Gitterbasisreduktion im Reellen

Diplomarbeit von
Jutta Bartholomes

angefertigt nach einem Thema von
Prof. Dr. Johannes Buchmann
am Fachbereich 14 Informatik
der Universität des Saarlandes

Januar 1993

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

Saarbrücken, im Januar 1993

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Buchmann für das interessante Thema und für seine Betreuung bedanken.

Des weiteren bedanke ich mich bei Dr. Stephan Düllmann für die Betreuung während der Einarbeitungsphase, bei Thomas Papanikolaou für die Bereitstellung der LibF - Arithmetik, sowie bei Christian Heckler und Oliver van Sprang für das Korrekturlesen der Arbeit.

Außerdem bedanke ich mich bei Thomas Denny, Sachar Paulus und Eric Schank für viele anregende Diskussionen und für die angenehme Arbeitsatmosphäre.

Mein besonderer Dank gilt meinen Eltern und meiner Schwester Andrea, die mich während meines Studiums immer unterstützt haben, und die mir auch in Phasen des Zweifels zur Seite gestanden haben.

Schließlich möchte ich mich noch bei meinem Freund Franz-Dieter Berger bedanken, mit dem ich ebenfalls viele anregenden Diskussionen hatte, und der mir seelisch eine große Stütze war.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Einleitung

Gegenstand der vorliegenden Arbeit ist die Berechnung einer reduzierten Gitterbasis aus einem Erzeugendensystem bei Gittern über $\mathbb{R}^n$, dem Vektorraum der reellen Zahlen. Unter einer reduzierten Gitterbasis verstehen wir eine Basis, die sich aus möglichst kurzen Vektoren des Gitters zusammensetzt. Auf die genaue Bedeutung von "möglichst kurz" in diesem Zusammenhang gehen wir erst im Verlauf der Arbeit ein.

Die Berechnung kurzer und kürzester Gittervektoren ist aufgrund der Gitterstruktur von Kristallen in der Kristallographie von besonderem Interesse. Durch die wachsende Bedeutung der Kryptographie haben Gitteralgorithmen auch in der Mathematik und in der Informatik an Bedeutung gewonnen, denn die Berechnung kurzer Gittervektoren ist zum Beispiel zur Lösung von Knapsackproblemen und zur Polynomfaktorisierung geeignet. Um die Komplexität der Probleme einordnen zu können, die in Zusammenhang mit Gittern interessant sind, stellen wir einige davon kurz dar. Dabei bezeichnen wir mit L ein Gitter der Dimension n, mit $\underline{a}_1, \ldots, \underline{a}_n$ die n erzeugenden Vektoren von L, mit $\|\underline{x}\|$ die euklidische Norm eines Vektors $\underline{x}$ und mit $d(L)$ die Determinante des Gitters. (Die exakten Definitionen dieser Begriffe werden erst später angegeben.)

- Sei $L \subseteq \mathbb{Q}^n$ ein Gitter und C eine Zahl. Finde eine Basis $(\underline{a}_1, \ldots, \underline{a}_n)$ von L, so daß
$$\|\underline{a}_1\| \cdot \ldots \cdot \|\underline{a}_n\| \leq C \cdot d(L).$$
Dieses Problem hat eine Lösung für $C \geq n^n$. Eine Basis von L zu finden, für die $\|\underline{a}_1\| \cdot \ldots \cdot \|\underline{a}_n\|$ minimal ist, ist NP - hart.

- Sei $L \subseteq \mathbb{Q}^n$. Gesucht ist der kürzeste Nichtnullvektor von L. Es ist nicht bekannt, ob dieses Problem NP - vollständig ist oder nicht.

Der in dieser Arbeit besprochene und von J. Buchmann und M. Pohst 1989 in [BuPo89] und von J. Buchmann und V. Kessler 1991 in [BuKe91] vorgestellte RLLL - Algorithmus ist polynomiell (in der Dimension und Anzahl der erzeugenden Vektoren), denn er basiert auf dem 1982 von A.K. Lenstra, H.W. Lenstra Jr. and L. Lovasz in [LeLeLo82] vorgestellten LLL - Algorithmus. Dieser ist auch polynomiell (in der Dimension des Gitters), berechnet allerdings in der Theorie Gittervektoren, die noch exponentiell weit (ebenfalls in der Dimension des Gitters) vom kürzesten

Nichtnullvektor des Gitters entfernt sind. Die in der Praxis damit erzielten Ergebnisse sind jedoch häufig wesentlich besser. Eine Schwäche des Algorithmus ist allerdings, daß er als Eingabe eine Basis des Gitters benötigt. Durch seine polynomielle Laufzeit ist der LLL - Algorithmus ein Gitterbasisreduktionsalgorithmus von besonderem Interesse, und daher auch häufig modifiziert worden, so etwa von M. Pohst zur Berechnung einer reduzierten Basis aus einem Erzeugendensystem bei Gittern über $\mathbb{Q}^n$ (siehe auch [Po87]), oder von C.P. Schnorr und M. Euchner [SchnoEu91] zur Zulassung reellwertiger Zwischenergebnisse und damit zur Beschleunigung des Algorithmus in der Praxis.
Die Übertragung der Reduktionsalgorithmen auf Gitter über $\mathbb{R}^n$ ermöglicht uns die über $\mathbb{Q}^n$ bekannten Ansätze auf weitere Probleme anzuwenden. Dabei handelt es sich zum Beispiel um die Reduktion von $\mathbb{Z}$-Basen von Ordnungen. Kern der vorliegenden Arbeit wird daher zum einen die Erläuterung des RLLL - Algorithmus und seine Anwendung auf die Reduktion von $\mathbb{Z}$-Basen von Ordnungen sein. Weiterhin haben wir eine Reihe von Testergebnissen erhalten, die es uns ermöglichen unsere praktischen Erfahrungen bei dieser Klasse von Beispielen zu beschreiben.

Nachfolgend die Gliederung der Arbeit:

In Kapitel 1 führen wir zunächst einige später benötigte mathematische Bezeichnungen ein, erläutern die Darstellungsweise der Algorithmen und führen den Gitterbegriff und einige Eigenschaften von Gittern ein.

In Kapitel 2 erläutern wir in Abschnitt 2.1 den LLL - Algorithmus, so wie ihn A.K. Lenstra, H.W. Lenstra Jr. and L. Lovasz in [LeLeLo82] eingeführt haben. In Abschnitt 2.2 beschäftigen wir uns mit der Variante von C.P. Schnorr und M. Euchner [SchnoEu91], dem LLLFP - Algorithmus.

In Kapitel 3 besprechen wir in Abschnitt 3.1 den RLLL - Algorithmus, der aufbauend auf dem LLL - Algorithmus zur Gitterbasisreduktion über $\mathbb{R}^n$ geeignet ist. In Abschnitt 3.2 lösen wir Probleme, die bei der Implementierung auftreten und wenden den Algorithmus auf die Reduktion von $\mathbb{Z}$ - Basen von Ordnungen an.

In Kapitel 4 fassen wir Testergebnisse zusammen, die wir bei ausgiebigen Tests mit unserer Implementierung erhalten haben.

In Kapitel 5 erläutern wir die der Implementierung zugrundeliegende Datenstruktur und die entstandenen C++ - Libraries.

Im Anhang sind schließlich die Programmlistings im Einzelnen aufgeführt: Anhang A enthält die entstandene Vektorarithmetik, Anhang B die entstandene Matrizenarithmetik und Anhang C die höheren Funktionen

Kapitel 1

Einführung

1.1 Bezeichnungen

1.1.1 Mathematische Bezeichnungen

Wir bezeichnen die Zahlenbereiche mit folgenden Symbolen:

$\mathbb{N}$ Menge der natürlichen Zahlen,

$\mathbb{Z}$ Menge der ganzen Zahlen,

$\mathbb{Q}$ Menge der rationalen Zahlen,

$\mathbb{R}$ Menge der reellen Zahlen.

Für eine Zahl $r \in \mathbb{R}$ bezeichnen wir mit $\lceil r \rfloor \in \mathbb{Z}$ die zu r nächste ganze Zahl, wobei

$$\lceil r \rfloor := \lceil r - \frac{1}{2} \rceil.$$

Analog dazu bezeichnen wir mit $\lfloor r \rceil \in \mathbb{Z}$ die zu $r \in \mathbb{R}$ nächste ganze Zahl, wobei

$$\lfloor r \rceil := \lfloor r + \frac{1}{2} \rfloor.$$

Um den Unterschied dieser beiden Notationen hervorzuheben, betrachten wir Zahlen $r \in (2\mathbb{Z} + 1)/2$. Es gelten dann die Beziehungen

$$\lceil r \rfloor = r - \frac{1}{2}$$

und

$$\lfloor r \rceil = r + \frac{1}{2}.$$

Zur Verdeutlichung dieses Unterschiedes betrachten wir folgendes Beispiel:

Beispiel 1.1 : Sei $r = 1.5$

$$\begin{aligned} \lceil r \rfloor &= \lceil 1.5 \rfloor = 1 \\ \lfloor r \rceil &= \lfloor 1.5 \rceil = 2. \end{aligned}$$

Einen Vektor $\underline{v} \in \mathbb{R}^n$ betrachten wir als Spaltenvektor. Mit $\underline{v}^\top$ bezeichnen wir dann einen Zeilenvektor. Mit v_i $(1 \leq i \leq n)$ bezeichnen wir das i-te Element von $\underline{v}$.
Seien $\underline{v}$ und $\underline{w} \in \mathbb{R}^n$, dann bezeichnen wir mit $< \underline{v}, \underline{w} >= \sum\limits_{i=1}^{n} v_i \cdot w_i$ das Standardskalarprodukt. Folgende Bezeichnungen führen wir für die Vektornormen ein:
Sei $\underline{v} \in \mathbb{R}^n$

$$\begin{aligned} \|\underline{v}\| &= \|\underline{v}\|_2 = \left(\sum_{i=1}^{n} |v_i|^2\right)^{1/2} && l_2\text{-Norm (euklidische Norm)} \\ \|\underline{v}\|_1 &= \sum_{i=1}^{n} |v_i| && l_1\text{-Norm} \\ \|\underline{v}\|_\infty &= \max_{i=1,\ldots,n} |v_i| && l_\infty\text{-Norm.} \end{aligned}$$

Dabei gelten folgende Beziehungen:

$$\|\underline{v}\|_1 \leq \sqrt{n} \cdot \|\underline{v}\| \tag{1.1}$$

$$\|\underline{v}\| \leq \sqrt{n} \cdot \|\underline{v}\|_\infty \tag{1.2}$$

$$\|\underline{v}\|_1 \leq n \cdot \|\underline{v}\|_\infty \,. \tag{1.3}$$

Sei $A \in \mathbb{R}^{m \times n}$ eine Matrix, dann bezeichnet – wie üblich – a_{ij} $(1 \leq i \leq m,$ $1 \leq j \leq n)$ das Matrixelement in der i-ten Zeile und j-ten Spalte.
Mit $det\, A$ bezeichnen wir die Determinante der Matrix, mit A^{-1} das zu A Inverse und mit $I \in \mathbb{R}^{n \times n}$ die n-dimensionale Einheitsmatrix.

1.1.2 Darstellung der Algorithmen

Zur Darstellung der Algorithmen verwenden wir eine modifizierte Form der Nassi-Shneidermann-Diagramme, die in [NaShn73] ausführlich beschrieben werden. In diesen Diagrammen werden Algorithmen als Aneinanderreihung sogenannter Strukturblöcke dargestellt.
Der einfachste dieser Strukturblöcke ist der Anweisungsblock:

Anweisung	(1)

Unter einer Anweisung verstehen wir dabei eine Zuweisung oder einen Prozeduraufruf. Für den Rest dieses Abschnittes verwenden wir den Anweisungsblock stellvertretend für einen oder mehrere solcher Anweisungsblöcke.
Zur Darstellung von Schleifen stehen uns folgende Strukturblöcke zur Verfügung:

1. Zählschleifen

FOR $z \leftarrow a$ TO e		(1)
	Anweisung	(2)

Es handelt sich hierbei um eine Zählschleife, in der aufwärts gezählt wird. z ist die Zählvariable, a($\hat{=}$ Anfang) und e($\hat{=}$ Ende) sind Konstanten oder Variablen. Ist $a > e$, so nennen wir die Zählschleife **abweisend**. In Zählschleifen, in denen abwärts gezählt wird, ist in obigem Strukturblock TO durch DOWNTO ersetzt. Entsprechend sind diese abweisend, wenn $a < e$ gilt. Dieses Zählschleifenkonstrukt wird nur dann verwendet, wenn innerhalb des Schleifenrumpfes weder die Zählvariable, noch a oder e, falls es sich dabei um Variablen handelt, verändert werden.

2. Schleifen, deren Fortsetzung bzw. Beendigung von einer oder mehreren Bedingungen abhängt:

WHILE Bedingung		(1)
	Anweisung	(2)

	Anweisung	(1)
UNTIL Bedingung		(2)

Der Unterschied dieser beiden Konstrukte besteht darin, daß der Rumpf der WHILE-Schleife gegebenenfalls überhaupt nicht ausgeführt wird, während der Rumpf der UNTIL-Schleife mindestens einmal ausgeführt wird, da die Abbruchbedingung erst am Ende jedes Schleifendurchlaufes überprüft wird.

Als Kontrollstrukturen stehen uns zur Verfügung:

IF Bedingung		(1)
THEN	Anweisung	(2)

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

Neben den bisher erläuterten Strukturblöcken wird die Anweisung RETURN(z) verwendet, mit deren Hilfe eine Funktion (eine Prozedur) vorzeitig beendet wird und $z \in \mathbb{Z}$ als Funktionswert zurückgeliefert wird.
Zur Kennzeichnung von Kommentaren innerhalb der Algorithmen verwenden wir die in der Programmiersprache "C" übliche Notation

Anweisung	/* Kommentar */	(1)

1.2 Grundlagen

In diesem Abschnitt wiederholen wir zunächst einige Begriffe und Ergebnisse aus der Algebra, die dem Leser bekannt sein sollten; daher werden wir keine Beweise ausführen. Die hier eingeführten Begriffe sind zum Beispiel in den Büchern von E. Lamprecht [La78] und [La80] nachzulesen.

- **Ein Ring** $(R\,;+,\cdot)$, $R \neq \emptyset$ ist eine additive abelsche Gruppe, für die die Multiplikation assoziativ ist, und in der das Distributivgesetz gilt. Existiert in R ein neutrales Element der Multiplikation, so heißt $(R\,;+,\cdot)$ **ein Ring mit Einselement**.

- Sei $(R\,;+,\cdot)$ ein Ring mit Einselement. **Ein R-Modul** $(M\,;+)$ ist eine additive abelsche Gruppe, für die gilt: es gibt eine äußere Verknüpfung zwischen R und M, mit $R \times M \ni (\rho, x) \longmapsto \rho \cdot x \in M$, für die folgende Regeln gelten:

$$\begin{array}{lll} (i) & 1 \cdot x = x & \text{für alle } x \in M \\ (ii) & (\alpha \cdot \beta) \cdot x = \alpha \cdot (\beta \cdot x) & \text{für } \alpha, \beta \in R, x \in M \\ (iii) & (\alpha + \beta) \cdot x = \alpha \cdot x + \beta \cdot x & \text{für } \alpha, \beta \in R, x \in M \\ (iv) & \alpha \cdot (x + y) = \alpha \cdot x + \alpha \cdot y & \text{für } \alpha \in R, x, y \in M. \end{array}$$

  Ist $R = \mathbb{Z}$, so heißt M ein $\mathbb{Z}$-Modul.

- Eine linear unabhängige Teilmenge B eines R-Moduls M heißt R-**Basis**, falls sie ein Erzeugendensystem von R ist. Dem Nullraum $(\underline{0})$ wird dabei die leere Menge $\emptyset$ als Basis zugeordnet.

- Ein R-Modul heißt **frei**, wenn er eine R-Basis besitzt.

- Ein Ring $(K\,;+,\cdot)$ mit mindestens zwei verschiedenen Elementen heißt **Körper**, wenn $K^\star := K - \{0\}$ eine multiplikative abelsche Gruppe ist.

- Sei K ein Körper. Mit $GL(n, K)$ bezeichnen wir die Gesamtheit der regulären Matrizen über $K^{n \times n}$.

Zum Schluß erinnern wir noch an ein weiteres Hilfsmittel, das für Abschätzungen nützlich sein kann:

Lemma 1.2 (Ungleichung von arithmetischen und geometrischen Mittel) *Seien $a_1, \ldots, a_n \in \mathbb{R}$, dann gilt:*

$$\left(\prod_{i=1}^{n} a_i\right)^{\frac{1}{n}} \leq \frac{1}{n} \cdot \sum_{i=1}^{n} a_i. \tag{1.4}$$

Siehe dazu auch [Er64].

1.3 Gitter

Nach den einführenden Definitionen des letzten Abschnitts wenden wir uns nun den Objekten unseres Interesses zu. Zunächst definieren wir den Gitterbegriff und veranschaulichen ihn anhand einiger Beispiele. Des weiteren erläutern wir den Zusammenhang zwischen Gittern und quadratischen Formen, um den Begriff der Gitterdeterminante einführen zu können. Das erlaubt uns dann, auf den Zusammenhang zwischen der Determinante und der Länge der erzeugenden Vektoren einzugehen. Schließlich führen wir den Begriff der "sukzessiven Minima" eines Gitters ein und erläutern einige ihrer Eigenschaften.

Definition 1.3 (Gitter) *Seien $\underline{a}_1, \ldots, \underline{a}_k \in \mathbb{R}^n$.*

$$L = \sum_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i = \{\sum_{i=1}^{k} \lambda_i \cdot \underline{a}_i \,;\, \lambda_i \in \mathbb{Z}\}$$

heißt **das von $\underline{a}_1, \ldots, \underline{a}_k$ erzeugte Gitter (<u>L</u>attice)**. *Sind die Vektoren $\underline{a}_1, \ldots, \underline{a}_k$ linear unabhängig über $\mathbb{R}^n$, so heißt $(\underline{a}_1, \ldots, \underline{a}_k)$* **Basis von** *$L$; k heißt* **die Dimension des Gitters**. *L ist dann ein freier $\mathbb{Z}$ - Modul der Dimension k. Möchten wir betonen, daß die erzeugenden Vektoren linear unabhängig sind, schreiben wir L daher auch als*

$$L = \bigoplus_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i.$$

Bemerkung 1.4 Zur Vereinfachung schreiben wir ein Gitter auch oft als Matrix $A = (\underline{a}_1, \ldots, \underline{a}_k)$, wobei $\underline{a}_1, \ldots, \underline{a}_k$ die Spalten von A sind.

Bemerkung 1.5 Die Basis eines Gitters ist nicht eindeutig.

Hier einige Beispiele:

Beispiel 1.6 :

1) Ein einfaches 1-dimensionale Gitter ist $\mathbb{Z}$.

2) Ein einfaches 2-dimensionales Gitter ist das Einheitsgitter, das in Abb.1.1 für $l = 3$ dargestellt wird:

$$L = \mathbb{Z} \cdot \underbrace{\begin{pmatrix} 1 \\ 0 \end{pmatrix}}_{=:\underline{a}_1} + \mathbb{Z} \cdot \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{=:\underline{a}_2} \overset{(*)}{=} \mathbb{Z} \cdot \underbrace{\begin{pmatrix} l \\ 1 \end{pmatrix}}_{=:\underline{b}_1} + \mathbb{Z} \cdot \underbrace{\begin{pmatrix} l+1 \\ 1 \end{pmatrix}}_{=:\underline{b}_2}, \quad l \in \mathbb{Z}$$

Hierbei sieht man, daß ein Gitter unendlich viele Basen haben kann. Um die Gleichheit (*) zu überprüfen, muß man $\underline{a}_1$ und $\underline{a}_2$ in Abhängigkeit von $\underline{b}_1$ und $\underline{b}_2$ darstellen können und umgekehrt: $\underline{a}_1 = \underline{b}_2 - \underline{b}_1$, $\underline{a}_2 = (l+1) \cdot \underline{b}_1 - l \cdot \underline{b}_2$, $\underline{b}_1 = l \cdot \underline{a}_1 + \underline{a}_2$, $\underline{b}_2 = (l+1) \cdot \underline{a}_1 + \underline{a}_2$

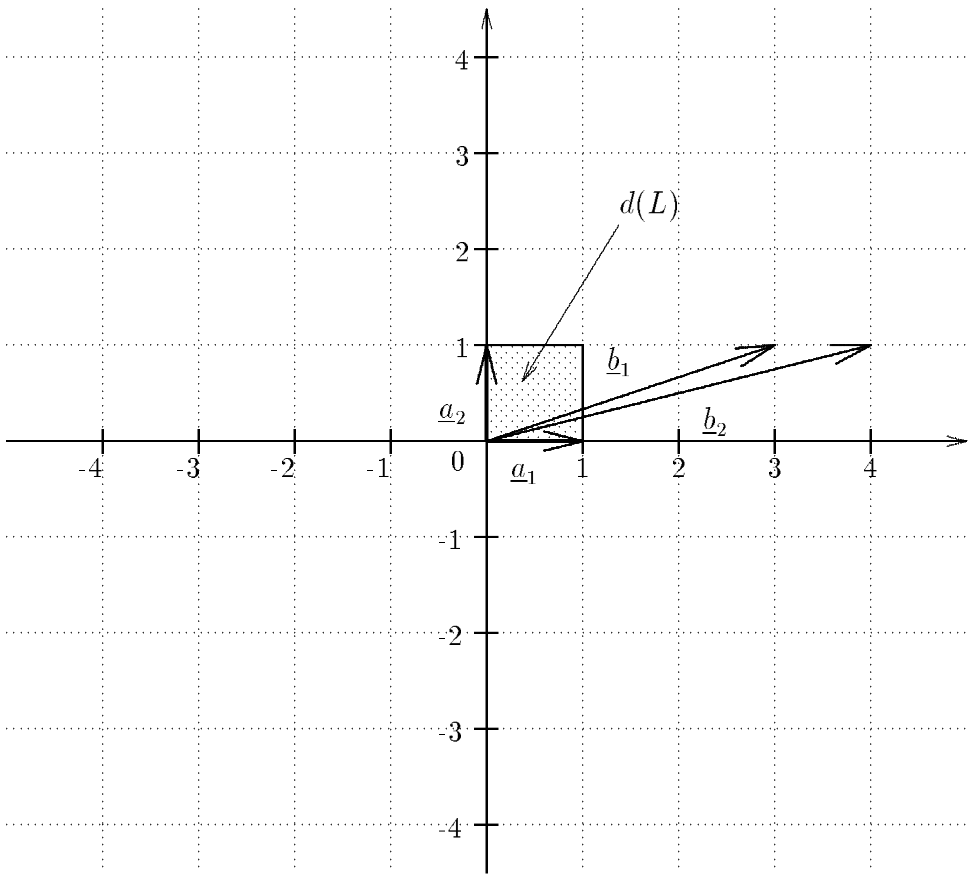


Abbildung1.1: Ein Beispielgitter

Ist ein Gitter statt durch eine Basis nur durch ein Erzeugendensystem gegeben, und möchte man die zwischen den erzeugenden Vektoren bestehenden linearen Abhängigkeiten beschreiben, so eignen sich dazu gut die sogenannten Relationen (oder Relationenvektoren).

Definition 1.7 (Relation, Relationenvektor) *Seien* $\underline{a}_1, \dots, \underline{a}_k \in \mathbb{R}^n$*. Ein ganzzahliger Vektor* $\underline{z} = (z_1, \dots, z_k)^T$ *heißt* **(nichttriviale) Relation** *von* $\underline{a}_1, \dots, \underline{a}_k$*, wenn gilt:*

- $\underline{z} \neq \underline{0}$,
- $\sum\limits_{i=1}^{k} z_i \cdot \underline{a}_i = \underline{0}$.

Definition 1.8 (Quadratische Form) [La78] *Ist* $B \in \mathbb{R}^{n \times n}$ *eine symmetrische Matrix, so heißt eine Abbildung*

$$\begin{array}{rccl} Q_B: & \mathbb{R}^n & \longrightarrow & \mathbb{R} \quad \text{, mit der Eigenschaft} \\ & \underline{x} & \longmapsto & Q_B(\underline{x}) = \underline{x}^\top \cdot B \cdot \underline{x} \end{array}$$

eine quadratische Form auf $\mathbb{R}^n$. *Gilt für* $\underline{x} \neq \underline{0}$ $\quad Q_B(\underline{x}) > 0$*, dann heißt* $Q_B(\underline{x})$ *eine* **positiv definite quadratische Form**

Man kann jeder Gitterbasis eindeutig eine positiv definite quadratische Form zuordnen.
Sei $L = \sum\limits_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i$, $\quad \underline{a}_i \in \mathbb{R}^n \ (1 \leq i \leq k)$, $\quad (\underline{a}_1, \dots, \underline{a}_k)$ eine Basis von L und $A = (\underline{a}_1, \dots, \underline{a}_k)$.
Sei $\underline{v} \in L$, dann kann man $\underline{v}$ schreiben als

$$\underline{v} = \sum_{i=1}^{k} x_i \cdot \underline{a}_i \quad , x_i \in \mathbb{Z} \ \ (1 \leq i \leq k).$$

Betrachte

$$\begin{array}{rcl} \|\underline{v}\|^2 \ = \ < \underline{v}, \underline{v} > & = & < \sum\limits_{i=1}^{k} x_i \cdot \underline{a}_i, \sum\limits_{j=1}^{k} x_j \cdot \underline{a}_j > \\ & = & \sum\limits_{i,j=1}^{k} x_i \cdot x_j \cdot < \underline{a}_i, \underline{a}_j > \end{array}$$

Dann ergibt sich daraus

$$\|\underline{v}\|^2 = \underline{x}^\top \cdot \underbrace{(< \underline{a}_i, \underline{a}_j >)_{i,j=1,\dots,k}}_{=:B} \cdot \underline{x} = \underline{x}^\top \cdot B \cdot \underline{x},$$

so daß wir die oben eingeführte Schreibweise einer quadratischen Form erhalten. Es bleibt noch zu zeigen, daß $\underline{x}^T \cdot B \cdot \underline{x}$ positiv definit ist. Da für $\underline{x} \neq \underline{0}$ gilt

$$\begin{aligned} \underline{x}^T \cdot B \cdot \underline{x} &= \underline{x}^T \cdot (A^T \cdot A) \cdot \underline{x} \\ &= (A \cdot \underline{x})^T \cdot (A \cdot \underline{x}) \\ &= \|A \cdot \underline{x}\|^2 > 0 \end{aligned}$$

folgt die positive Definitheit.
Nun haben wir die Mittel zur Verfügung, den Begriff der Gitterdeterminante einzuführen. Dies geschieht üblicherweise in folgender Form:

Definition 1.9 (Gitterdeterminante) *Sei* $L = \sum_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i, \quad \underline{a}_i \in \mathbb{R}^n \quad (1 \leq i \leq k)$, $(\underline{a}_1, \ldots, \underline{a}_k)$ *eine Basis von* L.
Man definiert **die Determinante von** L $(d(L))$ **als**

$$d(L) = |det(< \underline{a}_i, \underline{a}_j >)_{i,j=1,\ldots,k}|^{\frac{1}{2}}. \tag{1.5}$$

Für den Fall $k = n$ mit $A = (\underline{a}_1, \ldots, \underline{a}_n)$ gilt dabei:

$$d(L) = |det A|, \tag{1.6}$$

denn

$$|det(< \underline{a}_i, \underline{a}_j >)_{i,j=1,\ldots,n}| = (det A)^2 = d(L)^2.$$

Geometrisch veranschaulicht stellt die Gitterdeterminante das Volumen des kleinsten aufspannenden Parallelotops dar (siehe Abb.1.1). Nun wenden wir uns der Beziehung zwischen der Länge der erzeugenden Gittervektoren und der Determinante zu.

Lemma 1.10 (Ungleichung von Hadamard) *Sei* $L = \sum_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i$,
$\underline{a}_i \in \mathbb{R}^n$ $(1 \leq i \leq k)$, $(\underline{a}_1, \ldots, \underline{a}_k)$ *Basis von* L. *Dann gilt*

$$\|\underline{a}_1\| \cdot \|\underline{a}_2\| \cdot \ldots \cdot \|\underline{a}_k\| \geq d(L). \tag{1.7}$$

Einen Beweis der Ungleichung von Hadamard findet man z.B. in [MiMa65].

In diesem Zusammenhang ist in Hinblick auf die Berechnung kurzer Gittervektoren der von Minkowski eingeführte Begriff der "sukzessiven Minima" eines Gitters von besonderem Interesse.

Definition 1.11 (sukzessives Minimum) *Sei* $L = \sum_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i, \quad \underline{a}_1, \ldots, \underline{a}_k \in \mathbb{R}^n$ *linear unabhängig. Die Zahl*

$$\lambda_j(L) = \min\{r > 0;\ \textit{eine Kugel vom Radius } r \textit{ um den Ursprung enthält genau } j \textit{ linear unabhängige Vektoren aus } L\}, \quad (1 \leq j \leq k)$$

heißt **das** j**-te sukzessive Minimum von** L.

