
Gitterbasisreduktion in Java

Diplomarbeit

am Fachgebiet Theoretische Informatik - Kryptographie und
Computeralgebra



Prof. Dr. Johannes Buchmann
Technische Universität Darmstadt

vorgelegt von
Nemr Chehimi

Betreuer: Dipl.-Inform. Markus Rückert
Datum: 21. Juli 2009

Danksagung

Mein erster großer besonderer Dank geht an meine Frau Maha Zelzili, meine Familie und meinen Schwager für ihre Motivationshilfen und Unterstützung für die Fertigstellung meiner Arbeit. Zudem bedanke ich mich sehr bei meinem Betreuer Markus Rückert für die gute Betreuung und die wertvolle Diskussionen und Anregungen.

Weiterhin möchte ich mich auch bei Jonas Frey und Michael Kübert für ihr Korrekturlesen und ihre wissenschaftliche Unterstützung bedanken.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Mir ist nicht bekannt, dass die Arbeit bisher in gleicher oder ähnlicher Form in einer anderen Prüfungsbehörde vorgelegt bzw. veröffentlicht wurde.

Darmstadt den 21. Juli 2009

Nemr Chehimi

Inhaltsverzeichnis

1	Einführung	1
2	Grundlagen	5
2.1	Fakten aus der linearen Algebra	5
2.1.1	Notationen	5
2.1.2	Definitionen	6
2.2	Zahlendarstellung im Rechner	8
2.2.1	Gleitkommazahlen	10
2.2.2	Eindeutigkeit der Gleitkomma-Darstellung	12
2.2.3	Rundung reeller Zahlen in Maschinen-Darstellung	12
3	Einführung in die Gittertheorie	15
3.1	Sprache der Gitter	15
3.2	Unimodulare Transformation	19
3.3	Gitterprobleme	21
4	Orthogonalisierungsverfahren	23
4.1	Gram-Schmidtsches Orthogonalisierungsverfahren	25
4.2	Householder Reflections	27
4.3	Givens-Rotations	31
5	Gitterbasisreduktion	35
5.1	Größenreduktion	36
5.2	LLL-Reduktion	38
5.3	LLL-Reduktion in Gleitkommaarithmetik	40
5.3.1	Der Schnorr-Euchner Algorithmus	40
5.3.2	Der Schnorr-Koy Algorithmus	42
5.3.3	Andere Verfahren zur Gitterbasisreduktion	46

6	Implementierung und Diskussion	49
6.1	Die benutzten Datentypen	50
6.1.1	Primitive ganzzahlige Datentypen	50
6.1.2	Datentypen für große Ganzzahlen	51
6.1.3	Primitive Gleitkomma-Datentypen	53
6.1.4	Datentypen für große Gleitkommazahlen	54
6.2	Benutzte Bibliotheken	55
6.2.1	Benchmark Beschreibung	56
6.2.1.1	Matrix-Multiplikation	57
6.2.1.2	Determinantenberechnung	58
6.2.1.3	Die Matrixinvertierung	59
6.2.1.4	Die QR-Zerlegung	59
6.2.1.5	Die Cholesky-Zerlegung	60
6.2.2	Resultate	61
6.3	Matrizendarstellung	61
6.4	Implementierung der Gitterbasisreduktionsalgorithmen	62
6.4.1	Beschreibung der Methoden	62
7	Experimentelle Resultate	65
8	Zusammenfassung	69
	Abbildungsverzeichnis	71
	Tabellenverzeichnis	73
	Algorithmenverzeichnis	77
	Literaturverzeichnis	79
A	Grundlagen der Komplexitätstheorie	85
B	Benchmark-Tabellen für Langganzzahlarithmetik	89
C	Benchmark-Tabellen für Arithmetik auf großen Gleitkommazahlen	93
D	Klassendiagramm des Programms	97

1 Einführung

Gitter sind diskrete, additive Untergruppen eines reellen Vektorraums \mathbb{R}^n . Mit der Gitterbasisreduktion wird das Ziel verfolgt, möglichst kurze Vektoren für die Basis eines Gitters zu finden. Dabei stellt dies ein viel diskutiertes Problem dar, welches bereits von **Hermite**, **Korkine-Zolotareff** und **Gauß** im Zusammenhang mit der Sprache der quadratischen Formen behandelt wurde.

Nachdem die Gitterbasisreduktion für eine gewisse Zeit in den Hintergrund rückte, waren Lenstra, Lenstra und Lovász [LLL82] diejenigen, die den polynomiellen **LLL**-Algorithmus darstellten und somit die Anwendung der Gitterbasisreduktion in der ganzzahligen Optimierung, der Kodierungstheorie und der Kryptographie, begründeten. Ihre Bedeutung für die Kryptographie ergibt sich u.a. durch ihre Anwendung in zahlreichen Angriffen auf kryptographische Verfahren. Beispiele dafür sind die Kryptosysteme, die auf dem Rucksackproblem und Polynomfaktorisierungsproblem in Restklassenringen von Polynomen beruhen (Merkle-Hellman, [ChRi], **NTRU** [HoPiSi], ect.), von denen die meisten inzwischen als gebrochen gelten ([Adleman], [Odlyzko]).

Typische Gitterprobleme sind **SVP** (Shortest Vector Problem), **CVP** (Closest Vector Problem) und **SBP** (Smallest Basis Problem). Bei einem gegebenen Gitter L mit der Basis $[b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ wird beim **SVP** nach dem nicht-trivialen kürzesten Gittervektor gesucht, wohingegen beim **CVP** zu einem gegebenen Punkt ein nächster Gittervektor zu bestimmen ist. Die Untersuchungen dieser Probleme, von denen man zumindest ihre \mathcal{NP} -Schwierigkeit beweisen kann, haben in den vergangenen 25 Jahren eine Reihe von interessanten Ergebnissen hervorgebracht.

Auf Grund der Schwierigkeit diese Probleme exakt zu lösen, wurden Approximationsvarianten α -**SVP** (approximiertes Shortest Vector Problem) und α -**CVP** (approximiertes Closest Vector Problem) betrachtet. Beim α -**SVP** wird nach einem Vektor b_1 mit $\|b_1\| \leq \alpha \lambda_1$ gesucht, wobei $\|\cdot\|$ die euklidische Norm

und λ_1 der kürzeste von Null verschiedene Gittervektor ist, während beim α -CVP zu einem gegebenen Gittervektor b ein Gittervektor b_1 zu bestimmen ist, für den gilt $\|b - b_1\| \leq \alpha \min\{\|b - b_2\| \mid b_2 \in L\}$. Lenstra, Lenstra und Lovász [LLL82] präsentierten den ersten Algorithmus namens **LLL**-Algorithmus in polynomieller Laufzeit. Dieser berechnet zu gegebener Basismatrix eine **LLL**-reduzierte Basismatrix, für deren kürzesten Vektor b_1 gilt $\|b_1\| \leq 2^{\frac{n-1}{2}} \lambda_1$.

Ziel der vorgelegten Arbeit ist es zunächst die Gittertheorie in umfassender Form darzustellen. Sodann erfolgt die Implementierung der Gitterbasisreduktionsalgorithmen, welche in der Programmiersprache Java [JAVA] durchgeführt wurde. Dabei handelt es sich insbesondere um die Implementierung und experimentelle Evaluierung des von **Schnorr** und **Euchner** [SE93] im Jahr **1993** vorgestellten Algorithmus, der auf der Basis der Arbeiten von A.K. Lenstra, H.W. Lenstra Jr. und L.Lovász [LLL82] beruht sowie des von **Schnorr** und **Koy** [KoySch] entwickelten Algorithmus. Anschließend wurden die Messzeiten der implementierten Algorithmen mit dem von **Victor Shoup** in [NTL] implementierten **Schnorr** und **Euchner** Algorithmus verglichen.

Inhaltliche Gliederung der Diplomarbeit:

Nach einer kompakten Vorstellung einiger mathematischen Hintergründe für Gitter (Kapitel 2 auf Seite 5), wird in Kapitel 3 auf Seite 15 eine Einführung in die Gittertheorie in kurzer Form dargestellt (Definitionen, Sukzessives Minimum, Unimodulare Transformation, Hermitesche Normalform sowie die Gitterprobleme). Ferner beschäftigen wir uns in diesem Kapitel mit der Darstellung von Gleitkommazahlen in Computern und den (numerischen) Eigenschaften der Gleitkommazahlen-Arithmetik, um die Merkmale von Gleitkomma-Datentypen und ihre Auswirkungen in der Praxis zu verstehen.

In Kapitel 4 auf Seite 23 werden einige Verfahren für die Orthogonalisierung bzw. für die **QR**-Zerlegung der Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eines Gitters L , die bei der Gitterbasisreduktion benutzt werden, vorgestellt. Behandelt wird also in diesem Kapitel das Gram-Schmidt'sche Orthogonalisierungsverfahren, das hauptsächlich für den **LLL**-Algorithmus verwendet wird. Dieses Verfahren gilt als ein numerisch instabiles Verfahren. Aus diesem Grund werden zusätzliche numerisch stabilere Verfahren für die **QR**-Zerlegung wie

Householder-Reflexionen und Givens-Rotationen eingeführt.

Das Kapitel 5 auf Seite 5 widmet sich den Verfahren zur Gitterreduktion. Der **LLL**-Reduktionsbegriff für geordnete Gitterbasis B beliebigen Ranges d wurde 1982 von A. K. Lenstra, H. W. Lenstra und L. Lovász in [LLL82] eingeführt. Es ist das erste Reduktionsverfahren für ganzzahlige Gitterbasis, das die sukzessiven Minima eines Gitters approximiert und in Polynomialzeit eine reduzierte Gitterbasis erzeugt. Zudem wird die Variante des **LLL**-Algorithmus, die von **Schnorr** und **Euchner** [SE93] vorgestellt wurde, in diesem Kapitel dargestellt. Diese Verfahren arbeitet mit der schnelleren Gleitkomma-Arithmetik und basiert auf das Gram-Schmidt'sche Orthogonalisierungsverfahren. Ferner beschäftigen wir uns in diesem Kapitel mit dem Verfahren von **Schnorr** und **Koy** [KoySch], welches die Orthogonalisierung nach Gram-Schmidt durch Householder-Reflexionen ersetzt.

Das Kapitel 6 befasst sich mit der Implementierung unseres Programms. Für die Implementierung wurden Benchmarks erstellt um die beste Bibliothek auszuwählen, die für den weitem Gebrauch vorzuziehen ist.

In Kapitel 7 werden dann die vorgestellten Algorithmen von **Schnorr-Euchner** 5.3.2 und **Schnorr-Koy** 5.3.3 nach ihrer Laufzeit analysiert. Sodann folgt ein Performance-Vergleich dieser Algorithmen zu der C++ Bibliothek **NTL** [NTL].

Im letzten Kapitel folgt abschließend die Zusammenfassung dieser Arbeit.

Im Anhang A erfolgt eine kurze Einführung in die Grundlagen der Komplexitätstheorie, wobei unter anderem die Definition der \mathcal{NP} -Vollständigkeit angeführt wurde. Anhang B befasst sich mit Benchmarktabellen für Langganzzahlarithmetik. Weiterhin beinhaltet Anhang C Benchmarktabellen für Arithmetik auf grossen Gleitkommzahlen und schließlich ist im Anhang D das **UML**-Klassendiagramm zur graphischen Darstellung der von uns implementierten Klassen, Schnittstellen, sowie deren Beziehungen, zu finden.

2 Grundlagen

2.1 Fakten aus der linearen Algebra

In diesem Kapitel werden einige mathematische Hintergründe für Gitter vorgestellt. Es beinhaltet Notation, Definitionen, Sätze und hilfreiche Eigenschaften von Gittern, die die Basis der nächsten Kapitel bilden.

2.1.1 Notationen

Es wird mit

$$M^n := M \times \dots \times M \tag{2.1}$$

das n -fache kartesische Produkt der Menge M und mit, E_d die $d \times d$ Einheitsmatrix und mit $M^{n \times d}$ die Menge aller $n \times d$ -Matrizen mit Einträgen aus der Menge M bezeichnet.

Wird $M = \mathbb{Z}$ oder $M = \mathbb{R}$ gesetzt, so wird von der Menge aller ganzzahligen bzw. reellen $n \times d$ Matrizen gesprochen.

Zudem wird mit B^T als die zu B transponierte Matrix und mit B^{-1} die zu B inverse Matrix bezeichnet.

Weiterhin bezeichnen wir mit $\lceil x \rceil$ die nächste ganze Zahl, $\lfloor x \rfloor$ die kleinste ganze Zahl größer oder gleich x und $\lceil x \rceil$ die größte ganze Zahl kleiner oder gleich x zu einer reellen Zahl $x \in \mathbb{R}$. Sei weiter $|x|$ der Betrag von x und $\text{sign}(a) = \pm 1$ ($\text{sign}(0) = 1$) das Vorzeichen von x .

2.1.2 Definitionen

Definition 2.1.1 (Norm)

Sei V ein **reeller Vektorraum** (über dem Körper \mathbb{R}). V darf endlich oder unendlich viele Dimensionen haben. Ein Längenmaß für Vektoren heißt **Norm**, wenn gilt

$$\underbrace{\|\cdot\|}_{\text{Norm}}: V \longrightarrow \mathbb{R}, \quad u \mapsto \|u\|$$

- a) $\|u\| \geq 0$ *positive Definitheit*
 $\|u\| = 0 \Leftrightarrow u = 0$ *für alle $u \in V$*
- b) $\|\nu u\| = |\nu| \cdot \|u\|$ *für alle $u \in V, \nu \in \mathbb{R}$* *positive Homogenität*
- c) $\|u + v\| \leq \|u\| + \|v\|$ *für alle $u, v \in V$* *Dreiecksungleichung*

Das Paar $(\mathbb{R}, \|\cdot\|)$ heißt dann normierter Raum.

Durch das Standard-Skalarprodukt wird der \mathbb{R}^n zu einem euklidischen Vektorraum.

Definition 2.1.2 (Das euklidische Skalarprodukt)

Sei $u, v \in \mathbb{R}^n$ dann wird durch $\langle \cdot, \cdot \rangle: \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$

$$\langle u, v \rangle = \|u\| \|v\| \cos \angle(u, v) = \sum_{j=1}^n u_j v_j = u_1 v_1 + \cdots + u_n v_n \quad (2.2)$$

wobei $\angle(u, v)$ der Winkel zwischen den beiden Vektoren u und v , das übliche oder Standard-Skalarprodukt auf \mathbb{R}^n definiert.

Aus der Definition ergibt sich folgendes :

$$\cos \angle(u, v) = \frac{\langle u, v \rangle}{\|u\| \|v\|} \quad (2.3)$$

Für $\langle u, v \rangle = 0$ ergibt sich ein rechter Winkel, für $\langle u, v \rangle > 0$ gilt $\angle(u, v) < \pi/2$ und für $\langle u, v \rangle < 0$ gilt $\angle(u, v) > \pi/2$.

Es gelten folgende Rechenregeln:

- (i) $\langle u, u \rangle \geq 0$
 $\langle u, u \rangle = 0 \Leftrightarrow u = 0$ für alle $u \in \mathbb{R}^n$
- (ii) $\langle u, v + w \rangle = \langle u, v \rangle + \langle u, w \rangle$
 $\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$ für alle $u, v, w \in \mathbb{R}^n$
(bilinear)
- (iii) $\langle ru, v \rangle = \nu \langle u, v \rangle = \langle u, \nu v \rangle$ für alle $u, v \in \mathbb{R}^n, \nu \in \mathbb{R}$
- (iv) $\langle u, v \rangle = \langle v, u \rangle$ für alle $u, v \in \mathbb{R}^n$.

Definition 2.1.3 (Die allgemeine p -Norm)

Die p -Norm ist für $u = [u_1, u_2, \dots, u_n]^T \in \mathbb{R}^n$ und $1 < p < \infty$ definiert als:

$$\|u\|_p = \left(\sum_{j=1}^n |u_j|^p \right)^{\frac{1}{p}}. \quad (2.4)$$

Ein Spezialfall ist die euklidische- oder 2-Norm gegeben durch

$$\|u\|_2 = \sqrt{\langle u, u \rangle} = \left(\sum_{j=1}^n u_j^2 \right)^{\frac{1}{2}} = \sqrt{u_1^2 + u_2^2 + \dots + u_n^2}. \quad (2.5)$$

Die 1-Norm, auch **Betragsnorm** genannt, ist für einen Vektor $u = [u_1, u_2, \dots, u_n]^T \in \mathbb{R}^n$ definiert als

$$\|u\|_1 = \|[u_1, u_2, \dots, u_n]^T\| = \sum_{j=1}^n |u_j| = |u_1| + |u_2| + \dots + |u_n|. \quad (2.6)$$

Für ($p = \infty$) wird mit

$$\|u\|_\infty = \max\{|u_1|, |u_2|, \dots, |u_n|\} \quad (2.7)$$

das übliche Maximum- oder **Supremumsnorm** definiert.

Satz 2.1.1 (Die Cauchy-Schwarz-Ungleichung)

Für die Beziehung zwischen dem Skalarprodukt und der zugehörigen Norm $\|u\|_2$ gilt:

$$|\langle u, v \rangle| \leq \|u\|_2 \cdot \|v\|_2 \quad \text{für alle } u, v \in \mathbb{R}^n. \quad (2.8)$$

Bemerkung 1

Es gilt für alle $u, v \in \mathbb{R}^n$:

$$\|u + v\|_2^2 = \|u\|_2^2 + \|v\|_2^2 + 2\langle u, v \rangle. \quad (2.9)$$

Verhältnis der Normen zueinander: Für die drei Normen und für alle $u, v \in \mathbb{R}^n$ gilt:

$$\|u\|_1 \geq \|u\|_2 \geq \|u\|_\infty \geq \frac{1}{n} \|u\|_1 \quad (2.10)$$

$$\|u\|_2 \leq \|u\|_1 \leq \sqrt{n} \|u\|_2 \quad (2.11)$$

$$\|u\|_\infty \leq \|u\|_2 \leq n \cdot \|u\|_\infty. \quad (2.12)$$

Definition 2.1.4 (lineare Hülle)

Sei die Menge

$$L = \left\{ v_i \mid 1 \leq i \leq d, v_i \in \mathbb{R}^n \right\}.$$

Dann heißt die Menge

$$\text{span}(L) = \text{span}(v_1, v_2, \dots, v_d) = \left\{ \sum_{i=1}^d \nu_i v_i \mid \nu_i \in \mathbb{R} \right\}$$

lineare Hülle der Menge L .

Die lineare Hülle ist der kleinste Untervektorraum des \mathbb{R}^n , der die Menge L enthält.

Definition 2.1.5 (Isometrie, orthogonale Matrix)

- Eine lineare Abbildung $f : V \rightarrow W$ mit linearen Räumen V, W heißt *isometrisch* wenn f das Skalarprodukt erhält, d.h. wenn $\langle u_1, u_2 \rangle = \langle f(u_1), f(u_2) \rangle$ für alle $u_1, u_2 \in V$.
- Eine Matrix $B \in \mathbb{R}^{n \times d}$, $n \geq d$ heißt *isometrisch* wenn die Abbildung $u \rightarrow Bu$ isometrisch ist, d.h. wenn $B^T B = E_n$. Isometrische Quadratmatrix $B \in \mathbb{R}^{n \times n}$ bezeichnet man als *orthogonal*. B ist orthogonal genau dann, wenn $B^{-1} = B^T$.

2.2 Zahlendarstellung im Rechner

Soll eine mathematische Software geschrieben oder verwendet werden, so ist es notwendig, sich mit den Eigenheiten der Zahlendarstellung am Computer

vertraut zu machen. Entgegen der üblichen Dezimaldarstellung arbeiten Computer im allgemeinen in der Binärdarstellung. So ist die Zweierkomplement-Darstellung die häufigste Darstellungsform für ganze Zahlen und ist folgendermaßen definiert

Definition 2.2.1

Sei $x = a_{n-1}, \dots, a_0$ eine n -stellige Dualzahl. Dann heißt

$$\mathbf{K}_2(x) = (1 - a_{n-1}, \dots, 1 - a_0) + 1 \pmod{2^n} \quad (2.13)$$

das **2-Komplement** von x .

Bei der Darstellung von ganzen bzw. reellen Zahlen erfolgt eine Beschränkung auf Dualzahlen aus der Menge $\sum_2 = \{0, 1\}$. Dabei wird z.B. für eine ganze Zahl in einer Zweierkomplement-Darstellung mit n Bits das höchstwertige Bit a_{n-1} als Vorzeichen verwendet. Ist $a_{n-1} = 0$, so die restlichen Bits für den Betrag der Zahl fest, ansonsten werden die restlichen Bits negiert und schließlich mit einer 1 addiert. Bei dieser Darstellung lassen sich 2^n verschiedene Zahlen repräsentieren und die Werte von -2^{n-1} bis $2^{n-1} - 1$ in Abständen von 1 darstellen¹.

Es sind Zahlen mit einem fiktiven Komma vorzustellen, welches auf der äußersten rechten Seite steht. Sollen arithmetische Operationen durchgeführt werden, so kann der zulässige Wertebereich eventuell überschritten werden. Wird die obere Grenze überschritten, so ist von einem Überlauf (Overflow bzw. Exponentenüberlauf) die Rede, wird die untere Grenze unterschritten, so ist von einem Unterlauf (Underflow bzw. Exponentenunterlauf) zu sprechen.

Wird das Komma um m Stellen nach links verschoben, so bedeutet dies eine Division durch 2^m . Die Abstände zwischen den Zahlen betragen nun 2^{-m} , wodurch sich näherungsweise rationale Zahlen darstellen lassen. Wird das Komma um m Stellen nach rechts verschoben, dann lassen sich noch sehr große Zahlen darstellen, allerdings nur mit dem Abstand 2^m .

¹Viele Computersprachen beschränken sich sogar auf einen bestimmten Bereich, etwa -2^{31} bis $2^{31} - 1$ (kann durch 32 Bit dargestellt werden) oder -2^{63} bis $2^{63} - 1$ (kann durch 64 Bit dargestellt werden). Die Darstellung geschieht oft vorzeichenlos

Definition 2.2.2 (Festkomma-Darstellung)

Bei der Festkomma-Darstellung wird bei vorgegebener Stellenzahl N eine feste Nachkommastelle vereinbart, das heißt

$$x = (-1)^s \cdot (a_{N-2}a_{N-3} \dots a_k.a_{k-1} \dots a_0)_\beta = (-1)^s \cdot \beta^{-k} \sum_{j=0}^{N-2} a_j b^j \quad (2.14)$$

mit k Nachkomma- und $(N-k-1)$ Vorkommastellen, dabei wird mit $s \in \{0, 1\}$ für das Vorzeichen und $\beta \in \{2, 3, 4, \dots\}$ für die Basis.

Der erhebliche Nachteil der Festkomma-Darstellung besteht darin, dass der darstellbare Bereich stark beschränkt ist, insbesondere ist die Genauigkeit zwischen zwei benachbarten Zahlen immer gleich, was für numerische Rechnungen inakzeptabel ist. Dieser Nachteil kann behoben werden, wenn die Skalierung in Gleichung (2.14) variiert wird.

