wurden in einer einzigen Funktion realisiert und sind hier nur der Übersichtlichkeit wegen aufgespalten.

6.5.4 Test von $\mathbf{r}_{n+1}$ auf lineare Abhängigkeit

In folgendem Algorithmus wird getestet, ob die Vektoren in $\mathbf{r}_{n+1}$ linear abhängig oder linear unabhängig sind. Dieser Test benutzt ebenfalls einen etwas abgewandelten Gauss-Algorithmus.

- initialisiere Vektorblock **h**; kopiere $\mathbf{r}_{n+1}$ auf **h**; (1)
- FOR $i \leftarrow 0$ TO k (2)
  - suche Pivotelement in **h**; (3)
  - IF kein Pivotelement gefunden (4)
    - THEN IF Bit $j \neq 0$, d.h. Spalteneintrag $\neq 0$ in Zeile i (5)
      - THEN tausche Spalte j mit Spalte i; (6)
      - ELSE /* **h** hat die Gestalt einer rechten oberen Dreiecksmatrix, alle übrigen Zeilen sind 0 */ RETURN(" lineare Abhängigkeit entdeckt "); (7)
  - IF Pivotelement nicht in Zeile i gefunden (8)
    - THEN Zeilenvertauschung; (9)
  - /* Reduktion in **h** */ (10)
  - FOR $j \leftarrow 0$ TO N (11)
    - IF $j \neq i$ und i–tes Bit in der j–ten Zeile gesetzt (12)
      - THEN eliminiere i–tes Bit in Zeile j in **h**; (13)

Der Algorithmus benötigt Laufzeit $O(\ N \cdot \texttt{WordSize}\ )$; die äußere Schleife wird k–mal durchlaufen, und die Operationen Spaltentausch und Suche des Pivotelementes benötigen Zeit $O(\ N \cdot \texttt{WordSize}\ )$, ebenso die Reduktion in h, die explizit duch die innere Schleife dargestellt ist (vgl. wieder S. 73 ff.). Man hat also insgesamt höchstens

$$
\begin{aligned}
c_{LA} &= k \cdot (\ 2 \cdot N + 5 \cdot k + 6 + 10 \cdot N + 5 \cdot N\ ) + c_{CV} \\
&= k \cdot (\ 17 \cdot N + 5 \cdot k + 6\ ) + c_{CV}
\end{aligned}
$$

elementare Operationen.

6.5.5 Berechnung von ψ_{n+1}

Sind alle Vektoren in $\mathbf{r}_{n+1}$ linear unabhängig, so existiert nur eine Matrix ψ_{n+1} mit $\mathbf{r}_{n+1}\psi_{n+1} = 0$, nämlich $\psi_{n+1} = 0$.
Sind Vektoren in $\mathbf{r}_{n+1}$ dagegen linear abhängig, so wird eine Matrix ψ_{n+1} berechnet,

die die Bedingung

$$\mathbf{r}_{n+1}\psi_{n+1} = 0 \quad \text{mit} \quad \psi_{n+1} \neq 0$$

erfüllt (vergleiche S.41, oder [16]).
Nun folgt der Algorithmus zur Berechnung von ψ_{n+1}.

setze ψ_{n+1} auf Nullmatrix;		(1)
IF lineare Abhängigkeit im Vektorblock		(2)
THEN	setze ν auf die ersten k Zeilen von $\mathbf{h}$;	(3)
	berechne Lösungsvektor λ als Element des Lösungsraums von $\nu \cdot x = 0$ mit Rückwärtssubstitution /* ν wurde oben schon auf Dreiecksform gebracht */;	(4)
	betrachte λ als $1 \times k$-Matrix und setze $\psi_{n+1} \leftarrow \lambda$;	(5)

Nach Beendigung des Algorithmus gilt $\psi_{n+1}.columns = 1$ und $\psi_{n+1}.rows = k$.
Dieser Algorithmus benötigt insgesamt höchstens

$$c_\psi = 3 \cdot c_{CM} + 1 + c_{RM}$$

elementare Operationen und damit Laufzeit $O($ `WordSize`^{2} $)$ wegen der Rückwärtssubstitution (vgl. auch hierzu wieder S. 73 ff.).

6.5.6 Berechnung des Resultats im asymmetrischen Fall

Sei B die große dünne Matrix, $\mathbf{y}$ ein Vektorblock und λ eine kleine Matrix. Es soll der Vektorblock $\hat{\mathbf{z}}$ mit

$$\hat{\mathbf{z}} = \begin{pmatrix} B^{\mathbf{T}}\mathbf{y} \\ -\lambda \end{pmatrix}$$

berechnet werden. λ ist dabei die Lösung des Gleichungssystems $\mathbf{b}\lambda = \tilde{\mathbf{b}}$ und wird mit Gauss-Elimination berechnet (vgl. S. 36).