Bemerkung 1.12 Es gilt:

$$1)\quad \lambda_1(L) \;\le\; \lambda_2(L) \;\le\; \dots \;\le\; \lambda_k(L) \tag{1.8}$$

$$2)\quad \lambda_1(L) \;=\; \min\{\|\underline{v}\|;\ \underline{v} \neq \underline{0},\ \underline{v} \in L\}, \tag{1.9}$$

d.h. $\lambda_1(L)$ ist die Länge eines kürzesten Nichtnullvektors von L.

Beispiel 1.13 :

$$1)\quad L_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \cdot \mathbb{Z} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot \mathbb{Z} = \underline{a}_1 \cdot \mathbb{Z} + \underline{a}_2 \cdot \mathbb{Z}$$

$$\Rightarrow \lambda_1(L_1) = \lambda_2(L_1) = 1\ (sieheAbb.1.2)$$

$$2)\quad L_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \cdot \mathbb{Z} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} \cdot \mathbb{Z} = \underline{c}_1 \cdot \mathbb{Z} + \underline{c}_2 \cdot \mathbb{Z}$$

$$\text{mit } \underline{c}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ und } \underline{c}_2 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

$$\Rightarrow \lambda_1(L_2) = 1,\ \lambda_2(L_2) = 2\ (sieheAbb.1.2)$$

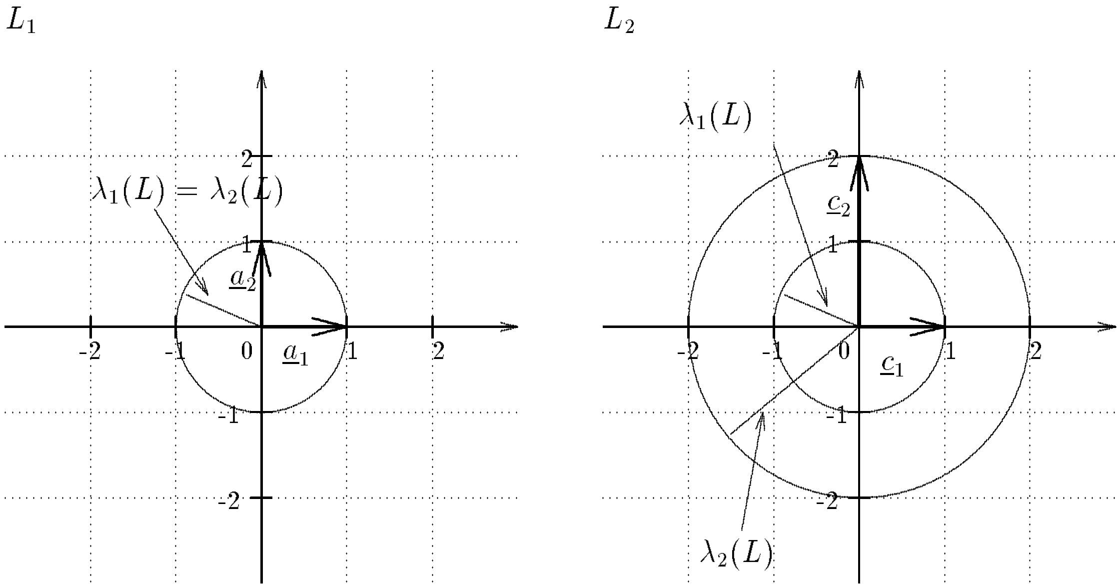


Abbildung1.2: Sukzessive Minima eines Gitters

Satz 1.14 *Sei* $L = \sum_{i=1}^{k} \mathbb{Z} \cdot \underline{a}_i,\ \underline{a}_1, \dots, \underline{a}_k \in \mathbb{R}^n$ *linear unabhängig. Dann gilt*

$$1)\quad d(L) \overset{(a)}{\le} \lambda_1(L) \cdot \ldots \cdot \lambda_k(L) \overset{(b)}{\le} c_k \cdot d(L) \tag{1.10}$$

$$2)\quad \lambda_1(L)^k \le (\gamma_k)^{\frac{k}{2}} \cdot d(L), \tag{1.11}$$

wobei c_k *und* γ_k *Konstanten sind, die nur von* k *abhängen.*

Beweisskizze und Erläuterung:

zu 1a) Sei $\underline{a}_1, \ldots, \underline{a}_k$ eine Basis von L für die gilt $\|\underline{a}_i\| = \lambda_i(L),\ 1 \leq i \leq k$. Dann gilt wegen der Ungleichung von Hadamard (Formel (1.7)):

$$\begin{array}{rcl} & d(L) & \leq & \|\underline{a}_1\| \cdot \ldots \cdot \|\underline{a}_k\| \\ \Rightarrow & d(L) & \leq & \lambda_1(L) \cdot \ldots \cdot \lambda_k(L) \end{array} \tag{1.12}$$

zu 1b) Es handelt sich hierbei um eine Folgerung aus Minkowskis "Convex Body Theorem". Wir werden an dieser Stelle nicht näher auf den Beweis eingehen, da wir uns möglichst schnell der algorithmischen Seite dieses Gebietes zuwenden möchten. Einen Beweis findet man z.B. in [PoZa89], S. 197-198. Die dort berechneten Werte für c_k sind jedoch nicht besonders gut. Ein besseres Ergebnis stellt $c_k = (\gamma_k)^{\frac{k}{2}}$ dar, wobei wir mit γ_k die k-te Hermite - Konstante bezeichnen. Im folgenden wenden wir uns direkt der k-ten Potenz der k-ten Hermite - Konstante (γ_k^k) zu, da der RLLL - Algorithmus (siehe Kapitel 3) auf diese Werte zugreift. Leider kennt man die genauen Werte für γ_k^k nur für $k \leq 8$:

k	1	2	3	4	5	6	7	8
γ_k^k	1	4/3	2	4	8	64/3	64	256

Für $k > 2$ kennt man folgende Rekursionsformel:

$$\gamma_1^1 = 1, \quad \gamma_2^2 = \frac{4}{3}, \quad \gamma_k^k \leq \gamma_{k-1}^{\frac{(k-1)\cdot k}{(k-2)}}. \tag{1.13}$$

Man kann aber auch folgende Abschätzung benutzen:

$$\gamma_k^k \leq \left(\frac{4}{3}\right)^{\frac{k\cdot(k-1)}{2}}. \tag{1.14}$$

Die Abschätzung (1.14) ist zwar ungenauer, aber zum Implementieren besser geeignet als eine Rekursion wie in (1.13).

zu 2) Man ersetze in Ungleichung 1b) c_k durch $(\gamma_k)^{\frac{k}{2}}$. Dann ergibt sich:

$$\lambda_1(L)^k \overset{s.(1.8)}{\leq} \lambda_1(L) \cdot \ldots \cdot \lambda_k(L) \leq (\gamma_k)^{\frac{k}{2}} \cdot d(L)$$

Kapitel 2

Der LLL - Algorithmus

In diesem Kapitel wenden wir uns der Aufgabe zu, kurze Gittervektoren zu berechnen. Der i-te berechnete Vektor wird dabei eine akzeptable Approximation an das i-te sukzessive Minimum darstellen. In diesem Zusammenhang erörtern wir den Begriff der LLL - Reduziertheit, der 1982 von A.K. Lenstra, H.W. Lenstra Jr. und L. Lovasz in [LeLeLo82] eingeführt wurde, um dann den von ihnen entwickelten Algorithmus, der bei Eingabe einer Gitterbasis eine LLL - reduzierte Basis liefert, vorzustellen. In einem ersten Abschnitt werden wir auf die Theorie des Algorithmus eingehen, in einem zweiten auf Details der Implementierung. Dabei betrachten wir eine Variante des LLL - Algorithmus, den LLLFP - Algorithmus, der 1991 in [SchnoEu91] von C.P. Schnorr und M. Euchner vorgestellt wurde, und bei dem die $\mu_{i,j}$ (die Zwischenergebnisse der Orthogonalisierung) in Floatingpointdarstellung (FP - Darstellung) gehalten werden. Die hier vorgestellten Algorithmen eignen sich in der Praxis nur für ganzzahlige Gitter, da sie bei der Rechnung mit reellwertigen Einträgen rundungsfehleranfällig sind.

2.1 Die Theorie des LLL - Algorithmus

Ausgangssituation: im folgenden sei $L = \sum_{i=1}^{n} \mathbb{Z} \cdot \underline{a}_i$, $\underline{a}_1, \ldots, \underline{a}_n \in \mathbb{R}^n$ linear unabhängig, $n \in \mathbb{N}$, und $A = (\underline{a}_1, \ldots, \underline{a}_n)$

Zum weiteren Verständnis wiederholen wir zunächst kurz das Gram - Schmidt'sche Orthogonalisierungsverfahren, indem wir die Prozedur ORTHOGONALIZE(A) einführen. Weiteres zum Thema Orthogonalisierung findet man z.B. in [La78] ab S.141.

Algorithmus 2.1 ORTHOGONALIZE(A)

Parameter: $A = (\underline{a}_1, \ldots, \underline{a}_n) \in \mathbb{R}^n$, $\underline{a}_1, \ldots, \underline{a}_n$ linear unabhängig

Ergebnis: $\underline{a}_1^\star, \ldots, \underline{a}_n^\star \in \mathbb{R}^n$ bilden eine Orthogonalbasis des $\mathbb{R}^n$

FOR $i = 1$ TO n	(1)
$\underline{a}_i^\star \leftarrow \underline{a}_i;$	(2)
FOR $j = 1$ TO $i-1$ /* abweisende Zählschleife für $i = 1$*/	(3)
$\mu_{i,j} \leftarrow < \underline{a}_i, \underline{a}_j^\star > / B_j;$	(4)
$\underline{a}_i^\star \leftarrow \underline{a}_i^\star - \mu_{i,j} \cdot \underline{a}_j^\star;$	(5)
$B_i \leftarrow < \underline{a}_i^\star, \underline{a}_i^\star >;$ /* Initialisierung von B_i */	(6)

Anschaulich gesehen ist $\underline{a}_i^\star$ die Projektion von $\underline{a}_i$ auf das orthogonale Komplement von

$$\sum_{j=1}^{i-1} \mathbb{R} \cdot \underline{a}_j.$$

Im $\mathbb{R}^2$ ist also $\underline{a}_2^\star$ die Projektion von $\underline{a}_2$ auf das orthogonale Komplement von $\underline{a}_1$ (siehe Abbildung2.1). Da außerdem gilt

$$\sum_{j=1}^{i-1} \mathbb{R} \cdot \underline{a}_j = \sum_{j=1}^{i-1} \mathbb{R} \cdot \underline{a}_j^\star \quad \text{(Beweis durch Einsetzen)},$$

folgt: $\underline{a}_1^\star, \ldots, \underline{a}_n^\star$ ist eine Orthogonalbasis des $\mathbb{R}^n$.

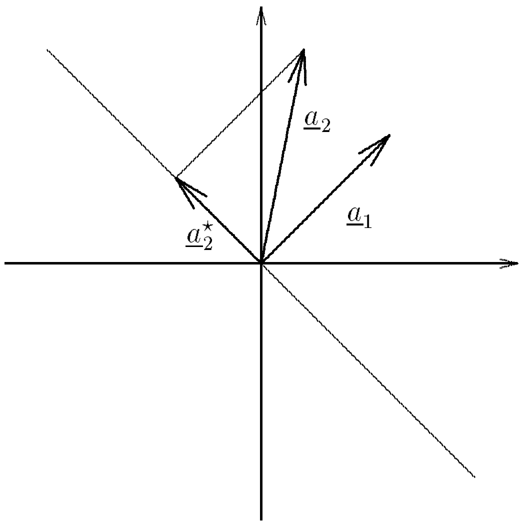


Abbildung2.1: Orthogonale Projektion

Nun haben wir die benötigten Hilfsmittel zur Verfügung, um den Begriff der LLL - Reduziertheit zu definieren.

Definition 2.2 (LLL - reduzierte Basis) *Eine Basis* $\underline{a}_1, \ldots, \underline{a}_n$ *von* L *heißt* **LLL - reduziert**, *falls gilt:*

$$1) \qquad |\mu_{i,j}| \leq \frac{1}{2}, \quad \textit{für } 1 \leq j < i \leq n \tag{2.1}$$

$$2) \quad \|\underline{a}_i^\star + \mu_{i,i-1} \cdot \underline{a}_{i-1}^\star\|^2 \geq \frac{3}{4} \cdot \|\underline{a}_{i-1}^\star\|^2, \quad \textit{für } 1 < i \leq n. \tag{2.2}$$

Dabei sind die $\mu_{i,j}$ $(1 \leq j < i \leq n)$ *die Zwischenergebnisse des Orthogonalisierungsverfahrens (siehe Algorithmus 2.1, Anweisung (4)).*

Bemerkung 2.3 Ist nur Bedingung 1) erfüllt, so nennen wir $\underline{a}_1, \ldots, \underline{a}_n$ **schwach reduziert**.

Was stellen wir uns nun unter der LLL - Reduziertheit vor?

1. Die Werte der $\mu_{i,j}$ $(1 \leq j < i \leq n)$ sind ein Maß für die Abweichung der Orthogonalität der Gittervektoren, denn

$$\mu_{i,j} = 0 \Leftrightarrow \underline{b}_i \perp \underline{b}_j^\star, \quad 1 \leq j < i \leq n. \tag{2.3}$$

   Daher möchte man erreichen, daß gilt

$$\mu_{i,j} \approx 0, \qquad \text{für } 1 \leq j < i \leq n. \tag{2.4}$$

   Gilt also

$$|\mu_{i,j}| \leq \frac{1}{2} \; (\approx 0), \qquad \text{für } 1 \leq j < i \leq n \tag{2.5}$$

   so folgt, daß die Vektoren $\underline{b}_i$ und $\underline{b}_j^\star$ für $1 \leq j < i \leq n$ annähernd orthogonal sind (siehe dazu auch [GrLoSchri88]).

2. Gegeben sei die Basis $(\underline{a}_1, \ldots, \underline{a}_{i-1}, \underline{a}_i, \ldots, \underline{a}_n)$. Für die Orthogonalisierungen $\underline{a}_{i-1}^\star$ und $\underline{a}_i^\star$ der Vektoren $\underline{a}_{i-1}$ und $\underline{a}_i$ gilt:

$$\underline{a}_{i-1}^\star = \underline{a}_{i-1} - \sum_{j=1}^{i-2} \mu_{i-1,j} \cdot \underline{a}_j^\star$$

   und

$$\underline{a}_i^\star = \underline{a}_i - \sum_{j=1}^{i-1} \mu_{i,j} \cdot \underline{a}_j^\star.$$

   Vertauscht man in dieser Basis die Vektoren $\underline{a}_{i-1}$ und $\underline{a}_i$, so erhält man die Basis $(\underline{a}_1, \ldots, \underline{a}'_{i-1}, \underline{a}'_i, \ldots, \underline{a}_n)$ mit $\underline{a}'_{i-1} = \underline{a}_i$ und $\underline{a}'_i = \underline{a}_{i-1}$. Dann gilt für die Orthogonalisierung $\underline{a}'^\star_{i-1}$ von $\underline{a}'_{i-1}$:

$$\underline{a}'^\star_{i-1} = \underline{a}_i^\star + \mu_{i,i-1} \cdot \underline{a}_{i-1}^\star.$$

Ungleichung (2.2) besagt dann, daß $\underline{a}_{i-1}^{\star}$ durch das Vertauschen von $\underline{a}_{i-1}$ und $\underline{a}_i$ nicht wesentlich verkürzt wird (siehe [GrLoSchri88]). Wie wir später sehen werden, ist das Vertauschen zweier Vektoren ein wesentlicher Bestandteil des LLL - Algorithmus. Diese Bedingung liefert uns also eine untere Schranke für die Länge der Orthogonalvektoren beim Vertauschungsschritt.
Eine andere mögliche Interpretation ist die folgende:
$\underline{a}_i^{\star} + \mu_{i,i-1} \cdot \underline{a}_{i-1}^{\star}$ und $\underline{a}_{i-1}^{\star}$ sind die Projektionen von $\underline{a}_i$ und $\underline{a}_{i-1}$ auf das orthogonale Komplement von $\sum_{j=1}^{i-2} \mathbb{R} \cdot \underline{a}_j$. Um uns das besser veranschaulichen zu können, betrachten wir zunächst den zweidimensionalen Fall, d.h. $L = \underline{a}_1 \cdot \mathbb{Z} + \underline{a}_2 \cdot \mathbb{Z}$. Dann lautet die Bedingung:

$$\|\underline{a}_2^{\star} + \mu_{2,1} \cdot \underline{a}_1^{\star}\|^2 \geq \frac{3}{4} \cdot \|\underline{a}_1^{\star}\|^2.$$

Wegen Algorithmus 2.1 Anweisung (5) kann diese Ungleichung umgeformt werden zu:

$$\|\underline{a}_2\|^2 \geq \frac{3}{4} \cdot \|\underline{a}_1\|^2.$$

Im Zweidimensionalen bedeutet diese Reduziertheitsbedingung also, daß der Vektor $\underline{a}_2$ nicht viel kürzer sein darf als der Vektor $\underline{a}_1$. Im n-dimensionalen Fall betrachtet man sukzessiv die zweidimensionalen Gitter, die durch die Vektoren $\underline{a}_i^{\star} + \mu_{i,i-1} \cdot \underline{a}_{i-1}^{\star}$ und $\underline{a}_{i-1}^{\star}$ aufgespannt werden, und man bildet die Analogien zum Zweidimensionalen; d.h. man imitiert auf den so aufgespannten zweidimensionalen Gittern die LLL - Reduktion (im Zweidimensionalen).
Die Konstante 3/4 ist dabei willkürlich gewählt und kann durch einen beliebigen Wert y mit $1/4 < y < 1$ ersetzt werden. Die Praxis zeigt jedoch, daß der LLL - Algorithmus für $y \approx 3/4$ am schnellsten konvergiert. Siehe dazu auch [BaKa].

Bevor wir auf den LLL - Algorithmus eingehen, um dann seine Praktikabilität zu besprechen, betrachten wir kurz einige Eigenschaften LLL - reduzierter Basen.

Satz 2.4 *Sei $(\underline{a}_1, \ldots, \underline{a}_n)$ eine LLL - reduzierte Basis des Gitters L. Dann gilt:*

$$(i) \quad \|\underline{a}_i\| \leq 2^{\frac{n-1}{2}} \cdot \lambda_i(L), \quad 1 \leq i \leq n \tag{2.6}$$

$$(ii) \quad \|\underline{a}_1\| \leq 2^{\frac{n-1}{4}} \cdot d(L)^{\frac{1}{n}} \tag{2.7}$$

$$(iii) \quad \|\underline{a}_1\| \cdot \ldots \cdot \|\underline{a}_n\| \leq 2^{\frac{n \cdot (n-1)}{4}} \cdot d(L). \tag{2.8}$$

Dabei bedeutet Eigenschaft (i) für $i = 1$, daß die Vektoren einer LLL - reduzierten Basis noch exponentiell weit in der Dimension des Gitters vom kürzesten Nichtnullvektor des Gitters entfernt sein können.

Einen Beweis dieses Satzes findet man in [LeLeLo82].

Nun zum Algorithmus:
Er setzt sich im wesentlichen aus den Prozeduren

- ORTHOGONALIZE(A),
- REDUCE($\mu_{k,l}$) und
- SWAP($\underline{a}_k, \underline{a}_{k-1}$)

zusammen. ORTHOGONALIZE(A) haben wir schon zu Beginn dieses Kapitels besprochen. REDUCE($\mu_{k,l}$) erreicht, daß $|\mu_{k,l}|$ für festes $k, l \in \mathbb{N} \leq \frac{1}{2}$ wird. Diese Prozedur bewirkt also, daß die Basisvektoren annähernd orthogonal zueinander werden. War das entsprechende $\mu_{k,l}$ vor Beginn dieser Prozedur dem Betrage nach $> \frac{1}{2}$, so erreichen wir hiermit aber auch, daß der zugehörige Vektor $\underline{a}_k$ echt kürzer wird, denn der im Verlauf dieser Prozedur neu berechnete Vektor $\underline{a}_k$ hat die Eigenschaft, daß $|\mu_{k,l}| \leq \frac{1}{2}$ in der Darstellung bezüglich der gleichen Orthogonalvektoren wie zum Zeitpunkt des Eintritts in REDUCE($\mu_{k,l}$). SWAP vertauscht zwei Basisvektoren $\underline{a}_{k-1}$ und $\underline{a}_k$ miteinander und gewährleistet, daß die $\mu_{i,j}$ $(1 \leq j < i \leq n)$ der Vertauschung entsprechend angepaßt werden. Zunächst werden nun diese beiden Prozeduren vorgestellt und im Anschluß daran sofort der LLL - Algorithmus.

Algorithmus 2.5 REDUCE($\mu_{k,l}$)

Parameter: $\mu_{k,l} \in \mathbb{R},\ k, l \in \mathbb{N}$ fest, Zwischenergebnis der Orthogonalisierung.

Außerdem müssen bekannt sein: $\underline{a}_k, \underline{a}_l,\ k, l \in \mathbb{N},\ \mu_{k,j}, \mu_{l,j}\ 1 \leq j \leq l-1$.

Ergebnis: $\underline{a}_k$ so, daß $|\mu_{k,l}| \leq \frac{1}{2}$.

IF $\lvert\mu_{k,l}\rvert > \frac{1}{2}$			(1)
THEN	$\mu \leftarrow \lceil \mu_{k,l} \rfloor$;		(2)
	$\underline{a}_k \leftarrow \underline{a}_k - \mu \cdot \underline{a}_l$;		(3)
	FOR $j \leftarrow 1$ TO $l-1$		(4)
		$\mu_{k,j} \leftarrow \mu_{k,j} - \mu \cdot \mu_{l,j}$;	(5)
	$\mu_{k,l} \leftarrow \mu_{k,l} - \mu$;		(6)

Algorithmus 2.6 SWAP($\underline{a}_k, \underline{a}_{k-1}$)

Parameter: $\underline{a}_k, \underline{a}_{k-1} \in \mathbb{R}^n$ Basisvektoren von L.

Außerdem müssen bekannt sein: $\mu_{i,j}$ $(1 \leq j < i \leq n)$, $B_i =< \underline{a}_i^\star, \underline{a}_i^\star > \in \mathbb{R}$, $i = k, k-1$.

Ergebnis: $\underline{a}_{k-1}$ und $\underline{a}_k$ vertauscht und die zugehörigen $\mu_{i,j}$ $(1 \leq j < i \leq n)$ entsprechend angepaßt, so daß gilt $\|\underline{a}_k^\star + \mu_{k,k-1} \cdot \underline{a}_{k-1}^\star\|^2 \geq \frac{3}{4}\|\underline{a}_{k-1}^\star\|^2$.

$c \leftarrow \mu_{k,k-1}$;		(1)
$B \leftarrow B_k + c^2 \cdot B_{k-1}$;		(2)
$\mu_{k,k-1} \leftarrow c \cdot B_{k-1}/B$;		(3)
$B_k \leftarrow B_{k-1} \cdot B_k/B$;		(4)
$B_{k-1} \leftarrow B$;		(5)
$(\underline{a}_{k-1}, \underline{a}_k) \leftarrow (\underline{a}_k, \underline{a}_{k-1})$;		(6)
FOR $j \leftarrow 1$ TO $k-2$		(7)
	$(\mu_{k-1,j}, \mu_{k,j}) \leftarrow (\mu_{k,j}, \mu_{k-1,j})$;	(8)
FOR $i \leftarrow k+1$ TO n		(9)
	$(\mu_{i,k-1}, \mu_{i,k}) \leftarrow (\mu_{i,k-1}, \mu_{i,k}) \cdot \begin{pmatrix} 0 & 1 \\ 1 & -c \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ \mu_{k,k-1} & 1 \end{pmatrix}$;	(10)

Der eigentliche Vertauschungsschritt findet bei diesem Algorithmus in den Anweisungen (1) - (6) statt. Mit den Anweisungen (7) - (10) wird dann gewährleistet, daß nach dem Vertauschen von $\underline{a}_k$ und $\underline{a}_{k-1}$ die Darstellungen dieser beiden Vektoren bzgl. der Orthogonalvektoren wieder korrekt sind.

Algorithmus 2.7 LLL - Algorithmus

Eingabe: $\underline{a}_1, \ldots, \underline{a}_n \in \mathbb{R}^n$ Basis des Gitters L.

Ausgabe: $\underline{a}_1, \ldots, \underline{a}_n \in \mathbb{R}^n$ LLL - reduzierte Basis von L.

ORTHOGONALIZE();				(1)
$k \leftarrow 2$;				(2)
WHILE $k \leq n$ /* Die Invariante gilt für k */				(3)
	$l \leftarrow k-1$;			(4)
	REDUCE($\mu_{k,l}$);			(5)
	IF $B_k < \left(\frac{3}{4} - \mu_{k,k-1}^2\right) \cdot B_{k-1}$			(6)
	THEN	SWAP($\underline{a}_k, \underline{a}_{k-1}$);		(7)
		$k \leftarrow \max\{k-1, 2\}$;		(8)
	ELSE	FOR $l \leftarrow k-2$ DOWNTO 1		(9)
			REDUCE($\mu_{k,l}$);	(10)
		$k \leftarrow k+1$; /* Nun gilt die Invariante für $k+1$ */		(11)

Invariante des Algorithmus: zu Beginn von Schritt k ist die Basis bis zu dem $k-1$-ten Vektor LLL - reduziert. Am Ende des Algorithmus gilt dann die Invariante für $k = n + 1$.
Des weiteren ist zu beachten, daß nach dem Vertauschen von $\underline{a}_k$ und $\underline{a}_{k-1}$ die Variable k um 1 dekrementiert werden muß, da sich durch das Vertauschen die Darstellungen der beiden Vektoren bzgl. der von ORTHOGONALIZE(A) berechneten Orthogonalbasis verändert haben. Daher gilt die Invariante nach SWAP($\underline{a}_k, \underline{a}_{k-1}$) nur noch für $k-1$.
Hier nun noch einige Gitterbasen und deren LLL - reduzierte Basen:

Beispiel 2.8

1) $$L = \left(\begin{pmatrix} 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 1 \end{pmatrix} \right) \overset{LLL}{\rightarrow} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)$$

2) $$L = \left(\begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix} \right) \overset{LLL}{\rightarrow} \left(\begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}, \begin{pmatrix} -1 \\ 3 \\ 0 \end{pmatrix} \right)$$

3) $$L = \left(\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 7 \\ 8 \\ 5 \end{pmatrix} \right) \overset{LLL}{\rightarrow} \left(\begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \\ 3 \end{pmatrix} \right)$$

2.2 Implementierung des LLL - Algorithmus

Zur Implementierung eignet sich der LLL -Algorithmus nur im Integerbereich, da die wichtigsten Operationen auf den Gittervektoren Additionsoperationen (d.h. Addition bzw. Subtraktion) sind. Additionsoperationen sind schlecht konditioniert, d.h. es können dabei große Rundungsfehler auftreten.

Bemerkung 2.9 Im Allgemeinen wird eine numerische Aufgabe als **gut konditioniert** bezeichnet, wenn eine kleine Störung der Eingangsdaten auch nur eine kleine Änderung der Ergebnisse zur Folge hat (siehe [Ra86] S. 1.4.).

Dieser Rundungsfehlerproblematik werden wir uns im folgenden zuwenden. Dazu führen wir zunächst eine eindeutige Schreibweise für Gleitkommazahlen ein, die wir ab sofort verwenden werden.

Bemerkung 2.10 Eine Zahl $z \in \mathbb{R}$ läßt sich bezüglich einer Basis b darstellen als

$$z = \pm\ 0.m_{-1}m_{-2}\dots m_{-r} \cdot b^{\pm e_{s-1}e_{s-2}\dots e_0},$$

wobei $m_{-1} \neq 0,\ 0 \le m_{-i} < b$ für $2 \le i \le r$ und $0 \le e_j < b$ für $0 \le j \le s-1$ ist. Es handelt sich hierbei um die **normalisierte Gleitkommadarstellung einer Zahl z zur Basis b.**

Im folgenden werden wir zwischen den Begriffen Floatingpointdarstellung (FP - Darstellung) und der Darstellung einer Zahl in einer “multiprecision” - Arithmetik (MP - Darstellung) unterscheiden.

- Mit FP - Darstellung bezeichnen wir die Darstellung einer Zahl mit Hilfe der Gleitkomma - Standarddatentypen. Da die Implementierung in der Programmiersprache C++ erfolgt ist, entspricht dies also den Datentypen “float” beziehungsweise “double”. In der vorliegenden Implementierung wird für Zahlen in FP - Darstellung der Typ “double” verwendet. Diese Entscheidung ist gegenüber der Verwendung des Types “float” zu Gunsten der höheren Präzision ausgefallen. Nähere Auskunft über die Darstellung dieser beiden Datentypen durch Sun C (ANSI IEEE 754 – 1985) gibt uns Tabelle2.1. Eine “float” oder

Typ	Bits	Name	Inhalt
float	31	Sign	1, falls die Zahl negativ ist.
	23 – 30	Exponent	8-Bit Exponent, “biased” mit 127. Werte wenn alle Bits = 0, bzw. = 1 reserviert.
	0 – 22	Mantisse	23-Bit Mantisse in Darstellung.
double	63	Sign	1, falls die Zahl negativ ist.
	52 – 62	Exponent	8-Bit Exponent, “biased” mit 127. Werte wenn alle Bits = 0, bzw. = 1 reserviert.
	0 – 51	Mantisse	52-Bit Mantisse in Darstellung.

