Johann Wolfgang Goethe-Universität
Frankfurt am Main
Institut für Informatik am Fachbereich Biologie und Informatik (15)

Implementierung der Gitterbasenreduktion in Segmenten

Diplomarbeit

von

Bartol Filipović

4. März 2002

Betreuer: Prof. Dr. Claus Peter Schnorr
Arbeitsgruppe Mathematische Informatik

Diese Diplomarbeit wurde mit dem Textsatzsystem TEX (Version 3.14159) gesetzt. Bei der Erzeugung dieses Dokumentes kamen pdfTEX (Version 3.14159-1.00a-pretest-20010806), LATEX 2_ε (Versionsdatum 2001/06/01) und diverse LATEX-Ergänzungspakete zur Anwendung.

Dokument-Version: 2002.03.04.final.0.pdf

Implementierung der Gitterbasenreduktion in Segmenten

Diplomarbeit von
Bartol Filipović
am Institut für Informatik der Johann Wolfgang Goethe-Universität Frankfurt

Zusammenfassung

Die Gitterbasenreduktion hat in der algorithmischen Zahlentheorie und der Kryptologie bedeutende und praktisch relevante Anwendungen [Joux und Stern, 1998; Nguyen und Stern, 2000; Nguyen, 2001]. Ein wesentlicher Beitrag auf dem Gebiet der Gitter-Reduktionsalgorithmen ist der LLL-Algorithmus [Lenstra, Lenstra und Lovász, 1982] und auch die β-Reduktion (BKZ-Reduktion) von Gitterbasen [Schnorr, 1987, 1988, 1994] ist von großer Bedeutung[1]. Bei Implementierungen dieser Algorithmen auf modernen Rechnerarchitekturen erfolgen viele Berechnungen aus Gründen der schnelleren Verarbeitungsgeschwindigkeit in Gleitpunktzahlen-Arithmetik. Aufgrund inhärenter Rundungsfehler kommt es dabei zu numerischen Instabilitäten. Vor [Koy und Schnorr, 2001b] gab es keine erfolgreichen Ansätze die bei der Gitterbasenreduktion auftretenden Rundungsfehler so zu kontrollieren, dass auch Gitterbasen in der Dimension ≥ 400 reduziert werden können. Diese Diplomarbeit beschäftigt sich mit den praktischen Aspekten der Gitterbasenreduktion in Segmenten. Dabei handelt es sich um die erstmalige Implementierung und experimentelle Evaluierung der folgenden beiden Verfahren:

- Skalierte k-Segment LLL-Reduktion nach [Koy und Schnorr, 2001b]. Koy und Schnorr stellen in ihrer Arbeit wirksame und effiziente Strategien zur Vermeidung von numerischen Instabilitäten bei der Gitterbasenreduktion mit Gleitpunktzahlen vor. Unsere Implementierung des Verfahrens ermöglicht die Reduktion von Gitterbasen in Dimension ≥ 1000.
- Gitterbasenreduktion in primalen/dualen Segmenten nach [Koy, 2002]. Dieses Verfahren liefert für eine feste Segmentgröße beweisbar gute Reduktionsergebnisse bei polynomieller Laufzeit. Wir vergleichen das Verfahren mit der β-Reduktion.

Die Verfahren wurden in `C++` implementiert. Diese Programmiersprache ermöglicht aufgrund des Template-Konzeptes eine generische Programmierung: Die Reduktionsalgorithmen brauchen nur einmal programmiert werden, können aber verschiedene Gleitpunktzahlen-Datentypen – auch die nicht-Standard Typen aus [Shoup, 2001] – verwenden. Wir haben das Kommandozeilen Programm `latred` erstellt, um Experimente einfach durchführen und auswerten zu können. Darin sind die skalierte LLL-Reduktion [Koy und Schnorr, 2001b], die skalierte k-Segment LLL-Reduktion, die primal/duale Segmentreduktion und die Reduktionsverfahren aus [Shoup, 2001] integriert. Unser Programm bietet auch die Möglichkeit, bestimmte reduzierte/nicht-reduzierte Gitterbasenpaare zu erzeugen, welche als Eingabe- bzw. Referenzbasen für Experimente dienen. Mit `latred` sind alle relevanten Parameter für ein gewähltes Reduktionsverfahren beim Programmaufruf über die Kommandozeile einstellbar. So kann beispielsweise angegeben werden, welcher Gleitpunktzahlen-Datentyp bei der Reduktion zu verwenden ist. Das Programm kann Protokolldateien des Reduktionsverlaufes erstellen, anhand derer man die Reduktion verfolgen kann. Insgesamt betrachtet ist `latred` ein nützliches Programm zur Gitterbasenreduktion.

[1] Die β-Reduktion liefert in der Praxis sehr gute Reduktionsergebnisse, allerdings ist keine polynomielle Laufzeitschranke bekannt.

Nach einer kompakten Einführung in die Gittertheorie (Kapitel 2) werden in Kapitel 3 einige Verfahren zur Gitterbasenreduktion kurz vorgestellt (Längenreduktion, LLL-Reduktion, β-Reduktion, Reduktion in lokalen Koordinaten), welche die spätere Beschreibung der Segmentreduktionsverfahren vorbereiten.

In Kapitel 4 beschäftigen wir uns mit der Darstellung von Gleitpunktzahlen in Computern und den (numerischen) Eigenschaften der Gleitpunktzahlen-Arithmetik. In diesem Kontext werden die Householder-Reflexion und die Givens-Rotation als numerisch stabile Methoden für die QR-Faktorisierung (Orthogonalisierung) einer Gitter-Basismatrix vorgestellt (die Orthogonalisierung von Vektoren ist ein Bestandteil der Algorithmen zur Gitterbasenreduktion). Zum Abschluss des Kapitels 4 motivieren wir den Gebrauch von Gleitpunktzahlen bei der Gitterbasenreduktion.

Das Kapitel 5 widmet sich den beiden Verfahren zur Gitterreduktion in Segmenten: (skalierte) k-Segment LLL-Reduktion und primal/duale Segmentreduktion. Die Eigenschaften der Verfahren und die Reduktionsalgorithmen werden vorgestellt.

Das Kapitel 6 befasst sich mit der Implementierung des `latred` Programms. Die Orthogonalisierungsroutinen und die Routinen für die beiden Segmentreduktionsverfahren werden beschrieben. Wir passen die HRS-Routine aus [Koy und Schnorr, 2001b] den verwendeten Gleitpunktzahlen an. Die Algorithmen zur Erzeugung von reduzierten/nicht-reduzierten Gitterbasenpaaren werden angegeben.

Abschließend präsentieren wir in Kapitel 7 die mit `latred` erzeugten experimentellen Resultate.

Danksagung

Bei Prof. Dr. Claus Peter Schnorr und Henrik Koy bedanke ich mich sehr für wertvolle Diskussionen und Anregungen. Auch für ihre freundliche Genehmigung, die Arbeit [Koy und Schnorr, 2001b] im Anhang dieser Diplomarbeit abzudrucken, bin ich beiden dankbar. Meinen Eltern und meinem Bruder danke ich herzlichst für ihre Unterstützung.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfaßt und keine anderen Hilfsmittel als die angegebenen Quellen verwendet habe.

Bartol Filipović Frankfurt am Main, den 4. März 2002

Inhaltsverzeichnis

1 Einleitung

Diese Diplomarbeit beschäftigt sich mit neuen Varianten der Gitterbasenreduktion nach [Koy und Schnorr, 2001a,b][1] und [Koy, 2002]. Wir beschreiben die Implementierung der Verfahren und stellen experimentelle Ergebnisse vor.

Abschnitt 1.1 befasst sich mit der historischen Entwicklung von Algorithmen zur Gitterbasenreduktion. Es werden auch praktisch relevante Anwendungen angesprochen.

Der Abschnitt 1.2 auf Seite 5 widmet sich der Intention dieser Diplomarbeit und den erbrachten Leistungen.

1.1 Über Gitter und Reduktion von Gitterbasen

Gitter sind diskrete[2], nicht-triviale[3] additive Untergruppen des $\mathbb{R}^d$. Ein Gitter wird durch eine Basis, das sind n linear unabhängige Vektoren aus dem $\mathbb{R}^d$, erzeugt. Genauer gesagt bezeichnet man die (unendlich große) Menge aller *ganzzahligen* Linearkombinationen der Basisvektoren als Gitter. Die Anzahl n der Basisvektoren nennt man den Rang des Gitters; falls $n = d$ sprechen wir lediglich von der Dimension n des Gitters. Die Basisvektoren des Gitters werden üblicherweise in Matrizenschreibweise angegeben (Basismatrix). Gitter besitzen vom Nullvektor verschiedene kürzeste Gittervektoren. Die Länge eines Vektors ist dabei durch eine Norm auf dem $\mathbb{R}^d$ bestimmt – meistens betrachtet man die euklidische Norm. Eine Gitterbasis ist nicht eindeutig: Zu einem Gitter mit Rang $n > 1$ gibt es unendlich viele Basen, die alle das gleiche Gitter erzeugen. Das Ziel der Gitterbasenreduktion ist, Basen des Gitters zu finden, die aus Vektoren bestehen, die möglichst kurz und zueinander orthogonal sind. Liegen die reduzierten Basisvektoren der Länge nach aufsteigend geordnet vor, so entspricht im Idealfall der erste Basisvektor einem vom Nullvektor verschiedenen kürzesten Gittervektor (das sogenannte *shortest vector problem* wäre gelöst). Natürlich ist man an möglichst effizienten Reduktionsalgorithmen interessiert, das heißt eine akzeptable Laufzeit und eine ausreichende Approximation des kürzesten Gittervektors soll erreicht werden. Der Grund für das Interesse an kurzen Vektoren in Gittern ist die Tatsache, dass Probleme aus verschiedenen Bereichen der Informatik und Mathematik auf das Bestimmen des kürzesten Gittervektors (bzw. einer Approximation von diesem) in einem Gitter reduzierbar sind. Das heißt also, falls man einen solchen Gittervektor findet, hat man das

[1] Die Arbeit [Koy und Schnorr, 2001b] ist in Anhang C auf Seite 115 wiedergegeben.
[2] Eine Menge $S \subseteq \mathbb{R}^d$ heißt diskret, wenn S keinen Häufungspunkt in $\mathbb{R}^d$ hat.
[3] Eine Menge $S \subseteq \mathbb{R}^d$ heißt nicht-trivial, wenn $S \neq \{0\}$.

Problem (oder eine Instanz des Problems) gelöst. Typischerweise sind Probleme, denen eine gewisse Linearität innewohnt, auf Gitterprobleme reduzierbar, so zum Beispiel *Rucksackprobleme*. Darüberhinaus wurden auch Probleme beweisbar in einen Zusammenhang mit der Gitterbasenreduktion gebracht, die *a priori* nicht linear sind [siehe Coppersmith, 1997, 2001; Nguyen und Stern, 1999; Boneh, 2000; Bleichenbacher und Nguyen, 2000]. An dieser Stelle sei der Hinweis erlaubt, dass es üblich ist, statt von Gitterbasenreduktion auch einfach von Gitterreduktion zu sprechen.

Die Theorie der Gitterreduktion geht zurück auf die äquivalente Reduktionstheorie von quadratischen Formen. Diese wurde unter anderen entwickelt in [Lagrange, 1773; Gauß, 1801; Hermite, 1850; Korkine und Zolotareff, 1872, 1873, 1877; Minkowski, 1891]. Die geometrische Betrachtungsweise der Gitter wurde in der „Geometrie der Zahlen“ von [Minkowski, 1896] entwickelt. Der Gauß'sche Reduktionsbegriff [Gauß, 1801] beschränkt sich auf Gitter mit Rang $n = 2$. Gauß-reduzierte Gitterbasen lassen sich effizient berechnen und bestehen immer aus den beiden kürzesten linear unabhängigen Gittervektoren. Die Begriffe der Hermite-Korkine-Zolotareff-reduzierten Gitterbasen [Hermite, 1850; Korkine und Zolotareff, 1872, 1873, 1877] und der Minkowski-reduzierten Gitterbasen [Minkowski, 1891, 1896] sind für beliebigen Gitterrang definiert. Bei diesen Basen ist der erste Basisvektor stets ein von Null verschiedener kürzester Gittervektor. Es ist jedoch kein effizienter Algorithmus zur Konstruktion solcher Basen bekannt.

Mit dem Einsatz von Computern erlangte die Gitterreduktion zu Beginn der 80'er Jahre neue Impulse als Hendrik W. Lenstra zwecks Integer-Programmierung für *feste* Dimension eine polynomial-Zeit Reduktionsmethode zur Konstruktion eines kürzesten Gittervektors vorstellte. Publiziert wurde diese Arbeit allerdings erst später [Lenstra, 1983]. Durch diesen Algorithmus inspiriert entwickelte László Lovász einen polynomial-Zeit Algorithmus, der für beliebige Dimension eine reduzierte Basis eines Gitters berechnet. Daraus resultierte der berühmte und oft als Meilenstein angesehene LLL-Algorithmus (auch als L^3-Algorithmus bezeichnet). Benannt ist er nach den Initialen von Arjen K. Lenstra, Hendrik W. Lenstra und László Lovász, die in der Arbeit [Lenstra, Lenstra und Lovász, 1982] den LLL-Algorithmus vorstellten und ihn zur Faktorisierung rationaler Polynome angewandt haben. Der LLL-Algorithmus berechnet zu gegebener Basismatrix eine LLL-reduzierte Basismatrix, für deren kürzesten Vektor $\boldsymbol{b}_1$ gilt: $\|\boldsymbol{b}_1\| \leq 2^{\frac{n-1}{2}} \lambda_1$, wobei $\|\cdot\|$ die euklidische Norm und λ_1 der kürzeste von Null verschiedene Gittervektor ist. Die Laufzeit ist dabei polynomiell in der Dimension des Raumes d, dem Rang des Gitters n und der Größe der Einträge in der Eingabe-Basismatrix.

In der Folgezeit wurden einige Verbesserungen zu dem LLL-Algorithmus vorgeschlagen. Claus Peter Schnorr hat für die euklidische Norm die *Blockreduktion* (auch β- oder BKZ-Reduktion genannt) und *blockreduzierte Gitterbasen* eingeführt, um die kürzesten Gittervektoren besser zu approximieren [Schnorr, 1987, 1988, 1994]. Die älteren Reduktionsbegriffe sind ein Speziallfall der Blockreduktion: Hermite-Korkine-Zolotareff-reduzierte Gitterbasen mit Rang m sind blockreduziert mit Blockgröße m, LLL-reduzierte Gitterbasen sind blockreduziert mit Blockgröße 2. Eine polynomielle Laufzeitschranke für die Blockreduktion ist nicht bekannt. Bei kleiner Blockgröße ist die Blockreduktion ähnlich effizient wie die LLL-Reduktion. Experimente zeigen, dass

bei Blockgröße 20 die Blockreduktion etwa das 10-fache der Laufzeit einer LLL-Reduktion benötigt, wobei sie aber deutlich kürzere Gittervektoren liefert [Hörner, 1994; Schnorr und Hörner, 1995]. Die Laufzeit der Blockreduktion ist aber exponentiell in der Blocklänge. Der Grund dafür liegt in der Bestimmung des kürzesten Gittervektor in einem Block durch *Aufzählung* (*Enumeration*), und damit ist ein – im Worst-Case – in der Blockgröße exponentieller Aufwand verbunden [Fincke und Pohst, 1985; Schnorr und Euchner, 1994; Hörner, 1994; Schnorr und Hörner, 1995; Ritter, 1997; Backes, 1998]. Ein Ansatz, den Prozess der Enumeration von Gittervektoren effizienter durchzuführen, ist die *geschnittene Aufzählung* [Hörner, 1994; Schnorr und Hörner, 1995], die auf einer Heuristik nach Gauss beruht.

Auf Schnorr geht der Vorschlag zurück, die Gleitpunktzahlen in Computern, für die während der Gitterreduktion anfallenden Berechnungen, zu nutzen [Schnorr, 1988; Schnorr und Euchner, 1994]. Dieses Vorgehen hat den Vorteil, dass deutlich schneller gerechnet wird, da Gleitpunktarithmetik in heutigen Prozessoren hardwareseitig implementiert ist. Allerdings erkauft man sich diesen Geschwindigkeitsvorteil aufgrund inhärenter Rundungsfehler[4] mit numerischen Instabilitäten [Goldberg, 1991; Oevel, 1996]. Die Hauptursache für die bei dem Rechnen mit der schnellen Gleitpunktarithmetik in Implementierungen von Gitterreduktionsalgorithmen, auftretenden Rundungsfehler wird in [Koy und Schnorr, 2001b] identifiziert. In der Arbeit wird auch begründet, warum es bisher nicht möglich war, Gitterbasen in Dimensionen größer als 400 numerisch stabil zu reduzieren. Darüberhinaus zeigen die beiden Arbeiten [Koy und Schnorr, 2001a,b] auf, wie eine numerisch stabile Gitterbasenreduktion erfolgen kann. Auf diese Thematik bezieht sich ein wesentlicher Teil dieser Diplomarbeit. Wir kommen in Abschnitt 1.2.1 auf Seite 6 ausführlicher darauf zu sprechen.

Die Gitterreduktion wird in den letzten 20 Jahren zunehmend *zur Lösung* unterschiedlicher Probleme eingesetzt. Charakteristisch dabei ist, dass man die Verfahren zur Gitterbasenreduktion als einfach anzuwendende Werkzeuge benutzt, deren Funktionsweise aus Anwendersicht eigentlich nicht näher verstanden sein muss (*black box tool*). Insbesondere in der Kryptologie finden Gitter erfolgreiche Anwendungen: Zum einen in der Kryptanalyse und zum anderen im Design von neuen Kryptoverfahren, die auf schwierigen Gitterproblemen beruhen. Phong Q. Nguyen spricht daher von „The Two Faces of Lattices in Cryptology“ [Nguyen, 2001]. Kryptographisch relevanten Anwendungen der Gitterbasenreduktion werden in [Joux und Stern, 1998; Nguyen und Stern, 2000] beschrieben.

1.1.1 Gitter in der Kryptologie

Neben dem bereits in Abschnitt 1.1 auf Seite 1 angesprochenen *shortest vector problem*, gibt es in der Gittertheorie noch einige weitere „Probleme“, etwa das *closest vector problem* oder das *shortest basis problem*. Wie diese genau definiert sind, stellen wir in Abschnitt 2.6 auf Seite 24 vor. Aufgrund des erfolgreichen Einsatzes der Gitterbasenreduktion in unterschiedlichen Gebieten könnte man glauben, dass Gitterprobleme

[4] Rundungsfehler können die Gitterbasis (also das Gitter) verändern, daher müssen die Basisvektoren in exakter Darstellung gespeichert werden.

nicht besonders schwer sind. Diese Einschätzung scheint jedoch nicht richtig zu sein. Denn in den letzten Jahren sind Resultate bekannt geworden, die eher das Gegenteil vermuten lassen: Unter komplexitätstheoretischen Annahmen sind Gitterprobleme schwierige Probleme [Ajtai, 1998; Blömer und Seifert, 1999; Cai, 1999, 2000; Seifert, 2000]. Auf Grund dieser Ergebnisse kann man Gitterprobleme zunächst als vielversprechende Grundlage für das Design von neuen Public-Key Kryptosystemen halten. Allerdings haben sich die bisherigen Versuche, solche Systeme zu realisieren, als entweder unsicher oder nicht effizient erwiesen; siehe zum Beispiel [Ajtai und Dwork, 1997; Nguyen und Stern, 1998] oder auch [Goldreich, Goldwasser und Halevi, 1997; Nguyen, 1999; Koy, 1999]. Bei der Aufdeckung der Sicherheitsmängel in diesen Verfahren sind praktische Gitterreduktionprogramme – beispielsweise die Programmbibliothek [Shoup, 2001] – zum Einsatz gekommen. Dabei handelt es sich hauptsächlich um die Implementierungen der Reduktionsalgorithmen und Konzepte aus [Schnorr und Euchner, 1994; Schnorr und Hörner, 1995]. Aber auch leicht modifizierte Algorithmen kamen zum Einsatz, etwa in [Koy, 1999].

Erfolgreiche Angriffe auf das Gitterbasiertes GGH-Kryptosystem aus [Goldreich, Goldwasser und Halevi, 1997]:

- In [Koy, 1999] wurde eine geheime GGH-Gitterbasis in Dimension 200 berechnet. Dazu wurde eine Gitterbasis aus den öffentlichen Parametern konstruiert und dann reduziert. Die Reduktion erfolgte durch Kombination der LLL-Block-Reduktion mit der (geschnittenen) BKZ-Reduktion. Der Angriff wurde bereits in der Sicherheitsanalyse in [Goldreich, Goldwasser und Halevi, 1997] erwähnt. Jedoch vermuteten die Autoren, dass die vorgeschlagenen Parameter diesen Angriff unpraktikabel machen würden.

- In [Nguyen, 1999] wurden Fehler im Design des GGH-Kryptosystems ausgenutzt. Das Problem, den cipher-Text zu entschlüsseln[5], reduziert sich auf auf ein spezielles closest vector problem, das einfach zu lösen ist.

Prinzipiell lassen sich die Schwachstellen des GGH-Kryptosystems beseitigen [siehe Nguyen, 1999; Micciancio, 2001], dennoch ist es aus praktischer Sicht nicht besonders konkurrenzfähig, da viel Speicherplatz benötigt wird. Bezugnehmend auf Gitterbasierte Kryptoverfahren stellt die Arbeit [Micciancio, 2001] einen Ansatz vor, mit dem der Speicherplatz Bedarf von Gitterbasen beachtlich verringert werden kann. Ein einfaches Reduktionsargument zeigt, dass dabei keine „Sicherheit verloren“ geht: Jeder erfolgreiche Angriff auf die modifizierte Gitterbasis ist auch bei der ursprünlichen Basis erfolgreich. Somit wäre es möglich, die Sicherheitsparameter bei Gitterbasierten Kryptoverfahren (etwa die Dimension des Gitters) deutlich zu erhöhen – ohne dabei verschwenderisch mit dem Speicherplatz umzugehen. Ob dadurch zum Beispiel das [Goldreich, Goldwasser und Halevi, 1997]-Kryptosystem für den praktischen Einsatz interessant werden kann ist nicht klar, denn im Vergleich zu den etablierten Systemen braucht es immer noch zu viel Speicher.

[5] In [Nguyen, 1999] wird das Kryptosystem von [Goldreich u. a., 1997] gebrochen. Dies geschieht durch die Rekonstruktion des plain-Textes aus dem cipher-Text. Der Angriff vereinfacht aber nicht das Auffinden des geheimen Schlüssels (eine Gitterbasis) in höheren Sicherheitsstufen.

1.2 Intention dieser Diplomarbeit

Wir stellen im Rahmen unserer Arbeit die ersten Implementierungen der *skalierten k-Segment LLL-Reduktion* [Koy und Schnorr, 2001a,b] und der *primal/dualen Segmentreduktion* [Koy, 2002] vor und führen Experimente mit den neuen Verfahren durch, wobei die Folgen verschiedener Verfahrens-Parameter deutlich werden. Wir untersuchen unterschiedliche Gleitpunktzahlen-Datentypen bei der Reduktion.

Die Verfahren sind in der Programmiersprache `C++` unter Verwendung der zahlentheoretischen Bibliothek NTL [Shoup, 2001] implementiert worden. Das *Template*-Konzept von `C++` wird für eine generische Programmierung genutzt: Die Reduktionsalgorithmen brauchen nur einmal programmiert werden, können aber verschiedene Gleitpunktzahlen-Datentypen – auch die nicht-Standard Typen aus [Shoup, 2001] – verwenden. Wir haben das Kommandozeilen Programm `latred`[6] erstellt, um Experimente einfach durchführen und auswerten zu können. Darin sind die skalierte LLL-Reduktion [Koy und Schnorr, 2001b], die skalierte k-Segment LLL-Reduktion und die primal/duale Segmentreduktion, sowie die Reduktionsverfahren aus [Shoup, 2001] integriert. Unser Programm bietet auch die Möglichkeit, bestimmte reduzierte/nichtreduzierte Gitterbasenpaare zu erzeugen, welche als Eingabe- bzw. Referenzbasen für Experimente dienen. Mit `latred` sind alle relevanten Parameter für ein gewähltes Reduktionsverfahren beim Programmaufruf über die Kommandozeile einstellbar. So kann beispielsweise angegeben werden, welcher Gleitpunktzahlen-Datentyp bei der Reduktion zu verwenden ist. Das Programm kann Protokolldateien des Reduktionsverlaufes erstellen, anhand derer man die Reduktion verfolgen kann. Insgesamt betrachtet ist `latred` ein nützliches Programm zur Gitterbasenreduktion. Über seine Benutzung informiert Anhang A auf Seite 107.

Der `C/C++` Quellcode von `latred` wird *nicht* in allen *Einzelheiten* vorgestellt. Beispielsweise beschreiben wir nicht die Implementierung der skalierten LLL-Reduktion aus [Koy und Schnorr, 2001b] und auch die Erzeugung der Protokolldateien wird nicht behandelt. Die NTL spezifischen Datenstrukturen werden lediglich kurz erläutert, genauere Details sind in [Shoup, 2001] zu finden. Wir geben die relevanten Algorithmen zur Gitterbasenreduktion in Segmenten in Pseudocode an oder erklären sie in Worten. Falls das unmittelbar auf die tatsächliche Implementierung übertragbar ist, belassen wir es dabei. Andernfalls wird auf die Unterschiede oder Schwierigkeiten der konkreten Programmierung eingegangen. Somit sollte ein geübter `C++` Programmierer, der den Inhalt dieser Diplomarbeit mathematisch begreift, in der Lage sein, die Implementierung der Gitterbasenreduktion in Segmenten im `latred`-Quellcode zu verstehen.

Die Dokumentation der aufgrund der neuen Reduktion in Segmenten erzielten Fortschritte – im Vergleich zu anderen Implementierungen [Koy, 1999; Shoup, 2001] – ist ein weiterer Aspekt dieser Arbeit. Das Verhalten von bisherigen Programmen zur Gitterreduktion ist zum Beispiel anhand von [Koy, 1999; May, 1999] oder auch [Backes, 1998; Backes und Wetzel, 2000] erkennbar. Um die erreichten Verbesserungen deutlich zu machen, benutzen wir als Testeingaben für unsere Experimente GGH-Gitterbasen. Diese wurden in [Goldreich, Goldwasser und Halevi, 1997] vorgestellt und sind als

[6] `latred` steht für `lat`tice `red`uction (im englischen bezeichnet man ein Gitter als lattice).

„schwierig“ reduzierbar eingestuft. Wir entschieden uns für diese Basen unter anderen deshalb, weil sich die beiden Arbeiten [Koy, 1999; Nguyen, 1999] auch mit solchen Gitterbasen-Typen beschäftigen. Dadurch lassen sich unsere Ergebnisse bzw. die erreichten Verbesserungen bei der Gitterreduktion gut einordnen:

- Ein Vergleich unserer experimentellen Ergebnisse mit denen in [Koy, 1999] weist die deutlich verbesserte numerische Stabilität und effizienz der skalierten Segment LLL-Reduktion nach. Diesen Fortschritt erkauft man durch eine etwas abgeschwächte – im Vergleich zu der klassischen LLL-Reduktion – Reduktionsgüte.

- Ein Vergleich zwischen der Gitterreduktion in primalen/dualen Segmenten und der β-Reduktion (BKZ-Reduktion) macht deutlich, dass die primal/duale Reduktion die Gitterbasen zwar schneller reduziert, aber mit geringerer Reduktiongüte.

1.2.1 Skalierte Segment LLL-Reduktion

Wegen numerischen Instabilitäten kommt es bei *bisherigen* Implementierungen der Reduktionsalgorithmen [Koy, 1999; Shoup, 2001] bei Gitterbasen in hohen Dimensionen zu Endlosschleifen, das heißt die Programme terminieren dann nicht mehr. Versuche, durch zusätzliche Korrekturschritte im Algorithmus dieses Problem in den Griff zu bekommen, waren nur bis in Dimension ca. 150 erfolgreich [Schnorr und Euchner, 1994; Hörner, 1994, Abschnitt 5.2, Seite 46].

Numerisch kritische Rundungsfehler treten hauptsächlich bei der Orthogonalisierung von Vektoren im LLL-Algorithmus auf. Die Eigenschaften von verschiedenen Verfahren zur QR-Faktorisierung (Orthogonalisierung) einer Matrix mit Gleitpunktzahlen sind bekannt [siehe etwa Deuflhard und Hohmann, 1991; Oevel, 1996]. Numerisch stabile Methoden haben akzeptable Fehlerabschätzungen. Auch Gitterreduktions-Programme mit bewiesenermaßen stabilen Orthogonalisierungsverfahren haben Grenzen: In [Koy, 1999, S. 49–50] wird berichtet, dass eine auf Householder-Transformationen[7] basierende Implementierung einer LLL-Block-Reduktion nur Gitterbasen mit Dimension bis inklusive 361 stabil reduziert. Für Gitter mit Dimension 400 ist die Reduktion instabil. Diese Beobachtung wird in [Koy und Schnorr, 2001b, Abschnitt 2] begründet: Householder-Transformationen mit Gleitpunktzahlen, die 53 Präzisionsbits haben (ein Standard bei aktuellen Prozessoren), sind bei der (auf dem LLL-Verfahren basierenden) Gitterbasenreduktion in Dimension ≥ 400 numerisch instabil.

In [Shoup, 2001] wird durch Heuristiken getestet, ob die Reduktions-Routine in einer Endlosschleife festhängt. Eine beweisbar stabile Reduktion von Gitterbasen in hohen Dimensionen fehlte bis zu der Arbeit [Koy und Schnorr, 2001b]. Die neuen stabilen Reduktionsalgorithmen beruhen auf geeignetem Skalieren, einer wohlbekannten Methode zur Stabilitätsverbesserung. Skalieren wurde in der Gitterreduktion erstmals von Henrik Koy eingesetzt. Zusammen mit dem Vorschlag von Claus Peter Schnorr, die

[7] Householder-Transformationen (auch Householder-Reflexionen genannt) gelten in der numerischen Mathematik als stabile Methode eine Matrix, bei Verwendung von Gleitpunktzahlen, zu orthogonalisieren. Wir kommen darauf in Abschnitt 4.2 auf Seite 44 zu sprechen.

LLL-Reduktion etwas abzuschwächen, indem man die Basismatrix in so genannte *Segmente* aufteilt, diese dann *lokal* LLL-reduziert und zwischen je zwei Segmenten einen beschränkten Verlust der Reduktionsgüte zulässt, führt dies dazu, dass es nun möglich ist, Gitterbasen in hohen Dimensionen schnell und stabil zu reduzieren. Diese Reduktion in Segmenten nach [Koy und Schnorr, 2001b] bezeichnen wir als *skalierte Segment LLL-Reduktion* von Gitterbasen.

Unsere Ergebnisse

Wir passen die HRS-Routine aus [Koy und Schnorr, 2001b] den verwendeten Gleitpunktzahlen an. Die Vor- und Nachteile der unterschiedlichen Gleitpunktzahlen-Datentypen machen sich experimentell deutlich bemerkbar: In den meisten für uns interessanten Fällen liefert ein nicht-Standard Gleitpunktzahlen-Datentyp (NTL Typ `quad_float`, der 128 Bits pro Objekt belegt und 105 Präzisionsbits hat) besonders gute Resultate. Unsere Experimente zeigen die Effizienz der Implementierung und bestätigen die Wirksamkeit der Vorschläge aus [Koy und Schnorr, 2001b]. Die Implementierung der skalierten Segment LLL-Reduktion in unserem Programm `latred` führt zu stabilen (wegen Exponentenüberlauf leider nicht immer, siehe weiter unten) und bei guter Wahl der Segmentgröße und des Reduktionsparameters δ auch zu schnellen Reduktionen. Damit ist es im Rahmen dieser Arbeit erstmals gelungen, Gitterbasen mit deutlich größerem Rang als bisher effizient zu reduzieren. Zum Beispiel reduziert unser Programm eine 1000×1000 Basismatrix mit Einträgen von 400-Bit Ganzzahlen in 10 Stunden[8]. Die reduzierte[9] Basismatrix hat Einträge in der Größenordnung von 53-Bit Ganzzahlen.

Die in [Koy und Schnorr, 2001b, Abschnitt 8] zitierten Ergebnisse wurden mit einer frühen Version von `latred` erzeugt.

Ein Grund, warum wir keine Gitterbasen in größeren Dimensionen reduziert haben, liegt in dem dazu benötigten hohen Hauptspeicherbedarf. Da neben der eigentlichen Basismatrix B noch eine skalierte Variante B^s derselbigen und die isometrische Matrix R gespeichert werden müssen, heißt das zum Beispiel, dass bei einer 1200×1200 Basismatrix mit Einträgen von 400-Bit Ganzzahlen für die erwähnten drei Matrizen alleine schon *mindestens* 164 MByte RAM (Haupt-) Speicher nötig ist: Die beiden Matrizen B und B^s benötigen jeweils mindestens[10] 71 MByte Speicher; die Einträge der isometrischen Matrix sind Gleitpunktzahlen und jeder ihrer Einträge belegt typischerweise 64 Bits (`C/C++` Typ `double`) oder 128 Bits (NTL Typ `quad_float`) – bei 128 Bits pro

[8] Auf einem Rechner mit AMD Athlon 800 MHz Prozessor und 448 MByte RAM Hauptspeicher; Segmentgröße $k = 50$; Reduktionsparameter $\delta = 0.99$; Verwendung von Gleitpunktzahlen des Typs `quad_float`.

[9] Die Bitlänge der Matrix-Einträge ist nur ein grober Indikator für die Qualität der Reduktion – die eigentlich interessanten Werte sind die Längen (die Normen) der Vektoren. Allerdings sind die Längen in den meisten Fällen sehr große Zahlen, so dass man dann auf die Angabe der Bitlängen ausweicht.

[10] Sei $w = 32$ die Wortbreite des Rechners und $b_{\min}$ die Bitlänge des kleinsten Eintrages einer ganzzahligen $d \times n$ Matrix. Für den Speicherplatzbedarf s der Matrix gilt: $s \geq n \cdot d \cdot (\lceil b_{\min}/w \rceil \cdot w)$ Bits $= n \cdot d \cdot (\lceil b_{\min}/w \rceil \cdot w)/(8 \cdot 2^{20})$ MByte.

Eintrag ergibt sich ein Speicherplatzbedarf von ca. 22 MByte für R. Tatsächlich ist der gesamte Speicherbedarf nicht so einfach abzuschätzen, denn um die ganzzahligen Matrizen zu speichern, benötigt man noch mehr Platz als eben angenommen (siehe auch Bemerkung 6.1.2 auf Seite 78) und es müssen zusätzlich auch andere Daten gespeichert werden, die allerdings – im Vergleich zu dem Speicherbedarf der Matrizen – nicht sehr ins Gewicht fallen. Zu berücksichtigen wäre auch, dass die Einträge in der Matrix B^s größer als die in der Matrix B sind und B^s entsprechend mehr Speicher belegt. Insgesamt ist somit die untere Schranke für den Bedarf an Speicher im allgemeinen deutlich größer als eben berechnet.

Es ergibt sich aber noch ein weiteres Problem: Die aus praktischer Sicht besonders interessanten Gleitpunktzahlen vom Typ `double` bzw. `quad_float` stellen jeweils „nur“ 11 Bits für die Binärkodierung des Exponenten bereit. Nach Beispiel 4.1.9 auf Seite 42 ergibt sich damit ein Exponentenbereich von $-1022, \ldots, +1023$. Bei der skalierten Segment LLL-Reduktion von Basismatrizen die Einträge mit mehr als ca. 500-Bit Länge haben, kann mit diesem zu kleinen Exponentenbereich nicht mehr richtig skaliert werden. Die Skalierung erfolgt durch Multiplikation der Vektoreinträge mit $2^{e_i}, e_i \in \mathbb{N}$. Das bedeutet eine Addition von e_i zum Exponenten und unsere Experimente zeigen, dass es in den erwähnten Fällen zu dem Problem eines *Exponentenüberlaufes* (siehe Bemerkung 6.3.6 auf Seite 90) kommt, das heißt es existieren i für die $e_i > 1023$ wird. Abhilfe können andere Gleitpunktzahlen-Datentypen schaffen, die einen größeren Exponentenbereich bieten. Eine effiziente Implementierung solcher Datentypen stand uns allerdings nicht zur Verfügung.

In [Koy und Schnorr, 2001a, Abschnitt 4] wird auch die so genannte *iterierte Segment LLL-Reduktion* (ein *divide & conquer* Vorgehen) vorgestellt. Mit diesem Ansatz kann die Laufzeitschranke der Segment LLL-Reduktion asymptotisch verbessert werden. Die experimentellen Ergebnissen zur skalierten Segment LLL-Reduktion lassen uns vermuten, dass eine Implementierung der iterierten Variante erst für Gitterbasen mit sehr großer Dimension sinnvoll ist. Da dann der bereits auf der vorherigen Seite angesprochene Speicherbedarf enorm groß ist und eine effiziente Implementierung der iterierten Segment LLL-Reduktion sehr aufwendig wäre, verzichten wir im Rahmen dieser Diplomarbeit darauf.

1.2.2 Primal/Duale Segmentreduktion

Wir stellen in unserer Arbeit auch die Gitterbasenreduktion in primalen/dualen Segmenten nach Henrik Koy vor [Koy, 2002]. Es werden *primale* und *duale* Segmente in der Matrix R betrachtet, wobei R eine zu der Gitter-Basismatrix B isometrische Matrix ist. Ähnlich wie bei der β-Reduktion werden durch Enumeration in den primalen bzw. dualen Segmenten lokal kürzeste Gittervektoren bestimmt. Das Verfahren ist bei *fester* Segmentlänge in Polynomialzeit durchführbar und liefert eine beweisbar gute Approximation des kürzesten Gittervektors. Wir geben die theoretischen Grundlagen und Resultate der primal/dualen Segmentreduktion lediglich zusammengefasst wieder. Genauere Einzelheiten sind in [Koy, 2002] zu finden. Neben der einfachen primal/dualen Segmentreduktion beschreiben wir auch eine erweiterte Variante aus [Koy, 2002], bei der es zu einer „Überlappung der Segmente“ kommt.

Unsere Ergebnisse

Die Eigenschaften (Laufzeit, Reduktionsgüte) der erweiterten primal/dualen Segmentreduktion werden *experimentell* mit denen der β-Reduktion verglichen.