Es ist sodann von einer Festkommazahl die Rede, wenn das Komma an einer festen Stelle vorzustellen ist. Werden zusätzliche Bits verwendet, die die Stellung des Kommas angeben, dann wird von einer Gleitkommazahl (floating point number) oder einer halblogarithmischen Darstellung gesprochen.

2.2.1 Gleitkommazahlen

Da sowohl große Zahlen wie etwa πe^{50} , periodische Zahlen als auch kleine Zahlen wie etwa $\frac{e^{-30}}{60!}$ mit derselben Güte approximiert werden, verwenden numerische Computersprachen bzw. Mikroprozessoren in der Regel die so genannte Gleitkomma-Darstellung.

Im Folgenden wird beschrieben wie die interne Repräsentation von reellen Zahlen in den meisten heutigen Rechnern abläuft. Daneben soll diskutiert werden wie sich die endliche Zahlendarstellung auf die Rechengenauigkeit auswirkt. Eine ausführlichere Besprechung dieser Thematik, sowie die meisten hier weggelassenen Beweise, sind [OeWa] zu finden.

Definition 2.2.3 (Gleitkomma-Darstellung)

Gleitkommazahlen von $x \in \mathbb{R} \setminus \{0\}$ zur Basis $\beta \in \{2, 3, 4, \dots\}$ haben die Form

$$x = (-1)^s (0.a_1 a_2 \dots a_t)_\beta \cdot \beta^e = (-1)^s m \cdot \beta^{e-t}, \quad (2.15)$$

wobei $t \in \mathbb{N}$ die Zahl der erlaubten signifikanten Stellen a_j mit $0 \leq a_j \leq \beta - 1$, $m = a_1 a_2 \dots a_t$ eine ganze Zahl, Mantisse genannt, mit $0 \leq m \leq \beta^t - 1$ und e eine ganze Zahl, Exponent genannt, sind. Der Exponent kann innerhalb des endlichen Intervalls von zulässigen Werten variieren: Man erlaubt $e_{min} \leq e \leq e_{max}$, $e_{min} < 0$, $e_{max} > 0$ für ganzzahlige Zahlen e_{min} und e_{max} .

Die N Speicherstellen sind nun eingeteilt in das Vorzeichen (eine Stelle), die signifikanten Stellen (t Stellen) und die verbleibenden $N - t - 1$ Stellen für den Exponenten. Typischerweise sind in Java zwei Standardformate für die Gleitkomma-Darstellung verfügbar: einfach für **float** und doppelt genau für **double**. Im Fall der binären Darstellung entsprechen diese Formate in der Standardversion der Darstellung mit $N = 32$ bits (**float**).

1 bit	8 bits	23 bits
s	e	m

und mit $N = 64$ bits (**double**)

1 bit	11 bits	52 bits
s	e	m

Tabelle 6.5 auf Seite 54 zeigt die Wertebereiche der Gleitkomma-Datentypen. Bei der rechnerinternen Darstellung muss die Anzahl der Bits für die Mantisse bzw. den Exponenten zur Verfügung festgelegt werden. Dies resultiert aus der Menge der darstellbaren Maschinenzahlen, die durch

$$\mathbb{F} = \mathbb{F}(\beta, t, e_{min}, e_{max})$$

$$= \{0\} \cup \left\{ x \in \mathbb{R} \mid x = (-1)^s \beta^e \sum_{i=1}^t a_i \beta^{-i}, (e_{min} \leq e \leq e_{max}) \right\}$$

die Menge der Gleitkommazahlen mit t signifikanten Stellen, der Basis $\beta \geq 2$, $0 \leq a_i \leq \beta - 1$, und dem Bereich (e_{min}, e_{max}) mit $(e_{min} \leq e \leq e_{max})$ bezeichnet wird.

2.2.2 Eindeutigkeit der Gleitkomma-Darstellung

Um die Eindeutigkeit der Zahlendarstellung zu garantieren, wird typischerweise angenommen, dass $a_1 \neq 0$ und somit $m \geq \beta^{t-1}$ gilt. In solch einem Fall heißt a_1 die führende signifikante Stelle, während a_t die letzte signifikante Stelle ist und die Darstellung von x heißt normalisiert. Die Mantisse m variiert nun zwischen β^{t-1} und $\beta^t - 1$. Ohne die Annahme, dass $a_1 \neq 0$ ist, die Zahl 0 ihr eigenes Vorzeichen hat (Standard $s = 0$), würde jede Zahl verschiedene Darstellungen haben, wodurch die Eindeutigkeit nicht mehr garantiert werden kann.

Es kann unmittelbar festgestellt werden, dass die aus $x \in \mathbb{F}(\beta, t, e_{min}, e_{max})$ auch $-x \in \mathbb{F}(\beta, t, e_{min}, e_{max})$ folgt. Des Weiteren gelten untere und obere Schranke für den Absolutbetrag von x

$$x_{min} = \beta^{e_{min}-1} \leq |x| \leq \beta^{e_{max}}(1 - \beta^{-t}) = x_{max}.$$

Beispiel 2.2.1

Die positiven Zahlen in der Menge $\mathbb{F}(2, 2, -1, 1)$ sind

$$\begin{array}{lll} (0.11)_2 \cdot 2^1 = \frac{3}{2} & (0.11)_2 \cdot 2^0 = \frac{3}{4} & (0.11)_2 \cdot 2^{-1} = \frac{3}{8} \\ (0.10)_2 \cdot 2^1 = 1 & (0.10)_2 \cdot 2^0 = \frac{1}{2} & (0.10)_2 \cdot 2^{-1} = \frac{1}{4}. \end{array}$$

Sie sind zwischen $x_{min} = \beta^{e_{min}-1} = 2^{-2} = \frac{1}{4}$ und $x_{max} = \beta^{e_{max}}(1 - \beta^{-t}) = 2^1(1 - 2^{-2}) = \frac{3}{2}$.

2.2.3 Rundung reeller Zahlen in Maschinen-Darstellung

Die Tatsache, dass auf jedem Computer nur eine Teilmenge \mathbb{F} von \mathbb{R} tatsächlich verfügbar ist, verursacht verschiedene praktische Probleme, vor allem die Darstellbarkeit jeder gegebenen reellen Zahl in \mathbb{F} . Eine wichtige Beobachtung ist, dass das Ergebnis einer Operation auf zwei Zahlen, die in \mathbb{F} liegen, nicht zwingend in \mathbb{F} liegt. Darüber hinaus entsteht ein Unterlauf bzw. ein Überlauf falls das Ergebnis in der Menge $(-\infty, -x_{max}) \cup (x_{max}, \infty)$ bzw. in der Menge $(-x_{min}, x_{min})$ liegt. Deshalb muss eine Arithmetik auf \mathbb{F} definiert werden. Die Lösung dieses Problems wäre in der Rundung dieser Zahl zu finden und zwar in der Weise, dass die gerundete Zahl zu \mathbb{F} gehört.

Definition 2.2.4 (Rundung)

Die Rundung ist eine Abbildung $\mathbf{fl} : \mathbb{R} \rightarrow \mathbb{F}$ mit den Eigenschaften

$$\mathbf{fl}(x) = (-1)^s (0.a_1 a_2 \dots \tilde{a}_t) \cdot \beta^e, \quad (2.16)$$

$$\tilde{a}_t = \begin{cases} a_t, & \text{wenn } a_{t+1} < \frac{\beta}{2} \text{ (Abschneiden)} \\ a_t + 1, & \text{wenn } a_{t+1} \geq \frac{\beta}{2} \text{ (Aufrunden)}. \end{cases} \quad (2.17)$$

Abgesehen von Ausnahmesituationen kann der Fehler, absolut und relativ, leicht quantifiziert werden. Dies erfolgt durch Substitution von $\mathbf{fl}(x)$ für x . Es kann somit das folgende Resultat gezeigt werden.

Satz 2.2.1

Ist $x \in \mathbb{R}$ derart, dass $x_{\min} \leq |x| \leq x_{\max}$, so gilt

$$\mathbf{fl}(x) = x(1 + \delta) \text{ mit } |\delta| \leq u, \quad (2.18)$$

wobei

$$u = \frac{1}{2}\beta^{1-t} = \frac{1}{2}\epsilon_M \quad (2.19)$$

die so genannte Maschinengenauigkeit ist ².

Somit gilt aus der Gleichung 2.18 die folgende Schranke für den relativen Fehler

$$E_{\text{relativ}}(x) = \frac{|x - \mathbf{fl}(x)|}{|x|} \leq u, \quad (2.20)$$

wobei es für den absoluten Fehler aus den Ungleichungen 2.20 und 2.16

$$E_{\text{absolut}}(X) \leq \frac{1}{2}\beta^{e-t} \quad (2.21)$$

gilt.

² In IEEE Standard 754-1985 for Binary Floating-point Arithmetic ist die Maschinengenauigkeit für den float precision $\epsilon_M = 2^{-23} = 1.1920929 \cdot 10^{-7}$ double Precision $\epsilon_M = 2^{-52} = 2.22044605 \cdot 10^{-16}$

3 Einführung in die Gittertheorie

3.1 Sprache der Gitter

Gitter als Punktmenge des Vektorraums \mathbb{R}^n sind Gegenstand der Geometrie der Zahlen, die Minkowski um 1900 entwickelt hat. Die ganzzahligen Lösungen eines Gleichungssystems bilden ein typisches Gitter. Gitter bilden in diesem Sinne ein diskretes Analogon zu den linearen Räumen. In älteren Arbeiten wird oft noch die Sprache der quadratischen Formen anstelle der Linearen Algebra verwendet, um Gitter zu beschreiben.

Definition 3.1.1 (Gitter)

Eine Teilmenge im \mathbb{R}^n heißt Gitter, falls es linear unabhängige Vektoren $b_1, b_2, \dots, b_d \in \mathbb{R}^n$ gibt mit:

$$L = L(b_1, b_2, \dots, b_d) = \sum_{j=1}^d b_j \mathbb{Z}$$

die von den Vektoren erzeugte additive Untergruppe des \mathbb{R}^n . Sind die Vektoren b_1, b_2, \dots, b_d linear unabhängig in \mathbb{R}^n , dann wird mit $L(b_1, b_2, \dots, b_d)$ ein Gitter der Dimension (vom Rang) $\dim(L) := d$ genannt.

Bemerkung 2

Ein Gitter heißt volldimensional, wenn es eine additive Untergruppe in \mathbb{R}^n , und vom Rang n ist.

Die Darstellung eines Gitters L erfolgt durch Angabe seiner Basisvektoren b_1, b_2, \dots, b_d die als Matrix geschrieben werden, wobei die einzelnen Vektoren

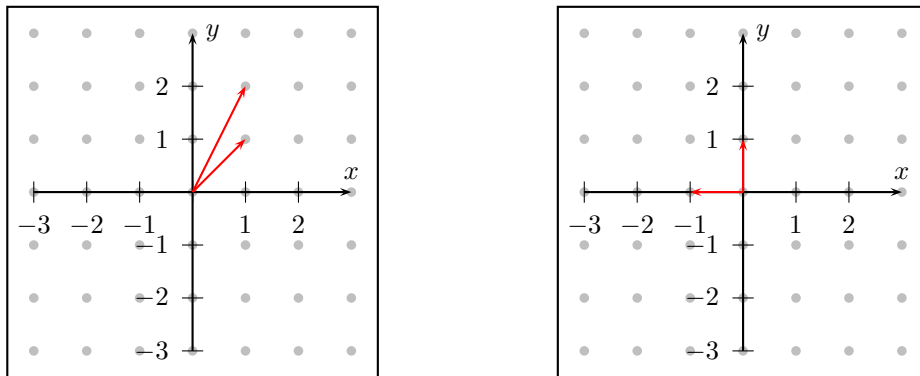


Abbildung 3.1: Beispiel für ein 2-dimensionales Gitter mit den verschiedenen Basen B_1 und B_2 aus Beispiel 3.1.1

b_i Spaltenvektoren der Matrix (Basismatrix)

$$B = [b_1, b_2, \dots, b_{d-1}, b_d] \in \mathbb{R}^{n \times d}$$

sind.

Beispiel 3.1.1

Zwei verschiedene Basen für dasselbe 2-dimensionale Gitter wären

$$B_1 = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \quad \text{und} \quad B_2 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Definition 3.1.2 (Das orthogonale Gitter)

Das zu einem Vektor $u = [u_1, u_2, \dots, u_n]^T \in \mathbb{Z}^n \setminus \{0\}$ orthogonale Gitter ist definiert als

$$L_u = \{b \in \mathbb{Z}^n \mid \langle u, b \rangle = 0\}.$$

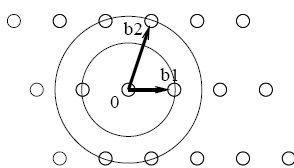
Definition 3.1.3 (Sukzessives Minimum)

Sei das Gitter $L \in \mathbb{R}^n$ und $j \in \{1, 2, \dots, d\}$. Die reelle Zahl

$$\lambda_j = \lambda_j(L) = \min \left\{ \max_{1 \leq i \leq j} \|u_i\| \mid u_1, u_2, \dots, u_j \in L \text{ sind linear unabhängig} \right\} \quad (3.1)$$

heißt das j -te **sukzessive Minimum** des Gitters L bzgl. der euklidischen Norm.

Der Wert λ_j kann als der kleinste Radius r einer Kugel um den Nullpunkt aufgefasst werden, die j linear unabhängige Gittervektoren aus L enthält.

Abbildung 3.2: Sukzessive Minima $\lambda_1 = \|b_1\|$, $\lambda_2 = \|b_2\|$ **Satz 3.1.1**

Die sukzessiven Minima besitzen folgende Eigenschaften:

- Es ist $\lambda_1 \leq \lambda_2, \dots \leq \lambda_d$.
- Für linear unabhängige Gittervektoren $b_1, b_2, \dots, b_i \in L$ mit $\|b_1\| \leq \|b_2\| \leq \dots \leq \|b_i\|$ gilt $\lambda_i \leq \|b_i\|$.
- Es existieren linear unabhängige Vektoren $b_1, b_2, \dots, b_d \in L$ mit $\lambda_i = \|b_i\|$ für $1 \leq i \leq d$.

Definition 3.1.4 (Gram-Matrix)

Die Matrix $B^T B = [\langle b_i, b_j \rangle]_{1 \leq i, j \leq d} \in \mathbb{R}^{d \times d}$ bestehend aus den Skalarprodukten der Basisvektoren wird als die Gram-Matrix zur Basis B bezeichnet.

Definition 3.1.5 (Gitterdeterminante)

Die Determinante eines Gitters $L \subseteq \mathbb{R}^n$ ist definiert als

$$\det(L) = \sqrt{B^T B} \quad (3.2)$$

für eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ des Gitters L .

Die Gitterdeterminante $\det(L)$ des Gitters $L = L(B) \subset \mathbb{R}^n$ ist das Volumen des Parallelepipeds $\mathcal{P}(B)$, das von den Vektoren $b_1, b_2, \dots, b_{d-1}, b_d$ aufgespannt wird

$$\det(L) = \text{Vol}(\mathcal{P}(B)) = \text{Vol} \left(\left\{ Bu \mid u \in [0, 1]^n \right\} \right) \quad (3.3)$$

Definition 3.1.6 (Orthogonalisierungsdefekt)

Sei $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ die Basismatrix des Gitters L . Der Orthogonalisierungsdefekt von L ist definiert als

$$\text{odef}(L) = \frac{\prod_{j=1}^d \|b_j\|}{\det(L)}. \quad (3.4)$$

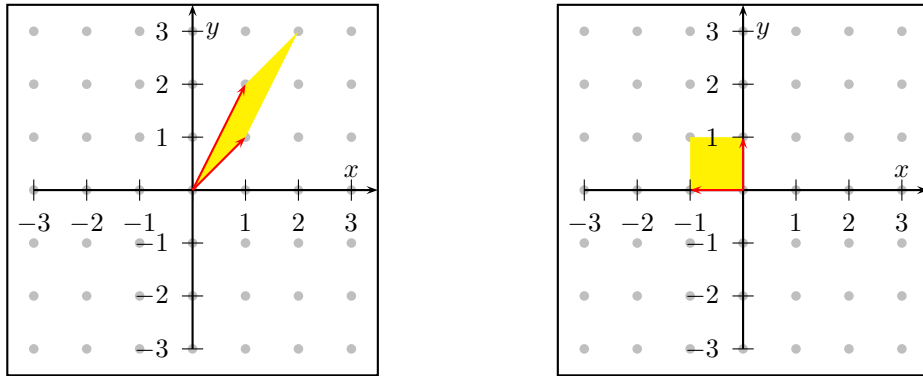


Abbildung 3.3: Beispiel für die Parallelepede beider Gitterbasen aus Beispiel 3.1.1

Im Kapitel 5 wird die Aufgabe der Gitterreduktion auch alternativ als Minimierung des Orthogonalisierungsdefekts beschrieben.

Als nächstes erfolgt die Einführung der Hermite-Konstante. Das Ziel vieler Reduktionsalgorithmen liegt darin, diesem Wert durch die Approximation des kürzesten Gittervektors nahezukommen.

Definition 3.1.7 (Hermite-Konstante)

Die Hermite-Konstante γ_d ist definiert als

$$\gamma_d := \max \left\{ \frac{\lambda_1(L)^2}{(\det(L))^{\frac{2}{d}}} \mid L \subset \mathbb{R}^n \text{ vom Rang } d \right\}. \quad (3.5)$$

Ist die entsprechende Hermite-Konstante γ_d bekannt, so kann mit

$$\lambda_1(L)^2 \leq \gamma_d (\det L)^{2/d} \quad (3.6)$$

eine obere Schranke für die Länge des kürzesten Gittervektors festgelegt werden.

Für große d ist eine auf Gauß zurückgehenden Volumen-Heuristik ausreichend, die asymptotische Schranken für γ_d liefert. Nach dieser gilt

$$\frac{d}{2\pi e} + \mathbf{o}(d) < \gamma_d < \frac{d}{\pi e} + \mathbf{o}(d).$$

Eine Schranke für das Produkt aller $\lambda_i(L)$ und eine Verschärfung der Ungleichung 3.6 liefert die Minkowski-Ungleichung.

Satz 3.1.2

Sei $L \in \mathbb{R}^n$ ein Gitter mit $\dim(L) = d$. Dann gilt für die euklidische Norm

$$\prod_{i=1}^d \lambda_i(L) \leq \sqrt{\gamma_d^d} \cdot \det(L).$$

3.2 Unimodulare Transformation

Die Vektoren können bzgl. ihrer euklidischen Norm sehr groß sein, wobei eine Gitterbasis mit möglichst kurzen Vektoren erreicht werden soll. Solche trivialen Basen sind einfach die Basen, deren Vektoren orthogonal und normiert sind. Dies kann jedoch nicht für jede gegebenen Basis angegeben werden, weil eine solche nicht für jedes Gitter existiert. Um einen solchen Basis-Wechsel bzw. eine Basistransformation zu erreichen, wird eine formale Beschreibung, die so genannte **unimodulare** Transformation benötigt.

Definition 3.2.1 (Unimodulare Matrix)

Die Basismatrix $U \in \mathbb{Z}^{d \times d}$ heißt **unimodular**, falls $\det(U) = \pm 1$.

Die multiplikative Gruppe der ganzzahligen, invertierbaren, unimodularen $d \times d$ Matrizen wird mit $GL_d(\mathbb{Z}) := \left\{ U \in \mathbb{Z}^{d \times d} \mid \det(U) = \pm 1 \right\}$ bezeichnet.

Eine **unimodulare** Matrix beschreibt eine **unimodulare** Transformation des Gitterbasis $B \rightarrow BU$. Die inverse der Matrix B ist ebenfalls **unimodular**. Dies spielt in der ganzzahligen Optimierung eine große Rolle ¹

Satz 3.2.1

Seien $A, B \in \mathbb{Z}^{n \times d}$ zwei ganzzahligen Gitterbasismatrizen. Dann ist $L(A) = L(B)$ genau dann, wenn es eine **unimodulare** Matrix U gibt, sodass $AU = B$

Es soll erneut das Beispiel 3.1.1 betrachtet werden. Dabei wird festgestellt, dass

$$L(A) = L(B) \text{ für } A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \text{ und } B = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

¹Um die ganzzahlige Lösung eines Gleichungssystems $x = A^{-1}b$ zu garantieren, stellt die Cramersche Regel sicher, dass A^{-1} ganzzahlig ist, für eine Matrix A mit $\det(A) = \pm 1$.

denn

$$\underbrace{\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}}_B = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix}}_U = \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}}_A \cdot \underbrace{\begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}}_{U_1} \cdot \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}}_{U_2}$$

für eine **unimodulare** Matrix U , deren Determinante $\det U = -1$. Zu bemerken ist, dass U sich als Produkt zweier **unimodularer** Matrizen U_1 und U_2 darstellen lässt.

Satz 3.2.2

Die Gitterdeterminante ist unabhängig von der Wahl der Gitterbasis.

Beweis Seien A, B zwei Basismatrizen mit $L(A) = L(B)$. Dann existiert eine Matrix $U \in GL_d(\mathbb{Z})$ mit der Eigenschaft $AU = B$ und es gilt $\det L(A) = \det L(B)$, denn

$$\det(B^T B) = \det(U^T A A U) = \det(U^T) \det(A^T A) \det(U) = \det(A^T A).$$

Definition 3.2.2 (Unimodulare Transformation)

*Die folgenden Operationen an einer Matrix heißen **unimodulare** (elementare) Spalten- (Zeilen-) Operationen:*

- Vertauschen zweier Spalten (Zeilen).
- Multiplikation einer Spalte (Zeile) mit -1 .
- Addition eines ganzzahligen Vielfachen einer Spalte (Zeile) zu einer anderen Spalte (Zeile).

Definition 3.2.3 (Die Hermitesche Normalform (HNF))

*Eine Matrix $B = [b_{ij}] = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ mit $d \leq n$ ist in **Hermitesche Normalform**, wenn*

1. $b_{ij} = 0$ für $i < j$, d.h. B ist eine untere Dreiecksmatrix.
2. $a_{ii} > 0$ für $i \in \{1, 2, \dots, n\}$.
3. $0 \leq a_{ij} < a_{ii}$ für $j < i$.

Vereinfacht dargestellt versteht man unter einer Hermite-Normalform einer ganzzahligen Matrix A eine zu dieser Matrix äquivalente untere Dreiecksmatrix H , bei der die Einträge unterhalb der Diagonalen modulo dem zugehörigen

Diagonalelement reduziert sind, d.h. betragsmäßig zwischen 0 und dem jeweiligen Diagonalelement liegen. Eine Matrix A ist zur einer Matrix H genau dann äquivalent, wenn eine ganzzahlige, unimodulare Matrix U existiert, so dass gilt $AU = H$.

Beispiel 3.2.1

Sei

$$A = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

dann lautet die **HNF** von A

$$H = AU = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}$$

3.3 Gitterprobleme

Interessant bei einem Gitter $L \in \mathbb{R}^n$ mit der Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ sind die kurzen Gittervektoren und die Annäherung eines Punktes im \mathbb{R}^n durch einen Gittervektor. Dieses Problem wird in den meisten Anwendungen von Gittern auf eines der folgenden Berechnungsprobleme zurückgeführt.