/* berechne zuerst $B^{\mathbf{T}}\mathbf{y}$ */			(1)
FOR $i \leftarrow 0$ TO N			(2)
	$s \leftarrow 0$; $g \leftarrow A.row[i].size$;		(3)
	FOR $j \leftarrow 0$ TO g		(4)
		$f \leftarrow A.row[i].column[j]$;	(5)
		$\hat{\mathbf{z}}.array[f] \leftarrow (\ \hat{\mathbf{z}}.array[f]$ XOR $\mathbf{y}.array[i]\ )$;	(6)
IF ursprüngliche Gleichung homogen			(7)
THEN vertikale Konkatenation von $\hat{\mathbf{z}}$ mit λ;			(8)

Ob die Matrix A symmetrisch war, wird vor Beginn des eigentlichen Lösungsalgorithmus in einer Boolschen Variablen gespeichert. Der obige Algorithmus braucht dann nur durchlaufen zu werden, wenn diese Variable nicht gesetzt ist. Ebenso wird vor Beginn des Algorithmus in einer Boolschen Variable gespeichert, ob das ursprüngliche Gleichungssystem homogen war— d.h. $\mathbf{b} = 0$, $\hat{B}$ wurde aufgespalten und deren rechten `WordSize` Spalten auf $\mathbf{b}$ gesetzt, um ein inhomogenes Gleichungssystem zu erhalten— oder ob das ursprüngliche Gleichungssystem bereits inhomogen war.
Der Algorithmus benötigt Laufzeit $O(\ N + T\ )$ wegen der Anwendung von $B^{\mathbf{T}}$ auf einen Vektorblock (siehe auch S. 79 f.). Die vertikale Konkatenation benötigt zusätzlich $l, 0 \leq l \leq$ `WordSize` Zuweisungen, also insgesamt

$$c_{AR} = 2 \cdot N + 2 \cdot (\ N + T\ ) + l$$

elementare Operationen.

6.5.7 Berechnung von σ_n

Seien α, β und σ_n kleine Matrizen. Setze $k = \alpha.columns + \beta.columns$.
Hierbei ist $\alpha = (\ \mathtt{Proj}(\ \mathbf{r}_{n+1}; A\mathbf{f}_{n-1}\ )$ und $\beta = \mathtt{Proj}(\ \mathbf{r}_{n+1}; A\mathbf{x}_n\ )$.
Berechne eine größte Matrix mit vollem Rang σ_n, sodaß

$$(\ \alpha \mid \beta\ )\sigma_n = 0.$$

Diese Gleichung stellt ein unterbestimmtes Gleichungssystem dar, d.h. ein Gleichungssystem mit $\alpha.rows$ Gleichungen und k Unbekannten, wobei $\alpha.rows \leq k$ gelten muss. Nun erweitert man dieses System auf ein Gleichungsystem mit k Gleichungen und k unbekannten, indem man die bisher nicht vorhandenen $k - \alpha.rows$ Gleichungen auf Null setzt.

Aus der linearen Algebra ist bekannt, daß der Rang des erweiterten Systems gleich dem Rang des ursprünglichen Systems ist(siehe [16]). Die Vektoren einer größten Matrix mit vollem Rang σ_n, die die ursprüngliche Gleichung lösen, bilden also eine Basis des von den Lösungsvektoren dieser erweiterten Gleichung aufgespannten Raumes (zum Beweiß vgl. wieder [16]).

IF $\texttt{size}(\ \mathbf{r}_{n+1}\ ) = 0$				(1)
THEN	setze $\sigma_n \leftarrow$ I;			(2)
ELSE	initialisiere h; berechne $h \leftarrow (\ \alpha \mid \beta\ )$;			(3)
	bringe h auf rechte obere Dreiecksform; /* dazu wird die Gauss-Elimination benutzt */			(4)
		bestimme einen möglichen Lösungsvektor durch Lösen des unterbestimmten Gleichungssystems aus (2);		(5)
		füge so gefundenen Vektor zu σ_n hinzu;		(6)
		IF so konstruiertes σ_n ist linear abhängig		(7)
		THEN	lösche den in (4) zugefügten Vektor wieder aus σ_n;	(8)
	UNTIL σ_n hat maximalen Rang r			(9)

Ist r der Rang der Matrix $h = (\ \alpha \mid \beta\ )$, so gilt $r \leq \alpha.rows \leq k$. Weiter ist die Dimension des Lösungsraums der homogenen Gleichung $h \cdot \sigma_n = 0$ $k - r$.
Der Algorithmus findet genau $k - r$ linear unabhängige Vektoren aus $GF(2)^k$. Also gilt : $\sigma_n.columns = k - r$ und $\sigma_n.rows = k$.
Der Algorithmus benötigt im schlechtesten Falle Laufzeit $O(\ \texttt{WordSize}^3\ )$ Dies kann nur dann auftreten, wenn $\texttt{size}(\ \mathbf{r}_{n+1}\ ) > 0$. Im besten Falle läuft der Algorithmus jedoch in Zeit $O(\ \texttt{WordSize}\ )$ (wenn $\texttt{size}(\ \mathbf{r}_{n+1}\ ) = 0$). Die Umformung der Matrix h auf obere Dreiecksform mit Gauss-Elimination benötigt Zeit $O(\ \texttt{WordSize}^2\ )$, ebenso die Bestimmung eines Lösungsvektors des Gleichungssystems mit Rückwärtssubstitution. Weiter benötigt das Testen von a Vektoren aus $GF(2)^k$, $a \leq k$, auf lineare Unabhängigkeit mit Gauss-Elimination ebenfalls Zeit $O(\ \texttt{WordSize}^2)$. Das Zufügen und Löschen eines Vektors zu bzw. aus einer kleinen Matrix führt jeweils `WordSize` Zuweisungs– Bitverschiebe– und OR–Operationen aus. Da eine Basis eines Unterraums des $GF(2)^k$ mit Dimension $k - r$ genau $k - r$ Elemente enthält, wird die Schleife auch so oft durchlaufen. Dies erreicht man, indem man die frei wählbaren Variablen des Gleichungssystems bei jedem Durchlauf der Schleife auf einen unterschiedlichen Einheitsvektor aus $GF(2^{k-r}$) setzt (siehe auch [15]). Man hat also insgesamt höchstens

$$c_\sigma = 2 \cdot c_{CM} + c_{KM} + c_{GM} + (\ k - r\ ) \cdot (\ c_{RM} + 3 \cdot \texttt{WordSize} + c_{IM_2}\ )$$

elementare Operationen.