Um Experimente mit beiden Verfahren zu ermöglichen, wurden die entsprechenden Algorithmen in `latred` integriert. Die erweiterte primal/duale Reduktion wurde dazu erstmals implementiert – wobei sich die Implementierung an einem Vorschlag von Henrik Koy orientiert: die Reduktion der Segmente ist in eine gerade und ungerade „Phase“ organisiert. Wir geben den Algorithmus an. In unserer Implementierung kann auch die geschnittene Aufzählung benutzt werden. Die β-Reduktion ist bereits in der NTL [Shoup, 2001] implementiert und wurde lediglich in `latred` verfügbar gemacht.

Unsere Experimente machen folgendes deutlich: Reduziert man die selbe Gitterbasis mit der β-Reduktion und der primal/dualen Reduktion, so benötigt die β-Reduktion zwar deutlich mehr Zeit im Vergleich zur primal/dualen Segmentreduktion, dafür reduziert sie die Basis aber auch deutlich besser.

1.3 Inhaltliche Gliederung dieser Diplomarbeit

Nach einer kompakten Einführung in die Gittertheorie (Kapitel 2 auf Seite 11) werden in Kapitel 3 auf Seite 25 einige Verfahren zur Gitterbasenreduktion kurz vorgestellt (Längenreduktion, LLL-Reduktion, β-Reduktion, Reduktion in lokalen Koordinaten), die bei den Segmentreduktionsverfahren benutzt werden.

Um die Merkmale von Gleitpunktzahlen-Datentypen vorzustellen und ihre Auswirkungen in der Praxis zu verstehen, beschäftigen wir uns in Kapitel 4 auf Seite 37 mit der Darstellung von Gleitpunktzahlen in Computern und den (numerischen) Eigenschaften der Gleitpunktzahlen-Arithmetik. In diesem Kontext stellen wir die Householder-Reflexion und die Givens-Rotation als numerisch stabile Methoden für die QR-Faktorisierung (Orthogonalisierung) einer Gitter-Basismatrix vor. Zum Abschluss von Kapitel 4 motivieren wir den Gebrauch von Gleitpunktzahlen bei der Gitterbasenreduktion.

Das Kapitel 5 auf Seite 55 widmet sich den beiden Verfahren zur Gitterreduktion in Segmenten: (skalierte) k-Segment LLL-Reduktion und primal/duale Segmentreduktion. Es werden die wesentlichen Eigenschaften der Verfahren wiedergegeben und die entsprechenden Reduktionsalgorithmen werden vorgestellt.

Das Kapitel 6 befasst sich mit der Implementierung des `latred` Programms. Die Routinen für die Orthogonalisierung und für die beiden Segmentreduktionsverfahren werden beschrieben. Die Algorithmen zur Erzeugung von reduzierten/nicht-reduzierten Gitterbasenpaaren werden angegeben.

Abschließend präsentieren wir in Kapitel 7 auf Seite 99 die mit `latred` erzeugten experimentellen Resultate.

1.4 Notationen

Das Symbolverzeichnis ab Seite 129 gibt eine Übersicht einiger in dieser Arbeit verwendeten mathematischen Symbole. Auf die Bedeutung der dort aufgeführten Symbole wird meistens nicht näher eingegangen, da sie allgemein gebräuchlich sind und wir sie deshalb als bekannt voraussetzen. Allerdings verwenden wir auch einige unübliche Notationen, die wir in diesem Abschnitt noch nicht einführen können, da der Kontext, aus dem sie hervorgehen, hier nicht wiedergegeben werden kann. Diese Notationen sind im Symbolverzeichnis mit einer Seitenreferenz – die ein Nachschlagen der Bedeutung ermöglicht – angegeben.

Einige elementare Notationen werden im folgenden vorgestellt. Mit $\mathcal{S}^{m\times n}$ bezeichnen wir die Menge aller $m \times n$ Matrizen mit Einträgen aus der Menge $\mathcal{S}$. Zum Beispiel ist $\mathbb{Z}^{m\times n}$ die Menge aller ganzzahligen $m\times n$ Matrizen. Zur Matrix A bezeichne A^{T} die transponierte und A^{-1} die inverse Matrix. Die Gruppe der invertierbaren, unimodularen $n \times n$ Matrizen über $\mathbb{Z}$ bezeichnen wir mit $\mathsf{SL}_n(\mathbb{Z}) := \{U \in \mathbb{Z}^{n\times n} \mid \det U \in \{\pm 1\}\}$. Für die $n\times n$ Einheitsmatrix schreiben wir E_n. Die Elemente aus $\mathbb{Z}^n, \mathbb{R}^n$, etc. schreiben wir, sofern nicht anders angegeben, als Spaltenvektoren:

1.4.1 Notation (Spaltenvektor) *Mit einem kleinen, kursiv-fettgedruckten, lateinischen Buchstaben bezeichnen wir einen Spaltenvektor.*

Zum Beispiel steht $\boldsymbol{a} \in \mathbb{R}^2$ für den Vektor $\left(\begin{smallmatrix} a_1 \\ a_2 \end{smallmatrix}\right)$, mit $a_1, a_2 \in \mathbb{R}$. Demnach verstehen wir unter dem transponierten Vektor von $\boldsymbol{a}$ einen Zeilenvektor: $\boldsymbol{a}^{\mathsf{T}} = (a_1, a_2) \in \mathbb{R}^2$.

In Kapitel 6 auf Seite 73 bevorzugen wir die Darstellung von Vektoren als Zeilenvektoren (siehe Bemerkung 6.2.1 auf Seite 79). Daher vereinbaren wir eine andere Schreibweise für die Zeilenvektoren einer Matrix A, um sie eindeutig von den Spaltenvektoren von A unterscheiden zu können:

1.4.2 Notation (Zeilenvektor) *Mit einem kleinen, kursiv-fettgedruckten, lateinischen Buchstaben, der einen kleinen Pfeil über sich hat, bezeichnen wir einen Zeilenvektor.*

Zum Beispiel steht $\vec{\boldsymbol{a}} \in \mathbb{R}^2$ für den Vektor (a_1, a_2), mit $a_1, a_2 \in \mathbb{R}$. Demnach verstehen wir unter dem transponierten Vektor von $\vec{\boldsymbol{a}}$ einen Spaltenvektor: $\vec{\boldsymbol{a}}^{\mathsf{T}} = \left(\begin{smallmatrix} a_1 \\ a_2 \end{smallmatrix}\right) \in \mathbb{R}^2$.
Eine Matrix $A \in \mathbb{R}^{2\times 3}$ können wir somit beschreiben durch die Gleichungen

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} = [\boldsymbol{a}_1 \ \boldsymbol{a}_2 \ \boldsymbol{a}_3] = \begin{bmatrix} \vec{\boldsymbol{a}}_1 \\ \vec{\boldsymbol{a}}_2 \end{bmatrix} = (\vec{\boldsymbol{a}}_1, \vec{\boldsymbol{a}}_2)^{\mathsf{T}}, \text{ wobei}$$

$$\boldsymbol{a}_1 = \begin{pmatrix} a_{1,1} \\ a_{2,1} \end{pmatrix}, \ \boldsymbol{a}_2 = \begin{pmatrix} a_{1,2} \\ a_{2,2} \end{pmatrix}, \ \boldsymbol{a}_3 = \begin{pmatrix} a_{1,3} \\ a_{2,3} \end{pmatrix} \text{ und } \vec{\boldsymbol{a}}_1 = (a_{1,1}, a_{1,2}, a_{1,3}), \ \vec{\boldsymbol{a}}_2 = (a_{2,1}, a_{2,2}, a_{2,3}).$$

2 Einführung in die Gittertheorie

Die interessantesten theoretischen Eigenschaften von Gittern sind bereits seit langer Zeit bekannt (Minkowski Sätze, siehe Abschnitt 2.4 auf Seite 20). Aktuelle Fragestellungen betreffen die algorithmische Komplexität gewisser Gitterprobleme bzw. die Anwendungsmöglichkeiten von Gittern in verschiedenen mathematischen Bereichen, etwa der Kryptologie. Daher werden auch heute weiterhin Arbeiten in Zusammenhang mit Gittern und deren Anwendungen publiziert. Dies macht deutlich, dass die Theorie der Gitter ein sehr fruchtbares und weitreichendes Forschungsgebiet ist.

In diesem Kapitel widmen wir uns den theoretischen Grundlagen der Gittertheorie, die für ein Verständnis der weiteren Ausführungen dieser Arbeit nötig sind. Unter anderen führen wir die Notationen und Definitionen für benötigte Begriffe ein. Neben der Vorstellung von Grundbegriffen geben wir auch einige relevante mathematische Fakten wieder. Ergänzende Informationen über die Theorie der Gitter finden sich etwa in [Gruber und Lekkerkerker, 1987; Cohen, 1996; Conway und Sloane, 1998; Schnorr, 1996, 1997, 1998; Fischlin, 2001]. Arbeiten über die Komplexität von Gitterproblemen sind beispielsweise [Cai, 2000; Seifert, 2000].

2.1 Notationen und Fakten aus der Linearen Algebra

2.1.1 Definition (Erzeugnis, Spann) *Sei V ein reeller Vektorraum und A eine nichtleere Teilmenge von V. Man bezeichnet mit* $\mathsf{span}\ A$ *diejenige Teilmenge von V, die aus allen Linearkombinationen von Elementen von A besteht, also*

$$\mathsf{span}\ A := \{\lambda_1 a_1 + \lambda_2 a_2 + \cdots + \lambda_n a_n \mid n \geq 1,\ \lambda_i \in \mathbb{R},\ a_i \in A\}.$$

Man nennt $\mathsf{span}\ A$ *das* Erzeugnis *oder den* Spann *von A und setzt noch* $\mathsf{span}\ \emptyset := \{0\}$. *Für $A = \{a_1, a_2, \ldots, a_n\}$ schreibt man abkürzend* $\mathsf{span}(a_1, a_2, \ldots, a_n) = \mathsf{span}\ A$.

Einen reellen Vektorraum V auf dem ein Skalarprodukt $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ definiert ist nennt man *euklidischen Vektorraum*. Ist auf V eine Norm $\|\cdot\| : V \to \mathbb{R}_+$ definiert, so ist V ein *normierter* Vektorraum. Aus einem euklidischen Vektorraum V (mit beliebigen Skalarprodukt $\langle \cdot, \cdot \rangle$) erhält man die *euklidische Norm* (auch *euklidische Länge* genannt) durch $\|u\| := \sqrt[2]{\langle u, u \rangle}$. Ein euklidischer Vektorraum ist also auch immer ein normierter Vektorraum.

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

2.1.2 Definition (Standard-Skalarprodukt) *Für* $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)^\mathsf{T} \in \mathbb{R}^n$ *und* $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)^\mathsf{T} \in \mathbb{R}^n$ *wird durch*

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle := \boldsymbol{u}^\mathsf{T} \boldsymbol{v} = \sum_{i=1}^{n} u_i v_i$$

das übliche *oder* Standard-*Skalarprodukt auf* $\mathbb{R}^n$ *definiert.*

2.1.3 Definition (ℓ_p-Norm) *Sei* $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)^\mathsf{T} \in \mathbb{R}^n$. *Für* $1 \leq p < \infty$ *ist die* ℓ_p*-Norm definiert als:*

$$\|\boldsymbol{u}\|_p := \left(\sum_{i=1}^{n} |u_i|^p \right)^{\frac{1}{p}}.$$

Für $p = \infty$ *setzt man:*

$$\|\boldsymbol{u}\|_p := \max_{i=1,2,\ldots,n} |u_i|.$$

Die ℓ_1-Norm nennt man auch *Betragsnorm*. In dieser Arbeit wird hauptsächlich das Standard-Skalarprodukt verwendet. Weil in diesem Fall die ℓ_2-Norm und die euklidische Norm identisch sind, kann auf eine Unterscheidung zwischen ihnen verzichtet werden: statt $\|\cdot\|_2$ schreiben wir auch einfach $\|\cdot\|$. Die ℓ_∞-Norm wird auch als *sup-Norm* oder *Maximums-Norm* bezeichnet.

In einem normierten Vektorraum V definiert man den *Abstand* (oder die *Distanz*) d zweier Vektoren $u, v \in V$ durch $\mathsf{d}(u, v) := \|u - v\|$.

Neben der Abstandsmessung zweier Vektoren in einem Vektorraum V ist man auch an dem *Winkel* zwischen von Null verschiedenen Vektoren $u, v \in V$ interessiert. In der linearen Algebra wird gezeigt, dass für alle $u, v \in V$ die Zahl $\frac{\langle u,v \rangle}{\|u\| \cdot \|v\|}$ zwischen -1 und $+1$ liegt (*Cauchy-Schwarzsche*-Ungleichung). Aus der Analysis ist bekannt, dass der Cosinus eine bijektive Abbildung $\cos : [0, \pi] \to [-1, +1]$ herstellt. Deshalb gibt es eine eindeutig bestimmte Zahl $\omega_{u,v}$ mit

$$\cos \omega_{u,v} = \frac{\langle u, v \rangle}{\|u\| \cdot \|v\|} \qquad \text{und} \qquad 0 \leq \omega_{u,v} \leq \pi.$$

In Analogie zur Geometrie in der euklidischen Ebene nennt man $\omega_{u,v}$ den Winkel zwischen den Vektoren u und v (siehe auch **Abbildung 2.1** auf der nächsten Seite).

Zwei Elemente u, v eines euklidischen Vektorraumes heißen *orthogonal* oder senkrecht *zueinander*, wenn $\langle u, v \rangle = 0$. Dafür schreiben wir auch $u \perp v$.

An **Abbildung 2.1** auf der nächsten Seite kann man sich die beiden folgenden Regeln veranschaulichen:

$$u, v \text{ linear abhängig} \iff \omega_{u,v} \in \{0, \pi\}.$$
$$u, v \text{ orthogonal} \iff \omega_{u,v} = \frac{\pi}{2}.$$

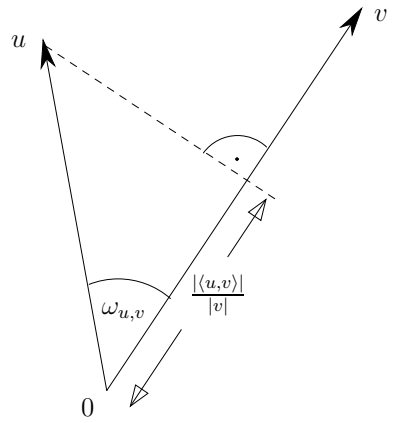


Abbildung 2.1: Winkel zwischen zwei Vektoren in der euklidischen Ebene

2.1.4 Definition (Orthogonales Komplement) *Ist U eine Teilmenge des euklidischen Vektorraumes V, so heißt*

$$U^{\perp} := \{v \in V \mid v \perp u \text{ für alle } u \in U\}$$

das orthogonale Komplement *von U. Statt „$v \perp u$ für alle $u \in U$“ schreibt man auch kurz: $v \perp U$.*

2.1.5 Definition (Orthogonale oder isometrische Abbildung) *Seien V und W euklidische Vektorräume. Eine lineare Abbildung $f : V \to W$ heißt* orthogonal *oder* isometrisch, *wenn*

$$\langle f(u), f(v)\rangle = \langle u, v\rangle$$

für alle $u, v \in V$.

Eine orthogonale oder isometrische Abbildung erhält das Skalarprodukt. Insbesondere euklidische Länge von Vektoren bleiben unverändert.

Die Vorstellung, dass zwei Vektoren eine Fläche aufspannen oder drei Vektoren einen Quadar, wird auf den allgemeinen Fall erweitert. Dazu betrachten wir die Menge aller Linearkombinationen mit Skalaren aus dem reellen Intervall $[0, 1)$:

2.1.6 Definition (Parallelepiped) *Seien $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^n$. Dann nennen wir*

$$P(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) := \left\{ \sum_{i=1}^{n} t_i \boldsymbol{b}_i \;\middle|\; t_i \in [0,1) \subseteq \mathbb{R} \text{ für } i = 1, 2, \ldots, n \right\}$$

das von $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ aufgespannte Parallelepiped *(auch: Parallelotop, Parallelflach).*

2.1.7 Satz (Volumen) *Seien $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^n$ und $\widehat{\boldsymbol{b}}_1, \widehat{\boldsymbol{b}}_2, \ldots, \widehat{\boldsymbol{b}}_n \in \mathbb{R}^n$ die zugehörigen Orthogonalvektoren*[1]*. Dann gilt für das n-dimensionale Volumen vol_n des Parallelepipeds $P(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)$:*

$$\mathsf{vol}_n\left(P(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)\right) = \prod_{i=1}^{n} \|\widehat{\boldsymbol{b}}_i\|.$$

Das n-dimensionale Volumen vol_n bezieht sich stets auf das Standard-Skalarprodukt (siehe Definition 2.1.2 auf Seite 12).

Die geometrische Anschauung des Volumens setzt reelle Vektorräume voraus. Für beliebige Körper ohne Betragsfunktion verallgemeinert man die Volumenfunktion zur vorzeichenbehafteten *Determinantenfunktion* $\mathsf{det}(\cdot)$.

2.2 Notationen und Fakten zu Gittern

2.2.1 Definition (Gitter, Basis, Rang, Basismatrix) *Zu linear unabhängigen Vektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ heißt die Menge*

$$L := \mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) := \left\{ \sum_{i=1}^{n} t_i \boldsymbol{b}_i \;\middle|\; t_i \in \mathbb{Z} \text{ für } i = 1, 2, \ldots, n \right\}$$

Gitter *mit (geordneter)* Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ *und* Rang n. *Die $d \times n$ Matrix*

$$B := [\boldsymbol{b}_1 \; \boldsymbol{b}_2 \; \ldots \; \boldsymbol{b}_n]$$

nennt man Basismatrix. *$\mathcal{L}(B)$ ist das von den Spaltenvektoren der Basismatrix erzeugte Gitter.*[2]

Da die Basisvektoren linear unabhängig sein müssen gilt immer $n \leq d$, denn $d + 1$ Vektoren aus dem $\mathbb{R}^d$ sind stets linear abhängig. Ein wichtiger Spezialfall sind Gitter deren Rang n der Dimension des Raumes d entspricht, also $n = d$. Solche Gitter nennt man *vollständig* oder *volldimensional*, und man unterscheidet dann meist nicht mehr zwischen dem Rang des Gitters und der Dimension des Raumes: Es wird einfach von der *Dimension* $n = d$ des Gitters gesprochen.

Eine alternative Definition eines Gitters ermöglicht Satz 2.2.2 auf der nächsten Seite: Man kann Gitter als diskrete, nicht-triviale additive Untergruppen des $\mathbb{R}^d$ definieren. Es ist dann allerdings zu zeigen, dass es erzeugenden Systeme (Gitterbasen) für das Gitter gibt. Unsere Definition 2.2.1 assoziiert mit einem Gitter gleich seine Gitterbasis, und ist daher für unsere Zwecke „natürlicher“. Wir möchten mit Satz 2.2.2 auf der nächsten Seite also bewußt machen, dass es auch alternative Definitionen von Gittern gibt.

[1] Ein Orthogonalvektor $\widehat{\boldsymbol{b}}_i$ steht senkrecht zu allen vorherigen Vektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1}$ und es gilt $\mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) = \mathsf{span}(\widehat{\boldsymbol{b}}_1, \widehat{\boldsymbol{b}}_2, \ldots, \widehat{\boldsymbol{b}}_n)$. Ein konstruktives Verfahren zur Berechnung von Orthogonalvektoren ist das *Schmidt'sche Orthogonalisierungsverfahren* (siehe Abschnitt 2.3 auf Seite 18).

[2] Manchmal werden in der Literatur die Basisvektoren des Gitters durch Zeilenvektoren repräsentiert, das heißt die Basismatrix ist dann eine $n \times d$ Matrix. Im englischen bezeichnet man Gitter als *lattice*.

2.2.2 Satz *Sei $L \subseteq \mathbb{R}^d$ eine additive Untergruppe. Es gilt:*

$$L \text{ ist diskret (das heißt hat keinen Häufungspunkt)} \iff L \text{ ist ein Gitter.}$$

2.2.3 Definition (Untergitter) *Seien $L, L_{\text{sub}} \subseteq \mathbb{R}^d$ Gitter. L_{sub} heißt* Untergitter *von L, falls $L_{\text{sub}} \subseteq L$. Das Untergitter L_{sub} hat vollen Rang, wenn beide Gitter den gleichen Rang besitzen.*

Ist B eine Basismatrix des Gitters L vom Rang n und A eine Basismatrix des Untergitters L_{sub} vom Rang m, so existiert eine ganzzahlige $n \times m$ Matrix T mit $A = BT$, denn die Spaltenvektoren der Matrix A sind als Gittervektoren von $L = \mathcal{L}(B)$ als ganzzahlige Linearkombinationen der Basisvektoren B darstellbar. Vorausgesetzt T ist eine quadratische Matrix (also $n = m$) hängt die Determinante $\mathsf{det}\, T$ nur von Gitter und Untergitter ab, aber nicht von den gewählten Basen.

2.2.4 Definition (Ganzzahliges Gitter) *Ein Gitter $L \subseteq \mathbb{R}^d$ vom Rang n heißt* ganzzahlig, *wenn es ein Untergitter von $\mathbb{Z}^d$ ist.*

Zu jedem Gitter (mit Dimension $n > 1$) gibt es unendlich viele Basen. Dies ist eine unmittelbare Folgerung aus dem Satz 2.2.5.

2.2.5 Satz *Sei $L := \mathcal{L}(B) \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Eine $d \times n$ Matrix B' ist genau dann eine Basis des Gitters L, wenn eine unimodulare Matrix $T \in \mathsf{SL}_n(\mathbb{Z}) := \{U \in \mathbb{Z}^{n \times n} \mid \det U \in \{\pm 1\}\}$ mit $B' = BT$ existiert.*

Es existieren Gittereigenschaften die unabhängig von der gewählten Basis sind (so genannte *Gitterinvarianten*). Die Gitterdeterminante (Definition 2.2.7) ist eine Gitterinvariante.

Das Ziel der Gitterreduktion ist die algorithmische Transformation einer gegebenen Basis B_1 des Gitters L in eine Basis B_2 von L, wobei B_2 aus kürzeren Vektoren besteht.

Zu den von uns ausschließlich betrachteten ganzzahligen Gittern gibt es nur eine Basismatrix in der *Hermite-Normalform*. Die Berechnung der Hermite-Normalformen zweier Basen B_1 und B_2 ermöglicht eine leichte Überprüfung von $\mathcal{L}(B_1) \stackrel{?}{=} \mathcal{L}(B_2)$.

Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n und B eine Basismatrix zu L. Die positive definite Matrix $B^{\mathsf{T}}B$ heißt *Gram-Matrix* zu B. Die Determinante $\mathsf{det}(B^{\mathsf{T}}B)$ der Gram-Matrix nennt man die *Diskriminante* $\mathsf{disc}(L)$ des Gitters L.

2.2.6 Lemma *Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Dann ist die Determinante der Gram-Matrix $B^{\mathsf{T}}B$ für alle Basismatrizen B des Gitters L konstant.*

Statt der Diskriminanten $\mathsf{disc}(L)$ betrachtet man überlicherweise die *Gitterdeterminante:*

2.2.7 Definition (Gitterdeterminante) *Die* Gitterdeterminante *eines Gitters $L \subseteq \mathbb{R}^d$ ist erklärt als*

$$\mathsf{det}\, L := \big(\mathsf{det}(B^{\mathsf{T}}B)\big)^{\frac{1}{2}} = \sqrt{\mathsf{disc}(L)}$$

für eine Basismatrix B des Gitters.

Die Gitterdeterminante als Wurzel der Gitterdiskriminante ist wohldefiniert, denn nach Lemma 2.2.6 auf der vorherigen Seite ist die Determinante unabhängig von der Wahl der Basis des Gitters.

2.2.8 Definition (Grundmasche) *Die* Grundmasche *der Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ vom Rang n ist das Parallelepiped*

$$\mathcal{P}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) := \left\{ \sum_{i=1}^{n} x_i \boldsymbol{b}_i \;\middle|\; x_i \in [0,1) \text{ für } i = 1, 2, \ldots, n \right\}.$$

Man beachte bei dem folgenden Satz 2.2.9 auch Satz 2.1.7 auf Seite 14, insbesondere die der Definition folgende Anmerkung über die Beziehung des Volumens zu dem Standard-Skalarprodukt.

2.2.9 Satz *Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Für alle Basen $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters L gilt:*

$$\det L = \mathrm{vol}_n\big(\mathcal{P}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)\big).$$

Eine alternative Definition der Gitterderminante lautet

$$\det\big(\mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)\big) = \Big(\det\big(\big[\langle \boldsymbol{b}_i, \boldsymbol{b}_j \rangle\big]_{1 \le i,j \le n}\big)\Big)^{\frac{1}{2}}.$$

Diese Charakterisierung gibt einen Ansatz für Gitter und Gitterreduktion bezüglich eines beliebigen Skalarproduktes $\langle \cdot, \cdot \rangle$. Es existiert eine positiv definite Matrix S mit $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \boldsymbol{u}^{\mathsf{T}} S \boldsymbol{v}$ und eine reguläre Matrix C mit $S = C^{\mathsf{T}} C$. Die Gitterdeterminante wird mit dem Faktor $\det S$ multipliziert.

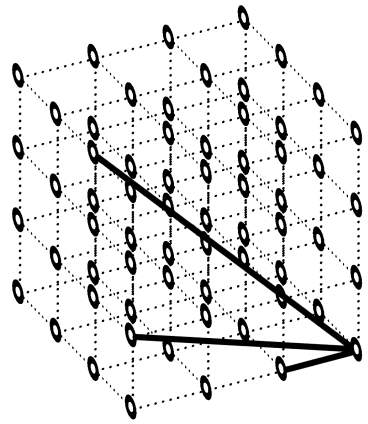
2.2.10 Beispiel Die **Abbildung 2.2** auf der nächsten Seite veranschaulicht einige der vorgestellten Begriffe. Dort ist ein Ausschnitt des Gitters $L = \mathbb{Z}^3 \subseteq R^3$ „im 3-dimensionalen Raum“ dargestellt. In **Abbildung 2.2(a)** ist eine „lange und schiefe“ Basis B von L zu sehen, wohingegen **Abbildung 2.2(b)** eine „kurze und orthogonale“ Basis B' des selben Gitters zeigt. Nach Satz 2.2.5 auf der vorherigen Seite gibt es unimodulare Matrizen T und T^{-1}, die die beiden Basismatrizen ineinander überführen. Aus $B' = BT \iff B^{-1}B' = T$ erhält man eine Matrix T. Eine einfache Berechnung von $\det T$ bestätigt das $T \in \mathsf{SL}_n$. Es gelten also die Gleichungen $B' = BT$ und $B'T^{-1} = B$, das heißt B und B' erzeugen tatsächlich das selbe Gitter. Mit $\boldsymbol{b}_i = (x, y, z)^{\mathsf{T}}$ bzw. $\boldsymbol{b}'_i = (x, y, z)^{\mathsf{T}}$ (vergleiche **Abbildung 2.2(d)** auf der nächsten Seite) erhält man in Matrizenschreibweise:

$$B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \boldsymbol{b}_3] = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 3 & 3 & 1 \end{bmatrix}, \qquad B' = [\boldsymbol{b}'_1\ \boldsymbol{b}'_2\ \boldsymbol{b}'_3] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

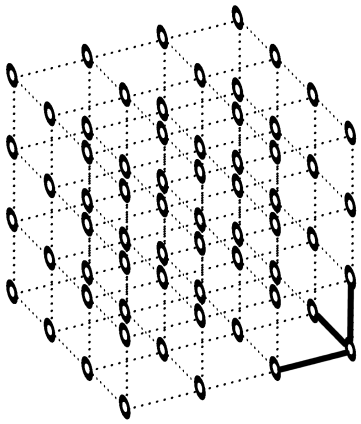
$$B^{-1} = T = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 6 & -3 & 1 \end{bmatrix} \quad \text{und} \quad \det T = 1\,.$$

In **Abbildung 2.2(c)** ist die Grundmasche des Gitters $\mathcal{L}(B') = \mathbb{Z}^3$ grau hervorgehoben. Die geometrische Interpretation der Gitterdeterminante als 3-dimensionales Volumen der Grundmasche ist dort besonders ersichtlich. Sie ist das Volumen des grauen Würfels mit Kantenlänge 1:

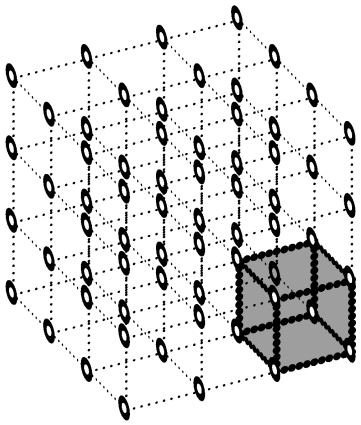
$$\det\big(\mathcal{L}(\boldsymbol{b}_1', \boldsymbol{b}_2', \boldsymbol{b}_3')\big) = \mathsf{vol}_3\big(\mathcal{P}(\boldsymbol{b}_1', \boldsymbol{b}_2', \boldsymbol{b}_3')\big) = 1 \ .$$



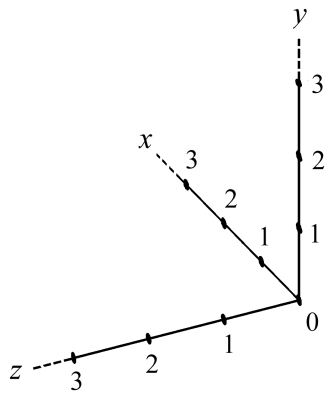
(a) Lange und schiefe Basis des Gitters



(b) Kurze und orthogonale Basis des Gitters



(c) Grundmasche des Gitters



(d) Koordinatensystem

Abbildung 2.2: Ausschnitt des Gitters $\mathbb{Z}^3$ „im 3-dimensionalen Raum“

2.3 Orthogonalsystem

2.3.1 Definition (Orthogonale Projektion) *Zur Gitterbasis* $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ *seien*

$$\pi_i : \mathbb{R}^d \longrightarrow \mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1})^\perp \quad \textit{für } i = 2, 3, \ldots, n$$

die Projektionen *in den zu* $\mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1})$ *orthogonalen Raum.*

Zu jedem Vektor $\boldsymbol{b} \in \mathbb{R}^d$ ist die Linearkombination $\boldsymbol{b} = \boldsymbol{u} + \boldsymbol{v}$ mit $\boldsymbol{u} = \boldsymbol{b} - \pi_i(\boldsymbol{b}) \in \mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1})$ und $\boldsymbol{v} = \pi_i(\boldsymbol{b}) \in \mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1})^\perp$ eindeutig bestimmt. Wegen $\boldsymbol{u} \perp \boldsymbol{v}$ gilt $\|\boldsymbol{b}\|^2 = \|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2$.

Gemäß dem *Schmidt'schen Orthogonalisierungsverfahren* ordnet man jeder Basis $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, …, $\boldsymbol{b}_n \in \mathbb{R}^d$ eines Gitters $L \subseteq \mathbb{R}^d$ ein *Orthogonalsystem* $\widehat{\boldsymbol{b}}_1, \widehat{\boldsymbol{b}}_2, \ldots, \widehat{\boldsymbol{b}}_n \in \mathbb{R}^d$ zu. Setzen wir $\widehat{\boldsymbol{b}}_1 := \pi_1(\boldsymbol{b}_1) := \boldsymbol{b}_1$, so ist die Berechnungsvorschrift, laut dem Verfahren, gegeben durch:

$$\widehat{\boldsymbol{b}}_i := \pi_i(\boldsymbol{b}_i) := \boldsymbol{b}_i - \sum_{j=1}^{i-1} \frac{\langle \boldsymbol{b}_i, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2} \widehat{\boldsymbol{b}}_j \qquad \text{für } i = 2, 3, \ldots, n. \tag{2.1}$$

Allgemeiner gilt:

$$\pi_i(\boldsymbol{b}_k) := \boldsymbol{b}_k - \sum_{j=1}^{i-1} \frac{\langle \boldsymbol{b}_k, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2} \widehat{\boldsymbol{b}}_j \qquad \text{für } 2 \le i \le k \le n.$$

Der Vektor $\widehat{\boldsymbol{b}}_i$ ist die orthogonale Projektion $\pi_i(\boldsymbol{b}_i)$ des Basisvektors $\boldsymbol{b}_i$ auf den Raum $\mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1})^\perp$:

$$\langle \widehat{\boldsymbol{b}}_i, \widehat{\boldsymbol{b}}_j \rangle = \begin{cases} \|\widehat{\boldsymbol{b}}_i\|^2 & \text{falls } i = j \\ 0 & \text{sonst.} \end{cases}$$

Es gilt $\mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_i) = \mathsf{span}(\widehat{\boldsymbol{b}}_1, \widehat{\boldsymbol{b}}_2, \ldots, \widehat{\boldsymbol{b}}_i)$ für $i = 1, 2, \ldots, n$. Die Vektoren $\pi_i(\boldsymbol{b}_i)$, $\pi_i(\boldsymbol{b}_{i+1})$, …, $\pi_i(\boldsymbol{b}_n)$ bilden eine Basis des *projizierten Gitters* $\pi_i(L)$ mit Rang $n - i + 1$. Die Länge $\|\widehat{\boldsymbol{b}}_i\|$ nennt man die *Höhe* des i-ten Basisvektors $\boldsymbol{b}_i$. Man definiert die *Gram-Schmidt-Koeffizienten* $\mu_{i,j}$ als

$$\mu_{i,j} := \begin{cases} 0 & \text{falls } i < j \\ 1 & \text{falls } i = j \\ \frac{\langle \boldsymbol{b}_i, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2} & \text{falls } i > j\,. \end{cases} \tag{2.2}$$

Damit sind die Gleichungen (2.1) auch als

$$\boldsymbol{b}_i = \widehat{\boldsymbol{b}}_i + \sum_{j=1}^{i-1} \mu_{i,j} \widehat{\boldsymbol{b}}_j$$

darstellbar, bzw. in der Matrixschreibweise erhält man

$$\underbrace{\begin{bmatrix}\boldsymbol{b}_1 & \cdots & \boldsymbol{b}_n\end{bmatrix}}_{=B} = \underbrace{\begin{bmatrix}\widehat{\boldsymbol{b}}_1 & \cdots & \widehat{\boldsymbol{b}}_n\end{bmatrix}}_{=:\widehat{B}} \cdot \underbrace{\begin{bmatrix} 1 & \mu_{2,1} & \cdots & \mu_{n-1,1} & \mu_{n,1} \\ 0 & 1 & & \mu_{n-1,2} & \mu_{n,2} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & & 0 & 1 & \mu_{n,n-1} \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}}_{=\left[\mu_{i,j}\right]^{\mathsf{T}}_{1\le i,j\le n}}. \tag{2.3}$$

Es gilt $\det L = \prod_{i=1}^{n} \|\widehat{\boldsymbol{b}}_i\|$, mit anderen Worten, die Gitterdeterminante ist gleich dem Produkt der Höhen der Basisvektoren, denn wegen $\det\left[\mu_{i,j}\right] = 1$ folgt aufgrund Gleichung (2.3):

$$\det(B \cdot B^{\mathsf{T}}) = \left(\det\left[\mu_{i,j}\right]\right)^2 \cdot \det(\widehat{B} \cdot \widehat{B}^{\mathsf{T}}) = \prod_{i=1}^{n} \|\widehat{\boldsymbol{b}}_i\|^2.$$

Wegen $\|\boldsymbol{b}_i\| \ge \|\widehat{\boldsymbol{b}}_i\|$ für $i = 1, 2, \ldots, n$ erhalten wir die Hadamard-Ungleichung:

2.3.2 Satz (Hadamard-Ungleichung) *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ die Basis eines Gitters $L \subseteq \mathbb{R}^d$ vom Rang n. Dann gilt*

$$\det \mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) \le \prod_{i=1}^{n} \|\boldsymbol{b}_i\|\,.$$

Aus der Darstellung (2.3) kann man unmittelbar die sogenannte *QR-Faktorisierung* der $d \times n$ Basismatrix B in eine orthogonale Matrix $Q \in \mathbb{R}^{d\times n}$ und eine obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ ableiten:

$$B = \underbrace{\begin{bmatrix} \dfrac{\widehat{\boldsymbol{b}}_1}{\|\widehat{\boldsymbol{b}}_1\|} & \cdots & \dfrac{\widehat{\boldsymbol{b}}_n}{\|\widehat{\boldsymbol{b}}_n\|} \end{bmatrix}}_{=:Q} \cdot \underbrace{\begin{bmatrix} \|\widehat{\boldsymbol{b}}_1\| & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \|\widehat{\boldsymbol{b}}_n\| \end{bmatrix} \cdot \left[\mu_{i,j}\right]^{\mathsf{T}}_{1\le i,j\le n}}_{=:R}. \tag{2.4}$$

Die QR-Zerlegung der Basismatrix B ist eindeutig, falls

$$R = [r_{i,j}]_{1\le i,j\le n} = \left[\, \|\widehat{\boldsymbol{b}}_i\| \cdot \mu_{j,i} \right]_{1\le i,j\le n} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n-1} & r_{1,n} \\ 0 & r_{2,2} & & r_{2,n-1} & r_{2,n} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{n-1,n-1} & r_{n-1,n} \\ 0 & \cdots & 0 & 0 & r_{n,n} \end{bmatrix} \tag{2.5}$$

positive Diagonaleinträge hat (siehe Satz 4.2.1 auf Seite 44).

Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ die Basis eines Gitters $L \subseteq \mathbb{R}^d$ von Rang n. Die Inverse der Matrix Q der QR-Zerlegung (2.4) liefert eine lineare Abbildung $\Phi : \mathsf{span}(L) \to \mathbb{R}^n$ mit

$$\Phi(\boldsymbol{b}_i) := Q^{-1} \cdot \boldsymbol{b}_i = \left[\mu_{i,1}\|\widehat{\boldsymbol{b}}_1\| \quad \mu_{i,2}\|\widehat{\boldsymbol{b}}_2\| \quad \cdots \quad \mu_{i,n}\|\widehat{\boldsymbol{b}}_n\|\right]^{\mathsf{T}}$$

für $i = 1, 2, \ldots, n$ bzw. $T(B) = R$. Man rechnet leicht nach, dass Φ eine Isometrie ist, das heißt für alle $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$ gilt:

$$\langle \Phi(\boldsymbol{u}), \Phi(\boldsymbol{v}) \rangle = \langle \boldsymbol{u}, \boldsymbol{v} \rangle.$$

Durch die Isometrie Φ betten wir das Gitter $L \subseteq \mathbb{R}^d$ vom Rang n in den $\mathbb{R}^n$ ein, wobei Skalarprodukte und folglich auch Längen (sowie Volumen) erhalten bleiben. Man kann sich daher bei Betrachtungen dieser *geometrischen Invarianten* o.B.d.A. auf volldimensionale Gitter beschränken.