Problem 3.3.1 (Approximiertes Shortest Vector Problem, α -SVP)

Das kürzeste Vektor-Problem lautet:

1. Gegeben sei eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eines Gitters L und ein Approximationsfaktor $\alpha \geq 1$.
2. Bestimme einen Gittervektor $b \in L, b \neq \mathbf{0}$, für den $\|b\| \leq \alpha \lambda_1(L)$ ist.

Problem 3.3.2 (Shortest Vector Problem, SVP)

Das kürzeste Vektor-Problem lautet:

1. Gegeben sei eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eines Gitters L .
2. Bestimme einen Gittervektor $b \in L, b \neq \mathbf{0}$, für den $\|b\| = \lambda_1(L)$ gilt.

Problem 3.3.3 (Approximiertes Closest Vector Problem, α -CVP)

Das nächste Vektor-Problem lautet:

1. Gegeben sei eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eines Gitters L , ein Vektor $b \in \mathbb{R}^n$ und ein Approximationsfaktor $\alpha \geq 1$.
2. Bestimme einen Gittervektor $b_1 \in L, b_1 \neq \mathbf{0}$, für den gilt

$$\|b - b_1\| \leq \alpha \min\{\|b - b_2\| \mid b_2 \in L\}.$$

Problem 3.3.4 (Closest Vector Problem, CVP)

Das nächste Vektor-Problem lautet:

1. Gegeben sei eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eines Gitters L und ein Vektor $b \in \mathbb{R}^n$.
2. Bestimme einen Gittervektor $b_1 \in L, b_1 \neq \mathbf{0}$, für den gilt

$$\|b - b_1\| = \min\{\|b - b_2\| \mid b_2 \in L\}.$$

Problem 3.3.5 (Approximate Shortest Basis Problem, α -SBP)

Das approximierte kürzeste Basis-Problem lautet:

1. Gegeben sei eine Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ und ein Approximationsfaktor $\alpha \geq 1$.
2. Bestimme eine Gitterbasis $B' = [b'_1, b'_2, \dots, b'_d] \in \mathbb{R}^{n \times d}$ von L , so dass

$$\prod_{j=1}^d \|b'_j\| \leq \alpha \min \left\{ \prod_{j=1}^d \|a_j\| \mid [a_1, a_2, \dots, a_d] \in \mathbb{R}^{n \times d} \text{ ist eine Basis von } L \right\}$$

oder anderes ausgedrückt

$$\text{odef}(B') \leq \alpha \min\{\text{odef}(A) \mid A \text{ ist eine Basis von } L\}.$$

4 Orthogonalisierungsverfahren

Die Vektoren eines Gitters L können bezüglich ihrer euklidischen Norm sehr groß sein und man möchte eine Gitterbasis mit möglichst kurzen Vektoren finden. Bei einem Vektorraum ist es in der Regel nicht schwierig, da dort immer eine sog. Orthogonalbasis angegeben werden kann. Zu einer vorhandenen Gitterbasis $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ des Gitters L könnte es im Allgemeinen jedoch keine Orthogonalbasis erzeugt werden, denn eine solche existiert nicht für jedes Gitter. Das liegt daran, dass es aus einer gegebenen Gitterbasis nur durch unimodulare Transformationen zu einer neuen Gitterbasis kommen könnte.

Nun soll zu der gegebenen Gitterbasis B des Gitters L die Orthogonalisierung betrachtet werden. Dabei wird versucht zu den linear unabhängigen Vektoren $b_i \in B$ ein Orthogonalsystem von d paarweise orthogonalen Vektoren \hat{b}_i zu finden, das denselben Untervektorraum erzeugt. Gesucht wird also nach einer Gitterbasis $\hat{B} = [\hat{b}_1, \hat{b}_2, \dots, \hat{b}_d] \in \mathbb{R}^{n \times d}$ für die gegebene Gitterbasis B , deren Vektoren orthogonal bzw. senkrecht zueinander sind, d.h.

$$\forall \hat{b}_i, \hat{b}_j \in \hat{B} : 1 \leq i \neq j \leq d \text{ gilt } \langle \hat{b}_i, \hat{b}_j \rangle = 0 \wedge \hat{b}_i \neq \mathbf{0} \forall i \in \{1, 2, \dots, d\}.$$

Sodann werden wir in diesem Kapitel einige Verfahren für die Orthogonalisierung bzw. für die **QR**-Zerlegung der Gitterbasismatrix B eines Gitters L , die bei der Gitterbasisreduktion benutzt werden, vorgestellt. In Kapitel 5 auf Seite 35 wird deutlich, dass die Orthogonalisierung ein elementarer Bestandteil von Algorithmen zur Gitterbasisreduktion ist. Die Diskussion von den einsetzbaren Verfahren zur Gitterbasisreduktion und ihrer numerischen Eigenschaften erfolgt in den Abschnitten 4.1, 4.2 und 4.3. Auch die Gitterreduktion, die in Kapitel 5 in Algorithmus 5.3.5 auf Seite 47 behandelt wird, basiert auf der Verfügbarkeit von R .

Über die Eigenschaften der QR -Zerlegung informiert der folgende Satz.

Satz 4.0.1 (QR-Zerlegung)

Sei $[b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ eine Matrix mit vollen Spaltenrang. Dann ist die QR-Zerlegung $B := QR$ der Basis B die eindeutige Zerlegung $B := QR$, so dass $Q \in \mathbb{R}^{n \times d}$ isometrisch ist und $R \in \mathbb{R}^{d \times d}$ obere Dreiecksmatrix mit positiven Diagonalelementen. R wird die geometrische Normalform (**GNF**) der Basis, Bezeichnung: $R := \mathbf{GNF}(B)$. Es gilt

$$[b_1, b_2, \dots, b_d] = QR = [q_1, q_2, \dots, q_d] \begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,d} \\ 0 & r_{2,2} & \dots & r_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & r_{d,d} \end{bmatrix}. \quad (4.1)$$

Beweis: Der Beweis dieses Satzes kann in [GLC] nachgelesen werden.

Das bekannteste konstruktive Verfahren der Orthogonalisierung ist das Gram-Schmidt'sche Orthogonalisierungsverfahren. Dieses Verfahren wird in der numerischen Mathematik hauptsächlich benutzt, um die QR-Zerlegung zu erzeugen. Aus numerischer Sicht ist es jedoch, sofern nicht genügend Bit-Genauigkeit zur Verfügung steht, nicht empfehlenswert, da es instabil ist. Alternativ bieten sich Householder-Reflexionen und Givens-Rotationen zur Konstruktion von Q und R sowie die Cholesky Zerlegung zur Berechnung von R an. Diese Methoden sind numerisch deutlich besser als das Verfahren von Gram-Schmidt, und daher bei der Verwendung von Gleitpunktzahlen-Arithmetik diesem vorzuziehen. Im Zusammenhang mit Reduktion von Gitterbasen (bei Verwendung von Gleitkomma-Arithmetik) hat sich Henrik Koy bereits in [Koy99] mit Householder-Reflexionen und Givens-Rotationen beschäftigt. Diese beiden Verfahren werden als Grundlage für die Implementierung der Konzepte aus [KoySch] in unserem Programm gebraucht.

Die Unterschiede zwischen Householder-Reflexion und Givens-Rotation liegen zum ersten im Aufwand (die Anzahl der zur Zerlegung nötigen Operationen) und zum zweiten in ihrer numerischen Güte. Bevor wir in den nächsten Abschnitten die Verfahren näher beschreiben, geben wir in Tabelle 4.1 die interessantesten Eigenschaften der bisher angesprochenen Methoden wieder [Koy99]. Die Angaben beziehen sich auf die Zerlegung einer Matrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ unter Verwendung von Maschinenzahlen \mathbb{F} mit Maschinengenauigkeit τ . Dabei sei $R' = [r_{i,j}]_{1 \leq i, j \leq d} \in \mathbb{F}^{d \times d}$ die numerische Approximation der exakten Lösung $R = [r_{i,j}]_{1 \leq i, j \leq d} = Q^T B \in \mathbb{R}^{d \times d}$.

Methode	Operationen	Fehler $ r'_{j,l} - r_{j,l} $	Exakte Arithmetik
Gram-Schmidt	$\mathcal{O}(nd^2 + d^3)$	–	Ja
Householder	$\mathcal{O}(2d^2(n - \frac{d}{3}))$	$\leq 2.5n^2 \cdot 2^{-\tau} \ b_l\ $	nein
Givens	$\mathcal{O}(3d^2(n - \frac{d}{3}))$	$\leq 14n^2 \cdot 2^{-\tau} \ b_l\ $	nein

Tabelle 4.1: Aufwand und numerische Güte von Orthogonalisierungsverfahren

4.1 Gram-Schmidtsches Orthogonalisierungsverfahren

In diesem Abschnitt wird das Verfahren von Gram-Schmidt zu Orthogonalisierung dargestellt. Es ist ein elementarer Bestandteil des **LLL**-Algorithmus 5.2.2 zur Gitterbasisreduktion.

Definition 4.1.1 (Gram-Schmidt-Orthogonalisierung)

Zu der geordneten Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ bezeichne

$$\pi_i : \mathbb{R}^n \longrightarrow \text{span}(b_1, b_2, \dots, b_{i-1})^\perp$$

die orthogonale Projektion derart, dass für alle $b \in \mathbb{R}^n$

$$b - \pi_i(b) \in \text{span}(b_1, b_2, \dots, b_{i-1}).$$

Die Vektoren $\widehat{b}_i := \pi_i(b_i)$ sind für $i = 1, 2, \dots, d$ paarweise orthogonal. Sie werden durch das Gram-Schmidt-Verfahren berechnet

$$\widehat{b}_1 := \pi_1(b_1) := b_1, \widehat{b}_i := \pi_i(b_i) := b_i - \sum_{j=1}^{i-1} \mu_{i,j} \widehat{b}_j \text{ für } i = 1, 2, \dots, d, \quad (4.2)$$

wobei

$$\mu_{i,j} := \begin{cases} \frac{\langle b_i, \widehat{b}_j \rangle}{\langle \widehat{b}_j, \widehat{b}_j \rangle} & \text{für } i > j \\ 1, & \text{für } i = j \\ 0, & \text{sonst} \end{cases} \quad (4.3)$$

die Gram-Schmidt-Koeffizienten sind. Der folgende Skizze in Abb. 4.1 soll die Erläuterung visualisieren:

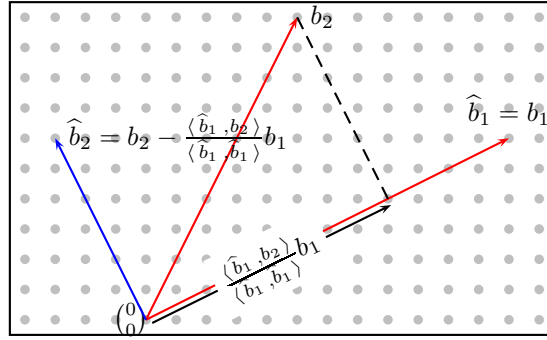


Abbildung 4.1: Geometrische Interpretation des Gram-Schmidt-Verfahrens

Diese Darstellung der Basisvektoren im Orthogonalsystem lautet in Matrixform:

$$\underbrace{[b_1, b_2, \dots, b_d]}_B = \underbrace{[\hat{b}_1, \hat{b}_2, \dots, \hat{b}_d]}_{\hat{B}} \cdot \underbrace{\begin{bmatrix} 1 & \mu_{2,1} & \dots & \mu_{d-1,1} & \mu_{d,1} \\ 0 & 1 & \dots & \mu_{d-1,2} & \mu_{d,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & \mu_{d,d-1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{[\mu_{i,j}]_{1 \leq i,j \leq d}^T}. \quad (4.4)$$

Aus der Darstellung 4.4 kann unmittelbar die sogenannte **QR**-Zerlegung der Basismatrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ in eine orthogonale Matrix $Q \in \mathbb{R}^{n \times d}$ und eine obere Dreiecksmatrix $R \in \mathbb{R}^{d \times d}$ abgeleitet werden. Somit gilt

$$Q := [q_1, q_2, \dots, q_n] = \left[\frac{\hat{b}_1}{\|\hat{b}_1\|}, \frac{\hat{b}_2}{\|\hat{b}_2\|}, \dots, \frac{\hat{b}_d}{\|\hat{b}_d\|} \right]$$

und für $R := [r_{i,j}]_{1 \leq i,j \leq d}$

$$r_{i,i} = \|\hat{b}_i\|, \quad r_{i,j} := \mu_{j,i} \cdot r_{i,i}. \quad (4.5)$$

In Matrixform lautet das demnach

$$R := \begin{bmatrix} \|\widehat{b}_1\| & 0 & \dots & \dots & 0 \\ 0 & \|\widehat{b}_2\| & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \|\widehat{b}_{d-1}\| & 0 \\ 0 & \dots & 0 & 0 & \|\widehat{b}_d\| \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 1 & \mu_{2,1} & \dots & \mu_{d-1,1} & \mu_{d,1} \\ 0 & 1 & \dots & \mu_{d-1,2} & \mu_{d,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & \mu_{d,d-1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{[\mu_{i,j}]_{1 \leq i,j \leq d}^T}.$$

1: **Input:** Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$
 2: **Output:** $[\widehat{b}_1, \widehat{b}_2, \dots, \widehat{b}_d]$, $[q_1, q_2, \dots, q_d]$, $[\mu_{i,j}]_{1 \leq i,j \leq d}^T$ und $[r_{i,j}]_{1 \leq i,j \leq d}$
 3: $\mu_{1,1} \leftarrow 1$; $\widehat{b}_1 \leftarrow b_1$; $r_{1,1} = \|\widehat{b}_1\|$;
 4: **for** ($i = 2$ to d)
 5: $\mu_{i,i} \leftarrow 1$; $r_{i,i} = \|\widehat{b}_i\|$; $\text{sum} \leftarrow 0$;
 6: **for** ($j = 1$ to $i - 1$)
 7: $\mu_{i,j} \leftarrow \frac{\langle b_i, \widehat{b}_j \rangle}{\langle \widehat{b}_j, \widehat{b}_j \rangle}$; $\mu_{j,i} \leftarrow 0$;
 8: $r_{j,i} \leftarrow \mu_{i,j} \cdot r_{j,j}$; $r_{i,j} \leftarrow 0$;
 9: $\text{sum} \leftarrow \text{sum} + \mu_{i,j} \cdot \widehat{b}_j$; $\triangleright \sum_{j=1}^{i-1} \mu_{i,j} \widehat{b}_j$
 10: **end for**
 11: $\widehat{b}_i \leftarrow b_i - \text{sum} \cdot \widehat{b}_j$; $q_i = \frac{b_i}{\widehat{b}_i}$; $\triangleright \widehat{b}_i := b_i - \sum_{j=1}^{i-1} \mu_{i,j} \widehat{b}_j$
 12: **end for**

Algorithmus 4.1.1: Die QR -Zerlegung mittels Gram-Schmidt Orthogonalisierung

4.2 Householder Reflections

Die Idee des Householder-Verfahrens angewendet auf Matrizen besteht darin, aus einer gegebenen Matrix B elementare Schrittweise orthogonale Matrizen $H^i \in \mathbb{R}^{n \times n}$ so zu konstruieren, dass $H^{n-1} \dots H^2 H^1 B$ eine obere Dreiecksmatrix

ist. Für eine Matrix $B \in \mathbb{R}^{3 \times 3}$ lässt sich die Householder-Transformation H^i schematisch wie folgt beschreiben:

$$B \xrightarrow{H^1} \left[\begin{array}{c|ccc} \star & \star & \star & \star \\ \hline 0 & \star & \star & \star \\ 0 & \star & \star & \star \\ 0 & \star & \star & \star \end{array} \right] \xrightarrow{H^2} \left[\begin{array}{cc|cc} \star & \star & \star & \star \\ \hline 0 & \star & \star & \star \\ 0 & 0 & \star & \star \\ 0 & 0 & \star & \star \end{array} \right] \xrightarrow{H^3} \left[\begin{array}{ccc|c} \star & \star & \star & \star \\ \hline 0 & \star & \star & \star \\ 0 & 0 & \star & \star \\ \hline 0 & 0 & 0 & \star \end{array} \right] = R.$$

Definition 4.2.1

Die Householdertransformation

$$H = E - 2\|v\|^{-2}vv^T \quad (4.6)$$

beschreibt die Spiegelung an der Hyperebene H orthogonal zu einem Vektor durch Null in einem euklidischen Raum. Im dreidimensionalen Raum ist sie eine lineare Abbildung, die eine Spiegelung an einer Ebene (durch den Ursprung und senkrecht auf v) beschreibt. Der Vektor v heißt Householder-Vektor.

Abbildung 4.2 veranschaulicht das Verfahren. Im ersten Schritt wird aus der

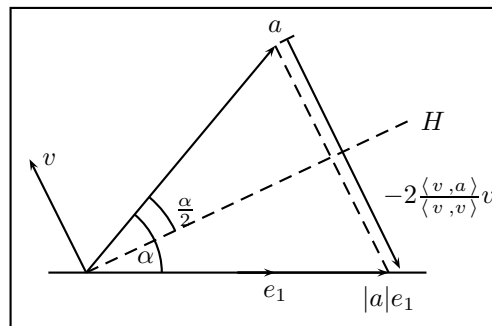


Abbildung 4.2: Householder-Reflexion

gegebenen Matrix $B^1 = B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ wird eine orthogonale Matrix $H^1 \in \mathbb{R}^{n \times n}$ so bestimmt, dass $H^1 B^1$ die Form $\begin{bmatrix} \star & \star \\ 0 & \tilde{B}^1 \end{bmatrix}$ hat. Im

nächsten Schritt ist analog zu der Berechnung von H^1 aus der Matrix \tilde{B}^1 die um einen Rang reduzierte Matrix $\tilde{H}^2 \in \mathbb{R}^{n-1 \times n-1}$ und daraus die Matrix $H^2 \in \mathbb{R}^{n \times n}$ zu bestimmen, wobei $H^2 = \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & \tilde{H}^2 \end{array} \right]$ ist. Schließlich entsteht die Matrix $R \in \mathbb{R}^{n \times d}$ und $Q = H^{n-1} \dots H^2 H^1$, indem die einzelnen Householder-Matrizen berechnet und miteinander multipliziert werden.

Wird H auf einen Vektor $r \in \mathbb{R}^n$ angewandt, so gilt

$$r \longrightarrow Hr = \left(E_n - 2 \frac{vv^T}{v^T v} \right) r = r - 2 \frac{v^T r}{v^T v} v. \quad (4.7)$$

Definition 4.2.2 (Erzeugungs-Vektor, Householder-Vektor)

Zu $j \in \{1, 2, \dots, n\}$ und dem Erzeugungs-Vektor $r = r_j = (r_1, r_2, \dots, r_n)^T \in \mathbb{R}^n$ definiere die von r erzeugte Householder-Matrix $H^j(r) \in \mathbb{R}^{n \times n}$ mittels Ersetzen des Vektors v in der Formel (4.6) durch den Householder-Vektor v^j

$$v = v^j(r) = (0, 0, \dots, 0_{j-1}, t + r_j, r_{j+1}, \dots, r_n)^T.$$

Dabei ist

$$t = t^j(r) = \mathbf{sign}(r_j) \left(\sum_{i=j}^n r_i^2 \right)^{\frac{1}{2}}, \quad \mathbf{sign}(r_j) = \begin{cases} 1, & : r_j \geq 0 \\ -1, & : r_j < 0 \end{cases}. \quad (4.8)$$

Mit $t^2 = \sum_{i=j}^n r_i^2$ folgt

$$v^T v = \|v\|^2 = (t + r_j)^2 + r_{j+1}^2 + \dots + r_n^2 = t^2 + 2tr_j + t^2 = 2tr_j + 2t^2 \quad (4.9)$$

und

$$2v^T r = 2(tr_j + t^2).$$

Es gilt dann

$$\frac{2v^T r}{v^T v} = 1 \quad (4.10)$$

und damit das folgende Lemma

¹Für Numerische Stabilität durch Vermeidung von Auslöschung

Lemma 4.2.3 Für die durch den Erzeugungs-Vektor $r = (r_1, r_2, \dots, r_n)^T \in \mathbb{R}^n$, den Householder-Vektor v^j und Einsetzen in der Formel (4.7) definierte Householder-Matrix $H = H^j(r)$ gilt:

$$H^j r = r - v^j = (r_1, r_2, \dots, r_{j-1}, -t, 0, 0, \dots, 0)^T \in \mathbb{R}^n. \quad (4.11)$$

Nun kann der Algorithmus aus der gewonnenen Gleichungen formuliert und zudem die folgende Rekursion definiert werden

$$R_0 = B \quad R^j = H^j R^{j-1} \quad \text{für } j = 1, 2, \dots, n \quad (4.12)$$

$$R = R^n \quad H = H^n H^{n-1} \dots H^1 = (H^1)^T (H^2)^T \dots (H^n)^T. \quad (4.13)$$

Dabei ist die Matrix H^j , die über die Gleichung 4.6, vom Vektor b_j erzeugte Householder-Matrix und orthogonal ist. Am Ende des $(n-1)$ -ten Schrittes vorliegende Matrix $R = R^n$ ist von oberer Dreiecksgestalt.

Bei der Gitterbasisreduktion In Kapitel 5.3.2 Algorithmus 5.3.3 auf Seite 42 wird zu sehen sein, dass an der orthogonalen Matrix Q nicht explizit interessiert ist. Diese wird daher nicht berechnet bzw. gespeichert.

Input: Basisvektoren $b_1, b_2, \dots, b_l \in \mathbb{Z}^n$ $v^1, v^2, \dots, v^{l-1} \in \mathbb{F}^n$ und $r_1, r_2, \dots, r_{l-1} \in \mathbb{F}^n$

Output: $r_l = (r_{l,1}, r_{l,2}, \dots, r_{l,n})$

1: $r = b_l$

2: **for** ($j \leftarrow 1$ **to** $l-1$)

3: $r \leftarrow r - 2\langle v^j, r \rangle v^j$

4: **end for**

5: $t = \text{sign}(r_l) (\sum_{i=l}^n r_i^2)^{\frac{1}{2}}$ ▷ Gleichung 4.8

6: $v^l = (0, 0, \dots, 0_{l-1}, t + r_l, r_{l+1}, \dots, r_n)^T / \sqrt{2tr_l + 2t^2}$ ▷ Normierung von v^l

4.9

7: $r_l = (r_1, r_2, \dots, r_{l-1}, -t, 0, 0, \dots, 0)^T$ ▷ Gleichung 4.11

Algorithmus 4.2.1: Orthogonalisierungsabschnitt des **Householder-**Algorithmus

Im Schritt 1 des Algorithmus 4.2.1 ist der Vektor r zu Beginn eine Approximation des Basisvektors b_l , das heißt die Einträge des Vektors r sind

Gleitpunktzahlen Varianten der ganzzahligen Einträge der Vektoren b_l . Die Basisvektoren werden sukzessiv orthogonalisiert. Dabei werden zusätzlich zu den berechneten Vektoren r_l die Householder-Vektoren v^j und die Zwischenergebnisse gespeichert. Die r_l werden sukzessive durch den Algorithmus 4.2.2 bestimmt. Für die Orthogonalisierung des l -ten Basisvektors werden die vorhergehenden $l - 1$ Householder-Vektoren benötigt. Da die Householder-Vektoren stets gleich bleiben, ist es sinnvoll, jeden von ihnen einmal zu berechnen und dann zur weiteren Verfügbarkeit zu speichern.