Tabelle2.1: Rechnerinterne Floatingpointdarstellung

  “double” Zahl ist dann dargestellt als:

$$\boxed{(-1)^{Sign} 2^{(Exponent-bias)} 1.f}$$

  wobei mit f in dieser Schreibweise die Mantisse bezeichnet wird.

- Mit MP - Darstellung bezeichnen wir die Darstellung einer Gleitkommazahl bei Verwendung einer “multiprecision” - Arithmetik. Die vorliegende Implementierung wurde für Zahlen in MP - Darstellung mit Hilfe der von Thomas Papanikolaou [Pa93] implementierten “libF” - Arithmetik erstellt, bei der die zur Rechnung verwendete Präzision zwischen 1 und 300000 beliebig eingestellt werden kann. Diese Arithmetik werden wir im folgenden auch mit “bigfloat” - Arithmetik bezeichnen, da sie den “multiprecision” - Datentyp “bigfloat” zur Verfügung stellt. Sowohl bei der FP - als auch bei der MP - Darstellung handelt es sich um normalisierte Gleitkommadarstellungen zur Basis 2.

Bezeichnung 2.1 *Unter der zur Berechnung verwendeten* **Präzision einer Arithmetik** *wollen wir im folgenden die Anzahl der Dezimalstellen der Mantisse in* *der Darstellung der jeweiligen Arithmetik verstehen. Sprechen wir von der Anzahl der binären Stellen der Mantisse, so verwenden wir dafür die Bezeichnung* **binäre Präzision**.

Als nächstes machen wir uns das fehlerverstärkende Verhalten der Addition (wir interpretieren die Subtraktion als Addition mit negativem Vorzeichen) an einem Beispiel klar.

Beispiel 2.11 Dezimale Gleitpunktrechnung mit $r = 4$ und $s = 1$

$$\begin{array}{rclcrcl} x_1 & = & 0.11258762 \cdot 10^2 & \to & rd(x_1) & = & 0.1126 \cdot 10^2 \\ x_2 & = & 0.11244891 \cdot 10^2 & \to & rd(x_2) & = & 0.1124 \cdot 10^2 \end{array}$$

$$\begin{array}{rclrcl} x_1 + x_2 & = & 0.\underline{2250}3653 \cdot 10^2, & rd(x_1) + rd(x_2) & = & 0.\underline{2250} \cdot 10^2 \\ x_1 - x_2 & = & 0.13871 \cdot 10^{-1}, & rd(x_1) - rd(x_2) & = & 0.2000 \cdot 10^{-1}. \end{array}$$

Mit $rd(x)$ bezeichnen wir die Funktion, die eine Dezimalzahl auf die r-te Nachkommastelle rundet. Wir können hierbei beobachten, daß sich das Ergebnis der Subtraktion der gerundeten Werte schon in der ersten Nachkommastelle von dem der exakten Rechnung unterscheidet, d.h. wir stellen eine ca. 1000-fache Fehlerverstärkung gegenüber dem Fehler der Eingangsdaten fest. Hätten wir die Rechnung mit $r = 8$ durchgeführt und dann das Ergebnis auf die vierte Nachkommastelle gerundet, so wäre der Fehler $\leq 10^{-3}$, d.h. höchstens in der vierten Nachkommastelle aufgetreten.
Rechnung mit $r = 8$ und Rundung des Ergebnisses auf $r = 4$:

$$\Rightarrow rd(x_1) = x_1,\ rd(x_2) = x_2$$

$$\begin{array}{rcl} x_1 - x_2 & = & 0.\underline{1387}1 \cdot 10^{-1} \\ rd(x_1 - x_2) & = & 0.\underline{1387} \cdot 10^{-1}. \end{array}$$

Generell kann bei der Addition ähnlich großer Zahlen mit verschiedenen Vorzeichen die im Beispiel demonstrierte "Auslöschung" signifikanter Stellen vorkommen. Um den Fehler in der Addition höchstens in der letzten Nachkommastelle zu erhalten, empfiehlt es sich daher, die Addition mit doppelter Genauigkeit durchzuführen. Die Multiplikation und die Division hingegen sind generell gut konditionierte Operationen. Das kann man sich leicht überlegen, da sowohl bei der Multiplikation als auch bei der Division das Ergebnis mehr Nachkommastellen hat als die Eingangsdaten. Der Fehler liegt dann in diesen niederwertigen Bits. Möchte man allerdings das Ergebnis der Multiplikation exakt verwalten können, so muß man als Genauigkeit die Summe der Wortlängen der einzelnen Multiplikanden wählen. Multipliziert man also zwei gleichlange Zahlen miteinander, so ist in diesem Fall für das Ergebnis ebenfalls doppelte Genauigkeit angebracht.
Da das Ergebnis des LLL - Algorithmus, wie man aus den Algorithmen 2.5 und 2.7

ersehen kann, im wesentlichen von der iterativen Anwendung von Additionsoperationen abhängt, tritt zusätzlich noch eine Akkumulierung der Rundungsfehler auf, so daß das Ergebnis bei Rechnung mit Vektoren in Gleitkommadarstellung zu ungenau werden kann, und somit der Algorithmus zu unstabil wird. Daher eignet sich der LLL - Algorithmus nur für den Integerbereich. Aber selbst da ist der Algorithmus, so wie er im vorherigen Abschnitt vorgestellt wurde, zu ineffizient, denn um die Werte der $\mu_{i,j}$ $(1 \leq j < i \leq n)$ und die der $\underline{a}_i^\star$ $(1 \leq i \leq n)$ exakt berechnen zu können, müßte man aufbauend auf einer zur Implementierung verwendeten langen (Integer -) Arithmetik eine Arithmetik zur Bruchrechnung implementieren. Daher ist für praktische Zwecke der LLLFP - Algorithmus [SchnoEu91], eine Variante des LLL, besser geeignet. Schnorr und Euchner schlagen vor, die Werte der $\mu_{i,j}$ $(1 \leq j < i \leq n)$ und die der Orthogonalvektoren $\underline{a}_i^\star$ $(1 \leq i \leq n)$ in FP - Darstellung zu verwalten (die Basisvektoren $\underline{a}_i$ $(1 \leq i \leq n)$ bleiben in Integerdarstellung) und gegebenenfalls, wenn die Werte im Verlauf der Rechnung zu ungenau werden, gewisse Zwischenrechnungen exakt auszuführen oder Korrekturschritte durchzuführen. Um die dabei auftretenden Ungenauigkeiten der Rechnung kontrollieren zu können, werden allerdings nach dem Vertauschen zweier Vektoren die Werte der $\mu_{i,j}$ $(1 \leq j < i \leq n)$ und die von $\|\underline{a}_i^\star\|$ $(1 \leq i \leq n)$ nicht der Vertauschung entsprechend angepaßt sondern neu berechnet. Um den Algorithmus in diesem Punkt dennoch effizient zu machen, ist ein Vorgehen notwendig, das leicht von dem des original LLL - Algorithmus abweicht. Statt die Orthogonalvektoren $\underline{a}_i^\star$ für $1 \leq i \leq n$ mit allen Zwischenergebnissen, d.h. den $\mu_{i,j}$ $(1 \leq j < i \leq n)$ vorzuberechnen, berechnet und speichert man diese Werte immer nur für $1 \leq i \leq k,\ 1 \leq k \leq n$; dabei bezeichnen wir mit $\underline{a}_k$ den im Algorithmus aktuell betrachteten Vektor. Auf diese Weise erspart man sich die Nachkorrektur der $\mu_{i,k}$ und $\mu_{i,k-1}$ $(k+1 \leq i \leq n)$ (siehe Algorithmus 2.6, Anweisung (9)-(10)).

Wie schon den LLL - Algorithmus, unterteilen wir den LLLFP - Algorithmus ebenfalls in drei Prozeduren:

- ORTHOGONALIZEFP(k),
- REDUCEFP($\mu_{k,j}$),
- SWAPFP($\underline{a}_{k-1}, \underline{a}_k$).

ORTHOGONALIZEFP(k) berechnet $\mu_{k,1}, \ldots, \mu_{k,k-1}$ und $\|\underline{a}_k^\star\|^2 \in \mathbb{R}^n$, $1 \leq k \leq n$, ohne dabei die Orthogonalvektoren explizit zu berechnen, REDUCEFP($\mu_{k,j}$) bewirkt, daß $|\mu_{k,j}| \leq \frac{1}{2}$ für festes $k, j \in \mathbb{N}$, und SWAPFP($\underline{a}_{k-1}, \underline{a}_k$) vertauscht $\underline{a}_{k-1}$ und $\underline{a}_k$ für $1 < k \leq n$. Diesen Prozeduren und schließlich auch dem LLLFP - Algorithmus wenden wir uns nun stärker zu. Dazu bezeichnen wir mit
w^{FP} den zu einem exakten Wert w zugehörigen FP - Wert von w.

Algorithmus 2.12 ORTHOGONALIZEFP(k)

Parameter: $k \in \mathbb{N}$ mit $1 \leq k \leq n$

Außerdem müssen bekannt sein: $\underline{a}_1, \ldots, \underline{a}_k,\ \underline{a}_1^{FP}, \ldots, \underline{a}_k^{FP},\ \mu_{i,j}\ (1 \leq j < i < k),\ c_i = \|\underline{a}_i^{\star FP}\|^2\ (1 \leq i < k),\ r =$ die zugrunde gelegte binäre Präzision in der FP - Darstellung (in unserem Fall ist $r = 52$)

Ergebnis: $\mu_{k,1}, \ldots, \mu_{k,k-1}\ \in \mathbb{R},\ c_k = \|\underline{a}_k^{\star FP}\|^2,\ k \in \mathbb{N}.$

IF $k = 2$			(1)
THEN	$c_1 \leftarrow \|\underline{a}_1^{FP}\|^2;$		(2)
$c_k \leftarrow \|\underline{a}_k^{FP}\|^2;$			(3)
FOR $j \leftarrow 1$ TO $k-1$			(4)
	IF $\mid < \underline{a}_k^{FP}, \underline{a}_j^{FP} > \mid < 2^{-\frac{r}{2}} \cdot \|\underline{a}_k^{FP}\| \cdot \|\underline{a}_j^{FP}\|$		(5)
	THEN	$s \leftarrow < \underline{a}_k, \underline{a}_j >^{FP};$	(6)
	ELSE	$s \leftarrow < \underline{a}_k^{FP}, \underline{a}_j^{FP} >;$	(7)
	$\mu_{k,j} \leftarrow (s - \sum_{i=1}^{j-1} \mu_{j,i} \cdot \mu_{k,i} \cdot c_i)/c_j;$ /* Berechnung von $\mu_{k,1}, \ldots, \mu_{k,k-1}$ */		(8)
	$c_k \leftarrow c_k - \mu_{k,j}^2 \cdot c_j;$ /* Berechnung von $\underline{c}_k = \|\underline{a}_k^{\star FP}\|^2$ */		(9)

In dieser Prozedur geschieht die wichtigste Veränderung zum original LLL - Algorithmus in Bezug auf die Berechnung der Werte der $\mu_{i,j}\ (1 \leq j < i \leq n)$ und der $\|\underline{a}_i^\star\|\ (1 \leq i \leq n)$ in FP - Darstellung und der dabei durch die Floatingpointberechnung möglichen Fehler. Es handelt sich um die Schritte (5) - (7). Ohne den Faktor $2^{-\frac{r}{2}}$ ist die Ungleichung in Zeile (5) des Algorithmus trivialerweise erfüllt, denn nach der Cauchy-Schwarzschen Ungleichung (siehe [La78] S.140, Satz 7.3, Formel (II.7.3c)) gilt für $\underline{x}, \underline{y} \in \mathbb{R}^n$

$$\mid < \underline{x}, \underline{y} > \mid^2 \leq \|\underline{x}\|^2 \cdot \|\underline{y}\|^2$$

und daraus folgt durch Ziehen der Quadratwurzel

$$\mid < \underline{x}, \underline{y} > \mid \leq \|\underline{x}\| \cdot \|\underline{y}\|.$$

Wie wir weiter vorne gesehen haben, ist aufgrund der Additionsoperation die Skalarproduktberechnung schlecht konditioniert. Da außerdem das Ergebnis des Skalarproduktes durch iterative Anwendung der Addition entsteht, können sich die bei jeder Addition entstehenden Rundungsfehler akkumulieren und somit Auslöschung signifikanter - weil führender - Stellen (Bits) provozieren. Dabei sehen wir als führende Bits die oberen $r/2$ Bits an. Daraus erklärt sich die Wahl des Faktors $2^{-\frac{r}{2}}$ (zur Erinnerung: r ist die zugrunde gelegte Präzision der FP - Darstellung zur Basis 2). Hinzu kommt das Problem der FP - Darstellung der Null. Im allgemeinen ist die Gleitkommadarstellung der Null $0.00 \cdot 10^{-exp}$ $(= 0.00 \cdot 2^{-exp'})$, wobei exp (bzw. exp') von der internen Struktur der jeweils verwendeten Arithmetik abhängt. Anders ausgedrückt heißt das, die Null wird nicht eindeutig als 0 dargestellt sondern als irgendeine sehr kleine Zahl. Ist nun in Zeile (5) des Algorithmus ORTHOGONALIZEFP(k)

$$\lvert < \underline{a}_k^{FP}, \underline{a}_j^{FP} > \rvert < 2^{-\frac{r}{2}} \cdot \|\underline{a}_k^{FP}\| \cdot \|\underline{a}_j^{FP}\|,$$

so weiß man a priori (wegen der FP - Darstellung der Null) nicht, ob

1. $\lvert < \underline{a}_k^{FP}, \underline{a}_j^{FP} > \rvert = 0$ oder
2. bei der Berechnung von $\lvert < \underline{a}_k^{FP}, \underline{a}_j^{FP} > \rvert$ Auslöschung der führenden Stellen aufgetreten ist oder
3. der berechnete Wert korrekt ist.

Daher muß in diesem Fall das Skalarprodukt exakt berechnet werden, d.h. in Integerdarstellung mit anschließender Transformation in FP - Darstellung (also $< \underline{a}_k, \underline{a}_j >^{FP}$), bevor sein Wert zur weiteren Verwendung der Variable s zugewiesen wird.

Als nächstes kommen wir nun zu

Algorithmus 2.13 REDUCEFP($\mu_{k,j}$)

Parameter: $\mu_{k,j} \in \mathbb{R},\ k, j \in \mathbb{N}$ fest, Zwischenergebnis der Orthogonalisierung

Außerdem müssen bekannt sein: $\underline{a}_k, \underline{a}_j \in \mathbb{R}^n,\ k, j \in \mathbb{N},\ \mu_{k,i}, \mu_{j,i} \in \mathbb{R},\ 1 \le i \le j-1$; F_c, F_r sind boolesche Variablen, die global oder im Hauptprogramm deklariert sein müssen.

Ergebnis: $\lvert\mu_{k,j}\rvert \le \frac{1}{2}$

IF $\lvert\mu_{k,j}\rvert > \frac{1}{2}$			(1)
THEN	$\mu \leftarrow \lceil \mu_{k,j} \rfloor$;		(2)
	$F_r \leftarrow true$;		(3)
	IF $\lvert\mu\rvert > 2^{\frac{r}{2}}$		(4)
	THEN	$F_c \leftarrow true$;	(5)
	FOR $i \leftarrow 1$ TO $j-1$		(6)
		$\mu_{k,i} \leftarrow \mu_{k,i} - \mu \cdot \mu_{j,i}$;	(7)
	$\mu_{k,j} \leftarrow \mu_{k,j} - \mu$;		(8)
	$\underline{a}_k \leftarrow \underline{a}_k - \mu \cdot \underline{a}_j$;		(9)

Hier ist in Hinblick auf die Berechnung einiger Werte in FP - Darstellung folgendes zu beachten:

1. F_r wird auf *true* gesetzt, wenn $|\mu_{k,j}| > 1/2$, also der Reduktionsvorgang von $\mu_{k,j}$ und $\underline{a}_k$ wirklich stattfindet. Auf diese Weise "merkt" sich der Algorithmus, daß $\underline{a}_k$ reduziert wurde, um dann im späteren Verlauf an geeigneter Stelle die FP - Darstellung von $\underline{a}_k$, also $\underline{a}_k^{FP}$ neu zu berechnen (siehe Algorithmus 2.15 Zeile (12)). Dadurch wird $\underline{a}_k^{FP}$ nur dann neu berechnet, nachdem $\underline{a}_k$ bezüglich aller Vektoren $\underline{a}_1, \ldots, \underline{a}_{k-1}$ auf Reduziertheit überprüft worden ist.

2. F_c wird auf *true* gesetzt, wenn $\mu > 2^{\frac{r}{2}}$. Das bedeutet, daß ein sehr großer "Reduktionskoeffizient" $\mu_{k,j}$ aufgetreten ist, und daß daher wesentliche Änderungen an den Koeffizienten $\mu_{k,i}$ $(1 \leq i \leq j)$ und dem Vektor $\underline{a}_k$ vorgenommen worden sind. Dies wiederum zieht wesentliche Veränderungen der Werte der $\mu_{k-1,j}, \mu_{k,j}$ $(1 \leq j \leq k-1)$ und der Werte $c_{k-1}, c_k, \underline{a}_{k-1}^{FP}$ und $\underline{a}_k^{FP}$ nach sich. Daher wird in diesem Fall der LLLFP - Algorithmus mit der Betrachtung von $\underline{a}_{k-1}$ als aktuellem Vektor fortgesetzt (siehe Algorithmus 2.15 Zeile(14)), so daß die oben genannten Werte korrigiert werden.

Die Prozedur SWAPFP($\underline{a}_{k-1}, \underline{a}_k$) ist selbsterklärend, daher stellen wir im Anschluß daran sofort den LLLFP - Algorithmus vor.

Algorithmus 2.14 SWAPFP($\underline{a}_{k-1}, \underline{a}_k$)

Parameter: $\underline{a}_{k-1}, \underline{a}_k \in \mathbb{R}^n$

Ergebnis: $\underline{a}_{k-1}$ und $\underline{a}_k$, $\underline{a}_{k-1}^{FP}$ und $\underline{a}_k^{FP}$ jeweils vertauscht

$(\underline{a}_{k-1}, \underline{a}_k) \leftarrow (\underline{a}_k, \underline{a}_{k-1})$;	(1)
$(\underline{a}_{k-1}^{FP}, \underline{a}_k^{FP}) \leftarrow (\underline{a}_k^{FP}, \underline{a}_{k-1}^{FP})$;	(2)

Algorithmus 2.15 LLLFP

Eingabe: $\underline{a}_1, \ldots, \underline{a}_n \in \mathbb{R}^n$ Basis des Gitters L

Ausgabe: $\underline{a}_1, \ldots, \underline{a}_n \in \mathbb{R}^n$ LLL - reduzierte Basis von L

```
k ← 2;
FOR i ← 1 TO n
    a_i^{FP} ← (a_i)^{FP};
WHILE k ≤ n
    F_c ← true;
    WHILE F_c = true
        ORTHOGONALIZEFP(k);
        F_c ← F_r ← false;
        FOR j ← k − 1 DOWNTO 1
            REDUCEFP(μ_{k,j});
        IF F_r = true
        THEN a_k^{FP} ← (a_k)^{FP};
        IF F_c = true
        THEN k ← max{k − 1, 2};
    IF 3/4 · c_{k−1} > c_k + μ_{k,k−1}^2 · c_{k−1}
    THEN SWAPFP(a_{k−1}, a_k);
         k ← max{k − 1, 2};
    ELSE k ← k + 1;
```

Im Rahmen der vorliegenden Arbeit sind zwei Implementierungen des LLLFP - Algorithmus entstanden:

- In der ersten werden die Werte der $\mu_{i,j}$ für $1 \leq j < i \leq n$ und die der c_i und der $\underline{a}_i^{FP}$ für $1 \leq i \leq n$ in MP - Darstellung gehalten. Dadurch erhält man eine stabile Implementierung, da die Präzision der verwendeten "bigfloat" - Arithmetik bis zu 300000 festgesetzt werden kann. Der Nachteil dieser Implementierung ist aus Tabelle 2.2 ersichtlich: Sie ist langsam.

- In der zweiten werden die Werte der $\mu_{i,j}$ für $1 \leq j < i \leq n$ und der c_i und $\underline{a}_i^{FP}$ für $1 \leq i \leq n$ in FP - Darstellung gehalten. Wie man aus Tabelle 2.2 entnehmen kann, wird dadurch der Algorithmus stark beschleunigt. Schnorr und Euchner konnten jedoch nicht beweisen, daß die vorgenommenen Korrekturschritte genügen und der Algorithmus immer ein korrektes Ergebnis liefert (siehe [SchnoEu91]). Wir können allerdings ihre Beobachtung, daß der Algorithmus in der Praxis das gewünschte Ergebnis liefert, anhand aller unserer bisher durchgeführten Tests bestätigen.

In beiden Implementierungen sind die Gittervektoren ebenfalls mit Hilfe der "bigfloat" - Arithmetik dargestellt, in der man durch Setzen eines Bits eine Zahl als "lange Integer" kennzeichnen kann, so daß damit Integeroperationen ausgeführt werden. Die Elemente der Gittervektoren sind in diesem Sinne als "lange Integer" dargestellt.

Zum Ende dieses Kapitels wenden wir uns dem Laufzeit - und Rundungsfehlerverhalten des LLLFP - Algorithmus in der Praxis zu. Dazu betrachten wir folgende Klasse von Beispielgittern $L \subseteq \mathbb{Z}^n$, die sich aus Kapitel 3 ergeben: sei L gegeben durch

$$A = \begin{pmatrix} 1 & & & 0 \\ & \cdot & & \\ & & \cdot & \\ & & & \cdot \\ 0 & & & 1 \\ & & & \\ & & * & \\ & & & \end{pmatrix},$$

wobei die Spalten von A linear unabhängig sind und ein Gitter der Dimension r aufspannen. In der folgenden Tabelle setzen wir die Dimension der Matrix A, die Dimension des Gitters r und die zur Rechnung verwendete Präzision prec in Beziehung zu der auf einer Sparc ELC (21 Mips) mit 32 MB RAM gemessenen Laufzeit. Dabei vergleichen wir die mit der "bigfloat" - Version gemessene Zeit mit der von der "double" - Version benötigten. "$\sharp$ Skal.-Prod." steht für die Anzahl der exakt berechneten Skalarprodukte (siehe Algorithmus 2.12 Zeile (5)-(7)), "$\sharp$ It" für die Anzahl der Korrekturschritte, die gemacht wurden (siehe Algorithmus 2.13 Zeile (4)-(5)). Die in der Spalte $\sharp$ Skal.-Prod. angegebenen Bereiche sind wie folgt zu verstehen: sie werden begrenzt durch die in unseren Tests bei der entsprechenden Matrixdimension kleinste und größte vorgekommene Anzahl exakter Skalarproduktberechnungen. Die in der Spalte prec vorkommenden Bereiche werden begrenzt durch die bei dieser Matrixdimension kleinste und größte zur Berechnung verwendeten Präzision. Primär kann man festellen, daß die Laufzeit in erster Linie mit der Matrixdimension wächst und erst in zweiter Linie mit der Gitterdimension. Erwartungsgemäß zeigt die Praxis ebenfalls eine Laufzeitverbesserung bei Nutzung der "double" - Version gegenüber der "bigfloat" - Version. Betrachtet man die Laufzeiten, die bei gleicher Matrixdimension jedoch unterschiedlicher Präzision *prec* gemessen wurden, so stellt man wie erwartet bei beiden Versionen des Algorithmus eine Laufzeitverschlechterung bei steigender Präzision fest. In Tabelle 2.2 kann man dies als schwache Tendenz bei der Matrixdimension 10×5 beobachten. Sie bestätigt sich jedoch auch an dazu exemplarisch getesteten Beispielen. Zu diesem Zweck wurden bei festen Gittern die Laufzeiten bei jeweils zwei verschiedenen Präzisionen gemessen und die Ergebnisse in Tabelle 2.3 zusammengefaßt. Es zeigt sich in Tabelle 2.2 jedoch auch der zunächst erstaunliche Effekt, daß bei Beispielen höherer Matrixdimension der Algorithmus viel schneller gelaufen ist als bei Beispielen niedrigerer Dimension. Dieser Effekt erklärt sich dadurch, daß das Erzeugendensystem des in diesem Fall betrachteten

Matrix-Dim.	r	prec	bigfloat			double		
			♯Skal.-Prod.	♯It.	mm:ss	♯Skal.-Prod.	♯It.	mm:ss
4×2	2	28	0 - 1	0	00:00.01	0 - 1	0	00:00
5×3	3	28	0 - 2	0	00:00.09	0 - 2	0	00:00.01
6×3	3	28	0	0	00:00.04	0	0	00:00.04
7×4	4	28	0 - 1	0	00:00.2	0 - 1	0	00:00.05
8×5	5	28	0 - 4	0	00:00.5	4 - 8	0	00:00.01
9×6	6	28	3 - 12	0	00:00.8	7 - 13	0	00:00.2
10×5	5	28	0	0	00:00.1	0	0	00:00.02
10×5	5	33	0	0	00:00.1	0	0	00:00.05
20×15	15	47	681	13	00:18	681	13	00:05
20×10	10	43 - 71	0 - 28	0	00:00.7	0 - 28	0	00:00.1
30×20	20	59	224	0	00:53	1720	109	00:27
40×20	20	77 - 113	0 - 128	0	00:10	0 - 171	0	00:02
50×30	30	127	527	0	05:23	5906	418	04:52
60×30	30	161 - 235	10 - 54	0	00:31	36 - 651	0	00:16
70×40	40	177	499	0	17:56	12730	927	14:40
80×40	40	237 - 247	13 - 18	0	01:42	43 - 49	0	00:03
90×50	50	255	663	0	51:19	25881	2045	47:30
100×50	50	319 - 333	0	0	11:03	0	0	02:00
120×60	60	385 - 407	6 - 18	0	16:34	36 - 82	0	00:18
140×70	70	491	8 - 15	0	38:38	0	0	00:06

Tabelle2.2: LLLFP: Laufzeiten

Gitters schon aus fast LLL - reduzierten Vektoren bestand, daher die kürzere Laufzeit. Des weiteren kann man feststellen, daß mit der "double" - Version wesentlich öfter exakte Skalarprodukte berechnet und Korrekturschritte durchgeführt werden als mit der "bigfloat" - Version.

Matrixdimension	prec	bigfloat	double
		mm:ss	mm:ss
20×10	45	00:00.7	00:00.1
	100	00:01	00:00.1
25×15	52	00:18	00:06
	100	00:27	00:07
30×20	59	00:53	00:27
	100	01:17	00:30

Tabelle2.3: LLLFP: Laufzeiten bei verschiedenen Präzisionen

Kapitel 3

Berechnung einer reduzierten Gitterbasis aus einem Erzeugendensystem

In diesem Kapitel betrachten wir das Problem aus einem gegebenen Erzeugendensystem, das nicht notwendigerweise aus linear unabhängigen Vektoren besteht, eine reduzierte Basis des dadurch aufgespannten Gitters zu berechnen. Über $\mathbb{Q}^n$ wird dieses Problem mit Hilfe des "modifizierten LLL - Algorithmus" (MLLL) von M.Pohst [Po87] gelöst. Wir konzentrieren uns im folgenden auf die Probleme, die sich ergeben, wenn man Gitter über $\mathbb{R}^n$ betrachtet. Mit anderen Worten: wir wenden uns dem Problem zu, Gitter, deren Vektoren reellwertige Einträge haben, zu reduzieren. Dazu gliedert sich dieses Kapitel in zwei Abschnitte: in dem ersten wird die Theorie des RLLL - Algorithmus, der oben Genanntes leistet, erläutert, in dem zweiten wird der Algorithmus eingeführt. Dabei liegt der Schwerpunkt auf praktischen Problemen, die sich bei der Implementierung ergeben. Im Rahmen dieses zweiten Abschnittes wird auch kurz die Anwendung des Algorithmus auf die Reduktion von $\mathbb{Z}$-Basen von Ordnungen erläutert.

3.1 Die Theorie des RLLL - Algorithmus

Von nun an betrachten wir folgende Ausgangssituation: sei

$$L = \sum_{j=1}^{k} \mathbb{Z} \cdot \underline{a}_j \tag{3.1}$$

das von $\underline{a}_1, \ldots, \underline{a}_k \in \mathbb{Z}^{n_1} \times \mathbb{R}^{n_2}$ erzeugte r - dimensionale Gitter ($r \in \mathbb{N}$), und sei $n = n_1 + n_2$. Weiterhin nehmen wir an, wir kennen folgende Schranken:

$$\mu \leq \lambda_1(L) = \min\{\|\underline{v}\| \mid \underline{0} \neq \underline{v} \in L\} \tag{3.2}$$

(μ nicht zu verwechseln mit dem μ aus den Algorithmen 2.5 und 2.13),
und

$$\alpha \geq \max\{\|\underline{a}_j\| \mid 1 \leq j \leq k\}. \tag{3.3}$$

Die hier beschriebene Ausgangssituation läßt sofort zwei Probleme erkennen:

1. Die erzeugenden Vektoren haben reellwertige Einträge. Um das Problem in seiner ganzen Tragweite zu erfassen, stellen wir uns insbesondere irrationale Vektoreinträge vor, d.h. Zahlen, die eine nichtabbrechende Gleitkommadarstellung haben und daher rechnerintern nicht exakt dargestellt werden können. Daraus resultiert die Notwendigkeit, rationale Approximationen der Vektoreinträge zu berechnen.