2.4 Sukzessive Minima und Hermite-Konstanten

Auf Minkowski[3] geht die Definition der sukzessiven Minima eines Gitters zurück:

2.4.1 Definition (Sukzessive Minima) *Zu einem Gitter $L \subseteq \mathbb{R}^d$ vom Rang n heißen die Werte*

$$\lambda_i(L) := \inf \left\{ r > 0 \;\middle|\; \begin{array}{l} \textit{Es existieren linear unabhängige} \\ \boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_i \in L \textit{ mit } \max \|\boldsymbol{a}_i\| \leq r \end{array} \right\}$$

für $i = 1, 2, \ldots, n$ die sukzessiven Minima *von L.*

Das Minimum $\lambda_i(L)$ wird angenommen, da L diskret ist. Wir schreiben kurz λ_i statt $\lambda_i(L)$, wenn das Gitter L aus dem Kontext hervorgeht. Es gilt $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, insbesondere ist λ_1 die Länge des kürzesten, nicht-trivialen Gittervektors (der Ursprung $\mathbf{0}$ ist trivialerweise mit der Länge 0 der kürzeste Gitterpunkt). Als Norm $\|L\|$ des Gitters L wird das erste sukzessive Minimum $\lambda_1(L)$ bezeichnet. Wir sagen, Gittervektoren $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \in L$ *realisieren die sukzessiven Minima*, wenn diese linear unabhänggig sind und $\|\boldsymbol{a}_i\| = \lambda_i(L)$ für $i = 1, 2, \ldots, n$.

Gittervektoren $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \in L$, welche die sukzessiven Minima realisieren, bilden im allgemeinen keine Basis des Gitters, sondern erzeugen lediglich ein Untergitter. Für $n = 1$ trivialerweise und für $n = 2$ existiert stets eine Basis, deren Vektoren die sukzessiven Minima realisieren [Vallée, 1991].

Mit $\lambda_{i,\infty}$ werden die sukzessiven Minima des Gitters bezüglich der Maximumsnorm $\|\boldsymbol{v}\|_\infty := \max_{1 \leq i \leq d} |v_i|$ bezeichnet. Für die Maximumsnorm $\|L\|_\infty$ des Gitters L gilt:

2.4.2 Satz (Minkowski, 1896) *Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Dann gilt*

$$\lambda_{1,\infty}(L) \leq (\mathsf{det}\, L)^{\frac{1}{n}}.$$

[3] Der Titel seines Buches „Geometrie der Zahlen“ [Minkowski, 1896] hat diesem Bereich der Mathematik seinen Namen gegeben.

Diese Schranke ist scharf, wie man am Beispiel $\mathbb{Z}^d$ sieht. Da allgemein $\|\boldsymbol{v}\| \leq \sqrt{d} \cdot \|\boldsymbol{v}\|_\infty$ für $\boldsymbol{v} \in \mathbb{R}^d$ gilt, liefert Minkowskis Abschätzung für Gitter $L \subseteq \mathbb{R}^d$ vom Rang n folgende obere Schranke für das erste sukzessiven Minimum:

$$\lambda_1(L) \leq \sqrt{d} \cdot (\det L)^{\frac{1}{n}}. \tag{2.6}$$

Im Fall von Gittern mit Rang n gilt

$$\lambda_1(L) \leq \sqrt{\gamma_n} \cdot (\det L)^{\frac{1}{n}} \tag{2.7}$$

für die so genannte Hermite-Konstante γ_n, die sich auf das Standardskalarprodukt und zugehörige Norm bezieht:

2.4.3 Definition (Hermite-Konstante) *Die* Hermite-Konstanten γ_n *sind definiert als:*

$$\gamma_n := \sup \left\{ \frac{\lambda_1(L)^2}{(\det L)^{\frac{2}{n}}} \;\middle|\; L \subseteq \mathbb{R}^d \text{ Gitter vom Rang } n \right\}.$$

Das Supremum wird angenommen, die entsprechenden Gitter heißen *kritisch*. Aus historischen Gründen, die mit quadratische Formen zusammenhängen, betrachtet man das Quadrat des Quotienten $\lambda_1(L)/(\det L)^{\frac{1}{n}}$. Die Division durch $(\det L)^{\frac{2}{n}}$ dient der Normierung, denn skalieren wir das Gitter $L \subseteq \mathbb{R}^n$ mit $\alpha > 0$, gilt $\lambda_1(\alpha L) = \alpha \cdot \lambda_1(L)$ und $\det(\alpha L) = \alpha^n \det L$. Aus diesem Grund genügt es, das Supremum über Gitter mit $\lambda_1 = 1$ oder alternativ $\det L = 1$ zu nehmen und, da Determinante und sukzessive Minima invariant unter isometrischer Abbildung sind, kann man sich ferner auf Gitter vollen Ranges beschränken.

Aus Satz 2.4.2 auf der vorherigen Seite folgt $\gamma_n \leq n$. Die Hermite-Konstanten $\gamma_2, \gamma_3, \gamma_4, \gamma_5$ wurden in der zweiten Hälfte des 19. Jahrhunderts von Korkine und Zolotareff bestimmt, Blichfeldt (1935) hat $\gamma_6, \gamma_7, \gamma_8$ ermittelt. In **Tabelle 2.1** sind die bekannten Werte angegeben. Weitere Werte sind bis heute nicht bekannt [Conway und Sloane, 1998]. Asymptotisch gelten folgende Schranken für die Hermite-Konstanten [Nguyen und Stern, 2000]:

$$\frac{n}{2\pi e} + \frac{\ln(\pi n)}{2\pi e} + o(1) \leq \gamma_n \leq \frac{1.744n}{2\pi e}\left(1 + o(1)\right). \tag{2.8}$$

Gitter-Rang n	1	2	3	4	5	6	7	8
Hermite-Konstante $(\gamma_n)^n$	1	$\frac{4}{3}$	2	4	8	$\frac{2^6}{3}$	2^6	2^8

Tabelle 2.1: Hermite-Konstanten γ_n für $n = 1, 2, \ldots, 8$

2.4.4 Satz (Minkowski-Ungleichung) *Sei $L \subseteq \mathbb{R}^d$ ein Gitter von Rang n. Dann gilt:*

$$\prod_{i=1}^{n} \lambda_i(L) \leq (\gamma_n)^{\frac{n}{2}} \cdot \det L.$$

Für kritische Gitter $L \subseteq \mathbb{R}^d$ folgt aus Satz 2.4.4, dass $\lambda_1 = \lambda_2 = \cdots = \lambda_n$, denn einerseits gilt $\lambda_1^n = (\gamma_n)^{\frac{n}{2}} \cdot \det L$ und wegen $\lambda_i \geq \lambda_1$ ist $\prod_{i=1}^{n} \lambda_i \geq \lambda_1^n$.

2.4.5 Satz *Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Dann gilt:*

$$\det L \leq \prod_{i=1}^{n} \lambda_i(L).$$

Beweis von 2.4.5:

Seien $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \in L$ Gittervektoren, welche die sukzessiven Minima realisieren. Diese Vektoren erzeugen ein Untergitter von L, so dass

$$\det \mathcal{L}(\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n) \geq \det L.$$

Der Satz folgt aus der Ungleichung von Hadamard, Satz 2.3.2 auf Seite 19:

$$\prod_{i=1}^{n} \lambda_i(L) = \prod_{i=1}^{n} \|\boldsymbol{a}_i\| \geq \det \mathcal{L}(\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n) \geq \det L.$$

❏

2.4.6 Satz (Erster Satz von Minkowski, 1893) *Sei $L \subseteq \mathbb{R}^n$ volldimensionales Gitter, $\|\cdot\|$ eine beliebige Norm und $B := \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| \leq 1\}$. Falls $\mathrm{vol}_n(B) > 2^n \cdot \det L$, existiert ein Gittervektor $\boldsymbol{b} \in (L \cap B) \setminus \{\boldsymbol{0}\}$.*

Der Satz 2.4.5 liefert für das Produkt der sukzessiven Minima bezüglich der Euklidischen Norm die untere Schranke $\det L \leq \prod_{i=1}^{n} \lambda_i$, während für die sukzessiven Minima in beliebiger Norm gilt [Gruber und Lekkerkerker, 1987, Kap. 2, §9.1]:

2.4.7 Satz (Zweiter Satz von Minkowski, 1907) *Sei $L \subseteq \mathbb{R}^n$ ein volldimensionales Gitter und $\lambda_{i,\|\cdot\|}$ die sukzessiven Minima bezüglich einer beliebigen Norm $\|\cdot\|$. Sei $B := \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| \leq 1\}$ und $V := \mathrm{vol}_n(B)$. Dann gilt:*

$$\frac{\det L}{n!} \leq \frac{V}{2^n} \cdot \prod_{i=1}^{n} \lambda_{i,\|\cdot\|} \leq \det L.$$

Als Zweiter Satz von Minkowski wird oftmals nur die obere Schranke bezeichnet.

2.5 Duales Gitter

2.5.1 Definition (Duales Gitter, selbstdual) *Das zu einem Gitter $L \subseteq \mathbb{R}^d$* duale Gitter *ist erklärt als:*

$$L^* := \left\{ \boldsymbol{u} \in \mathsf{span}(L) \;\middle|\; \boldsymbol{u}\boldsymbol{b}^\mathsf{T} \in \mathbb{Z} \text{ für alle } \boldsymbol{b} \in L \right\}$$

Ein Gitter L mit $L^* = L$ heißt selbstdual *oder* unimodular.

Das duale Gitter heißt auch *reziprokes* oder *polares* Gitter, in der Regel betrachtet man das duale Gitter in Verbindung mit den Standardskalarprodukt, weshalb dann statt $\boldsymbol{u}\boldsymbol{b}^\mathsf{T} \in \mathbb{Z}$ äquivalenterweise $\langle \boldsymbol{u}, \boldsymbol{b} \rangle \in \mathbb{Z}$ geschrieben wird. Ein triviales Beispiel eines selbstdualen Gitters ist $\mathbb{Z}^d$.

2.5.2 Satz *Sei $L \subseteq \mathbb{R}^d$ ein Gitter mit Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$. Dann gilt:*

a) $\mathsf{det}\, L^* = (\mathsf{det}\, L)^{-1}$,

b) $(L^*)^* = L$,

c) Eine Basis des dualen Gitters L^* bilden $\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_n^* \in \mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)$ mit

$$\boldsymbol{b}_i^* \cdot \boldsymbol{b}_j^\mathsf{T} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{sonst.} \end{cases}$$

d) L und das zugehörige duale Gitter L^* haben den gleichen Rang.

Man nennt $\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_n^* \in \mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n) \subseteq \mathbb{R}^d$ die zugehörige (eindeutig bestimmte) *duale Basis* zu $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$. Für die Basismatrizen B und B^* gilt $B^* \cdot B^\mathsf{T} = E_n$, im Fall volldimensionaler Gitter ist die Basismatrix der dualen Basis gegeben als $B^* := (B^{-1})^\mathsf{T}$. Ein Gitter vollen Ranges ist genau dann selbstdual, wenn es eine orthogonale Basismatrix des Gitters gibt.

Die Beziehungen zwischen sukzessiven Minima des primalen und des dualen Gitters heißen *Transferschranken*. Aus der Definition der Hermite-Konstanten $\lambda_1(L) \cdot (\mathsf{det} L)^n \leq \sqrt{\gamma_n}$ und $\lambda_1(L^*) \cdot (\mathsf{det}\, L^*)^n \leq \sqrt{\gamma_n}$ folgt in Verbindung mit $\mathsf{det}\, L \cdot \mathsf{det}\, L^* = 1$, dass $\lambda_1(L) \cdot \lambda_1(L^*) \leq \gamma_n$. Allgemeiner hat Mahler 1939 die untere Schranke $1 \leq \lambda_i(L) \cdot \lambda_{n-i+1}(L^*)$ bewiesen, und die beste, bekannte obere Schranke für $\lambda_i(L) \cdot \lambda_{n-i+1}(L^*)$ geht auf Banaszczyk 1993 zurück [Cai, 1999]:

2.5.3 Satz (Mahler 1939, Banaszczyk 1993) *Sei $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n. Dann gibt es eine Konstante c, so dass für $i = 1, 2, \ldots, n$ gilt:*

$$1 \leq \lambda_i(L) \cdot \lambda_{n-i+1}(L^*) \leq cn.$$

Die unter Schranke ist scharf, die obere bis auf einen konstanten Faktor.

2.6 Algorithmische Gitterprobleme und ihre Komplexität

Wir stellen die wichtigsten Gitterprobleme vor.

Shortest Vector Problem (SVP) Dieses zentrale Gitterproblem wurde erstmals von DIRICHLET im Jahre 1842 formuliert: Gegeben sei eine Basis $[\boldsymbol{b}_1\ \boldsymbol{b}_1\ \ldots\ \boldsymbol{b}_n]$ des Gitters $L \subseteq \mathbb{Q}^d$, finde $\boldsymbol{v} \in L$ mit $\|\boldsymbol{v}\| = \lambda_1(L)$. Bei der approximativen Variante des SVP fragt man nach einem Vektor $\boldsymbol{v} \in L$ ($\boldsymbol{v} \neq \boldsymbol{0}$) dessen Norm durch einen Approximationsfaktor beschränkt ist: $\|\boldsymbol{v}\| \leq f(n)\lambda_1(L)$. Mit SVP_∞ bezeichnet man das Problem bezüglich der Maximums-Norm.

Closest Vector Problem (CVP) Gegeben sei eine Basis des Gitters $L \subseteq \mathbb{Q}^d$ und ein Vektor $\boldsymbol{v} \in \mathbb{R}^d$, finde einen Gittervektor $\boldsymbol{u} \in L$, der den Abstand $\mathsf{d}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|$ minimiert.

Shortest Basis Problem (SBP) Der Begriff einer kürzesten Basis ist nicht kanonisch. Die gängigste SBP Variante lautet: Finde eine Gitterbasis $[\boldsymbol{b}_1\ \boldsymbol{b}_1\ \ldots\ \boldsymbol{b}_n]$ für $L \subseteq \mathbb{Q}^d$ mit

$$\max_{1\leq i\leq n} \|\boldsymbol{b}_i\| = \min_{\text{Basen } [\boldsymbol{b}'_1\ \boldsymbol{b}'_1\ \ldots\ \boldsymbol{b}'_n] \text{ von } L} \max_{1\leq i\leq n} \|\boldsymbol{b}'_i\| \,.$$

AJTAI zeigte in [Ajtai, 1998], dass das SVP unter *randomisierter* Reduktion NP-hart ist. Die NP-Härte des SVP unter *deterministischen* (Karp-) Reduktionen ist weiterhin ein offenes Problem.

Bereits länger bekannt ist die NP-Härte des CVP und SBP. Genaueres dazu findet man in [Cai, 1999].

Viele Ergebnisse der letzten Jahre über die Komplexität von Gitterproblemen wurden von der Arbeit [Ajtai, 1998] inspiriert: Ihr folgten einige Vereinfachungen des Beweises und Verschärfungen hinsichtlich der NP-Härte approximativer Varianten; eine Übersicht dieser Thematik geben die Arbeiten [Cai, 1999, 2000]. Mit der algorithmischen Komplexität von Gitterproblemen und Grenzen der Approximierbarkeit beschäftigt sich [Seifert, 2000].

3 Reduktionsbegriffe und Gitterbasenreduktion

Wir stellen Reduktionsbegriffe und Algorithmen vor, die bei der Segmentreduktion (Kapitel 5 auf Seite 55) benutzt werden.

Auf die numerischen Probleme von Computer-Programmen zur Gitterreduktion kommen wir in diesem Kapitel noch nicht zu sprechen. Die Gitterbasenreduktion in Segmenten stellen wir in Kapitel 5 (Seite 55) vor, nachdem in Kapitel 4 (Seite 37) die benötigten Grundlagen aus der Numerik behandelt worden sind.

3.1 Reduktion von Gitterbasen

Ziel der Gitterreduktion ist es, eine reduzierte Basis für ein gegebenes Gitter zu finden. Die Vektoren der Basis sollen weitgehend

- orthogonal sein
- und die Länge der Basisvektoren den sukzessiven Minima entsprechen.

Allerdings existiert zu einem gegebenen Gitter im allgemeinen keine Basis, deren Vektoren orthogonal sind oder deren Längen den sukzessiven Minima entsprechen.

Um diese Zielsetzung zu motivieren, betrachte die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ vom Rang n. Für das zugehörige Orthogonalsystem gilt für $i = 1, 2, \ldots, n$:

$$\|\widehat{\boldsymbol{b}}_i\|^2 \leq \|\boldsymbol{b}_i\|^2 = \|\widehat{\boldsymbol{b}}_i\|^2 + \sum_{j=1}^{i-1} |\mu_{ij}|^2 \|\widehat{\boldsymbol{b}}_j\|^2. \tag{3.1}$$

Mit Hilfe der Höhen erhalten wir untere Schranken für die sukzessiven Minima:

3.1.1 Lemma *Sei $L \subseteq \mathbb{R}^d$ ein Gitter von Rang n. Dann gilt für jede Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters und $i = 1, 2, \ldots, n$:*

$$\min_{i \leq j \leq n} \|\widehat{\boldsymbol{b}}_j\| \leq \lambda_i(L).$$

Die im weiteren vorgestellten Reduktionsbegriffe wie LLL-, HKZ- oder die BKZ-Reduktion basieren auf dem Orthogonalsystem der Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$. Der Vektor

$\pi_i(\boldsymbol{b}_i) = \widehat{\boldsymbol{b}}_i$ soll ein kurzer Vektor im projizierten Gitter[1] $\pi_i(L)$ bzw. einem Untergitter von $\pi_i(L)$ sein. Zusätzlich sollen die Basisvektoren paarweise weitgehend orthogonal sein, also die Absolutbeträge der Gram-Schmidt-Koeffizienten klein sein (Längenreduktion).

3.2 Längenreduktion

Das Ziel der Längenreduktion ist die Transformation der Basis in eine solche mit weitgehend orthogonalen Vektoren. Der Begriff der Längenreduktion (engl. *weakly reduced*) geht auf HERMITE zurück.

3.2.1 Definition (Längenreduktion) *Die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ ist* längenreduziert, *wenn $|\mu_{i,j}| \leq \frac{1}{2}$ für $1 \leq j < i \leq n$.*

3.2.2 Satz *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine längenreduzierte Basis eines Gitter $L \subseteq \mathbb{R}^d$ vom Rang n. Dann gilt für $i = 1, 2, \ldots, n$:*

$$\|\boldsymbol{b}_i\|^2 \leq \|\widehat{\boldsymbol{b}}_i\|^2 + \sum_{j=1}^{i-1} \mu_{i,j}^2 \|\widehat{\boldsymbol{b}}_j\|^2 \leq \|\widehat{\boldsymbol{b}}_i\|^2 + \tfrac{1}{4} \sum_{j=1}^{i-1} \|\widehat{\boldsymbol{b}}_j\|^2.$$

Eine gegebene Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ samt Gram-Schmidt-Koeffizienten $\mu_{i,j}$ läßt sich in eine längenreduzierte Basis des gleichen Gitters mit **Algorithmus 3.1** transformieren.

Algorithmus 3.1 Längenreduktion

Eingabe: Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eines Gitters $L \subset \mathbb{Z}^d$; zugehörige $\mu_{i,j}$

```
1: for i = 1, 2, ..., n do
2:    /* Längenreduziere b_i */
3:    for j = i − 1, i − 2, ..., 1 do
4:       t ← ⌈μ_{i,j}⌋
5:       b_i ← b_i − t · b_j
6:       μ_{i,j} ← μ_{i,j} − t
7:    end for
8: end for
```

Ausgabe: Längenreduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L; veränderte $\mu_{i,j}$

Betrachten wir die Schleife in **Algorithmus 3.1**. Die Anweisung $\boldsymbol{b}_i^{\text{neu}} := \boldsymbol{b}_i - t \cdot \boldsymbol{b}_j$ bewirkt

$$\mu_{i,j}^{\text{neu}} = \frac{\langle \boldsymbol{b}_i^{\text{neu}}, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2} = \frac{\langle \boldsymbol{b}_i - t \cdot \boldsymbol{b}_j, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2} = \underbrace{\frac{\langle \boldsymbol{b}_i, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2}}_{=\mu_{i,j}} - t \cdot \underbrace{\frac{\langle \boldsymbol{b}_j, \widehat{\boldsymbol{b}}_j \rangle}{\|\widehat{\boldsymbol{b}}_j\|^2}}_{=\mu_{j,j}=1} = \mu_{i,j} - t$$

[1] Das projizierte Gitter ist auf Seite 18 erklärt.

und somit $|\mu_{i,j}^{\text{neu}}| \leq \frac{1}{2}$ nach Wahl von $t = \lceil \mu_{ij} \rfloor$. Die Höhen der Basisvektoren bleiben erhalten. Anhand des Orthogonalsystems

$$\left[\widehat{\boldsymbol{b}}_1 \quad \cdots \quad \widehat{\boldsymbol{b}}_n\right] \cdot \begin{bmatrix} 1 & \mu_{2,1} & \cdots & \mu_{n-1,1} & \mu_{n,1} \\ 0 & 1 & & \mu_{n-1,2} & \mu_{n,2} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & & 0 & 1 & \mu_{n,n-1} \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}$$

ist zu erkennen, dass die Längenreduktion von $\boldsymbol{b}_i$ die bereits längenreduzierten Vektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1}$ bzw. deren Gram-Schmidt-Koeffizienten unverändert bleibt.

3.2.3 Lemma *Die gegebene Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{Z}^n$ kann in $O\left(n^3\right)$ arithmetischen Schritten längenreduziert werden.*

3.3 LLL-Reduktion

Der LLL-Reduktionsbegriff geht auf A.K. Lenstra, H.W. Lenstra und L. Lovász [Lenstra, Lenstra und Lovász, 1982] zurück:

3.3.1 Definition (LLL-Reduktion) *Die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ ist* LLL-reduziert *mit Parameter $\delta \in (\frac{1}{4}, 1]$, wenn*

L1: $|\mu_{i,j}| \leq \frac{1}{2}$ *für* $1 \leq j < i \leq n$.

L2: $\delta \cdot \|\pi_i(\boldsymbol{b}_i)\|^2 \leq \|\pi_i(\boldsymbol{b}_{i+1})\|^2$ *für* $i = 1, 2, \ldots, n-1$.

Der LLL-Reduktionsbegriff wurde in [Lenstra, Lenstra und Lovász, 1982] für $\delta = \frac{3}{4}$ behandelt. Die Forderung L1 entspricht der Längenreduktion und Ungleichung L2 lautet

$$\delta \cdot \|\widehat{\boldsymbol{b}}_i\|^2 \leq \mu_{i+1,i}^2 \cdot \|\widehat{\boldsymbol{b}}_i\|^2 + \|\widehat{\boldsymbol{b}}_{i+1}\|^2. \tag{3.2}$$

Ungleichung (3.2) wird oftmals als *Lovàsz-Bedingung* bezeichnet.

Viele Abschätzungen für LLL-reduzierte Basen $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ basieren auf der Ungleichung

$$\|\widehat{\boldsymbol{b}}_i\|^2 \leq \alpha \|\widehat{\boldsymbol{b}}_{i+1}\|^2 \leq \cdots \leq \alpha^{j-i} \|\widehat{\boldsymbol{b}}_j\|^2$$

mit $\alpha = (\delta - \frac{1}{4})^{-1}$ und $i \leq j$, die unmittelbar aus L2 in Verbindung mit $\mu_{i+1,i}^2 \leq \frac{1}{4}$ (wegen L1) folgt. In [Schnorr, 1996, 1997] werden die Resultate aus [Lenstra, Lenstra und Lovász, 1982] für $\delta \in (\frac{1}{4}, 1]$ angegeben:

3.3.2 Satz *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine mit Parameter $\delta \in (\frac{1}{4}, 1]$ LLL-reduzierte Basis eines Gitters $L \subseteq \mathbb{R}^d$. Für $\alpha := (\delta - \frac{1}{4})^{-1}$ gilt:*

a) $\alpha^{1-i}\lambda_i(L)^2 \leq \|\widehat{\boldsymbol{b}}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

b) $\|\boldsymbol{b}_i\|^2 \leq \alpha^{n-1}\lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

c) $\|\boldsymbol{b}_i\|^2 \leq \alpha^{j-1}\|\widehat{\boldsymbol{b}}_j\|^2$ *für* $1 \leq i \leq j \leq n$.

d) $\|\boldsymbol{b}_1\|^2 \leq \alpha^{\frac{n-1}{2}}(\det L)^{\frac{1}{n}}$.

e) $\prod_{i=1}^{n} \|\boldsymbol{b}_i\|^2 \leq \alpha^{\frac{n(n-1)}{2}}(\det L)^2$.

Der erste Basisvektor approximiert λ_1 bis auf einen exponentiellen Faktor: $\|\boldsymbol{b}_1\| \leq 2^{(n-1)/2}\lambda_1$ für $\delta = 3/4$.

Eine gegebene Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ läßt sich in eine LLL-reduzierte Basis des gleichen Gitters mit **Algorithmus 3.2** transformieren. Der Reduktionsalgorithmus geht auf L. Lovász zurück, dessen Verfahren auf einen Polynomialzeit-Algorithmus von Lenstra [Lenstra, 1983] zur Integer-Programmierung bei fester Dimension basiert.

Algorithmus 3.2 LLL-Reduktion

Eingabe: Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eines Gitters $L \subset \mathbb{Z}^d$;
Reduktionsparameter $\delta \in (\frac{1}{4}, 1]$

```
1: k ← 2
2: berechne Gram-Schmidt-Koeffizienten μ_{1,1}, μ_{2,1}, μ_{2,2}  /* Orthogonalisiere */
3: while k ≤ n do
4:    /* Schleifeninvariante: b_1, b_2, ..., b_{k-1} ist LLL-reduziert */
5:    längenreduziere b_k  /* siehe Algorithmus 3.1 auf Seite 26 */
6:    for j = 1, 2, ..., k − 1 do
7:       berechne bzw. aktualisiere μ_{k,j}  /* Orthogonalisiere */
8:    end for
9:    if (δ · ||b̂_{k-1}||^2 ≤ μ^2_{k,k-1} · ||b̂_{k-1}||^2 + ||b̂_k||^2) then
10:      k ← k + 1  /* erhöhe Stufe k */
11:   else
12:      swap(b_k, b_{k-1})  /* vertausche b_k und b_{k-1} */
13:      k ← max(k − 1, 2)  /* erniedrige Stufe k */
14:   end if
15: end while
```

Ausgabe: Mit Parameter δ LLL-reduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L

3.3.3 Satz (Lenstra, Lenstra und Lovász, 1982) *Die gegebene Basis* $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ *eines Gitters* $L \subseteq \mathbb{R}^d$ *kann für* $\frac{1}{4} < \delta < 1$ *in LLL-reduzierte Basis des Gitters in* $O\left(dn^3 \log M\right)$ *arithmetischen Schritte überführt werden, wobei* $M := \max_i \|\boldsymbol{b}_i\|^2$.

Der Beweis von Satz 3.3.3 auf der vorherigen Seite kann in [Lenstra, Lenstra und Lovász, 1982] nachgelesen werden. Man beachte, dass die polynomielle Laufzeit des Verfahrens nur für $\frac{1}{4} < \delta < 1$ gezeigt worden ist. Der LLL-Algorithmus reduziert die Basisvektoren mit größer werdenden δ besser, allerdings steigt entsprechend die Laufzeit in der Praxis. Bei ganzzahligen Eingabe-Basismatrizen werden die arithmetischen Operationen auf ganzen Zahlen der Bitlänge $O(d \cdot \log M)$ ausgeführt.

3.4 HKZ- und β-Reduktion

Zwar kann eine LLL-reduzierte Basis eines gegebenen Gitters effizient berechnet werden, der kürzeste Vektor wird nur bis auf einen exponentiellen Faktor approximiert. Der folgende Reduktionsbegriff geht auf HERMITE (1851), als auch KORKINE und ZOLOTAREFF [Korkine und Zolotareff, 1873] zurück:

3.4.1 Definition (HKZ-Reduktion) *Die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ ist* HKZ-reduziert, *wenn*

H1: $|\mu_{i,j}| \leq \frac{1}{2}$ *für* $1 \leq j < i \leq n$.

H2: $\|\pi_i(\boldsymbol{b}_i)\| = \lambda_1(L_i)$ *mit* $L_i := \mathcal{L}\big(\pi_i(\boldsymbol{b}_i), \pi_i(\boldsymbol{b}_{i+1}), \ldots, \pi_i(\boldsymbol{b}_n)\big)$ *für* $i = 1, 2, \ldots, n$.

Der Basisvektor $\boldsymbol{b}_1$ realisiert das erste sukzessive Minimum λ_1, ist also ein kürzester, nicht-trivialer Gittervektor. Das projizierte Gitter L_i ist auf auf Seite 18 eingeführt worden.

3.4.2 Lemma *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine Basis eines Gitters $L \subseteq \mathbb{R}^d$ ein Gitter vom Rang n und $L_i := \mathcal{L}\big(\pi_i(\boldsymbol{b}_i), \pi_i(\boldsymbol{b}_{i+1}), \ldots, \pi_i(\boldsymbol{b}_n)\big)$. Dann gilt für $i = 1, 2, \ldots, n$:*

$$\lambda_1(L_i) \leq \lambda_i(L).$$

Beweis von 3.4.2:

Seien $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_i \in L$ Gittervektoren, welche die ersten i sukzessiven Minima realisieren:

$$\|\boldsymbol{a}_1\| \leq \|\boldsymbol{a}_2\| \leq \cdots \leq \|\boldsymbol{a}_i\| \leq \lambda_i(L).$$

Einer dieser Vektoren $\boldsymbol{a}_k$ ist unter der Projektion π_i ungleich Null, denn sonst wäre die Dimension von $\mathsf{span}(\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_i) < i$. Wir erhalten $\lambda_1(L_i) \leq \|\pi_i(\boldsymbol{a}_k)\| \leq \lambda_i(L)$.

Eine HKZ-reduzierte Basis hat nachstehende Eigenschaften [Lagarias, Lenstra und Schnorr, 1990]:

3.4.3 Satz (Lagarias, Lenstra und Schnorr, 1990) *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine HKZ-reduzierte Basis eines Gitters $L \subseteq \mathbb{R}^d$ vom Rang n. Dann gilt:*

a) $\|\boldsymbol{b}_i\|^2 \le \frac{i+3}{4} \cdot \lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

b) $\frac{4}{i+3} \cdot \lambda_i(L)^2 \le \|\boldsymbol{b}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

c) $\prod_{i=1}^{n} \|\boldsymbol{b}_i\|^2 \le \left(\gamma_n^n \prod_{i=1}^{n} \frac{i+3}{4}\right)(\mathsf{det}\, L)^2$.

3.4.4 Satz (Lagarias, Lenstra und Schnorr, 1990) *Sei* $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ *eine HKZ-reduzierte Basis eines Gitters* $L \subseteq \mathbb{R}^d$ *vom Rang* n. *Dann gilt für das Orthogonalsystem* $\widehat{\boldsymbol{b}}_1, \widehat{\boldsymbol{b}}_2, \ldots, \widehat{\boldsymbol{b}}_n$*:*

a) $\lambda_1(L)^2 \le i^{1+\ln i} \cdot \|\widehat{\boldsymbol{b}}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

b) $\|\boldsymbol{b}_i\|^2 \le i^{2+\ln i} \cdot \|\widehat{\boldsymbol{b}}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

Die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters L heißt *dual HKZ-reduziert*, wenn die Vektoren $\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_n^*$ der dualen Basis in umgekehrter Reihenfolge eine HKZ-Basis des dualen Gitters L^* bilden [Lagarias, Lenstra und Schnorr, 1990].

3.4.5 Satz (Lagarias, Lenstra und Schnorr, 1990) *Sei* $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ *eine dual HKZ-reduzierte Basis eines Gitters* $L \subseteq \mathbb{R}^d$ *vom Rang* n. *Dann gilt für* $i = 1, 2, \ldots, n$*:*

$$\lambda_1(L) \le \gamma_i \cdot \|\widehat{\boldsymbol{b}}_i\|.$$

Von Schnorr [Schnorr, 1987] stammt eine Weiterentwicklung eines Verfahren von Kannan, der eine gegebene Basis in eine HKZ-reduzierte des gleichen Gitters transformiert. Kernstück des Algorithmus' ist der Aufzählungsalgorithmus. Die Laufzeit ist in der Größenordnung $dn(n^{O(n)} + n^2) \cdot \log M$ mit $M := \max_i \|\boldsymbol{b}_i\|^2$.

Schnorr [Schnorr, 1987, 1988] hat eine Hierarchie von Reduktionsbegriffen eingeführt. Ausgangspunkt dazu ist die folgende Beobachtung: Im Fall einer mit Parameter $\delta = 1$ LLL-reduzierte Basis gilt

$$\|\pi_i(\boldsymbol{b}_i)\| = \lambda_1\big(\pi_i(\mathcal{L}(\boldsymbol{b}_i, \boldsymbol{b}_{i+1}))\big) = \lambda_1\big(\mathcal{L}(\pi_i(\boldsymbol{b}_i), \pi_i(\boldsymbol{b}_{i+1}))\big)$$

für $i = 1, 2, \ldots, n-1$. Unter der Projektion π_i realisiert $\boldsymbol{b}_i$ das erste sukzessive Minimum des von den beiden projizierten Vektoren $\pi_i(\boldsymbol{b}_i)$ und $\pi_i(\boldsymbol{b}_{i+1})$ erzeugten Gitters. Schnorr hat dieses Konzept von 2 auf $\beta \ge 2$ Vektoren verallgemeinert:

3.4.6 Definition (β-Reduktion) *Die Basis* $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ *eines Gitters* $L \subseteq \mathbb{R}^d$ *vom Rang* n *ist* β*- oder* blockreduziert *mit Blockweite* $\beta \in \{2, 3, \ldots, n\}$*, wenn*

B1: $|\mu_{i,j}| \le \frac{1}{2}$ *für* $1 \le j < i \le n$.

B2: $\|\pi_i(\boldsymbol{b}_i)\| = \lambda_1(L_{i,\beta})$ *mit* $L_{i,\beta} := \mathcal{L}\big(\pi_i(\boldsymbol{b}_i), \pi_i(\boldsymbol{b}_{i+1}), \ldots, \pi_i(\boldsymbol{b}_{\min(i+\beta-1,n)})\big)$.

β-reduzierte Basen mit $\beta = 2$ sind LLL-reduziert mit $\delta = 1$, bei $\beta = n$ sind die Basen HKZ-reduziert. Einen Polynomialzeit-Algorithmus zur β-Reduktion kennt man nicht, allerdings Verfahren, die sich in den der Praxis für moderate Blockweiten bewährt haben. Die β-Reduktion nennt man auch *BKZ-Reduktion* (Blockweise-Korkine-Zolotareff-Reduktion). Es gilt [Schnorr, 1994]:

3.4.7 Satz (Schnorr, 1994) *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine β-reduzierte Basis eines Gitters $L \subseteq \mathbb{R}^d$. Dann gilt:*

a) $\|\boldsymbol{b}_i\|^2 \leq \frac{i+3}{4} \cdot \gamma_\beta^{\frac{2(n-1)}{\beta-1}} \cdot \lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

b) $\frac{4}{i+3} \cdot \gamma_\beta^{\frac{\beta-1}{2(i-1)}} \cdot \lambda_i(L)^2 \leq \|\boldsymbol{b}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

c) $\|\widehat{\boldsymbol{b}}_i\|^2 \leq \gamma_\beta^{\frac{2(n-i)}{\beta-1}} \cdot \lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

Für eine β-reduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters L gilt insbesondere

$$\|\boldsymbol{b}_1\| \leq \gamma_\beta^{\frac{n-1}{\beta-1}} \cdot \lambda_1. \tag{3.3}$$

Ritter [Ritter, 1997] berücksichtigt den Parameter δ und gibt eine Verallgemeinerung von Definition 3.4.6 auf der vorherigen Seite an:

3.4.8 Definition ((β, δ)-Reduktion) *Die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eines Gitters $L \subseteq \mathbb{R}^d$ vom Rang n ist (β, δ)- oder δ-*blockreduziert *mit Blockweite $\beta \in \{2, 3, \ldots, n\}$ und $\delta \in (0, 1]$, wenn*

D1: $|\mu_{i,j}| \leq \frac{1}{2}$ *für* $1 \leq j < i \leq n$.

D2: $\delta \cdot \|\pi_i(\boldsymbol{b}_i)\| \leq \lambda_1(L_{i,\beta})$ *mit* $L_{i,\beta} := \mathcal{L}\big(\pi_i(\boldsymbol{b}_i), \pi_i(\boldsymbol{b}_{i+1}), \ldots, \pi_i(\boldsymbol{b}_{\min(i+\beta-1,n)})\big)$.

3.4.9 Satz (Ritter, 1997) *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ eine (β, δ)-blockreduzierte Basis eines Gitters $L \subseteq \mathbb{R}^d$. Dann gilt:*

a) $\|\widehat{\boldsymbol{b}}_i\|^2 \leq \frac{1}{\delta} \cdot \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{2(n-1)}{\beta-1}} \cdot \lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

b) $\|\boldsymbol{b}_i\|^2 \leq \frac{1}{\delta} \cdot \frac{i+3}{4} \cdot \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{2(n-1)}{\beta-1}} \cdot \lambda_i(L)^2$ *für* $i = 1, 2, \ldots, n$.

c) $\lambda_i(L)^2 \leq \frac{1}{\delta} \cdot \frac{i+3}{4} \cdot \left(\frac{\gamma_\beta}{\delta}\right)^{\frac{2(i-1)}{\beta-1}} \cdot \|\boldsymbol{b}_i\|^2$ *für* $i = 1, 2, \ldots, n$.