6.5.8 Berechnung von ω_n

Seien σ_n und ω_n kleine Matrizen. Setze $k = \sigma_n.columns$ (vgl. voriger Abschnitt). Fülle σ_n zu einer Basis von $GF(2)^k$ auf und lasse die übrigen Spaltenvektoren die Matrix ω_n bilden.

<table>
<tr><td colspan="4">setze $\omega_n \leftarrow 0$;</td><td>(1)</td></tr>
<tr><td colspan="4">IF σ_n ist nicht quadratisch</td><td>(2)</td></tr>
<tr><td rowspan="8">THEN</td><td colspan="3">initialisiere Matrix h;
kopiere σ_n auf h;</td><td>(3)</td></tr>
<tr><td rowspan="6"></td><td colspan="2">bestimme Einheitsvektor e_i;</td><td>(4)</td></tr>
<tr><td colspan="2">füge so gefundenen Vektor zu h hinzu;</td><td>(5)</td></tr>
<tr><td colspan="2">IF ist die so konstruierte Matrix h linear abhängig</td><td>(6)</td></tr>
<tr><td>THEN</td><td>lösche den in (2) zugefügten Vektor aus h;</td><td>(7)</td></tr>
<tr><td>ELSE</td><td>füge den Einheitsvektor e_i zu ω_n hinzu;</td><td>(8)</td></tr>
<tr><td colspan="2">erhöhe i;</td><td>(9)</td></tr>
<tr><td colspan="3">UNTIL h hat Rang k</td><td>(10)</td></tr>
</table>

Obiger Algorithmus ergänzt σ_n zu einer Basis von $GF(2)^k$. Ist $l = \sigma_n.columns$ der Rang von σ_n, so findet obiger Algorithmus weiterhin $k - l$ Einheitsvektoren aus $GF(2)^k$, die zur Matrix ω_n zusammengefaßt werden.
Der Algorithmus benötigt im besten Fall Laufzeit $O(\texttt{WordSize})$, im schlechtesten Fall aber Laufzeit $O(\texttt{WordSize}^3)$. Das Testen von a Vektoren aus $GF(2)^k$, $a \leq k$, auf lineare Unabhängigkeit benötigt mit Gauss-Elimination Zeit $O(\texttt{WordSize}^2)$, das Löschen bzw. Hinzufügen eines Vektors aus $GF(2)^k$ aus bzw. zu einer kleinen Matrix benötigt Zeit $O(\texttt{WordSize})$. Es sind höchstens k Einheitsvektoren zu testen, also wird die Schleife höchstens k–mal durchlaufen.
Man hat also insgesamt höchstens

$$c_\omega = c_{CM} + k \cdot (k + 3 \cdot \texttt{WordSize} + c_{IM_2})$$

elementare Operationen.

6.5.9 Berechnung von ρ_n

Seien α, ρ_n, σ_n und ω_n kleine Matrizen. Sei $\mathbf{x}_n$ ein Vektorblock
Setze $s = \sigma_n.rows$.
Berechne die Matrix ρ_n mit

$$\sigma_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \alpha^{-1} \end{pmatrix} \rho_n = 0, \qquad \omega_n^{\mathbf{T}} \begin{pmatrix} I & 0 \\ 0 & \alpha^{-1} \end{pmatrix} \rho_n = I.$$

Also gilt

$$\rho_n = \begin{pmatrix} I & 0 \\ 0 & \alpha \end{pmatrix} \cdot \begin{pmatrix} \sigma_n^{\mathbf{T}} \\ \omega_n^{\mathbf{T}} \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 \\ I \end{pmatrix}.$$

Hierbei ist $\alpha = \xi_n^{-1} = (\ \mathbf{x}_n^{\mathbf{T}} \mathbf{x}_n\ )^{-1}$ und wurde schon vorberechnet.

initialisiere kleine Matrix h;	(1)
bilde $\iota \leftarrow (\ \sigma_n \mid \omega_n\ )$;	(2)
bilde $\nu \leftarrow \iota^{\mathbf{T}}$;	(3)
invertiere ν;	(4)
setze h auf die Einheitsmatrix aus $GF(2)^{s \times s}$; setze die unteren α.rows Zeilen von h auf α;	(5)
multipliziere die Teilergebnisse aus (4) und (5);	(6)
setze $k \leftarrow \omega_n.columns$; setze h auf die Einheitsmatrix aus $GF(2)^{k \times k}$; verschiebe h um $\sigma_n.columns$ Zeilen nach unten; setze oberen $\sigma_n.columns$ Zeilen auf 0;	(7)
multipliziere die Teilergebnisse aus (6) und (7); setze ρ_n auf die erhaltene Matrix;	(8)