2. Das Erzeugendensystem kann lineare Abhängigkeiten enthalten, und daraus entsteht die Aufgabe, eine Basis zu berechnen.

Zusätzlich zu diesen beiden offensichtlichen Zielen bleibt natürlich auch in diesem Kapitel als weiteres Ziel eine aus "möglichst kurzen" Vektoren bestehende Basis des Gitters L zu berechnen. Was hierbei unter "möglichst kurz" zu verstehen ist, wird an späterer Stelle noch genauer erläutert.
Unsere algorithmische Vorgehensweise wird also die folgende sein:

- Berechnung eines ganzzahligen Approximationsgitters $\tilde{L}$ mit der Eigenschaft, daß die erzeugenden Vektoren linear unabhängig sind.
- Anwendung des LLL - Algorithmus auf die erzeugenden Vektoren von $\tilde{L}$.
- Berechnung einer aus "möglichst kurzen" Vektoren bestehenden Basis von L mit Hilfe der LLL - reduzierten Vektoren von $\tilde{L}$.

Zunächst zur Berechnung des Approximationsgitters:
Sei $q \in \mathbb{N}$, und $\underline{\hat{a}}_j = (\hat{a}_{1,j}, \ldots, \hat{a}_{n,j})^T \quad$ für $1 \leq j \leq k$.
Wir setzen

$$\hat{a}_{i,j} := \lfloor 2^q \cdot a_{i,j} \rceil, \quad (1 \leq i \leq n,\ 1 \leq j \leq k).$$

Dann gilt

$$\|2^{-q} \cdot \underline{\hat{a}}_j - \underline{a}_j\| \leq 2^{-q-1} \cdot \sqrt{n_2}, \quad (1 \leq j \leq k), \tag{3.4}$$

d.h. der ganzzahlige Approximationsfehler beträgt höchstens 2^{-q-1} (siehe auch [BuKe91]). 2^q ist also die kleinste zum Rechnen benötigte Präzision, so daß der Approximationsfehler höchstens Formel (3.4) entspricht. Durch diese Konstruktion sind die Elemente von $\underline{\hat{a}}_j$ $(1 \leq j \leq k)$ ganzzahlig geworden, und man könnte den LLL - (LLLFP -) Algorithmus auf diese Vektoren anwenden wären da nicht noch die linearen Abhängigkeiten. Daher konstruieren wir folgendes Gitter:

$$\tilde{L} = \bigoplus_{j=1}^{k} \mathbb{Z} \cdot \underline{\tilde{a}}_j,$$

mit

$$\underline{\tilde{a}}_j = \begin{pmatrix} \underline{e}_j \\ \underline{\hat{a}}_j \end{pmatrix} \in \mathbb{Z}^{k+n}, \quad (1 \leq j \leq k)$$

wobei wir mit $\underline{e}_j$ den j-ten Einheitsvektor aus $\mathbb{Z}^k$ bezeichnen. Die Vektoren $\underline{\tilde{a}}_j$ $(1 \leq j \leq k)$ sind wegen der Einheitsvektoren im oberen Teil linear unabhängig, und daher kann auf diesen Vektoren schließlich der LLL - (LLLFP -) Algorithmus durchgeführt werden:

$$(\underline{\tilde{a}}_1, \ldots, \underline{\tilde{a}}_k) \stackrel{LLL\ (LLLFP)}{\longrightarrow} (\underline{\tilde{b}}_1, \ldots, \underline{\tilde{b}}_k),$$

wobei $(\underline{\tilde{b}}_1, \ldots, \underline{\tilde{b}}_k)$ eine LLL - reduzierte Basis des Gitters $\tilde{L}$ ist. Diese reduzierten Basisvektoren von $\tilde{L}$ können wir nun schreiben als

$$\underline{\tilde{b}}_j = \begin{pmatrix} \underline{m}_j \\ \underline{\hat{b}}_j \end{pmatrix}, \quad (1 \leq j \leq k)$$

wobei

$$\underline{m}_j = (m_{1,j}, \ldots, m_{k,j})^T \in \mathbb{Z}^k \text{ und } \underline{\hat{b}}_j = (\hat{b}_{1,j}, \ldots, \hat{b}_{n,j})^T \in \mathbb{Z}^n, \quad (1 \leq j \leq k).$$

Als nächstes stellt sich uns nun die Frage nach der Wahl von q und danach, wie sich aus den Vektoren $\underline{\tilde{b}}_1, \ldots, \underline{\tilde{b}}_k$ eine Basis von L berechnen läßt. Dazu führen wir zunächst folgende Bezeichnung ein:

Definition 3.1 *Sei*

$$L = \sum_{j=1}^{k} \mathbb{Z} \cdot \underline{a}_j, \quad \underline{a}_j \in \mathbb{Z}^{n_1} \times \mathbb{R}^{n_2} \quad (1 \leq j \leq k)$$

ein Gitter der Dimension r. Seien $\underline{a}_1, \ldots, \underline{a}_r$ linear unabhängig über $\mathbb{Z}^{n_1} \times \mathbb{R}^{n_2}$. Dann bezeichnen wir mit

$$L_r := \bigoplus_{i=1}^{r} \mathbb{Z} \cdot \underline{a}_i$$

ein Teilgitter von L.

Eine Antwort auf die Frage nach der Wahl von q und der Berechnung einer Basis ergibt sich dann aus

Satz 3.2 *Sei*

$$2^q > \frac{(\sqrt{n_2 \cdot k} + 2) \cdot \lambda \cdot 2^{\frac{k-3}{2}}}{\mu} \tag{3.5}$$

mit

$$\lambda = \left(\frac{k \cdot \sqrt{n_2}}{2} + \sqrt{k} \right) \cdot \frac{\alpha^r}{d(L)} \tag{3.6}$$

dann gilt:

$$\|\underline{m}_j\| \leq \begin{cases} 2^{\frac{k-1}{2}} \cdot \lambda, & \textit{für } 1 \leq j \leq k-r, \\ 2^{\frac{k-1}{2}+q+1}, & \textit{für } k-r+1 \leq j \leq k. \end{cases}$$

Außerdem gilt: die Vektoren $\underline{m}_1, \ldots, \underline{m}_{k-r}$ sind linear unabhängige Relationen von $\underline{a}_1, \ldots, \underline{a}_k$ und die Vektoren

$$\underline{b}_j = \sum_{i=1}^{k} m_{i,k-r+j} \cdot \underline{a}_i, \quad (1 \le j \le r) \tag{3.7}$$

bilden eine Basis von L und ihre Länge genügt der Abschätzung

$$\|\underline{b}_j\| \le (\sqrt{k \cdot n_2} + 2) \cdot 2^{\frac{k-1}{2}} \lambda_j(L_r), \quad (1 \le j \le r). \tag{3.8}$$

Beweis: Siehe [BuKe91]. ■

In Worten heißt das, wählt man q wie in Gleichung 3.5 vorgeschlagen, so garantiert die Aussage von Satz 3.2, daß die Anwendung des LLL - Algorithmus auf das Approximationsgitter $\tilde{L}$ die linear abhängigen Vektoren in dem Erzeugendensystem von L detektiert und eine Möglichkeit bereitstellt, eine Basis von L zu berechnen. Um eine Basis von L so wie in Satz 3.2 Formel (3.7) berechnen zu können, ist es notwendig prüfen zu können, ob ein Vektor eine Relation für $\underline{a}_1, \ldots, \underline{a}_k$ ist. Einen solchen Test stellt das folgende Lemma dar:

Lemma 3.3 *Sei $\underline{z} = (z_1, \ldots, z_k)^T \in \mathbb{Z}^k - \{\underline{0}\}$ und es gelte*

$$2^q > \frac{\sqrt{n_2}}{\mu} \cdot \|\underline{z}\|_1. \tag{3.9}$$

Dann ist $\underline{z}$ eine Relation für $\underline{a}_1, \ldots, \underline{a}_k$ genau dann, wenn gilt

$$\|\sum_{i=1}^{k} z_i \cdot \hat{\underline{a}}_i\| \le \frac{\sqrt{n_2}}{2} \cdot \|\underline{z}\|_1. \tag{3.10}$$

Beweis: Siehe [BuKe91]. ■

Auf die bis hierher beschriebene Weise können wir also aus einem Erzeugendensystem $(\underline{a}_1, \ldots, \underline{a}_k)$ von $L = \sum_{j=1}^{k} \mathbb{Z} \cdot \underline{a}_j, \quad L \subseteq \mathbb{R}^n$ eine Basis $(\underline{b}_1, \ldots, \underline{b}_r)$ von L berechnen, wobei für die Länge der Basisvektoren gilt

$$\|\underline{b}_j\| \le (\sqrt{k \cdot n_2} + 2) \cdot 2^{\frac{k-1}{2}} \cdot \lambda_j(L_r), \quad (1 \le j \le r).$$

Dabei ist zu beachten, daß die Länge der Basisvektoren hier durch ein Vielfaches der sukzessiven Minima von L_r also eines Teilgitters von L beschränkt ist. Es wäre jedoch wünschenswert, eine schärfere Schranke in Termen der Dimension und der sukzessiven Minima des vollständigen Gitters angeben zu können. Das ist allerdings, wie die Praxis zeigt, an dieser Stelle nicht möglich. Wendet man aber die auf den vorangehenden Seiten beschriebene Methode zur Berechnung einer Basis $(\underline{b}_1, \ldots, \underline{b}_r)$ aus dem Erzeugendensystem $(\underline{a}_1, \ldots, \underline{a}_k)$ von L, auf die Basis $(\underline{b}_1, \ldots, \underline{b}_r)$ selbst noch einmal an, so erhält man eine neue Basis $(\underline{b}'_1, \ldots, \underline{b}'_r)$ von L, die den gewünschten Anforderungen entspricht: für die Länge der Vektoren $\underline{b}'_1, \ldots, \underline{b}'_r$ gilt dann

$$\|\underline{b}'_j\| \le (\sqrt{r \cdot n_2} + 2) \cdot 2^{\frac{r-1}{2}} \cdot \lambda_j(L), \quad (1 \le j \le r). \tag{3.11}$$

3.2 Der Algorithmus

Die grobe Vorgehensweise des Algorithmus haben wir schon im vorigen Abschnitt erläutert. Wir wiederholen sie daher nur kurz in einer stärker strukturierten Form, um dann detaillierter auf die einzelnen Teile der Berechnung (die einzelnen Prozeduren) einzugehen.
Dazu führen wir zunächst folgende Bezeichnungen ein:

- $A := (\underline{a}_1, \ldots, \underline{a}_k) \in \mathbb{R}^{n \times k}$ ist die Matrix, die als Spalten die erzeugenden Vektoren von L hat,

- $\tilde{A} := (\underline{\tilde{a}}_1, \ldots, \underline{\tilde{a}}_k) \in \mathbb{R}^{(n+k) \times k}$ ist die Matrix, die die erzeugenden Vektoren des Approximationsgitters $\tilde{L}$ als Spalten hat,

- $M := (\underline{m}_1, \ldots, \underline{m}_k) \in \mathbb{R}^{k \times k}$ ist die Transformationsmatrix,

- $B := (\underline{b}_1, \ldots, \underline{b}_r) \in \mathbb{R}^{n \times r}$ ist die Matrix, die die berechneten Basisvektoren von L als Spalten hat (B ist die Ergebnismatrix nach Anwendung von RLLL(A)),

- $B' := (\underline{b}'_1, \ldots, \underline{b}'_r) \in \mathbb{R}^{n \times r}$ ist die Matrix, die die kürzeren Basisvektoren von L als Spalten hat (B' ist die Ergebnismatrix nach Anwendung von RLLL(B)).

Algorithmus 3.4 RLLL

Eingabe: $A = (\underline{a}_1, \ldots, \underline{a}_k) \in \mathbb{R}^{n \times k}$, Erzeugendensystem des Gitters L

Ergebnis: $B = (\underline{b}_1, \ldots, \underline{b}_r) \in \mathbb{R}^{n \times r}$, Basis von L

/* Berechnung des Approximationsgitters */		
$power \leftarrow COMPUTE\_POWER\_OF\_TWO(A)$;		
FOR $j \leftarrow 1$ to k		
	FOR $i \leftarrow 1$ to n	
		$\underline{\hat{a}}_{i,j} \leftarrow \lfloor power \cdot \underline{a}_{i,j} \rceil$;
FOR $j \leftarrow 1$ to k		
	$\underline{\tilde{a}}_j \leftarrow \begin{pmatrix} \underline{e}_j \\ \underline{\hat{a}}_j \end{pmatrix}$;	
/* Approximationsgitter berechnet */		
$LLLFP(\tilde{A})$;		
/* Berechnung der Transformationsmatrix */		
FOR $j \leftarrow 1$ to k		
	FOR $i \leftarrow 1$ to k	
		$m_{i,j} \leftarrow \tilde{a}_{i,j}$;
/* Transformationsmatrix berechnet */		
/* Berechnung einer Basis von L */		
$COMPUTE\_BASIS(A, M, B)$;		
/* Basis von L berechnet und in B gespeichert */		

Als erstes werden wir genauer auf die Berechnung des Faktors 2^q durch die Prozedur $COMPUTE\_POWER\_OF\_TWO(A)$ eingehen. Dabei treten folgende Probleme auf:

1. Wir möchten 2^q so berechnen, daß die Aussage von Satz 3.2 gewährleistet wird, aber 2^q auch möglichst klein ist, damit die Werte bei der Berechnung nicht explodieren und der Algorithmus schnell ist.

2. Wir benötigen eine untere Schranke für μ, die Länge des kürzesten Nichtnullvektors von L.

3. Wir benötigen ebenfalls eine untere Schranke für die Determinante $d(L)$ von L.

4. Wir kennen die Dimension r des Gitters nicht.

zu 1) Zur Erinnerung: es soll gelten

$$2^q > \frac{(\sqrt{n_2 \cdot k} + 2) \cdot \lambda \cdot 2^{\frac{k-3}{2}}}{\mu}$$

mit

$$\lambda = \left(\frac{k \cdot \sqrt{n_2}}{2} + \sqrt{k} \right) \cdot \frac{\alpha^r}{d(L)}, \tag{3.12}$$

und q soll möglichst klein sein. Daraus folgt: es soll gelten

$$2^{q-1} \leq \frac{(\sqrt{n_2 \cdot k} + 2) \cdot \lambda \cdot 2^{\frac{k-3}{2}}}{\mu},$$

und das ist äquivalent zu

$$2^{q} \leq \frac{(\sqrt{n_2 \cdot k} + 2) \cdot \lambda \cdot 2^{\frac{k-1}{2}}}{\mu}.$$

Wählen wir also

$$2^{q} = \frac{(\sqrt{n_2 \cdot k} + 2) \cdot \lambda \cdot 2^{\frac{k-1}{2}}}{\mu}, \tag{3.13}$$

mit λ wie in Formel (3.12), so gewährleistet uns Satz 3.2, daß der Algorithmus eine Basis von L findet.

zu 2) Nun ist allerdings 2^q in dieser Formel beschränkt durch ein Vielfaches des Reziproken von μ wobei $\mu \leq \{\|\underline{v}\| \mid \underline{0} \neq \underline{v} \in L\}$. Der kürzeste Nichtnullvektor von L ist aber das, was wir letztlich mit diesem Algorithmus suchen. Also benötigen wir eine untere Schranke für μ um 2^q berechnen zu können. Eine allgemeine untere Schranke für μ wäre die folgende:

$$\mu \geq \min\{\|\underline{a}_1^\star\|, \ldots, \|\underline{a}_k^\star\|\}.$$

Diese Schranke ist allerdings bei der Rechnung mit Gleitkommazahlen unzuverlässig, da das Gram - Schmidt'sche Orthogonalisierungsverfahren ein schlecht konditionierter Algorithmus ist. Daher werden wir von der Rechnung mit Gleitkommazahlen unabhängige, jedoch problembezogene untere Schranken für μ betrachten. Dazu wenden wir uns einem Anwendungsgebiet des Algorithmus zu, der Reduktion von $\mathbb{Z}$-Basen von Ordnungen:

Zunächst wiederholen wir einige Begriffe aus der algebraischen Zahlentheorie, die z.B. in den Büchern von S. Lang [Lang86] oder von P. Bundschuh [Bund88] nachgelesen werden können.

- Ein **algebraischer Zahlkörper** K ist eine endlich Erweiterung des Körpers $\mathbb{Q}$ der rationalen Zahlen.
- $\alpha \in K$ heißt **algebraisch über** $\mathbb{Q}$ oder einfach **algebraisch**, wenn α Nullstelle eines Polynomes $f \in \mathbb{Q}[x] - \{0\}$ ist.

Sei $f(x) = x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0 \in \mathbb{Q}[x]$ ein über $\mathbb{Q}[x]$ irreduzibles Polynom. Sei ρ eine Nullstelle von $f(x)$, und sei

$$K = \mathbb{Q}(\rho) = \{b_0 + b_1 \cdot \rho + \ldots + b_{n-1} \cdot \rho^{n-1} \mid b_i \in \mathbb{Q}, \quad 0 \leq i \leq n-1\}$$

ein algebraischer Zahlkörper. Dann ist eine Ordnung O in K gleich

$$O = \{b'_0 + b'_1 \cdot \rho + \ldots + b'_{n-1} \cdot \rho^{n-1} |\ b'_i \in \mathbb{Z}, \quad 0 \leq i \leq n-1\}.$$

Seien $\rho^{(1)}, \ldots, \rho^{(r_1)}$ die reellen Nullstellen und $\rho^{(r_1+1)}, \ldots, \rho^{(n)}$ die komplexen Nullstellen von $f(x)$. Für jedes $\rho \in \mathbb{C}$, ρ Nullstelle von $f(x)$ ist auch $\overline{\rho}$ Nullstelle von $f(x)$. Sei r_2 die Anzahl der komplexen Nullstellenpaare. Dann ordnen wir die komplexen Nullstellen so, daß für $1 \leq i \leq r_2$ gilt:

$$\rho^{(r_1+r_2+i)} = \overline{\rho^{(r_1+i)}}$$

Wir betrachten den Körpermonomorphismus

$$\begin{array}{rccc} & K & \longrightarrow & \mathbb{C} \\ \text{mit} & \alpha & \longmapsto & \alpha^{(i)} \end{array}$$

wobei

$$\alpha = \sum_{j=0}^{n-1} b'_j \cdot \rho^j \quad \text{und} \quad \alpha^{(i)} = \sum_{j=0}^{n-1} b'_j (\rho^{(i)})^j \quad \text{ist.}$$

$\alpha^{(i)} \in \mathbb{C}$ heißt **i-te Konjugierte von** $\alpha \in K$. Als nächstes betrachten wir die Abbildung:

$$\begin{array}{rcl} K & \longrightarrow & \mathbb{R}^n \\ \text{mit } \alpha & \longmapsto & \underline{\alpha}, \quad \text{wobei} \end{array}$$

$$\underline{\alpha} = (\alpha^{(1)}, \ldots, \alpha^{(r_1)}, Re(\alpha^{(r_1+1)}), Im(\alpha^{(r_1+1)}), \ldots, Re(\alpha^{(r_1+r_2)}), Im(\alpha^{(r_1+r_2)}))$$

Das Bild $\underline{\alpha} \in \mathbb{R}^n$ von α heißt **der Konjugiertenvektor von** $\alpha \in K$.
Das Bild $\underline{O}$ von O ist dann ein Gitter über dem $\mathbb{R}^n$, auf das unser Algorithmus angewendet werden kann, um eine $\mathbb{Z}$-Basis zu berechnen.
Nun wollen wir eine untere Schranke für $\|\underline{\alpha}\|$ berechnen. Dazu definieren wir zunächst die Norm einer algebraischen Zahl:

Definition 3.5 *Sei $\alpha \in K$, dann heißt*

$$N(\alpha) = \prod_{i=1}^{n} \alpha^{(i)}$$

die Norm von α über $\mathbb{Q}$.

Nun zur Berechnung der unteren Schranke für $\|\underline{\alpha}\|$: Aus

$$\prod_{i=1}^{n} |\alpha^{(i)}| = |N(\alpha)| \geq 1 \tag{3.14}$$

folgt

$$\begin{aligned}
\|\underline{\alpha}\|^2 &\geq \frac{1}{2}\cdot\sum_{i=1}^{n}|\alpha^{(i)}|^2 \\
&= \frac{n}{2}\cdot\frac{1}{n}\cdot\sum_{i=1}^{n}|\alpha^{(i)}|^2 \\
&\overset{(1.4)}{\geq} \frac{n}{2}\left(\prod_{i=1}^{n}|\alpha^{(i)}|^2\right)^{\frac{1}{n}} \\
&\overset{(3.14)}{=} \frac{n}{2}\cdot|N(\alpha)|^{\frac{2}{n}} \\
&\overset{(3.14)}{\geq} \frac{n}{2}
\end{aligned}$$

$$\Rightarrow \|\underline{\alpha}\| \geq \sqrt{\frac{n}{2}}$$

Also gilt für die Anwendung des Algorithmus auf die Reduktion von $\mathbb{Z}$-Basen von Ordnungen:

$$\mu \geq \sqrt{\frac{n}{2}}.$$

Mit dieser Abschätzung für μ läßt sich dann 2^q nach Formel (3.13) berechnen.

zu 3) Nun bleibt noch zu beachten, daß für die Berechnung von λ nach Formel (3.12) $d(L)$ berechnet werden muß. Das kann man nicht, solange man keine Basis von L kennt. Daher muß man $d(L)$ abschätzen durch

$$d(L) \geq \frac{\mu^r}{(\gamma_r)^{\frac{r}{2}}}$$

$$\begin{aligned}
\Rightarrow\ \lambda &\leq \left(\frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}\right)\cdot\frac{\alpha^r\cdot(\gamma_r)^{\frac{r}{2}}}{\mu^r} \\
\text{Da } 2^q &\leq \frac{(\sqrt{n_2\cdot k}+2)\cdot\lambda\cdot 2^{\frac{k-1}{2}}}{\mu} \\
\text{folgt } 2^q &\leq \frac{(\sqrt{n_2\cdot k}+2)\cdot 2^{\frac{k-1}{2}}\cdot\left(\frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}\right)\cdot\alpha^r\cdot(\gamma_r)^{\frac{r}{2}}}{\mu^{r+1}}.
\end{aligned}$$

Nun kann man μ wie oben beschrieben durch $\sqrt{\frac{n}{2}}$ abschätzen.

$$\Rightarrow 2^q \leq \frac{(\sqrt{n_2\cdot k}+2)\cdot 2^{\frac{k-1}{2}}\cdot\left(\frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}\right)\cdot\alpha^r\cdot(\gamma_r)^{\frac{r}{2}}}{\left(\frac{n}{2}\right)^{\frac{r+1}{2}}}.$$

zu 4) Als letztes haben wir das Problem, daß wir die Dimension r des Gitters bei der Bestimmung des Faktors 2^q noch nicht kennen. Die Formel, mit der wir 2^q berechnen können hängt jedoch von r ab. Da die Dimension des Gitters höchstens so groß sein kann wie die Dimension n oder die Anzahl k der Eingabevektoren, liegt es nahe in Formel (3.12) zur Berechnung von λ, r durch $\beta = \min\{n,k\}$ zu ersetzen. Wir wählen daher:

$$
\begin{aligned}
2^q &:= \frac{(\sqrt{n_2 \cdot k}+2)\cdot 2^{\frac{k-1}{2}} \cdot \left(\frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}\right)\cdot \alpha^{\beta}\cdot(\gamma_\beta)^{\frac{\beta}{2}}}{\left(\frac{n}{2}\right)^{\frac{\beta+1}{2}}} && (3.15)\\
&\geq \frac{(\sqrt{n_2 \cdot k}+2)\cdot 2^{\frac{k-1}{2}} \cdot \left(\frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}\right)\cdot \alpha^{r}\cdot(\gamma_r)^{\frac{r}{2}}}{\left(\frac{n}{2}\right)^{\frac{r+1}{2}}} && (3.16)\\
&> \frac{(\sqrt{n_2 \cdot k}+2)\cdot \lambda \cdot 2^{\frac{k-3}{2}}}{\mu} && (3.17)
\end{aligned}
$$

mit λ wie in Formel (3.12) und $\beta = \min\{n,k\}$, so daß die Voraussetzungen von Satz 3.2 erfüllt sind, und daher der Algorithmus garantiert eine Basis berechnet.

Bevor wir uns der Aufgabe der Basisberechnung widmen, geben wir nun noch die Prozedur COMPUTE_POWER_OF_TWO(A) an:

Algorithmus 3.6 COMPUTE_POWER_OF_TWO(A)

Parameter: $A \in \mathbb{R}^{n\times k}$,

Außerdem müssen bekannt sein: $k, n, n_2 \in \mathbb{N}$, $\gamma_\beta \in \mathbb{R}$ die β-te Hermite - Konstante,

Ergebnis: $power := 2^q$ wird als Funktionswert zurückgeliefert.

$\beta \leftarrow \min\{n,k\}$;	(1)
$\alpha \leftarrow \max\{\lVert \underline{a}_i \rVert \mid 1 \leq i \leq k\}$;	(2)
$\lambda \leftarrow \frac{k\cdot\sqrt{n_2}}{2}+\sqrt{k}$;	(3)
$power \leftarrow \frac{(\sqrt{n_2\cdot k}+2)\cdot 2^{\frac{k-1}{2}}\cdot\lambda\cdot\alpha^{\beta}\cdot(\gamma_\beta)^{\frac{\beta}{2}}}{\left(\frac{n}{2}\right)^{\frac{\beta+1}{2}}}$;	(4)
RETURN($power$)	(5)

Nun wollen wir uns der genauen Berechnung einer Basis zuwenden. Das zentrale Problem dabei besteht im Detektieren der berechneten Relationenvektoren, um damit lineare Abhängigkeiten ausfindig zu machen. Wie wir bereits gesehen haben, kann dazu Lemma 3.3 herangezogen werden. Man testet mit Hilfe dieses Kriteriums, angefangen bei $\underline{m}_1$ und höchstens bis $\underline{m}_k$, ob es sich bei den Spaltenvektoren der Transformationsmatrix M um Relationen von $\underline{a}_1, \ldots, \underline{a}_k$ handelt. Ist $\underline{m}_s$ der erste Spaltenvektor von M, der keine Relation von $\underline{a}_1, \ldots, \underline{a}_k$ ist, so existieren $s-1$ linear abhängige Vektoren unter $\underline{a}_1, \ldots, \underline{a}_k$, und die Dimension des Gitters ist

$$r = k - s + 1.$$

Mit Hilfe von $\underline{m}_s, \ldots, \underline{m}_k$ kann man dann nach Satz 3.2 die Basisvektoren $\underline{b}_1, \ldots, \underline{b}_r$ von L berechnen. Da unser Testkriterium ebenfalls von μ abhängt, können wir auch hier für μ die gleiche problembezogene Abschätzung verwenden wie zur Berechnung von 2^q. Die Prozedur COMPUTE_BASIS(A, M, B) lautet dann folgendermaßen:

Algorithmus 3.7 COMPUTE_BASIS(A, M, B)

Parameter: $A \in \mathbb{R}^{n \times k}, M \in \mathbb{R}^{k \times k}, B \in \mathbb{R}^{n \times r}$,

Außerdem müssen bekannt sein: $\underline{\hat{a}}_1, \ldots, \underline{\hat{a}}_k \in \mathbb{Z}^n, power = 2^q \in \mathbb{R}$,

Ergebnis: B ist die Matrix der Basisvektoren.

/* Detektieren der Relationen */	(1)
$s \leftarrow 0$; /* Initialisierung */	(2)
$s \leftarrow s + 1$;	(3)
$\underline{sum} \leftarrow \underline{0}$; /* Initialisierung */	(4)
FOR $i \leftarrow 1$ TO k	(5)
$\underline{sum} \leftarrow \underline{sum} + m_{i,s} \cdot \underline{\hat{a}}_i$;	(6)
UNTIL $((power > \sqrt{\frac{2 \cdot n_2}{n}} \cdot \Vert\underline{m}_s\Vert_1)$ and $(\Vert\underline{sum}\Vert \leq \frac{\sqrt{n_2}}{2} \cdot \Vert\underline{m}_s\Vert_1))$ or $(s = k)$	(7)
/* Relationen detektiert */	(8)
/* Berechnung der Basisvektoren */	(9)
$r \leftarrow k - s + 1$;	(10)
FOR $j \leftarrow 1$ TO r	(11)
$\underline{b}_j \leftarrow \underline{0}$; /* Initialisierung */	(12)
FOR $i \leftarrow 1$ TO k	(13)
$\underline{b}_j \leftarrow \underline{b}_j + m_{i,k-r+j} \cdot \underline{a}_i$;	(14)
/* Basisvektoren berechnet */	(15)

Die Abfrage $s = k$ zum Abbruch der UNTIL - Schleife in Zeile (7) des Algorithmus wird benötigt, wenn für alle Spalten der Transformationsmatrix festgestellt wurde, daß es sich dabei um Relationen für $\underline{a}_1, \ldots, \underline{a}_k$ handelt. Dieser Fall kann eintreten, wenn die Präzision mit der gerechnet wurde, zu gering war, oder wenn eine untere Schranke, die nicht auf das Problem abgestimmt war, für den kürzesten Nichtnullvektor des Gitters verwendet wurde.