Algorithmus 3.3 (β, δ)-Blockreduktion

Eingabe: Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eines Gitters $L \subset \mathbb{Z}^d$;
Blockweite $\beta \in \{2, 3, \ldots, n\}$; Reduktionsparameter $\delta \in (0, 1]$

$j \leftarrow n-1$
$z \leftarrow 0$
/* z zählt die Positionen j, an denen die Forderung

$$\delta \cdot \|\widehat{\boldsymbol{b}}_j\| \leq \lambda_1\Big(\mathcal{L}\big(\pi_j(\boldsymbol{b}_j), \pi_j(\boldsymbol{b}_{j+1}), \ldots, \pi_j(\boldsymbol{b}_k)\big)\Big) \tag{3.4}$$

erfüllt ist. Beachte das Gleichung (3.4) der Forderung D2 in Definition 3.4.8 auf der vorherigen Seite entspricht. */
while $z < n-1$ **do**
  $j \leftarrow j+1$
  if $j = n$ **then**
    $j \leftarrow 1$
  end if
  /* j wird zyklisch über die Zahlen $1, 2, \ldots, n-1$ variiert. $j = n$ wird ausgelassen, da die Bedingung (3.4) für $j = n$ stets gilt. */
  $k \leftarrow \min(j + \beta - 1, n))$
  $c_j \leftarrow \|\pi_j(\boldsymbol{b}_j)\|$ /* $c_j = \|\widehat{\boldsymbol{b}}_j\|$ */
  $(\bar{c}_j, \boldsymbol{b}_j^{\text{neu}}) \leftarrow$ `ENUM(j,k)`
  /* `ENUM(j,k)` bestimmt die ganzzahligen Koeffizienten $(u_j, u_{j+1}, \ldots, u_k)$ der Darstellung eines Vektors $\boldsymbol{b}_j^{\text{neu}} = \sum_{i=j}^{k} u_i \boldsymbol{b}_i$ mit $\bar{c}_j := \|\pi_j(\boldsymbol{b}_j^{\text{neu}})\| = \lambda_1\Big(\mathcal{L}\big(\pi_j(\boldsymbol{b}_j), \pi_j(\boldsymbol{b}_{j+1}), \ldots, \pi_j(\boldsymbol{b}_k)\big)\Big)$. Genaueres findet man in [Schnorr und Euchner, 1994; Hörner, 1994; Schnorr und Hörner, 1995; Ritter, 1997]. */
  $k \leftarrow \min(k+1, n)$
  if $\delta \cdot c_j > \bar{c}_j$ **then**
    /* Es gilt: $\delta \cdot \|\widehat{\boldsymbol{b}}_j\| > \lambda_1\Big(\mathcal{L}\big(\pi_j(\boldsymbol{b}_j), \pi_j(\boldsymbol{b}_{j+1}), \ldots, \pi_j(\boldsymbol{b}_k)\big)\Big)$ */
    ergänze
    $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{j-1}, \boldsymbol{b}_j^{\text{neu}}$ zu einer Basis
    $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{j-1}, \boldsymbol{b}_j^{\text{neu}}, \boldsymbol{b}_{j+1}^{\text{neu}}, \ldots, \boldsymbol{b}_k^{\text{neu}}, \boldsymbol{b}_{k+1}, \boldsymbol{b}_{k+2}, \ldots, \boldsymbol{b}_n$ von L
    längenreduziere $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{j-1}, \boldsymbol{b}_j^{\text{neu}}, \boldsymbol{b}_{j+1}^{\text{neu}}, \ldots, \boldsymbol{b}_k^{\text{neu}}$ /* siehe **Algorithmus 3.1** auf Seite 26 */
    $z \leftarrow 0$
  else
    /* Es gilt: $\delta \cdot \|\widehat{\boldsymbol{b}}_j\| \leq \lambda_1\Big(\mathcal{L}\big(\pi_j(\boldsymbol{b}_j), \pi_j(\boldsymbol{b}_{j+1}), \ldots, \pi_j(\boldsymbol{b}_k)\big)\Big)$ */
    längenreduziere $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_k$ /* siehe **Algorithmus 3.1** auf Seite 26 */
    $z \leftarrow z+1$
  end if
end while

Ausgabe: (β, δ)-blockreduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L

Der **Algorithmus 3.3** auf der vorherigen Seite berechnet – sofern er terminiert – eine (β, δ)-reduzierte Basis zu dem von $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ erzeugten Gitter $L \subseteq \mathbb{R}^d$. Die Variable z zählt zyklisch, das heißt auf n folgt 1, die aufeinanderfolgenden Indizes, an denen Bedingung D2 erfüllt ist. Wir setzen $z := 0$, falls das Gitter $L_{j,\beta}$ verändert wird. Da parallel die Gitterbasis längenreduziert wird, ist im Fall $z = n$ die Basis (β, δ)-reduziert. Der Kern des Verfahrens ist die `ENUM(j,k)`-Routine[2] zur Berechnung der ganzzahligen Minimalstelle des Ausdrucks

$$\bar{c}_j(u_j, u_{j+1}, \ldots, u_k) = \left\| \pi_j \left(\sum_{i=j}^{k} u_i \boldsymbol{b}_i \right) \right\|^2 .$$

Der `ENUM`-Algorithmus, der mittels vollständiger Aufzählung (in einem Suchbaum) einen kürzesten Gittervektor bestimmt, wird in [Schnorr und Euchner, 1994] vorgestellt. Eine abgeschwächte Variante davon ist die geschnittene Aufzählung. Sie basiert auf einer Volumenheuristik die auf Gauss zurückgeht und hat den Vorteil das der Suchbaum nicht vollständig durchlaufen wird, mehr dazu in [Hörner, 1994; Schnorr und Hörner, 1995].

Der Aufwand um einen kürzesten Gittervektor zu bestimmen ist *sehr* groß, daher sind die erwähnten Aufzählungsverfahren nur bedingt praktikabel. Mit Aufzählungsalgorithmen beschäftigen sich auch [Fincke und Pohst, 1985; Ritter, 1997; Backes, 1998].

3.5 Reduktion in lokalen Koordinaten

Wir erläutern das Konzept der Reduktion in lokalen Koordinaten, welches auf die Idee der *Semi-Reduktion* von Schönhage zurückgeht [Schönhage, 1984].

Eine Isometrie (auch orthogonale Abbildung, siehe Abschnitt 2.1 auf Seite 11) Q ist ein Isomorphismus, der durch eine orthogonale Matrix beschrieben wird. Die Inverse Funktion Q^{-1} ist ebenfalls eine orthogonale Abbildung. Eine Isometrie bzw. orthogonale Matrix erhält das Skalarprodukt und die (euklidische) Länge:

$$\begin{aligned} \langle \boldsymbol{u}, \boldsymbol{v} \rangle &= \langle Q\boldsymbol{u}, Q\boldsymbol{v} \rangle \\ \|\boldsymbol{u}\| &= \|Q\boldsymbol{u}\|. \end{aligned}$$

Um die Basisvektoren $B = [\boldsymbol{b}_1 \ \boldsymbol{b}_2 \ \ldots \ \boldsymbol{b}_n] \in \mathbb{R}^{d \times n}$ eines Gitters zu reduzieren, kann man also auch eine zu B isometrische Matrix R reduzieren.

Im folgenden gehen wir davon aus, dass der Reduktionsbegriff auf dem Skalarprodukt basiert. Der genaue Reduktionsbegriff ist nicht von wesentlicher Bedeutung, es könnte sich etwa um eine LLL-Reduktion handeln. Um eine Basismatrix $B = QR$ (vergleiche Gleichung (2.4) auf Seite 19) eines Gitters $L \subseteq \mathbb{R}^d$ zu reduzieren, können wir auch die Basis $R = Q^{\mathsf{T}}B = Q^{-1}B$ des Gitters $\mathcal{L}(Q^{-1}B)$ für eine Isometrie $Q : \mathbb{R}^d \mapsto \mathbb{R}^d$ reduzieren und die Transformationsmatrix T auf B anwenden:

[2] Enumeration, engl. Aufzählung.

$$\begin{array}{ccc} B & \xrightarrow{\text{Reduktion}} & BT = QRT \\ {\scriptstyle Q^{-1}}\big\downarrow & & \big\uparrow{\scriptstyle Q} \\ R = Q^{-1}B & \xrightarrow{\text{Reduktion}} & RT \end{array}$$

Die Gleichung (2.4) auf Seite 19 liefert eine Isometrie Q, so dass die Basismatrix R des Gitters $\mathcal{L}(Q^{-1}B)$ eine obere Dreieckmatrix ist (vergleiche auch Gleichung (2.5) auf Seite 19):

$$R = [\boldsymbol{r}_1\ \boldsymbol{r}_2\ \ldots\ \boldsymbol{r}_n] = \begin{bmatrix} \|\widehat{\boldsymbol{b}}_1\| & * & \cdots & * \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & * \\ 0 & \cdots & 0 & \|\widehat{\boldsymbol{b}}_n\| \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

Wegen

$$\mathsf{span}(\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_i) = Q\big(\mathsf{span}(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_i)\big) \quad \text{für } i = 1, 2, \ldots, n,$$

gilt für alle s, k mit $1 \le s \le s+k \le n$:

$$\mathcal{L}\big(\pi_s(\boldsymbol{b}_{s+1}), \pi_s(\boldsymbol{b}_{s+2}), \ldots, \pi_s(\boldsymbol{b}_{s+k})\big) = Q\Big(\mathcal{L}\big(\pi_s(\boldsymbol{r}_{s+1}), \pi_s(\boldsymbol{r}_{s+2}), \ldots, \pi_s(\boldsymbol{r}_{s+k})\big)\Big).$$

Die Basismatrix von $\mathcal{L}\big(\pi_{s+1}(\boldsymbol{r}_{s+1}), \pi_{s+1}(\boldsymbol{r}_{s+2}), \ldots, \pi_{s+1}(\boldsymbol{r}_{s+k})\big)$ hat die Form

$$R^{s,k} := \underbrace{\begin{bmatrix} 0 & 0 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 \\ r_{s+1,s+1} & r_{s+1,s+2} & \cdots & r_{s+1,s+k} \\ 0 & r_{s+2,s+2} & \cdots & r_{s+2,s+k} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & r_{s+k,s+k} \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}}_{=[\pi_{s+1}(\boldsymbol{r}_{s+1})\ \ \pi_{s+1}(\boldsymbol{r}_{s+2})\ \ \ldots\ \ \pi_{s+1}(\boldsymbol{r}_{s+k})]} \in \mathbb{R}^{d\times k}.$$

Angenommen, wir wollen die Basismatrix $R^{s,k}$ reduzieren. Weil die ersten s Einträge und die letzten $d-(s+k)$ Einträge aller Gittervektoren Null sind, können wir diese Koordinaten entfernen und es genügt, die Reduktion nur auf die Matrix

$$\sigma_{s,k}\big(R^{s,k}\big) := \begin{bmatrix} r_{s+1,s+1} & r_{s+1,s+2} & \cdots & r_{s+1,s+k} \\ 0 & r_{s+2,s+2} & \cdots & r_{s+2,s+k} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & r_{s+k,s+k} \end{bmatrix} \in \mathbb{R}^{k\times k}.$$

anzuwenden. Die Darstellung der Basisvektoren $\pi_{s+1}(\boldsymbol{r}_{s+1}), \pi_{s+1}(\boldsymbol{r}_{s+2}), \ldots, \pi_{s+1}(\boldsymbol{r}_{s+k})$ unter der $\sigma_{s,k}$-Abbildung nennt man *Darstellung in lokalen Koordinaten.* Ist $T \in \mathsf{SL}_k(\mathbb{Z})$ die Transformationsmatrix, also $\sigma_{s,k}\big(R^{s,k}\big)\cdot T$ ist reduziert, so ist auch $\big(R^{s,k}\big)\cdot T$ reduziert.

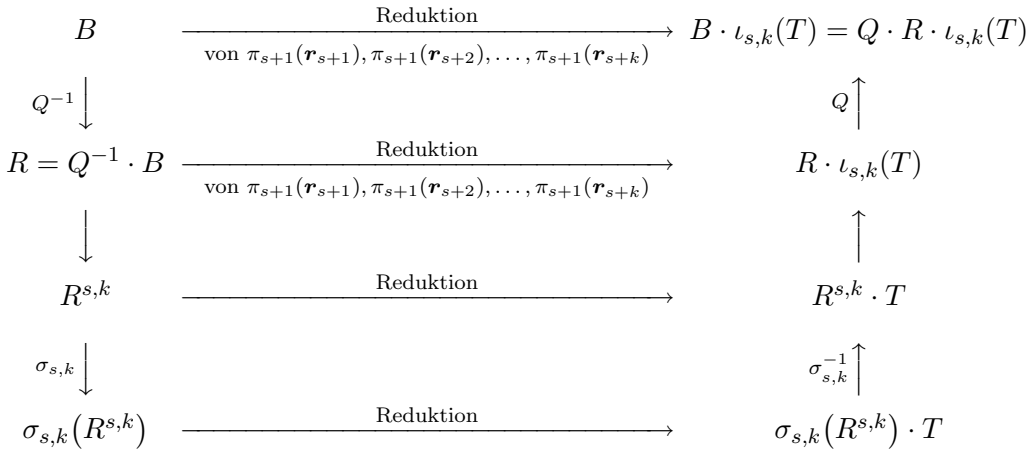
Für die QR-Faktorisierung einer Basismatrix $B = QR \in \mathbb{R}^{d\times n}$ haben wir folgende Darstellung:

$$R = \begin{bmatrix} r_{1,1} & & & \cdots & & & r_{1,n} \\ 0 & \ddots & & & & & \\ & & \underbrace{\begin{bmatrix} r_{s+1,s+1} & r_{s+1,s+2} & \cdots & r_{s+1,s+k} \\ 0 & r_{s+2,s+2} & \cdots & r_{s+2,s+k} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & r_{s+k,s+k} \end{bmatrix}}_{=\sigma_{s,k}\left(R^{s,k}\right)} & & \vdots \\ & & & & \ddots & & \\ 0 & & & \cdots & 0 & r_{n,n} \end{bmatrix} \in \mathbb{R}^{n\times n}. \tag{3.5}$$

Die Darstellung in lokalen Koordinaten ermöglicht das effiziente Reduzieren von Basisvektoren, da auf einer verkleinerten Teilmatrix von R gerechnet wird. Zu beachten ist, dass auch bei einer ganzzahligen Eingabe-Basismatrix $[\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{d\times n}$ die Einträge der Matrix $\sigma_{s,k}\big(R^{s,k}\big)$ reell sind, und daher müssen Berechnungen damit – in Computern – durch Gleitpunktzahlen-Arithmetik approximiert werden. Dies ist für große Zahlen auf der einen Seite ein Vorteil, denn das Rechnen mit den Gleitpunktzahlen ist bei großen Zahlen deutlicher schneller, als das Rechnen mit exakten ganzen Zahlen. Auf der anderen Seite ergeben sich Probleme durch inhärente Rundungsfehler. Näheres zur Gleitpunktzahlen-Arithmetik findet man in Kapitel 4 auf Seite 37.

Die Reduktion in lokalen Koordinaten ist ein Bestandteil der Gitterbasenreduktion in Segmenten (siehe Kapitel 5 auf Seite 55).

Um zum Beispiel die Vektoren $\pi_{s+1}(\boldsymbol{r}_{s+1}), \pi_{s+1}(\boldsymbol{r}_{s+2}), \ldots, \pi_{s+1}(\boldsymbol{r}_{s+k})$ einer gegebenen Gitterbasis $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n]$ zu reduzieren, kannn man wie folgt vorgehen:



Dabei bezeichne $\iota_{s,k}(T)$ die Einbettung der $k \times k$ Matrix T in die $n \times n$ Einheitsmatrix E_n ab Position $(s+1, s+1)$:

$$\iota_{s,k}(T) := \begin{bmatrix} E_s & 0 & 0 \\ 0 & T & 0 \\ 0 & 0 & E_{n-s-k} \end{bmatrix}.$$

4 Numerische Betrachtungen

Im Rechner wird für die Darstellung eines Zahlenwertes nur ein begrenzter Speicherplatz reserviert, so dass die irrationalen Zahlen (und die meisten rationalen Zahlen) im Rechner approximiert werden müssen. Daher führt die Arithmetik bei numerischen Verfahren, beim Einsatz von Rechnern, zwangsläufig zu *Darstellungsfehlern* (auch *Rundungsfehler* genannt). Rechenverfahren, die die an einem Punkt aufgetretenen Rechenfehler beim Weiterrechnen verstärken, heißen *instabil*. In der *numerischen Mathematik* beschäftigt man sich mit den Eigenschaften von Verfahren hinsichtlich solcher Instabilitäten. Die dort eingesetzten Methoden zur Modellierung der numerischen Qualität von Verfahren sind meistens heuristische Hilfsmittel zur approximativen Analyse von Rundungsfehlern und ihrer Fortpflanzung, da exakte Techniken zu aufwendig und inpraktikabel sind. Wir verzichten auf eine detaillierte Besprechung der Fehleranalyse und Fehlerfortpflanzung und geben lediglich die für unsere Zwecke relevanten Resultate wieder.

Ein Verständnis der numerischen Auswirkungen von Gleitpunktzahlen bei Rechenverfahren in Computern soll Abschnitt 4.1 ermöglichen. Wir stellen dort die wesentlichen Überlegungen an, welche die Implementierungsstrategie in Kapitel 6 auf Seite 73 motivieren: Eine wichtige Eigenschaft unseres Reduktionsprogramms ist, dass es mit unterschiedlichen (auch nicht-Standard) Gleitpunktzahlen-Datentypen rechnen kann. Kapitel 7 auf Seite 99 macht deutlich, das die Verwendung unterschiedlicher Datentypen beachtenswerte Auswirkungen in der Praxis hat.

Aufgrund Abschnitt 2.3 auf Seite 18 und des in Kapitel 3 auf Seite 25 behandelten Stoffes wird deutlich, dass die Orthogonalisierung ein elementarer Bestandteil von Algorithmen zur Gitterbasenreduktion ist. Unser Augenmerk liegt deshalb auf den numerischen Eigenschaften von Verfahren zur Orthogonalisierung (auch als QR-Faktorisierung bekannt) einer Matrix. Die Diskussion von einigen dazu einsetzbaren Verfahren und ihrer numerischen Eigenschaften erfolgt in Abschnitt 4.2 auf Seite 44. Auch die Reduktion in Segmenten, die in Kapitel 5 auf Seite 55 behandelt wird, basiert auf der Vefügbarkeit von R.

Der Abschnitt 4.3 auf Seite 53 widmet sich speziell den Problemen, die bei der Gitterbasenreduktion mit Rechnern auftreten.

4.1 Gleitpunktzahlen und Rundungsfehler

Wir beschreiben im folgenden wie die interne Representation von Zahlen in den meisten heutigen Rechnern geschieht, und wir diskutieren wie sich die endliche Zahlendar-

stellung auf die Rechengenauigkeit auswirkt. Eine ausführlichere Besprechung dieser Thematik, sowie die meisten hier weggelassenen Beweise, findet man in [Oevel, 1996, Abschnitt 3.1].

Da man sowohl „große“ Zahlen wie etwa πe^{30} als auch „kleine“ Zahlen wie etwa $e^{-20}/40!$ mit derselben Güte approximieren will, verwenden numerische Computersprachen bzw. Mikroprozessoren in der Regel die so genannte *Gleitpunktdarstellung*.

4.1.1 Definition (Gleitpunktdarstellung) *Die* Gleitpunktdarstellung *von* $x \in \mathbb{R} \setminus \{0\}$ *zur* Basis $\beta \in \{2, 3, 4, \dots\}$ *ist*

$$x = \mathsf{sign}(x)\, z \times \beta^{\,e}$$

mit dem Vorzeichen[1] $\mathsf{sign}(x) \in \{\pm 1\}$, *der* Mantisse $z \in [\beta^{-1}, 1) \subset \mathbb{R}$ *und dem* Exponenten $e \in \mathbb{Z}$.

Die Mantisse z gibt den Zahlenwert an, der Exponent e charakterisiert die Größenordnung der Zahl. In der Praxis wählt man für die Basis β meist eine der Zahlen 2, 8, 10 oder 16. Um die Zahl x eindeutig darzustellen, *normalisiert* man die Mantisse durch die Forderung $\frac{1}{\beta} \le z < 1$ für $z \neq 0$.

4.1.2 Bemerkung (Eindeutigkeit bei gegebener Basis) *Bei gegebener Basis β ist die Gleitpunktdarstellung eindeutig:*

$$e = \lfloor \log_\beta(|x| + 1) \rfloor \ , \ z = |x| \beta^{\,-e}.$$

Zur Darstellung einer Zahl $x \mathrel{\widehat{=}} (\pm, z, \beta)$ wird die Mantisse z nur bis auf eine vorgegebene „Stellenzahl“ N im Rechner gespeichert. Dazu werden folgendermaßen Ziffern der Darstellung definiert:

4.1.3 Lemma (Zerlegung der Mantisse in Ziffern) *Zu $z \in [\beta^{-1}, 1) \cap \mathbb{R}$ existieren* Ziffern $z_1, z_2, \dots \in \{0, 1, \dots, \beta - 1\}$ *mit $z_1 \neq 0$, so dass $z = \sum_{i=1}^{\infty} z_i\, \beta^{-i}$.*

Die Eindeutigkeit der Zifferndarstellung nach Lemma 4.1.3 ist nur in dem Ausnahmefall verletzt, wo die Ziffernfolge den „Grenzwert“ $\beta - 1$ hat (zum Beispiel $0.12999\ldots = 0.13000\ldots$ zur Basis $\beta = 10$). Daher wird folgende Vereinbarung getroffen:

4.1.4 Konvention (Vermeidung unzulässiger Ziffernfolgen) *Eine Ziffernfolge der Form $z_{n_0+1} = z_{n_0+2} = \cdots = \beta - 1$ (mit $z_{n_0} < \beta - 1$) ist nicht zulässig und wird umgeschrieben:*

$$(z_1, \dots, z_{n_0}, \beta - 1, \beta - 1, \dots) \mathrel{\widehat{=}} (z_1, \dots, z_{n_0} + 1, 0, 0, \dots) \, .$$

4.1.5 Lemma (Eindeutigkeit der Ziffernfolge) *Mit Konvention 4.1.4 ist die Ziffernfolge für $z \in [\beta^{-1}, 1) \subset \mathbb{R}$ eindeutig.*

[1] Eigentlich ist $\mathsf{sign}(x) \in \{\pm 1, 0\}$. Bei der Gleitpunktdarstellung verzichtet man auf die 0, das heißt das Vorzeichen ist entweder positiv oder negativ.

Bei *N-stelliger Darstellung* werden nur die ersten N Ziffern gespeichert, der Rest wird entweder abgeschnitten oder gerundet. Dieser Prozeß soll allgemein als *Rundung* bezeichnet werden:

4.1.6 Definition (Rundung) *Die Rundungsfunktion* $\mathsf{rd} : [\beta^{-1}, 1) \longrightarrow [\beta^{-1}, 1) \cap \mathbb{Q}$ *wird definiert durch:*

$$\begin{aligned} \mathsf{rd}(z) &= \mathsf{rd}(z_1\,\beta^{-1} + \cdots + z_{N-1}\,\beta^{-(N-1)} + z_N\,\beta^{-N} + z_{N+1}\,\beta^{-(N+1)} + \cdots) \\ &:= \begin{cases} z_1\,\beta^{-1} + \cdots + z_{N-1}\,\beta^{-(N-1)} + z_N\,\beta^{-N} & \text{(Abschneiden)} \\ z_1\,\beta^{-1} + \cdots + z_{N-1}\,\beta^{-(N-1)} + \tilde{z}_N\,\beta^{-N} & \text{(Rundung)}\,. \end{cases} \end{aligned}$$

Hierbei entsteht die letzte Ziffer $\tilde{z}_N$ *bei der Rundung durch den* Rundungsprozeß:

$$\tilde{z}_N := \begin{cases} z_N & \textit{falls } \sum_{i>N} z_i\,\beta^{-i} < \frac{1}{2}\,\beta^N \text{ (Abrunden)} \\ z_N + 1 & \textit{falls } \sum_{i>N} z_i\,\beta^{-i} \geq \frac{1}{2}\,\beta^N \text{ (Aufrunden)}\,. \end{cases}$$

Beim Aufrunden entstehen eventuell Überträge (wie beispielweise in $\mathsf{rd}(4.99997) = 5.0000$), falls z_N den größtmöglichen Wert $z_N = \beta - 1$ angenommen haben sollte und durch Aufrundung zu der nicht zulässigen Ziffer $\tilde{z}_N = \beta$ führt. Dieser Übertrag ergibt durch die Umdefinitionen $\tilde{z}_N \mapsto 0$, $z_{N-1} \mapsto z_{N-1} + 1$ neue Ziffern der Darstellung, wobei dieser Prozeß rekursiv fortzusetzen ist, falls wiederum $z_{N-1} = \beta - 1$ etc. gilt.

Man beachte, dass man mit $z - \sum_{i=1}^{N} z_i\beta^{-i} = \sum_{i>N} z_i\beta^{-i}$ und der für jedes n geltenden Ungleichung $0 \leq z - (z_1\beta^{-1} + z_2\beta^{-2} + \cdots + z_n\beta^{-n}) < \beta^{-n}$ erhält, dass die Abschätzung $0 \leq \sum_{i>N} z_i\beta^{-i} < \beta^{-N}$ gilt. Für den Rundungsprozeß nach unserer Definition 4.1.6 folgt somit

$$|z - \mathsf{rd}(z)| = \begin{cases} \sum_{i>N} z_i\,\beta^{-i} < \beta^{-N} & \text{(Abschneiden)} \\ \left|(z_N - \tilde{z}_N)\beta^{-N} + \sum_{i>N} z_i\,\beta^{-i}\right| \leq \frac{1}{2}\,\beta^{-N} & \text{(Rundung)}\,. \end{cases}$$

Mit $z \geq \beta^{-1}$ ergibt sich das Endresultat

$$\frac{|z - \mathsf{rd}(z)|}{z} \begin{cases} < \beta^{1-N} & \text{(Abschneiden)} \\ \leq \frac{1}{2}\,\beta^{1-N} & \text{(Rundung)}\,. \end{cases}$$

Dieser durch rd beschriebene Approximationsprozeß ist ein Modell für die in Computerrechnungen auftretenden Rundungsfehler. Ob abgeschnitten oder gerundet wird, hängt dabei von der verwendeten *Softwareumgebung* ab.

Dem Rundungsprozeß für die Mantisse z entspricht eine Rundung der kompletten Zahl x; dies halten wir fest:

4.1.7 Definition (N-stellige Gleitpunktapproximation) *Die* N-stellige Gleitpunktapproximation zur Basis β *mit* N *signifikanten Ziffern einer Zahl* $x \in \mathbb{R} \setminus \{0\}$ *ist gegeben durch*

$$\begin{aligned} \mathsf{rd}(x) = \mathsf{rd}(\mathsf{sign}(x)\,z \times \beta^e) := \mathsf{sign}(x)\,\mathsf{rd}(z) \times \beta^e &= \pm\left(\sum_{i=1}^{N} z_i\,\beta^{-i}\right)\beta^e \qquad (4.1) \\ &\mathrel{\widehat{=}} (\pm, z_1, z_2, \ldots, z_N, e)\,. \end{aligned}$$

Statt N-stellige Gleitpunktapproximation sagt man auch dass $\mathsf{rd}(x)$ eine N-stellige Gleitpunktzahl ist. Zur internen Darstellung reeller Zahlen speichert der Rechner nach Definition 4.1.7 auf der vorherigen Seite: das Vorzeichen $\mathsf{sign}(x) \in \{\pm 1\}$ (welches meist durch ein Bit repräsentiert wird), die N signifikanten Ziffern $z_1, z_2, \ldots, z_N$ der Mantisse und den Exponenten $e \in \mathbb{Z}$. Die meisten heutigen Rechnerarchitekturen verwenden maschinenintern in der Regel die Basis $\beta = 2$ (binäres System). Bei N-stelliger Gleitpunktapproximation im binären System nennt man die N signifikanten Ziffern der Mantisse auch *Präzisionsbits*. Die Mantisse selbst wird dann auch als *Signifikante* bezeichnet.

Es folgt die Abschätzung

$$\left|\frac{x - \mathsf{rd}(x)}{x}\right| = \left|\frac{z - \mathsf{rd}(z)}{z}\right| \;\leq\; \tau := \begin{cases} < \beta^{1-N} & \text{(Abschneiden)} \\ \leq \frac{1}{2}\,\beta^{1-N} & \text{(Rundung)}\,. \end{cases} \tag{4.2}$$

Die Zahl τ heißt *relative Genauigkeit* der für die Rechnungen verwendeten Computersprache, sie wird im Folgenden kurz als (relative) *Maschinengenauigkeit* bezeichnet werden. Sie ist der Parameter, welcher die Größe der auftretenden Rundungsfehler bestimmt.

Als wesentliches Ergebnis ist zu merken, dass Zahlen auf dem Rechner mit einer *relativen* Genauigkeit dargestellt werden, das heißt, die vom Rechner benutzte Gleitpunktzahl $\mathsf{rd}(x)$ hängt mit der eigentlichen Zahl $x \in \mathbb{R}$ zusammen über

$$\mathsf{rd}(x) \;=\; x\,(1 + \tau_x)\,, \tag{4.3}$$

wobei für den relativen Darstellungsfehler $\tau_x = (\mathsf{rd}(x) - x)/x$ die Abschätzung

$$|\tau_x| \;\leq\; \tau \;=\; \text{Maschinengenauigkeit} \tag{4.4}$$

gilt. Die Darstellung (4.3) mit der Abschätzung (4.4) stellt den *fundamentalen* Rundungsfehler dar, aus denen die bei Rechnungen auftretenden Fehler abzuschätzen sind.

Im Zweifelsfalle läßt sich die implementierte Maschinengenauigkeit τ leicht austesten, da sie die kleinste positive Zahl darstellt, die man zur 1 addieren kann, ohne dass das Ergebnis durch Rundung wieder zu 1 wird. Ein entsprechendes Pseudocode-Programm ist in **Algorithmus 4.1** abgedruckt.

Algorithmus 4.1 Maschinengenauigkeit austesten

Eingabe: –

```
1: /* Sei τ eine Objekt-Instanz bzw. eine Variable eines Gleitpunktzahl-Datentyps */
2: τ ← 1
3: while 1 + τ > 1 do
4:     τ ← τ/2
5: end while
```

Ausgabe: Die Maschinengenauigkeit τ

Typische Werte für die Maschinengenauigkeit sind $\tau = 2^{-23} \approx 1.2 \times 10^{-7}$ (single precision) und $\tau = 2^{-53} \approx 1.1 \times 10^{-16}$ (double precision, siehe auch Beispiel 4.1.9 auf der nächsten Seite). Grob gesprochen sind bei einer Maschinengenauigkeit τ in der Größenordnung 10^{-16} die ersten 16 signifikanten Dezimalziffern richtig dargestellt. Es hängt von dem jeweils verwendeten Compiler der für die Rechnungen benutzten Sprache ab, welche Speicherlänge für reelle Zahlen wirklich verwendet wird. In der Regel wählt der Programmierer aus unterschiedlichen zur Verfügung stehenden Datentypen[2] den geeignetsten aus. Weiterhin gibt es unterschiedliche Implementierungen, wieviele Bits der Mantisse und wieviele Bits dem Exponenten jeweils zugeordnet sind, so dass die davon abhängenden Größen wie Maschinengenauigkeit und Exponentenbereich variieren können. Die Beschreibung des jeweils verwendeten Compilers (bzw. seiner Bibliotheken) gibt im einzelnen Fall darüber Auskunft. Die Betrachtungen in Beispiel 4.1.9 auf der nächsten Seite zu 64-Bit-Variablen und die dort abgeleitete Maschinengenauigkeit $\tau = 2^{-53} \approx 10^{-16}$ gelten in der Praxis also nicht allgemein. Im Prinzip existiert ein internationaler Standard (IEEE 754 bzw. auch IEEE 854), der diese Parameter festlegt und von Compilerbauern eingehalten werden sollte [Goldberg, 1991]. Die Existenz eines solchen Standards ist allerdings keine Garantie, dass die benutzte Software (Compiler) bzw. Hardware (Floating-Point-Unit des Prozessors) diese Empfehlungen tatsächlich einhält.

Zusammenfassend halten wir fest:

4.1.8 Definition (Maschinenzahlen) *Für eine gegebene Basis $\beta \in \{2, 3, 4, \dots\}$, eine Stellenzahl $N \in \mathbb{N} \setminus \{0\}$ und einen Exponentenbereich $e_{min}, e_{min}+1, \dots, e_{max} \in \mathbb{Z}$ ist*

$$\mathcal{M}(\beta, N, e_{min}, e_{max}) := \{0\} \cup \left\{ \pm \left(\sum_{i=1}^{N} z_i \, \beta^{-i} \right) \beta^{e} \;\middle|\; z_i \in \{0, 1, \dots, \beta - 1\}\,,\ z_1 \neq 0\,, \right.$$
$$\left. e \in \{e_{min}, e_{min}+1, \dots, e_{max}\} \right\}$$

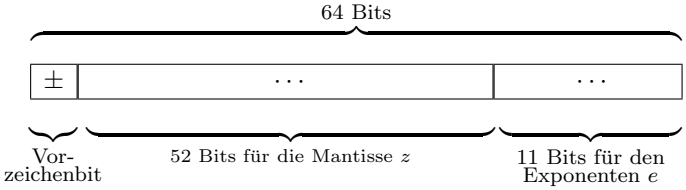
die Menge der vom Rechner darstellbaren Maschinenzahlen.

Falls $\beta, N, e_{\min}$ und $e_{\max}$ aus dem Kontext hervorgehen oder in einem Zusammenhang nicht relevant sind schreiben wir statt $\mathcal{M}(\beta, N, e_{\min}, e_{\max})$ einfach $\mathcal{M}$. Üblicherweise ist die Menge $\mathcal{M}$ der Maschinenzahlen durch die Hardware- bzw. Software-Umgebung des Rechners festgelegt, das heißt $N, e_{\min}$ und $e_{\max}$ sind durch den gewählten Datentyp einer Programmiersprache vorgegeben (vergleiche Beispiel 4.1.9 auf der nächsten Seite). Benötigt man (Gleitpunkt-) Zahlen-Datentypen die die gewählte Programmiersprache nicht von sich aus zur Verfügung stellt, so kann man eine Langzahlarithmetik-Bibliothek einsetzen. Diese ergänzt die Programmiersprache um entsprechende Datentypen. Eine solche Software-Lösung ist natürlich nicht sehr schnell – im Vergleich zu Datentypen die komplett in Hardware realisiert sind. Wir kommen in Kapitel 6 auf Seite 73 genauer auf die uns zur Verfügung stehenden Datentypen zu sprechen; denn

[2] In der Programmiersprache `C/C++` etwa den Datentyp `float` für single precision Variablen oder `double` für double precision Variablen.

insbesondere hinsichtlich der Reduktion von Gitterbasen mit sehr großen Einträgen bieten nicht-Standard Datentypen Vorteile.

4.1.9 Beispiel (double precision Gleitpunktzahl) Die maschineninterne Darstellung von $\mathsf{rd}(x) = \mathsf{rd}(z)\,\beta^e$ mit $\beta = 2$ sieht bei der Verwendung von 64-Bit-Variablen (double precision nach IEEE 754 [Goldberg, 1991]) im Prinzip[3] etwa folgendermaßen aus:



Mantissengenauigkeit: Da die führende Ziffer $z_1 \in \{1, 2, \ldots, \beta - 1\}$ in Gleichung (4.1) auf Seite 39 nicht verschwinden darf und für $\beta = 2$ demnach gleich 1 sein muß, brauchen nur die weiteren Ziffern gespeichert werden. Die 52 Bits für die Mantisse lassen somit die Verwendung von insgesamt $N = 53$ Binärziffern zu, was mit Gleichung (4.2) auf Seite 40 bei Rundung eine Maschinengenauigkeit von $\tau = 2^{-52}/2 \approx 1.1 \times 10^{-16}$ ergibt. Das nichtgespeicherte Bit $z_1 = 1$ nennt man „verstecktes“ Bit.

Exponentenbereich: Die 11 Bits für den Exponenten ermöglichen die Speicherung einer Vorzeicheninformation sowie die Binär-Kodierung der Exponenten von 0 bis $\sum_{i=0}^{9} 2^i = 2^{10} - 1 = 1023$. Wegen $2^{1023} \approx 10^{308}$, lassen sich mit 64 Bits im Prinzip alle reellen Zahlen x im Bereich $10^{-308} < |x| < 10^{308}$ mit relativer Genauigkeit $\tau = 2^{-53}$ von etwa 16 Dezimalstellen darstellen. In der Regel wird als Exponentenbereich lediglich $-1022, \ldots, 1023$ zugelassen, wobei die dem kleinsten im Prinzip darstellbaren Exponenten -1023 entsprechende Bit-Kombination dazu verwendet wird, auf „nichtnormierte“ Mantissen umzuschalten. Man erlaubt dabei, dass auch die führende Ziffer z_1 in Gleichung (4.1) auf Seite 39 verschwinden kann, so dass dann noch kleinere Zahlen dargestellt werden können, welche die Zahlenlücke zwischen 0 und 2^{-1022} bis zu einem gewissen Grade auffüllen.

Zusammenfassend halten wir fest, dass der von double precision Datentypen abgedeckte Zahlenbereich gegeben ist durch $\mathcal{M}' := \mathcal{M}(2, 53, -1022, 1023) \cup \mathcal{X}$, wobei $\mathcal{X}$ die Menge der bereits erwähnten Zahlen ist, die die Zahlenlücke zwischen 0 und 2^{-1022} auffüllen.

◇

Der *IEEE Standard 754-1985 for Binary Floating-point Arithmetic* verlangt $\beta = 2$ und spezifiziert die in **Tabelle 4.1** auf der nächsten Seite angegebenen Formate für Gleitpunktzahlen [Goldberg, 1991]. Man beachte die Benutzung des „versteckten“ Bits (siehe Beispiel 4.1.9) bei dem single bzw. dem double Format. Nach [Goldberg, 1991] benutzt man bei der Hardware Implementierung der extended precision Formate nicht das „versteckte“ Bit, so dass zum Beispiel für double-extended statt der spezifizierten 79 Bits tasächlich 80 Bits gebraucht werden.

[3] Die genaue Darstellung ist abhängig von der Rechnerarchitektur. Ein Problem in diesem Zusammenhang ist das *byte-ordering*: zum Beispiel *big endian* oder *little endian*. Beim Austausch elektronischer Dateien (insbesondere Binärdateien) kann es unter Umständen zu Fehlern bei der Verarbeitung der Dateiinhalte kommen, falls die beteiligten Rechnerarchitekturen unterschiedliches byte-ordering verwenden.