Aus obigen Gleichungen (und dem Algorithmus) folgt, daß $\rho_n.rows = s$ und $\rho_n.columns = \omega_n.columns$. Da $\gamma = (\ \sigma_n \mid \omega_n\ )$ Basis von $GF(2)^s$, ist γ regulär, also invertierbar. Operation (5) ergibt genau die erwünschte erste Teilmatrix wegen der Anordnung der Matrizen im Speicher.
Der Algorithmus benötigt Laufzeit $O(\texttt{WordSize}^2)$, da das Invertieren bzw. Transponieren einer kleinen Matrix, ebenso wie die Multiplikation von zwei kleinen Matrizen Zeit $O(\texttt{WordSize}^2)$ benötigt, und alle anderen Operationen wie Setzen oder Löschen von Matrizen, Konkatenation von kleinen Matrizen und Kopieren von kleinen Matrizen in Zeit $O(\ \texttt{WordSize}\ )$ laufen. Man hat also insgesamt

$$c_\rho = c_{KM} + c_{TM} + c_{IM_1} + 2 \cdot c_{CM} + 2 \cdot c_{MM}$$

elementare Operationen.

6.5.10 Löschen von überflüssigen Vektoren

Seien $\mathbf{v}$ und $\mathbf{w}$ Vektorblöcke. Es gilt : $\mathbf{v} = \mathbf{v}_{n+2}$ und $\mathbf{w} = \mathbf{w}_{n+1}$.
Es sollen überflüssige, d.h. redundante Vektoren aus $\mathbf{v}$ und $\mathbf{w}$ gelöscht werden. Überflüssige Vektoren in den Vektorblöcken sind solche, die nur eine triviale oder zu anderen linear abhängige Lösung des Gleichungssystems ergeben.

setze s_1 auf die Anzahl Vektoren in $\mathbf{v}$; finde linear unabhängige Spalten in $\mathbf{v}$ durch Gauss-Elimination; setze k_1 auf die Anzahl dieser Spalten;	(1)
/* jeder Einheitsvektor repräsentiert eine linear unabhängige Spalte in $\mathbf{v}$ */ komponiere eine $s_1 \times k_1$-Matrix δ aus Einheitsvektoren, wobei jeder Einheitsvektor einer linear unabhängigen Spalte in $\mathbf{v}$ entspricht;	(2)
$\mathbf{v} \leftarrow (\mathbf{v} \cdot \delta)$;	(3)
setze s_2 auf die Anzahl Vektoren in $\mathbf{w}$; finde linear unabhängige Spalten in $\mathbf{w}$ durch Gauss-Elimination; setze k_2 auf die Anzahl dieser Spalten;	(4)
/* jeder Einheitsvektor repräsentiert eine linear unabhängige Spalte in $\mathbf{w}$ */ komponiere eine $s_2 \times k_2$-Matrix δ aus Einheitsvektoren, wobei jeder Einheitsvektor einer linear unabhängigen Spalte in $\mathbf{w}$ entspricht;	(5)
$\mathbf{w} \leftarrow (\mathbf{w} \cdot \delta)$;	(6)

Nach Ende des Algorithmus sind $\mathbf{v}$.size und $\mathbf{w}$.size verringert worden, also gilt $k_1 < s_1$ und $k_2 < s_2$
Der Algorithmus benötigt Laufzeit $O(N \cdot \texttt{WordSize})$.Das Testen eines Vektorblocks auf lineare Abhängigkeit mit leicht modifizierter Gauss-Elimination und die Multiplikation eines Vektorblocks mit einer kleinen Matrix benötigen jeweils Zeit $O(N \cdot \texttt{WordSize})$. Außerdem kann die Komposition der kleinen Matrix δ in $O(\texttt{WordSize}^2)$ Zeit durch Zufügen von höchstens `WordSize` Einheitsvektoren aus $GF(2^{s_1})$ bzw. $GF(2^{s_2})$ berechnet werden. Man hat also insgesamt höchstens

$$c_{LÜ} = 2 \cdot (c_{LA} + c_{VM}) + \texttt{WordSize} \cdot (s_1 + s_2)$$

elementare Operationen.

6.5.11 Berechnung des 'Projektions'operators durch Aufspalten

Sei $\tau = \tau_{n+1}^{-1}$ eine kleine Matrix, seien $\mathbf{r}_{n+1}$, $\mathbf{r}_n$ und $\mathbf{w}_n$ Vektorblöcke.
Es gilt : $\alpha = \mathtt{Proj}(\ \mathbf{r}_{n+1}; \mathbf{r}_n\ )$ und $\beta = \mathtt{Proj}(\ \mathbf{r}_{n+1}; A\mathbf{w}_n\ )$ sind beides kleine Matrizen (vgl. S. 31 ff.), und beide Matrizen sind Untermatrizen von $\tau_{n+1^{-1}}$, genauer : Ist δ die Matrix, die aus den untersten $\mathtt{size}(\ \mathbf{r}_{n+1}\ )$ Zeilen von τ besteht, so gilt :

$$\delta = (\ \mathtt{Proj}(\ \mathbf{r}_{n+1}; \mathbf{r}_n\ ) \mid \mathtt{Proj}(\ \mathbf{r}_{n+1}; A\mathbf{w}_n\ )\ )$$