Schließlich möchten wir noch verifizieren, ob die durch den RLLL - Algorithmus berechneten Basisvektoren den Längenanforderungen (siehe Formel (3.11)) entsprechen. Dazu liefert uns das folgende Korollar ein einfaches Testkriterium:

Korollar 3.8 *Die Basisvektoren aus Ungleichung (3.11) genügen folgenden Abschätzungen:*

$$\prod_{i=1}^{r} \|b_i'\| \leq (n+2)^r \cdot 2^{\frac{r\cdot(r-1)}{2}} \cdot d(L), \tag{3.18}$$

und

$$\|b_1'\| \leq (n+2) \cdot 2^{\frac{r-1}{2}} \cdot d(L)^{\frac{1}{r}} \tag{3.19}$$

(siehe dazu auch [BuKe91]).

Darauf aufbauend kann man leicht die Funktion TEST_RESULT(L) entwickeln, die prüft, ob Korollar 3.8 erfüllt ist.

Algorithmus 3.9 TEST_RESULT(L)

Parameter: L gegeben durch $\underline{b}_1', \ldots, \underline{b}_r' \in \mathbb{R}^n$, die Basisvektoren von L, die man nach zweimaliger Anwendung von RLLL auf L erhält,

Ergebnis: 1 , falls Korollar 3.8 erfüllt,
0 , falls Korollar 3.8 nicht erfüllt.

$det \leftarrow$ DET_COMPUTE(L); /* Berechnet $d(L)$ */		(1)
$prod \leftarrow 1$; /* Initialisierung */		(2)
FOR $i \leftarrow 1$ TO r		(3)
	$prod \leftarrow prod \cdot \|b_i'\|$;	(4)
IF $(prod \leq (n+2)^r \cdot 2^{\frac{r\cdot(r-1)}{2}} \cdot det)$ and $(\|b_1'\| \leq (n+2) \cdot 2^{\frac{r-1}{2}} \cdot det^{\frac{1}{r}})$		(5)
THEN	$RETURN(1)$;	(6)
ELSE	$RETURN(0)$;	(7)

Dabei eignet sich für die Prozedur COMPUTE_DET(L), die die Determinante von L berechnet, zum Beispiel der Algorithmus von Cholesky (siehe [Ra86]).

Kapitel 4

Praktische Ergebnisse

In diesem Kapitel gehen wir nun auf die praktischen Erfahrungen ein, die wir mit dem RLLL - Algorithmus gemacht haben. Dazu stellen wir folgende drei Fragen:

1. Wie reagiert der Algorithmus auf Abweichungen des in der Theorie vorgeschlagenen Wertes der binären Präzision q zur Bildung der rationalen Approximationen?
   Zur Beantwortung dieser Frage wurde getestet, wie klein man q wählen darf, ohne daß sich die im Algorithmus berechnete Transformationsmatrix T ändert.

2. Wie reagiert in Abhängigkeit der Ergebnisse von Frage 1 der Algorithmus auf Änderungen der zur Rechnung verwendeten dezimalen Präzision *prec*?
   Zur Beantwortung dieser Frage wurde bei Wahl des kleinstmöglichen q getestet, wie stark die zur Rechnung verwendete dezimale Präzision noch verringert werden darf.

3. Bis zu welcher Gitterdimension lassen sich die Probleme der Reduktion von $\mathbb{Z}$-Basen von Ordnungen mit dieser Implementierung lösen?

Zur Beantwortung dieser Fragen wurden die Testergebnisse in den folgenden Abschnitten tabelliert. Dabei wurde lediglich noch die Version des RLLL - Algorithmus betrachtet, die den LLLFP - Algorithmus in FP - Darstellung einbindet (=: Version a)), denn wie man in Kapitel 2, Tabelle 2.2 sehen kann, ist der Laufzeitvorteil dieser Version beträchtlich. Die Version, die den LLLFP - Algorithmus in MP - Darstellung einbindet (=: Version b)), wurde parallel zu Version a) nur weiterentwickelt, um in kritischen Fällen einen Stabilitätsvergleich zu haben.

Die betrachteten Beispiele wurden mit Hilfe des Computeralgebrasystems PARI-GP [BaBeCoOl92] erzeugt. Dabei war die Vorgehensweise die Folgende:

- Eingabe eines Polynoms und Berechnung der Nullstellen des Polynoms,

- Berechnung des Konjugiertenvektors,
- Berechnung des zugehörigen Gitters,
- Schrittweise Hinzunahme linear abhängiger Vektoren.

Zunächst muß darauf hingewiesen werden, daß die "bigfloat" - Arithmetik intern zur Rechnung 28 Dezimalstellen mehr verwendet, als man explizit angibt. Das hat zur Folge, daß die dezimale Präzision darauf aufbauender Programme mindestens 28 ist. In der vorliegenden Implementierung wurde diese Tatsache wie folgt berücksichtigt:

- Ist eine dezimale Präzision $prec \leq 28$ gewünscht, so wird $prec$ auf 0 gesetzt, d.h. es wird genau mit Präzision 28 gerechnet.
- Ist eine dezimale Präzision $prec > 28$ gewünscht, so wird $prec$ auf $prec - 28$ gesetzt, was bedeutet, daß genau mit der gewünschten Präzision $prec$ gerechnet wird.

In beiden Fällen wird zur Ausgabe die dezimale Präzision auf den ursprünglich gewünschten Wert zurückgesetzt, denn die zusätzlichen 28 Dezimalstellen werden nicht angezeigt sondern stehen lediglich für interne Rechenvorgänge zur Verfügung.

Als erstes betrachten wir die Fälle, in denen die dezimale Präzision $prec$ aufgrund der theoretischen Betrachtungen in Kapitel 3, Abschnitt 3.2 und in Kapitel 2, Abschnitt 2.2 kleiner oder gleich 28 gesetzt wurde. In diesen Fällen bewirkt eine Verkleinerung der binären Präzision q gegenüber dem theoretisch berechneten Wert, der sich aus Formel (3.15) ergibt, und der von der Prozedur COMPUTE_POWER_OF_TWO(...) berechnet wird nichts, da keine dezimale Präzision kleiner als 28 verwendet werden kann. Tabelle 4.1 gibt einen Überblick über die durchschnittlich aufgetretenen Laufzeiten auf einer Sparc ELC (21 Mips) mit 32 MB RAM bei Beispielen, bei denen die theoretisch berechnete dezimale Präzision kleiner als 28 ist. Da die Auflösung der

Matrixdimension	Gitterdimension	Laufzeit in 1/100 sec.
2×2	2	16
2×3	2	20
3×3	3	21
3×4	3	25
3×5	3	34
3×6	3	47
5×5	5	38

Tabelle4.1: Laufzeiten bei $prec = 28$

Zeitmessungsroutine 1/100 sec. beträgt, ist diese Messung relativ ungenau, denn die Ergebnisse liegen etwa in der gleichen Größenordnung. Dennoch erkennt man, daß die Laufzeit im wesentlichen von der Dimension der Eingabematrix abhängt.

Als nächstes betrachten wir nun die Beispiele, bei denen der in der Theorie berechnete Wert für *prec* größer als 28 war. Hier zeigt die Praxis, daß der in der Theorie berechnete Wert der binären Präzision q in den meisten Fällen zu hoch ist. Dazu betrachten wir im folgenden tabellarische Gegenüberstellungen bei Gittern verschiedener Dimensionen. Unter "kleinstmöglichem q" beziehungsweise "kleinstmöglichem *prec*" verstehen wir dabei das kleinste q beziehungsweise die kleinste Präzision *prec*, bei der die gleiche Transformationsmatrix T berechnet wird wie mit dem in der Theorie berechneten q beziehungsweise dem in der Theorie berechneten *prec*.

Matrix-dimension	r	*prec* in Theorie	*prec* bei kleinst-möglichem q	kleinstmögliches q mit der hiesigen Implementierung	kleinstmögliches *prec*
5×5	5	33	28	1	28
5×15	5	47	28	1	28
10×10	10	43-71	28	1	28
10×10	10	99	39	17	39
10×11	10	44-46	28	1	28
10×15	10	50-52	28	1	28
10×20	10	59	33	1	28
20×20	20	77-113	43-45	1	28
20×30	20	127	55	1	28
30×30	30	161-167	63	1	50-54
30×30	30	207	73	10	73
30×30	30	235	87	34	87
30×40	30	177	73	1	56
40×40	40	237-245	83-85	1	79-83
40×40	40	247	85	4	84
40×50	40	255	93	1	85
50×50	50	319	109	10	109
50×50	50	333	117	24	117
60×60	60	385	133	17	132
60×60	60	407	143	34	143
70×70	70	491	165	37	165

Tabelle4.2: Vergleich der benötigten Präzision in Theorie und Praxis

Wie man aus Tabelle 4.2 ersehen kann, genügt bis auf wenige Ausnahmen bei alleiniger Variation von q in der Praxis als dezimale Präzision *prec* ein Wert von ca. $\frac{2}{5}$ bis $\frac{1}{2}$ des in der Theorie berechneten Wertes von *prec*. Ausnahmen liegen zum Beispiel dann vor, wenn in der Praxis die dezimale Präzision 28 möglich ist.

Tabelle 4.3 stellt das Verhalten der Präzisionen bei einem festen Gitter und schrittweiser Hinzunahme linear abhängiger Vektoren dar. Es fällt auf, daß der kleinstmögliche Wert für q konstant ist. Dieses Ergebnis ist erstaunlich, da der Wert für q in der Theorie direkt von der Dimension des Erzeugendensystems abhängt. Dennoch konnte das Phänomen bei allen Beispielen beobachtet werden.

Matrix-dimension	*prec* in Theorie	*prec* bei kleinst-möglichem q	kleinstmögliches q mit der hiesigen Implementierung	kleinstmögliches *prec*
40×40	237-245	83-85	1	79-83
40×41	242	84	1	83
40×42	245	85	1	83
40×43	246	86	1	84
40×44	247	87	1	83
40×45	248	88	1	85
40×46	249	89	1	85
40×47	250	90	1	84
40×48	251	91	1	84
40×49	254	92	1	84
40×50	255	93	1	85

Tabelle4.3: Ein Gitter der Dimension 40

In den Tabellen 4.4 und 4.5 werden die von dem RLLL - Algorithmus benötigten Laufzeiten bei der in der Theorie berechneten Präzision und bei der in der Praxis ausreichenden Präzision verglichen. Dabei kann man aus der Tabelle 4.4 ersehen, daß der Algorithmus wie erwartet bei geringerer Präzision kürzer läuft. Viel stärker fällt jedoch auf, daß bei quadratischen Eingabematrizen die Laufzeit frappierend geringer ist als bei Gittern gleicher Dimension, deren Erzeugendensystem lineare Abhängigkeiten enthält. Tatsächlich läßt sich feststellen, daß die Laufzeit mit der Anzahl der zu eliminierenden linearen Abhängigkeiten in starkem Maße wächst. Dazu werden in Tabelle 4.5 die Zeiten aufgelistet, die bei einem festen Beispielgitter der Dimension 40 gemessen wurden. Jede Zeile der Tabelle entspricht Laufzeitmessungen bei diesem Gitter, wobei jeweils ein von den anderen Vektoren linear abhängiger hinzugenommen wurde. Vergleicht man die Tabellen 2.2, 4.1 und 4.4, so stellt man wie erwartet fest, daß der LLLFP - Algorithmus den größten Anteil an der Laufzeit des RLLL - Algorithmus hat, nämlich bis zu 99%. Im Durchschnitt beträgt der Laufzeitanteil des LLLFP - Algorithmus an der Laufzeit des RLLL - Algorithmus jedoch lediglich 45%.

Matrixdimension	Gitterdimension	hh:mm:ss bei prec in Theorie	hh:mm:ss bei prec in Praxis
5×5	5	00: 00 :00.29	00: 00 :00.30
5×15	5	00: 00 :06	00: 00 :06
10×10	10	00: 00 :01	00: 00 :01
10×20	10	00: 00 :29	00: 00 :26
20×20	20	00: 00 :04	00: 00 :03
20×30	20	00: 04 :56	00: 04 :01
30×30	30	00: 00 :20	00: 00 :14
30×40	30	00: 14 :46	00: 12 :08
40×40	40	00: 00 :10	00: 00 :08
40×50	40	00: 47 :41	00: 35 :45
50×50	50	00: 02 :10	00: 01 :13
60×60	60	00: 00 :36	00: 00 :28
70×70	70	00: 00 :37	00: 00 :11

Tabelle4.4: Laufzeitvergleich RLLL

Matrixdimension	hh:mm:ss bei prec in Theorie	hh:mm:ss bei prec in Praxis
40×40	00: 00 :10	00: 00 :08
40×41	00: 01 :01	00: 00 :49
40×42	00: 02 :56	00: 02 :08
40×43	00: 05 :28	00: 03 :58
40×44	00: 09 :01	00: 06 :29
40×45	00: 13 :06	00: 09 :37
40×46	00: 24 :13	00: 13 :20
40×47	00: 23 :46	00: 17 :36
40×48	00: 30 :54	00: 22 :51
40×49	00: 40 :16	00: 29 :03
40×50	00: 47 :41	00: 35 :45

Tabelle4.5: Laufzeitvergleich RLLL bei einem Gitter der Dimension 40

Wie aus den Tabellen ersichtlich ist, haben wir bisher Gitter bis zur Dimension 70 getestet, ohne dabei an die Grenzen der Implementierung zu stoßen. Es ist jedoch auch klar, daß die Grenzen der Implementierung erst in zweiter Linie von der Gitterdimension abhängen, in erster Linie aber von der Dimension der Eingabematrix und damit direkt von der Bereitstellung des erforderlichen Speicherplatzes. Die Größe des von unserer Implementierung benötigten Speicherplatzes in Abhängigkeit der Dimension der Eingabematrix und der Präzision $prec$ geht aus Tabelle 4.6 hervor.

Matrixdimension	Gitterdimension	$prec$	Speicherplatz in kB
10×10	10	71	35
10×20	10	59	71
30×30	30	167	491
30×40	30	177	656
50×50	50	333	2387
60×60	60	407	4129
70×70	70	491	6564

Tabelle4.6: RLLL: Speicherplatzverbrauch

Weiterhin ist von Interesse, wie oft der zweite Durchlauf des Algorithmus wirklich benötigt wurde. Dazu listen wir in Tabelle 4.7 Interpretationen der möglichen Ergebnisse für die Transformationsmatrix (T) beim 1-ten beziehungsweise 2-ten Durchlauf auf. Von 82 mit dieser Implementierung getesteten Beispielen ist 43 mal Fall 1 eingetreten, 34 mal Fall 2 und 5 mal Fall 3. Prozentual betrachtet heißt das: Fall 1 trat in 52% auf, Fall 2 in 42% und Fall 3 in 6% aller Fälle.

Fall	1. Durchlauf	2. Durchlauf	Interpretation
1	$T \neq I$	$T \neq I$	$(\underline{a}_1, \ldots, \underline{a}_k)$ $\big\downarrow$ 1. Durchlauf $(\underline{b}_1, \ldots, \underline{b}_r)$ $\big\downarrow$ 2. Durchlauf $(\underline{b}'_1, \ldots, \underline{b}'_r)$ $\underline{b}'_j$ genügt Formel 3.11 für $1 \leq j \leq r$
1	$T \neq I$	$T = I$	$(\underline{a}_1, \ldots, \underline{a}_k)$ $\big\downarrow$ 1. Durchlauf $(\underline{b}_1, \ldots, \underline{b}_r)$, $r = k$ möglich, aber nicht notwendig $\big\downarrow$ 2. Durchlauf $(\underline{b}_1, \ldots, \underline{b}_r)$ $\underline{b}_j$ genügte schon nach 1. Durchlauf Formel 3.11 für $1 \leq j \leq r$
1	$T = I$	$T = I$	$(\underline{a}_1, \ldots, \underline{a}_k)$ $\big\downarrow$ 1. Durchlauf $(\underline{a}_1, \ldots, \underline{a}_r)$, es gilt: $r = k$ $\big\downarrow$ 2. Durchlauf $(\underline{a}_1, \ldots, \underline{a}_r)$ $\underline{a}_j$ genügt Formel 3.11 für $1 \leq j \leq r$

Tabelle4.7: Interpretation der Transformationsmatrix

Kapitel 5

Programmdokumentation

Zur Implementierung der in den vorigen Kapiteln beschriebenen Algorithmen wurden die Programmiersprache C++ und eine in ihr durch Thomas Papanikolaou mit der libF - Bibliothek [Pa93] zur Verfügung gestellte MP - Arithmetik verwendet. Diese Arithmetik liefert uns den Datentyp “bigfloat”, der einen multiprecision - Datentyp darstellt (als integer und float verwendbar). Die Verwendung dieses Datentyps erfordert eine Analyse der zur Rechnung benötigten dezimalen Pazision *prec*. Dieser Analyse werden wir uns in diesem Kapitel als erstes zuwenden.

Weiterhin sind im Rahmen dieser Arbeit aufbauend auf dem Datentyp “bigfloat” die Klassen

- Fvector und
- Fmatrix

und aufbauend auf dem Datentyp “double” die Klassen

- vector und
- matrix

entstanden.

Im folgenden werden die zugrundeliegenden Datenstrukturen erläutert. Daraufhin werden die Vektor- und Matrizenarithmetiken vorgestellt und die Library - Struktur erläutert, mit der die entstandenen Funktionen (sowohl arithmetische als auch höhere) zur Verfügung gestellt werden. Die Funktionen selbst sind ausführlich dokumentiert und ihre Programmlistings werden im Anhang im Einzelnen aufgeführt. Des weiteren erklären wir kurz die Erzeugung der Libraries und des ausführbaren Programmes `rlll`, und schließlich erläutern wir die von `rlll` erwarteten Eingaben.

5.1 Berechnung der benötigten dezimalen Präzision *prec*

In diesem Abschnitt erläutern wir kurz, wie das ausführbare Programm `rlll` die zur Rechnung benötigte dezimale Präzision *prec* berechnet. Dabei hängt *prec* natürlich von dem jeweils betrachteten Eingabegitter ab.
Aus Kapitel 3, Abschnitt 3.2 geht hervor, daß der RLLL - Algorithmus zur Rechnung die binäre Präzision q benötigt. Dabei ist allerdings noch nicht berücksichtigt, daß in Unterprozeduren möglicherweise Zwischenergebnisse enstehen, deren binäre Länge größer ist als q. Auch diese Ergebnisse müssen verwaltet werden können. Dieses Phänomen kann bei Additions - und Multiplikationsoperationen auftreten, daher also insbesondere bei Skalarprodukt - oder Normberechnungen. Da in diesem Zusammenhang unter den aufgezählten Operationen die Skalarproduktberechnung die kritischste ist, genügt es, die benötigte dezimale Länge für diesen Fall abzuschätzen:
Seien $\underline{a}, \underline{b} \in \mathbb{Z}^n$, und sei

$$s = \max_{i=1,\ldots,n} \{ \lfloor \log_{10}(a_i) + 1 \rfloor,\ \lfloor \log_{10}(b_i) + 1 \rfloor;\ 1 \leq i \leq n \},$$

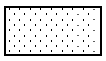
d.h. s ist die Anzahl der Dezimalstellen, des längsten Eintrages der Vektoren $\underline{a}$ und $\underline{b}$; dann hat $erg = < \underline{a}, \underline{b} >$ höchstens $2 \cdot s + n - 1$ Dezimalstellen (siehe auch Kapitel 2, Abschnitt 2.2).
Das Programm `rlll` berechnet dieser Analyse entsprechend die zur Rechnung benötigte dezimale Präzision *prec* als

$$prec = 2 \cdot (\lfloor \log_{10}(2^q) + 1 \rfloor + \max\{ \lfloor \log_{10}(a_{i,j}) + 1 \rfloor;\ 1 \leq i \leq n,\ 1 \leq j \leq k \}) + n - 1.$$

Möchte man diesen Wert von *prec* verändern, so kann man das - wie in Abschnitt 5.5 beschrieben - während des Programmablaufs.
Nun zu den Datenstrukturen:

5.2 Die Datenstrukturen

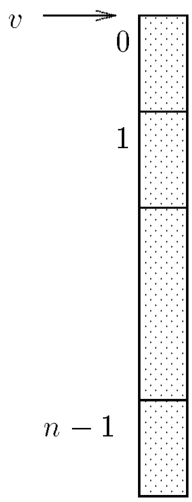
Die Grunddatentypen "double" und "bigfloat" werden wir als "Black-Box" verwenden und mit folgenden Symbolen kennzeichnen:

double: 

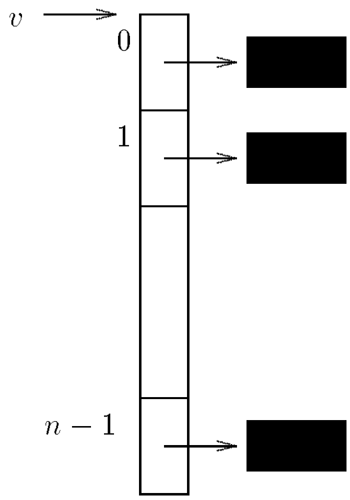
bigfloat: 

Ein Vektor $v(n), n \in \mathbb{N}$ ist ein Zeiger auf ein Feld der Größe n des jeweiligen Grunddatentypes. Graphisch veranschaulicht heißt das:

- vector $v(n)$:



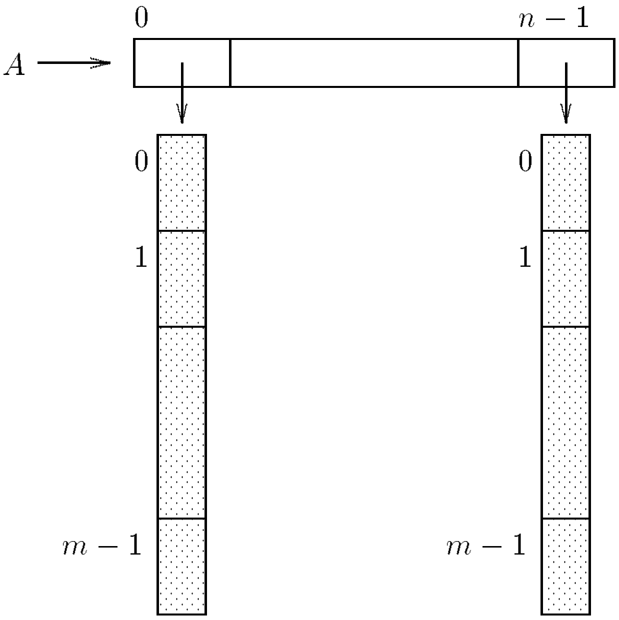
- Fvector $v(n)$:



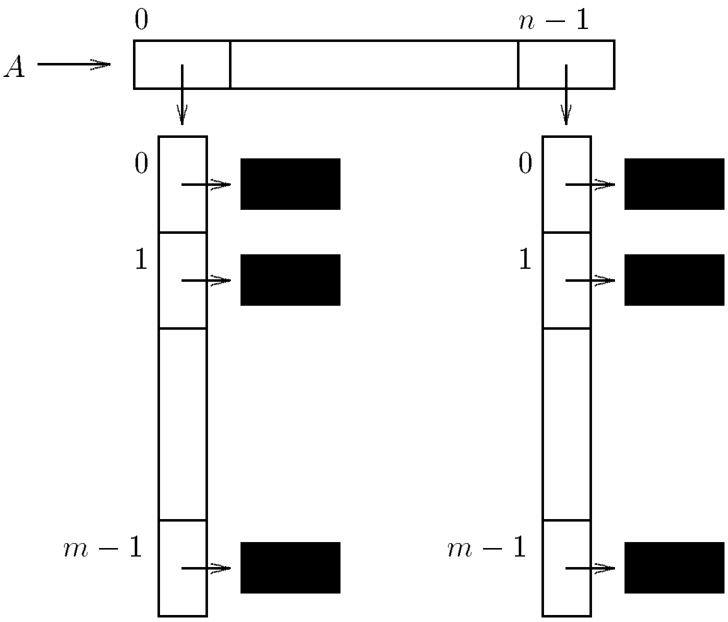
Dabei wird deutlich, daß der Typ "double" ein Standarddatentyp ist, während der Typ "bigfloat" ein benutzerdefinierter Datentyp ist und daher auch Pointerstruktur hat.