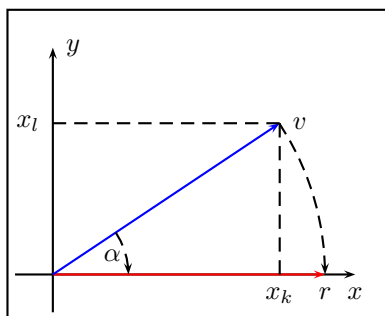
Input: Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$
Output: $R = [r_1, r_2, \dots, r_d]$
1: **for** ($l \leftarrow 1$ to d)
2: Führe den Algorithmus 4.2.1 mit Eingabe $[b_1, b_2, \dots, b_l] \in \mathbb{R}^{n \times l}$ aus!
3: **end for**

Algorithmus 4.2.2: Berechnung des Faktors R mittels **Householder**-Algorithmus

4.3 Givens-Rotations

Givensrotationen sind numerisch stabil, jedoch sollten sie nur mit Gleitkommaarithmetik verwendet werden, da sie mit exakter Arithmetik große Zahlen erzeugen können.

Die Givens-Rotation ist eine Drehung in einer Ebene, die durch zwei Koordinatenachsen aufgespannt wird. Als Givens Rotationen werden Matrizen der

Abbildung 4.3: Rotation von v bei einem Winkel α

Form

$$G_{i,j} = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \ddots & & & & & \vdots \\ \vdots & & c & & s & & \vdots \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & -s & & c & & \vdots \\ \vdots & & & & & \ddots & 0 \\ 0 & \dots & \dots & \dots & \dots & 0 & 1 \end{bmatrix} \begin{array}{l} \leftarrow \text{Zeile } i \\ \leftarrow \text{Zeile } j \end{array}$$

\uparrow \uparrow
 Spalte i Spalte j

bezeichnet, die von Wallace Givens im Jahre 1953 erstmalig verwendet wurden. Dabei stehen c und s für $\cos(\alpha)$ und $\sin(\alpha)$ in Relation $c^2 + s^2 = 1$. Geometrisch gesehen beschreibt die obige Matrix eine Drehung um den Winkel α in der (i, j) Ebene. Diese geometrische Interpretation spielt hier eine untergeordnete Rolle, so dass die Abhängigkeit von α nicht explizit angegeben wird.

Es ist erkennbar, dass $G_{i,j}^T G_{i,j} = E$. Das heißt, es handelt sich um orthogonale Matrizen. Wird $G_{i,j}$ auf einen Vektor $v \in \mathbb{R}^n$ angewandt, so folgt

$$y = G_{i,j}v \text{ wobei } y_k = (G_{i,j}v)_k = \begin{cases} cv_i + sv_j, & \text{wenn } k = i \\ -sv_i + cv_j, & \text{wenn } k = j \\ v_k, & \text{sonst.} \end{cases}$$

Wird die Matrix $B = [b_1, b_2, \dots, b_d] \in \mathbb{R}^{n \times d}$ von links mit der Matrix $G_{i,j}$

multipliziert so operiert die Givens-Rotation auf den Spalten, das heißt

$$G_{i,j}B = [G_{i,j}b_1, G_{i,j}b_2, \dots, G_{i,j}b_d].$$

Nun bleibt die Frage wie die c und s zu wählen sind, damit eine Komponente des Vektors y zu Null gemacht wird! Im Spezialfall $n = 2$ gilt

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_i \\ x_j \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \quad \text{d.h.} \quad \begin{aligned} x_i c + x_j s &= r \\ -x_i s + x_j c &= 0 \end{aligned}$$

und daraus ergibt sich dann

$$r = \sqrt{x_i^2 + x_j^2}, c = \frac{x_i}{r} \quad \text{und} \quad s = \frac{x_j}{r}.$$

Zur Vermeidung von Exponentenüberlauf ist es günstiger, die Parameter c und s mit dem Algorithmus zu berechnen.

```

1: Input:  $x_i$  und  $x_j$ 
2: if ( $x_j = 0$ )
3:    $c \leftarrow 1$ ;  $s \leftarrow 0$ ;
4: else
5:   if ( $|x_i| < |x_j|$ )
6:      $\nu \leftarrow \frac{x_i}{x_j}$ ;  $s \leftarrow \frac{1}{\sqrt{1+\nu^2}}$ ;  $c \leftarrow s\nu$ ;
7:   else
8:      $\nu \leftarrow \frac{x_j}{x_i}$ ;  $c \leftarrow \frac{1}{\sqrt{1+\nu^2}}$ ;  $s \leftarrow c\nu$ ;
9:   end if
10: end if
11: return  $[c, s] = \text{rotate}(x_i, y_i)$ 

```

Algorithmus 4.3.1: Givens Rotations. $\text{rotate}(x_i, y_i)$

Wird nun die Matrix $G_{i,j}$ auf den Vektor v gewandt, so erhalten wir

$$G_{i,j}v = G_{i,j} \cdot [v_1, v_2, \dots, v_i, \dots, v_j, \dots, v_n]^T = [v_1, v_2, \dots, \sqrt{x_i^2 + x_j^2}, \dots, 0, \dots, v_n]^T.$$

Für eine Matrix $B \in \mathbb{R}^{3 \times 3}$ lässt sich die Givens Transformation $G_{i,j}$ schematisch wie folgt beschreiben:

$$B = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{G_{2,3}} \begin{bmatrix} \times & \times & \times \\ \star & \star & \star \\ 0 & \star & \star \end{bmatrix} \xrightarrow{G_{1,2}} \begin{bmatrix} \star & \star & \star \\ 0 & \star & \star \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G_{2,3}} \begin{bmatrix} \times & \times & \times \\ 0 & \star & \star \\ 0 & 0 & \star \end{bmatrix} = R.$$

Dabei sind mit \times die Werte der Matrix gekennzeichnet, die nicht geändert wurden und mit \star die Werte zum Ausdruck gebracht, die durch die Formel 4.3 modifiziert wurden. Somit kann daraus abgelesen werden, dass die QR -Zerlegung der Matrix durch elementare Umformungen der Matrix B entsteht und damit gilt

$$G_{n-1,n}G_{n-2,n-1} \dots G_{1,2}G_{n-1,n}G_{n-2,n-1} \dots G_{2,3} \dots G_{n-1,n}B = R.$$

Wird die Bezeichnung

$$Q = (G_{n-1,n}G_{n-2,n-1} \dots G_{1,2}G_{n-1,n}G_{n-2,n-1} \dots G_{2,3} \dots G_{n-1,n})^T$$

eingeführt, so lässt sich A in der faktorisierten Form $A = QR$ angeben.

5 Gitterbasisreduktion

Die Gitterbasisreduktion beschäftigt sich mit der Aufgabe für ein gegebenes Gitter $L \subseteq \mathbb{R}^n$ eine reduzierte Gitterbasis davon zu finden, in der die Längen der Basisvektoren den sukzessiven Minima entsprechen und orthogonal zueinander stehen. Die Lösung dieses Problem ist schwierig und besitzt nur Algorithmen mit exponentieller Laufzeit. Sehr oft genügt es statt eines kürzesten Vektors (Siehe **SVP** auf Seite 3.3.2) nur einen relativ kurzen Vektor (Siehe α -**SVP** auf Seite 3.3.1) zu berechnen, also einen Vektor, der bis auf einen Faktor α die Länge von λ_1 hat. Der Algorithmus, der dabei eingesetzt wird, bezeichnet man als **LLL**-Algorithmus oder auch L3-Algorithmus.

In den nächsten Abschnitten werden hauptsächlich ganzzahlige Gitter statt reeller Gitter behandelt, dafür sind fast alle Algorithmen bereitgestellt. Die Komplexität des Problems wird dadurch nicht wesentlich geringer, aber auf Grund der Tatsache, dass Kryptosysteme und viele andere algorithmische Probleme auf ganzzahlige Argumente basieren, kann mit dieser Annahme gearbeitet werden.

Zunächst beschreiben wir in Abschnitt 5.1 die Gauß-reduzierten Gitterbasen der Dimension 2 sowie die Größenreduktion von Gitterbasen beliebiger Dimension. In Abschnitt 5.2 beschäftigen wir uns mit dem **LLL**-Algorithmus in exakter Arithmetik. Dieser Algorithmus basiert auf der Gauß-Reduktion in Dimension 2. Bei einer Eingabe einer Gitterbasis $B = [b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ und einer Konstante $M = \max_{1 \leq i \leq d} \|b_i\|^2$ benötigt das Verfahren $\mathcal{O}(d^3 n \log M)$ arithmetische Schritte. Als weitere Variante beschreiben wir in Abschnitt 5.3 den **LLL**-Algorithmus mit iterativer Orthogonalisierung in Gleitkomma-Arithmetik. Ferner besprechen wir in Abschnitt 5.3.2 den **LLL**-Algorithmus in Gleitkomma-Arithmetik basierend auf der **QR**-Zerlegung. Schließlich werden andere zu diesem Zweck vorgestellten Algorithmen erwähnt.

5.1 Größenreduktion

Wir betrachten erneut die Basen B_1 und B_2 des Gitters L aus dem Beispiel 3.1.1, welches auf Seite 16 zu finden ist. Die Äquivalenz der Basen dieser Gitter wurde durch unimodulare Transformation auf Seite 3.2 gezeigt. Wir stellen fest, dass die Längen der Vektoren in B_2 den sukzessiven Minima entsprechen. Ferner sind die Vektoren nach Def. 5.1.2 größenreduziert.

Definition 5.1.1 (Gauß-reduzierte Basis)

Eine geordnete Gitterbasis $b_1, b_2 \in \mathbb{Z}^n$ ist Gauß-reduziert bezüglich der Euklidischen Norm, wenn

$$\|b_1\| \leq \|b_2\| \leq \|b_1 \pm b_2\|.$$

Für Gauß-reduzierte Basen b_1, b_2 ist b_1 ein kürzester Gittervektor. Der Algorithmus 5.1.1 bestimmt eine Gauß-reduzierte Basis. Abbildung 5.1 zeigt die Reduktionsbedingung im Fall des Standard Skalarproduktes. Der Winkel zwischen den beiden Gittervektoren der reduzierten Basis $B = [b_1, b_2] \in \mathbb{Z}^{n \times 2}$ mit $\mu_{2,1} \geq 0$ sollte im Bereich $[\frac{\pi}{3}, \frac{\pi}{2}] \cup [-\frac{\pi}{3}, -\frac{\pi}{2}]$ liegen.

<p>t</p> <ol style="list-style-type: none"> 1: Input: Gitterbasis $B = [b_1, b_2] \in \mathbb{R}^{n \times 2}$ mit $\ b_1\ \leq \ b_2\$ 2: Output: Reduzierte Gitterbasis $B = [b_1, b_2] \in \mathbb{R}^{n \times 2}$ 3: while ($\mu_{21} > \frac{1}{2}$) ▷ $\mu_{21} \leftarrow b_1^T b_2 \ b_1\ ^{-2}$ 4: $b_2 \leftarrow b_2 \cdot \text{sign}(\mu_{21})$ ▷ Damit wird erreicht $\mu_{2,1} \geq 0$ 5: $b_2 \leftarrow b_2 - \lceil \mu_{21} \rceil b_1$ ▷ $\lceil \mu_{2,1} \rceil = \lceil \mu_{2,1} - \frac{1}{2} \rceil$ 6: if ($\ b_1\ > \ b_2\$) 7: vertausche b_1 und b_2 8: end if 9: end while
--

Algorithmus 5.1.1: Gauß-Reduktionsalgorithmus für die euklidische Norm

Definition 5.1.2

Eine Basis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ ist größenreduziert, wenn $|\mu_{i,j}| \leq \frac{1}{2}$ für

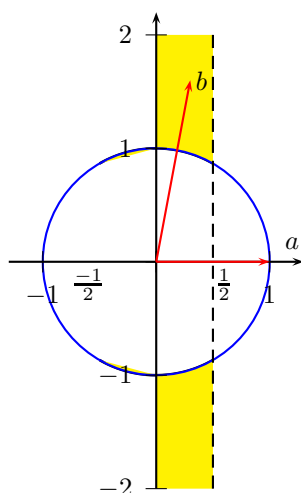


Abbildung 5.1: Bereich der reduzierten Basen b_1, b_2 mit $0 \leq \mu_{2,1} \leq \frac{1}{2}$

$1 \leq j < i \leq n$, mit

$$\mu_{i,j} := \begin{cases} \frac{\langle b_i, \widehat{b}_j \rangle}{\langle \widehat{b}_j, \widehat{b}_j \rangle} & \text{für } i > j \\ 1, & \text{für } i = j \\ 0, & \text{sonst.} \end{cases}$$

Satz 5.1.1

Für die größenreduzierte Basis $B = [b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ gilt

$$\|b_i\|^2 \leq \|\widehat{b}_i\|^2 + \frac{1}{4} \sum_{j=1}^{i-1} \|\widehat{b}_j\|^2. \quad (5.1)$$

Die Berechnung einer größenreduzierten Basis eines Gitters kann mit dem folgenden Algorithmus 5.1.2, der aus einer gegebenen Gitterbasis, die Gram-Schmidt Koeffizienten und eine Stufe l , in Polynomialzeit erfolgen, wobei $\lceil \mu_{l,j} \rceil^1$ das gewöhnliche Runden zu der nächsten ganzen Zahl bedeutet.

Beweis der Korrektheit des Algorithmus 5.1.2: Die Substitution $b_l := b_l - \mu_{l,j} b_j$ erzeugt einen kürzeren Gittervektor und erhält die lineare Unabhän-

¹ $\lceil \mu_{l,j} \rceil = \lceil \mu_{l,j} - \frac{1}{2} \rceil$

Input: Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ und $[\mu_{i,j}]_{1 \leq i, j \leq l}^T$

Output: Größenreduzierte Gittervektoren b_l für $1 \leq l \leq d$ bzw. Größenreduzierte Gitterbasis und Aktualisierung von $[\mu_{i,j}]_{1 \leq i, j \leq l}^T$

```

1: for ( $l \leftarrow 2$  to  $d$ )
2:   for ( $j \leftarrow l - 1$  to  $1$ )
3:      $\mu \leftarrow \lceil \mu_{l,j} \rceil$ 
4:     if ( $\mu \neq 0$ )
5:       for ( $i \leftarrow 1$  to  $j$ )
6:          $\mu_{l,i} \leftarrow \mu_{l,i} - \mu \mu_{j,i}$   $\triangleright \mu_l \leftarrow \mu_l - \mu \mu_i$ 
7:       end for
8:        $b_l \leftarrow b_l - \mu b_j$   $\triangleright$  Größenreduktion von  $b_l$ 
9:     end if
10:  end for
11: end for

```

Algorithmus 5.1.2: Größenreduktion

gigkeit. Für das neue $\mu_{l,i}$ gilt

$$\mu_{l,i} = \frac{\langle b_l - \mu_{l,j} b_j, \widehat{b}_i \rangle}{\|\widehat{b}_i\|^2} = \frac{\langle b_l, \widehat{b}_i \rangle}{\|\widehat{b}_i\|^2} - \lceil \mu_{l,j} \rceil \frac{\langle b_j, \widehat{b}_i \rangle}{\|\widehat{b}_i\|^2} = \mu_{l,i} - \lceil \mu_{l,j} \rceil \mu_{j,i}.$$

Komplexität des Algorithmus 5.1.2: Die Größenreduktion des Vektors b_l im Algorithmus 5.1.2 benötigt $2ld + l^2$ arithmetische Operationen. Zur Größenreduktion der Basis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ werden höchstens $\sum_{l=2}^d 2ln^2 + l^2 \approx nd^2 + \frac{d^2}{3}$ benötigt. Ein volldimensionales Gitter hat damit eine Laufzeit von $\mathcal{O}(d^3)$.

5.2 LLL-Reduktion

Der **LLL**-Reduktionsbegriff für geordnete Gitterbasis $b_1, \dots, b_d \in \mathbb{Z}^n$ beliebigen Ranges d wurde 1982 von A. K. Lenstra, H. W. Lenstra und L. Lovász in [LLL82] eingeführt. Das Verfahren zur **LLL**-Reduktion ist das erste Reduktionsverfahren für ganzzahlige Gitterbasis, es approximiert die sukzessiven Minima eines Gitters und erzeugt in Polynomialzeit eine reduzierte Gitterbasis,

die bis auf einen Faktor optimal ist. Der Faktor hängt allerdings exponentiell vom Rang des Gitters ab. Ferner verwendet es die Gramm-Schmidt Orthogonalisierung aus dem Algorithmus 4.1 sowie die Größenreduktion aus dem Algorithmus 5.1.2.

Definition 5.2.1 (LLL-reduzierte Basis)

Eine geordnete größenreduzierte Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ heißt **LLL-reduziert** zum Reduktionsparameter $\delta \in (\frac{1}{4}, 1)$, falls

$$\delta \|\widehat{b}_{l-1}\|^2 \leq \|\widehat{b}_l\|^2 + \mu_{l,l-1}^2 \|\widehat{b}_{l-1}\|^2 \quad \text{für } l \in \{2, 3, \dots, d\}. \quad (5.2)$$

Mit der orthogonalen Projektion π_l aus der Definition 4.1.1 auf Seite 25 gilt

$$\delta \|\widehat{b}_{l-1}\|^2 \leq \|\widehat{b}_l\|^2 + \mu_{l,l-1}^2 \|\widehat{b}_{l-1}\|^2 \iff \delta \|\pi_{l-1}(b_{l-1})\|^2 \leq \|\pi_{l-1}(b_l)\|^2.$$

Satz 5.2.1

Für jede mit $\delta \in (\frac{1}{4}, 1)$ **LLL-reduzierte** Gitterbasis $B = [b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ und ein $\alpha = (\delta - \frac{1}{2})^{-1}$ gilt

$$\alpha^{1-j} \lambda_j(L)^2 \leq \|\widehat{b}_j\|^2 \quad \text{für } j = 1, 2, \dots, d, \quad (5.3)$$

$$\|b_j\|^2 \leq \alpha^{d-1} \lambda_j(L)^2 \quad \text{für } j = 1, 2, \dots, d, \quad (5.4)$$

$$\|b_i\|^2 \leq \alpha^{j-1} \|\widehat{b}_j\|^2 \quad \text{für } 1 \leq i \leq j \leq d, \quad (5.5)$$

$$\|b_1\|^2 \leq \alpha^{\frac{d-1}{2}} (\det L)^{\frac{1}{d}}, \quad (5.6)$$

$$\prod_{j=1}^d \|b_j\|^2 \leq \alpha^{\frac{d(d-1)}{2}} (\det L)^2. \quad (5.7)$$

Der **LLL**-Algorithmus transformiert die gegebene Basis B in eine **LLL-reduzierte** Basis des gleichen Gitters. Er arbeitet stufenweise und iteriert solange bis die Lovász-Bedingung 5.2 für alle Vektoren erfüllt ist. In jeder Stufe wird nur der Vektor b_l orthogonalisiert und anschließend deren Größenreduktion berechnet. Im Schritt 10 wird überprüft, ob für die Vektor b_l und b_{l-1} die Tauschbedingung (Lovász Bedingung) erfüllt ist. Ist dies der Fall so ist auszutauschen und l zu verringern, sonst wird l um eins erhöht. Pro Tauschoperation müssen demnach $\mathcal{O}(d^2)$ arithmetische Operationen durchgeführt werden.

Input: Basisvektoren $b_1, b_2, \dots, b_l \in \mathbb{Z}^n$
Output: $\mu_{l,1}, \mu_{l,2}, \dots, \mu_{l,l}$ und $\|\widehat{b}_l\|^2$

- 1: $\mu_{l,l} \leftarrow 1$
- 2: **for** ($j \leftarrow 1$ **to** $l-1$)
- 3: $\mu_{l,j} \leftarrow \frac{\langle b_l, b_j \rangle - \sum_{i=1}^{j-1} \mu_{l,j} \mu_{j,i} \|b_i\|^2}{\|b_j\|^2}$
- 4: **end for**
- 5: $\|b_l\|^2 \leftarrow \langle b_l, b_l \rangle - \sum_{j=1}^{l-1} \mu_{l,j}^2 \|b_j\|^2$

Algorithmus 5.2.1: Orthogonalisierungsabschnitt der **LLL**-Reduktion

Satz 5.2.2

Der **LLL**-Algorithmus 5.2.2 benötigt für eine gegebene Basis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$, $M = \max_{1 \leq i \leq d} \|b_i\|^2$ und $\delta \in (\frac{1}{4}, 1)$ $\mathcal{O}(d^3 n \log M)$ arithmetische Schritte unter Verwendung von $d \log M$ -bit Zahlen.

Beweis von Satz 5.2.2 kann in [LLL82] nachgelesen werden.

5.3 LLL-Reduktion in Gleitkommaarithmetik

Der **LLL**-Algorithmus 5.2.2 arbeitet sehr stabil in exakter Arithmetik, ist jedoch in der Praxis langsam, da exakte Arithmetik hohe Laufzeiten benötigen. Aus diesem Grund besteht Interesse an einem schnellen, praktischen und stabilen Verfahren.

5.3.1 Der Schnorr-Euchner Algorithmus

Basierend auf dem **LLL**-Algorithmus 5.2.2 wurden verschiedene Ansätze vorgestellt, die mit Approximationen arbeiten und die die Laufzeit für die Gitterbasisreduktion deutlich verringern. Eines dieser Ansätze ist der von Schnorr und Euchner [SE93] entwickelten Algorithmus 5.3.2. Der Basis-Algorithmus 5.2.2 wurde so modifiziert, dass die Berechnungen der Gram-Schmidt Koeffizienten $[\mu_{i,j}]_{1 \leq i, j \leq d}^T$ sowie die Höhenquadrate $\|\widehat{b}_l\|^2$ in Gleitkommaarithmetik, die Größenreduktion in exakter Arithmetik durchgeführt werden.

Da die Anzahl der Präzisionsbits τ der primitiven Gleitkomma-Datentypen

Input:	Gitterbasis $B = [b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$, $\delta \in (\frac{1}{4}, 1)$
Output:	Mit Parameter δ LLL -reduzierte Basis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$
1:	$l \leftarrow 2, c_1 = \ b_1\ ^2$
2:	for ($i \leftarrow 1$ to d)
3:	$\mu_{i,i} \leftarrow 1$
4:	end for
5:	while ($l \leq d$) ▷ l ist die Stufe
6:	Berechne $\mu_{l,1}, \mu_{l,2}, \dots, \mu_{l,l}$ und $c_l = \ \widehat{b}_l\ ^2$ gemäß Algo. 5.2.1.
7:	Größenreduziere b_l und aktualisiere $\mu_{l,1}, \mu_{l,2}, \dots, \mu_{l,l-1}$ gemäß Algorithmus 5.1.2 Schritt 2-10.
8:	if ($\delta c_{l-1} > c_l + \mu_{l,l-1}^2 c_{l-1}$) ▷ Lovász-Bedingung
9:	swap (b_l, b_{l-1}) ▷ Vertausche die Vektoren b_l und b_{l-1}
10:	if ($l = 2$)
11:	update c_1 ▷ Aktualisierung von $c_1 = \ b_1\ ^2$
12:	end if
13:	$l \leftarrow \max(l - 1, 2)$ ▷ Verringere Stufe l um eins
14:	else
15:	$l \leftarrow l + 1$ ▷ erhöhe Stufe l um eins
16:	end if
17:	end while

Algorithmus 5.2.2: LLL-Reduktion

im Rechner beschränkt ist (siehe Kapitel 6.1.3 auf Seite 53 Tabelle 6.5), kann es zu einem Überlauf bzw. Unterlauf kommen, falls die Zahlen zu groß, zu klein bzw. die Resultate der Skalarprodukten der Vektoren sehr klein sind. Dies führt zur Instabilität der Algorithmus ab einer bestimmten Dimension. Im Algorithmus so wird das berücksichtigt, indem versucht wird, die Fehler zu minimieren.