(vgl. S.31 ff.).
Setze $k = \mathtt{size}(\ \mathbf{w}_n\ )$, $l = \mathtt{size}(\ \mathbf{r}_{n+1}\ )$.
Die Berechnung der Projektionsmatrizen führt der folgende Algorithmus durch :

lösche α; lösche β;			(1)
IF $\ l > 0$			(2)
THEN	/* bestimme Bitmaske m, bei der die ersten $\mathtt{size}(\ \mathbf{w}_n\ )$ Bits gesetzt sind */ $m \leftarrow (\ 2^k - 1\ )$;		(3)
	FOR $\ i \leftarrow (\ \tau.rows - l\ )$ TO $\tau.rows$		(4)
		/* es werden die untersten Zeilen von τ genommen */ $\alpha.matrix[i - \tau.rows] \leftarrow (\ \tau.matrix[i] \cdot 2^k\ )$;	(5)
		$\beta.matrix[i - \tau.rows] \leftarrow (\ \tau.matrix[i]$ AND $m\ )$;	(6)

Der Algorithmus benötigt Laufzeit $O(\ \mathtt{WordSize}\ )$.
Die Berechnung der Bitmaske benötigt eine Zuweisung und eine NOT–und Bitverschiebeoperation; die Schleife wird l–mal durchlaufen und führt zusammen $2 \cdot l$ Zuweisungen und Subtraktionen und l Bitverschiebeoperationen aus, und das Löschen einer kleinen Matrix benötigt Zeit $O(\ \mathtt{WordSize}\ )$. Man hat also insgesamt

$$c_{PO} = c_{CM} + 3 + 3 \cdot l$$

elementare Operationen.

Kapitel 7

Performancetests

In diesem Kapitel beschreibe ich die Daten einiger Testläufe und einige bei Testläufen aufgetretene Beobachtungen.
Dieses Kapitel ist unterteilt in zwei Abschnitte :

- Daten der Testläufe
- Beobachtungen

7.1 Daten einiger Testläufe

Die Daten aller Testläufe wurden auf einer SPARC8-Station mit 32 Bit Wortgröße erstellt.

In der ersten Testreihe hat die Matrix 10000 Zeilen und 10000 Spalten. Die Spalteneinträge werden zufällig erzeugt, wobei die Anzahl der Einträge pro Zeile variiert. Die Laufzeit und der benötigte Speicherplatz werden für verschiedene Werte der durchschnittlichen Besetzung pro Zeile angegeben.
In dieser Testreihe wurde kein Preprocessing durchgeführt.

Besetzung	Zeit[min,sec]	Speicherplatz[KBytes]
10	6,15	1415
20	7,33	1834
30	8,38	2230
40	9,55	2627
50	11,01	3025

Tabelle 7.1: 1. Testreihe

In der zweiten Testreihe hat die Matrix 10000 Zeilen und 10000 Spalten. Die Spalteneinträge werden zufällig erzeugt, wobei die durchschnittliche Anzahl der Einträge pro Zeile immer = 10 ist. Die Laufzeit und der benötigte Speicherplatz werden für verschiedene Werte der Startgröße des Vektorblocks $\mathbf{w}_0$ angegeben.
In dieser Testreihe wurde kein Preprocessing durchgeführt.

Startgröße	Zeit[min,sec]	Speicherplatz[KBytes]	Iterationen
8	8,16	1413	1185
12	6,58	1413	789
16	6,15	1415	591
20	5,41	1414	474
24	5,30	1414	395

Tabelle 7.2: 2. Testreihe

In der dritten Testreihe werden die Spalten der Matrix ebenfalls zufällig erzeugt, wobei die durchschnittliche Anzahl der Einträge $\neq 0$ pro Zeile 20 ist. Die Laufzeit und der benötigte Speicherplatz werden für verschieden Werte der der Anzahl der Zeilen angegeben. Die erzeugte Matrix ist quadratisch.
In dieser Testreihe wurde Präprocessing durchgeführt, die Laufzeit für Preprocessing und die reduzierten Zeilengrößen werden in den Spalten PZeit und PZeilen der untenstehenden Tabelle angegeben.

Zeilen	Zeit[h:min,sec]	Speicherplatz[KBytes]	PZeit[min, sec]	PZeilen
1000	0:0,14	202	0,08	973
2000	0:0,52	285	0,10	1887
5000	0:1,35	708	0,18	4723
10000	0:7,33	1834	1,08	9455
20000	0:28,41	3627	4,55	18912
50000	3:43,15	9055	31,49	47203
100000	34:37,49	18117	6:07,15	94496

Tabelle 7.3: 3. Testreihe

In einem Beispiel mit Testdaten aus der Faktorisierung großer ganzer Zahlen, die bei der Faktorisierung einer 50–stelligen Zahl mit dem number field sieve aufgetreten waren, schließlich hatte die Matrix 13.963 Zeilen und 14.063 Spalten.
Nach durchgeführtem Preprocessing hatte die Matrix noch 13.783 Zeilen und 13.932 Spalten, es wurden also 180 Zeilen und 131 Spalten der ursprünglichen Matrix gelöscht.
Vor Preprocessing hatte die Matrix 1.381.199 Einträge $\neq 0$, also durchschnittlich $98,9$ Einträge $\neq 0$ pro Zeile. Nach Preprocessing hatte die Matrix noch 1.361.154 Einträge $\neq 0$, also durchschnittlich $98,8$ Einträge $\neq 0$ pro Zeile. Also wurden durch Preprocessing ca. 1.45% der Einträge $\neq 0$ gelöscht.