Format	Parameter					
	Anzahl der Bits für					
	das Format	das Vorzeichen	die Mantisse	den Exponenten	$e_{\min}$	$e_{\max}$
single	32	1	23 ($N = 24$)	8	-126	$+127$
single-extended	43	1	≥ 32	≤ 11	≤ -1022	$+1023$
double	64	1	52 ($N = 53$)	11	-1022	$+1023$
double-extended	79	1	63	15	≤ -16382	$+16383$

Tabelle 4.1: IEEE 754 Format Parameter für Gleitpunktzahlen

Tabelle 6.2 auf Seite 76 zeigt die Parameter für die von uns hauptsächlich benutzten Gleitpunktzahlen-Datentypen (siehe auch Definition 4.1.8 auf Seite 41).

Ein Problem[4] mit Maschinenzahlen ist, dass die mathematisch exakte Arithmetik die Menge $\mathcal{M}$ verlassen kann (zum Beispiel $x, y \in \mathcal{M} \Rightarrow x + y \in \mathbb{R} \setminus \mathcal{M}$). Daher wird die exakte Arithmetik durch eine Ersatzarithmetik approximiert, die wieder Werte in den Maschinenzahlen annimmt. Zur allgemeinen Beschreibung dieser Ersatzarithmetik vereinbart man das folgende

4.1.10 Rechenmodell (Arithmetik auf Maschinenzahlen) *Für eine auf Maschinenzahlen auszuführende Operation der Grundarithmetik wird das mathematisch exakte Ergebnis auf die nächstliegende Maschinenzahl gerundet und so abgespeichert.*

Die Rundung auf die nächstliegende Zahl ist in den von uns benutzten Compilern standardmäßig eingestellt. Es ist jedoch möglich den Rundungsprozeß zu verändern, um etwa immer „in Richtung“ Null oder $\pm\infty$ zu runden. Genaueres entnimmt man der Beschreibung des verwendeten Compilers bzw. der entsprechenden Bibliotheken, etwa [Free Software Foundation, 2001].

Sollten die arithmetischen Operationen aus dem Bereich der Maschinenzahlen herausführen, so werden bei den Programmier-Hochsprachen in der Regel Fehlermeldungen (underflow/overflow) ausgegeben.

Die Ersatzarithmetik läßt sich nur schwer mit den üblichen Mitteln der Analysis handhaben. Aus diesem Grunde ist es schwierig, die auftretenden Rundungsfehler bei einer komplizierten Folge von Rechenoperationen mathematisch zufriedenstellend zu beschreiben. Beim Vergleichen von Sätzen die die numerische Güte von Verfahren abschätzen ist das zugrundeliegende Rechenmodell zur Fehlerfortpflanzung zu beachten.

[4] Ein anderes Problem ist, dass die Arithmetik auf $\mathcal{M}$ im allgemeinen – auf Grund der Rundung – nicht assoziativ ist.

4.1.1 Fehlerfortpflanzung

Bei der Arithmetik auf Maschinenzahlen entstehen zwangsläufig Rundungsfehler. Die wesentliche Frage lautet: Was passiert mit solchen Fehlern im Verlauf der weiteren Rechnungen. Die numerische Mathematik lehrt, dass von den vier Grundrechenarten $(+, -, \times, /)$ lediglich die Subtraktion zweier ähnlicher (also zweier etwa gleich großer) Zahlen kritisch ist (*Auslöschung von Präzisionsbits* in der Mantisse), da hierbei relative Fehler extrem verstärkt werden können. In jeder Ausführung einer Elementaroperation entsteht ein zusätzlicher Darstellungsfehler, der durch die Maschinengenauigkeit beschränkt ist.

4.1.11 Beispiel(Auslöschung) Im folgenden symbolisieren durch „?“ angedeutete Ziffern den Darstellungsfehler. Gegeben seien zwei Maschinenzahlen

$$x = 0.98765444??\ldots \quad \text{und} \quad y = 0.9876533??\ldots \quad .$$

Obwohl hier x und y im Zehnersystem ($\beta = 10$) auf jeweils 8 Ziffern genau dargestellt sind, ist die Differenz durch Auslöschung nur bis auf 3 Ziffern genau bestimmt:

$$\begin{array}{rcl} x & = & 0.98765444??\ldots \\ y & = & 0.98765333??\ldots \\ \hline x - y & = & 0.00000111??\ldots \end{array}$$

4.2 QR-Faktorisierung mit Gleitpunktzahlen

Die in Kapitel 5 (Seite 55 ff.) behandelte Gitterbasenreduktion in Segmenten setzt voraus, dass die Matrix R der Faktorisierung $B = QR$ zur Verfügung steht. Über die Eigenschaften der QR-Faktorisierung informieren die beiden folgenden Sätze.

4.2.1 Satz (QR-Faktorisierung) *Sei $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{R}^{d\times n}$ eine Matrix mit vollen Spaltenrang. Dann ist die Zerlegung*

$$B^{(i)} := [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_i] = Q^{(i)}R^{(i)} \quad \textit{für}\ \ i = 1, 2, \ldots, n$$

eindeutig bestimmt, falls $Q^{(i)}$ orthogonal und $R^{(i)}$ eine obere Dreiecksmatrix mit positiven Diagonaleinträgen ist.

Den Beweis von Satz 4.2.1 kann man in [Goloub und Loan, 1989] nachlesen.

4.2.2 Satz (QR-Faktorisierung) *Jede quadratische Matrix A besitzt eine Zerlegung der Form $A = QR$ mit einer orthogonalen Matrix Q und einer oberen Dreiecksmatrix R. Für invertierbares A ist diese Faktorisierung bis auf eine Multiplikation $\bar{Q} = QD$, $\bar{R} = DR$ mit einer Diagonalmatrix $D = \mathrm{diag}(\pm 1, \ldots, \pm 1)$ eindeutig.*

Einen Beweis von Satz 4.2.2 auf der vorherigen Seite findet man beispielsweise in [Oevel, 1996, Abschnitt 5.9].

Eine konstruktive Methode zur QR-Faktorisierung von B ist beispielsweise das (Gram-) Schmidt'sche Orthogonalisierungsverfahren, das in Abschnitt 2.3 auf Seite 18 angegeben ist. Aus numerischer Sicht ist dieses Verfahren jedoch nicht empfehlenswert, da es instabil ist. Als Alternativen zur Konstruktion von Q und R bieten sich *Householder-Reflexionen* und *Givens-Rotationen* an. Beide Methoden sind numerisch deutlich besser als das Verfahren von Schmidt, und daher bei der Verwendung von Gleitpunktzahlen-Arithmetik diesem vorzuziehen. Im Zusammenhang mit Reduktion von Gitterbasen (bei Verwendung von Gleitpunktzahlen-Arithmetik) hat sich Henrik Koy bereits in [Koy, 1999] mit Householder-Reflexionen und Givens-Rotationen beschäftigt. Wir brauchen diese beiden Verfahren als Grundlage für die Implementierung der Konzepte aus [Koy und Schnorr, 2001b] in unserem `latred`-Programm. Daher geben wir im Folgenden die den Verfahren zugrundeliegenden Fakten wieder. Auf eine Rundungsfehleranalyse der hier angesprochenen Orthogonalisierungsmethoden verzichten wir. Abschätzungen für die auftretenden Fehler finden sich in [Koy, 1999, Abschnitt 1.3.6] und auch in [Koy und Schnorr, 2001b]. Weitere Informationen zum Orthogonalisieren mit Gleitpunktzahlen, insbesondere auch zu Householder-Reflexionen und Givens-Rotationen, findet man in einführenden Büchern über numerische Mathematik, etwa [Deuflhard und Hohmann, 1991; Oevel, 1996].

Die Unterschiede zwischen Householder-Reflexion und Givens-Rotation liegen zum ersten im Aufwand (die Anzahl der zur Faktorisierung nötigen Operationen) und zum zweiten in ihrer numerischen Güte. Bevor wir in den nächsten zwei Abschnitten die Verfahren näher beschreiben, geben wir in **Tabelle 4.2** die interessantesten Eigenschaften der bisher angesprochenen Methoden wieder [nach Koy, 1999]. Die Angaben beziehen sich auf die Faktorisierung einer Matrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \dots\ \boldsymbol{b}_n] \in \mathbb{Z}^{d\times n}$ unter Verwendung von Maschinenzahlen $\mathcal{M}$ mit Stellenzahl N (siehe Definition 4.1.8 auf Seite 41). Dabei sei $R' = [r'_{i,j}]_{1\le i,j\le n} \in \mathcal{M}^{n\times n}$ die numerische Approximation der exakten Lösung $R = [r_{i,j}]_{1\le i,j\le n} = Q^{\mathsf{T}}B \in \mathbb{R}^{n\times n}$.

Methode	Operationen	Fehler $\lvert r'_{j,\nu} - r_{j,\nu}\rvert$	Arithmetik
Gram-Schmidt	$O\left(dn^2 + n^3\right)$	—	über $\mathbb{Q}$
Householder	$O\left(2n^2\left(d - \frac{n}{3}\right)\right)$	$\le 2.5d^2 \cdot 2^{-N}\lVert\boldsymbol{b}_\nu\rVert$	über $\mathcal{M}$
Givens	$O\left(3n^2\left(d - \frac{n}{3}\right)\right)$	$\le 14d \cdot 2^{-N}\lVert\boldsymbol{b}_\nu\rVert$	über $\mathcal{M}$

Tabelle 4.2: Aufwand und numerische Güte von Orthogonalisierungsverfahren

Die Berechnung der QR-Zerlegung einer Basismatrix $B \in \mathbb{Z}^{d\times n}$ eines Gitters $\mathcal{L}(B)$, der Gleichung (2.4) auf Seite 19 entsprechend, erfolgt in unserem `latred`-Programm hauptsächlich durch Householder-Reflexionen (siehe dazu Abschnitt 6.3.1.1 auf Seite 83). Eine Givens-Rotation setzen wir lediglich zur effizienten Korrektur einer Ma-

trix R ein, die fast obere[5] Dreiecksgestalt hat (siehe dazu Abschnitt 6.3.1.2 auf Seite 86). Warum letztlich beide Methoden eingesetzt werden, ist auf Seite 52 und in Abschnitt 6.3.1 auf Seite 81 erklärt.

4.2.1 Householder-Reflexion

Wir beschreiben die Methode zur QR-Zerlegung durch die so genannten Householder-Reflexionen. Ausgehend von einer Eingabematrix $B \in \mathbb{R}^{d\times n}$ wird durch eine Folge von Householder-Reflexionen sukzessive die obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ konstruiert. Man erinnere sich an die Darstellung nach Gleichung (2.4) auf Seite 19.

4.2.3 Definition (Householder-Matrix) *Die dem Vektor $\boldsymbol{v} \in \mathbb{R}^d$ zugeordnete* Householder-Matrix *ist*

$$H = H(\boldsymbol{v}) := \begin{cases} E_d - 2\dfrac{\boldsymbol{v}\boldsymbol{v}^\mathsf{T}}{\boldsymbol{v}^\mathsf{T}\boldsymbol{v}} & \text{falls } \boldsymbol{v} \neq \boldsymbol{0}, \\ E_d & \text{falls } \boldsymbol{v} = \boldsymbol{0}. \end{cases}$$

4.2.4 Bemerkung (Eigenschaften der Householder-Matrix) *Die Householder-Matrix $H \in \mathbb{R}^{d\times d}$ ist orthogonal ($HH^\mathsf{T} = H^\mathsf{T}H = E_d$), symmetrisch ($H^\mathsf{T} = H$) und involutorisch ($H^2 = E_d$).*

4.2.5 Definition (Erzeugungs-Vektor, Householder-Vektor, Householder-Reflexion) *Zu $j \in \{1, 2, \ldots, d\}$ und dem* Erzeugungs-Vektor $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)^\mathsf{T} \in \mathbb{R}^d$ *definiere die von $\boldsymbol{x}$ erzeugte Householder-Matrix $H^{(j)}(\boldsymbol{x}) \in \mathbb{R}^{d\times d}$ durch*

$$\begin{aligned} t &= t^{(j)}(\boldsymbol{x}) := \mathsf{sign}(x_j)\sqrt{x_j^2 + x_{j+1}^2 + \cdots + x_d^2}\,, \\ \boldsymbol{v} &= \boldsymbol{v}^{(j)}(\boldsymbol{x}) := (\underbrace{0, 0, \ldots, 0}_{j-1 \text{ Nullen}}, t + x_j, x_{j+1}, \ldots, x_d)^\mathsf{T}\,, \\ H &= H^{(j)}(\boldsymbol{x}) := \begin{cases} E_d - 2\dfrac{\boldsymbol{v}\boldsymbol{v}^\mathsf{T}}{\boldsymbol{v}^\mathsf{T}\boldsymbol{v}} & \text{falls } \boldsymbol{v} \neq \boldsymbol{0}, \\ E_d & \text{falls } \boldsymbol{v} = \boldsymbol{0}. \end{cases} \end{aligned} \tag{4.5}$$

Wir vereinbaren $\mathsf{sign}(0) := 1$. *Mit dem Erzeugungs-Vektor $\boldsymbol{x}$ ist $H^{(j)}(\boldsymbol{x})$ die* Householder-Reflexion *zum* Householder-Vektor $\boldsymbol{v}^{(j)}(\boldsymbol{x})$.

Falls aus dem Kontext ersichtlich welcher Erzeugungs-Vektor $\boldsymbol{x}$ und Index j gemeint ist, schreiben wir statt $\boldsymbol{v}^{(j)}(\boldsymbol{x})$ bzw. $H^{(j)}(\boldsymbol{x})$ wegen der Übersichtlichkeit einfach $\boldsymbol{v}$ bzw. H.

4.2.6 Lemma *Für die durch den Erzeugungs-Vektor $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)^\mathsf{T} \in \mathbb{R}^d$ definierte Householder-Matrix $H = H^{(j)}(\boldsymbol{x})$ gilt:*

$$H\boldsymbol{x} = (x_1, x_2, \ldots, x_{j-1}, -t, 0, 0, \ldots, 0)^\mathsf{T} \in \mathbb{R}^d.$$

[5] Da wir in `latred` mit transponierten Matrizen arbeiten, haben die Matrizen, die mit Givens-Rotationen „repariert“ werden, fast untere Dreiecksgestalt. Siehe Kapitel 6 auf Seite 73.

Jeder Vektor der Form $\boldsymbol{y} = (y_1, y_2, \ldots, y_{j-1}, 0, 0, \ldots, 0)^\mathsf{T} \in \mathbb{R}^d$ *wird von dieser Matrix invariant gelassen:* $H\boldsymbol{y} = \boldsymbol{y}$.

Beweis von 4.2.6:

Mit $t^2 = x_j^2 + x_{j+1}^2 + \cdots + x_d^2$ folgt

$$\frac{2\boldsymbol{v}^\mathsf{T}\boldsymbol{x}}{\boldsymbol{v}^\mathsf{T}\boldsymbol{v}} = \frac{2((t+x_j)x_j + x_{j+1}^2 + \cdots + x_d^2)}{(t+x_j)^2 + x_{j+1}^2 + \cdots + x_d^2} = \frac{2(tx_j + t^2)}{t^2 + 2tx_j + t^2} = 1$$

und

$$H\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{v}\frac{2\boldsymbol{v}^\mathsf{T}\boldsymbol{x}}{\boldsymbol{v}^\mathsf{T}\boldsymbol{v}} = \boldsymbol{x} - \boldsymbol{v} = (x_1, x_2, \ldots, x_{j-1}, -t, 0, 0, \ldots, 0)^\mathsf{T} \in \mathbb{R}^d.$$

Für $\boldsymbol{x} = (x_1, x_2, \ldots, x_{j-1}, 0, 0, \ldots, 0)^\mathsf{T}$ (der einzige Fall mit $H = E_d$) gilt die Aussage ebenfalls (beachte $t = 0$). Für $\boldsymbol{y} = (y_1, y_2, \ldots, y_{j-1}, 0, 0, \ldots, 0)^\mathsf{T}$ gilt $\boldsymbol{v}^\mathsf{T}\boldsymbol{y} = 0$, also $H\boldsymbol{y} = \boldsymbol{y}$.

❏

Diese Householder-Matrizen haben also die Eigenschaft, in dem zur Konstruktion der Matrix verwendeten Vektor $\boldsymbol{x}$ die unterhalb der j-ten Stelle stehenden Komponenten zu eliminieren, während für alle Vektoren die oberhalb der j-ten Stelle stehenden Komponenten invariant gelassen werden.

4.2.7 Bemerkung (Numerische Stabilität durch Vermeidung von Auslöschung) *Im Beweis von Lemma 4.2.6 auf der vorherigen Seite wurde nur* $t^2 = x_j^2 + x_{j+1}^2 + \cdots + x_d^2$ *benutzt, womit das Vorzeichen von* t *beliebig wählbar ist. Das in Definition 4.2.5 auf der vorherigen Seite gewählte Vorzeichen vermeidet, dass es in der* j*-ten Koordinate von* $\boldsymbol{v}^{(j)}(\boldsymbol{x})$ *zur Auslöschung kommt (siehe auch Beispiel 4.1.11 auf Seite 44). Für* $x_j = 0$ *setzt man entweder* $\mathsf{sign}(0) = +1$ *oder* $\mathsf{sign}(0) = -1$*, die sonst übliche Vereinbarung* $\mathsf{sign}(0) = 0$ *führt hier zu falschen Ergebnissen.*

4.2.8 Konstruktion (QR-Zerlegung durch Householder-Transformationen) Eine zu faktorisierende $d \times n$ Matrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n]$, welche durch ihre Spaltenvektoren gegeben ist, soll durch Multiplikation mit Householder-Matrizen auf obere Dreiecksform transformiert werden. Dazu setzt man

$$B^{(1)} = \left[\boldsymbol{b}_1^{(1)}\ \boldsymbol{b}_2^{(1)}\ \ldots\ \boldsymbol{b}_n^{(1)}\right] = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] = B$$

und wählt die über Gleichung (4.5) durch die erste Spalte definierte Matrix

$$H^{(1)} = H^{(1)}\left(\boldsymbol{b}_1^{(1)}\right).$$

Nach Lemma 4.2.6 auf der vorherigen Seite werden durch Multiplikationen mit $H^{(1)}$ in der ersten Spalte von $B^{(1)}$ Nullen unterhalb der ersten Komponente

erzeugt. Es ergibt sich

$$B^{(2)} = H^{(1)}B^{(1)} = \left[H^{(1)}\boldsymbol{b}_1^{(1)}\ H^{(1)}\boldsymbol{b}_2^{(1)}\ \dots\ H^{(1)}\boldsymbol{b}_n^{(1)}\right] = \left[\begin{pmatrix}*\\0\\\vdots\\0\end{pmatrix}\ \boldsymbol{b}_2^{(2)}\ \dots\ \boldsymbol{b}_n^{(2)}\right].$$

Die mittels Gleichung (4.5) auf Seite 46 mit $j = 2$ von der zweiten Spalte $\boldsymbol{b}_2^{(2)} = H^{(1)}\boldsymbol{b}_2^{(1)}$ erzeugte Householder-Matrix

$$H^{(2)} = H^{(2)}\left(\boldsymbol{b}_2^{(2)}\right)$$

transformiert dieses Schema auf die nächste Zwischenform

$$\begin{aligned} B^{(3)} = H^{(2)}B^{(2)} &= \left[H^{(2)}\boldsymbol{b}_1^{(2)}\ H^{(2)}\boldsymbol{b}_2^{(2)}\ H^{(2)}\boldsymbol{b}_3^{(2)}\ \dots\ H^{(2)}\boldsymbol{b}_n^{(2)}\right] \\ &= \left[\begin{pmatrix}*\\0\\0\\\vdots\\0\end{pmatrix}\ \begin{pmatrix}*\\*\\0\\\vdots\\0\end{pmatrix}\ \boldsymbol{b}_3^{(3)}\ \dots\ \boldsymbol{b}_n^{(3)}\right]. \end{aligned}$$

Gemäß Lemma 4.2.6 auf Seite 46 werden hierbei in der zweiten Spalte die Komponenten unter der Diagonalen eliminiert, während die erste Spalte invariant gelassen wird. Offensichtlich lassen sich diese Schritte fortsetzen, bis das Schema obere Dreiecksform annimmt.

Der Algorithmus besteht also aus der Rekursion

$$B^{(1)} = B\,, \quad B^{(j+1)} = H^{(j)}B^{(j)}\,,$$

wobei $H^{(j)}$ die über Gleichung (4.5) auf Seite 46 von der j-ten Spalte von $B^{(j)}$ erzeugte Householder-Matrix ist. Die am Ende des $(n-1)$-ten Schrittes vorliegende Matrix $R = B^{(n)}$ ist von oberer Dreiecksform. Mit $\left(H^{(j)}\right)^2 = E_d$ folgt

$$\begin{aligned} R = B^{(n)} &= H^{(n-1)}H^{(n-2)}\cdots H^{(2)}H^{(1)}B \\ \Longrightarrow\ B &= H^{(1)}H^{(2)}\cdots H^{(n-2)}H^{(n-1)}R\,, \end{aligned}$$

wobei die Matrix $Q = H^{(1)}H^{(2)}\cdots H^{(n-2)}H^{(n-1)}$ als Produkt orthogonaler Matrizen wieder orthogonal ist. Die gesuchte Faktorisierung $B = QR$ ist damit nach $n-1$ Schritten gefunden. Da die Householder-Matrix nach Gleichung (4.5) auf der Seite 46für jeden beliebigen Vektor $\boldsymbol{x}$ existiert, ist jeder Schritt des Verfahrens stets durchführbar, so dass jede Matrix in der angegebenen Form faktorisiert werden kann.

Für den i-ten Spaltenvektoren von $B^{(j)}$ $(j = 1, 2, \dots, n)$ gilt allgemein:

$$\boldsymbol{b}_i^{(j)} = \begin{cases} H^{(j-1)} \cdot H^{(j-2)} \cdots H^{(2)} \cdot H^{(1)} \cdot \boldsymbol{b}_i & \text{falls } i \geq j, \\ H^{(i)} \cdot H^{(i-1)} \cdots H^{(2)} \cdot H^{(1)} \cdot \boldsymbol{b}_i & \text{falls } i < j, \end{cases}$$

wobei $H^{(k)} = H^{(k)}\left(\boldsymbol{b}_k^{(k)}\right)$ für $k = 1, 2, \ldots, n-1$ ist. Damit kann die obere Dreiecksmatrix $R = [\boldsymbol{r}_1\ \boldsymbol{r}_2\ \ldots\ \boldsymbol{r}_n] \in \mathbb{R}^{n\times n}$ der Faktorisierung $B = QR$ sukzessive durch

$$\boldsymbol{r}_i = \boldsymbol{b}_i^{(i+1)} = H^{(i)} \cdot H^{(i-1)} \cdots H^{(2)} \cdot H^{(1)} \cdot \boldsymbol{b}_i \quad \text{für } 1 \leq i \leq n-1\,, \tag{4.6}$$

und

$$\boldsymbol{r}_n = \boldsymbol{b}_n^{(n+1)} = H^{(n-1)} \cdot H^{(n-2)} \cdots H^{(2)} \cdot H^{(1)} \cdot \boldsymbol{b}_n \tag{4.7}$$

berechnet werden, wobei $H^{(k)} = H^{(k)}\left(\boldsymbol{b}_k^{(k)}\right)$ für $k = 1, 2, \ldots, n-1$ ist.

◇

4.2.9 Bemerkung (Householder-Vektoren bei sukzessiver Orthogonalisierung speichern) *Nach Gleichung* (4.6) *bzw. Gleichung* (4.7) *kann man die Vektoren von B sukzessive orthogonalisieren. Da zur Berechnung von $\boldsymbol{r}_i$ die Householder-Vektoren*

$$\boldsymbol{v}^{(1)}\left(\boldsymbol{b}_1^{(1)}\right), \boldsymbol{v}^{(2)}\left(\boldsymbol{b}_2^{(2)}\right), \ldots, \boldsymbol{v}^{(i)}\left(\boldsymbol{b}_i^{(i)}\right) \tag{4.8}$$

vorliegen müssen, ist es sinnvoll diese zu speichern, um sie bei der Berechnung von $\boldsymbol{r}_{i+1}$ wiederzuverwenden, wo sie erneut benötigt werden. Zusätzlich ist dann lediglich der Erzeugungs-Vektor $\boldsymbol{b}_{i+1}^{(i+1)}$ und der entsprechende Householder-Vektor $\boldsymbol{v}^{(i+1)}\left(\boldsymbol{b}_{i+1}^{(i+1)}\right)$ zu berechnen, wobei sich

$$\boldsymbol{b}_{i+1}^{(i+1)} = H^{(i)} \cdot H^{(i-1)} \cdots H^{(1)} \cdot \boldsymbol{b}_{i+1} \tag{4.9}$$

leicht aus den gespeicherten Werten (4.8) *und $\boldsymbol{b}_{i+1}$ ergibt.*

Bemerkung 4.2.9 ist besonders im Zusammenhang mit der Gitterbasenreduktion in Segmenten nützlich (siehe Abschnitt 5.2 auf Seite 56 und Abschnitt 5.3 auf Seite 62): Dort benötigen wir nicht von Beginn an die komplette Faktorisierung $B = QR$, sondern es macht Sinn die Faktorisierung sukzessive – gemäß Satz 4.2.1 auf Seite 44 – zu berechnen.

In Abschnitt 6.3.1.1 auf Seite 83 beschreiben wir die Implementierung der Householder-Reflexion in unserem `latred`-Programm.

4.2.2 Givens-Rotation

Wir beschreiben die Methode zur QR-Zerlegung durch sogenannte Givens-Rotationen. Ausgehend von einer Eingabematrix $B \in \mathbb{R}^{d\times n}$ wird durch eine Folge von Givens-Rotationen sukzessive die obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ konstruiert.

4.2.10 Definition (Givens-Rotation) *Gegeben sei der Vektor* $\boldsymbol{x} = (x_1, \ldots, x_d)^\mathsf{T} \in \mathbb{R}^d$. *Die für* $k, \ell \in \{1, 2, \ldots, d\}$ *erklärte* Givens-Rotation $G_{k,\ell}(\boldsymbol{x}) : \mathbb{R}^d \longrightarrow \mathbb{R}^d$

$$\begin{pmatrix} x_1 \\ \vdots \\ x_{k-1} \\ x_k \\ x_{k+1} \\ \vdots \\ x_{\ell-1} \\ x_\ell \\ x_{\ell+1} \\ \vdots \\ x_d \end{pmatrix} \longmapsto G_{k,\ell}(\boldsymbol{x}) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_{k-1} \\ x_k \\ x_{k+1} \\ \vdots \\ x_{\ell-1} \\ x_\ell \\ x_{\ell+1} \\ \vdots \\ x_d \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_{k-1} \\ \sqrt{x_k^2 + x_\ell^2} \\ x_{k+1} \\ \vdots \\ x_{\ell-1} \\ 0 \\ x_{\ell+1} \\ \vdots \\ x_d \end{pmatrix}, \tag{4.10}$$

rotiert x_ℓ *in die Koordinate* k. *Die Matrix* $G_{k,\ell}(\boldsymbol{x}) \in \mathbb{R}^{d \times d}$ *sieht wie folgt aus:*

$$G_{k,\ell}(\boldsymbol{x}) = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \\ \begin{array}{ccccccc} & & k & & \ell & & \end{array} \end{array} \begin{array}{l} \\ \\ k \\ \\ \ell \\ \\ \\ \end{array} \quad \text{mit} \quad \begin{aligned} c &= \frac{x_k}{\sqrt{x_k^2 + x_\ell^2}}, \\ s &= \frac{x_\ell}{\sqrt{x_k^2 + x_\ell^2}}. \end{aligned}$$

Alle Einträge außerhalb der Diagonalen sind – mit Ausnahme in den Positionen (k, ℓ) und (ℓ, k) – Null. Es gilt $c^2 + s^2 = 1$. Die Givens-Rotation ist eine zweidimensionale Rotationsmatrix, die in der k-ten und ℓ-ten Dimension rotiert. Offensichtlich ist $G_{k,\ell}(\boldsymbol{x})$ orthogonal.

4.2.11 Bemerkung (Berechnung von c und s) *Aus numerischen Gründen berechnet man* c *und* s *günstiger mit den Formeln*[6]

$$\tau := x_k / x_\ell \,, \quad s := 1/\sqrt{1 + \tau^2} \,, \quad c := s\tau \,, \quad \textit{falls} \;\; |x_\ell| > |x_k| \,;$$

und

$$\tau := x_\ell / x_k \,, \quad c := 1/\sqrt{1 + \tau^2} \,, \quad s := c\tau \,, \quad \textit{falls} \;\; |x_\ell| \leq |x_k| \,.$$

[6] Siehe [Deuflhard und Hohmann, 1991, Abschnitt 3.2.1]. Damit vermeidet man zugleich Exponentenüberlauf.

4.2.12 Bemerkung (Wirkung auf eine Matrix) *Multipliziert man eine Matrix*

$$A = [\boldsymbol{a}_1\ \boldsymbol{a}_2\ \ldots\ \boldsymbol{a}_n] \in \mathbb{R}^{d\times n}$$

von links mit $G_{k,\ell}$, so operiert die Givens-Rotation auf den Spalten, das heißt

$$G_{k,\ell}\, A = \big[G_{k,\ell}\,\boldsymbol{a}_1 \quad G_{k,\ell}\,\boldsymbol{a}_2 \quad \ldots \quad G_{k,\ell}\,\boldsymbol{a}_n\big]\,.$$

Es werden daher wegen Gleichung (4.10) *auf der vorherigen Seite nur die beiden Zeilen k und ℓ der Matrix A verändert. Dies lässt sich ausnutzen, falls man bei der Transformation möglichst Besetzungsstrukturen der Matrix erhalten möchte.*

Auf die Matrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{R}^{d\times n}$ angewandt, rotiert die Matrixmultiplikation mit $G_{k,\ell}(\boldsymbol{b}_j)$ den Eintrag $b_{\ell,j}$ zu 0. Sukzessive auf die Einträge unterhalb der Diagonalen angewandt, erhalten wir also durch $\sum_{k=1}^{n}(d-k) = nd - n(n+1)/2$ viele orthogonale Transformationen eine obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ (man beachte, dass die orthogonalen Matrizen mit der Multiplikation eine Gruppe bilden):

$$R = \Big(\prod G_{k,\ell}(\cdot)\Big)\, B \quad \Longleftrightarrow \quad \underbrace{\Big(\prod G_{k,\ell}(\cdot)\Big)^{\mathsf{T}}}_{=:\,Q} R = B\,.$$

4.2.13 Beispiel (QR-Zerlegung durch Givens-Rotationen) Wir beschreiben die QR-Zerlegung anhand der 3×2 Matrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2]$. Durch drei Givens-Rotationen werden die Matrixeinträge $b_{3,1}, b_{2,1}$ und $b_{3,2}$ nach Null rotiert:

$$\underbrace{\begin{bmatrix} b_{1,1}^{(1)} & b_{1,2}^{(1)} \\ b_{2,1}^{(1)} & b_{2,2}^{(1)} \\ b_{3,1}^{(1)} & b_{3,2}^{(1)} \end{bmatrix}}_{=:\,[\boldsymbol{b}_1\ \boldsymbol{b}_2]=B} \xrightarrow{G_{2,3}(\boldsymbol{b}_1)} \underbrace{\begin{bmatrix} b_{1,1}^{(1)} & b_{1,2}^{(1)} \\ b_{2,1}^{(2)} & b_{2,2}^{(2)} \\ 0 & b_{3,2}^{(2)} \end{bmatrix}}_{=:\,\left[\boldsymbol{b}_1^{(2)}\ \boldsymbol{b}_2^{(2)}\right]} \xrightarrow{G_{1,2}\left(\boldsymbol{b}_1^{(2)}\right)} \underbrace{\begin{bmatrix} b_{1,1}^{(2)} & b_{1,2}^{(2)} \\ 0 & b_{2,2}^{(3)} \\ 0 & b_{3,2}^{(2)} \end{bmatrix}}_{=:\,\left[\boldsymbol{r}_1\ \boldsymbol{b}_2^{(3)}\right]}$$

$$\underbrace{\begin{bmatrix} b_{1,1}^{(2)} & b_{1,2}^{(2)} \\ 0 & b_{2,2}^{(3)} \\ 0 & b_{3,2}^{(2)} \end{bmatrix}}_{=\left[\boldsymbol{r}_1\ \boldsymbol{b}_2^{(3)}\right]} \xrightarrow{G_{2,3}\left(\boldsymbol{b}_2^{(3)}\right)} \underbrace{\begin{bmatrix} b_{1,1}^{(2)} & b_{1,2}^{(2)} \\ 0 & b_{2,2}^{(4)} \\ 0 & 0 \end{bmatrix}}_{=:\,[\boldsymbol{r}_1\ \boldsymbol{r}_2]=R}.$$

Aus $R = G_{2,3}(\boldsymbol{b}_2^{(3)}) \cdot G_{1,2}(\boldsymbol{b}_1^{(2)}) \cdot G_{2,3}(\boldsymbol{b}_1) \cdot B$ ergibt sich die orthogonale Matrix Q, der Zerlegung $B = QR$, als $Q = G_{2,3}(\boldsymbol{b}_1)^{\mathsf{T}} \cdot G_{1,2}(\boldsymbol{b}_1^{(2)})^{\mathsf{T}} \cdot G_{2,3}(\boldsymbol{b}_2^{(3)})^{\mathsf{T}}$.

Die Givensmethode benötigt zur Berechnung der oberen Dreiecksmatrix R maximal $3n^2(d - n/3)$ Gleitpunkt-Operationen. Zu beachten ist aber, dass die Methode eine dünnbesetzte Matrix sehr schnell auf Dreiecksform bringt, da Einträge gezielt „zu Null

rotiert“ werden können. Beispielsweise benötigt man nur $n-1$ Givens-Rotationen, um eine so genannte *Hessenberg-Matrix*[7] auf Dreiecksgestalt zu bringen.

Um eine gegebene Basismatrix B zu faktorisieren, also die dazu isometrische Matrix R zu berechnen, benutzen wir in unserer Implementierung von `latred` hauptsächlich die Householder-Reflexionen. Die Givens-Rotationen wenden wir lediglich in speziellen Situationen an, die im folgenden beschrieben sind. Im Verlauf der Algorithmen zur Gitterbasenreduktion werden oftmals *benachbarte* Vektoren in der Basismatrix bzw. in R vertauscht. Dabei zerstört man die obere Dreiecksgestalt von R. In unserem Programm `latred` benutzen wir die Givens-Rotation daher zur Korrektur einer Matrix R', die aus R durch Vertauschung von Vektoren hervorgeht und die deshalb fast Dreiecksgestalt hat. Um das präziser auszudrücken bedienen wir uns der folgenden Definition:

4.2.14 Definition (Einfach gestört in der j-ten Spalte) *Sei $R \in \mathbb{R}^{d\times n}(d \geq n)$ eine Matrix in* oberer *Dreiecksgestalt. Vertauscht man in R die beiden Spaltenvektoren j und $j+1$ für $j \in \{1, 2, \ldots, n-1\}$, so sagen wir für die resultierende Matrix $R' \in \mathbb{R}^{d\times n}$, dass sie* einfach gestört in der j-ten Spalte *ist.*

Einfach gestörte Matrizen haben also fast Dreiecksgestalt. Daher lassen sich sehr effizient mit einer Givens-Rotation „reparieren“. Dazu ein Beispiel:

4.2.15 Beispiel (Korrektur einer einfach gestörten Matrix) Gegeben sei eine Matrix $R \in \mathbb{R}^{d\times n}(d \geq n)$ in oberer Dreiecksgestalt. Durch Vertauschung zweier *benachbarter* Spaltenvektoren wird die obere Dreiecksform zerstört. Allerdings ist die Korrektur (das heißt die Matrix wieder auf obere Dreiecksgestalt bringen) mit einer Givens-Rotation leicht möglich. Wir veranschaulichen die Situation durch die folgende 7×6 Matrix:

$$\underbrace{\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & r_{1,5} & r_{1,6} \\ 0 & r_{2,2} & r_{2,3} & r_{2,4} & r_{2,5} & r_{2,6} \\ 0 & 0 & r_{3,3} & r_{3,4} & r_{3,5} & r_{3,6} \\ 0 & 0 & 0 & r_{4,4} & r_{4,5} & r_{4,6} \\ 0 & 0 & 0 & 0 & r_{5,5} & r_{5,6} \\ 0 & 0 & 0 & 0 & 0 & r_{6,6} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{=:[\boldsymbol{r}_1\ \boldsymbol{r}_2\ \ldots\ \boldsymbol{r}_6]=R} \xrightarrow{\boldsymbol{r}_3 \leftrightarrow \boldsymbol{r}_4} \underbrace{\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,4} & r_{1,3} & r_{1,5} & r_{1,6} \\ 0 & r_{2,2} & r_{2,4} & r_{2,3} & r_{2,5} & r_{2,6} \\ 0 & 0 & r_{3,4} & r_{3,3} & r_{3,5} & r_{3,6} \\ 0 & 0 & r_{4,4} & 0 & r_{4,5} & r_{4,6} \\ 0 & 0 & 0 & 0 & r_{5,5} & r_{5,6} \\ 0 & 0 & 0 & 0 & 0 & r_{6,6} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{=:[\boldsymbol{r}'_1\ \boldsymbol{r}'_2\ \ldots\ \boldsymbol{r}'_6]=R'} .$$

Der Spaltenvektor $\boldsymbol{r}'_3 = (r_{1,4}, r_{2,4}, r_{3,4}, r_{4,4}, 0, 0)^\mathsf{T}$ stört offenbar die Dreiecksgestalt: Die Matrix R' ist einfach gestört in der 3-ten Spalte. Die 4-te Koordinate von $\boldsymbol{r}'_3$ wird durch eine Givens-Rotation $G_{3,4}$ in die 3-te Koordinate rotiert. Durch die Multiplikation $G_{3,4}(\boldsymbol{r}'_3) \cdot R'$ erhält man wieder eine obere Dreiecksmatrix $\bar{R}$:

$$\bar{R} := G_{3,4}(\boldsymbol{r}'_3) \cdot R' = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,4} & r_{1,3} & r_{1,5} & r_{1,6} \\ 0 & r_{2,2} & r_{2,4} & r_{2,3} & r_{2,5} & r_{2,6} \\ 0 & 0 & \bar{r}_{3,3} & \bar{r}_{3,4} & \bar{r}_{3,5} & \bar{r}_{3,6} \\ 0 & 0 & 0 & \bar{r}_{4,4} & \bar{r}_{4,5} & \bar{r}_{4,6} \\ 0 & 0 & 0 & 0 & r_{5,5} & r_{5,6} \\ 0 & 0 & 0 & 0 & 0 & r_{6,6} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

[7] Eine Hessenberg-Matrix ist fast schon eine obere Dreiecksmatrix, da sie nur in der Nebendiagonale weitere von Null verschiedene Einträge besitzt.