Falls

$$|\langle b'_l, b'_j \rangle| \leq 2^{-\frac{\tau}{2}} \|b'_l\| \|b'_j\|$$

so wird das Skalarprodukt $\langle b_l, b_j \rangle$ mit Hilfe der exakten Darstellung der Basisvektoren anstelle der Gleitkomma-Werte exakt berechnet. Dies verhindert, dass der kleinere Absolutbetrag von der Gleitkommaarithmetik wegen großer Koeffizienten der Basisvektoren relativ stark vom korrekten Resultat abweicht. Tritt bei der Größenreduktion von b_l $\lceil |\mu_{l,j}| \rceil > 2^{\frac{\tau}{2}}$ auf, so wird dann vor dem

Lovász Bedingung neu iteriert. Dabei werden die Gram-Schmidt Koeffizienten $\mu_{l-1,j}$, $\mu_{l,j}$ für $j \in \{1, 2, \dots, l\}$ neu berechnet.

Input: Basisvektoren $b_1, b_2, \dots, b_l \in \mathbb{Z}^n$ $b'_1, b'_2, \dots, b'_l \in \mathbb{F}^n$
Output: $\mu_{l,1}, \mu_{l,2}, \dots, \mu_{l,l}$ und $c_l \leftarrow \|\widehat{b}_l\|^2$

- 1: **for** ($j \leftarrow 1$ **to** $l - 1$)
- 2: **if** ($|\langle b'_l, b'_j \rangle| \leq 2^{-\frac{\tau}{2}} \|b'_l\| \|b'_j\|$)
- 3: $s \leftarrow \langle \widehat{b}_l, b_j \rangle$
- 4: **else**
- 5: $s \leftarrow \langle b'_l, b'_j \rangle$
- 6: **end if**
- 7: $\mu_{l,j} \leftarrow \frac{s - \sum_{i=1}^{j-1} \mu_{l,j} \mu_{j,i} c_i}{c_j}$
- 8: $c_l \leftarrow c_l - \mu_{l,j}^2 c_j$
- 9: **end for**

Algorithmus 5.3.1: Orthogonalisierungsabschnitt des **SE**-Algorithmus

5.3.2 Der Schnorr-Koy Algorithmus

Dieser Ansatz versucht die Problematik der Instabilität bis zur Dimension 400 durch numerische stabilere Verfahren für die Orthogonalisierung wie Housholder-Reflexion, die bereits in Kapitel 4.2 eingeführt wurde, zu lösen. Die Transformation auf der ganzzahligen Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ werden weiter in exakter Arithmetik berechnet. Die Arithmetik auf den Zahlen $\mu_{i,j}$ und $\|\widehat{b}_j\|^2$ wird durch die τ -stellige Gleitpunkt-Arithmetik auf den reellwertigen Zahlen $r_{j,i} = \mu_{i,j} \|\widehat{b}_j\|^2$ durchgeführt.

Darüberhinaus werden die Höhenquadrate $\|\widehat{b}_{l-1}\|^2$ und $\|\widehat{b}_l\|^2$ aufgrund der Gleichung 4.5 in Kapitel 4.1 auf Seite 26 durch $\|\widehat{b}_{l-1}\|^2 = r_{l-1,l-1}^2$, $\|\widehat{b}_l\|^2 = r_l^2$ ersetzt. Ferner gilt für $\mu_{l,l-1}^2 \|\widehat{b}_{l-1}\|^2$

$$\mu_{l,l-1}^2 \|\widehat{b}_{l-1}\|^2 = \mu_{l,l-1}^2 r_{l-1,l-1}^2 = \mu_{l,l-1} r_{l-1,l-1} \mu_{l,l-1} r_{l-1,l-1} = r_{l-1,l} r_{l-1,l} = r_{l-1,l}^2.$$

Somit kann die Lovász-Bedingung umformuliert werden und es gilt

$$\delta \|\widehat{b}_{l-1}\|^2 \leq \|\widehat{b}_l\|^2 + \mu_{l,l-1}^2 \|\widehat{b}_{l-1}\|^2 \iff \delta r_{l-1,l-1}^2 \leq r_l^2 + r_{l-1,l}^2.$$

```

Input:    Gitterbasis  $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ ,  $\delta \in (\frac{1}{4}, 1)$ ,  $\eta$  (Standard  $\eta = \frac{1}{2}$ )
Output:  Mit Parameter  $\delta$  LLL-reduzierte Basis  $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ 
1: for ( $i \leftarrow 1$  to  $d$ )
2:    $b'_i \leftarrow (b_i)'$ ,  $\mu_{i,i} \leftarrow 1$ 
3: end for
4:  $l \leftarrow 2$ ,  $F \leftarrow \text{False}$ 
5: while ( $l \leq d$ )
6:    $c_l \leftarrow \|b'_l\|^2$ 
7:   if ( $l = 2$ )
8:      $c_1 \leftarrow \|b'_1\|^2$ 
9:   end if
10:  Berechne  $\mu_{l,1}, \mu_{l,2}, \dots, \mu_{l,l}$  und  $\|\widehat{b}_l\|^2$  gemäß Algorithmus 5.3.1
11:  for ( $j \leftarrow l - 1$  to  $1$ )
12:    if ( $|\mu_{l,j}| > \eta$ )
13:       $\mu \leftarrow \lceil \mu_{l,j} \rceil$ 
14:      if ( $|\mu| > 2^{\frac{\tau}{2}}$ )
15:         $F \leftarrow \text{True}$ 
16:      end if
17:      for ( $i \leftarrow 1$  to  $j$ )
18:         $\mu_{l,i} \leftarrow \mu_{l,i} - \mu \mu_{j,i}$  ▷  $\mu_l \leftarrow \mu_l - \mu \mu_i$ 
19:      end for
20:       $b_l \leftarrow b_l - \mu b_j$ ,  $b'_l \leftarrow (b_l)'$  ▷ Größenreduziere  $b_l$ 
21:    end if
22:  end for
23:  if ( $F$ )
24:     $F \leftarrow \text{False}$ ,  $l \leftarrow \max(2, l - 1)$  ▷ GOTO Zeile 5.3.2.5
25:  else
26:    if ( $\delta c_{l-1} \leq c_l + \mu_{l,l-1}^2 c_{l-1}$ )
27:       $l \leftarrow l + 1$  ▷ erhöhe Stufe  $l$ 
28:    else
29:      swap( $b_l, b_{l-1}$ ), swap( $b'_l, b'_{l-1}$ )
30:       $l \leftarrow \max(l - 1, 2)$ 
31:    end if
32:  end if
33: end while

```

Algorithmus 5.3.2: LLL Schnorr Euchner Algorithmus für Gleitkommaarithmetik

Bemerkungen

- Der Orthogonalisierungsabschnitt (Algorithmus 5.3.3 Schritt 4-11) benötigt für eine gegebene Basis $B = [b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ $\mathcal{O}(dn)$ arithmetische Schritte und eine Wurzelberechnung und somit benötigt der Schnorr-Koy Algorithmus 5.3.3 für die gegebene Basis, ein $\delta \in (\frac{1}{4}, 1)$ und $M = \max_{1 \leq i \leq d} \|b_i\|^2$ $\mathcal{O}(d^3 n \log M)$ arithmetische Schritte.
- Um die Vermeidung von Auslöschungen kann die Berechnung im Algorithmus 5.3.3 Schritt 8 für die Norm von r ersetzt werden durch

$$t = \mathbf{sign}(r_l) \left(\sum_{i=l}^n r_i^2 \right)^{\frac{1}{2}} = \mathbf{sign}(r_l) \kappa \left(\sum_{i=l}^n \left(\frac{r_i}{\kappa} \right)^2 \right)^{\frac{1}{2}} \text{ für } \kappa = \max_{1 \leq i \leq n} r_i.$$

Es ist möglich auf die Orthogonalisierungsphase (Schritt 2-8) während des Ablaufs des Algorithmus 5.3.3 zu verzichten, um eine Beschleunigung und statt der quadratischen Komplexität in der **while**-Schleife nur eine lineare zu erzielen. Voraussetzung dafür ist eine vollständige Orthogonalisierung bzw. die Berechnung der **QR**-Zerlegung für die gesamte Basismatrix vor der **while**-Schleife. Sollten die Basisvektoren in Schritt 22 vertauscht werden, so müssen die Spalten r_l und r_{l-1} von R auch vertauscht werden, da R und B isometrisch sind, d.h. die Matrix R ist nicht mehr von oberer Dreiecksform. Man nennt dann R' eine einfache gestörte Matrix in der Zeile l . Nach dem Vektortausch gilt

$$R' := [r_1, \dots, r_l, r_{l-1}, \dots, r_d] \cong [b_1, \dots, b_l, b_{l-1}, \dots, b_d] = B'$$

mit

$$R' = \begin{bmatrix} r_{1,1} & \dots & r_{1,l} & r_{1,l-1} & \dots & r_{1,d} \\ 0 & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & r_{l-1,l} & r_{l-1,l-1} & \dots & r_{l-1,d} \\ \vdots & & r_{l,l} & 0 & \dots & r_{l,d} \\ \vdots & & & & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & r_{d,d} \end{bmatrix}.$$

Soeben wurde im Kapitel 4 auf S. 23 Abschnitt 4.3 die **Givens**-Rotationen dargestellt. Es ist ohne Weiteres möglich mittels einer **Givens**-Rotation $G_{l-1,l}$

```

Input:    Gitterbasis  $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ ,  $\delta \in (\frac{1}{4}, 1)$ ,  $\eta = \frac{1}{2}$ 
Output:  Mit Parameter  $\delta$  LLL-reduzierte Basis  $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ 
1:  $l \leftarrow 1$ 
2: while ( $l \leq d$ )
3:   /* Orthogonalisierung */
4:    $r \leftarrow b_l$ 
5:   for ( $j \leftarrow 1$  to  $l-1$ )
6:      $r \leftarrow r - 2\langle v^j, r \rangle v^j$ 
7:   end for
8:    $t \leftarrow \text{sign}(r_l) (\sum_{i=1}^n r_i^2)^{\frac{1}{2}}$  ▷ Gleichung 4.8
9:    $z \leftarrow \sqrt{2tr_l + 2t^2}$  ▷ Die Norm von  $v_l$  4.9
10:   $v^l \leftarrow (0, 0, \dots, 0_{l-1}, \frac{t+r_l}{z}, \frac{r_{l+1}}{z}, \dots, \frac{r_n}{z})^T$ 
11:   $r_l \leftarrow (r_1, r_2, \dots, r_{l-1}, -t, 0, 0, \dots, 0)^T$  ▷ Gleichung 4.11
12:  /* Größenreduktion */
13:  for ( $j \leftarrow l-1$  to 1)
14:     $\xi \leftarrow \lceil \frac{r_{j,l}}{r_{j,j}} \rceil$ 
15:    if ( $\xi > \eta$ )
16:       $b_l \leftarrow b_l - \xi b_j$ 
17:       $r_l \leftarrow r_l - \xi r_j$ 
18:    end if
19:  end for
20:  if ( $l > 1$ )
21:    if ( $\delta r_{l-1,l-1}^2 > r_{l,l}^2 + r_{l-1,l}^2$ ) ▷ Lovász Bedingung
22:      swap( $b_l, b_{l-1}$ )
23:       $l \leftarrow l-1$ 
24:    else
25:       $l \leftarrow l+1$ 
26:    end if
27:  else
28:     $l \leftarrow l+1$ 
29:  end if
30: end while

```

Algorithmus 5.3.3: LLL Koy-Schnorr Algorithmus für Gleitkommaarithmetik

die Matrix R' effizient zu korrigieren und auf obere Dreiecksgestalt zu bringen. Algorithmus 5.3.4 zeigt die Vorgehensweise der Korrektur einer Matrix (Aktualisierung der Zeilen r_{l-1} und r_l).

Input: Matrix $R = [r_{i,j}] \in \mathbb{R}^{d \times d}$, $l \in \{1, 2, \dots, d\}$
Output: Matrix $R = [r_{i,j}] \in \mathbb{R}^{d \times d}$ in oberer Dreiecksgestalt

- 1: $i \leftarrow l + 1$; $\tau \leftarrow |r_{l,l}| + |r_{i,i}|$
- 2: $\nu \leftarrow \tau \left(\left(\frac{r_{l,l}}{\tau} \right)^2 + \left(\frac{r_{i,l}}{\tau} \right)^2 \right)^{\frac{1}{2}} \quad \triangleright$ Zur Vermeidung von Auslöschung
- 3: $c \leftarrow \frac{r_{l,l}}{\nu}$; $s \leftarrow \frac{r_{i,l}}{\nu}$; $r_{l,l} \leftarrow \nu$; $r_{i,l} \leftarrow 0$
- 4: **for** ($k \leftarrow i$ **to** d)
- 5: $\tau \leftarrow r_{l,k}$
- 6: $r_{l,k} \leftarrow c\tau + sr_{i,k} \quad \triangleright$ Korrigiere Zeile l
- 7: $r_{i,k} \leftarrow cr_{i,k} - s\tau \quad \triangleright$ Korrigiere Zeile $l + 1$
- 8: **end for**

Algorithmus 5.3.4: Givens-Rotation. Korrektur einer fast obere Dreiecksmatrix, die in der Zeile l gestört ist.

Damit kann der **LLL**-Algorithmus für Gleitkommaarithmetik modifiziert werden und neuer Gestalt so wie in Algorithmus 5.3.5 zu sehen ist, bekommen.

5.3.3 Andere Verfahren zur Gitterbasisreduktion

Neben den oben erwähnten Verfahren zur Gitterreduktion existieren noch weitere Algorithmen, die auf verschiedene Heuristiken basieren wie etwa der von **Schnorr RSR** (Random Sampling Research) [Sch03]. Der L^2 -Algorithmus [StNg], welcher auf Basis **Cholesky-Zerlegung** der Gram-Matrix G entwickelt wurde. Dabei wird G auf exakter Arithmetik ausgeführt. Es gilt

$$G = B^T B = (QR)^T (QR) = R^T Q^T QR = R^T R.$$

<p>Input: Gitterbasis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$, $\delta \in (\frac{1}{4}, 1)$ und η</p> <p>Output: Mit Parameter δ LLL-reduzierte Basis $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$</p> <p>1: Berechne R gemäß Algorithmus 4.2.2.</p> <p>2: $l \leftarrow 2$</p> <p>3: while ($l \leq d$)</p> <p>4: <i>/*Groessenreduktion*/</i></p> <p>5: for ($j \leftarrow l - 1$ to 1)</p> <p>6: $\xi \leftarrow \lceil \frac{r_{j,l}}{r_{j,j}} \rceil$</p> <p>7: if ($\xi > \eta$) ▷ (Standard $\eta = \frac{1}{2}$)</p> <p>8: $b_l \leftarrow b_l - \xi b_j$</p> <p>9: $r_l \leftarrow r_l - \xi r_j$</p> <p>10: end if</p> <p>11: end for</p> <p>12: if ($\delta r_{l-1,l-1}^2 > r_{l,l}^2 + r_{l-1,l}^2$) ▷ Lovász Bedingung</p> <p>13: swap(b_l, b_{l-1})</p> <p>14: $l \leftarrow l - 1$</p> <p>15: Korrigiere die Matrix R gemäß Algorithmus 5.3.4 mit Eingabe l.</p> <p>16: else</p> <p>17: $l \leftarrow l + 1$</p> <p>18: end if</p> <p>19: end while</p>
--

Algorithmus 5.3.5: LLL Algorithmus für Gleitkommaarithmetik mittels Housholder-Reflexion und **Givens**-Rotation

6 Implementierung und Diskussion

In diesem Kapitel sollen die Implementierungen der bereits erwähnten Algorithmen und ihre Laufzeiten in Java dargestellt werden. Zudem wurden Benchmark-Tests für die benötigten linearen Algebraischen Operationen für die Gitterreduktion ausgeführt.

Für die Implementierung wurden einige Datentypen sowie Datenstrukturen zur Speicherung der Basismatrizen und die Approximation der Basisvektoren verwendet, die wir in Abschnitt 6.1 näher diskutieren werden. Die Datentypen unterscheiden sich in ihrer Bitlänge und ihrem Anwendungsbereich. Es werden zum Beispiel nach der Annahme in Kapitel 5 für die Basisvektoren bzw. für die Basismatrix $[b_1, b_2, \dots, b_d] \in \mathbb{Z}^{n \times d}$ ganzzahlige Datentypen benötigt.

In Abschnitt 6.2 wird ein Überblick über wichtige fertige Matrizenbibliotheken in Java, welche verschiedene Operationen in der linearen Algebra zum Inhalt haben, dargestellt. Darüber hinaus werden die erstellten Benchmarks und die gewonnenen Resultate in Graphen aufgezeigt und diskutiert. Es stellt sich heraus, dass MTJ eine bessere Performance in Java hat.

In Abschnitt 6.3 werden die Datenstrukturen für die Speicherung der Basisvektoren bzw. der Basismatrix in exakter Arithmetik sowie die approximative Darstellung dieser Vektoren in Gleitkomma-Arithmetik kurz vorgestellt. Ferner befasst sich Abschnitt 6.4 mit der Implementierung der Gitterbasisreduktionsalgorithmen. Schließlich werden die implementierten Methoden in Abschnitt 6.4.1 beschrieben.

Einen Überblick über Rechnerarchitekturen, Betriebssysteme und Compiler auf bzw. mit denen die Implementierungen und die Benchmarks erfolgreich erstellt und benutzt wurden, gewährt Tabelle 6.1.

Rechner architektur	Java Version	Hauptspeicher	Taktfrequenz	Betriebssystem
AMD 64 Bit Opteron	1.6.0_12	128 GB	16 × 2.4 GHz	Linux
Intel Pen- tium IV	1.6.0_13	1 GB	2.4 GHz	Windows XP
Genuine Intel(R) T2400	1.6.0_13	512 MB	1.86 GHz	Windows XP

Tabelle 6.1: Benutzte Hard- und Softwareumgebungen

6.1 Die benutzten Datentypen

Wie soeben erwähnt, wurden verschiedene Datentypen zur Speicherung der Basismatrizen und Approximation der Basisvektoren verwendet. In **Java** und fast allen anderen Programmiersprachen existieren zwei Standard-Datentypen, jeweils für ganzzahlige bzw. reelle Zahlen. Letztere sind vom Rechner unter Verwendung eines Gleitkomma-Datentyps darstellbar.

6.1.1 Primitive ganzzahlige Datentypen

Java stellt vier ganzzahlige Standard-Datentypen zur Verfügung: **byte**, **short**, **int**, **long**. Sie unterscheiden sich in ihrer Länge, die jeweils 1, 2, 4, und 8 Byte umfasst. Die definierte Länge ist eine wesentliche Eigenschaft von **Java**. Ganzzahlige Typen sind in **Java** immer vorzeichenbehaftet; einen Modifizierer **unsigned** wie in **C++** gibt es nicht. In **Java** bilden **long** und **short** einen eigenen Datentyp. Sie dienen nicht wie in **C++** als Modifizierer. Eine Deklaration wie **long int** ist demnach in **Java** nicht möglich. Negative Zahlen werden durch Voranstellen eines Minuszeichens gebildet.

Neben der Standard-Datentypen in **Java** bietet die Bibliothek **JScience** [DaJM] zusätzlich die Möglichkeiten ganze Zahlen durch die Klasse **Integer64**, die intern durch den Datentyp **long** zusammengebaut ist, zu speichern.

Für diesen Datentyp wurde ein Benchmark erstellt welches zeigt, dass Inte-

ger64 langsamer als der Standard Datentyp long ist (siehe Tab. 6.3).

type	Kleinster Wert	Größter Wert	Speicherplatz
byte	-2^7	$2^7 - 1$	1 Byte (8 Bit)
short	-2^{15}	$2^{15} - 1$	2 Byte (16 Bit)
int	-2^{31}	$2^{31} - 1$	4 Byte (32 Bit)
long	-2^{63}	$2^{63} - 1$	8 Byte (64 Bit)
Integer64	-2^{63}	$2^{63} - 1$	8 Byte (64 Bit)

Tabelle 6.2: Ganzzahlige Datentypen in java

Operation	Addition	Multiplika- tion	Substrak- tion
long	2.355 ns	2.355 ns	1.533 ns
Integer64	29.66 ns	23.34 ns	32.55 ns

Operation	Addition	Multiplika- tion	Substra- ktion
long	18.16 ns	20.82 ns	20.5 ns
Integer64	72.06 ns	79.57 ns	69.21 ns

Operation	Addition	Multiplika- tion	Subtra- ktion
long	29.53 ns	34.84 ns	31.71 ns
Integer64	100.0 ns	107.8 ns	101.8 ns

Tabelle 6.3: Gemittelte Zeiten aus $2 \cdot 10^8$ Additionen und Multiplikationen ganzer Zahlen. Links: 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM** 2.4 GHz); Betriebssystem: Linux. Rechts: 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**); Betriebssystem : **MacBook**. Mitte: 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz); Betriebssystem: Windows

6.1.2 Datentypen für große Ganzzahlen

Da die Basisvektoren eines Gitters meistens sehr große Einträge haben, müssen sie exakt gespeichert werden, da das Gitter sonst während der Transformation verändert wird. Dies gilt insbesondere für ganzzahlige Gitterbasis eines Gitters $L \subseteq \mathbb{Z}^n$.

Ein Nachteil bei diesen Datentypen ist, dass sie langsam bei der Berechnung sind, da sie nicht direkt vom Prozessor unterstützt und emuliert werden müssen.

Aus diesem Grund besteht das Interesse fertige implementierte Bibliotheken bzw. Klassen, die sich mit Langzahlarithmetik beschäftigen, zu benutzen. Da die feste Länge der primitiven Datentypen für die diversen Berechnungen auf den Gittern nicht ausreichend ist.

Für solche Anwendungen bietet Java die Klasse **BigInteger** an, mit der es möglich ist, beliebig große Zahlen (zumindest solange der Hauptspeicher reicht) anzulegen, zu verwalten und damit zu rechnen.

Ein **BigInteger**-Objekt wird intern wie der primitive Datentyp **int** im Zweierkomplement dargestellt. Allerdings wird das Objekt immer so weit ausgedehnt, wie die entsprechenden Ergebnisse Platz benötigen (infinite word size). Bei Operationen auf zwei **BigInteger**-Objekten mit unterschiedlichen Bitlängen wird das Kleinere dem Größeren durch Replikation des Vorzeichen-Bits angepasst. Nachteilig wirkt sich dabei aus, dass Java kein Operator Overloading anbietet. Hinzu kommt der Overhead der jeweiligen Größenanpassung bei einzelnen Rechenschritten, wobei diese zu Performanceverlusten führt. Da die Größe des Datentyps bei Bedarf immer ausgedehnt wird, sind außerdem einige Operationen (z.B. Bitverschiebung ohne Vorzeichenbeachtung) nicht übertragbar.