Das Programm fand 16 linear unabhängige Lösungen des homogenen Gleichungssystems nach 861 Iterationen.
Dabei dauerte das Einlesen der Matrix 10,41 Minuten, das Preprocessing 5,24 Minuten, und das Lösen des Systems schließlich 52,55 Minuten. Insgesamt belegte das Programm dabei 6.835 KBytes Speicherplatz.

7.2 Beobachtungen

Alle Aussagen in diesem Abschnitt verwenden die Bezeichnungen aus Kap. 3.4.

- Sei g die Anzahl Vektoren in dem Vektorblock $\mathbf{b}$. Der Algorithmus findet wie beschrieben einen Vektorblock $\mathbf{y}$ aus g Vektoren, der die Gleichung $A \cdot \mathbf{y} = \mathbf{b}$ löst. Ist dabei A singulär, so stellt sich heraus, daß die Vektoren in $\mathbf{y}$ linear abhängig sind, also gilt nur $A \cdot \mathbf{y} \subseteq \mathbf{b}$.
  Ferner ist es in wenigen Testfällen vorgekommen, daß dann der Vektorblock $\mathbf{y}$ aus weniger als g Vektoren bestand.
- Hat man einen Vektorblock $\mathbf{y}$ mit $A \cdot \mathbf{y} \subseteq \mathbf{b}$ gefunden, so kann es vorkommen, daß $A \cdot \mathbf{y} = 0$. In diesem Fall war die Matrix A wie oben singulär.
- Ist das urpsrüngliche System unsymmetrisch, und hat man eine Lösung $\mathbf{y}$ mit $A \cdot \mathbf{y} = 0$ gefunden, so kann es weiter vorkommen, daß die Lösung des unsymmetrischen Systems $B \cdot \mathbf{z} = \mathbf{b}$, die ja durch $\mathbf{z} = B^{\mathbf{T}} \cdot \mathbf{y}$ berechnet wird, ebenfalls $= 0$ ist. In diesem Fall war $\mathbf{y}$ eine verschwindende Linearkombination der Spalten in $B^{\mathbf{T}}$.
- In allen bisherigen Testläufen stellte sich heraus, daß die Größe des Vektorblocks $\mathbf{r}_{n+1}$ in fast 50% aller Iterationen 0 ist. In diesem Fall ist die Matrix η_{n+1} regulär, und $\tau_{n+1} = I$. Ferner sind dann alle Matrizen, die als Resultat des 'Projektions'operators auftreten, leer. Dann ist auch die Matrix ω_n leer, da in diesem Fall $\sigma_n = I$ gesetzt wird.
- In wenigen Fällen kann es vorkommen, daß der Algorithmus aussteigt, ohne eine Lösung produziert zu haben. In diesen Fällen wurde der Vektorblock $\mathbf{r}_{n+1}$ zu groß, als das er noch weiter bearbeitet werden konnte. Abhilfe hierzu schafft z.B. :
  - man läßt den Algorithmus mit einer veränderten Startgröße von w_0 wieder laufen.
  - man wählt Preprocessing, falls es verhindert wurde.

Anhang A

Das LOG-File

Das Programm schreibt nach jedem Lauf, gleichgültig ob korrekt beendet oder wegen eines Fehlers abgebrochen, ein LOG-File namens "error.log" in das aktuelle Arbeitsdirectory. War vor dem Programmlauf schon ein solches File im akuellen Directory vorhanden, so werden die Daten des jetzigen Programmlaufs an das alte File angehängt. War noch kein File vorhanden, so wird es kreiert.
In diesem File sind die im folgenden aufgeführten Daten mit Beschreibung über den Programmablauf enthalten :
Der erste Eintrag betrifft die Zeit, zu der das LOG-File geschrieben wurde.
Die nächsten Einträge betreffen das Preprocessing. Es sind :

- Anzahl Zeilen vor Preprocessing
- Anzahl Spalten vor Preprocessing
- Anzahl Zeilen nach Preprocessing
- Anzahl Spalten nach Preprocessing

Nun folgen Einträge, die die Anzahl Einträge $\neq 0$ beschreiben. Dies sind :

- die gesamte Anzahl Einträge $\neq 0$ der Matrix
- die durchschnittliche Anzahl Einträge $\neq 0$ pro Zeile

Dann erfolgen Einträge betreffend Anfangsgrößen von Vektorblöcken. Es sind dies :

- die anfängliche Größe von **b**
- die maximale Größe von **r**

Als nächstes folgen die Fehlerbeschreibungen der beim Programmlauf aufgetretenen Fehler sowie die Schritte, die Operationen und die Operanden, bei denen diese Fehler aufgetreten sind. Ist im Programmlauf kein Fehler aufgetreten, so wird lediglich dies hier vermerkt.

Die nächsten Einträge beschreiben die im Programmlauf benötigte Zeit :

- Dauer der Initialisierung
- Dauer des Preprocessings
- Dauer des Lösens des Gleichungssystems

Die letzten drei Einträge schließlich beschreiben den Gesamtverbrauch an Speicherplatz in KBytes, die Anzahl der gefundenen Lösungsvektoren sowie die Anzahl Iterationen in der Hauptschleife.