Man beachte, dass sich R' und $\bar{R}$ nur in den Zeilen 3 und 4 Unterscheiden (siehe auch Bemerkung 4.2.12 auf Seite 51).

In Abschnitt 6.3.1.2 auf Seite 86 beschreiben wir die Implementierung der Givens-Rotation in unserem `latred`-Programm.

4.3 Gitterbasenreduktion mit Computern

Die Basisvektoren eines Gitters *müssen* exakt gespeichert werden, da sonst das Gitter verändert wird. Dies gilt auch für ganzzahlige Gitterbasen eines Gitters $L \subset \mathbb{Z}^d$. Eine zu reduzierende Gitterbasis $B = [b_{i,j}]_{1 \le i \le d; 1 \le j \le n} \in \mathbb{Z}^{d \times n}$ hat meistens sehr große Einträge. Sei $\ell = \left(\sum_{i=1}^{d} \sum_{j=1}^{n} \mathsf{bitlength}(b_{i,j})\right)/(d \cdot n)$ die mittlere Bitlänge der Einträge in B. Typischerweise ist in unseren Experimenten $50 \le \ell \le 500$, und $d = n$ liegt zwischen 100 und 1000. Diese Werte orientieren sich an Parametern aus [Goldreich, Goldwasser und Halevi, 1997; Koy, 1999] – wobei wir allerdings im Unterschied zu [Koy, 1999] auch Gitter mit Dimension größer als 400 reduzieren.

Die meisten Programmiersprachen haben *standardmäßig* keine effizienten Datenstrukturen/Datentypen für „sehr große“ ganze Zahlen: Beispielsweise ist eine ganze Zahl, die in binärer Darstellung mehr als $w = 8*$`sizeof(long int)` Bits benötigt, in der Programmiersprache `C/C++` nicht mehr durch eine Variable des Typs `long int`[8] darstellbar. Auf einer 32-Bit Rechnerarchitektur ist üblicherweise $w = 32$. Damit ist es offensichtlich nicht möglich, die von uns benötigten Zahlen in einer `long int` Variablen zu speichern. Eine zusätzliche Langzahlarithmetik-Bibliothek, die Datentypen für beliebig große Zahlen zur Verfügung stellt, ist erforderlich. Verschiedene Bibliotheken realisieren die Darstellung der Zahlen und die Arithmetik mit ihnen unterschiedlich effizient. Da es sich immer um Software Lösungen (eventuell mit Maschinenspracheoptimierten Routinen) handelt, ist die Verwendung solcher Datentypen deutlich langsamer – im Vergleich zu Datentypen die direkt von der Rechner-Hardware unterstützt werden. Auf die von uns verwendeten Bibliotheken kommen wir in Abschnitt 6.1 auf Seite 74 zu sprechen.

Bei der Reduktion von (ganzzahligen) Gitterbasen werden Gleitpunktzahlen eingesetzt. Diese können einen sehr großen Zahlenbereich abdecken (siehe Definition 4.1.8 auf Seite 41 und Beispiel 4.1.9 auf Seite 42) und mit ihnen wird sehr schnell gerechnet, da Gleitpunktarithmetik vom Prozessor direkt unterstützt[9] wird. Man konvertiert die ganzen Zahlen in Gleitpunktzahlen und berechnet dann die zu B isometrische Matrix R (siehe Abschnitt 2.3 auf Seite 18 und Abschnitt 4.2 auf Seite 44). Die Einträge in R sind Gleitpunktzahlen, das heißt sie sind mit numerischen Ungenauigkeiten behaftet. Mit R kann, wegen der Gleitpunktarithmetik, effizient gearbeitet werden. Allerdings erkauft man sich das mit inhärenten Rundungsfehlern. Diese versucht man unter Kontrolle zu halten, etwa mit den Vorschlägen in [Koy und Schnorr, 2001b]. Bei der Reduktion von R speichert man die entsprechenden unimodularen Transformationen

[8] `long int` ist in `C/C++` der Datentyp für hardwareseitig unterstützte große ganze Zahlen.
[9] Dies gilt zumindest für die beiden Formate „single“ und „double“ in **Tabelle 4.1** auf Seite 43.

in einer Matrix. In diesem Zusammenhang erinnere man sich auch an Satz 2.2.5 auf Seite 15. Die gemerkten unimodularen Transformationen werden bei Bedarf von R auf die eigentliche Basismatrix B übertragen. Dies geschieht dann allerdings zwangsläufig mit der langsamen ganzzahl Arithmetik. Der Geschwindigkeitsvorteil dieser Vorgehensweise resultiert aus der Tatsache, dass man t viele (lokale) Teiltransformationen $T_1, T_2, \ldots, T_t \in \mathsf{SL}_n(\mathbb{Z})$ schnell (Stichwort: Gleitpunktarithmetik) auf R durchführen $(R \cdot T_1 \cdot T_2 \cdots T_t)$ und diese in *einer* Transformationsmatrix $T = T_1 \cdot T_2 \cdots T_t \in \mathsf{SL}_n(\mathbb{Z})$ speichern kann. Dabei werden unter Umständen einige Schritte verworfen und es wird neu iteriert. Erst bei zufriedenstellenden Reduktionsfortschritt wird anschließend T direkt auf B übertragen: BT (dabei wird mit langsamer ganzzahl Arithmetik gerechnet). Charakteristisch ist also das schnelle Rechnen auf R, wobei eventuell mehrmals iteriert wird, und das anschließende langsame Übertragen des erzielten Fortschritts auf B. Statt auf der $n \times n$ Matrix R zu rechnen, ist es effizienter auf Submatrizen von R zu arbeiten: Man vergleiche ein solches Vorgehen auch mit dem Konzept der Reduktion in lokalen Koordinaten (siehe Abschnitt 3.5 auf Seite 33) und dem Konzept der Reduktion in Segmenten (siehe Kapitel 5 auf der nächsten Seite).

Ein Problem bei der Gitterbasenreduktion mit Gleitpunktzahlen sind die Rundungsfehler. Sie treten hauptsächlich bei der Orthogonalisierung von Vektoren auf. In [Koy, 1999, S. 49–50] konnte eine numerisch stabile Reduktion von ganzzahligen Gitterbasen mit Dimension 400 nicht durchgeführt werden, obwohl Householder-Reflexionen zur Orthogonalisierung benutzt wurden. In [Koy und Schnorr, 2001b] wird begründet warum die Householder-Reflexion bei der Gitterbasenreduktion in Dimension ≥ 400 numerisch instabil ist, falls Gleitpunktzahlen mit 53 Präzisionsbits verwendet werden. Das wesentliche Problem sind die unterschiedlichen Höhen $\|\widehat{\boldsymbol{b}}_i\|$ der Basisvektoren, denn nach Gleichung (2.4) auf Seite 19 bestimmen sie die Einträge in der isometrischen Matrix R. Und nach Abschnitt C.2 auf Seite 117 gibt es einen unmittelbaren Zusammenhang zwischen den Einträgen in R und der Stabilität des Orthogonalisierungsprozesses: In die Abschätzung der Rundungsfehler geht die Norm der Matrix R ein. In [Koy und Schnorr, 2001b] werden die Basisvektoren *skaliert*, so dass ihre Höhen ungefähr gleich sind. Die skalierten Vektoren können stabil orthogonalisiert werden. Die in [Koy und Schnorr, 2001b] beschriebene *slash* Heuristik beschleunigt die Reduktion. Die Segmentreduktion [Koy und Schnorr, 2001a] begünstigt die Umsetzung des skalierens und des slashens. In Abschnitt 5.2.1 auf Seite 60 kommen wir darauf zu sprechen; die Auswirkungen der zur Verfügung stehenden Präzisionsbits von Gleitpunktzahlen auf die *skalierte k-Segment LLL-Reduktion* werden in Abschnitt 6.3.2.1 auf Seite 88 deutlich gemacht.

5 Gitterbasenreduktion in Segmenten

Die Einteilung von Gitterbasen in Segmente ermöglicht es, effizient in lokalen Koordinaten zu rechnen. Denn zum ersten wird die Dimension der zu betrachtenden Vektoren verkleinert, was dazu führt, dass lokal weniger Operationen durchgeführt werden müssen. Und zum zweiten geschieht das Rechnen in den lokalen Koordinaten in Gleitpunktarithmetik mit einem deutlichen Geschwindigkeitsvorteil[1].

Wir stellen in diesem Kapitel die *k-Segment LLL-Reduktion* (Abschnitt 5.2 auf der nächsten Seite) nach KOY und SCHNORR vor [Koy und Schnorr, 2001a,b]. Dieses Reduktionsverfahren ist so konzipiert, dass möglichst viel Reduktionsfortschritt in lokalen Koordinaten (mit Gleitpunktarithmetik) geschieht und es anschließend – jedoch nicht „unnötig“ oft – zu einer globalen Reduktion der Basisvektoren in exakter Arithmetik kommt. Durch die lokalen Segmente wird auch die Kontrolle bzw. die Vermeidung von Gleitpunktzahlen-Rundungsfehlern beim Rechnen begünstigt. Die entsprechende Variante der k-Segment LLL-Reduktion bezeichnen wir als *skalierte k-Segment LLL-Reduktion* – darauf geht Abschnitt 5.2.1 auf Seite 60 ein.

Abschnitt 5.3 auf Seite 62 beschäftigt sich mit der *primal/dualen Segmentreduktion*, die auf KOY zurückgeht [Koy, 2002]. Dabei handelt es sich um einen Reduktionsbegriff, der zu benachbarten Segmente das jeweils primale bzw. duale Gitter betrachtet. Bei fester Segmentweite k hat das Verfahren polynomielle Laufzeit. Die reduzierte Basis besteht aus kurzen Vektoren mit guten beweisbaren Schranken. Die primal/dual reduzierten Basen reichen in der Praxis allerdings nicht an die Güte von β-reduzierten Basen heran[2].

5.1 Einteilung einer Basismatrix in Segmente

Gegeben seien die Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$ eines Gitter $L \subseteq \mathbb{R}^d$. Wir teilen die Basismatrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{R}^{d \times n}$ in $m \in \{1, 2, \ldots, n\}$ disjunkte Segmente $B_1, B_2, \ldots, B_m$ auf. Dabei unterscheiden wir folgende zwei Fälle:

1. Im Fall $n = k \cdot m$ setzen wir

$$B_\ell := [\boldsymbol{b}_{k(\ell-1)+1}\ \boldsymbol{b}_{k(\ell-1)+2}\ \ldots\ \boldsymbol{b}_{k\ell}] \in \mathbb{R}^{d \times k} \quad \text{für } \ell = 1, 2, \ldots, m. \tag{5.1}$$

[1] Bei sehr großen ganzen Zahlen ist das Rechnen in exakter Arithmetik deutlich langsamer, als das Rechnen mit entsprechenden Gleitpunktzahlen. Siehe auch Abschnitt 4.3 auf Seite 53.

[2] Bei gleicher Eingabe-Basis braucht die β-Reduktion aber (zeitlich) sehr viel länger, als die primal/-duale Reduktion, um die Basis zu reduzieren.

2. Im Fall $n = k \cdot (m-1) + \varrho\,,\ \varrho \in \{1, 2, \ldots, k-1\}$, hat das letzte Segment B_m ϱ Vektoren:

$$B_m := \begin{bmatrix} \boldsymbol{b}_{k(m-1)+1} & \boldsymbol{b}_{k(m-1)+2} & \ldots & \boldsymbol{b}_{k(m-1)+\varrho} \end{bmatrix} \in \mathbb{R}^{d \times \varrho}. \tag{5.2}$$

Statt den Parameter m werden wir oft nur die Segmentgröße $k \in \{1, 2, \ldots, n\}$ angeben. Die Anzahl der Segmente ergibt sich dann als $m = \lceil n/k \rceil$. Die Segmente $B_1, B_2, \ldots, B_{m-1}$ bestehen aus k Vektoren und werden nach Gleichung (5.1) auf der vorherigen Seite gebildet. Die Anzahl der Vektoren in Segment B_m ist abhängig von k und n. Falls $n \pmod k = 0$, das heißt $n = k \cdot m$, so erhält man B_m nach Gleichung (5.1) auf der vorherigen Seite. Andernfalls gilt $n \pmod k \neq 0$, das heißt $n = k \cdot \lfloor n/k \rfloor + \varrho\,,$ $\varrho \in \{1, 2, \ldots, k-1\}$, und Gleichung (5.2) kommt bei der Bestimmung von B_m zur Anwendung.

Meistens gehen wir davon aus, dass $n = k \cdot m$ gilt. Dadurch ist es nicht nötig den Spezialfall B_m nach Gleichung (5.2) zu betrachten. Es sei jedoch erwähnt, dass es in der Implementierung der Segment LLL-Reduktionsverfahren (in `latred`) nicht nötig ist, dass $n = k \cdot m$ gilt. Dort wird die oben vorgestellte Fallunterscheidung vorgenommen.

5.2 k-Segment LLL-Reduktion

Gegeben sei die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eines Gitters, $n = k \cdot m$, mit der Faktorisierung[3] $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] = QR$ (siehe Gleichung (2.4) auf Seite 19). Für zwei aufeinander folgende Segmente $B_\ell, B_{\ell+1} \in \mathbb{Z}^{d \times k}$ der Basismatrix $B \in \mathbb{Z}^{d \times n}$ betrachten wir die Darstellung in lokalen Koordinaten (analog zu Gleichung (3.5) auf Seite 35). Es wird auf Submatrizen von R gearbeitet. In lokalen Koordinaten entspricht die $2k \times 2k$ Submatrix

$$R_{\ell,\ell+1} := [r_{k\ell+i,k\ell+j}]_{-k<i,j\leq k} \in \mathbb{R}^{2k \times 2k}\ ,\ \ell \in \{1, 2, \ldots, m-1\},$$

von $R = [r_{i,j}]_{1 \leq i,j \leq n} \in \mathbb{R}^{n \times n}$ den zwei Segmenten $B_\ell, B_{\ell+1} \in \mathbb{Z}^{d \times k}$ (siehe auch **Abbildung 5.1** auf Seite 58). Nach Gleichung (2.5) auf Seite 19 ergeben sich die Einträge in $R_{\ell,\ell+1}$ aus den Gram-Schmidt-Koeffizienten $\mu_{k\ell+i,k\ell+j}$ und den Höhen $\|\widehat{\boldsymbol{b}}_{k\ell+i}\|^2$.

Aufbau der Matrix $R_{\ell,\ell+1}$. Die im folgenden beschriebene Einteilung wird bei der Erläuterung der Implementierung des Verfahrens in `latred` nützlich sein (siehe Abschnitt 6.3.2 auf Seite 88). Um die Indizes der Einträge von R in den folgenden Darstellungen übersichtlich zu halten, setzen wir

$$i_l := k \cdot (\ell - 1) \quad \text{und} \quad i_m := i_l + k. \tag{5.3}$$

Man erinnere sich an die $\sigma_{s,k}\big(R^{s,k}\big)$ Notation aus Abschnitt 3.5 auf Seite 33 ff. Mit den drei Teilmatrizen

[3] Mit numerischen Verfahren zur QR-Faktorisierung beschäftigen wir uns in Abschnitt 4.2 auf Seite 44.

$$R^a_{\ell,\ell+1} := \sigma_{i_l,k}\big(R^{i_l,k}\big)$$
$$= \begin{bmatrix} r_{i_l+1,i_l+1} & r_{i_l+1,i_l+2} & \cdots & r_{i_l+1,i_l+(k-1)} & r_{i_l+1,i_l+k} \\ 0 & r_{i_l+2,i_l+2} & \cdots & r_{i_l+2,i_l+(k-1)} & r_{i_l+2,i_l+k} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_l+(k-1),i_l+(k-1)} & r_{i_l+(k-1),i_l+k} \\ 0 & \cdots & 0 & 0 & r_{i_l+k,i_l+k} \end{bmatrix} \in \mathbb{R}^{k\times k},$$

$$R^b_{\ell,\ell+1} := \begin{bmatrix} r_{i_l+1,i_m+1} & r_{i_l+1,i_m+2} & \cdots & r_{i_l+1,i_m+(k-1)} & r_{i_l+1,i_m+k} \\ r_{i_l+2,i_m+1} & r_{i_l+2,i_m+2} & \cdots & r_{i_l+2,i_m+(k-1)} & r_{i_l+2,i_m+k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ r_{i_m-1,i_m+1} & r_{i_m-1,i_m+2} & \cdots & r_{i_m-1,i_m+(k-1)} & r_{i_m-1,i_m+k} \\ r_{i_m+1,i_m+1} & r_{i_m+1,i_m+2} & \cdots & r_{i_m+1,i_m+(k-1)} & r_{i_m+1,i_m+k} \end{bmatrix} \in \mathbb{R}^{k\times k},$$

$$R^d_{\ell,\ell+1} := \sigma_{i_m,k}\big(R^{i_m,k}\big)$$
$$= \begin{bmatrix} r_{i_m+1,i_m+1} & r_{i_m+1,i_m+2} & \cdots & r_{i_m+1,i_m+(k-1)} & r_{i_m+1,i_m+k} \\ 0 & r_{i_m+2,i_m+2} & \cdots & r_{i_m+2,i_m+(k-1)} & r_{i_m+2,i_m+k} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_m+(k-1),i_m+(k-1)} & r_{i_m+(k-1),i_m+k} \\ 0 & \cdots & 0 & 0 & r_{i_m+k,i_m+k} \end{bmatrix} \in \mathbb{R}^{k\times k}$$

erhalten wir in der Kästchenschreibweise

$$R_{\ell,\ell+1} = \sigma_{i_l,2k}\big(R^{i_l,2k}\big) = \begin{bmatrix} R^a_{\ell,\ell+1} & R^b_{\ell,\ell+1} \\ 0 & R^d_{\ell,\ell+1} \end{bmatrix} \in \mathbb{R}^{2k\times 2k}. \tag{5.4}$$

Bei der *Segment LLL-Reduktion* werden Vektoren innerhalb eines Segments gemäß der Lovàsz-Bedingung (siehe Gleichung (3.2) auf Seite 27) ausgetauscht und auch Längenreduktionen finden statt – und zwar geschieht dies alles in lokalen Koordinaten auf der Matrix $R_{\ell,\ell+1}$ deren Einträge (in der Praxis) Gleitpunktzahlen sind. Nach dieser LLL-Reduktion in lokalen Koordinaten erfolgt eine Übertragung der lokalen Transformationen auf die beiden Segmente $B_\ell, B_{\ell+1}$ (und zwar auch lokal). Die Übertragung der Transformation auf B ist (in der Praxis) sehr teuer, da sie mit exakter Arithmetik durchgeführt werden muss (siehe Abschnitt 4.3 auf Seite 53).

Das Ziel der in [Koy und Schnorr, 2001a] vorgestellten *k-Segment LLL-Reduktion* ist es, die globalen Kosten während der Gitterbasenreduktion zu minimieren.

5.2.1 Notation (Lokale Gram'sche Determinante) *Es bezeichne*

$$D(\ell) := \|\widehat{\boldsymbol{b}}_{k(\ell-1)+1}\|^2 \cdot \|\widehat{\boldsymbol{b}}_{k(\ell-1)+2}\|^2 \cdots \|\widehat{\boldsymbol{b}}_{k\ell}\|^2$$

die lokale Gram'sche Determinante *von Segment* B_ℓ.

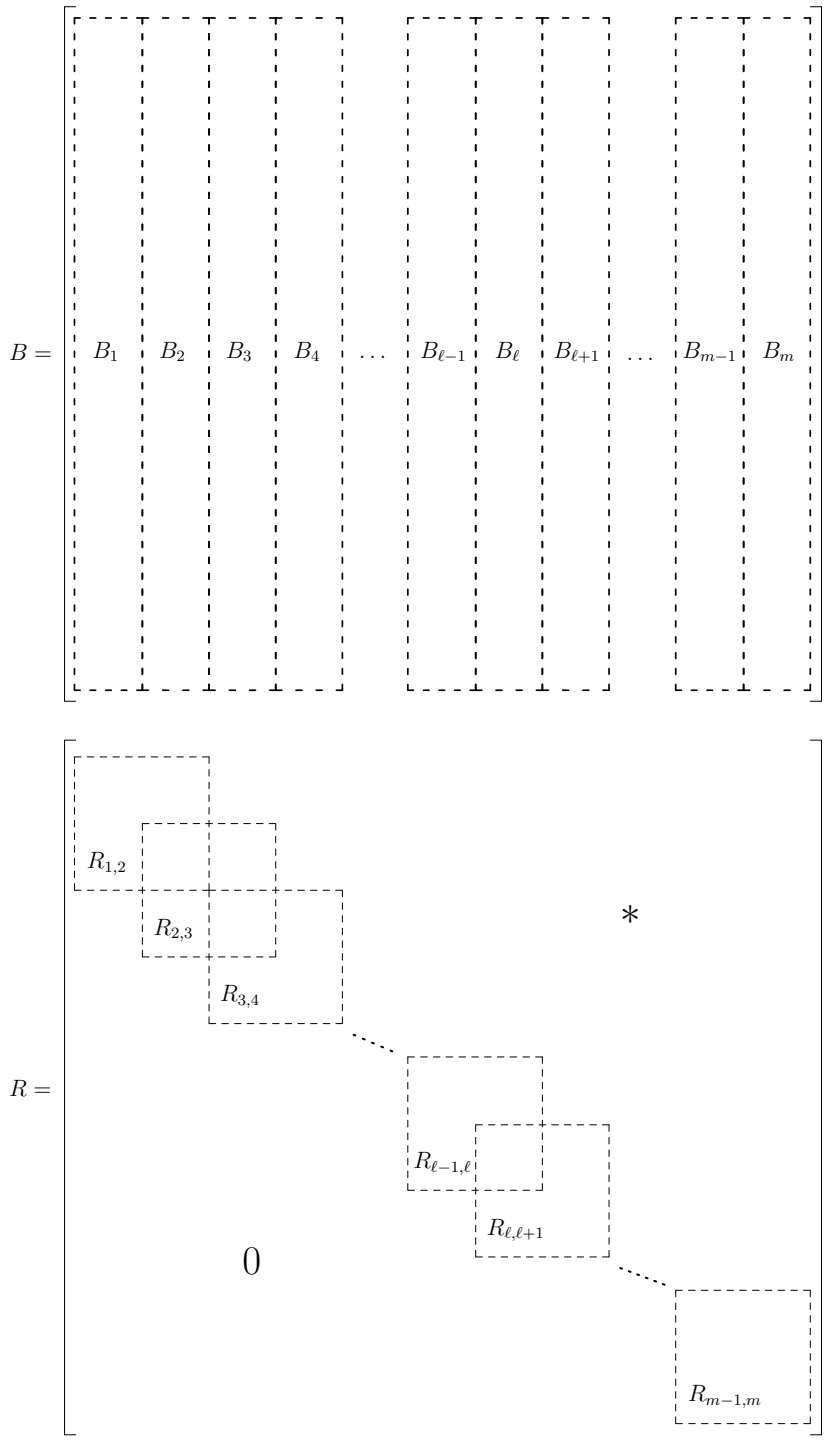


Abbildung 5.1: Einteilung einer Basismatrix B und der entsprechenden R Matrix in Segmente bei der k-Segment LLL-Reduktion

5.2.2 Definition (k-Segment LLL-Reduktion) *Eine geordnete Gitterbasis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$, $n = k \cdot m$, ist* k-Segment LLL-reduziert *mit $\delta \in (\frac{1}{4}, 1]$, wenn sie längenreduziert ist und für $\alpha = 1/(\delta - \frac{1}{4})$ gilt:*

KS1: $\delta \, \|\widehat{\boldsymbol{b}}_i\|^2 \leq \mu_{i+1,i}^2 \|\widehat{\boldsymbol{b}}_i\|^2 + \|\widehat{\boldsymbol{b}}_{i+1}\|^2$ *für* $i \not\equiv 0 \pmod{k}$,

KS2: $D(\ell) \leq (\alpha/\delta)^{k^2} D(\ell+1)$ *für* $\ell = 1, 2, \ldots, m-1$,

KS3: $\delta^{k^2} \, \|\widehat{\boldsymbol{b}}_{k\ell}\|^2 \leq \alpha \|\widehat{\boldsymbol{b}}_{k\ell+1}\|^2$ *für* $\ell = 1, 2, \ldots, m-1$.

Die Forderungen, dass die Basisvektoren längenreduziert sind und gleichzeitig Bedingungen KS1 erfüllt sind, bedeuten, dass die Basisvektoren für $\ell = 1, 2, \ldots, m-1$ innerhalb jedes *Doppelsegments* $[B_\ell \; B_{\ell+1}]$ LLL-reduziert sind. Die Bedingungen KS2 vergleichen die lokalen Gram'schen Determinanten zweier aufeinander folgenden Segmente. In KS3 betrachtet man die „Nahtstelle“ (den Übergang) von den Segmenten B_ℓ und $B_{\ell+1}$: Ein beschränktes Ansteigen der Höhen der Basisvektoren ist beim Übergang von dem Segment B_ℓ nach $B_{\ell+1}$ erlaubt. Die Bedingung KS3 erlaubt also einen Verlust der Reduktionsgüte (bezüglich der LLL-Reduktion) beim Übergang von einem Segment zum nächsten.

An dieser Stelle sei angemerkt, dass wir meistens die explizite Angabe von k weglassen und einfach *Segment LLL-Reduktion* schreiben.

5.2.3 Satz (Koy und Schnorr, 2001a) *Sei $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eine k-Segment LLL-reduzierte Basis mit $\delta \in (\frac{1}{4}, 1]$. Dann gilt für $i = 1, 2, \ldots, n$:*

$$\delta^{2k^2+n-1} \, \|b_i\|^2 \leq \alpha^{n-1} \lambda_i^2 \quad \text{und}$$
$$\delta^{k^2+i-1} \, \|b_1\|^2 \leq \alpha^{i-1} \|\widehat{b}_i\|^2 ,$$

wobei $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ die sukzessiven Minima des Gitters seien.

Der **Algorithmus 5.1** auf der nächsten Seite berechnet eine k-Segment LLL-reduzierte Basis eines Gitters. Man beachte auch **Abbildung 5.1** auf der vorherigen Seite und erinnere sich an Gleichung (2.5) auf Seite 19, wonach sich die Einträge in $R_{\ell,\ell+1}$ aus den Gram-Schmidt-Koeffizienten $\mu_{k\ell+i,k\ell+j}$ und den Höhen $\|\widehat{\boldsymbol{b}}_{k\ell+i}\|^2$ ergeben, das heißt, einige in Zeile 7 benötigten Werte sind in der R Matrix zu finden.

5.2.4 Satz (Koy und Schnorr, 2001a) *Sei $D_i := \|\widehat{\boldsymbol{b}}_1\|^2 \cdot \|\widehat{\boldsymbol{b}}_2\|^2 \cdots \|\widehat{\boldsymbol{b}}_i\|^2$ und $M := \max_{i=1,2,\ldots,n}(2^n, \|\boldsymbol{b}_i\|^2, D_i)$. Für $k = \Theta(m) = \Theta(\sqrt{n})$ führt der **Algorithmus 5.1** auf der nächsten Seite $O(nd \log_{1/\delta} M)$ arithmetische Schritte durch und rechnet dabei mit ganzen Zahlen der Bitlänge $O(\log_2 M)$.*

Abschnitt 4.3 auf Seite 53 beschäftigt sich mit den Auswirkungen von Zahlen-Datentypen auf die Effizienz und numerische Qualität von Algorithmen zur Gitterbasenreduktion. In [Koy und Schnorr, 2001b] werden Basisvektoren skaliert, um eine numerisch stabile Segment LLL-Reduktion zu erreichen. Auf die Vorschläge von Koy und Schnorr gehen wir im folgenden Abschnitt 5.2.1 auf der nächsten Seite ein.

Algorithmus 5.1 k-Segment LLL-Reduktion

Eingabe: Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L; $n = km$; $\delta \in (\frac{1}{4}, 1]$; $\alpha = 1/(\delta - \frac{1}{4})$

1: $\ell \leftarrow 1$
2: **while** $\ell \leq m$ **do**
3: **if** $\ell < m$ **then**
4: `loc-LLL(`ℓ`)` /* `loc-LLL` nur für $\ell \in \{1, 2, \ldots, m-1\}$ aufrufen, siehe **Algorithmus 5.2**. */
5: /* Der Aufruf `loc-LLL(`ℓ`)` bearbeitet $R_{\ell,\ell+1}$ bzw. $B_\ell, B_{\ell+1}$ (vergleiche mit **Abbildung 5.1** auf Seite 58). */
6: **end if**
7: **if** $\Big(\ell > 1$ **and** $\big(D(\ell-1) > (\alpha/\delta)^{k^2} D(\ell)$ **or** $\delta^{k^2} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha \|\widehat{b}_{k(\ell-1)+1}\|^2\big)\Big)$ **then**
8: $\ell \leftarrow \ell - 1$
9: **else**
10: $\ell \leftarrow \ell + 1$
11: **end if**
12: **end while**

Ausgabe: Mit Parameter δ k-Segment LLL-reduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters L

Algorithmus 5.2 `loc-LLL`

Eingabe: $\ell \in \{1, 2, \ldots, m-1\}$ und Vektoren von $B_\ell, B_{\ell+1}$ (siehe auch folgenden Text)

1: Ausgehend von einer gegebenen Orthogonalisierung $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k\ell}$ der bereits k-Segment LLL-reduzierten Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k\ell}$ (also der Segmente $B_1, B_2, \ldots, B_\ell$) berechne die Orthogonalisierung $\boldsymbol{r}_{k\ell+1}, \boldsymbol{r}_{k\ell+2}, \ldots, \boldsymbol{r}_{k(\ell+1)}$ (siehe Bemerkung 4.2.9 auf Seite 49) und die Längenreduktion zu $B_{\ell+1}$.
2: Aus $[\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell+1)}]$ bestimmt man die Matrix $R_{\ell,\ell+1} = [r_{k\ell+i,k\ell+j}]_{-k<i,j\leq k} \in \mathbb{R}^{2k\times 2k}$, die den beiden Segmenten $B_\ell, B_{\ell+1}$ entspricht (siehe **Abbildung 5.1** auf Seite 58).
3: Es erfolgt eine lokale LLL-Reduktion von $R_{\ell,\ell+1}$, wobei die dabei durchgeführten LLL-Transformationen in einer Transformationsmatrix $T \in \mathsf{SL}_{2k}(\mathbb{Z})$ „gemerkt“ werden.
4: Die beiden Segmente werden durch $[B_\ell \; B_{\ell+1}] \cdot T$ lokal LLL-reduziert, dann erfolgt eine globale Längenreduktion von $[B_\ell \; B_{\ell+1}]$.
5: Ausgehend von einer gegebenen Orthogonalisierung $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell-1)}$ der bereits k-Segment LLL-reduzierten Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k(\ell-1)}$ (also der Segmente $B_1, B_2, \ldots, B_{\ell-1}$) berechne die Orthogonalisierung $\boldsymbol{r}_{k(\ell-1)+1}, \boldsymbol{r}_{k(\ell-1)+2}, \ldots, \boldsymbol{r}_{k(\ell+1)}$ (siehe Bemerkung 4.2.9 auf Seite 49) und die Längenreduktion zu $B_\ell, B_{\ell+1}$.

Ausgabe: siehe obigen Text

5.2.1 Skalierte k-Segment LLL-Reduktion

Die bereits in Kapitel 4 auf Seite 37 beschriebenen numerischen Grundlagen und insbesondere das dort vorgestellten Orthogonalisierungsverfahren mit Householder-Reflexionen (siehe Abschnitt 4.2.1 auf Seite 46) bilden gemeinsam mit der eben (in Abschnitt 5.2 auf Seite 56) behandelten k-Segment LLL-Reduktion das Fundament der Ausführungen in [Koy und Schnorr, 2001b]. Daher sollte der Leser mit den in dieser Arbeit bereits vermittelten Kenntnissen die Problemstellung und die Lösungsvorschläge

in [Koy und Schnorr, 2001b] nachvollziehen können, und da das Paper von Koy und Schnorr in Anhang C auf Seite 115 vollständig wiedergegeben ist, verzichten wir an dieser Stelle auf eine detaillierte Beschreibung der dort vorgestellten Konzepte. In einem Satz zusammengefasst lautet die Botschaft: Die *skalierte k-Segment LLL-Reduktion* nach [Koy und Schnorr, 2001b] kombiniert numerisch relevanten Verbesserungen des Orthogonalisierungsprozesses und der Längenreduktion mit der k-Segment LLL-Reduktion, so dass dadurch die Gitterbasenreduktion in hohen Dimensionen möglich wird. Die wesentliche Feststellung ist, dass es nach Abschnitt C.2 auf Seite 117 einen unmittelbaren Zusammenhang zwischen den Einträgen in R und der numerischen Stabilität des Orthogonalisierungsprozesses (mit Householder-Reflexionen) gibt. Um numerisch negative Auswirkungen der Einträge in R zu vermeiden, bedienen Koy und Schnorr sich der beiden folgenden Strategien:

- Der zur Berechnung der R Matrix durchgeführte Orthogonalisierungsprozess erfolgt mit Hilfe einer zusätzlichen Matrix B^s, deren Vektoren skalierte Varianten der Basisvektoren sind: Für eine Basismatrix $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \dots\ \boldsymbol{b}_n] \in \mathbb{Z}^{d\times n}$ ergibt sich die skalierte Matrix $B^s = [2^{e_1}\boldsymbol{b}_1\ \ 2^{e_2}\boldsymbol{b}_2\ \ \dots\ \ 2^{e_n}\boldsymbol{b}_n] \in \mathbb{Z}^{d\times n}$, wobei die Skalierungsexponenten $e_i \in \mathbb{N}$ so gewählt bzw. im Verlauf des Verfahrens bei Bedarf dynamisch angepasst werden, dass die Höhen (die orthogonalen Projektionen) der (skalierten Basis-) Vektoren ungefähr gleich sind. Die R Matrix wird sukzessive (siehe Bemerkung 4.2.9 auf Seite 49) numerisch stabil berechnet, das heißt es wird Segment nach Segment abgearbeitet und die Bestimmung der Skalierungsexponenten geschieht innerhalb der Segmente. Warum und wie dies numerische Verbesserungen bringt, kann in Anhang C auf Seite 115 nachgelesen werden.
- Aufgrund der Einteilung der Basismatrix in Segmente und der Reduktion in lokalen Koordinaten kann die *slash* Heuristik auf $R_{\ell,\ell+1}$ angewandt werden (siehe Seite 126), wodurch versucht wird, die Norm der Matrix klein zu halten.

Es findet eine aktive und dynamische Anpassung der Orthogonalisierungs- und Längenreduktionsschritte an die numerischen Erfordernisse statt. Der wichtigste Bestandteil der skalierten Reduktion ist die HRS-Routine (**Algorithmus C.1** auf Seite 121) bzw. leicht modifizierte Varianten davon.

Der Algorithmus zur skalierten k-Segment LLL-Reduktion ist in [Koy und Schnorr, 2001b] fehlerhaft abgedruckt (siehe auch Seite 115). Unser **Algorithmus 5.3** auf der nächsten Seite zeigt eine korrigierte Variante.

5.2.5 Bemerkung (Betreffend Subroutine `loc-sLLL()`) *Die Prozedur `loc-sLLL(ℓ)` ist auf auf Seite 125 erläutert. Dort wird fälschlicherweise (siehe Seite 115) beschrieben, dass bei dem Aufruf von `loc-sLLL(ℓ)` die Matrix $R_{\ell-1,\ell}$, die den beiden Segmenten $B_{\ell-1}, B_\ell$ entspricht, bearbeitet wird. Nimmt man allerdings den korrekten **Algorithmus 5.3** auf der nächsten Seite zugrunde, so muss die Prozedur `loc-sLLL(ℓ)` die Matrix $R_{\ell,\ell+1}$ bzw. das Doppelsegment $B_\ell, B_{\ell+1}$ bearbeiten.*

Die Beschreibung der Implementierung der skalierten Segment LLL-Reduktion und der dazugehörenden Orthogonalisierungsverfahren erfolgt in Abschnitt 6.3 auf Seite 80.

Algorithmus 5.3 Skalierte k-Segment LLL-Reduktion

Eingabe: Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L; $n = km$; $\delta \in (\frac{1}{4}, 1]$; $\alpha = 1/(\delta - \frac{1}{4})$

1: $\ell \leftarrow 1$
2: **while** $\ell \leq m$ **do**
3: **if** $\ell < m$ **then**
4: `loc-sLLL(`ℓ`)` /* `loc-sLLL` nur für $\ell \in \{1, 2, \ldots, m-1\}$ aufrufen; man beachte Bemerkung 5.2.5 auf der vorherigen Seite. */
5: /* Der Aufruf `loc-sLLL(`ℓ`)` bearbeitet $R_{\ell,\ell+1}$ bzw. $B_\ell, B_{\ell+1}$ (vergleiche mit **Abbildung 5.1** auf Seite 58). */
6: **end if**
7: **if** $\Big(\ell > 1$ **and** $\big(D(\ell-1) > (\alpha/\delta)^{k^2} D(\ell)$ **or** $\delta^{k^2} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha \|\widehat{b}_{k(\ell-1)+1}\|^2\big)\Big)$ **then**
8: $\ell \leftarrow \ell - 1$
9: **else**
10: $\ell \leftarrow \ell + 1$
11: **end if**
12: **end while**

Ausgabe: Mit Param. δ skaliert-k-Segment LLL-reduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters L

5.2.2 Iterierte k-Segment LLL-Reduktion

Die *iterierte k-Segment LLL-Reduktion* ist in [Koy und Schnorr, 2001a] vorgestellt worden. Es ist eine natürliche Erweiterung des Konzeptes der Segmente. Die Segmente werden in Unter-Segmente partitioniert. Diese können wiederrum weiter in Unter-Unter-Segmente partitioniert werden und so weiter. Durch diese iterative *divide & conquer* Strategie kann man eine asymptotisch bessere Laufzeit zeigen [Koy und Schnorr, 2001a]. Eine effiziente Implementierung des Verfahrens ist aufgrund des rekursiven Vorgehens sehr aufwendig. Insbesondere falls eine *skalierte iterierte k-Segment LLL-Reduktion* programmiert werden soll, hat man sehr viel zu verwalten (etwa verschiedene Skalierungen der unterschiedlichen Sub-Segmente). Im Rahmen dieser Diplomarbeit haben wir deshalb auf eine prototypische Umsetzung verzichtet, zumal die experimentellen Ergebnisse – mit der skalierten k-Segment LLL-Reduktion – uns vermuten lassen, dass die iterierte k-Segment LLL-Reduktion in der Praxis bei den betrachteten Gitterbasen keinen signifikanten Laufzeitgewinn bringen würde. Daneben benötigen die Gitterbasen, die wir mit den uns zur Verfügung stehenden Rechnern reduzieren können, bereits viel Speicherplatz. Noch größere Dimensionen, bei denen die iterierte Segment LLL-Reduktion „greifen“ würde, können wir zur Zeit nicht Speichern – zumindest nicht auf unseren Rechnern.