Kandidaten für große Ganzzahlen-Datentyp waren noch vier Klassen in vier verschiedenen Bibliotheken. Das Software Projekt Java Algebra System (JAS) [JAS] beschäftigt sich mit der Computer Algebra und ist an der Universität Mannheim entwickelt worden. Es bietet die Klasse **jas.arith.BigInteger**, die intern mit **BigInteger** aus der **Java.math** Objekten zusammengebaut ist. Zusätzlich bietet die Bibliothek **JScience** [DaJM] die Klasse **LargeInteger** an, die intern durch ein Longarray (**long[]_words**) für die Ziffern und ein **boolean** Objekt für das Vorzeichen zusammengebaut ist. Ferner ist **Apint** eine im Projekt Apfloat [Tomm] in der aktuellen Version 1.5.1, und als fünfter Kandidat ist die Klasse **FPBigInt**, die an der TU-Darmstadt entwickelt und auf Laufzeit optimiert wurde.

Ein Benchmark für die fünf Klassen wurde auf den Rechnern, die in Tabelle 6.1 bereits angegeben sind, durchgeführt. Die Ergebnisse, die in den Tabellen

B.1 bis B.9 im Anhang B auf Seite 89 stimmten, aufgrund der verwendeten Operationen (Subtraktion und Multiplikation) bei der Größenreduktion im Algorithmus 5.1.2, für den Datentyp **BigInteger** aus der **java.math** Bibliothek überein.

6.1.3 Primitive Gleitkomma-Datentypen

Zur approximativen Darstellung von reellen Zahlen bzw. für Gleitkommazahlen einfacher und erhöhter Genauigkeit bietet Java für das Speichern der **Gram-Schmidt** Koeffizienten bzw. der Elemente der Matrix **R** aus der **QR**-Zerlegung die Datentypen **float** und **double**. Die Datentypen sind im **IEEE – 754**-Standard beschrieben und haben eine Länge von 4 Byte für **float** und 8 Byte für **double** an. Gleitkommaliterale können, so wie es im Kapitel 2.2.1 beschrieben wurde, einen Vorkommateil und einen Nachkommateil besitzen, die durch einen Dezimalpunkt (kein Komma) getrennt sind. Neben den Datentyp **double** ist die Klasse **Float64** in der Bibliothek [DaJM] zu finden, für die ebenso ein Benchmark erstellt wurde. In der Tabelle 6.4 ist ohne Weiteres zu erkennen, dass die Berechnung auf dem Standard **double** Typ schneller ist, als auf dem Datentyp **Float64**.

Operation	Addition	Multiplikation	Subtraktion
double	13.55 ns	9.755 ns	13.86 ns
Float64	15.61 ns	16.61 ns	15.31 ns

Operation	Addition	Multiplikation	Subtraktion
double	22.42 ns	22.14 ns	21.92 ns
Float64	28.39 ns	28.12 ns	27.81 ns

Operation	Addition	Multiplikation	Subtraktion
double	24.56 ns	24.68 ns	16.68 ns
Float64	73.37 ns	75.15 ns	74.84

Tabelle 6.4: Gemittelte Zeiten aus $2 \cdot 10^8$ Additionen und Multiplikationen von Gleitkommazahlen. Links: 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM** 2.4 GHz); Betriebssystem: Linux. Rechts: 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**); Betriebssystem: **MacBook**. Mitte: 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz); Betriebssystem: Windows

type	Wertebereich	Speicherplatz
float	$-3.40282347 \cdot 10^{38}$ bis $3.40282347 \cdot 10^{38}$ (ergibt 7 signifikanten Dezimalziffern Genauigkeit)	4 Byte (32 Bit)
double	$-1.79769313486231570 \cdot 10^{308}$ bis $1.79769313486231570 \cdot 10^{308}$ (ergibt 16 signifikanten Dezimalziffern Genauigkeit)	8 Byte (64 Bit)

Tabelle 6.5: Eigenschaften und Wertebereiche von Gleitkomma-Datentypen in **Java**

	Ganzzahlen	Gleitkommazahlen
Java	BigInteger	BigDecimal
Java Algebra System	jas.arith.BigInteger	jas.arith.BigDecimal
JScience	LargeInteger	FloatingPoint
Apfloat	Apint	Apfloat
FPBigInt	FPBigInt	-

Tabelle 6.6: Betrachtete große Ganzzahl- und Gleitkomma-Datentypen

6.1.4 Datentypen für große Gleitkommazahlen

Da aus Tabelle 6.5 ersichtlich ist, dass die Anzahl der Präzisionsbits für die Mantisse auf höchstens 53 Bits beschränkt ist und es dadurch bei der Gitterbasisreduktion zu erheblichen Stabilitätsverlusten kommt, besteht ein großes Interesse daran, in Java implementierte Klassen mit einstellbarer Genauigkeit anzuschauen, um diese Instabilität zu vermeiden und schließlich auf Laufzeit zu testen.

In Java gibt es bereits die Klasse **BigDecimal**, deren Objekt sich aus zwei **integer** Zahlen für die Mantisse und den Exponenten zusammensetzt. Neben dieser Klasse wurden soeben im Abschnitt 6.1.2 auf Seite 51 zwei anderen Bibliotheken **JScience** [DaJM] und **Apfloat** [Tomm] betrachtet, die die Klassen **FloatingPoint** und **Apfloat** anbieten. In den Tabellen C.1 bis C.9 im Anhang C auf Seite 93 wurde ein Benchmark für alle drei Klassen auf den Rechnern, die in der Tabelle 6.1 angegeben sind, erstellt. Auf allen Prozessoren zeigten die Ergebnisse in den Tabellen C.1 bis C.9, dass die Berechnungen mit

großen Gleitkommazahlen in **BigDecimal** mit wachsender Bitlänge und Precision langsamer sind als mit **FloatingPoint** und **Apfloat**. Grund dafür ist, dass beispielsweise die Multiplikation in der Klasse **FloatingPoint** mit dem parallelen **Karatsuba**-Algorithmus durchgeführt wird und **Apfloat** ab einer gewissen Stellenzahl von der Schulmethode auf die parallele Multiplikation mittels Fast-Fourier-Transformation umschaltet.

6.2 Benutzte Bibliotheken

In diesem Abschnitt wird ein Überblick über wichtige fertige Matrizenbibliotheken in Java, welche verschiedene Operationen in der linearen Algebra zum Inhalt haben, dargestellt. Bei der Arbeit stand die Performance im Vordergrund, welche in den folgenden Operationen, die wiederum bei der Gitterbasisreduktion von Bestandteil sind, getestet wurden; Berechnung der Multiplikation, des Transponierens, der Inversion, der **QR**-Zerlegung, **Cholesky**-Zerlegung und der Determinantenberechnung von Matrizen.

Zunächst werden die benutzten Bibliotheken, die in Java implementiert sind, aufgezählt und vorgestellt. Danach werden die erstellten Benchmarks und die gewonnenen Resultate aufgezeigt.

Jama[JM]: Das Paket ist eine Entwicklung von MathWorks und der Bundesbehörde der Vereinigten Staaten [NIST] (National Institute of Standards and Technology)¹ und liegt als freie Referenzimplementierung vor, die Sun später als Standard-Bibliothek anbieten soll. Es bietet die Operationen auf dicht besetzbaren Matrizen an, die intern auf einem zweidimensionalen Array vom Gleitkomma-Datentyp **double** gespeichert werden.

Jampack [JP]: Zusätzlich zu der Matrix-Klasse als Teil von Jama haben das NIST und die Universität von Maryland das Paket Jampack als alternativen Ansatz für Matrixberechnung entwickelt. Die interne Berechnung auf den Objekten wird ebenfalls auf dem zweidimensionalen **double**-Array ausgeführt.

¹Das NIST gehört zur technologischen Administration des Handelsministeriums und ist für Standardisierungsprozesse zuständig.

JScience: wurde vom Author Jean-Marie Dautelle [DaJM] entwickelt. Die interne Berechnung auf den Objekten erfolgt auf dem zweidimensionalen **Float64**-Array, welcher bereits diskutiert wurde. Das Paket ist auf Multiprozessorsystem optimiert; zum Beispiel für paralleles Multiplizieren von Matrizen.

Colt [Colt]: ist von der European Organization for Nuclear Research für diverse numerische Berechnungen entwickelt worden.

Matrix Toolkit Java: [MTJ] Das Paket bietet verschiedene Datenstrukturen für dünn bzw. dicht besetzbaren Matrizen an. Diese Bibliothek enthält Funktionen, die auf Basis der BLAS (Basic linear Algebra Subprograms [BLAS]) und Lapack (Linear Algebra Package [LAPAK]) zurückzuführen sind.

CommonMath [CoMath], Java Native Library [JNL], JSci [JSci] und JMAT [JMAT] sind ebenfalls in Java implementiert und bieten analog zu den oben genannten Paketen verschiedene Datenstrukturen für Matrizen an.

6.2.1 Benchmark Beschreibung

Im diesem Abschnitt werden die verschiedenen Bibliotheken nach ihren Laufzeiten getestet. Für die Analyse wurden für jede Operation 10000 zufällige Matrizen generiert, deren Elemente im Intervall $[-200, 200]$ liegen.

Alle Performance-Experimente wurden auf dem Mehrprozessor-System mit 64-Bit Wortbreite Quad-Core AMD Opteron(tm) Processor 8356 mit 128 GB RAM. Die Prozessoren sind mit 2.3 GHz getaktet. Der Universitätsrechner nutzt als Betriebssystem Debian GNU/Linux in der Version 2.6.26-1.

Die vorgelegten Benchmarks sind auf dem Gleitkomma-Datentyp **double** durchgeführt worden. Resultate werden in den Graphen dargestellt, welche die gemessene Performance in **Mflops/s** angeben.

Für die Zeitmessung wurde die Klasse **ThreadMXBean** aus der Bibliothek **Java.lang.management** importiert. Diese Klasse ist in Java 5.0 eingeführt worden und bietet die Methode **getCurrentThreadCpuTime()**, die die benötigte Zeit vom Prozessor liefert. Dies wird damit begründet, dass Messungen mit der Wanduhr (Wall Clock Time) zu ungenauen Resultaten führen würde,

da der Prozessor aufgrund anderer Aktivitäten belastet wäre. Die tatsächliche CPU-Zeit und User-Zeit wird damit für jeden Thread geliefert, indem die Methode vor und nach einer beliebigen Operation aufgerufen und die Differenz der gelieferten Zeiten ausgegeben wird. Diese Zeitmessungen werden nicht von anderen Aktivitäten im System beeinflusst.

6.2.1.1 Matrix-Multiplikation

Abbildung 6.1 stellt die Laufzeit für die Matrixmultiplikation, $A \cdot B = C = (c_{ij})_{i=1\dots m, j=1\dots p}$ und $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$ für $A = (a_{ij})_{i=1\dots m, j=1\dots n}$ und $B = (b_{ij})_{i=1\dots n, j=1\dots p}$, dar. Bibliotheken, in denen sie implementiert wurden, waren **Jama**, **Jampack**, **JScience**, **Colt**, **CommonMath**, **JNL**, **MTJ**, **JSci** und **JMAT**. Die Ergebnisse aus der Abbildung 6.1 zeigen, dass **Jama**, **JScience** und **MTJ** nahezu gleich bezgl. ihrer Laufzeit sind.

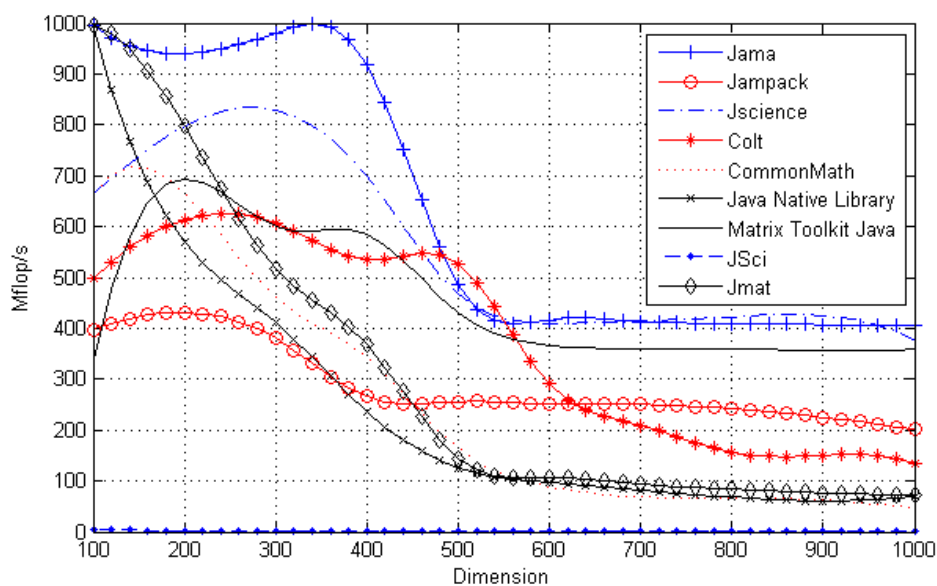


Abbildung 6.1: Graph für Matrixmultiplikation-Benchmarks

6.2.1.2 Determinantenberechnung

In den Bibliotheken **Jama**, **Jampack**, **JScience**, **Colt**, **CommonMath**, **JNL**, **MTJ**, **JSci** und **JMAT** erfolgte die Implementierung der Determinantenberechnung. Die Berechnung der Determinante einer Matrix B erfolgt in den meisten Fällen mit der **LU-Zerlegung** ($\det(B) = \det(LU) = \det(L) \det(U) = \det(U)$). In Abbildung 6.2 ist zu sehen, dass die Bibliothek **MTJ** einen klaren Vorteil gegenüber der anderen hat und insgesamt den Vorzug verdient.

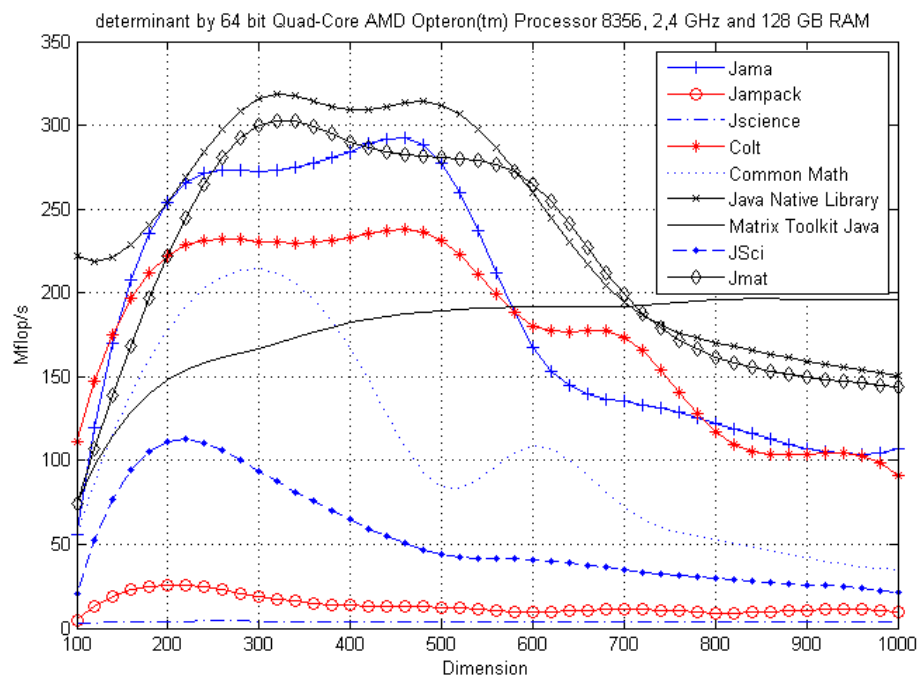


Abbildung 6.2: Graph für Determinantenbenchmarks

6.2.1.3 Die Matrixinvertierung

Abbildung 6.3 zeigt das Verhältnis der Berechnung der Inversen einer quadratischen Matrix A^{-1} , für die die Gleichung

$$A \cdot A^{-1} = A^{-1} \cdot A = E \quad (6.1)$$

erfüllt ist. Die betrachteten Bibliotheken, welche die Inverse einer Matrix implementierten waren **Jama**, **Jampack**, **JScience**, **Colt**, **CommonMath**, **JNL**, **MTJ**, **JSci** und **JMAT**. Erneut lässt sich feststellen, dass die Bibliothek **MTJ** einen Vorteil gegenüber anderen Bibliotheken hat und somit vorzuziehen ist.

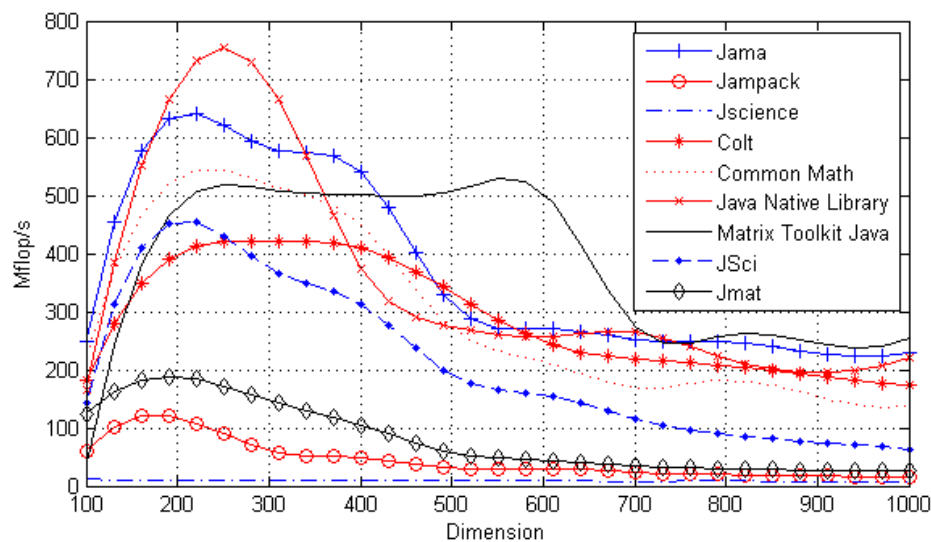


Abbildung 6.3: Graph für Matrixinversion-Benchmarks

6.2.1.4 Die QR-Zerlegung

In Kapitel 5.3.2 wurde bereits der Vorteil der **QR-Zerlegung** einer Matrix aufgezeigt. Für die Berechnung dieser Zerlegung wurde auch ein Benchmark

erstellt und ist in der Abbildung 6.4 zu sehen. **JNL** hat die beste Laufzeit, jedoch zeigten Tests, dass die Reihenfolge der Spalten in **R** nicht korrekt war. Aus diesem Grund wurde **MTJ** vorgezogen.

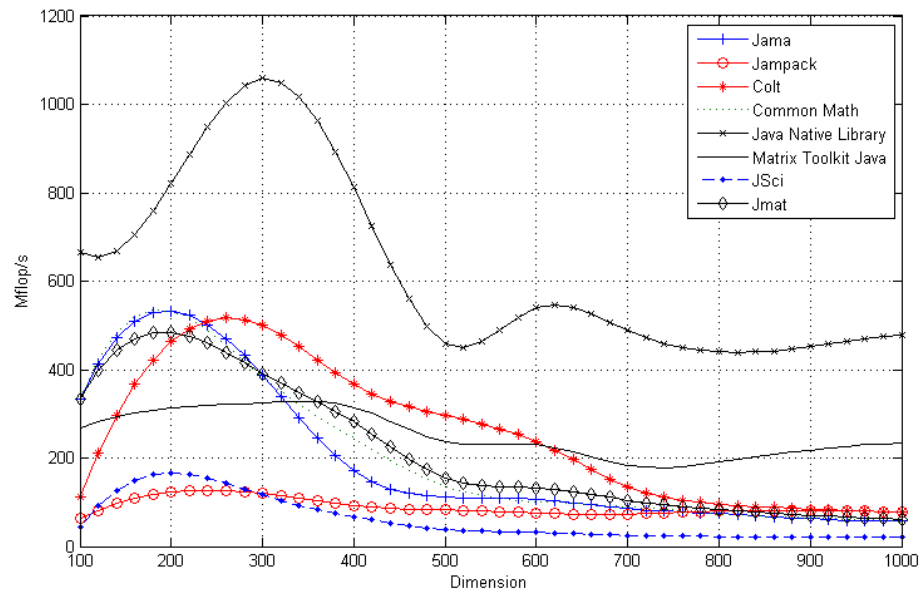


Abbildung 6.4: Graph für QR-Zerlegung-Benchmarks

6.2.1.5 Die Cholesky-Zerlegung

Für die **Cholesky-Zerlegung** wurden die Bibliotheken **Jama**, **JamPack**, **Colt**, **JNL**, **MTJ**, **JSci** getestet.

Die Ergebnisse sind in Abbildung 6.5 und 6.6 dargestellt. In Abbildung 6.5 ist zu sehen, dass die Zerlegung in **JSci** langsamer ist als **JamPack**. Diesen Paket haben wir wiederum mit den übrigen Paketen (**Jama**, **JamPack**, **JNL**, **Colt** sowie **MTJ**) und ist in Abbildung 6.5 **JamPack** zusehen. Wir bekamen als Ergebnis, dass die Zerlegung in **MTJ** am schnellsten ist.

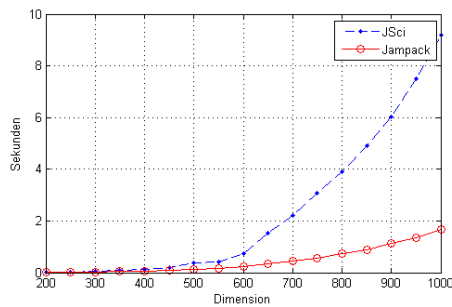


Abb. 6.5: Graph für Cholesky-Zerlegung-Benchmarks der Paketen Jsci und Jampack

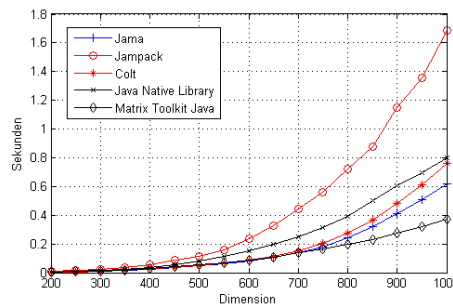


Abb. 6.6: Benchmarks der Paketen Jampack, Jama, Colt, JNL sowie MTJ

6.2.2 Resultate

Insgesamt ist auf Grundlage verschiedener Abbildungsergebnisse (6.1, 6.2, 6.3, 6.4, 6.6) die Bibliothek **MTJ** für die spätere Verwendung in der Gitterbasisreduktion auszuwählen.