Ein typischer LOG-Fileeintrag für einen Programmlauf sieht folgendermaßen aus :

```
***********************************************************

Fehlerbeschreibung fuer den Block-Lanczos-Algorithmus
Mon Jun 20 22:01:54 1994

 Anzahl Zeilen vor Praeprocessing : 40000
 Anzahl Spalten vor Praeprocessing : 40100
 Anzahl Zeilen nach Praeprocessing : 39000
 Anzahl Spalten nach Praeprocessing : 39062

 Gesamte Anzahl Eintraege != 0 : 799263
 Durchschnittliche Anzahl Eintraege != 0 pro Zeile: 20.5
 Anfaengliche Groesse von w : 16
 Maximale Groesse von r : 7

Kein Fehler waehrend der Berechnung aufgetreten !
Die Initialisierung dauerte 0,2 Minuten.
Das Einlesen bzw. Kreieren der grossen Matrix dauerte 0:0,16 Stunden.
Das Preprocessing dauerte 0:33,37 Stunden.
Das Loesen des Systems dauerte 4:27,9 Stunden.
Es wurden 16 Loesungen gefunden.
Die Berechnung benoetigte 2439 Iterationen der Hauptschleife.
Der gesamte Speicherverbrauch des Programms war 7331 KBytes.
===========================================================
```

Anhang B

Fileformate des Programms

Hier wird das selbstdefinierte Format der Files, die von dem Programm erkannt werden beschrieben.
Dieses sind im einzelnen :

- ein File, das die große schwachbesetzte Matrix enthält
- Files, die die Eingabe– bzw. Lösungsvektorblöcke enthalten

Außerdem kann das Programm noch Matrizen in dem Format, das der am Lehrstuhl von Prof. Buchmann implementierte Structured Gauss Algorithmus als Eingabe benötigt, lesen, und Vektoren in dem von dieser Implementierung verwendeten Format ausgeben.

B.1 Das Format der Matrix

Die Matrix wird als ASCII-Text abgespeichert. Jedes File enthält eine einzige große schwachbesetzte Matrix.
Der Kopf des Files sieht folgendermaßen aus :

1. die Anzahl Zeilen der Matrix
2. die Anzahl Spalten der Matrix

Danach folgt die eigentliche Matrix. Sie wird zeilenweise abgespeichert, wobei, um den Vorteil der schwachen Besetzung in Analogie zu der verwendeten Datenstruktur zu erhalten, nicht die absoluten Werte der Zeile, sondern nur die Positionen, an

denen Einträge $\neq 0$ stehen, als ASCII-Texte gespeichert werden.
Jede Zeile beginnt mit der Kennung `Z`. Danach folgt ein Wert, der die Anzahl Einträge $\neq 0$ in dieser Zeile angibt. Die folgenden Werte sind dann, wie eben erwähnt, die relativen Positionen dieser Einträge $\neq 0$ in der Zeile. Das Ende einer jeden Zeile wird durch eine `CR/LF`–Kombination angezeigt.

B.2 Das Format der Vektorblöcke

Die Vektorblöcke ebenfalls als ASCII-Texte abgespeichert.
Jedes File enthält nur einen einzigen Vektorblock. Zu Beginn des Files werden gespeichert :

1. die Anzahl Zeilen des Vektorblocks
2. die Anzahl Vektoren, die in diesem Vektorblock zusammengefaßt sind

Danach folgt der eigentliche Vektorblock :
Jede Zeile des Vektorblocks wird so abgespeichert, daß der Wert jedes Bits dieser Zeile (d.h. 0 oder 1) als Text in einer Textzeile abgespeichert wird; das höchste Bit erscheint dabei am Anfang der Zeile, und das niedrigste Bit am Ende dieser Zeile. Ist eine Zeile des Vektorblocks zu Ende, so wird dies ebenfalls durch eine `CR/LF`–Kombination angezeigt.

Anhang C

Aufruf des Programms

In diesem Kapitel wird der Aufruf des Programms sowie die verschiedenen Optionen, die beim Programmaufruf angegeben werden können, beschrieben.

Das Programm wird folgendermaßen aufgerufen :

```
mat [ -|/ s<file>|r<file>|so<file>|ro<file>|ir<file>|g#|z|np|i|m|c ]
```

Dabei gilt : ist <file> nur ein Filename ohne Pfadangabe, sucht das Programm nur im aktuellen Arbeitsdirektory. Befindet sich das File in anderen Direktorys, muß der volle Pfadname mit angegeben werden.
\# steht für eine positive ganze Zahl.
Die Optionen bedeuten im einzelnen :

r<file> :
: Die dünnbesetzte Matrix wird aus dem File <file> in dem Format eingelesen, das auch der am Lehrstuhl Buchmann vorhandene Structured Gauss Algorithmus verwendet.
Ist diese oder die ro<file> –Option nicht angegeben, wird eine zufällige Matrix erzeugt.

s<file> :
: Die Lösungsvektoren werden in den Files <file>.l# in dem Format abgespeichert, das auch der am Lehrstuhl Buchmann vorhandene Structured Gauss Algorithmus verwendet. Hat das Programm mehrere Lösungsvektoren gefunden, so wird jeder dieser Vektoren in einem eigenen File abgespeichert. An den Filenamen <file> wird dabei jeweils ein Suffix .l# angehängt, wobei # die Nummer des jeweiligen Lösungsvektors ist. Ist diese oder die so<file> –Option nicht angegeben, so werden die Lösungsvektoren auf dem StandardOutput ausgegeben.