5.3 Primal/Duale Segmentreduktion

Die *primal/duale Segmentreduktion* geht auf Henrik Koy zurück [Koy, 2002]. Ähnlich wie bei der k-Segment LLL-Reduktion teilt man auch hier die zu reduzierende Basismatrix B bzw. die dazu isometrische Matrix R in Segmente auf und rechnet in

lokalen Koordinaten. Der eigentliche Reduktionsvorgang geschieht dabei in *primalen* und *dualen* Segmenten und unterscheidet sich grundlegend von der k-Segment LLL-Reduktion.

Gegeben sei die Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ eines Gitters, $n = k \cdot m$, mit der Faktorisierung[4] $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] = QR$ (siehe Gleichung (2.4) auf Seite 19).

Bei der k-Segment LLL-Reduktion betrachtet man $2k \times 2k$ Submatrizen $R_{\ell,\ell+1}$ von R, da stets zwei aufeinander folgende Segmente $B_\ell, B_{\ell+1} \in \mathbb{R}^{d\times k}$ der Basismatrix B in lokalen Koordinaten reduziert werden (siehe Abschnitt 5.2 auf Seite 56 und **Abbildung 5.1** auf Seite 58). Das heißt für je *zwei* Segmente von B wird *eine* Submatrix von R bearbeitet.

Bei der primal/dualen Segmentreduktion betrachtet man aber zu *zwei* Segmenten $B_\ell, B_{\ell+1} \in \mathbb{R}^{d\times k}$ auch *zwei* $k \times k$ Submatrizen $R_\ell, R_{\ell+1} \in \mathbb{R}^{k\times k}$ von R. Wir benötigen für jedes Segment $B_\ell \in \mathbb{Z}^{d\times k}$ die Darstellung in lokalen Koordinaten (analog zu Gleichung (3.5) auf Seite 35). Setzen wir

$$i_l := k \cdot (\ell - 1)$$

so entspricht für $\ell \in \{1, 2, \ldots, m\}$ die $k \times k$ Submatrix

$$R_\ell := \sigma_{i_l,k}\big(R^{i_l,k}\big) = [r_{k(\ell-1)+i,k(\ell-1)+j}]_{1\le i,j\le k}$$
$$= \begin{bmatrix} r_{i_l+1,i_l+1} & r_{i_l+1,i_l+2} & \cdots & r_{i_l+1,i_l+(k-1)} & r_{i_l+1,i_l+k} \\ 0 & r_{i_l+2,i_l+2} & \cdots & r_{i_l+2,i_l+(k-1)} & r_{i_l+2,i_l+k} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_l+(k-1),i_l+(k-1)} & r_{i_l+(k-1),i_l+k} \\ 0 & \cdots & 0 & 0 & r_{i_l+k,i_l+k} \end{bmatrix} \in \mathbb{R}^{k\times k}.$$

von $R = [r_{i,j}]_{1\le i,j\le n} \in \mathbb{R}^{n\times n}$ bei der primal/dualen Segmentreduktion dem Segment $B_\ell \in \mathbb{Z}^{d\times k}$ (siehe auch **Abbildung 5.2** auf der nächsten Seite). Nach Gleichung (2.5) auf Seite 19 ergeben sich die Einträge in R_ℓ aus den Gram-Schmidt-Koeffizienten $\mu_{k\ell+i,k\ell+j}$ und den Höhen $\|\widehat{\boldsymbol{b}}_{k\ell+i}\|^2$. Wir bezeichnen die lokale Basis R_ℓ als *primales Segment*.

5.3.1 Lemma (Duale Basismatrix) *Sei $B = QR$ die Faktorisierung der Basismatrix $B \in \mathbb{R}^{d\times n}$ des Gitters $L = \mathcal{L}(B)$, das heißt die Matrix $R \in \mathbb{R}^{n\times n}$ ist isometrisch zu B. Die Matrix*

$$S := (R^{-1})^{\mathsf{T}} \in \mathbb{R}^{n\times n} \tag{5.5}$$

ist eine Basismatrix von L^* (L^* ist das duale Gitter zu L).

Beweis von 5.3.1:

Die Behauptung folgt unmittelbar aus der Aussage c) von Satz 2.5.2 auf Seite 23.

❏

[4] Mit numerischen Verfahren zur QR-Faktorisierung beschäftigen wir uns in Abschnitt 4.2 auf Seite 44.

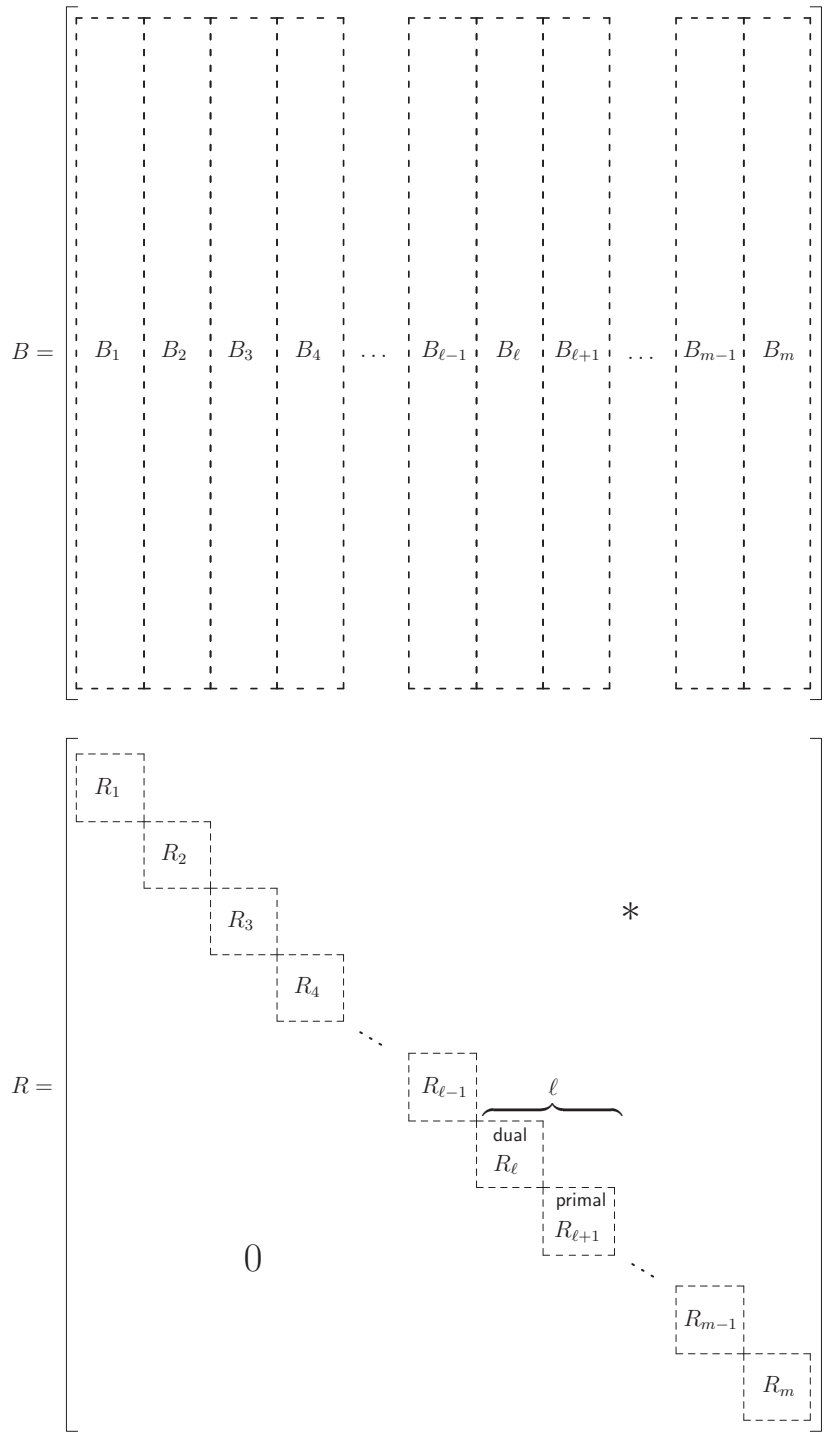


Abbildung 5.2: Einteilung einer Basismatrix B und der entsprechenden R Matrix in Segmente bei der primal/dualen Segmentreduktion

Aus der Matrix R lässt sich nach Lemma 5.3.1 auf Seite 63 in einfacher Weise eine duale Basis S berechnen. Weil R eine obere Dreiecksmatrix ist (siehe Gleichung (2.5) auf Seite 19), hat S untere Dreieckfsorm:

$$S = \begin{bmatrix} s_{1,1} & 0 & \cdots & 0 & 0 \\ s_{2,1} & s_{2,2} & & 0 & 0 \\ \vdots & & \ddots & \vdots & \vdots \\ s_{n-1,1} & s_{n-1,2} & \cdots & s_{n-1,n-1} & 0 \\ s_{n,1} & s_{n,2} & \cdots & s_{n,n-1} & s_{n,n} \end{bmatrix} \in \mathbb{R}^{k\times k}.$$

Das *duale Segment* $S_\ell, \ell \in \{1, 2, \ldots, m\}$, von S berechnet man einfach aus R_ℓ:

$$S_\ell := (R_\ell^{-1})^{\mathsf{T}} = [s_{k(\ell-1)+i,k(\ell-1)+j}]_{1\le i,j\le k} \tag{5.6}$$
$$= \begin{bmatrix} s_{i_l+1,1} & 0 & \cdots & 0 & 0 \\ s_{i_l+2,1} & s_{i_l+2,2} & & 0 & 0 \\ \vdots & & \ddots & \vdots & \vdots \\ s_{i_l+k-1,1} & s_{i_l+k-1,2} & \cdots & s_{i_l+k-1,k-1} & 0 \\ s_{i_l+k,1} & s_{i_l+k,2} & \cdots & s_{i_l+k,n-1} & s_{i_l+k,n} \end{bmatrix} \in \mathbb{R}^{k\times k}.$$

5.3.2 Lemma *Es gelten die folgenden Aussagen:*

a) Die lokalen Basen R_ℓ und S_ℓ sind dual.

b) $\mathsf{det}(R_\ell) \cdot \mathsf{det}(S_\ell) = 1$.

c) Für jede Basistransformation $T \in \mathsf{SL}_k(\mathbb{Z})$ gilt: $R_\ell \cdot (T^{-1})^{\mathsf{T}}$ und $S_\ell \cdot T$ sind duale Basen.

d) Wenn im dualen Segment S_ℓ der Vektor $\boldsymbol{s}_{k\ell}$ ein lokal kürzester Gittervektor ist, dann ist $|r_{k\ell,k\ell}| = \|\widehat{\boldsymbol{b}}_{k\ell}\|$ maximal.

5.3.3 Definition (Einfache primal/duale Segmentreduktion) *Sei $B = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{d\times n}$, $n = k{\cdot}m$, eine geordnete Gitterbasis. Gegeben sei die Faktorisierung $B = QR$, $R = [r_{i,j}]_{1\le i,j\le n}$. Die Gitterbasis B ist* primal/dual k-Segment reduziert *mit $\delta \in (\frac{1}{4}, 1]$, wenn für jedes $\ell = 1, 2, \ldots, m-1$ gilt: Die mit den Eigenschaften PD1, PD2 und PD3 reduzierten Segmente $B_\ell^\iota := [\boldsymbol{b}_{k(\ell-1)+1}^\iota\ \boldsymbol{b}_{k(\ell-1)+2}^\iota\ \ldots\ \boldsymbol{b}_{k\ell}^\iota] := [\boldsymbol{b}_{k(\ell-1)+1}^*\ \boldsymbol{b}_{k(\ell-1)+2}^*\ \ldots\ \boldsymbol{b}_{k\ell}^*] \cdot T_\ell^D$, $T_\ell^D \in \mathsf{SL}_k(\mathbb{Z})$, und $B'_{\ell+1} := [\boldsymbol{b}'_{k\ell+1}\ \boldsymbol{b}'_{k\ell+2}, \ldots, \boldsymbol{b}'_{k\ell+k}] := B_{\ell+1} \cdot T_{\ell+1}^P$, $T_{\ell+1}^P \in \mathsf{SL}_k(\mathbb{Z})$, erfüllen an der Grenze $\boldsymbol{b}_{k\ell}^\iota, \boldsymbol{b}'_{k\ell+1}$ die LLL-Reduktionsbedingung PD4.*

PD1: *Die in der Basis ersetzten Segmente $B_\ell^\iota, B'_{\ell+1}$ sind längenreduziert.*

PD2: *Der Basisvektor $\boldsymbol{b}_{k\ell}^\iota$ ist im Segment B_ℓ^ι lokal kürzester Gittervektor, das heißt in S_ℓ ist $\boldsymbol{s}_{k\ell}$ minimal.*

PD3: *Der Basisvektor $\boldsymbol{b}'_{k\ell+1}$ ist im Segment $B'_{\ell+1}$ lokal kürzester Gittervektor, das heißt in $R_{\ell+1}$ ist $|r_{k\ell+1,k\ell+1}| = \|\widehat{\boldsymbol{b}}_{k\ell+1}\|$ minimal.*

PD4: $\delta\|\widehat{\boldsymbol{b}^\iota}_{k\ell}\|^2 \le \mu_{k\ell+1,k\ell}^2\|\widehat{\boldsymbol{b}^\iota}_{k\ell}\|^2 + \|\widehat{\boldsymbol{b}'}_{k\ell+1}\|$.

Bedingung PD2 besagt in anderen Worten, dass $|r_{k\ell,k\ell}| = \|\widehat{\boldsymbol{b}}_{k\ell}\|$ in R_ℓ maximal ist.

Der **Algorithmus 5.4** auf der nächsten Seite reduziert eine Gitterbasis gemäß der Definition 5.3.3. Veranschaulicht wird die Reduktion in primalen/dualen Segmenten in **Abbildung 5.2** auf der vorherigen Seite.

Algorithmus 5.4 Einfache primal/duale Segmentreduktion

Eingabe: (LLL-reduzierte) Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L ; $n = km$; $\delta \in (\frac{1}{4}, 1]$

1: $\ell \leftarrow 1$
2: **while** $\ell \leq m-1$ **do**
3: `primal-dual(`ℓ`)` /* `primal-dual` nur für $\ell \in \{1, 2, \ldots, m-1\}$ aufrufen */
4: /* `primal-dual(`ℓ`)` bearbeitet B_ℓ dual, und $B_{\ell+1}$ primal (siehe auch **Abbildung 5.2** auf Seite 64) */
5: **if** $\big(\ell > 1$ **and** $\delta\|\widehat{\boldsymbol{b}}_{k\ell}\|^2 > \mu_{k\ell+1,k\ell}^2\|\widehat{\boldsymbol{b}}_{k\ell}\|^2 + \|\widehat{\boldsymbol{b}}_{k\ell+1}\|\big)$ **then**
6: LLL-reduziere $B_\ell, B_{\ell+1}$
7: $\ell \leftarrow \ell - 1$
8: **else**
9: $\ell \leftarrow \ell + 1$
10: **end if**
11: **end while**

Ausgabe: Mit Parameter δ einfach primal/dual Segmentreduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters L

Algorithmus 5.5 `primal-dual(`ℓ`)`

Eingabe: $\ell \in \{1, 2, \ldots, m-1\}$ und Vektoren von $B_\ell, B_{\ell+1}$ (siehe auch folgenden Text)

1: Ausgehend von einer gegebenen Orthogonalisierung $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell-1)}$ der bereits primal/dual k-Segment reduzierten Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k(\ell-1)}$ (also der Segmente $B_1, B_2, \ldots, B_{\ell-1}$) berechne die Orthogonalisierung $\boldsymbol{r}_{k(\ell-1)+1}, \boldsymbol{r}_{k(\ell-1)+2}, \ldots, \boldsymbol{r}_{k(\ell+1)}$ zu $B_\ell, B_{\ell+1}$ (siehe Bemerkung 4.2.9 auf Seite 49).
2: Aus der Matrix $[\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell+1)}]$ werden die Matrizen $S_\ell = \big([r_{k(\ell-1)+i,k(\ell-1)+j}]_{1\leq i,j\leq k}^{-1}\big)^{\mathsf{T}} \in \mathbb{R}^{k\times k}$ und $R_{\ell+1} = [r_{k\ell+i,k\ell+j}]_{1\leq i,j\leq k} \in \mathbb{R}^{k\times k}$ bestimmt. S_ℓ entspricht B_ℓ, und $R_{\ell+1}$ entspricht $B_{\ell+1}$ (siehe **Abbildung 5.2** auf Seite 64).
3: /* In Schritt 4 und 5 müssen linear abhängige Vektoren eliminiert werden, siehe dazu auch [Seite 27 in Ritter, 1997]. */
4: Bestimme mittels `ENUM`-Algorithmus (siehe Seite 33) den in S_ℓ kürzesten Gittervektor $\boldsymbol{s}$, füge $\boldsymbol{s}$ an *letzter* Stelle in S_ℓ ein und eliminiere einen in diesem Segment linear abhängigen Gittervektor (der jetzt zwangsläufig vorhanden ist). Speichere Basistransformationen in $T_D \in \mathsf{SL}_k(\mathbb{Z})$. $T \leftarrow \Big([T_D]^{-1}\Big)^{\mathsf{T}}$. /* Man beachte die *Primalisierung* von T_D */
5: $B_\ell \leftarrow B_\ell T$
6: Bestimme mittels `ENUM`-Algorithmus (siehe Seite 33) den in $R_{\ell+1}$ kürzesten Gittervektor $\boldsymbol{r}$, füge $\boldsymbol{r}$ an *erster* Stelle in $R_{\ell+1}$ ein und eliminiere einen in diesem Segment linear abhängigen Gittervektor (der jetzt zwangsläufig vorhanden ist). Speichere Basistransformationen in $T' \in \mathsf{SL}_k(\mathbb{Z})$.
7: $B_{\ell+1} \leftarrow B_{\ell+1}T'$
8: Orthogonalisiere wie in Schritt 1.

Ausgabe: Siehe obigen Text

5.3.1 Segmentüberlappung und Reduktion in Phasen

In Algorithmus **Algorithmus 5.4** auf der vorherigen Seite bzw. in **Algorithmus 5.5** auf der vorherigen Seite werden auf Stufe ℓ, das duale Segment S_ℓ und das primale Segment $R_{\ell+1}$ bearbeitet. Dabei sind die Segmente disjunkt. KOY betrachtet auch die *Überlappung* des dualen und primalen Segments. Im folgenden formalisieren wir das und führen die in **Algorithmus 5.7** auf Seite 71 verwendete Notation ein. Dieser Algorithmus zur *pimal/dualen Reduktion mit überlappenden Segmenten* berücksichtigt neben den Bedingungen aus Definition 5.3.3 auf Seite 65 auch noch zwei zusätzliche Bedingungen: Die Duale-Reduktionsbedingung und die Primale-Reduktionsbedingung (siehe **Algorithmus 5.7** auf Seite 71). Die Gründe dafür sollen an dieser Stelle nicht erörtert werden – interessierte Leser seien an [Koy, 2002] verwiesen. Um die Indizes übersichtlicher schreiben zu können, setzen wir wieder

$$i_l := k \cdot (\ell - 1) \quad \text{und} \quad i_m := i_l + k. \tag{5.7}$$

Man erinnere sich an die $\sigma_{s,k}\big(R^{s,k}\big)$ Notation aus Abschnitt 3.5 auf Seite 33 ff. Mit den Teilmatrizen

$$R_\ell = \sigma_{i_l,k}\big(R^{i_l,k}\big) = \begin{bmatrix} r_{i_l+1,i_l+1} & r_{i_l+1,i_l+2} & \cdots & r_{i_l+1,i_m-1} & r_{i_l+1,i_m} \\ 0 & r_{i_l+2,i_l+2} & \cdots & r_{i_l+2,i_m-1} & r_{i_l+2,i_m} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_m-1,i_m-1} & r_{i_m-1,i_m} \\ 0 & \cdots & 0 & 0 & r_{i_m,i_m} \end{bmatrix} \in \mathbb{R}^{k\times k},$$

$$R_{\ell+1} = \sigma_{i_m,k}\big(R^{i_m,k}\big) = \begin{bmatrix} r_{i_m+1,i_m+1} & r_{i_m+1,i_m+2} & \cdots & r_{i_m+1,i_m+(k-1)} & r_{i_m+1,i_m+k} \\ 0 & r_{i_m+2,i_m+2} & \cdots & r_{i_m+2,i_m+(k-1)} & r_{i_m+2,i_m+k} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_m+(k-1),i_m+(k-1)} & r_{i_m+(k-1),i_m+k} \\ 0 & \cdots & 0 & 0 & r_{i_m+k,i_m+k} \end{bmatrix} \in \mathbb{R}^{k\times k}$$

erhalten wir in der Kästchenschreibweise

$$R^{\text{loc}} := \sigma_{i_l,2k}\big(R^{i_l,2k}\big) = \begin{bmatrix} R_\ell & * \\ 0 & R_{\ell+1} \end{bmatrix} \in \mathbb{R}^{2k\times 2k}. \tag{5.8}$$

Die lokalen Segmente $R_\ell, R_{\ell+1} \in \mathbb{R}^{k\times k}$ sollen einander überlappen. Dazu wird R_ℓ „nach rechts und unten“ und $R_{\ell+1}$ wird „nach links und oben“ erweitert. Formal geschrieben ergeben sich $R_\ell^+, R_{\ell+1}^- \in \mathbb{R}^{k+1\times k+1}$, die im Vergleich zu den ursprünglichen Matrizen

$R_\ell, R_{\ell+1}$ in den Dimensionen um 1 vergrößert sind und folgende Darstellungen haben:

$$R_\ell^+ := \sigma_{i_l,k+1}\big(R^{i_l,k+1}\big) = \begin{bmatrix} r_{i_l+1,i_l+1} & r_{i_l+1,i_l+2} & \cdots & r_{i_l+1,i_m-1} & r_{i_l+1,i_m} & \vdots & r_{i_l+1,i_m+1} \\ 0 & r_{i_l+2,i_l+2} & \cdots & r_{i_l+2,i_m-1} & r_{i_l+2,i_m} & \vdots & r_{i_l+2,i_m+1} \\ \vdots & & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & r_{i_m-1,i_m-1} & r_{i_m-1,i_m} & \vdots & r_{i_m-1,i_m+1} \\ 0 & \cdots & 0 & 0 & r_{i_m,i_m} & \vdots & r_{i_m,i_m+1} \\ \multicolumn{7}{c}{\dotfill} \\ 0 & \cdots & 0 & 0 & 0 & & r_{i_m+1,i_m+1} \end{bmatrix} \in \mathbb{R}^{k+1\times k+1},$$

$$R_{\ell+1}^- := \sigma_{i_m-1,k+1}\big(R^{i_m-1,k+1}\big) \in \mathbb{R}^{k+1\times k+1} = \begin{bmatrix} r_{i_m,i_m} & r_{i_m,i_m+1} & r_{i_m,i_m+2} & \cdots & r_{i_m,i_m+(k-1)} & r_{i_m,i_m+k} \\ \multicolumn{6}{c}{\dotfill} \\ 0 & r_{i_m+1,i_m+1} & r_{i_m+1,i_m+2} & \cdots & r_{i_m+1,i_m+(k-1)} & r_{i_m+1,i_m+k} \\ 0 & 0 & r_{i_m+2,i_m+2} & \cdots & r_{i_m+2,i_m+(k-1)} & r_{i_m+2,i_m+k} \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & r_{i_m+(k-1),i_m+(k-1)} & r_{i_m+(k-1),i_m+k} \\ 0 & 0 & \cdots & 0 & 0 & r_{i_m+k,i_m+k} \end{bmatrix}.$$

Man beachte, dass sowohl R_ℓ^+ als auch R_ℓ^- in einfacher Weise aus der Matrix R^{loc} (siehe Gleichung (5.8) auf der vorherigen Seite) zu bestimmen sind. Analog zu Gleichung (5.6) auf Seite 65 setzen wir

$$S_\ell^+ := \Big(\big[R_\ell^+\big]^{-1}\Big)^{\mathsf{T}} \in \mathbb{R}^{k+1\times k+1}.$$

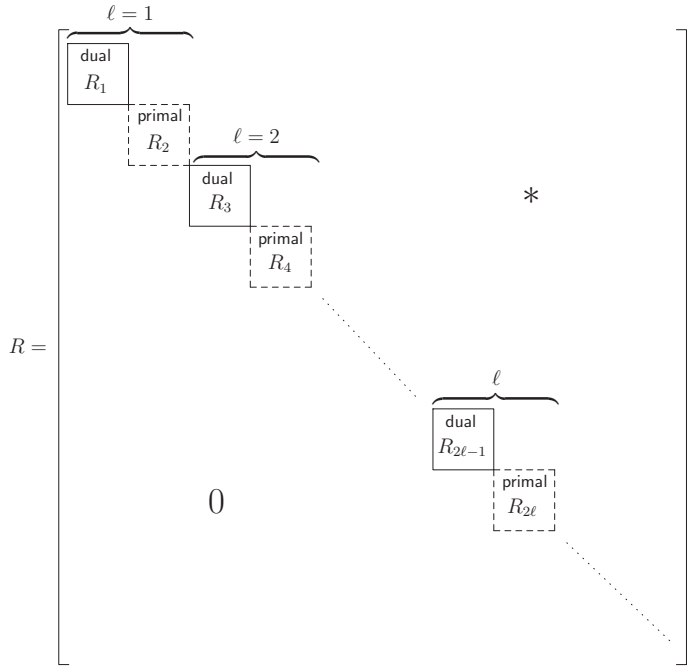
Das zu R_ℓ^+ gehörende Segment der Gitterbasis B ist

$$B_\ell^+ := [\boldsymbol{b}_{k(\ell-1)+1} \;\; \boldsymbol{b}_{k(\ell-1)+2} \;\; \ldots \;\; \boldsymbol{b}_{k\ell} \;\; \boldsymbol{b}_{k\ell+1}] \in \mathbb{R}^{d\times k+1},$$

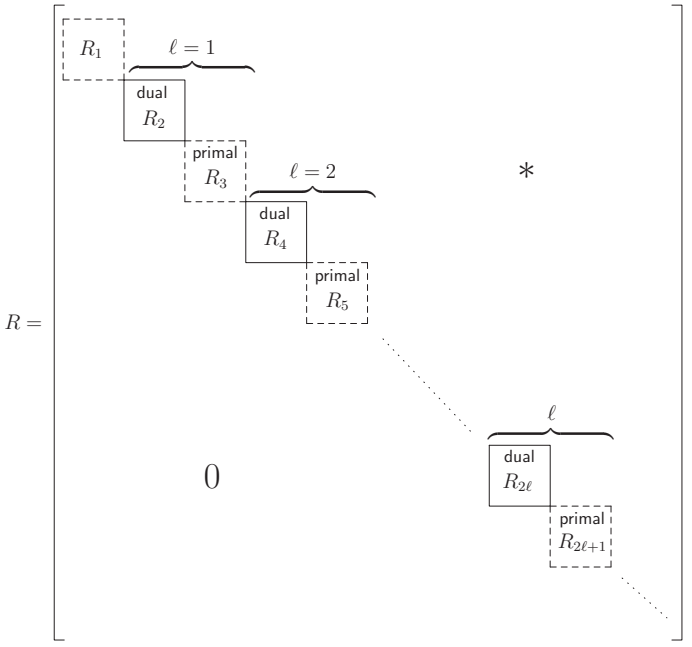
und $R_{\ell+1}^-$ entspricht konsequenterweise

$$B_{\ell+1}^- := [\boldsymbol{b}_{k\ell} \;\; \boldsymbol{b}_{k\ell+1} \;\; \boldsymbol{b}_{k\ell+2} \;\; \ldots \;\; \boldsymbol{b}_{k(\ell+1)}] \in \mathbb{R}^{d\times k+1}.$$

Ein nahliegender Ansatz zur *Parallelisierung* der primal/dualen Segmentreduktion ist die Aufteilung der Iterationen in zwei *Phasen* $p \in \{0,1\}$. Die Phasen wechseln sich ab und $\lfloor \frac{m}{2} \rfloor$ Prozessoren können alle in der aktuellen Phase relevanten Segmente gleichzeitig bearbeiten. Der Vorschlag stammt von Henrik Koy [Koy, 2002]. Das Vorgehen beschreibt **Algorithmus 5.6** auf Seite 70 und **Abbildung 5.3** auf der nächsten Seite veranschaulicht es.



(a) Reduktion der Segmente in der Phase $p = 0$



(b) Reduktion der Segmente in der Phase $p = 1$

Abbildung 5.3: Reduktion der primalen/dualen Segmente in zwei Phasen

5.3.4 Bemerkung *Für eine mit **Algorithmus 5.7** auf der nächsten Seite reduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$, $n = km$, des Gitters L gilt nach [Koy, 2002] die Abschätzung:*

$$\lambda_1^2\big(\mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_k)\big) \leq \left(\frac{\gamma_{k+1}^2}{\delta^2}\right)^{\frac{n}{k}+\frac{n}{k^2}-\left(1+\frac{1}{k}\right)},$$

falls die Eingabe Basis mit dem LLL-Verfahren vorreduziert ist. Die Laufzeit ist

$$O(n^3 d \log_{1/\delta} 2) + n^3 k^{O(k)}.$$

5.3.5 Bemerkung (Vorreduzierte Gitterbasen als Eingabe) *Für die Abschätzungen (Laufzeit, Reduktionsgüte) des Verfahrens zur primal/dualen Segmentreduktion verlangt* Koy*, dass die Eingabe Gitterbasis mit dem LLL-Verfahren vorreduziert ist. Statt einer LLL-Reduktion führen wir in der Praxis die skalierte Segment LLL-Reduktion vor der primal/dualen Reduktion durch. Die gewählten Segmentgrößen k_1 (für die skalierte Segment LLL-Reduktion) und k_2 (für die primal/duale Reduktion) müssen nicht unbedingt übereinstimmen, obwohl es sich anbietet $k_1 = k_2$ zu benutzen.*

Algorithmus 5.6 Erweiterte primal/duale Segmentreduktion in zwei Phasen

---

Eingabe: (LLL-reduzierte) Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n \in \mathbb{Z}^d$ des Gitters L ; $n = km$; $\delta \in (\frac{1}{4}, 1]$

1: $p \leftarrow 1$
2: $F_p \leftarrow$ true /∗ Nehme an, dass in Phase $p = 1$ die Basis transformiert worden sei ∗/
3: **repeat**
4: /∗ Wiederhole solange bis in beiden Phasen keine Segmentübergreifenden Basis-Transformationen erfolgt sind. Die Flags F_0 und F_1 sind Indikatoren ob in Phase $p \in \{0, 1\}$ solche Transformationen stattgefunden haben:

$F_p =$ false $\overset{\text{def}}{\Longleftrightarrow}$ keine *Segmentübergreifenden* Transformationen in Phase p durchgeführt.

∗/
5: $p \leftarrow p + 1 \pmod 2$ /∗ Phasenwechsel ∗/
6: **for** $\ell = 1, 2, \ldots, \lfloor \frac{m}{2} \rfloor$ **do**
7: $F_p \leftarrow$ `primal-dual-extended`$(2\ell - 1 + p,\ \delta)$ /∗ Bearbeite: $B_{2\ell-1+p}$ dual, $B_{2\ell+p}$ primal; vergleiche mit **Abbildung 5.3** auf der vorherigen Seite ∗/
8: /∗ `primal-dual-extended` setzt F_p auf true falls Segmentübergreifende Basis-Transformationen stattgefunden haben, ansonsten gilt $F_p =$ false. ∗/
9: **end for**
10: /∗ $\lfloor \frac{m}{2} \rfloor$ Prozessoren könnten die in Schritt 6–9 betrachteten Segmente *parallel* bearbeiten, da die Doppelsegmente in der Phase p für alle ℓ disjunkt sind (siehe **Abbildung 5.3** auf der vorherigen Seite). ∗/
11: **until** ($F_0 =$ false **and** $F_1 =$ false)

Ausgabe: Mit Parameter δ primal/dual Segmentreduzierte Basis $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ des Gitters L

Algorithmus 5.7 `primal-dual-extended`(ℓ, δ)

Eingabe: $\ell \in \{1, 2, \ldots, m-1\}$; $\delta \in (\frac{1}{4}, 1]$; siehe folgenden Text

1: Ausgehend von einer gegebenen Orthogonalisierung $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell-1)}$ der bereits primal/dual Segment reduzierten Basisvektoren $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k(\ell-1)}$ (also der Segmente $B_1, B_2, \ldots, B_{\ell-1}$) berechne die Orthogonalisierung $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{k(\ell+1)}$ der Segmente $B_\ell, B_{\ell+1}$ (siehe Bemerkung 4.2.9 auf Seite 49). Bestimme R^{loc} für die lokalen Rechnungen.
2: /∗ In den folgenden Schritten werden Subroutinen $\texttt{ENUM}_{\text{dual}}$ bzw. $\texttt{ENUM}_{\text{primal}}$ benutzt. Ihre Wirkung entspreche den Anweisungen in Schritt 4 ($\texttt{ENUM}_{\text{dual}}$) bzw. Schritt 6 ($\texttt{ENUM}_{\text{primal}}$) in **Algorithmus 5.5** auf Seite 66. Rückgabeparameter der beiden Routinen seien die (primalisierte) Transformationsmatrix T und die Länge/Höhe des (neu) gefundenen kürzesten Gittervektors. ∗/
3: $F_{\text{trans}} \leftarrow$ false /∗ Flag: $F_{\text{trans}} =$ false $\overset{\text{def}}{\Longleftrightarrow}$ es wurden keine *Segmentübergreifenden* Transformationen durchgeführt. ∗/
4: **repeat**
5: /∗ Wiederhole bis in dem Doppelsegment $B_\ell, B_{\ell+1}$ alle Reduktionsbedingungen erfüllt sind ∗/
6: $F_{\text{cond}} \leftarrow$ true /∗ Flag: $F_{\text{cond}} =$ true $\overset{\text{def}}{\Longleftrightarrow}$ in dem Doppelsegment sind alle Reduktionsbedingungen erfüllt. ∗/
7: $F_{\text{trans}} \leftarrow \texttt{LLL}(R^{\text{loc}}, \delta)$ /∗ Falls lokal transformiert wird, so sei $F_{\text{trans}} =$ true von `LLL` gesetzt, andernfalls gelte $F_{\text{trans}} =$ false. ∗/
8: $(T, \|\widehat{\boldsymbol{b}}^d_{k\ell}\|) \leftarrow \texttt{ENUM}_{\text{dual}}(S_\ell)$; $B_\ell \leftarrow B_\ell \cdot T$; $R_\ell \leftarrow R_\ell \cdot T$
9: $(T, \|\widehat{\boldsymbol{b}}^p_{k\ell+1}\|) \leftarrow \texttt{ENUM}_{\text{primal}}(R_{\ell+1})$; $B_{\ell+1} \leftarrow B_{\ell+1} \cdot T$; $R_{\ell+1} \leftarrow R_{\ell+1} \cdot T$
10: Orthogonalisiere wie in Schritt 1.
11: **if** $(\delta \cdot r^2_{k\ell,k\ell} > r^2_{k\ell+1,k\ell} \cdot r^2_{k\ell+1,k\ell+1})$ **then**
12: /∗ Die Lovász-Bedingung gilt nicht an Segmentgrenze ∗/
13: $F_{\text{cond}} \leftarrow$ false
14: $F_{\text{trans}} \leftarrow \texttt{LLL}(R^{\text{loc}}, \delta)$ /∗ Falls lokal transformiert wird, so setze `LLL` $F_{\text{trans}} =$ true, andernfalls gelte $F_{\text{trans}} =$ false. ∗/
15: **else**
16: /∗ Die Lovász-Bedingung ist an den Segmentgrenzen erfüllt. Überprüfe in Schritt 18 die duale-Reduktionsbedingung und nur falls diese auch erfüllt ist, wird in Schritt 24 die primale-Reduktionsbedingung verifiziert. ∗/
17: $(T, \|\widehat{\boldsymbol{b}}^d_{k\ell+1}\|) \leftarrow \texttt{ENUM}_{\text{dual}}(S^+_\ell)$
18: **if** $(\frac{1}{\delta} \cdot r^2_{k\ell+1,k\ell+1} < \|\widehat{\boldsymbol{b}}^d_{k\ell+1}\|)$ **then**
19: /∗ Duale-Reduktionsbedingung ist verletzt ∗/
20: $F_{\text{cond}} \leftarrow$ false; $B^+_\ell \leftarrow B^+_\ell \cdot T$; $R^+_\ell \leftarrow R^+_\ell \cdot T$; $F_{\text{trans}} \leftarrow$ true
21: Orthogonalisiere wie in Schritt 1.
22: **else**
23: $(T, \|\widehat{\boldsymbol{b}}^p_{k\ell}\|) \leftarrow \texttt{ENUM}_{\text{primal}}(R^-_{\ell+1})$
24: **if** $(\delta \cdot r^2_{k\ell,k\ell} > \|\widehat{\boldsymbol{b}}^p_{k\ell}\|)$ **then**
25: /∗ Primale-Reduktionsbedingung ist verletzt ∗/
26: $F_{\text{cond}} \leftarrow$ false; $B^-_{\ell+1} \cdot T$; $R^-_{\ell+1} \cdot T$; $F_{\text{trans}} \leftarrow$ true
27: Orthogonalisiere wie in Schritt 1.
28: **end if**
29: **end if**
30: **end if**
31: **until** $F_{\text{cond}} =$ true
32: **if** $F_{\text{trans}} =$ true **then**
33: Übertrage die lokalen Transformationen auf das gesamte Gitter und orthogonalisiere/längenreduziere.
34: **end if**

Ausgabe: $F_{\text{trans}} \in \{\text{true}, \text{false}\}$ und Doppelsegment $B_\ell, B_{\ell+1}$ in dem alle (drei) Reduktionsbedingungen erfüllt sind

6 `latred` und die Implementierung der Gitterbasenreduktion in Segmenten

Wir haben das Kommandozeilen Programm `latred`[1] in `C/C++` programmiert, um Experimente einfach durchführen und auswerten zu können. Darin sind die skalierte LLL-Reduktion [Koy und Schnorr, 2001b], die skalierte k-Segment LLL-Reduktion [Koy und Schnorr, 2001b] und die primal/duale Segmentreduktion [Koy, 2002] sowie die Reduktionsverfahren aus [Shoup, 2001] integriert. Unser Programm bietet auch die Möglichkeit, bestimmte reduzierte/nicht-reduzierte Gitterbasenpaare zu erzeugen. Dies ist nützlich, um Eingabe- bzw. Referenzbasen für Experimente zu haben. Mit `latred` sind alle relevanten Parameter für ein gewähltes Reduktionsverfahren beim Programmaufruf über die Kommandozeile einstellbar. So kann beispielsweise angegeben werden, welcher Gleitpunktzahlen-Datentyp bei der Reduktion zu verwenden ist. Das Programm kann Protokolldateien des Reduktionsverlaufes erstellen, anhand derer man die Reduktion verfolgen kann. Über seine Benutzung informiert Anhang A auf Seite 107. Eine Liste der Quellcode-Dateien mit einer kurzen Beschreibung ist in Anhang B auf Seite 113 abgedruckt. Dort werden auch Bezugsmöglichkeiten des `latred`-Quellcodes angegeben. **Tabelle 6.1** auf der nächsten Seite gibt eine Übersicht der Rechnerarchitekturen, Betriebssysteme und Compiler auf bzw. mit denen wir `latred` erfolgreich erstellt und benutzt haben.