Analog zu den Datenstrukturen für Vektoren wird eine Matrix $A(n, m), n, m \in \mathbb{N}$ als Zeiger auf ein Feld von n Vektoren der Größe m definiert.
Hierzu auch wieder die graphische Veranschaulichung:

- matrix $A(n, m)$:



- Fmatrix $A(n, m)$:



Die Klassendefinitionen der Klassen

vector

Fvector

matrix

und Fmatrix

sind dann die folgenden:

```
/**************************************/***********************/
/*                                   */*                        */
/* Include - file for vector operations */*      19.03.92       */
/*                                   */*                        */
/**************************************/***********************/
/*                                                              */
/* There are two kinds of implementations of vector - types,    */
/* namely                                                       */
/*             1) vector, which has double as basis - type      */
/*             2) Fvector, which has bigfloat as basis - type   */
/*                                                              */
/****************************************************************/

#ifndef _Fvector_h /* This preprocessor instructions assure that */
#define _Fvector_h /* vector.h is included only once            */

#ifndef _big_ext_h
#   include "big_ext.h"
#endif

#include <stream.h>
#include <fstream.h>
#include <stdlib.h>

/************/
/* typedefs */
/************/

typedef int boolean;

/*******************************************************/
/* Implementation of a safe vector type where a vector is a */
/* pointer to double                                   */
/*******************************************************/

class vector {
private:
   double* p; // base pointer to double
   int size;  // size = vectordimension
              // size can only be used by member - functions
public:
   int vectdim;  // The vectordimension
   //constuctors and destructor
   vector();     // creates a size MAXROWS vector
```

```
   vector(int);  // dynamic storage allocation
                 // for a vector type
   vector(vector&);   // copy - constructor
   virtual ~vector(); // destructor
   // overloading operators
   // I/O - functions
   friend istream& operator >>(istream&, vector&);
   friend ostream& operator <<(ostream&, vector&);
   // arithmetic operations
   // multiplication of a vector with a scalar of type double
   friend vector operator *(vector&, double);
   friend vector operator *(double, vector&);
   // multiplication of a vector with a scalar of type int
   friend vector operator *(vector&, int);
   friend vector operator *(int, vector&);
   vector operator +(vector&);  // element by element addition
                                // of two vectors
   vector operator -(vector&);  // element by element subtraction
                                // of two vectors
   vector& operator =(vector&); // assignes a vector to another
   // other member functions
   double& operator [](int);    // selects the range checked
                                // ith vector element
   void swap(vector&;           // swaps two vectors
   void init();                 // Initialization of a vector
                                // by zeros
};

/*****************************************************************/
/* Implementation of a safe vector type Fvector, where a Fvector */
/* is a pointer to bigfloat                                      */
/*****************************************************************/

class Fvector {
private:
   bigfloat* p; // base pointer
   int size;    // size = vectordimension
                // size can only be used by member - functions
public:
   int vectdim; // vectdim = size,
                // vectdim can be used by functions being not
                // members
   // constuctors and destructor
   Fvector();           // creates a size MAXROWS Fvector
   Fvector(int);        // dynamic storage allocation for
                        // a Fvector
```

```
   Fvector(Fvector&);  // copy - constructor
   virtual ~Fvector(); // destructor
   // overloading operators
   // I/O - functions
   friend istream& operator >>(istream&, Fvector&);
   friend ostream& operator <<(ostream&, Fvector&);
   // arithmetic - operations
   bigfloat& operator [](int i);  // selects the range checked ith
                                  // Fvector element
   Fvector& operator =(Fvector&); // assignes Fvector v to another
   // element by element addition of two Fvectors
   friend Fvector operator +(Fvector&, Fvector&);
   // element by element subtraction of two Fvectors
   friend Fvector operator -(Fvector&, Fvector&);
   // other member functions
   // multiplication of a Fvector with a scalar of type bigfloat
   friend Fvector scalmulF(Fvector& v, bigfloat& s);
   friend Fvector scalmul(Fvector& v, double& s);
   // other member functions
   void swap(Fvector& v); // swapping of two Fvectors
   void init();           // Initializes a Fvector by zeros
};

/*************************/
/* function declarations */
/*************************/

// the ordinary inner product:
double scalprod(vector&, vector&);
bigfloat scalprodF(Fvector&, Fvector&);

// Different kinds of norms
double l2_norm(vector&);     // the euclidian norm
bigfloat l2_normF(Fvector&); // the euclidian norm
double l1_norm(vector&);     // the l1 - norm
bigfloat l1_normF(Fvector&); // the l1 - norm

#endif


/**************************************/*************************/
/*                                    */*                       */
/* Include - file for matrix operations */*      11.06.92        */
/*                                    */*                       */
/**************************************/*************************/
/*                                                              */
```

```
/* From now on I will identify a lattice to the matrix which has   */
/* the lattice - vectors as its columns.                          */
/*                                                                 */
/*******************************************************************/
/*                                                                 */
/* There are two kinds of implementations of matrix - types,      */
/* namely                                                          */
/* 1) matrix, which has vector as basis - type                     */
/* 2) Fmatrix, which has Fvector as basis - type                   */
/*                                                                 */
/*******************************************************************/

// The following preprocessor - instructions assure you, that
// Fmatrix.h is included only once.

#ifndef _Fmatrix_h
#define _Fmatrix_h

// Fvector.h must be included because of the use of the
// vector - type.

#ifndef _Fvector_h
#   include "Fvector.h"
#endif

#include <math.h>

/***************************************/
/* Implementation of the matrix - type  */
/***************************************/

class matrix{
public:
   int cols,   // the number of vectors of the generating system
               // of the lattice (the number of columns in the
               // matrix)
       rows;   // the vectordimension (the number of rows)
   vector* vect;  // pointer to an array of vectors
   // constructors and destructors
   matrix();      // empty constructor
   matrix(int, int);   // dynamical store allocation
   matrix(long, long); // dynamical store allocation
   matrix(matrix&);    // copy - constructor
   virtual ~matrix();  // destructor
   // overloading operators
   friend istream& operator >>(istream&, matrix&);
```

```
   friend ostream& operator <<(ostream&, matrix&);
   // arithmetic operations
   // assignment
   matrix& operator =(matrix&);
};

/***************************************/
/* Implementation of the Fmatrix - type */
/***************************************/

class Fmatrix{
public:
   int cols,  // the number of Fvectors of the generating system
              // of the lattice
       rows;  // the vectordimension
   Fvector* vect; // pointer to an array of Fvectors
   // constructors and destructors
   Fmatrix();     // empty constructor
   Fmatrix(int, int);    // dynamical storage allocation
   Fmatrix(long, long);  // dynamical storage allocation
   Fmatrix(Fmatrix&);    // copy-constructor
   virtual ~Fmatrix();   // destructor
   // overloading operators
   friend istream& operator >>(istream&, Fmatrix&);
   friend ostream& operator <<(ostream&, Fmatrix&);
   // arithmetic operations
   // assignment
   Fmatrix& operator =(Fmatrix&);
};


/*************************/
/* function declarations */
/*************************/

// Function for Schnorrs algorithm:
void my_compute(int, Fmatrix&, Fmatrix&, Fmatrix&, Fvector&, long&,
 long&);
void my_compute(int, Fmatrix&, matrix&, matrix&, vector&, long&);
void size_reduce(int, Fmatrix&, Fmatrix&, Fmatrix&, boolean&, long);
void size_reduce(int, Fmatrix&, matrix&, matrix&, boolean&);
void lllfp(Fmatrix&, ofstream&);
void lllfp(Fmatrix&, long&, ofstream&);

// Other functions:
// Function to compute the weight factor 2^q
```

```
bigfloat alpha_compute(Fmatrix&);
bigfloat zwei_pot_q(Fmatrix&, int&, bigfloat&, long&, long&);
bigfloat gamma(int);

// The reduction algorithm
void rlll(Fmatrix&, Fmatrix&, int&, long, long, ofstream&);

// Function to compute the resulting basis - vectors
void spec_mat_mult(Fmatrix&, Fmatrix&, int, Fmatrix&);

// Function to compute the determinant of a matrix according to the
// Cholesky - algorithm
void cholesky(Fmatrix&, bigfloat&);

// Function which tests if the resulting vectors are short enough
int erg_test(Fmatrix&, int);

#endif
```

Diese Klassendefinitionen stehen in dem directory

`/home/bartholomes/C++/libF/INCLUDE`

in den Dateien `Fvector.h` und `Fmatrix.h`. Da die Definitionen der Matrixdatentypen auf denen der Vektordatentypen aufbauen, muß in `Fmatrix.h` die Datei `Fvector.h` eingebunden werden.
Da die jeweiligen Arithmetiken untrennbar mit den zugehörigen Datentypen verbunden sind, sind die arithmetischen Operationen und die Input/Output - Operatoren (I/O - Operatoren) als inline - Funktionen deklariert. Die Funktionsdefinitionen und die überladenen Operatoren findet man dann in den Dateien `vector.C`, `Fvector.C`, `matrix.C` und `Fmatrix.C`. Die zugehörigen Programmlistings sowie die include - Datei `Fglob.h`, in der globale Variablen zur dynamischen Speicherverwaltung deklariert werden, stehen im Anhang zur Verfügung. Die im Rahmen dieser Arbeit entstandenen Vektor- und Matrizenarithmetiken stellen allerdings im wesentlichen die zur Implementierung des RLLL - Algorithmus benötigten Funktionen und Operatoren zur Verfügung.

5.3 Die Library - Struktur

Die Library - Struktur, mit deren Hilfe alle entstandenen Funktionen bereitgestellt werden, orientiert sich ebenfalls an der Tatsache, daß sozusagen als "Abfallprodukt" Vektor- und Matrizenarithmetiken entstanden sind, denn sie gliedert sich in folgende Libraries:

1. `Fvector.a`,
2. `Fmatrix.a`,
3. `Frlll.a`.

zu 1) `Fvector.a` stellt als Vektorarithmetik bezüglich der Grunddatentypen "double" und "bigfloat" folgende Operatoren und Funktionen zur Verfügung:

- I/O - Operatoren,
- Selektion eines Vektorelementes,
- Zuweisen eines Vektors,
- Addition zweier Vektoren,
- Subtraktion zweier Vektoren,
- Multiplikation eines Vektors mit einem Skalar der Typen "double" beziehungsweise "bigfloat",
- Vertauschen zweier Vektoren,
- Initialisieren eines Vektors mit $\underline{0}$,
- Berechnung der l_1 - Norm eines Vektors,
- Berechnung der euklidischen (l_2 -) Norm eines Vektors,
- Berechnung des Standardskalarproduktes zweier Vektoren.

Die Syntax der hierzu überladenen Operatoren und Funktionen findet man auf den Seiten 60 - 62 und im Anhang A: "Die Vektorarithmetik".

zu 2) `Fmatrix.a` stellt aufbauend auf den Libraryfunktionen von `Fvector.a` als Matrizenarithmetik bezüglich der Grunddatentypen "double" beziehungsweise "bigfloat" folgende Operatoren und Funktionen zur Verfügung:

- I/O - Operatoren,
- Selektion eines Matrixelementes,
- Zuweisen einer Matrix.

Die Syntax der hierzu überladenen Operatoren und Funktionen findet man auf den Seiten 62 - 65 und im Anhang B: "Die Matrizenarithmetik".

zu 3) `Frlll.a` schließlich stellt aufbauend auf `Fmatrix.a` alle höheren Funktionen zur Verfügung, die zur Implementierung des RLLL - Algorithmus notwendig waren:

- `my_compute(...)` $\longmapsto$ ORTHOGONALIZEFP(...),
- `size_reduce(...)` $\longmapsto$ REDUCEFP(...),
- `lllfp(...)` $\longmapsto$ LLLFP(...),

- `gamma(`l`)` $\longmapsto$ Berechnung der l-ten Hermite - Konstante, $\quad l \in \mathbb{N}$,
- `alpha_compute` $\longmapsto$ Berechnung von α (siehe Kapitel 3.1, Formel 3.3),
- `zweipotq(...)` $\longmapsto$ COMPUTE_POWER_OF_TWO(...),
- `rlll(...)` $\longmapsto$ RLLL(...),
- `spec_mat_mult(...)` $\longmapsto$ COMPUTE_BASIS(...),
- `cholesky(...)` $\longmapsto$ Cholesky - Algorithmus (siehe [Ra86]),
- `erg_test(...)` $\longmapsto$ TEST_RESULT(...),
- `round(`z`)` $\longmapsto$ Rundung von $z \in \mathbb{R}$ nach IEEE - Standard, z vom Typ "double"
- `Fround(`z`)` $\longmapsto$ Rundung von $z \in \mathbb{R}$ nach IEEE - Standard, z vom Typ "bigfloat".

Die genaue Syntax dieser Funktionen findet man ebenfalls auf den Seiten 62 - 65 und im Anhang C: "Die höheren Funktionen".

Alle hier aufgeführten Funktionen stehen in Dateien in dem directory

`/home/bartholomes/C++/libF/FLOAT`.

Dabei ist zu beachten, daß von den Funktionen `my_compute(...)`, `size_reduce(...)` und `lllfp(...)` jeweils zwei überladene Versionen existieren:

a) mit $\mu_{i,j}, \quad 1 \leq j < i \leq n$ vom Typ "double",

b) mit $\mu_{i,j}, \quad 1 \leq j < i \leq n$ vom Typ "bigfloat".

Version a) findet man ebenfalls in oben genanntem directory, während Version b) unter gleichem Namen in dem directory

`/home/bartholomes/C++/libF`

zu finden ist.

5.4 Erzeugung des ausführbaren Programms `rlll`

Folgendes Makefile erstellt die drei beschriebenen Libraries und erzeugt das ausführbare Programm `rlll`. Dabei ist zu beachten, daß die Pfade der Macros `INCLUDE` und `LIB` bei Änderung der File - Struktur angepaßt werden müssen. In Abhängigkeit davon, ob das Makefile in dem directory

```
/home/bartholomes/C++/libF/FLOAT
```

oder in

```
/home/bartholomes/C++/libF
```

ausgeführt wird, wird für `my_compute(...)`, `size_reduce(...)` und `lllfp(...)` Version a) oder b) gewählt, und man erhält für den LLLFP - Algorithmus eine Version in FP- oder MP - Darstellung (siehe Kapitel 2.2).

```
#####################################################################
#   MAKEFILE producing the libraries Fvector.a, Fmatrix.a, Frlll.a  #
#   and linking rlll                                                #
#####################################################################
#                               12.5.92                             #
#####################################################################

PROGRAM  = rlll
PRGSOURCES = Fmain.C
VSOURCES = Fnorm.C Fscalprod.C Fvector.C vector.C norm.C scalprod.C
MSOURCES = Fmatrix.C matrix.C
LSOURCES = Flllfp.C Fmycompute.C Frlll.C Fsizereduce.C erg_test.C
spec_mat_mult.C cholesky.C Fround.C round.C
VLIB = Fvector.a
MLIB = Fmatrix.a
LLIB = Frlll.a
INCLUDE  = /home/bartholomes/C++/libF/INCLUDE
LIB  = /home/papanik/MP_PACKAGES/libF/libF.a /glob/staff/libsparc/\
libI.a
SYSLIB   = -lm -lmisc
CC       = CC
CFLAGS   = -g -I$(INCLUDE) $(LIB) $(SYSLIB)
VMEMBERS = $(VSOURCES:.C=.o)
MMEMBERS = $(MSOURCES:.C=.o)
LMEMBERS = $(LSOURCES:.C=.o)
OBJECTS  = $(PRGSOURCES:.C=.o)
```

```
all debug profile: $(VLIB) $(MLIB) $(LLIB) $(PROGRAM)

debug    := CFLAGS= -I$(INCLUDE) -g
profile  := CFLAGS= -pg -O

.KEEP_STATE:
.SUFFIXES: .C .o
.PRECIOUS: $(VLIB) $(MLIB) $(LLIB)

$(PROGRAM): $(OBJECTS)
   $(LINK.c) -o $(PROGRAM) $(OBJECTS) $(LLIB) $(MLIB) $(VLIB) $(LIB)\
$(SYSLIB)

$(VLIB): $(VLIB)($(VMEMBERS))
   ar rv $@ $?
   ranlib $@
   touch $(PRGSOURCES)

$(MLIB): $(MLIB)($(MMEMBERS))
   ar rv $@ $?
   ranlib $@
   touch $(PRGSOURCES)

$(LLIB): $(LLIB)($(LMEMBERS))
   ar rv $@ $?
   ranlib $@
   touch $(PRGSOURCES)

.C.o:
   $(COMPILE.c) $<

$(VLIB)(%.o): %.o
$(MLIB)(%.o): %.o
$(LLIB)(%.o): %.o

clean:
   rm -f $(VMEMBERS)
   rm -f $(MMEMBERS)
   rm -f $(LMEMBERS)
```

5.5 Erstellen einer Eingabedatei und Steuerung des Programms `rlll`

Zunächst führen wir die übliche Notation zur Beschreibung von Programmeingaben über Tastatur ein:

Bemerkung 5.1 Eine von einem Programm benötigte Eingabe über Tastatur schreiben wir in folgender Weise in spitzen Klammern: <`Eingabe`>.

Wir unterscheiden zwei Klassen von Eingaben, die von `rlll` benötigt werden:

- die dateigesteuerte Eingabe des zu bearbeitenden Gitters,
- Eingaben über Tastatur zur Steuerung des Programmablaufs.

zu 1) Die Eingabe des zu bearbeitenden Gitters erfolgt über eine Datei beliebigen Namens. Wir bezeichnen diese Datei im folgenden mit "`infile`". Das Programm erwartet den Namen der Datei "`infile`" als Argument aus der Kommandozeile. Die Datei `infile` muß wie in Tabelle 5.1 aufgebaut sein. Das

$k =$ Anzahl der erzeugenden Vektoren des Gitters
$n =$ Dimension der Gittervektoren
$a_{1,1}$
$a_{2,1}$
$\vdots$
$a_{n,1}$
$a_{1,2}$
$\vdots$
$a_{n,2}$
$\vdots$
$a_{1,n}$
$\vdots$
$a_{n,n}$

Tabelle5.1: Exemplarische Eingabedatei

Programm `rlll` erwartet in der ersten Zeile der Datei `infile` die Anzahl k der erzeugenden Vektoren des Gitters, in der zweiten Zeile die Dimension n der Vektoren und in jeder folgenden Zeile die Vektorkomponenten in ihrer natürlichen Reihenfolge, d.h. von 1 bis n. Die Reihenfolge der Vektoren kann dabei beliebig sein. Das Programm erzeugt dann eine Ausgabedatei, die wir hier mit "`outfile`" bezeichnen. Den Namen der Datei "`outfile`" erwartet das Programm ebenfalls als Argument aus der Kommandozeile. Diese Ausgabedatei enthält die zur Rechnung verwendete Präzision, die berechnete Transformationsmatrix und die gemessene Laufzeit.

Der Aufruf des Programms `rlll` ist:

`rlll infile outfile`.

Dabei ist zu beachten, daß die Eingabedatei `infile` das erste Argument ist. Erscheint sie erst als zweites Argument, so kann sie zerstört werden.

Beispiele, die mit Hilfe des PARI-GP (s. [BaBeCoOl92]) erzeugt und in eine Datei geschrieben wurden, haben im Allgemeinen nicht die erforderliche oben beschriebene Form. Das ausführbare Programm `analyse` führt daher eine Transformation des PARI-GP - Formats in das Eingabeformat des Programms `rlll` durch. `analyse` erwartet auf Abfrage die Tastatureingabe der Namen einer Eingabedatei und einer Ausgabedatei. Die Eingabedatei enthält in diesem Fall das PARI-GP - Format, die Ausgabedatei das von `rlll` erforderte Format.

zu 2) Das Programm `rlll` benötigt eine Reihe von Tastatureingaben, die den Programmablauf steuern. Diese Eingaben werden durch eine entsprechende Ausgabe angefordert. Wir listen diese Ausgaben und die jeweils dazu möglichen Eingaben in Form eines Blockdiagramms in Tabelle 5.2 auf, wobei jedem Block eine Eingabe oder eine Ausgabe entspricht. Da das Programm `rlll` auch zur Bestimmung des Ranges der Einheitengruppe verwendet werden kann, werden auch dahingehende Eingaben erwartet. Diese brauchen aber bei der Verwendung des Programms im Sinne der vorliegenden Arbeit nicht beachtet zu werden. Es ist in der nachfolgenden Tabelle also grundsätzlich der Fall "default = conjugate vectors" zu wählen.

<table>
<tr><td colspan="2">Which problem is given?
default = conjugate vectors
1 = logarithm vectors</td></tr>
<tr><td colspan="2">$< z >$, $z \in \mathbb{N}$, $z \neq 1$,
man hat eine Aufgabe aus dem Gebiet der Reduktion von $\mathbb{Z}$ - Basen von Ordnungen</td></tr>
<tr><td colspan="2"><1>, man hat als Aufgabe die Bestimmung des Ranges der Einheitengruppe</td></tr>
<tr><td colspan="2">binary length of 2^q, $q = z$, $z \in \mathbb{N}$
Do you want to change it? (y=yes/<return>=no)</td></tr>
<tr><td colspan="2"><return>, man ist mit der binären Präzision einverstanden und möchte sie nicht ändern</td></tr>
<tr><td colspan="2"><y>, man möchte die binäre Präzision ändern</td></tr>
<tr><td></td><td>Enter the new value of q:</td></tr>
<tr><td></td><td>$< z >$, $z \in \mathbb{N}$</td></tr>
<tr><td colspan="2">The precision is now: z, $z \in \mathbb{N}$
Do you want to change it? (y=yes/<return>=no)</td></tr>
<tr><td colspan="2"><return>, man ist mit der dezimalen Präzision einverstanden und möchte sie nicht ändern</td></tr>
<tr><td colspan="2"><y>, man möchte die dezimale Präzision ändern</td></tr>
<tr><td></td><td>Enter the new value of prec:</td></tr>
<tr><td></td><td>$< z >$, $z \in \mathbb{N}$</td></tr>
</table>

Tabelle5.2: Steuerung des Programms über Tastatureingaben

Anhang A

Die Vektorarithmetik

A.1 aufbauend auf dem Standarddatentyp “double”

```
/******************************************************************************/
/*                               13.3.92                                      */
/******************************************************************************/
/* This file contains the C++ - code for arithmetic operations with vectors   */
/******************************************************************************/

#include "Fvector.h"

extern int MAXROWS;

/******************************************************************************/
/* constructors and destructor                                                */
/******************************************************************************/

/******************************************************************************/
/* Dynamic storage allocation for vectors (1 - dim. array)                    */
/******************************************************************************/
vector::vector(int n)
{
   if (n <= 0) // Error handling
   {
      cerr << "In v_new.C:\nIllegal vector size: " << n << "\n";
      exit(1);
   }
   vectdim = size = n;   // the vectordimension
   p = new double[size]; // Allocation of a vector of double
}

/******************************************************************************/
/* Storage allocation for vectors of size MAXDIM. This function can be used   */
/* for real dynamic storage allocation because one can read MAXDIM from stdin */
/* before calling this function                                               */
/******************************************************************************/
vector::vector()
{
   if (MAXROWS < 0) // Error - handling
   {
      cerr << "Empty constructor:\nIllegal vectordimension " << MAXROWS << "\n";
```

```
      exit(1);
   }
   vectdim = size = MAXROWS; // the vectordimension
   p = new double[size];     // Allocation of a vector of double
}

/*****************************************************************************/
/* copy-constructor                                                          */
/*****************************************************************************/
vector::vector(vector& v)
{
   vectdim = size = v.size;
   p = new double[size];
   for (int i = 0; i < size; ++i) // initialization by vector v
      p[i] = v.p[i];
}

/*****************************************************************************/
/* Destructor for a vector type                                              */
/*****************************************************************************/
vector::~vector()
{
      delete [size] p;
}

/*****************************************************************************/
/* I/O - functions                                                           */
/*****************************************************************************/

/*****************************************************************************/
/* Reads a vector from stdin by overloading the operator >>                  */
/*****************************************************************************/
istream& operator >>(istream& in, vector& v)
{
   for (int i = 0; i < v.vectdim; ++i)
   {
      cout << "[" << i+1 << "]: ";
      in >> v[i];
   }
   return(in);
}

/*****************************************************************************/
/* writes a vector to stdout by overloading the operator <<                  */
/*****************************************************************************/
ostream& operator <<(ostream& out, vector& v)
{
   out << "(";
   for (int i = 0; i < v.vectdim-1; ++i)
      out << v[i] << ", ";
   out << v[v.vectdim-1] << ")\n";
   return(out);
}

/*****************************************************************************/
/* Element by element multiplication of a vector with a scalar of type double */
/*****************************************************************************/
vector operator *(vector& v, double s)
{
   vector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s;

   return (prod); // The result of the scalarmultiplication
}
```

```
vector operator *(double s, vector& v)
{
   vector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s;

   return (prod); // The result of the scalarmultiplication
}

/****************************************************************************/
/* Element by element multiplication of a vector with a scalar of type int    */
/****************************************************************************/
vector operator *(vector& v, int s)
{
   vector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s; // The scalarmultiplication

   return (prod); // The result of the scalarmultiplication
}

vector operator *(int s, vector& v)
{
   vector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s; // The scalarmultiplication

   return (prod); // The result of the scalarmultiplication
}

/****************************************************************************/
/* Element by element addition of two vectors by overloading +               */
/****************************************************************************/
vector vector::operator +(vector& v)
{
   vector sum(vectdim);

   if(vectdim != v.vectdim) // error handling
   {
      cerr << "Error! Addition of two different size vectors"
           << vectdim << " and " << v.vectdim << "\n";
      exit(1);
   }
   for (int i = 0; i < vectdim; ++i) // element by element addition
      sum.p[i] = p[i] + v.p[i];

   return (sum); // the result of the addition
}

/****************************************************************************/
/* Element by element subtraction of two vectors by overloading -            */
/****************************************************************************/
vector vector::operator -(vector& v)
{
   vector diff(vectdim);

   if(vectdim != v.vectdim) // error handling
   {
      cerr << "Error! Subtraction of two different size vectors "
           << vectdim << " and " << v.vectdim << "\n";
      exit(1);
   }
```

```
   for (int i = 0; i < vectdim; ++i) // element by element
      diff.p[i] = p[i] - v.p[i];     // subtraction

   return (diff); // the result of the subtraction
}

/*****************************************************************************/
/* Assignes the vector v to another by overloading =                         */
/*****************************************************************************/
vector& vector::operator =(vector& v)
{
   int r = (vectdim < v.vectdim) ? vectdim : v.vectdim;

   if (vectdim != v.vectdim) // warning
      cerr << "vectorassignment.C:\ncopying different size arrays!\n"
           << v.vectdim << " to " << vectdim << "\n";

   for (int i = 0; i < r; ++i) // if a vector is assigned to a smaller one there
      p[i] = v.p[i];           // are only used as many elements as are in the
                               // smaller one.

   for (i = r; i < vectdim; ++i) // if a smaller vector is assigned to a larger
      p[i] = 0;                  // one, the additional elements are initialized
                                 // by zero
   return (*this);
}

/*****************************************************************************/
/* Other usefull functions                                                   */
/*****************************************************************************/

/*****************************************************************************/
/* Overloading the operator [] for getting a vector element                  */
/*****************************************************************************/
double& vector::operator [](int i)
{
   if (i < 0 || i >= vectdim)   // error handling
   {
      cerr << "Overloading []:\nillegal vector index: " << i << "\n";
      exit(1);
   }
   return (p[i]);
}

/*****************************************************************************/
/* Swaps two elements of type vector                                         */
/*****************************************************************************/
/*                                                                           */
/* Function call: void v_swap(vector& v); where                              */
/* - v is a references to one of the vectors to be swapped                   */
/*                                                                           */
/*****************************************************************************/
/* The return - type is void                                                 */
/*****************************************************************************/
void vector::swap(vector& v)
{
   double* help;

   help = p;
   p = v.p;
   v.p = help;
}
```

```
/******************************************************************************/
/* Initialization of a vector by zeros                                        */
/******************************************************************************/
void vector::init()
{
   if (vectdim < 0)
   {
      cerr << "Initialization of a vector of dimension " << vectdim << "\n";
      exit(1);
   }
   for(int i = 0; i < vectdim; ++i)
      p[i] = 0;
}


/******************************************************************************/
/*                                     13.3.92                                */
/******************************************************************************/
/*                                                                            */
/* This file contains the C - code for computing                              */
/* - the l2_norm (the euclidean norm)                                         */
/* - the l1_norm of a vector v                                                */
/*                                                                            */
/******************************************************************************/
/* Computation of the euclidian norm of a vector v                            */
/******************************************************************************/
/*                                                                            */
/* function call: double l2_norm(vector& v), where                            */
/* - v is the vector of which to compute the norm                             */
/*                                                                            */
/******************************************************************************/
/* The norm is the return - value of this function                            */
/******************************************************************************/
/* Computation of the l1_norm of a vector v                                   */
/******************************************************************************/
/*                                                                            */
/* function call: double l1_norm(vector& v), where                            */
/* - v is the vector of which to compute the norm                             */
/*                                                                            */
/******************************************************************************/
/* The norm is the return - value of this function                            */
/******************************************************************************/

#include "Fvector.h"
#include <math.h>

double l2_norm(vector& v)
{
   double norm;

   norm = 0; // Initialisation
   for (int i = 0; i < v.vectdim; ++i)
      norm = norm + sqr(v[i]);
   norm = sqrt(norm);
   return(norm);
}

double l1_norm(vector& v)
{
   double norm;

   norm = abs(v[0]); // Initialisation
   for (int i = 1; i < v.vectdim; ++i)
      norm = norm + abs(v[i]);
   return(norm);
}
```

```
/*****************************************************************************/
/*                                  13.3.92                                  */
/*****************************************************************************/
/* Computes the ordinary inner product of two vectors.                      */
/*****************************************************************************/
/*                                                                           */
/* Function call: double scalprod(vector& v, vector& w), where               */
/* - res = (v, w) and (.,.) denotes the ordinary inner product               */
/*                                                                           */
/*****************************************************************************/
/* The value of the inner product is returned as a double - type            */
/*****************************************************************************/

#include "Fvector.h"

double scalprod(vector& v, vector& w)
{
   double res; // The result of the computation of the inner product

   if (v.vectdim != w.vectdim) // Error handling
   {
      cerr << "In scalprod.C:\nComputing the inner product of two different "
           << "size vectors, namely: " << v.vectdim << " and" << w.vectdim
           << "\n";
      exit(1);
   }
   res = 0;
   for (int i = 0; i < v.vectdim; ++i)
      res = res + v[i] * w[i];
   return (res);
}
```