6.3 Matrizendarstellung

Es ist von einigen Datenstrukturen der **MTJ** Gebrauch gemacht worden, wobei diese Bibliothek nicht in der Lage ist eine exakte Speicherung der Basisvektoren als Matrizen zu gewähren. Folge dessen ist die Anwendung eigener Datentypen für ganzzahlige Matrizen; **IntegerMatrix**, **LongMatrix** und **BigIntegerMatrix**. Zwar wurden zusätzlich andere Datentypen für die Darstellung ganzzahliger Vektoren wie **IntegerVector** und **BigIntegerVector** angewandt, jedoch ergaben Ergebnisse bei der Skalarproduktberechnungen in Orthogonalisierungsabschnitt 5.3.1 und bei der Größenreduktion des **SE**-Algorithmus 5.3.2 im Schritt 20, dass sie Overheads für sämtliche Objektinstanzen produzieren. Diese Berechnungen wurden somit direkt auf dem internen zweidimensionalen Array der zugehörigen Datenstruktur ausgeführt.

6.4 Implementierung der Gitterbasisreduktionsalgorithmen

In der Implementierung der Algorithmen zur Gitterbasisreduktion wurde ein Klassen-Template **LatticeReduction** $\langle \mathbf{ZZ}, \mathbf{FT} \rangle$ (Superklasse) erstellt, damit die programmierten Verfahren zur Gitterbasisreduktion auch mit unterschiedlichen Gleitkomma-Datentypen verwendet werden können. **ZZ** steht für die Deklaration ganzzahliger Datenstrukturen bzw. Deklaration der Basismatrix vom Typ **IntegerMatrix**, **LongMatrix** sowie **BigIntegerMatrix**. Weiterhin steht **FT** für die Deklaration von Gleitkomma-Datenstrukturen und für die Deklaration der approximativen Darstellung der Basismatrix vom Typ **DoubleMatrix** sowie **BigDecimalMatrix**. Im Vergleich zu C++ war in Java zu beachten, dass elementare Datentypen als Typparameter nicht erlaubt sind und Typparameter nicht für statische Klassen-Attribute genutzt werden können. Mehr über Generics in Java kann in [BrGi] nachgelesen werden.

In den gesamten Implementierungen repräsentieren die Zeilenvektoren einer Matrix und nicht die Spaltenvektoren die Basisvektoren eines Gitters. Demnach ist zu beachten, dass bei der Programmierung die meisten Matrizen bzw. Vektoren transponiert betrachtet werden müssen.

Ein UML-Klassendiagramm [Forb] zur graphischen Darstellung der implementierten Klassen, Schnittstellen, sowie deren Beziehungen wurde erstellt und ist im Anhang D Abbildung D.1 dargestellt.

6.4.1 Beschreibung der Methoden

In diesem Abschnitt werden die Hauptmethoden, die zur Gitterbasisreduktion dienen, kurz beschrieben.

public abstract void fpLLL_SE(double delta, double eta);: Diese Methode entspricht in ihrer Funktionsweise dem **SE**-Algorithmus 5.3.2 auf Seite 43. Der Parameter **delta** = δ (Reduktionsparameter) hat als Standardwert 0.99, **eta** = $\frac{1}{2}$.

public abstract void LLL_HO(double delta, double eta); : Diese Methode entspricht in ihrer Funktionsweise dem Algorithmus 5.3.3 auf Seite 45. Die Orthogonalisierung erfolgt iterativ innerhalb der **While-Schleife** mittels Householder. Der Parameter **delta** = δ (Reduktionsparameter) hat als Standardwert 0.99, **eta** = $\frac{1}{2}$.

public abstract void LLL_HO_MTJ(double delta, double eta); : Diese Methode entspricht in ihrer Funktionsweise dem Algorithmus 5.3.5 auf Seite 47. Die **QR**-Zerlegung erfolgt mit der Householder-Verfahren und mittels der Bibliothek **MTJ**. Dies geschieht vor der **While-Schleife**. Anschließend werden der Zeilen der Faktor R mittels Givens-Rotation korrigiert, falls die Lovász-Bedingung innerhalb der **While-Schleife** erfüllt ist. Der Parameter **delta** = δ (Reduktionsparameter) hat als Standardwert 0.99, **eta** = $\frac{1}{2}$.

public abstract void LLL_HO_Givens(double delta, double eta); : Diese Methode entspricht in ihrer Funktionsweise dem Algorithmus 5.3.5 auf Seite 47. Die **QR**-Zerlegung erfolgt mit der Householder-Verfahren und wurde von uns implementiert. Dies geschieht vor der **While-Schleife**. Anschließend werden der Zeilen der Faktor R mittels Givens-Rotation korrigiert, falls die Lovász-Bedingung innerhalb der **While-Schleife** erfüllt ist. Der Parameter **delta** = δ (Reduktionsparameter) hat als Standardwert 0.99, **eta** = $\frac{1}{2}$.

public void LLL_cholesky_Givens(final double delta, double eta); : Diese Methode entspricht in ihrer Funktionsweise dem Algorithmus 5.3.5 auf Seite 47. Die Berechnung der Faktor R erfolgt mit der Cholesky-Zerlegung aus der Bibliothek **MTJ**. Dies geschieht vor der **While-Schleife**. Anschließend werden der Zeilen der Faktor R mittels Givens-Rotation korrigiert, falls die Lovász-Bedingung innerhalb der **While-Schleife** erfüllt ist. Der Parameter **delta** = δ (Reduktionsparameter) hat als Standardwert 0.99, **eta** = $\frac{1}{2}$.

7 Experimentelle Resultate

In diesem Kapitel werden die Laufzeiten der in Kapitel 5 implementierten Algorithmen anhand von Graphen vorgestellt.

Ferner werden die Laufzeiten mit dem von **Victor Shoup** in [NTL] implementierten **Schnorr** und **Euchner** Algorithmus verglichen. Die im Quellcode frei verfügbare, auf Geschwindigkeit optimierte und portable C++ Bibliothek **NTL** (Number Theoretic Library) ist in der aktuellen Version 5.5.1 unter <http://www.shoup.net/ntl> zu finden.

In NTL [NTL] gibt es verschiedene Varianten für die Implementierung des **SE**-Algorithmus, die sich in der Darstellung der Approximation vor der Basismatrix sowie der Anzahl der Mantissenbits unterscheiden. Von diesen Varianten wurden die Klassen **LLL_FP**, die den Standard Datentyp **double** mit 53 Bits Genauigkeit verwendet, wohingegen **LLL_QP** den Datentyp **quad_float**¹ benutzt, für den Vergleich ausgewählt.

Im Rahmen der durchgeführten Experimente haben wir die modularen Gitter der Dimension $200 \leq d \leq 800$ verwendet, welche besonders für ihre beweisbare Schwierigkeit in der Challenge [BLRS] verfügbar gestellt wurden.

Für $50 \leq n \in \mathbb{N}$, $c_1, c_2 \in \mathbb{R}_{>0}$, so dass

$$c_1 \geq 50, c_2 \leq \ln(2) - \frac{\ln(2)}{50 \ln(50)},$$

und mit

$$m = \lfloor c_1 n \ln(n) \rfloor, q = \lfloor n^{c_2} \rfloor,$$

haben die Matrizen $B_m \in \mathbb{Z}^{m \times m}$ dieser Gitter folgende Form:

$$B_m = \begin{bmatrix} E_{m-n} & N \\ Z & q \cdot E_n \end{bmatrix},$$

¹Ein in [NTL] implementierter Gleitkomma-Datentyp, der 105 Bits für die Mantisse und 11 Bits für den Exponenten besitzt.

wobei E_{m-n} für die $(m-n) \times (m-n)$ Einheitsmatrix, N für die $(m-n) \times n$ -Nullmatrix, $Z \in \mathbb{Z}_q^{n \times (m-n)}$ ² und E_n für die $n \times n$ Einheitsmatrix steht³.

Die Basen B_m für $m = 200, 300, \dots, 800$ wurden zum einen mit dem **SE**-Algorithmus, und zum anderen mit dem **Koy-Schnorr**-Algorithmus reduziert (Abbildung 7.1). Im **Koy-Schnorr**-Algorithmus fand die **QR**-Zerlegung zum einen mit der Verwendung der Bibliothek **MTJ** (In der Abbildung bezeichnen wir das mit Householder und Givens mit MTJ) statt, zum anderen wurde sie von uns implementiert (In der Abbildung bezeichnen wir das mit Householder und Givens). Die Algorithmen wurden in Java implementiert. Für die Reduktion mit den beiden Algorithmen haben wir den Datentyp **BigInteger** für die Darstellung der Basismatrix sowie **double** für die Approximation der Matrix gewählt. Wir wählten $\delta = 0.99$. Mit $\delta = 0.99$ haben wir **double** Typen für die Gleitkomma-Arithmetik verwendet. Die Durchführung der Experimente erfolgte auf dem Mehrprozessor-System mit 64-Bit Wortbreite Quad-Core **AMD** Opteron(tm) Processor 8356 mit 128 GB **RAM**. Die Prozessoren sind mit 2.3 GHz getaktet.

Abbildung 7.1 links zeigt der Vergleich des implementierten **SE**- sowie **Koy-Schnorr** Algorithmus. Es ist deutlich zu sehen, dass die **Koy-Schnorr**-Variante besser als die SE-Variante abschneidet. Rechts im Bild (Abbildung 7.1) wird diese Variante weiter mit der in **NTL** implementierten **SE**-Algorithmus (**LLL_FP**, in der Abbildung wird das mit **NTL Schnorr Euchner** bezeichnet) verglichen. Eine deutliche Beschleunigung konnte nicht erreicht werden. Eine weitere Variante wurde mittels der **Cholesky-Zerlegung** und einer **Givens** Korrektur implementiert (Bezeichnung in der Abbildung (Cholesky und Givens)). Diese Variante zeigte eine größere Beschleunigung, jedoch aber nicht gegenüber dem **LLL_FP**.

Getestet auf dieselben Basen wurden die Algorithmen auch intern in **Java** auf den verschiedenen Datentypen **int**, **long**, **BigInteger**, in der Variante der **Cholesky**-Zerlegung, welche in Abbildung 7.2 dargestellt ist.

²Mit $\mathbb{Z}_q = \{0, 1, \dots, q-1\}$

³Gitter dieser Form bilden eine spezielle Klasse von Gittern und haben eine ähnliche Struktur wie die von Coppersmith und Shamir für das **NTRU**-Kryptosystem entdeckten Gitter [CoSh]. Bezeichnet werden sie als convolution modular lattices

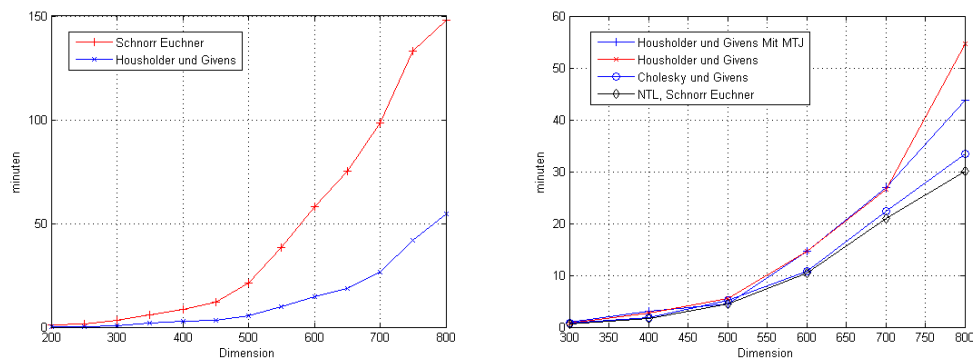


Abbildung 7.1: Laufzeiten der Gitterreduktionsalgorithmen mit unterschiedlichen Varianten mit $\delta = 0.99$.

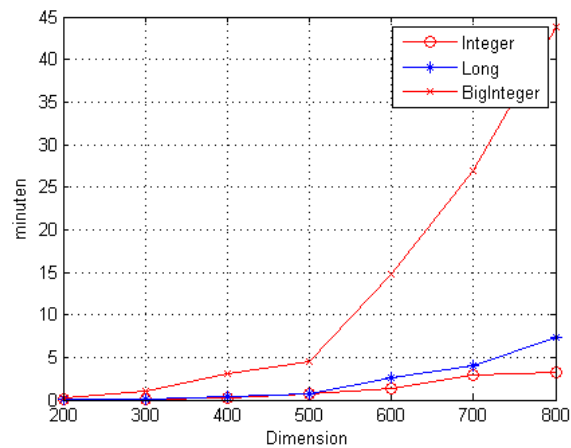


Abbildung 7.2: Laufzeiten der Gitterreduktionsalgorithmen mit der **Cholesky**-Zerlegung für die Datentypen **int**, **long**, **BigInteger** für $\delta = 0.99$

Zufälliges Gitter ist ein komplexer Begriff ([Ajtai06] [StNg06] [GoMa]). Für unsere Experimente haben wir zufälligen Gitter, die von **Goldstein** und **Meyer** [GoMa] entdeckt wurden, erzeugt. Die d -dimensionalen Gitter $L_{d,p}$ haben

folgende Form:

$$B_p^d = \begin{bmatrix} p & x_1 & x_2 & \dots & x_{d-1} \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix},$$

wobei p eine große Primzahl ist und $x_i \in \{0, \dots, p-1\}$ für $1 \leq i \leq d-1$.

Da die Primzahl p meistens sehr groß ist, benötigt man eine höhere Genauigkeit bei der Reduktion. Für die Reduktion braucht man natürlich Gleitkommazahlen mit beliebig einstellbarer Genauigkeit. Dies verlangsamt die Ausführungszeiten für der Operationen deutlich.

Somit haben wir für die Implementierung der Reduktions-Algorithmen eine Genauigkeit von 113-Bit für den **SE**-Algorithmus und den Datentyp **BigDecimal** für die Approximation der Basismatrix verwendet. Schließlich wurde die Ausführungszeiten dieser Implementierung mit dem **LLL_QP** verglichen. Im Zusammenhang betrachtet erfolgt der Vergleich auf den von **Goldstein** und **Meyer** betrachtete Gitter der Dimension 50. Jedoch zeigte es zur Implementierungen von **NTL** eine zu hohe Laufzeit und konnte nicht effizient in der Gitterbasisreduktion verwendet werden.

Aus diesem Vergleich ergibt sich, dass die Gitterbasisreduktion in Java keine Beschleunigung der Laufzeiten herbeiführt. Wohingegen der in **NTL** implementierte Algorithmus eine deutlich schnellere Geschwindigkeit nachweisen konnte. Auch der anschließende Umstieg auf die Varianten mit Householder und Givens (Abbildung 7.1) sowie die Cholesky-Zerlegung und Givens brachten keine deutliche Beschleunigung.

Nachdem wir die experimentelle Ergebnisse und Resultate in ausführlicher Form dargestellt haben, soll in Kapitel 8 eine Zusammenfassung folgen, die den Abschluss der vorliegenden Arbeit bildet.

8 Zusammenfassung

Betrachtet wurden im Rahmen dieser Arbeit die Laufzeiten und die Effizienz verschiedener in **Java** implementierter Algorithmen für Gitterbasisreduktion. Anschließend wurden diese mit der Prozedur **LLL_FP** aus der **C++** Bibliothek **NTL** verglichen, in der Hoffnung, dass es eine bessere Variante gibt, die schneller als **LLL_FP** ist. Im Ergebnis war dies bedauerlicherweise trotz Optimierungen am Code und Verwendung einer schnellen Bibliothek Names **MTJ** zu verneinen.

Ein Grund für die schlechtere Performance gegenüber **C++** ist sicherlich die Verwendung der Klasse **BigInteger**, da die Berechnungen in der Phase der Größenreduktion in der **Schnorr – Euchner** Prozedur auf Objekten dieser Klasse durchgeführt werden. Der Datentyp **ZZ** für Langganzzahlarithmetik der von **NTL** verwendeten Bibliothek **GMP**¹ [GMP] ist hier klar im Vorteil, da dessen kritische Routinen zum Teil von Hand in Assembler-Coder optimiert sind.

Ein weiteres Problem für Java sind **Arrayzugriffe**, die immer auf Bereichsüberschreitungen überprüft werden, und die komfortable **Garbage Collection**. Sie ist rechenaufwändiger als eine effiziente Zerstörung von Objekten von Hand. Dies führt vor allem bei Matrixoperationen zu erheblicher Verlangsamung. Ein Studie für die Parallelisierung für Gitterbasisreduktion in Java erfolgte von Dagdelen [DaOe] und zeigte, dass die Laufzeit der Implementierung des **SE** aus der **NTL** geringer als die parallelisierte **SE** ist.

¹<http://gmplib.org/>

Abbildungsverzeichnis

3.1	Beispiel für ein 2-dimensionales Gitter mit den verschiedenen Basen B_1 und B_2 aus Beispiel 3.1.1	16
3.2	Sukzessive Minima $\lambda_1 = \ b_1\ $, $\lambda_2 = \ b_2\ $	17
3.3	Beispiel für die Parallelepipede beider Gitterbasen aus Beispiel 3.1.1	18
4.1	Geometrische Interpretation des Gram-Schmidt-Verfahrens	26
4.2	Householder-Reflexion	28
4.3	Rotation von v bei einem Winkel α	32
5.1	Bereich der reduzierten Basen b_1, b_2 mit $0 \leq \mu_{2,1} \leq \frac{1}{2}$	37
6.1	Graph für Matrixmultiplikation-Benchmarks	57
6.2	Graph für Determinantenbenchmarks	58
6.3	Graph für Matrixinversion-Benchmarks	59
6.4	Graph für QR-Zerlegung-Benchmarks	60
6.5	Graph für Cholesky-Zerlegung-Benchmarks der Paketen Jsci und Jampack	61
6.6	Benchmarks der Paketen Jampack, Jama, Colt, JNL sowie MTJ	61
7.1	Laufzeiten der Gitterreduktionsalgorithmen mit unterschiedlichen Varianten mit $\delta = 0.99$	67

7.2	Laufzeiten der Gitterreduktionsalgorithmen mit der Cholesky - Zerlegung für die Datentypen int , long , BigInteger für $\delta = 0.99$	67
D.1	Teil-Klassendiagramm des Programm	98

Tabellenverzeichnis

4.1	Aufwand und numerische Güte von Orthogonalisierungsverfahren	25
6.1	Benutzte Hard- und Softwareumgebungen	50
6.2	Ganzzahlige Datentypen in java	51
6.3	Gemittelte Zeiten aus $2 \cdot 10^8$ Additionen und Multiplikationen ganzer Zahlen. Links: 64 bit AMD Prozessor (16 Kerne, 128 GB RAM 2.4 GHz); Betriebssystem: Linux. Rechts: 32 bit Genuine Intel(R) Prozessor (512 MB RAM); Betriebssystem : MacBook . Mitte: 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz); Betriebssystem: Windows	51
6.4	Gemittelte Zeiten aus $2 \cdot 10^8$ Additionen und Multiplikationen von Gleitkommazahlen. Links: 64 bit AMD Prozessor (16 Kerne, 128 GB RAM 2.4 GHz); Betriebssystem: Linux. Rechts: 32 bit Genuine Intel(R) Prozessor (512 MB RAM); Betriebssystem : MacBook . Mitte: 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz); Betriebssystem: Windows	53
6.5	Eigenschaften und Wertebereiche von Gleitkomma-Datentypen in Java	54
6.6	Betrachtete große Ganzzahl- und Gleitkomma-Datentypen	54
B.1	Addition großer ganzer Zahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM). Betriebssystem : Windows	89
B.2	Multiplikation großer ganzer Zahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM). Betriebssystem : Windows	90

B.3	Subtraktion großer ganzer Zahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM). Betriebssystem : Windows	90
B.4	Addition großer ganzer Zahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM , 2.4 GHz). Betriebssystem: Linux	90
B.5	Multiplikation großer ganzer Zahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM 2.4 GHz). Betriebssystem: Linux	90
B.6	Subtraktion großer ganzer Zahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM , 2.4 GHz). Betriebssystem: Linux	91
B.7	Addition großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	91
B.8	Multiplikation großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	91
B.9	Subtraktion großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	91
C.1	Addition großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM , 1.83 GHz). Betriebssystem : Windows	93
C.2	Subtraktion großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM , 1.83 GHz). Betriebssystem : Windows	94
C.3	Multiplikation großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB RAM , 1.83 GHz). Betriebssystem : Windows	94
C.4	Addition großer Gleitkommazahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM , 2.4 GHz). Betriebssystem: Linux	94
C.5	Subtraktion großer Gleitkommazahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM 2.4 GHz). Betriebssystem: Linux	94

C.6	Multiplikation großer Gleitkommazahlen auf einem 64 bit AMD Prozessor (16 Kerne, 128 GB RAM , 2.4 GHz). Betriebssystem: Linux	95
C.7	Addition großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	95
C.8	Subtraktion großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	95
C.9	Multiplikation großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB RAM , 2.4 GHz). Betriebssystem: Windows	95

Algorithmenverzeichnis

4.1.1 Die QR -Zerlegung mittels Gram-Schmidt Orthogonalisierung . .	27
4.2.1 Orthogonalisierungsabschnitt des Householder -Algorithmus . .	30
4.2.2 Berechnung des Faktors R mittels Householder -Algorithmus . .	31
4.3.1 Givens Rotations. $\text{rotate}(x_i, y_i)$	33
5.1.1 Gauß-Reduktionsalgorithmus für die euklidische Norm	36
5.1.2 Größenreduktion	38
5.2.1 Orthogonalisierungsabschnitt der LLL -Reduktion	40
5.2.2 LLL -Reduktion	41
5.3.1 Orthogonalisierungsabschnitt des SE -Algorithmus	42
5.3.2 LLL Schnorr Euchner Algorithmus für Gleitkommaarithmetik . .	43
5.3.3 LLL Koy-Schnorr Algorithmus für Gleitkommaarithmetik . . .	45
5.3.4 Givens -Rotation. Korrektur einer fast obere Dreiecksmatrix, die in der Zeile l gestört ist.	46
5.3.5 LLL Algorithmus für Gleitkommaarithmetik mittels Housholder - Reflexion und Givens -Rotation	47

Literaturverzeichnis

- [Adleman] Adleman, L.M.: *On breaking generalized knapsack public key cryptosystems*, Proc. of 15th STOC, S. 402-412, ACM, 1983
- [Ajtai02] Ajtai, M.: *Random lattices and a conjectured 0 – 1 law about their polynomial time computable properties*. In: FOCS. (2002) 733-742
- [Ajtai06] Ajtai, M.: *Generating random lattices according to the invariant distribution* (2006)
- [BaWe] Backes, Werner; Wetzel, Susanne: *An Efficient LLL Gram Using Buffered Transformations*, Proceedings of the 10th International Workshop on Computer Algebra in Scientific Computing (CASC), LNCS, Springer, 2007, pp. 31-44. Homepage: <http://www.cs.stevens.edu/~wbackes>
- [BLRS] Buchmann, Johannes; Lindner, Richard; Rückert Markus; Schneider, Michael: *Explicit hard instances of the shortest vector problem*. <http://eprint.iacr.org/2008/>. <http://www.latticechallenge.org/>
- [BLAS] Basic Linear Algebra Subprograms. Homepage: <http://www.netlib.org/blas/>
- [BrGi] Bracha, Gilad; *Generics in the Java Programming Language*. Homepage: <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- [ChRi] Chor, B; Rivest, R: *A Knapsack Type Public Key Cryptosystem Based on Arithmetic in Finite Fields*, *IEEE Trans. Information Theory* Vol. 34, No. 5, S.901-909, September 1988.
- [Colt] *Open Source Libraries for High Performance Scientific and Tech-*

- nical Computing in Java*. Homepage: <http://acs.lbl.gov/~hoschek/colt/>
- [CoMath] *The Apache Commons Mathematics Library*. Homepage: <http://commons.apache.org/math/>
- [CoSh] Coppersmith, D.; Shamir, A. : *Lattice attacks on NTRU*. In Advances in Cryptology; Eurocrypt 1997, pages 52-61, 1997.
- [DaJM] Dautelle, Jean-Marie *The Java Solution for Real-Time and Embedded Systems* <http://jscience.org/>
- [DaOe] Özgür D.: Parallelisierung von Gitterbasisreduktionen. Homepage: http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Oezguer_Dagdelen.diplom.pdf
- [BaFi] Filipović, Bartol: *Implementierung der Gitterbasenreduktion in Segmenten*. Diplomarbeit an der Wolfgang Goethe-Universität, Frankfurt am Main. Homepage <http://www.mi.informatik.uni-frankfurt.de/research/masterthesen/filipovic.diplom.2002.pdf>
- [Forb] Forbrig, Peter: *Objektorientierte Softwareentwicklung mit UML*; Verlag: Fachbuchverlag Leipzig; Auflage: 2 (2002); ISBN-13: 978-3446219755
- [Garey] Garey M. R; Johnson, D. S. : *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
- [HoPiSi] J. Hoffstein. J. Pipher und J.H. Silverman : *NTRU: a ring based public key cryptosystem*, Proc. of ANTS III, volume 1423 of LNCS, S. 267-288, Springer-Verlag 1998.
- [GLC] Golub, Gene H.; van Loan, Charles F.: *Matrix Computations (3rd ed.)*, Johns Hopkins, 1996, ISBN 978-0-8018-5414-9 .
- [GMP] *The GNU Multiple Precision Arithmetic Library*, the fastest bignum library on the planet. Homepage: <http://gmplib.org/>
- [GoMa] Golstein, D.; Mayer, A.; *On the equidistribution of Hecke Points*. Forum Mathematicum, 15:165-189, 2003.