`ro<file>` :

Die dünnbesetzte Matrix wird aus dem File `<file>` in dem auf S. 98 f. beschriebenen Format eingelesen.
Ist diese oder die `r<file>` –Option nicht angegeben, wird eine zufällige Matrix erzeugt.

`so<file>` :

Die Lösungsvektoren werden in den Files `<file>.l#` in dem auf S.98 f. beschriebenen Format abgespeichert. Ist diese oder die `s<file>` –Option nicht angegeben, so werden die Lösungsvektoren auf dem StandardOutput ausgegeben.

`ir<file>` :

Soll ein inhomogenes Gleichungssystem gelöst werden, so wird der Vektor in dem selbstdefinierten Format eingelesen.
Ist diese Option nicht angegeben, wird der Algorithmus zum Lösen eines homogenen Systems verwendet.

g# :

Diese Option ändert die Startgröße des Vektorblocks $\mathbf{w}_0$ zu Beginn der Iteration. Dabei ist # eine positive ganze Zahl zwischen Null und `WordSize`. Wird diese Option nicht verwendet, wird die Startgröße von $\mathbf{w}_0$ auf `WordSize`/2 gesetzt.

`z` :

Diese Option erzwingt das zufällige Erzeugen einer dünnbesetzten Matrix. Es werden dann die durchschnittliche Anzahl Einträge $\neq 0$ pro Zeile, sowie die Anzahl Zeilen und Spalten abgefragt.

`np` :

Hat man ein homogenes Gleichungssystem, so wird, wenn diese Option gewählt ist, kein Preprocessing vorgenommen, sonst wird es vorgenommen.

`i` :

Diese Option wird gewählt, wenn ein inhomogenes Gleichungssystem gelöst werden soll. Hat man mit `ir<file>` einen Lösungsvektor angegeben, so wird dieser gelesen, ansonsten wird er zufällig erzeugt.

`m` :

Diese Option wird nur äußerst selten verwendet. Ist sie gewählt, behandelt der Algorithmus die eingegebene Matrix als symmetrisch.

`c` :

Will man ein homogenes Gleichungssystem lösen, und hat die Matrix bedeutend mehr Spalten als Zeilen, so kann man mit dieser Option den größten Teil der überzähligen Spalten aus der Matrix eliminieren.

Literaturverzeichnis

[1] Don Coppersmith, "Solving linear equations over GF(2) : Block Lanczos Algorithm", Research Report RC 16997, updated version of 11.08.1992, **IBM T.J.** Watson Research Center, Yorktown Heights, N.Y., 10598.

[2] Don Coppersmith, "Solving Linear Equations over GF(2) : Block Wiedemann Algorithm", Research Report RC 17293, 17.10.1991, **IBM T.J.** Watson Research Center, Yorktown Heights, N.Y., 10598.

[3] Don Coppersmith, Andrew M. Odlyzko, Richard Schroeppel, "Discrete Logarithms in GF(p)",*Algorithmica* (1986) **1**, pp. 1-15.

[4] Jane K. Cullum, Ralph A. Willoughby *"Lanczos Algorithms for Large Symmetric Eigenvalue Computations Vol. I Theory"*, , Birkhäuser, 1985.

[5] Gerd Fischer *"Lineare Algebra"* , Vieweg 1981

[6] W. Hackbusch *"Iterative Lösung schwachbesetzter Gleichungssysteme"*, Stuttgart, 1991

[7] Brian W. Kernigham, Dennis. M. Ritchie : *"The C Programming Language, Second Edition (ANSI C)"*, Prentice Hall, 1988

[8] B. A. LaMacchia, A. M. Odlyzko,"Solving large sparse linear systems over finite fields", *Advances in Cryptology — CRYPTO '90.*

[9] C. Lanczos, "An iterative method for the solution of the eigenvalue problem of linear differential and integral operators", *J. Res. Nat. Bur. Standards, Sec. B* **45**(1950), pp 255-282.

[10] C. Lanczos, "Solution of systems of linear equations by minmalized iterations", *J. Res. Nat. Bur. Standards* **49**(1952), pp 33-53.

[11] J. van Leuven (Hrsg.) : *Handbook of theoretical computer science*, Volume A, Algorithms and complexity, Elsevier, Amsterdam, 1990

[12] I. Nassi, B. Shneidermann : *Flowchart techniques for structured programming*, SIGPLAN Notice 8 (1973), No.8, S. 12-26

[13] B. N. Parlett, D. R. Taylor, Z. A. Liu, "A look-ahead Lanczos algorithm for unsymmetric matrices", *Math. Comp.* **44**(1985), pp. 105-124.

[14] R. Sedgewick *"Algorithmen in C"* , Addison-Wesley, 1992

[15] J. Stoer, R. Bulirsch *"Einführung in die numerische Mathematik"*, Bd. I und II, Berlin 1990

[16] van der Waerden : *"Algebra"*, Bd. I und II, Springer 1982