A.2 aufbauend auf dem benutzerdefinierten Datentyp "bigfloat"

```
/*****************************************************************************/
/*                                   17.3.92                                 */
/*****************************************************************************/
/* This file contains the C++ - code of all member functions of the class    */
/* Fvector. This means constructors, destructor, arithmetic operations and   */
/* I/O - functions.                                                          */
/*****************************************************************************/

#include "Fvector.h"

extern int MAXROWS;

/*****************************************************************************/
/* Constructors and destructor                                               */
/*****************************************************************************/

/*****************************************************************************/
/* Storage allocation for Fvectors of size MAXROWS. This function can be used */
/* for real dynamic storage allocation because one can read MAXROWS from     */
/* stdin before calling this function. MAXROWS is defined in Fglob.h         */
/*****************************************************************************/

Fvector::Fvector()
{
   if (MAXROWS < 0) // Error - handling
   {
```

```
      cerr << "Empty constructor for Fvectors :\nIllegal vectordimension "
           << MAXROWS << "\n";
      exit(1);
   }
   vectdim = size = MAXROWS; // the vectordimension
   p = new bigfloat [size];  // Allocation of a vector of bigfloats
   for (int i = 0; i < size; ++i)
      p[i] = 0;
}

/****************************************************************************/
/* Dynamic storage allocation for a Fvector (a vector of bigfloats is a     */
/* 1 - dim. array having bigfloat as basis - type)                          */
/****************************************************************************/
Fvector::Fvector(int n)
{
   if (n <= 0) // Error handling
   {
      cerr << "Constructor:\nIllegal vector size: " << n << "\n";
      exit(1);
   }
   vectdim = size = n;      // the vectordimension
   p = new bigfloat [size]; // Allocation of a vector of bigfloat
   for (int i = 0; i < size; ++i)
      p[i] = 0;
}

/****************************************************************************/
/* Copy-constructor for a Fvector                                           */
/****************************************************************************/
Fvector::Fvector(Fvector& v)
{
   size = v.size;
   p = new bigfloat [size];
   for (int i = 0; i < size; ++i) // initialization by Fvector v
      p[i] = v.p[i];
   vectdim = size; // initialisation of the public member for the
                   // vectordimension
}

/****************************************************************************/
/* Destructor for a Fvector type                                            */
/****************************************************************************/
Fvector::~Fvector()
{
     delete [size] p;
}

/****************************************************************************/
/* I/O - functions                                                          */
/****************************************************************************/

/****************************************************************************/
/* Reads a Fvector from stdin by overloading the operator >>                */
/****************************************************************************/
istream& operator >>(istream& in, Fvector& v)
{
   for (int i = 0; i < v.vectdim; ++i)
   {
      cout << "[" << i+1 << "]: " << flush;
      in >> v.p[i];
   }
   return(in);
}
```

```
/*****************************************************************************/
/* writes a Fvector to stdout by overloading the operator <<               */
/*****************************************************************************/
ostream& operator <<(ostream& out, Fvector& v)
{
   out << "(";
   for (int i = 0; i < v.vectdim-1; ++i)
      out << v.p[i] << ", ";
   out << v.p[v.vectdim-1] << ")\n";
   return(out);
}

/*****************************************************************************/
/* arithmetic operations                                                     */
/*****************************************************************************/

/*****************************************************************************/
/* Assignes the Fvector v to another by overloading =                       */
/*****************************************************************************/
Fvector& Fvector::operator =(Fvector& v)
{
   int r = (vectdim < v.vectdim) ? vectdim : v.vectdim;

   if (vectdim != v.vectdim)   // warning
      cerr << "Fvectorassignment:\ncopying different size arrays!\n"
           << v.vectdim << " to " << vectdim << "\n";
   for (int i = 0; i < r; ++i) // if a Fvector is assigned to a smaller one
      p[i] = v.p[i]            // there are only used as many elements as are
                               // in the smaller one.
   for (i = r; i < vectdim; ++i) // if a smaller Fvector is assigned to a
      p[i] = 0;                  // larger one, the additional elements are
                                 // initialized by zero
   return (*this);
}

/*****************************************************************************/
/* Element by element addition of two Fvectors by overloading +             */
/*****************************************************************************/
Fvector operator +(Fvector& v, Fvector& w)
{
   Fvector sum(v.vectdim);

   if(v.vectdim != w.vectdim) // error handling
   {
      cerr << "Error! Addition of two different size Fvectors" << v.vectdim
           << " and " << w.vectdim << "\n";
      exit(1);
   }
   for (int i = 0; i < v.vectdim; ++i) // element by element addition
      sum.p[i] = v.p[i] + w.p[i];

   return (sum); // the result of the addition
}

/*****************************************************************************/
/* Element by element subtraction of two Fvectors by overloading -          */
/*****************************************************************************/
Fvector operator -(Fvector& v, Fvector& w)
{
   Fvector diff(v.vectdim);

   if(v.vectdim != w.vectdim) // error handling
   {
      cerr << "In bfv_sub.C:\n" << "Error! Subtraction of two different size "
           << "Fvectors " << v.vectdim << " and " << w.vectdim << "\n";
      exit(1);
```

```
   }
   for (int i = 0; i < v.vectdim; ++i) // element by element subtraction
      diff.p[i] = v.p[i] - w.p[i];

   return (diff); // the result of the subtraction
}

/****************************************************************************/
/* Element by element multiplication of a Fvector with a scalar of type     */
/* bigfloat. This operation is implemented as function, because of ambiguous */
/* interpretations by the compiler when overloading "*"                     */
/****************************************************************************/
Fvector scalmulF(Fvector& v, bigfloat& s)
{
   Fvector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s;
   return (prod);     // The result of the scalarmultiplication
}

/****************************************************************************/
/* Element by element multiplication of a Fvector with a scalar of type     */
/* double. This operation is implemented as function, because of ambiguous  */
/* interpretations by the compiler when overloading "*"                     */
/****************************************************************************/
Fvector scalmul(Fvector& v, double& s)
{
   Fvector prod(v.vectdim);

   for (int i = 0; i < v.vectdim; ++i)
      prod.p[i] = v.p[i] * s;
   return (prod);     // The result of the scalarmultiplication
}

/****************************************************************************/
/* Other usefull functions                                                  */
/****************************************************************************/

/****************************************************************************/
/* Swaps two elements of type Fvector                                       */
/****************************************************************************/
void Fvector::swap(Fvector& v)
{
   bigfloat* help;

   help = p;
   p = v.p;
   v.p = help;
}

/****************************************************************************/
/* Initialization of a Fvector by zeros                                     */
/****************************************************************************/
void Fvector::init()
{
   for(int i = 0; i < vectdim; ++i)
      p[i] = 0;
}

/****************************************************************************/
/* Overloading the operator [] for getting a Fvector element                */
/****************************************************************************/
bigfloat& Fvector::operator [](int i)
{
   if (i < 0 || i >= vectdim)   // error handling
```

```
   {
      cerr << "Overloading [] for Fvectors:\nillegal vector index: " << i
           << "\n";
      exit(1);
   }
   return (p[i]);
}


/****************************************************************************/
/*                                11.06.92                                  */
/****************************************************************************/
/*                                                                          */
/* This file contains the C++ - code for computing                          */
/* - the l2_norm (the euclidean norm)                                       */
/* - the l1_norm of a Fvector v                                             */
/*                                                                          */
/****************************************************************************/
/* Computation of the euclidian norm of a Fvector v                         */
/****************************************************************************/
/*                                                                          */
/* function call: bigfloat l2_normF(Fvector& v), where                      */
/* - v is the Fvector of which to compute the norm                          */
/*                                                                          */
/****************************************************************************/
/* The norm is the return - value of this function                          */
/****************************************************************************/
/* Computation of the l1_norm of a Fvector v                                */
/****************************************************************************/
/*                                                                          */
/* function call: bigfloat l1_normF(Fvector& v), where                      */
/* - v is the Fvector of which to compute the norm                          */
/*                                                                          */
/****************************************************************************/
/* The norm is the return - value of this function                          */
/****************************************************************************/
#include "Fvector.h"
#include <math.h>

bigfloat l2_normF(Fvector& v)
{
   bigfloat norm;

   norm = 0; // Initialisation
   for (int i = 0; i < v.vectdim; ++i)
      norm = norm + v[i] * v[i];
   Fsettype(norm, 1); // Fsqrt needs a real argument
   Fsqrt(norm, norm);

   return(norm);
}

bigfloat l1_normF(Fvector& v)
{
   bigfloat norm;

   norm = v[0]; // Initialisation
   Fabs(norm);
   for (int i = 1; i < v.vectdim; ++i)
   {
      bigfloat dummy;

      dummy = v[i];
      Fabs(dummy);
      norm = norm + dummy;
   }
```

```
   return(norm);
}


/*****************************************************************************/
/*                                11.06.92                                   */
/*****************************************************************************/
/* Computes the ordinary inner product of two vectors                        */
/*****************************************************************************/
/*                                                                           */
/* Function call:                                                            */
/* bigfloat scalprodF(Fvector& v, Fvector& w), where                         */
/* - res = (v, w) and (.,.) denotes the ordinary inner product               */
/*                                                                           */
/*****************************************************************************/
/* The return - type is the value res of the inner product and is of type    */
/* bigfloat                                                                  */
/*****************************************************************************/

#include "Fvector.h"

bigfloat scalprodF(Fvector& v, Fvector& w)
{
   bigfloat res = 0; // the inner product

   if (v.vectdim != w.vectdim) // Error handling
   {
      cerr << "In scalprod.C:\nComputing the inner product of two different "
           << "size Fvectors, namely: " << v.vectdim " and " << w.vectdim
           << "\n";
      exit(1);
   }
   for(int i = 0; i < v.vectdim; ++i)
   {
      res = res + v[i] * w[i];
   }
   return (res);
}
```

Anhang B

Die Matrizenarithmetik

B.1 aufbauend auf dem benutzerdefinierten Datentyp "vector"

```
/*****************************************************************************/
/*                                  13.3.92                                  */
/*****************************************************************************/
/* This file contains the C++ - code for arithmetic operations with matrixes */
/*****************************************************************************/

#include "Fmatrix.h"

extern int MAXROWS, MAXCOLS;

/*****************************************************************************/
/* Constructors and destructor                                               */
/*****************************************************************************/

/*****************************************************************************/
/* Storage allocation for matrixes of size MAXROWS x MAXCOLS. This function  */
/* can be used for real dynamic storage allocation because one can read      */
/* MAXROWS and MAXCOLS from stdin before calling this function.              */
/*****************************************************************************/

/*****************************************************************************/
/* Empty constructor                                                         */
/*****************************************************************************/
matrix::matrix()
{
   vect = new vector [MAXCOLS];  // Allocation of MAXCOLS vectors
   for (int i = 0; i < MAXCOLS; ++i)
      vect[i].vectdim = MAXROWS; // The dimension of each vector
                                 // is MAXROWS
   cols = MAXCOLS;
   rows = MAXROWS;
}

/*****************************************************************************/
/* Dynamical store allocation for a matrix - type                            */
/*****************************************************************************/
matrix::matrix(int k, int n)
{
```

```
   MAXROWS = n;            // Initialisation of the vectordimension
   vect = new vector [k];  // Allocation of k vectors
   for (int i = 0; i < k; ++i)
      vect[i].vectdim = n; // The dimension of each vector is n
   cols = k;               // k is the number of columns
   rows = n;               // n is the number of rows
}

/*****************************************************************************/
/* Copy-constructor                                                          */
/*****************************************************************************/
matrix::matrix(matrix& L)
{
   vect = new vector [L.cols];
   for (int i = 0; i < L.cols; ++i)
   {
      vect[i].vectdim = L.vect[i].vectdim;
      vect[i] = L.vect[i];
   }
   cols = L.cols;
   rows = L.rows;
}

/*****************************************************************************/
/* deletes allocated store for a matrix - type                               */
/*****************************************************************************/
matrix::~matrix()
{
     delete [cols] vect;
}

/*****************************************************************************/
/* I/O - functions                                                           */
/*****************************************************************************/

/*****************************************************************************/
/* Reads a matrix from standard input by overloading >>                      */
/*****************************************************************************/
istream& operator >>(istream& in, matrix& L)
{
   for (int i = 0; i < L.cols; ++i)
   {
      cout << "Please enter vector " << i+1 << "\n";
      in >> L.vect[i];
   }
   return (in);
}

/*****************************************************************************/
/* Writes a matrix to standard output by overloading <<. The columns of the   */
/* matrix are the lattice - vectors.                                         */
/*****************************************************************************/
ostream& operator <<(ostream& out, matrix& L)
{
   for (int i = 0; i < L.rows; ++i) // writes a matrix vector by vector
   {
      out << "(";
      for (int j = 0; j < L.cols - 1; ++j)
         out << L.vect[j][i] << ", ";
      out << L.vect[L.cols-1][i] << ")\n";
   }
   return(out);
}
```

```
/******************************************************************************/
/* Arithmetic operations                                                      */
/******************************************************************************/

/******************************************************************************/
/* Assignes a matrix to another by overloading =                              */
/******************************************************************************/
matrix& matrix::operator =(matrix& L)
{
   if (cols != L.cols) // error handling
   {
      cerr << "matrixassignment: Error! Assigning a matrix with " << L.cols
           << " generating vectors to one with " << cols
           << " generating vectors.\n";
      exit(1);
   }
   if (rows != L.rows)
   {
      cerr << "matrixassignment: Error! Assigning two matrixes of different "
           << "vectordimensions, namely: " << rows << " and " << L.rows
           << ".\n";
      exit(1);
   }
   for (int i = 0; i < cols; ++i)
      vect[i] = L.vect[i];
   return (*this);
}
```

B.2 aufbauend auf dem benutzerdefinierten Datentyp "Fvector"

```
/******************************************************************************/
/*                                  17.3.92                                   */
/******************************************************************************/
/* This file contains the C++ - code for arithmetic operations with Fmatrixes */
/******************************************************************************/

#include "Fmatrix.h"

extern int MAXROWS, MAXCOLS;

/******************************************************************************/
/* Constructors and destructor                                                */
/******************************************************************************/

/******************************************************************************/
/* Storage allocation for Fmatrixes of size MAXROWS x MAXCOLS. This function  */
/* can be used for real dynamic storage allocation because one can read       */
/* MAXROWS and MAXCOLS from stdin before calling this function.               */
/******************************************************************************/

/******************************************************************************/
/* Empty constructor                                                          */
/******************************************************************************/
Fmatrix::Fmatrix()
{
   vect = new Fvector [MAXCOLS]; // Allocation of MAXCOLS Fvectors
   for (int i = 0; i < MAXCOLS; ++i)
      vect[i].vectdim = MAXROWS; // The dimension of each vector
                                 // is MAXROWS
   cols = MAXCOLS;
```

```
   rows = MAXROWS;
}

/******************************************************************************/
/* Dynamical store allocation for a Fmatrix - type                           */
/******************************************************************************/
Fmatrix::Fmatrix(int k, int n)
{
   MAXROWS = n;    // Initialisation of the vectordimension
   vect = new Fvector [k]; // Allocation of k Fvectors
   for (int i = 0; i < k; ++i)
      vect[i].vectdim = n; // The dimension of each Fvector is n
   cols = k;    // k is the number of columns
   rows = n;    // n is the number of rows
}

/******************************************************************************/
/* Dynamical store allocation for a Fmatrix - type                           */
/******************************************************************************/
Fmatrix::Fmatrix(long k, long n)
{
   int hk = (int)k, hn = (int)n;

   MAXROWS = hn;     // Initialisation of the vectordimension
   vect = new Fvector [hk]; // Allocation of k Fvectors
   for (int i = 0; i < hk; ++i)
      vect[i].vectdim = hn; // The dimension of each Fvector is n
   cols =hk;     // k is the number of columns
   rows = hn;     // n is the number of rows
}

/******************************************************************************/
/* Copy-constructor                                                           */
/******************************************************************************/
Fmatrix::Fmatrix(Fmatrix& L)
{
   vect = new Fvector [L.cols];
   for (int i = 0; i < L.cols; ++i)
   {
      vect[i].vectdim = L.vect[i].vectdim;
      vect[i] = L.vect[i];
   }
   cols = L.cols;
   rows = L.rows;
}

/******************************************************************************/
/* destructor                                                                 */
/******************************************************************************/
Fmatrix::~Fmatrix()
{
     delete [cols] vect;
}

/******************************************************************************/
/* I/O - functions                                                            */
/******************************************************************************/

/******************************************************************************/
/* Reads a Fmatrix from standard input by overloading >>                      */
/******************************************************************************/
istream& operator >>(istream& in, Fmatrix& L)
{
   for (int i = 0; i < L.cols; ++i)
   {
      cout << "Please enter Fvector " << i+1 << "\n";
```

```
      in >> L.vect[i];
   }
   return (in);
}

/***************************************************************************/
/* Writes a Fmatrix to standard output by overloading <<. The columns of the  */
/* Fmatrix are the lattice - vectors.                                         */
/***************************************************************************/
ostream& operator <<(ostream& out, Fmatrix& L)
{
   for (int i = 0; i < L.rows; ++i) // writes a Fmatrix vector by vector
   {
      out << "(";
      for (int j = 0; j < L.cols - 1; ++j)
      out << L.vect[j][i] << ", ";
      out << L.vect[L.cols-1][i] << ")\n";
   }
   return(out);
}

/***************************************************************************/
/* Arithmetic operations                                                      */
/***************************************************************************/

/***************************************************************************/
/* Assignes a lattice to another by overloading =                             */
/***************************************************************************/
Fmatrix& Fmatrix::operator =(Fmatrix& L)
{
   if (cols != L.cols) // error handling
   {
      cerr << "assigning a Fmatrix: Error! Assigning a Fmatrix with " << L.cols
           << " generating vectors to one with " << cols
           << " generating vectors.\n";
      exit(1);
   }
   if (rows != L.rows)
   {
      cerr << "assigning a Fmatrix: Error! Assigning two Fmatrixes of different"
           << " vectordimensions, namely: " << rows << " and " << L.rows
           << ".\n";
      exit(1);
   }
   for (int i = 0; i < cols; ++i)
      vect[i] = L.vect[i];
   return (*this);
}
```