- [JM] Jama: *A Java Matrix Package Homepage*: <http://math.nist.gov/javanumerics/jama/>
- [JAVA] Sun Microsystems. Homepage: <http://de.sun.com>
- [JNL] *Java Native Library*, Visual Numerics. Homepage: <http://www.vni.com/>
- [JAS] *Java Algebra System (JAS) Project*. Homepage: <http://krum.rz.uni-mannheim.de/jas/>
- [JP] (JAMPACK) *A Java Package for matrix computations*. Homepage: <ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html>
- [JMAT] *Java Matrix Toolkit*. Homepage : <http://sourceforge.net/projects/jmat>
- [JSci] *A science API for Java*. Homepage: <http://jsci.sourceforge.net>
- [Koy99] Koy, Henrik: *Angriffe auf das GGH-Kryptosystem mittels Gitterreduktion in Blöcken*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Informatik, Diplomarbeit, Okt 1999. - Die Arbeit ist über <http://www.mi.informatik.uni-frankfurt.de/research/masterthesen.html> online erhältlich
- [Koy04] Koy, Hendrik; *Primale/duale Segment-Reduktion von Gitterbasen*. Homepage <http://www.mi.informatik.uni-frankfurt.de/research/papers/primdual.ps>.
- [KS02] Koy, Hendrik; Schnorr, Claus Peter: *Segment and Strong Segment LLL-Reduction of Lattice Bases*. Homepage: <http://www.mi.informatik.uni-frankfurt.de/research/papers.html>
- [KS01] Koy, Hendrik; Schnorr, Claus Peter: *Segment LLL-Reduction of Lattice Bases* Homepage: <http://www.mi.informatik.uni-frankfurt.de/research/papers.html>
- [KoySch] Koy, Henrik ; Schnorr, Claus Peter: *Segment LLL-Reduction*

- with Floating Point Orthogonalization. In: Silverman, Joseph H. (Hrsg.): *Cryptography and Lattices Conference 2001* (LNCS) Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Mar 2001 (Lecture Notes in Computer Science).
- [LAPAK] Homepage: <http://www.netlib.org/lapack/>
- [LLL82] Lenstra, A. K., Lenstra, H. W., and Lovász, L: *Factoring Polynomials with Rational Coefficients*. Math. Ann., volume 261: pages 515-534, 1982
- [MTJ] *The Matrix Toolkit Java*. Homepage: <http://rs.cipr.uib.no/mtj/overview.html>
- [NIST] *National Institute of Standards and Technology* homepage: www.nist.gov
- [NTL] *A Library for doing Number Theory*. Homepage: <http://www.shoup.net/ntl/>
- [Odlyzko] Odlyzko, A.M.: *The rise and fall of knapsack cryptosystems*, *Cryptography and Computational Number Theory*, volume 42 of Proc. of Symposia in Applied Mathematics, S. 75-88, A.M.S. 1990.
- [OeWa] Walter, Oevel: *Einführung in die Numerische Mathematik*. 1. Auflage. Heidelberg, Berlin, Oxford : Spektrum der Wissenschaft, 1996.
- [PH98] Patterson, David A., Hennessy, John L: *Computer Organization and Design: The Hardware/Software Interface*. Second Edition. San Francisco, CA, USA :Morgan Kaufmann, 1998.-ISBN 1-55860-428-6
- [SE93] Schnorr, Claus Peter: *Factoring Integers and Computing Discrete Logarithms via Diophantine Approximation*. In: Cai, Jim-Yi (Hrsg.): *Advances in Computational Complexity* Bd. 13, AMS, 1993, S. 171-182. Homepage: <http://www.mi.informatik.uni-frankfurt.de/research/papers.html>
- [Sch03] Schnorr, Claus Peter: *Lattice Reduction by Random Sampling and Birthday Methods*. In: H. Alt and M. Habib (editors), *STACS*

- 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of LNCS, pages 146-156. Springer, 2003.
- [Sch09] Schnorr, Claus Peter: *Vorlesung Gitter und Kryptographie*. Homepage: <http://www.mi.informatik.uni-frankfurt.de/teaching/SS2009/Vorlesung/gitter.pdf>
- [Sil01] Silverman, J. H.: *Cryptography and Lattices*, volume 2146 of LNCS. Springer-Verlag, 2001.
- [StNg] Nguyen, Phong; Stehlé, Damien: *Floating-point LLL Revisited* <http://perso.ens-lyon.fr/damien.stehle/FPLLL.html>
- [StNg06] Nguyen, P.Q., Stehlé, D.: *LLL on the average*. In: 7th International Symposium on Algorithmic Number Theory (ANTS 2006). (2006) 238-256 <http://perso.ens-lyon.fr/damien.stehle/downloads/average-corrected.pdf>
- [Tomm] Tommila, Mikko; *Apfloat Specification* www.apfloat.org.
- [Vi07] Villard, G.: *Certification of the QR factor R, and of Lattice Basis Reducedness*. <http://perso.ens-lyon.fr/gilles.villard/BIBLIOGRAPHIE/PDF/issac07qr.pdf>

Anhang A

Grundlagen der Komplexitätstheorie

Für eine formale Einführung in die Komplexität wird auf das Buch von Garey und Johnson [Garey] verwiesen, aus dem die folgende Beschreibung stammt. Die Anzahl der Schritte, die ein Algorithmus benötigt, wird als *Laufzeit* des Algorithmus bezeichnet. Der Begriff Schritt bezieht sich auf ein bestimmtes Maschinenmodell (Rechner). Es ist davon auszugehen, dass dieses Modell in der Lage ist, einen einzelnen Schritt in konstanter Zeit auszuführen. Die Laufzeit hängt dann im Allgemeinen von der Eingabe und genauer von der konkreten Implementierung des Algorithmus ab, insbesondere von der Länge der Eingabe, die auch als Problemgröße bezeichnet wird.

Definition A.0.1

Definiert sei zuerst die Bitlänge l für Zahlen bzw. Matrizen (ohne Vorzeichen) als

- $l(0) = 1$
- $l(n) = \lfloor \log_2(n) \rfloor + 1$ für $n \in \mathbb{N}$
- $l(p/q) = l(p) + l(q)$ mit $p, q \in \mathbb{N}$ und $\gcd(p, q) = 1$
- $l([a_{i,j}]) := \sum_{i,j} l(a_{i,j})$ für $[a_{i,j}] \in \mathbb{Q}^{n \times d}$

Die Zeitkomplexität ist eine Funktion $T(l)$ in Abhängigkeit von der Eingabelänge l . Der Wert von $T(l)$ ist die Laufzeit des Algorithmus im schlechtesten Fall (worst case), d.h. die Anzahl der Schritte, die bei einer beliebigen Eingabe höchstens ausgeführt werden. Die Zeit-Komplexität wird mit Hilfe

der \mathcal{O} -Notation angegeben. Die \mathcal{O} -Notation gibt nur die Größenordnung der Komplexität wieder, d.h. ob es sich z.B. um eine linear, quadratisch oder exponentiell wachsende Funktion handelt. Die Größenordnung wird in Form einer Komplexitätsklasse angegeben. Mathematisch gesehen steht $\mathcal{O}(g(l))$ für eine Menge von Funktionen und zwar für all diejenigen Funktionen, die nicht stärker wachsen als die Funktion $g(l)$. Das heißt für diejenigen, die durch die Funktion $g(l)$ (bis auf einen konstanten Faktor) nach oben beschränkt sind. Genauer gesagt gilt für $f, g : \mathbb{N} \rightarrow \mathbb{N}$

$$f(l) \in \mathcal{O}(g(l)) \iff \exists c, l_0 \in \mathbb{N} \text{ so dass } \forall l \geq l_0 \text{ gilt : } f(l) \leq c \cdot g(l)$$

Die Funktion $g(l)$ wird dann als asymptotische obere Schranke von f bezeichnet.

Allgemein sagt die Schreibweise $f(l) \in \mathcal{O}(g(l))$ aus, dass die Funktion f durch die Funktion g asymptotisch nach oben beschränkt ist. Ein Problem ist in polynomieller Zeit lösbar, wenn ein Algorithmus A und ein Polynom

$$P : \mathbb{N} \rightarrow \mathbb{N}$$

mit $\text{Grad} \leq K \in \mathbb{N}$ für A existieren, so dass für jede Instanz des Problems die Rechenzeit durch $P(|l|)$ begrenzt ist, wobei $|l|$ die Eingabegröße des Problems ist. Neben der Zeitkomplexität eines Algorithmus besteht das Interesse an die sogenannte Speicherkomplexität, auf die es hier nicht eingegangen wird.

Nach dem die Eingabelänge eines Problems definiert wurde, sollen nun die Algorithmen nach ihrer Laufzeit charakterisiert werden. Dabei sind Polynomialzeit-Algorithmen von besonderem Interesse.

Definition A.0.2

Ein Algorithmus hat Polynomialzeit-Komplexität, falls die Schrittzahl (Turing-Maschine oder Anzahl Bitoperationen) polynomiell in der Länge der (gültigen) Eingaben beschränkt ist, d.h. es existiert ein Polynom $P \in \mathbb{R}[X]$, so dass

$$\text{Schrittzahl (Eingabe)} \leq P(l \text{ (Eingabe)}).$$

Solche Polynomialzeit-Algorithmen werden in der Regel als effizient betrachtet.

Definition A.0.3 (Charakteristische Funktion)

Für die Sprache A mit Alphabet $\mathcal{A}^* = \{0, 1\}^*$ ¹ ist die charakteristische Funktion $\chi_A(w) : \mathcal{A}^* \rightarrow \{0, 1\}$ definiert durch

$$\chi_A(w) = \begin{cases} 1, & \text{falls } w \in A \\ 0, & \text{sonst} \end{cases}.$$

Definition A.0.4 (Polynomialzeit-Sprachen \mathcal{P})

Die Klasse \mathcal{P} der Polynomialzeit-Sprachen besteht genau aus den Sprachen A , für welche die charakteristische Funktion χ_A in Polynomialzeit berechenbar ist.

Definition A.0.5 (Die Klasse \mathcal{NP})

Die Klasse \mathcal{NP} der nichtdeterministischen Polynomialzeit-Sprachen $A \subseteq \{0, 1\}^*$ ist erklärt durch:

$$\begin{aligned} A \in \mathcal{NP} &\iff \exists B \in \{0, 1\}^* \times \{0, 1\}^*, B \in \mathcal{P} : \\ &A = \{x \in \{0, 1\}^* \mid \exists y \in \{0, 1\}^{P(l(x))} \text{ mit } (x, y) \in B\} \end{aligned}$$

Sei $(x, y) \in B$. Dann heißt y Zeuge für $x \in A$.

Für jedes Wort aus der Sprache A gibt es einen Zeugen, so dass wir in Polynomialzeit entscheiden können, ob $(x, y) \in B$ gilt. Ist in Polynomialzeit ein Zeuge berechenbar, bzw. zu entscheiden, dass es keinen Zeugen gibt, dann gilt $A \in \mathcal{P}$. Es folgt $\mathcal{P} \subseteq \mathcal{NP}$. Existiert ein Zeuge y für $x \in A$, so gibt es einen nichtdeterministischen Polynomialzeitalgorithmus, welcher $x \in A$ entscheiden kann, daraus folgt jedoch nicht, dass $A \in \mathcal{P}$, da für $x \notin A$ nicht garantiert ist, dass die Nichtexistenz eines Zeugen in Polynomialzeit bewiesen werden kann.

Definition A.0.6 (Cook-Karp-Reduktion)

Seien A, B Sprachen:

$$A \leq_{pol} B \iff \exists h: A \rightarrow B : \forall x \in \{0, 1\}^* : x \in A \iff f(x) \in B,$$

wobei h eine Abbildung ist, so dass für alle $x \in A$ der Funktionswert $h(x)$ in Polynomialzeit berechenbar ist.

Die Relation \leq_{pol} ist transitiv ($A \leq_{pol} B \wedge B \leq_{pol} C \Rightarrow A \leq_{pol} C$). Es gilt $A \leq_{pol} B$ wenn man $x \in A$ in Polynomialzeit mit einmaliger Anwendung eines Orakels für B entscheiden kann.

¹Ein Alphabet ist eine endliche Menge von voneinander unterscheidbaren Symbolen, die auch Zeichen oder Buchstaben genannt werden.

Definition A.0.7 (\mathcal{NP} -schwer)

Eine Sprache $A \subseteq \{0, 1\}^*$ heißt \mathcal{NP} -schwer, wenn

$$\forall B \in \mathcal{NP} : B \leq_{pol} A$$

Hat man ein Orakel für ein \mathcal{NP} -schweres Problem, so kann man jedes Problem in \mathcal{NP} in Orakel-Polynomialzeit lösen.

Definition A.0.8 (\mathcal{NP} -vollständig)

Eine Sprache A heißt \mathcal{NP} -vollständig, wenn $A \in \mathcal{NP}$ und $B \leq_{pol} A$ für alle $B \in \mathcal{NP}$.

Beispiel A.0.1 (\mathcal{NP} -vollständige Probleme)

- Das Entscheidungsproblem „Hat ein rationales bzw. ein ganzzahliges Ungleichungssystem $Ax \leq b$ eine ganzzahlige Lösung?“ ist \mathcal{NP} -vollständig.
- Das Rucksack-Problem ist für ein $c \in \mathbb{Z}^n$ und $a \in \mathbb{Z}^n$ gegeben durch

$$\max \left\{ \sum_{j=1}^n c_j x_j \mid \sum_{j=1}^n a_j x_j \leq b, x_j \in \{0, 1\} \text{ für } j \in \{1, 2, \dots, n\} \right\}$$

Anhang B

Benchmark-Tabellen für Langganzzahlarithmetik

Bitlänge	128	256	512	1024
BigInteger	229.6 ns	398.4 ns	584.3 ns	900.0 ns
jas.arith.BigInteger	232.8 ns	265.6 ns	556.2 ns	1.150 us
LargeInteger	98.43 ns	120.3 ns	242.1 ns	440.6 ns
Apint	2.290 us	2.996 us	3.965 us	6.142 us
FPBigInt	185.9 ns	234.3 ns	426.5 ns	665.6 ns

Tab. B.1: Addition großer ganzer Zahlen auf einem 32 bit Genuine Intel(R)
Prozessor (512 MB **RAM**). Betriebssystem : **Windows**

Bitlänge	128	256	512	1024
BigInteger	671.8 ns	1.703 us	7.093 us	25.75 us
jas.arith.BigInteger	703.1 ns	1.890 us	6.984 us	26.51 us
LargeInteger	781.2 ns	1.953 us	6.843 us	23.50 us
Apint	8.328 us	14.57 us	556.2 ns	1.150 us
FPBigInt	543.7 ns	1.653 us	6.145 us	24.72 us

Tab. B.2: Multiplikation großer ganzer Zahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**). Betriebssystem : **Windows**

Bitlänge	128	256	512	1024
BigInteger	87.50 ns	231.2 ns	403.1 ns	614.0 ns
jas.arith.BigInteger	132.8 ns	245.3 ns	373.4 ns	595.3 ns
LargeInteger	317.1 ns	420.3 ns	559.3 ns	962.5 ns
Apint	464.0 ns	525.0 ns	431.2 ns	492.1 ns
FPBigInt	239.0 ns	273.4 ns	381.2 ns	675.0 ns

Tab. B.3: Subtraktion großer ganzer Zahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**). Betriebssystem : **Windows**

Bitlänge	128	256	512	1024
BigInteger	126.0 ns	100.0 ns	112.0 ns	191.0 ns
jas.arith.BigInteger	220.0 ns	94.00 ns	122.0 ns	206.0 ns
LargeInteger	56.00 ns	50.00 ns	76.00 ns	215.0 ns
Apint	656.0 ns	540.0 ns	619.0 ns	749.0 ns
FPBigInt	110.0 ns	110.0 ns	174.0 ns	158.0 ns

Tab. B.4: Addition großer ganzer Zahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM**, 2.4 GHz). Betriebssystem: **Linux**

Bitlänge	128	256	512	1024
BigInteger	140.0 ns	360.0 ns	1.220 us	4.770 us
jas.arith.BigInteger	140.0 ns	400.0 ns	1.230 us	4.780 us
LargeInteger	240.0 ns	420.0 ns	1.200 us	5.820 us
Apint	2.000 us	1.880 us	5.090 us	15.96 us
FPBigInt	134.0 ns	326.0 ns	1.028 us	4.101 us

Tab. B.5: Multiplikation großer ganzer Zahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM** 2.4 GHz). Betriebssystem: **Linux**

Bitlänge	128	256	512	1024
BigInteger	24.00 ns	42.00 ns	62.00 ns	98.00 ns
jas.arith.BigInteger	114.0 ns	176 ns	278 ns	254 ns
LargeInteger	114.0 ns	176 ns	278 ns	254 ns
Apint	158 ns	106 ns	138 ns	106 ns
FPBigInt	82 ns	78 ns	102 ns	164 ns

Tab. B.6: Subtraktion großer ganzer Zahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM**, 2.4 GHz). Betriebssystem: **Linux**

Bitlänge	128	256	512	1024
BigInteger	253.1 ns	353.1 ns	526.5 ns	864.0 ns
jas.arith.BigInteger	251.5 ns	384.3 ns	532.8 ns	895.3 ns
LargeInteger	134.3 ns	200.0 ns	271.8 ns	479.6 ns
Apint	2.323 us	3.001 us	4.162 us	6.132 us
FPBigInt	181.2ns	229.6 ns	134.3 ns	473.4 ns

Tab. B.7: Addition großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Bitlänge	128	256	512	1024
BigInteger	515.6 ns	1.546 us	5.859 us	22.06 us
jas.arith.BigInteger	593.7 ns	1.750 us	5.843 us	22.23 us
LargeInteger	1.031 us	2.437 us	7.328 us	24.31 us
Apint	8.546 us	18.64 us	60.56 us	206.8 us
FPBigInt	514.0 ns	1.531 us	5.279 us	20.07 us

Tab. B.8: Multiplikation großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Bitlänge	128	256	512	1024
BigInteger	75.00 ns	109.3 ns	171 ns	306.2 ns
jas.arith.BigInteger	107 ns	114 ns	182 ns	339 ns
LargeInteger	251 ns	384 ns	509 ns	710 ns
Apint	600 ns	581 ns	490 ns	428 ns
FPBigInt	232 ns	257 ns	354 ns	540 ns

Tab. B.9: Subtraktion großer ganzer Zahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Anhang C

Benchmark-Tabellen für Arithmetik auf großen Gleitkommazahlen

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	410.9 ns	9.346 us ?	643.7 ns
FloatingPoint	2.173 us	2.368 ns	3.965 us
Apfloat	2.806 us	3.404 us	3.198 us

Tab. C.1: Addition großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**, 1.83 GHz). Betriebssystem :**Windows**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	548.4 ns	554.6 ns	787.5 ns
FloatingPoint	1.067 us	1.439 us	1.571 us
Apfloat	539.0 ns	548.4 ns	329.6 ns

Tab. C.2: Subtraktion großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**, 1.83 GHz). Betriebssystem : **Windows**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	10.29 us	27.42 us	81.17 us
FloatingPoint	3.654 us	8.726 us	29.42 us
Apfloat	9.046 us	16.35 us	39.91 us

Tab. C.3: Multiplikation großer Gleitkommazahlen auf einem 32 bit Genuine Intel(R) Prozessor (512 MB **RAM**, 1.83 GHz). Betriebssystem : **Windows**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	262 ns	3.867 us	218.0 ns
FloatingPoint	495.0 ns	598.0 ns	1.640 us
Apfloat	576.0 ns	557.0 ns	624.0 ns

Tab. C.4: Addition großer Gleitkommazahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM**, 2.4 GHz). Betriebssystem: **Linux**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	386.0 ns	337.0 ns	340.0 ns
FloatingPoint	218.0 ns	270.0 ns	408.0 ns
Apfloat	85.00 ns	81.00 ns	81.00 ns

Tab. C.5: Subtraktion großer Gleitkommazahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM** 2.4 GHz). Betriebssystem: **Linux**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	4.481 us	11.88 us	34.33 us
FloatingPoint	1.171 us	3.996 us	24.10 us
Apfloat	1.233 us	1.968 us	4.259 us

Tab. C.6: Multiplikation großer Gleitkommazahlen auf einem 64 bit **AMD** Prozessor (16 Kerne, 128 GB **RAM**, 2.4 GHz). Betriebssystem: **Linux**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	428.1 ns	10.73 us	560.9 ns
FloatingPoint	1.843 us	2.032 us	4.564 us
Apfloat	2.421 us	3.256 us	3.992 us

Tab. C.7: Addition großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	475.0 ns	471.8 ns	550.0 ns
FloatingPoint	840.6 ns	1.207 us	1.778 us
Apfloat	410.9 ns	417.1 ns	543.7 ns

Tab. C.8: Subtraktion großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Klasse	Bitlänge/Precision		
	128/112	256/223	512/453
BigDecimal	12.72 us	35.64 us	103.5 us
FloatingPoint	3.503 us	11.37 us	32.07 us
Apfloat	8.762 us	18.74 us	50.20 us

Tab. C.9: Multiplikation großer Gleitkommazahlen auf einem 32 bit Intel Pentium 4 Prozessor (1 GB **RAM**, 2.4 GHz). Betriebssystem: **Windows**

Anhang D

Klassendiagramm des Programms



Abbildung D.1: Teil-Klassendiagramm des Programm