Anhang C

Die höheren Funktionen

Zur Verwendung der Funktionen `round(...)` und `Fround(...)` ist es notwendig die Include - Datei `big_ext.h` einzubinden, daher wird sie der Auflistung vorangestellt.

```
/****************************************************************************/
/*                                  19.3.92                                 */
/****************************************************************************/
/* Include - file for further functions on bigfloats                        */
/****************************************************************************/

#ifndef _big_ext_h
#define _big_ext_h

// This preprocessor - instructions assure you, that big_ext.h is included only
// once.

// bigfloat.h must be included.
#ifndef _BIGFLOAT_H
#   include "/home/papanik/MP_PACKAGES/libF/bigfloat.h"
#endif

bigfloat Fround(const bigfloat&);
double round(double&);

#endif


/****************************************************************************/
/*                                  01.06.92                                */
/****************************************************************************/
/* Function for rounding a bigfloat. The rounding is done according to the  */
/* IEEE - standard. Half numbers (x.5) are rounded to the floor             */
/****************************************************************************/
/*                                                                          */
/* Function call: double Fround(double& arg), where                         */
/* - arg is the bigfloat to be rounded                                      */
/*                                                                          */
/****************************************************************************/
/* The function returns the rounded value of arg                            */
/****************************************************************************/

#include <math.h>
#include "big_ext.h"  // an extended include - file for bigfloats
```

```
double round(double& arg)
{
   double dummy;

   if (arg > 0)
      dummy = ceil(arg - 0.5);
   else
      dummy = floor(arg + 0.5);
   return (dummy);
}


/*****************************************************************************/
/*                                  1.04.92                                  */
/*****************************************************************************/
/* Function for rounding a bigfloat. The rounding is done according to the    */
/* IEEE - standard. Half numbers (x.5) are rounded to the floor               */
/*****************************************************************************/
/*                                                                           */
/* Function call: bigfloat Fround(const bigfloat& arg), where                */
/* - arg is the bigfloat to be rounded                                       */
/*                                                                           */
/*****************************************************************************/
/* The function returns the rounded value of arg                             */
/*****************************************************************************/

#include "big_ext.h"  // an extended include - file for bigfloats

bigfloat Fround(const bigfloat& arg)
{
   bigfloat dummy;

   if (arg > 0)
      Fceil(dummy, arg - 0.5);
   else
      Ffloor(dummy, arg + 0.5);
   return (dummy);
}


/*****************************************************************************/
/*                                  11.06.92                                 */
/*****************************************************************************/
/* Computes the Gram - coefficients (the provisional results of the Gram -    */
/* Schmidt orthogonalisation - scheme) up to the stage, without computing the */
/* orthogonalisation                                                         */
/*****************************************************************************/
/*                                                                           */
/* Literatur:                                                                */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*               Science, "Lattice Basis Reduction: Improved Practical       */
/*               Algorithms and Solving Subset Sum Problems"                 */
/*               C.P.Schnorr, M.Euchner                                      */
/*               Universitaet Frankfurt                                      */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* Function call:                                                            */
/* void my_compute(int k, Fmatrix& L, matrix& Lfloat, matrix& my, vector& c,  */
/*                 long& iterations), where                                  */
/* - k is the stage (-> Literatur)                                           */
/* - L is the matrix of which to compute the Gram - coefficients             */
/* - Lfloat = L in floatingpointrepresentation                               */
/* - the my's are the Gram - coefficients                                    */
/* - the c's are the length of the orthogonalization of the input - vectors   */
```

```
/* - iterations is a variable for counting rounding error correction - steps  */
/*                                                                            */
/******************************************************************************/
/* The return - type is void                                                  */
/******************************************************************************/

#include "Fmatrix.h"

void my_compute(int k, Fmatrix& L, matrix& Lfloat, matrix& my, vector& c,
long& iterations)
{
   double kjdummy, kdummy, jdummy;

   if (k == 1)
      c[0] = scalprod(Lfloat.vect[0], Lfloat.vect[0]);
   c[k] = scalprod(Lfloat.vect[k], Lfloat.vect[k]);
   for (int j = 0; j <= k-1; ++j)
   {
      double s; // declaration

      s = kjdummy = scalprod(Lfloat.vect[k], Lfloat.vect[j]);
      kdummy = l2_norm(Lfloat.vect[k]);
      jdummy = l2_norm(Lfloat.vect[j]);
      // Here it can occur that essential digits are extinguished. In this case
      // we compute the inner product in exact representation for prevention
      if (abs(kjdummy) < exp(-52/2.0 * log(2)) * kdummy * jdummy)
      {
         ++iterations;
         cerr << "In l_mycompute.C: Achtung! Hier wird <Lfloat.vect[" << k
              << "], Lfloat.vect[" << j << "]> exakt berechnet\n" << flush;
         // s becomes a floatingpoint
         s = dblasF(scalprodF(L.vect[k], L.vect[j]));
      }
      my.vect[k][j] = s;
      for (int i = 0; i <= j-1; ++i)
         my.vect[k][j] = my.vect[k][j] - my.vect[j][i] * my.vect[k][i] * c[i];
      my.vect[k][j] = my.vect[k][j] / c[j];
      c[k] = c[k] - my.vect[k][j] * my.vect[k][j] * c[j];
   }
}


/******************************************************************************/
/*                                  23.3.92                                   */
/******************************************************************************/
/* Computes the Gram - coefficients (the provisional results of the Gram -    */
/* Schmidt orthogonalisation - scheme) up to the stage, without computing the */
/* orthogonalisation                                                          */
/******************************************************************************/
/*                                                                            */
/* Literatur:                                                                 */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*               Science, "Lattice Basis Reduction: Improved Practical        */
/*               Algorithms and Solving Subset Sum Problems"                  */
/*               C.P.Schnorr, M.Euchner                                       */
/*               Universitaet Frankfurt                                       */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Function call:                                                             */
/* void my_compute(int k, Fmatrix& L, Fmatrix& Lfloat,                        */
/* Fmatrix& my, Fvector& c, long& prec), where                                */
/* - k is the stage (-> Literatur)                                            */
/* - L is the matrix of which to compute the Gram - coefficients              */
/* - Lfloat = L in floatingpointrepresentation                                */
/* - the my's are the Gram - coefficients                                     */
```

```
/* - the c's are also provisional results of the orthogonalisation scheme     */
/* - prec is the precision to compute with                                    */
/*                                                                            */
/******************************************************************************/
/* The return - type is void                                                  */
/******************************************************************************/

#include "Fmatrix.h"

void my_compute(int k, Fmatrix& L, Fmatrix& Lfloat, Fmatrix& my, Fvector& c,
long& prec, long& iterations)
{
   bigfloat kjdummy, kdummy, jdummy;

   if (k == 1)
      c[0] = scalprodF(Lfloat.vect[0], Lfloat.vect[0]);
   c[k] = scalprodF(Lfloat.vect[k], Lfloat.vect[k]);
   for (int j = 0; j <= k-1; ++j)
   {
      bigfloat s; //declaration

      s = kjdummy = scalprodF(Lfloat.vect[k], Lfloat.vect[j]);
      kdummy = l2_normF(Lfloat.vect[k]);
      jdummy = l2_normF(Lfloat.vect[j]);
      Fabs(kjdummy);
      // Here it can occur that essential digits are extinguished. In this case
      // we compute the inner product in exact representation for prevention
      if (kjdummy < exp(-prec/2.0 * log(10)) * kdummy * jdummy)
      {
         ++iterations;
         cerr << "In l_mycompute.C: Achtung! Hier wird <Lfloat.vect[" << k
              << "], Lfloat.vect[" << j << "]> exakt berechnet\n" << flush;
         s = scalprodF(L.vect[k], L.vect[j]);
         Fsettype(s, 1); // s becomes a floatingpoint
      }
      my.vect[k][j] = s;
      for (int i = 0; i <= j-1; ++i)
         my.vect[k][j] = my.vect[k][j] - my.vect[j][i] * my.vect[k][i] * c[i];
      my.vect[k][j] = my.vect[k][j] / c[j];
      c[k] = c[k] - my.vect[k][j] * my.vect[k][j] * c[j];
   }
}


/******************************************************************************/
/*                                  11.06.92                                  */
/******************************************************************************/
/* Reduces the Fmatrix - vector L.vect[k] by L.vect[j] if |my[k][j]| > 1/2    */
/******************************************************************************/
/*                                                                            */
/* Literatur:                                                                 */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*              Science, "Lattice Basis Reduction: Improved Practical         */
/*              Algorithms and Solving Subset Sum Problems"                   */
/*              C.P.Schnorr, M.Euchner                                        */
/*              Universitaet Frankfurt                                        */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Function call:                                                             */
/* void size_reduce(int k, Fmatrix& L, matrix& Lfloat, matrix& my,            */
/* boolean* Fc), where                                                        */
/* - k is the stage (-> Literatur)                                            */
/* - L is the Fmatrix of which to compute the Gram - coefficients             */
/* - Lfloat = L in floatingpointrepresentation                                */
/* - the my's are the Gram - coefficients                                     */
```

```
/* - Fc = TRUE, if there can occur important rouding errors                  */
/*                                                                           */
/*****************************************************************************/
/* The return - type is void                                                 */
/*****************************************************************************/

#include "Fmatrix.h"

#define TRUE 1
#define FALSE 0

void size_reduce(int k, Fmatrix& L, matrix& Lfloat, matrix& my, boolean& Fc)
{
   boolean Fr = FALSE;
   int ein;
   double absdummy,
          mydummy;

   for (int j = k-1; j >= 0; --j)
   {

      absdummy = abs(my.vect[k][j]);
      if (absdummy > 0.5)
      {
        // Initialization of mydummy
        mydummy = round(my.vect[k][j]);
        Fr = TRUE;
        // Floatingpoint - machineoperations are not commutativ or assoziativ
        if ((abs(mydummy) > exp(52/2.0 * log(2))) || (abs(mydummy) > exp(52 * lo
g(2)/2.0)) )
           Fc = TRUE;
        for (int i = 0; i <= j-1; ++i)
           my.vect[k][i] = my.vect[k][i] - mydummy * my.vect[j][i];
        my.vect[k][j] = my.vect[k][j] - mydummy;
        L.vect[k] = L.vect[k] - scalmul(L.vect[j], mydummy);
      }
   }
   if (Fr)
   {
      for (int i = 0; i < L.rows; ++i)
         Lfloat.vect[k][i] = dblasF(L.vect[k][i]);
   }
}


/*****************************************************************************/
/*                                    30.3.92                                */
/*****************************************************************************/
/* Reduces the Fmatrix - vector L.vect[k] by L.vect[j] if |my[k][j]| > 1/2   */
/*****************************************************************************/
/*                                                                           */
/* Literatur:                                                                */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*               Science, "Lattice Basis Reduction: Improved Practical       */
/*               Algorithms and Solving Subset Sum Problems"                 */
/*               C.P.Schnorr, M.Euchner                                      */
/*               Universitaet Frankfurt                                      */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* Function call:                                                            */
/* void size_reduce(int k, Fmatrix& L, Fmatrix& Lfloat, Fmatrix& my,         */
/* boolean* Fc, long prec), where                                            */
/* - k is the stage (-> Literatur)                                           */
/* - L is the Fmatrix of which to compute the Gram - coefficients            */
/* - Lfloat = L in floatingpointrepresentation                               */
```

```
/* - the my's are the Gram - coefficients                                      */
/* - Fc = TRUE, if there can occur important rouding errors                     */
/* - prec is the precision with which to compuite                               */
/*                                                                              */
/********************************************************************************/
/* The return - type is void                                                    */
/********************************************************************************/

#include "Fmatrix.h"

#define TRUE 1
#define FALSE 0

void size_reduce(int k, Fmatrix& L, Fmatrix& Lfloat, Fmatrix& my, boolean& Fc,
long prec)
{
   boolean Fr = FALSE;

   for (int j = k-1; j >= 0; --j)
   {
      bigfloat absdummy;

      absdummy = my.vect[k][j];
      Fabs(absdummy);
      if (absdummy > 0.5)
      {
         bigfloat mydummy;  // declaration

         mydummy = Fround(my.vect[k][j]); // Initialization of mydummy
         Fr = TRUE;
         absdummy = mydummy;
         Fabs(absdummy);
         // Floatingpoint - machineoperations are not commutativ or assoziativ
         if (absdummy > exp(prec/2.0 * log(10)))
           Fc = TRUE;
         if (absdummy > exp(prec * log(10)/2.0))
           Fc = TRUE;
         for (int i = 0; i <= j-1; ++i)
            my.vect[k][i] = my.vect[k][i] - mydummy * my.vect[j][i];
         my.vect[k][j] = my.vect[k][j] - mydummy;
         L.vect[k] = L.vect[k] - scalmulF(L.vect[j], mydummy);
      }
   }
   if (Fr)
   {
      Lfloat.vect[k] = L.vect[k];
      for (int i = 0; i < L.rows; ++i)
         Fsettype(Lfloat.vect[k][i], 1);
   }
}


/********************************************************************************/
/*                                    10.06.92                                  */
/********************************************************************************/
/* Computes a LLL - reduced matrix - basis using the method described in       */
/* Schnorrs paper (-> Literatur)                                                */
/********************************************************************************/
/*                                                                              */
/* Literatur:                                                                   */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*               Science, "Matrix Basis Reduction: Improved Practical          */
/*               Algorithms and Solving Subset Sum Problems"                    */
/*               C.P.Schnorr, M.Euchner                                         */
/*               Universitaet Frankfurt                                         */
/*                                                                              */
```

```
/*****************************************************************************/
/*                                                                           */
/* Function call:                                                            */
/* void lllfp(Fmatrix& L, long& prec), where                                 */
/* - L is the Fmatrix of which to compute the Gram - coefficients            */
/* - prec is the precision with which to compute                             */
/*                                                                           */
/*****************************************************************************/
/* The return - type is void                                                 */
/*****************************************************************************/

#include "Fmatrix.h"

void lllfp(Fmatrix& L, ofstream& ofile)
{
  int ein;
  matrix Lfloat(L.cols, L.rows); // The floatingpoint - version of L
  matrix my(L.cols, L.cols);     // the Gram - coefficients
  vector c(L.cols);              // inner products
  long iterations = 0, red_stage = 0;

   ofile << "LLL - dim: " << L.rows << " x " << L.cols << "\n" << flush;
   int k = 1;
   for (int i = 0; i < L.cols; ++i)
      for (int j = 0; j < L.rows; ++j)
      { // Lfloat is the floatingpoint - version of L
         Lfloat.vect[i][j] = dblasF(L.vect[i][j]);
      }
   int iteration = 0;
   while (k < L.cols)
   {
      boolean Fc = TRUE;
      while (Fc)
      {
         my_compute(k, L, Lfloat, my, c, iterations);
         Fc = FALSE;
         size_reduce(k, L, Lfloat, my, Fc);
         if (Fc)
         {
            ++red_stage;
            if (k > 1)
            {
               --k;
            }
         }
      }
      if ((0.75 * c[k-1]) > (c[k] + my.vect[k][k-1] * my.vect[k][k-1] * c[k-1]))
      {
         L.vect[k].swap(L.vect[k-1]);
         Lfloat.vect[k].swap(Lfloat.vect[k-1]);
         if (k > 1)
            --k;
      }
      else
         ++k;
   }
   ofile << "#iterations: " << iterations << "\n" << flush;
   ofile << "#red_stage: " << red_stage << "\n" << flush;
}


/*****************************************************************************/
/*                                  20.3.92                                  */
/*****************************************************************************/
/* Computes a LLL - reduced matrix - basis using the method described in     */
/* Schnorrs paper (-> Literatur)                                             */
```

```
/*****************************************************************************/
/*                                                                           */
/* Literatur:                                                                */
/* To appear in: Proceedings of the FCT'91 Springer Lecture Notes in Computer */
/*               Science, "Lattice Basis Reduction: Improved Practical       */
/*               Algorithms and Solving Subset Sum Problems"                 */
/*               C.P.Schnorr, M.Euchner                                      */
/*               Universitaet Frankfurt                                      */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* Function call:                                                            */
/* void lllfp(Fmatrix& L, long& prec), where                                 */
/* - L is the Fmatrix of which to compute the Gram - coefficients            */
/* - prec is the precision with which to compute                             */
/*                                                                           */
/*****************************************************************************/
/* The return - type is void                                                 */
/*****************************************************************************/

#include "Fmatrix.h"

#define TRUE 1
#define FALSE 0

void lllfp(Fmatrix& L, long& prec, ofstream& ofile)
{
   Fmatrix Lfloat(L.cols, L.rows), // The floatingpoint - version of L
           my(L.cols, L.cols);     // the Gram - coefficients
   Fvector c(L.cols);              // inner products
   long iterations = 0, red_stage = 0;

   ofile << "LLL - dim" << L.rows << " x " << L.cols << "\n" << flush;
   int k = 1;
   for (int i = 0; i < L.cols; ++i)
      for (int j = 0; j < L.rows; ++j)
      { // Lfloat is the floatingpoint - version of L
         Lfloat.vect[i][j] = L.vect[i][j];
         Fsettype(Lfloat.vect[i][j], 1);
      }
   while (k < L.cols)
   {
      boolean Fc = TRUE;
      while (Fc)
      {
         my_compute(k, L, Lfloat, my, c, prec, iterations);
         Fc = FALSE;
         size_reduce(k, L, Lfloat, my, Fc, prec);
         if (Fc)
         {
            ++red_stage;
            if (k > 1)
              --k;
         }
      }
      if ((0.75 * c[k-1]) > (c[k] + my.vect[k][k-1] * my.vect[k][k-1] * c[k-1]))
      {
         L.vect[k].swap(L.vect[k-1]);
         Lfloat.vect[k].swap(Lfloat.vect[k-1]);
         if (k > 1)
           --k;
      }
      else
         ++k;
   }
   ofile << "#iterations: " << iterations << "\n" << flush;
```

```
   ofile << "#red_stage: " << red_stage << "\n" << flush;
}


/******************************************************************************/
/*                               15.06.92                                     */
/******************************************************************************/
/* Computation of a reduced lattice - basis from a generating system of       */
/* A.cols real lattice - vectors A.vect[1],...,A.vect[A.cols] of dimension    */
/* A.rows. The notation in the following program is choosen according to the */
/* literatur mentioned below.                                                 */
/******************************************************************************/
/*                                                                            */
/* Literatur: J. Buchmann, V. Kessler: "Computing a reduced lattice basis     */
/*            from a generating system", May 6, 1991                          */
/*                                                                            */
/******************************************************************************/
/* This file contains the following functions, which are needed for the       */
/* reduce - algorithm:                                                        */
/******************************************************************************/
/* Computation of the binary precision q                                      */
/******************************************************************************/
/*                                                                            */
/* bigfloat zwei_pot_q(Fmatrix& A, int n2, bigfloat& alpha, long& q_ten,      */
/* long& q), where                                                            */
/* - A is the lattice to be reduced                                           */
/* - n2 is the number of real elements in the lattice vectors                 */
/* - alpha = max {norm(A.vect[1]),...,norm(A.vect[A.cols])}                   */
/* - q_ten is q to the basis 10                                               */
/*                                                                            */
/******************************************************************************/
/* This function returns the value of q                                       */
/******************************************************************************/
/*                                                                            */
/* bigfloat gamma(int l), where                                               */
/* - l is the index of the Hermite - constant to be computed                  */
/*                                                                            */
/******************************************************************************/
/* The return - value of this function is the l-th Hermite - constant. In     */
/* this context l can not be to big (it is at most the number of generating   */
/* vectors), so the l-th Hermite - constant can be of type double. */
/******************************************************************************/
/*                                                                            */
/* bigfloat alpha_compute(Fmatrix& A), where                                  */
/*     - A is the lattice to reduce                                           */
/*                                                                            */
/******************************************************************************/
/* This function returns                                                      */
/* alpha = max{norm(A.vect[1]),...,norm(A.vect[A.cols])}.                     */
/* The return - value of this function is bigfloat.                           */
/******************************************************************************/
/*                                                                            */
/* void rlll(Fmatrix& A, Fmatrix& M, int& r, long prec, long elexp,           */
/* ofstream& Outputfile), where                                               */
/* - A is the lattice to reduce                                               */
/* - M is the transformation - matrix                                         */
/* - r is the dimension of the lattice                                        */
/* - prec is the precision to compute with                                    */
/* - ellength is the length of the longest matrixelement                      */
/* - Outputfile is the file in which the output is written                    */
/*                                                                            */
/******************************************************************************/
/* This function returns the transformation - matrix. The Fmatrix A is        */
/* allowed to be of integers, because it is internally converted to a         */
/* floatingpoint - matrix.                                                    */
/******************************************************************************/
```

```
#include <math.h>
#include "Fmatrix.h"

extern "C" double log2(double);    // pure C - functions
extern "C" int start_timing(int*);
extern "C" int cont_timing(int*);
extern "C" int get_timing(int*);

bigfloat zwei_pot_q(Fmatrix& A, int& n2, bigfloat& alpha, long& q_ten, long& q)
{
   int beta = A.rows,
       sign = 1,
       problem;        // To choose the given problem
   bigfloat zweipotq,  // For computing the value of 2^q
            sqrtdummy, // temporary - variable for computing the sqrt of a
                       // bigfloat
            potdummy,  // dummy - variable for computing powers
            expdummy,  // for computing exponents
            mydummy,   // lower bound for the shortest nonzerovector of L
            gammadummy;
   double lambda,      // lambda = A.cols * sqrt(n2)/2 + sqrt(A.rows)
          help;
   char answer;

   if (A.cols < A.rows)
      beta = A.cols;
   lambda = A.cols * sqrt(n2) / 2.0 + sqrt(A.cols);
   Flog(expdummy, alpha);
   expdummy = expdummy * beta;
   Fexp(potdummy, expdummy);
   gammadummy = gamma(beta);
   Fsettype(gammadummy, 1);
   Fsqrt(gammadummy, gammadummy);
   zweipotq = lambda * (sqrt(n2 * A.cols) + 2) * gammadummy;
   zweipotq = zweipotq * potdummy;
   expdummy = (A.cols - 3) / 2.0 * log(2);
   Fexp(potdummy, expdummy);
   zweipotq = zweipotq * potdummy;
   cout << "Which problem is given?\n"
        << "default = conjugate vectors\n"
        << "1 = logarithm vectors\n" << flush;
   cin >> problem;
   switch(problem)
   {
      case 1: help = log(log(A.rows))/log(A.rows);
       if (help < 0)
       {
          sign = -1;
          help = abs(help);
       }
       mydummy = sqrt(2/(beta + 1)) * (1/1200 * sign * exp(3*log(help)) - 1/2880
000 * exp(6*log(help)));
      default: mydummy = sqrt(A.rows/2.0);
   }
   Flog(expdummy, mydummy);
   expdummy = expdummy * (beta + 1);
   Fexp(potdummy, expdummy);
   zweipotq = zweipotq / potdummy;
   Fceil(zweipotq, zweipotq);
   q =(long)Fexpo(zweipotq) + (long)Flog2(zweipotq) + 1;//bin. length
   cout << "binary length of 2^q, q= " << q << "\n" << flush;
   cout << "Do you want to change it? (y=yes/<return>=no) " << flush;
   if ((answer = getchar()) == 'y')
   {
     cout << "Enter the new value of q: " << flush;
```

```
      cin >> q;
      getchar();   // consumes the surplus <return>
    }
    cout << "q is now: " << q << "\n" << flush;
    q_ten = q * log(2) / log(10) + 1;//dec. length

    return(zweipotq);
}

bigfloat gamma(int l)
{
    bigfloat gamma,
             logarg,
             exparg;

    // Computation of the l-th Hermite - Constant gamma,
    switch(l)
    {
       case 1: gamma = 1; break;
       case 2: gamma = 4.0/3.0; break;
       case 3: gamma = 2; break;
       case 4: gamma = 4; break;
       case 5: gamma = 8; break;
       case 6: gamma = 64.0/3.0; break;
       case 7: gamma = 64; break;
       case 8: gamma = 256; break;
       default: Flog(logarg, 4.0/3.0);
                Fexp(gamma, l * (l-1) / 2.0 * logarg);
                break;
    }
    return(gamma);
}

bigfloat alpha_compute(Fmatrix& A)
{
    bigfloat alpha, // alpha = max{l2_normF(A.vect[1]),...,
                    //             l2_normF(A.vect[A.cols])}
             norm;  // norm = l2_normF(A.vect[i]), 0 <= i < A.cols

    alpha = l2_normF(A.vect[0]);
    for (int i = 1; i < A.cols; ++i)
    {
       norm = l2_normF(A.vect[i]);
       if (alpha < norm)
       alpha = norm;
    }
    return(alpha);
}

// The reduction Algorithm
void rlll(Fmatrix& A, Fmatrix& M, int& r, long prec, long elexp, ofstream&
Ofile)
{
   int ein;
   Fmatrix Atilde(A.cols, A.cols + A.rows), // The approximate lattice
           Ahead(A.cols, A.rows);           // The weighted Fmatrix
   Fvector help(A.rows),  // Help - vector
           rel(A.cols);   // A candidat for a relation
   bigfloat zweipotq,
            hzweipotq,
            norm1,  // To store the l1_norm
            norm2,  // To store the l2_norm
            vor,
            rechts,
            alpha,
            hprec,
```

```
         logzwei;
int n2 = A.rows, timer[2];
long q, q_ten;
char answer; // The answer to the question whether to change the
             // precision

 Ofile << "RLLL - dim: " << A.rows << " x " << A.cols << "\n" << flush;
 timer[0] = timer[1] = 0;
 for (int i = 0; i < A.cols; ++i)    // The elements of the Fmatrix A are set
    for (int j = 0; j < A.rows; ++j) // to be real floatingpoints for having
       Fsettype(A.vect[i][j], 1);    // floatingpoint - operations
 alpha = alpha_compute(A);
 zweipotq = zwei_pot_q(A, n2, alpha, q_ten, q);
 prec = 2 * (q_ten + elexp) + Atilde.rows - 1;
 // Now the precision is the needed one for computations using elements of
 // Atilde. It is to note that with this step the precision can be diminished.
 // This is to speed up the computation.
 cout << "The precision is now: " << prec << "\n" << flush
      << "Do you want to change it? (y=yes/<return>=no) " << flush;
 if ((answer = getchar()) == 'y')
 {
   cout << "Enter the new value of prec: " << flush;
   cin >> prec;
   getchar();   // consumes the surplus <return>
 }
 if (prec > 28)      // Internally the bigfloats compute using 28
   prec = prec - 28; // decimal digits more
 else
   prec = 0;
 Fsetprec(prec);
 cout << "The precision is now: " << prec+28 << "\n" << flush;
 Ofile << "The precision is set to be: " << prec+28 << "\n" << flush;
 // Preliminary computations
 for (i = 0; i < A.cols; ++i)
    for (j = 0; j < A.rows; ++j)
    {
       Ahead.vect[i][j] =  A.vect[i][j] * zweipotq;
       Frndtoi(Ahead.vect[i][j], Ahead.vect[i][j]);
       Fsettype(Ahead.vect[i][j], 1);
    }
 for (i = 0; i < A.cols; ++i)
 {
    for (int j = 0; j < A.cols; ++j)
    {
       if (i != j)
          Atilde.vect[i][j] = 0;
       else
          Atilde.vect[i][j] = 1;
    }
    for (j = A.cols; j < A.cols + A.rows; ++j)
    {
      Atilde.vect[i][j] = Ahead.vect[i][j - A.cols];
      Fsettype(Atilde.vect[i][j], 0);
    }

 }
 // The reduction - step
 start_timing(timer);  // Initialisierung der Zeitmessung
 lllfp(Atilde, Ofile);
 get_timing(timer);    // Abfrage der Zeitmessung
 Ofile << "LLL-Zeit:\tUser: " << timer[0] << " (1/100s)\n\t\tSystem: "
       << timer[1] << flush << " (1/100)\n";
 // Computation of the dimension r of the lattice, and elimination of the
 // relations
 int l = 0;
 do
```

```
   {
      help.init(); // Initializes help with the zero - vector
      for (int j = 0; j < A.cols; ++j)
      {
         rel[j] = Atilde.vect[l][j];
         help = help + scalmulF(Ahead.vect[j], rel[j]);
      }
      norm1 = l1_normF(rel);
      norm2 = l2_normF(help);
      ++l;
      vor = sqrt(2.0 * (double)n2/(double)A.rows) * norm1;
      rechts =  sqrt(n2)/2.0 * norm1;
   }
   while ((zweipotq > vor) && (norm2 <= rechts) && (l < A.cols));
   if (l >= A.cols)
     cerr << "Gitter der Dimension 1 detektiert! Falls diese Dimension nicht "
          << "erwartet, Eingabe ueberpruefen!\n" << flush;
   r = A.cols - l + 1; // r is the dimension of the lattice
   cout << "r: " << r << "\n" << flush;
   // Computation of the transformation - matrix
   for (i = 0; i < A.cols; ++i)
      for (j = 0; j < A.cols; ++j)
         M.vect[i][j] = Atilde.vect[i][j];
}
```

```
/******************************************************************************/
/*                                  17.3.92                                   */
/******************************************************************************/
/* Computes the lattice - basis by multiplying a part of the transformation - */
/* matrix with the generating system                                          */
/******************************************************************************/
/*                                                                            */
/* Function call:                                                             */
/* void spec_mat_mult(Fmatrix& A, Fmatrix& M, int r, Fmatrix& RES), where     */
/* - A is the Fmatrix of the generating system                                */
/* - M is the transformation Fmatrix                                          */
/* - r is the dimension of the basis                                          */
/* - RES is the resulting Fmatrix                                             */
/*                                                                            */
/******************************************************************************/
/* The return type is void                                                    */
/******************************************************************************/

#include "Fmatrix.h"

void spec_mat_mult(Fmatrix& A, Fmatrix& M, int r, Fmatrix& RES)
{
  int ein;
  Fmatrix PROD(RES.cols, RES.rows);
  // Computing the basis - vectors
  for (int j = 0; j < r; ++j)
    {
      PROD.vect[j].init();
      for (int i = 0; i < A.cols; ++i)
      PROD.vect[j] = PROD.vect[j] + scalmulF(A.vect[i], M.vect[A.cols - r + j][i
]);
    }
  RES = PROD;
}
```

```
/******************************************************************************/
/*                                  06.05.92                                  */
/******************************************************************************/
/* Tests if the resulting vectors are short enough. When using this function  */
/* there is premised that the resulting vectors form a basis of the lattice   */
/******************************************************************************/
/*                                                                            */
/* Literatur:                                                                 */
/* Johannes Buchmann and Volker Kessler: "Computing a reduced lattice basis   */
/* from a generating system", Corollary 4.2                                   */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Function call:                                                             */
/* int erg_test(Fmatrix& B, int r), where                                     */
/* - B is the Fmatrix containing the resulting vectors                        */
/* - r is the dimension of the lattice                                        */
/*                                                                            */
/******************************************************************************/
/* The return - code is                                                       */
/* 1 if the vectors are short enough                                          */
/* 0 if there is at least one vector which is to long                         */
/******************************************************************************/

#include "Fmatrix.h"

extern "C" double log2(double); // log2 is a pure C - function

int erg_test(Fmatrix& B, int r)
{
   bigfloat det,        // det is the determinant of B
            bound,      // the bound to be tested
            bounddummy, // a temporary bound - variable
            sqrtdummy,  // the result of the sqrt - computation
            prod,       // the product of the l2 - norm of the basis - vectors
            exp;        // exponent

   cholesky(B, det);
   cout << "det: " << det << "\n" << flush;
   Flog(exp, det);
   exp = exp / r;
   Fexp(bound, exp);             // bound = d(L)^(1/r)
   bound = bound * (B.rows + 2); // bound = (B.rows + 2) * d(L)^(1/r)
   Flog(exp, 2.0); // log(2) as bigfloat, to have maximal precision
   exp = exp * (r - 1)/2.0;
   Fexp(bounddummy, exp); // bounddummy = 2^((r-1)/2)
   bound = bound * bounddummy;
   if (l2_normF(B.vect[0]) > bound)
       return(0);
   exp = B.rows + 2;
   Fsettype(exp, 1);
   Flog(exp, exp);   // Flog needs real arguments
   exp = r * exp;
   Fexp(bound, exp); // bound = (B.rows + 2)^r
   Flog(exp, 2.0);
   exp = exp * r * (r - 1)/2.0;
   Fexp(bounddummy, exp); // bounddummy = 2^(r * (r - 1)/2)
   bound = bound * bounddummy * det;
   prod = 1;
   for (int i = 0; i < r; ++i)
     prod = prod * l2_normF(B.vect[i]); // prod = l2_norm(B.vect[0])
                                        //      * ... * l2_norm(B.vect[r])
   if (prod > bound)
      return(0);
   return(1);
}
```

```
/*****************************************************************************/
/*                                  05.06.92                                 */
/*****************************************************************************/
/* Computation of the determinant of a lattice according to the Cholesky -   */
/* Algorithm. If the input is not a lattice - basis, we get a multiple of the */
/* determinant.                                                              */
/*****************************************************************************/
/*                                                                           */
/* Literature:                                                               */
/* - a course in computeralgebra of J. Buchmann done in WS 1991/92           */
/* - R.Rannacher: Praktische Mathematik: Vorlesungsskriptum zum              */
/*               Wintersemester 1986/87, Fachbereich Mathematik,             */
/*               Universitaet des Saarlandes                                 */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* function call:                                                            */
/* void cholesky(Fmatrix& L, bigfloat& det), where                           */
/* - L is the lattice of which to compute the determinant                    */
/* - det is the determinant of the lattice.                                  */
/*                                                                           */
/*****************************************************************************/
/* The return - type is void.                                                */
/*****************************************************************************/

#include "Fmatrix.h"
#include <math.h>

void cholesky(Fmatrix& L, bigfloat& det)
{
   bigfloat **Q, // Q = (<Q[j], Q[i]>) the Fmatrix of the inner products. <.,.>
                 // denotes the ordinary inner product. Q is stored only up to
                 // the diagonal elements because it 's a symmetric matrix
            **A; // Cholesky - matrix

   Q = new bigfloat*[L.cols]; // dynamical storage allocation for Q
   A = new bigfloat*[L.cols]; // dynamical storage allocation for A

   det = 1; //Initialisation
   for (int i = 0; i < L.cols; ++i)
   {
      Q[i] = new bigfloat[i+1];  // storage allocation for i+1 -
      A[i] = new bigfloat[i+1];  // dim. vector of bigfloat
      for (int j = 0; j <= i; ++j)
         Q[i][j] = scalprodF(L.vect[j], L.vect[i]);
      for (j = 0; j < i; ++j)
      {
         A[i][j] = Q[i][j]/A[j][j];
         for (int l = j+1; l <= i; ++l)
            Q[i][l] = Q[i][l] - A[l][j] * A[i][j] * A[j][j];
      }
      A[i][j] = Q[i][i];
      if (A[i][i] == 0)
      {
         for (int l = i; l < L.cols - 1; ++l)
            L.vect[l] = L.vect[l+1];
         --L.cols; // To compute the dimension of the lattice
         --i;      // L.vect[i] has become new values, so it must be reregarded
      }
      else
         det = det * A[i][i];
      // Now A is the Cholesky - matrix
   }
   Fabs(det);  // The determinant of the lattice is |det(A)|
   Fsettype(det, 1);
   Fsqrt(det, det); // => det must have a positive value
```

```
   // To free the storage used
   for (i = 0; i < L.cols; ++i)
   {
      delete Q[i];
      delete A[i];
   }
   delete Q;
   delete A;
}
```

Anhang D

Das Hauptprogramm

```
/***************************************************************************/
/*                                  16.06.92                               */
/***************************************************************************/
/* program to compute a transformationmatrix and a reduced lattice basis from */
/* a system of generating vectors over the reals.                          */
/***************************************************************************/

#include "Fmatrix.h"
#include "Fglob.h"
#include "string.h"

extern "C" int start_timing(int*); // pure C - functions
extern "C" int cont_timing(int*);
extern "C" int get_timing(int*);

main(argc, argv)
int argc;
char* argv[];
{
   int quant, dim, r, timer[2];
   long prec = 1, ellength, elexp = 0, mainprec, dummy;
   char path[PATHLENGTH], string[STRINGLENGTH];
   ifstream Inputfile;
   ofstream Outputfile;

   if (argc != 3)
   {
      cout << "Usage: rlll argument1 argument2\n" << flush;
      exit(1);
   }
   // reading the filename and opening the file
   strcpy(path, argv[1]);
   Inputfile.open(path, ios::in); // opens the file named by path
   if (!Inputfile)                // Tests if the file has been opened correctly
   {
      cerr << path << " could not be opened.\n" << flush;
      exit(1);
   }
   strcpy(path, argv[2]);
   Outputfile.open(path, ios::out); // opens the file named by path
   if (!Outputfile) // Tests if the file has been opened correctly
   {
      cerr << path << " could not be opened.\n" << flush;
      exit(1);
```

```
}

// reading the matrixdimension
Inputfile.getline(string, STRINGLENGTH, '\n');
quant = atoi(string);
Inputfile.getline(string, STRINGLENGTH, '\n');
dim = atoi(string);

// Storageallocation for the matrices
Fmatrix A(quant, dim);   // The given lattice
Fmatrix M(quant, quant); // The transformation - Fmatrix

// Reading the input - matrix
for (int i = 0; i < quant; ++i)
   for (int j = 0; j < dim; ++j)
   {
      Inputfile.getline(string, STRINGLENGTH, '\n');
      ellength = strlen(string);
      if (string[0] == '-') // The '-' is counted in the stringlength, but
         ellength -= 2;     // needs not to be counted for the bigfloat -
                            // precision
      else                  // Furthermore the string - end symbol '\0' is
         ellength -= 1;     // counted too by strlen()
      if (ellength > prec)  // prec = the length of the longest element
      {
         prec = ellength;
         Fsetprec(prec);
      }
      atoF(string, A.vect[i][j]);
      dummy = (long)Fexpo(A.vect[i][j]) + (long)Flog2(A.vect[i][j]) + 1;
      dummy = dummy * log(2) / log(10) + 1;
      if (elexp < dummy)
         elexp = dummy;
   }
Inputfile.close();
ellength = prec; // ellength is the length of the longest input element
timer[0] = timer[1] = 0;

prec = 2 * prec + dim - 1; // The needed precision for computing an inner
                           // product
Fsetprec(prec);
mainprec = prec;
// Now the precision is high enough for computing alpha, what is needed for
// the computation of 2^q

Outputfile << "1-ter Lauf\n" << flush;
start_timing(timer);  // Initialization for the measurement of time
rlll(A, M, r, mainprec, elexp, Outputfile);
get_timing(timer);    // Reading the measured time
Outputfile << "Zeitmessung:\tUser: " << timer[0] << " (1/100)s\n\t\tSystem: "
           << timer[1] << flush << " (1/100)s\n";

Outputfile << "Dim.: " << r << "\n" << flush;
Fsetprec(mainprec);
Outputfile << "The transformation - matrix:\n" << M << "\n" << flush;

Fmatrix B(r, dim); // The matrix of the resulting basisvectors

spec_mat_mult(A, M, r, B);
Outputfile << "The resulting matrix:\n" << B << "\n" << flush;

M.cols = M.rows = r; // the lattice - dimension
Outputfile << "1-ter Lauf\n" << flush;
start_timing(timer); // Initialization for the measurement of time
rlll(B, M, r, mainprec, elexp, Outputfile);
get_timing(timer);   // Reading the measured time
```

```
   Outputfile << "Zeitmessung:\tUser: " << timer[0] << " (1/100)s\n\t\tSystem: "
              << timer[1] << flush << " (1/100)s\n";
   Outputfile << "Dim.: " << r << "\n" << flush;
   Fsetprec(mainprec);
   Outputfile << "The transformation - matrix:\n" << M << "\n" << flush;
   spec_mat_mult(B, M, r, B);
   Outputfile << "The resulting matrix:\n" << B << "\n" << flush;

   if (erg_test(B, r))
      Outputfile << "\nThe resulting vectors are short enough\n" << flush;
   else
      Outputfile << "\nThe resulting vectors are to long\n" << flush;
   Outputfile.close();
}
```

Literaturverzeichnis

[BaBeCoOl92] C. Batut, D. Bernardi, H. Cohen, M. Olivier: *User's Guide to PARI-GP*, last update May 20, 1992 for version 1.37.

[BaKa] A. Bachem, R. Kannan: *Lattices and the basis reduction algorithm*, Mathematisches Institut Universität Köln, Research Report.

[Bli14] H.F.Blichfeldt: *A new principle in the geometry of numbers, with some applications*, Transactions Amer. Math. Soc., **15** (1914), 227-35.

[BuKe91] J. Buchmann, V. Kessler: *Computing a reduced lattice basis from a generating system*, 1991.

[BuPo89] J. Buchmann, M. Pohst: *Computing a lattice basis from a system of generating vectors*, Proceedings EUROCAL 87, Springer Lecture Notes in Computer Science **378** (1989), 54-63.

[Bund88] P. Bundschuh: *Einführung in die Zahlentheorie*, Springer-Verlag Berlin Heidelberg, 1988.

[Er64] F. Erwe: *Differential- und Integralrechnung I*, Bibliographisches Institut Mannheim, 1964.

[GrLoSchri88] M. Grötschel, L. Lovasz, A. Schrijver: *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag Berlin Heidelberg, 1988

[La78] E. Lamprecht: *Einführung in die Algebra*, Birkhäuser Verlag Basel, 1978.

[La80] E. Lamprecht: *Lineare Algebra 1*, Birkhäuser Verlag Basel, 1980.

[Lang86] S. Lang: *Algebraic Number Theory*, Springer-Verlag New York Inc., 1986.

[LeLeLo82] A.K. Lenstra, H.W. Lenstra Jr. and L. Lovasz: *Factoring polynomials with rational coefficients*, Math. Ann. **261** (1982), 515-534.

[Lo86] L. Lovasz: *An algorithmic theory of numbers, graphs and convexity*, ISBN 0-89871-203-3.

[MiMa65] H. Minc, M. Marcus: *Introduction to Linear Algebra*, Macmillan, New York, 1965.

[NaShn73] I. Nassi, B. Shneidermann: *Flowchart techniques for structured programming*, SIGPLAN Notice 8 (1973), **No.8**, 12-26.

[Pa93] T. Papanikolaou: *LibF: eine lange Gleitpunktarithmetik*, Diplomarbeit an der Universität des Saarlandes, 1993.

[Po87] M. Pohst: *A modification of the LLL Reduction Algorithm*, j. Symbolic Computation, **4** (1987), 123-127.

[PoZa89] M. Pohst, H. Zassenhaus: *Algorithmic Algebraic Number Theory*, Cambridge University Press, 1989.

[Ra86] R. Rannacher: *Praktische Mathematik: Vorlesungsskriptum zum Wintersemester 1986/87*, Fachbereich Mathematik, Universität des Saarlandes.

[SchnoEu91] C.P.Schnorr, M. Euchner: *Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems* To appear in: Proceedings of the FCT'91, Springer Lecture Notes in Computer Science.