Das Augenmerk dieses Kapitels liegt primär auf einer Beschreibung der von uns programmierten Methoden, welche die *Gitterbasenreduktion in Segmenten* realisieren. Dazu zählen auch effiziente Routinen zur Householder-Reflexion und Givens-Rotation. Wir verzichten dabei bewusst auf das Abdrucken von `C/C++` Quellcode. Stattdessen werden die Algorithmen in Pseudocode angegeben oder es erfolgt eine Beschreibung in Worten. Dadurch sollte sich die konkrete `C/C++` Implementierung bei vorliegendem Quellcode leicht erschließen. Auf die Programmierung *aller* Bestandteile von `latred` können wir hier nicht eingehen.

In Abschnitt 6.1 auf der nächsten Seite stellen wir die Bibliotheken vor, die ergänzend zu den Standard-Libraries bei der Implementierung von `latred` bzw. der Algorithmen zur Gitterbasenreduktion benutzt wurden. Ohne diese Bibliotheken wäre eine ähnlich effiziente Realisierung von `latred` kaum möglich gewesen. In diesem Abschnitt gehen wir auch auf benutzte Datentypen ein.

Die Konzeption von `latred` wird in Abschnitt 6.2 auf Seite 79 erläutert.

[1] `latred` steht für `lat`tice `red`uction (im englischen bezeichnet man ein Gitter als lattice).

Der Hauptbestandteil von `latred` ist das Klassentemplate `CLattice`. Es enthält Datenstrukturen für die benötigten Matrizen und Vektoren über beliebigen Gleitpunktzahlen-Datentypen. In `CLattice` sind auch alle relevanten Methoden programmiert, die für unsere Gitter-Reduktionsalgorithmen von Belang sind. Daneben enthält `CLattice` auch Datenstrukturen und Methoden, um den Reduktionsverlauf und die Eigenschaften der Verfahren zu protokollieren und ausgeben zu können. Abschnitt 6.3 auf Seite 80 beschreibt die für die Reduktion in Segmenten wesentlichen Methoden. Wir passen die HRS-Routine aus [Koy und Schnorr, 2001b] den verwendeten Gleitpunktzahlen an.

Abschnitt 6.4 auf Seite 94 befasst sich mit Algorithmen, die zur Erzeugung von Gitterbasen dienen.

Rechnerarchitektur	Taktfrequenz	Hauptspeicher	Betriebssystem	Compiler
AMD Athlon	800 MHz	192 MByte 448 MByte	SuSE Linux 7.1 SuSE Linux 7.2 SuSE Linux 7.3 Win 98	GCC 2.95.2 GCC 2.95.3 GCC 2.95.3 VC++ 6.0
HP 9000/782	160 MHz	256 MByte	HP-UX	GCC 2.95.2
Intel Pentium II	350 MHz 400 MHz	64 MByte 128 MByte	RedHat Linux	

Tabelle 6.1: Benutzte Hard- und Software Umgebungen

6.1 Benutzte Bibliotheken

Wie in Abschnitt 4.3 auf Seite 53 erläutert, kann auf die Verwendung von Langzahlarithmetik Bibliotheken nicht verzichtet werden, falls man Gitterbasen mit sehr großen Einträgen reduzieren will. Neben den „Standard“ `C/C++`-Bibliotheken benutzten wir daher bei der Programmierung von `latred` auch die *Number Theoretic Library* (NTL). Die NTL profitiert von dem zusätzlichen Gebrauch der *GNU Multiple Precision Arithmetic Library* (GNU MP). Wir stellen die beiden Bibliotheken in Abschnitt 6.1.1 bzw. Abschnitt 6.1.2 auf der nächsten Seite kurz vor.

Die aktuelle Version unseres `latred`-Programms wurde unter Verwendung von NTL 5.2 in `C++` programmiert. Compiliert wurde es unter Einbindung von NTL 5.2 und GNU MP 4.0.

6.1.1 GNU MP – Langzahlarithmetik

Die GNU MP (*GNU Multiple Precision Arithmetic Library*) ist eine im Quellcode frei erhältliche Bibliothek für Langzahlarithmetik. Die aktuelle Version ist unter http://www.swox.com/gmp zu finden. Zur Zeit ist dies die Version GNU MP 4.0.1.

GNU MP ist eine in der Programmiersprache `C` und teilweise in Assembler geschriebene portable[2] Bibliothek für Arithmetik mit Zahlen, die beliebig[3] viele Bits haben können. Es gibt Funktionen für ganze Zahlen und rationale Zahlen, die beliebig groß oder klein sein können. Auch Gleitpunktzahlen mit variabler Anzahl von Präzisionsbits für die Mantisse werden unterstützt. Zu beachten ist jedoch, dass der in Abschnitt 4.1 auf Seite 41 erwähnte IEEE 754 Standard nicht eingehalten wird. Daher können Programme, die zwar auf dem selben Quellcode basieren, bei Berechnungen auf verschiedenen Rechnerarchitekturen unterschiedliche Resultate liefern.

Die GNU MP enthält für eine große Anzahl von Rechnerarchitekturen optimierte Low-Level Assembler-Routinen. Auch für die von uns eingesetzten Prozessoren steht solcher Assembler-Code zur Verfügung. Dadurch wird eine möglichst schnelle Ausführungsgeschwindigkeit der Routinen gewährleistet. Mit den Low-Level Routinen kommt der Benutzer bei der Programmierung im allgemeinen nicht in Berührung. Der Programmierer nutzt die zur Verfügung stehenden GNU MP `C`-Funktionen. Seit der Version 4.0 bietet die GNU MP Bibliothek auch eine `C++`-Überladung der arithmetischen Operatoren.

Die GNU MP Programmier-Funktionen werden nicht direkt von uns verwendet. Wir setzen aber die NTL-Bibliothek ein (siehe Abschnitt 6.1.2) und diese kann optional die GNU MP benutzen, um maximale Verarbeitungsgeschwindigkeit aus dem Prozessor zu holen auf dem unser Programm läuft.

6.1.2 NTL – Datenstrukturen und zahlentheoretische Algorithmen

Die NTL (*Number Theoretic Library*) von Victor Shoup ist im Quellcode frei erhältlich. Die aktuelle Version ist unter http://www.shoup.net/ntl zu finden. Zur Zeit ist dies die Version NTL 5.2.

NTL ist eine auf Geschwindigkeit optimierte[4] und portable `C++` Bibliothek. Im Gegensatz zur GNU MP ist sie keine Bibliothek, die sich nur auf Langzahlarithmetik spezialisiert hat. Vielmehr stellt NTL *Datenstrukturen* und *zahlentheoretische Algorithmen* für beliebig große ganze Zahlen und Gleitpunktzahlen mit beliebiger Präzision zur Verfügung. Unterstützt werden Vektoren und Matrizen sowie Polynome über den ganzen Zahlen und endlichen Körpern.

NTL enthält auch sehr schnelle Routinen zur Gitterbasenreduktion. Neben der LLL-Reduktion ist auch die geschnittene BKZ-Reduktion nach [Schnorr und Hörner, 1995] implementiert.

Unseren Quellcode einer früheren Version des `latred`-Programms haben wir Victor Shoup (dem Entwickler der NTL) zur Verfügung gestellt, zwecks einer eventuellen Integration der Verfahren aus [Koy und Schnorr, 2001b] in die NTL.

Einige wichtige Datentypen, die uns zur Verfügung stehen, beschreiben wir in Abschnitt 6.1.2.1 auf der nächsten Seite, Abschnitt 6.1.2.2 und Abschnitt 6.1.2.3 auf Seite 77.

[2] Bei Rechnerarchitekturen/Prozessoren, für die keine optimierten Assembler-Routinen gepflegt werden, kommt generischer `C`-Code zum Einsatz.
[3] Lediglich der verfügbare Speicher des Rechners beschränkt die Bitlänge der Zahlen.
[4] Insbesondere bei der Verwendung der GNU MP als Basisbibliothek für die Langzahlarithmetik.

6.1.2.1 Datentypen für Gleitpunktzahlen

In **Tabelle 6.2** sind die von uns benutzten Datentypen für Gleitpunktzahlen dargestellt[5] (siehe auch Definition 4.1.8 auf Seite 41). Dabei handelt es sich neben den Standard C/C++ Typen `float` und `double` auch um die NTL Typen `xdouble` und `quad_float`. Man vergleiche **Tabelle 6.2** auch mit **Tabelle 4.1** auf Seite 43 und beachte insbesondere, dass nur *ein* verstecktes Bit bei dem Datentyp `quad_float` zum Einsatz kommt, das heißt repräsentiert man `quad_float` Objekte durch zwei `double` Objekte – wie dies in der NTL geschieht – so ergeben sich *nicht* 106 Präzisionsbits für die Mantisse, sondern nur $2 \cdot 52 + 1 = 105$.

Datentyp	Parameter[6]					
	Anzahl der Bits für					
	eine Variable	das Vorzeichen	die Mantisse	den Exponenten	$e_{\min}$	$e_{\max}$
`float`	32	1	23 ($N = 24$)	8	-125	$+128$
`double`	64	1	52 ($N = 53$)	11	-1022	$+1023$
`quad_float`	105	1	104 ($N = 105$)	11	-1022	$+1023$
`xdouble`[7]	?	1	52 ($N = 53$)	≥ 32	?	?

Tabelle 6.2: Einige C/C++ und NTL Datentypen für Gleitpunktzahlen

Wie eben angedeutet sind die NTL Gleitpunktzahlen vom Typ `quad_float` und `xdouble` keine Standard-Datentypen, das heißt sie werden nicht direkt vom Prozessor unterstützt. Die NTL Typen sind in Software implementiert. Sie werden dabei durch Standard-Datentypen „zusammengebaut“. Beispielsweise setzt sich der NTL Datentyp `xdouble` aus einer `double` Variable (für die Mantisse und das Vorzeichen) und aus einer `long int` Variable für den Exponenten zusammen. Die Arithmetik mit `quad_float` bzw. `xdouble` Datentyp-Objekten ist dementsprechend langsam, weil mehrere Variablen (pro Datentyp-Objekt) durch ein Software-Programm bearbeitet werden müssen.

Bei der konkreten Implementierung der Algorithmen in diesem Kapitel ist zu beachten, dass die Berechnungen über den Maschinenzahlen (eines bestimmten Gleitpunktzahl-Datentyps) durchgeführt werden, und nicht über den reellen Zahlen $\mathbb{R}$. Manchmal unterscheiden wir daher zwischen $\mathbb{R}$ und den Maschinenzahlen (meistens unterscheiden wir aber nicht explizit); zu diesem Zweck ist die folgende Notation hilfreich.

[5] Der NTL Datentyp `RR` ist nicht aufgeführt, da wir ihn nicht einsetzen.

[6] Die Parameter $e_{\min}$ und $e_{\max}$ hängen von dem verwendeten Compiler bzw. den Bibliotheken des Compilers ab. Die angegebenen Werte beziehen sich auf GCC 2.95.3. Die Parameter für `xdouble` mit einem „?“ entnimmt man der NTL Dokumentation zu `xdouble`.

[7] `xdouble` hat einen *sehr großen* Exponentenbereich. Die genauen Schranken entnimmt man der NTL Dokumentation zu `xdouble`.

6.1.1 Notation (Datentyp abhängige Maschinenzahlen) *Mit* $\mathbb{R}_{\texttt{GZ-DT}} \subset \mathbb{R}$ *bezeichnen wir die Menge der reellen Zahlen, die vom Rechner unter Verwendung eines Gleitpunktzahl-Datentyps darstellbar sind. Haben wir einen bestimmten Gleitpunktzahl-Datentyp im Sinn, ersetzen wir* `GZ-DT` *durch diesen. So ist beispielsweise* $\mathbb{R}_{\texttt{double}}$ *die Menge der Maschinenzahlen, die von dem Typ* `double` *abgedeckt*[8] *werden. Entsprechend schreiben wir* $\mathbb{R}_{\texttt{xdouble}}$ *und* $\mathbb{R}_{\texttt{quad\_float}}$.

6.1.2.2 Datentyp für große ganze Zahlen

Neben den Datentypen für Gleitpunktzahlen verwenden wir den NTL Typ `ZZ` für „beliebig“ große ganze Zahlen. Diesen Datentyp benutzen wir, um die (ganzzahligen) Basisvektoren exakt zu speichern (siehe Abschnitt 4.3 auf Seite 53).

Ein Objekt des Typs `ZZ` besteht aus mehreren[9] Objekten des Basistyps `long int`, somit belegt es im Vergleich zu diesem entsprechend mehr Speicherplatz und auch das Rechnen damit ist um einiges langsamer, weil aufwendiger: Pro Rechenoperation, die sich auf ein `ZZ` Objekt bezieht, müssen mehrere `long int` Objekte bearbeitet werden.

Zwar ist `ZZ` durch den vorhandenen Speicher des Rechners beschränkt, jedoch sind in der Praxis alle auftretenden ganzen Zahlen „klein“ genug, um durch ein `ZZ` Objekt repräsentierbar zu sein. Unter „kleinen“ ganzen Zahlen verstehen wir solche, deren Darstellung im Binärsystem immerhin einige hundert oder tausend Bits hat. Eine solche Zahl kann problemlos in einem `ZZ` Objekt gespeichert werden. Bei sehr vielen und sehr großen Zahlen kann es jedoch zu Speicherplatzproblemen kommen. Aufgrund der letzten beiden Sätze unterscheiden wir meistens nicht zwischen `ZZ` und $\mathbb{Z}$, wohl aber müssen wir bei dem Umgang mit `mat_ZZ` und $\mathbb{Z}^{n\times d}$ auf Bemerkung 6.1.2 auf der nächsten Seite achten.

6.1.2.3 Vektoren und Matrizen

Wir machen von den Datenstrukturen der NTL Gebrauch, etwa Matrizen und Vektoren, die sich über allen zur Verfügung stehenden Zahlen-Datentypen definieren lassen.

Vektor-Datentypen werden in NTL durch den Prefix „`vec_`“ gekennzeichnet und die allgemeine Namenskonvention ist `vec_`*`Datentyp`*. Beispielsweise `vec_long` oder `vec_quad_float`.

Datentypen für Matrizen erhalten den Prefix „`mat_`“ und die allgemeine Namenskonvention ist `mat_`*`Datentyp`*. Die von uns benötigten Typen `mat_double`, `mat_xdouble` und `mat_quad_float` stellt NTL *nicht* zur Verfügung. Allerdings gibt es in NTL *Makros*, mit denen wir die benötigten Matrix-Datentypen selber programmiert haben. Die entsprechenden Quellcode Dateien sind in Anhang B auf Seite 113 aufgelistet. Der Typ `mat_ZZ` für ganzzahlige (Basis-) Matrizen wird von NTL standardmäßig angeboten. Die **Abbildung 6.1** auf der nächsten Seite zeigt den typischen Verlauf der

[8] Vergleiche Definition 4.1.8 auf Seite 41 und **Tabelle 6.2** auf der vorherigen Seite.

[9] Laut dem NTL Quellcode ist die Anzahl der benutzten `long int` Objekte standardmäßig ein vielfaches von 4. Das heißt es werden $\varphi \cdot 4, \varphi \in \mathbb{N} \setminus \{0\}$, `long int` Objekte benötigt, um ein `ZZ` Objekt zu repräsentieren.

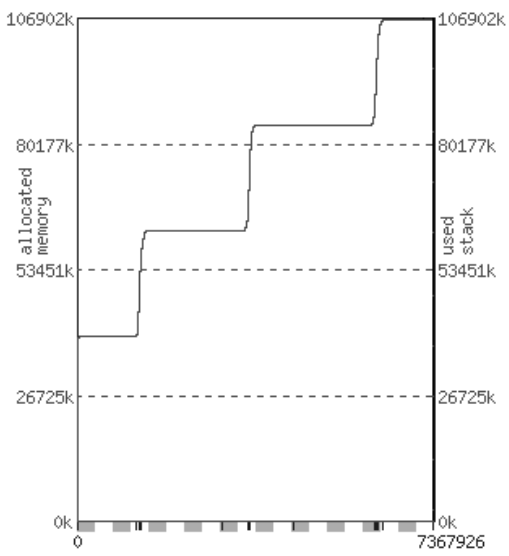


Abbildung 6.1: Verlauf der Speichernutzung bei der Erzeugung einer 1200×1200 Matrix mit $b_{\max} = 410$ Bits

Speichernutzung bei der Erzeugung von ganzzahligen Matrizen dieses Typs. Die Matrix wurde mit `latred` erzeugt, wobei **Algorithmus 6.6** auf Seite 96 verwendet wurde. Interessant ist der stufenweise Anstieg des Speicherbedarfs, wobei die „Sprünge“ recht groß sind, aber selten erfolgen. Die Ursache dafür klärt Bemerkung 6.1.2. Dass der in **Abbildung 6.1** angegebene Speicherbedarf (ca. 107 MByte) größer als die untere Schranke in Bemerkung 6.1.2 ist, basiert auf der Tatsache, dass die NTL Datentypen einen gewissen Verwaltungs-Overhead benötigen.

6.1.2 Bemerkung (Speicherplatzbedarf für `mat_ZZ`) *Sei w die Wortbreite des Rechners (typischerweise ist $w \in \{32, 64\}$) und $b_{\max}$ die Bitlänge des* größten *Eintrages in einer $n \times d$ Matrix `M` vom NTL Typ `mat_ZZ`. Da laut [Shoup, 2001] für* ein *`ZZ` Objekt standardmäßig immer Vielfache von* 4 *Worten reserviert werden (siehe auch Abschnitt 6.1.2.2 auf der vorherigen Seite) und $b_{\max}$ den Speicherplatzbedarf* aller *(!) Einträge bestimmt, gilt für den Speicherplatzbedarf s der Matrix `M` die Abschätzung*

$$s \geq n \cdot d \cdot (\lceil b_{\max}/(4 \cdot w) \rceil \cdot 4 \cdot w) \text{ Bit } = n \cdot d \cdot (\lceil b_{\max}/(4 \cdot w) \rceil \cdot 4 \cdot w)/(8 \cdot 2^{20}) \text{ MByte.}$$

Im Zusammenhang mit Vektoren und Matrizen beachte man auch Bemerkung 6.2.1 auf der nächsten Seite.

6.2 Konzeption von latred

Die wichtigsten Verfahren zur Gitterbasenreduktion sind mit latred einfach zu benutzen. Dazu zählen neben der skalierten Segment LLL-Reduktion und der primal/dualen Segmentreduktion auch die LLL- und BKZ-Reduktion in mehreren Varianten. Die beiden letztgenannten Verfahren sind bereits in der NTL vorhanden und wurden lediglich in latred integriert. Daneben haben wir noch die skalierte LLL-Reduktion nach [Koy und Schnorr, 2001b] programmiert und in latred eingebaut. Zu jedem Verfahren ist es möglich, bei dem Programmaufruf relevante Parameter anzugeben.

Bei dem Aufruf jedes Reduktionsverfahrens kann angegeben werden, mit welchem Datentyp die Gleitpunktzahlen-Arithmetik erfolgen soll. Alle in **Tabelle 6.2** auf Seite 76 angegebenen Typen werden unterstützt. In der NTL sind deswegen die Reduktionsverfahren für jeden Datentyp eigens programmiert. Wir verwenden für unsere Verfahren Templates. Dadurch ist unser Programmieraufwand deutlich geringer (bei leichten Einbußen in der Ausführungsgeschwindigkeit).

Auf Wunsch erzeugt latred Protokolldateien (nur für die von uns programmierten Verfahren – nicht für die aus der NTL) anhand derer man den Verlauf und Fortschritt der Reduktion nachvollziehen kann. Es ist auch möglich, die Aufrufhäufigkeit und Ausführungsdauer einiger Subroutinen aus der Klasse CLattice zu protokollieren und ausgeben zu lassen. Daraus gewinnt man interessante Einblicke in das Reduktiongeschehen.

Um Eingabe- und Referenzbasen für die Experimente einfach und effizient erzeugen zu können, sind entsprechende Algorithmen in latred implementiert und durch das Programm leicht zu bedienen.

Zu einer im NTL Format abgespeicherten Basismatrix eines Gitters können mittels latred interessante Informationen ermittelt werden (Längen bestimmter Vektoren, Determinante, und ähnliches).

Insgesamt betrachtet ist latred ein vielseitiges Programm zur Gitterbasenreduktion, das aufgrund seiner Protokolldateien auch die Auswertung des zeitlichen Verlaufes von Reduktionen ermöglicht. Eine Beschreibung der Kommandozeilenargumente und Optionen für die Benutzung von latred erfolgt in Anhang A auf Seite 107.

Der Quellcode von latred wurde so geschrieben, dass er – die benötigten Bibliotheken vorausgesetzt – auf den meisten Win32 und Linux/Unix Umgebungen übersetzbar sein sollte. Zumindest mit den uns zur Verfügung stehenden Rechnern und Compilern (siehe **Tabelle 6.1** auf Seite 74) kam es zu keinen Problemen bei der Programmerstellung. Eine Auflistung der Quellcode-Dateien ist in Anhang B auf Seite 113 zu finden.

6.2.1 Bemerkung *In unserer Implementierung der Algorithmen zur Gitterbasenreduktion (in latred) repräsentieren die* Zeilenvektoren[10] *einer Matrix (und nicht die Spaltenvektoren wie in Definition 2.2.1 auf Seite 14) die Basisvektoren eines Gitters. Demnach ist zu beachten, dass bei der Programmierung die meisten Matrizen bzw. Vektoren transponiert betrachtet werden müssen.*

[10] Dies geschieht aus effizienz Gründen: Vertauschen von Zeilenvektoren einer Matrix erreicht man in C/C++ durch einfachen Zeigertausch. Siehe auch die NTL Dokumentation [Shoup, 2001].

Wir erinnern an dieser Stelle nochmal an Notation 1.4.2 auf Seite 10. Die Basismatrix des Gitters $L = \mathcal{L}(\vec{b}_1\ \vec{b}_2\ \ldots\ \vec{b}_n) \subset \mathbb{Z}^d$ ist also nach Bemerkung 6.2.1 auf der vorherigen Seite gegeben durch

$$B = [b_{i,j}]_{1\leq i\leq n;1\leq j\leq d} = \begin{bmatrix} \vec{b}_1 \\ \vec{b}_2 \\ \vdots \\ \vec{b}_n \end{bmatrix} = (\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n)^\mathsf{T} \in \mathbb{Z}^{n\times d}.$$

Zu beachten ist das folgende Lemma:

6.2.2 Lemma (Rechenregel für transponiertes Matrizenprodukt) *Für Matrizen $A \in \mathbb{R}^{m\times n}$ und $B \in \mathbb{R}^{n\times p}$ gilt*

$$(AB)^\mathsf{T} = B^\mathsf{T} A^\mathsf{T}.$$

6.3 Das Klassentemplate `CLattice` von `latred`

Damit die von uns programmierten Verfahren zur Gitterbasenreduktion in Segmenten auch mit unterschiedlichen Gleitpunktzahlen-Datentypen zu verwenden sind, haben wir die in `C++` gebotene Möglichkeit genutzt, Templates zu verwenden. Deren Funktionsweise soll hier nicht erläutert werden. Informationen dazu finden sich in Lehrbüchern zum Thema *Programmieren mit `C++`*.

In dem Klassentemplate `CLattice` (siehe Dateien `lattice.h` und `lattice.cpp`) haben wir die relevanten Methoden zur Orthogonalisierung und Gitterbasenreduktion in Segmenten programmiert. Diese Klasse enthält auch die dafür benötigten Datenstrukturen. So etwa die Basismatrix B, eine Matrix B^s für die skalierten Basisvektoren und die Matrix R. Doch dazu gleich mehr. Zunächst geben wir die Deklaration der Klasse an, wobei es uns hier nur auf die Template-Parameter ankommt:

```
template<class mat_T1, class T2, class vec_T2, class mat_T2> class CLattice
{
 ...
};
```

Bei der Spezialisierung der Klasse sind die vier Template-Parameter (`mat_T1`, `T2`, `vec_T2`, `mat_T2`) durch die gewünschten Datentypen zu ersetzen. Wir benutzen die Konvention, dass `T1` ein Platzhalter für einen Datentyp für ganze Zahlen ist (also etwa `long int` oder `ZZ`), wogegen `T2` für Gleitpunktzahlen-Datentypen steht (einem der Typen aus **Tabelle 6.2** auf Seite 76).

Die Basismatrix B und ihr skaliertes Pendant B^s sind in dem Klassentemplate `CLattice` (siehe Dateien `lattice.h` und `lattice.cpp`) als Typ des Templateparameters `mat_T1` deklariert. Im allgemeinen spezialisieren wir das Klassentemplate `CLattice` mit `mat_ZZ` für `mat_T1`, da wir in unseren Experimenten ausschließlich ganzzahlige Gitterbasen betrachten. Das bedeutet also, dass B, B^s als NTL Matrizen mit Einträgen

des ZZ-Datentyps repräsentiert werden. Die R Matrix ist als Typ des Templateparameters `mat_T2` deklariert. Bei einer Spezialisierung substituieren wir dafür einen NTL Datentyp der eine Matrix mit Gleitpunktzahlen Einträgen repräsentiert, also `mat_double`, `mat_quad_float` oder `mat_xdouble`. Die Anzahl der Zeilen und Spalten von B und R wird identisch initialisiert. Das heißt für $B \in \mathbb{Z}^{n\times d}(n \leq d)$ hat die Matrix R ebenfalls n Zeilen und d Spalten. Mit Notation 6.1.1 auf Seite 76 können wir auch $R \in \mathbb{R}^{n\times d}_{\texttt{GZ-DT}}(n \leq d)$ schreiben, wobei $\mathbb{R}_{\texttt{GZ-DT}}$ einer der Mengen $\mathbb{R}_{\texttt{double}}$, $\mathbb{R}_{\texttt{quad\_float}}$ oder $\mathbb{R}_{\texttt{xdouble}}$ entspricht.

Um die Basismatrix in dem Klassentemplate zu initialisieren gibt es zwei Möglichkeiten. Die erste besteht darin, den entsprechenden Konstruktor der Klasse oder die öffentliche Methode `void SetBasismatrix(const mat_T1 &Basismatrix)` zu benutzen. Der Nachteil ist, dass die Basismatrix bereits gegeben sein muss und in der Klasse `CLattice` nochmal angelegt wird (inklusive zusätzlichen Speicherverbrauch). Daher ist die zweite Möglichkeit vorzuziehen: Es wird ein Objekt `L` der Klasse `CLattice` angelegt, dann kann die Basismatrix `L.B` direkt initialisiert werden. Es wird also keine zusätzliche Matrix verwendet. Beispiele wie dies genau geschieht sind der Datei `main.cpp` zu entnehmen.

Wie die anderen Datenstrukturen initialisiert werden und welche öffentlichen und privaten Methoden es – neben den von uns später beschriebenen – gibt, ist aus den Dateien `lattice.h` und `lattice.cpp` ersichtlich. Wir beschreiben hier nur die privaten Routinen der Klasse mit denen die Orthogonalisierungen und Reduktion durchführt werden. Die öffentlichen Klassenmethoden für die Reduktionsverfahren sind im Quellcode dokumentiert. Die Algorithmen in diesem Abschnitt wurden sinngemäß in der hier beschriebenen Art und Weise als Methoden der Klasse `CLattice` implementiert. Durch die hier erfolgende Beschreibung sollte ihre Implementierung für einen geübten `C++` Programmierer leicht nachvollziehbar sein, sofern er sich mit den mathematischen Grundlagen dieser Arbeit vertraut macht. Die Bezeichnungen der Methoden eines Klassentemplates erfolgen gemäß dem Muster

```
template<class mat_T1, class T2, class vec_T2, class mat_T2>
void CLattice<mat_T1,T2,vec_T2,mat_T2>::MethodenName(...);
```

Wir verzichten auf die Angabe des Teils, der die Methode als Funktionstemplate in dem Klassentemplate kennzeichnet und schreiben lediglich `void MethodenName(...)`.

6.3.1 QR-Faktorisierung

In der Klasse `CLattice` ist neben der Basismatrix $B = (\vec{\boldsymbol{b}}_1, \vec{\boldsymbol{b}}_2, \ldots, \vec{\boldsymbol{b}}_n)^{\mathsf{T}} \in \mathbb{Z}^{n\times d}$ des Gitters auch die skalierte Variante $B^s = (\vec{\boldsymbol{b}}^s_1, \vec{\boldsymbol{b}}^s_2, \ldots, \vec{\boldsymbol{b}}^s_n)^{\mathsf{T}} \in \mathbb{Z}^{n\times d}$ der Basismatrix zu faktorisieren, um die Gitterbasis gemäß der skalierten (Segment) LLL-Reduktion transformieren zu können. Wie in Bemerkung 6.2.1 auf Seite 79 erwähnt, sind die Zeilenvektoren der Matrizen zu orthogonalisieren: Die Faktorisierungen $B^{\mathsf{T}} = QR$ und $(B^s)^{\mathsf{T}} = Q'R'$ mit $R, R' \in \mathbb{R}^{n\times n}$ müssen berechnet werden und zwar unter der sinngemäßen Berücksichtigung von Bemerkung 6.3.1 auf der nächsten Seite. Dabei kommt

man mit *einer* Klassen-global zur Verfügung stehenden Matrix R vom Typ mat_T2 aus[11]. In diese Matrix werden die zu Orthogonalisierenden Basisvektoren sukzessive Zeile für Zeile kopiert und orthogonalisiert, dabei werden die eigentlich ganzzahligen Vektoren zu Vektoren über $\mathbb{R}_{\texttt{GZ-DT}}$. Diese Vorgehensweise ermöglicht eine flexible und sehr effiziente Orthogonalisierung unter Berücksichtigung von Bemerkung 6.3.1. Zusätzlich zu R = R benutzen einige Methoden lokale $2k \times 2k$ Matrizen loc_R vom Typ mat_T2, um die Matrix $R_{\ell,\ell+1}$ nach Gleichung (5.4) auf Seite 57 bzw. Matrix R^{loc} nach Gleichung (5.8) auf Seite 67 speichern zu können.

Wir benutzen die bereits vorgestellten Householder-Reflexionen, um die QR-Faktorisierung der ganzzahligen Matrizen B und/oder B^s durchzuführen. Die explizite Berechnung der orthogonalen Matrix Q ist nicht nötig, da wir lediglich R brauchen. Die Givens-Rotationen sind in bestimmten Situationen der Householder-Reflexion vorzuziehen, weil sie dann enorme Effizienzvorteile bieten: Um etwa ein Schema auf Dreiecksform zu bringen das schon fast Dreiecksgestalt hat. Matrizen mit solchen Strukturen begegnen uns bei der Gitterreduktion sehr oft. Die Ursache dafür sind die Vertauschungen von benachbarten Vektoren im LLL-Algorithmus. Einzelne Segmente R_ℓ bzw. $R_{\ell,\ell+1}$ werden mittels Reduktion in lokalen Koordinaten nach dem LLL-Verfahren reduziert. Das bedeutet, dass ausgehend von einer Matrix in Dreiecksform, aufgrund der Austauschschritte im LLL-Algorithmus, die Dreiecksgestalt gestört wird. Mittels *einer* Givens-Rotation lässt sich diese Störung effizient korrigieren (siehe Beispiel 6.3.4 auf Seite 86).

Bei den später angegebenen Algorithmen zur Orthogonalisierung beziehen sich die Eingabeparameter entweder auf B oder B^s, eine explizite Unterscheidung erfolgt nicht, da sich der zu orthogonalisierende Vektor erst aus dem Kontext ergibt, in dem der Algorithmus verwendet wird. Dass bei der Beschreibung der Verfahren als Eingaben die Basisvektoren der Matrix B gefordert werden, ist nicht von Relevanz – völlig analog können auch Vektoren von B^s mit den Algorithmen orthogonalisiert werden. Auch im folgenden beziehen sich unsere Ausführungen meist auf B, obwohl stets auch B^s gemeint ist.

Bei der Reduktion von Gitterbasen in Segmenten berechnen wir Teilmatrizen von R. Während der (skalierten) k-Segment LLL-Reduktion (siehe Abschnitt 5.2 auf Seite 56) werden für $\ell = 1, 2, \ldots, m-1$ sukzessive die $2k \times 2k$ Submatrizen $R_{\ell,\ell+1}$ benötigt, die den zwei Segmenten $B_\ell, B_{\ell+1}$ entsprechen. Für die primal/duale Reduktion von Gitterbasen (siehe Abschnitt 5.3 auf Seite 62) benötigen wir ebenfalls sukzessive Submatrizen von R, und zwar die Matrizen R_ℓ für $\ell = 1, 2, \ldots, m$, wobei R_ℓ die Darstellung in lokalen Koordinaten von B_ℓ ist. Daher berechnen wir in der Praxis *nicht* die komplette QR-Faktorisierung von B auf einmal, sondern wir orthogonalisieren B immer nur bis einschließlich des gerade benötigten Segments.

6.3.1 Bemerkung (Segmente und sukzessive Orthogonalisierung) *In der Praxis berechnen wir bei der Gitterbasenreduktion in Segmenten* nicht *die komplette QR-Faktorisierung von B bzw. B^s auf einmal, sondern wir orthogonalisieren nur bis einschließlich*

[11] R ist eine Matrix über $\mathbb{R}^{n\times d}_{\texttt{GZ-DT}}$, wobei GZ-DT vom Template-Gleitpunktzahlen-Typ T2 ist.

des gerade benötigten Segments B_ℓ bzw. $B_{\ell+1}$ (B^s_ℓ bzw. $B^s_{\ell+1}$). Dazu erfolgt eine Teil-Faktorisierungen von B (B^s) gemäß Satz 4.2.1 auf Seite 44. Dies geschieht mit Hilfe der Householder-Reflexion (siehe Konstruktion 4.2.8 auf Seite 47 und Bemerkung 4.2.9 auf Seite 49).

6.3.1.1 Householder-Reflexion

Man erinnere sich an Gleichung (2.4) auf Seite 19 und beachte Abschnitt 4.2.1 auf Seite 46. An der orthogonalen Matrix Q sind wir nicht explizit interessiert. Diese wird daher nicht berechnet bzw. gespeichert.

6.3.2 Bemerkung (Zeilenvektoren als Spaltenvektoren interpretieren) *Bei den hier angegebenen Algorithmen zur Householder-Reflexion kommt Lemma 6.2.2 auf Seite 80 nicht zur Anwendung. Wir* interpretieren *die Zeilenvektoren (vergleiche Bemerkung 6.2.1 auf Seite 79) als Spaltenvektoren, indem die Vektoreinträge einzeln betrachtet werden. Wir berechnen zu der Teil-Basismatrix $B^{(i)} = (\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_i)^\mathsf{T} \in \mathbb{Z}^{i\times d}$ die Faktorisierungen*

$$(B^{(i)})^\mathsf{T} = [\vec{b}_1^\mathsf{T} \;\; \vec{b}_2^\mathsf{T} \;\; \ldots \;\; \vec{b}_i^\mathsf{T}] = Q^{(i)}[\boldsymbol{r}_1 \;\; \boldsymbol{r}_1 \;\; \ldots \;\; \boldsymbol{r}_i] \quad \textit{für} \;\; i = 1, 2, \ldots, n.$$

Die Matrix R ist zu Beginn eine Approximation der Matrix B, das heißt die Einträge in R sind Gleitpunktzahlen Varianten der ganzzahligen Einträge aus der Matrix B. Daher ist R in `CLattice` keine $n \times n$ Matrix, sondern sie ist den Dimensionen von B entsprechend eine $n \times d$ ($n \leq d$) Matrix. Nach Bemerkung 6.3.1 auf der vorherigen Seite werden die „Basisvektoren in R“ sukzessive orthogonalisiert. Dabei werden zusätzlich zu den berechneten Vektoren $\boldsymbol{r}_i$ (die als $\boldsymbol{r}_i^\mathsf{T}$ gespeichert werden) auch die Householder-Vektoren und die Zwischenergebnisse des Orthogonalisierungsprozesses in R (zwischen-) gespeichert. Die $\boldsymbol{r}_i$ werden sukzessive bestimmt, und zwar gemäß Bemerkung 4.2.9 auf Seite 49. Für die Orthogonalisierung des i-ten Basisvektors werden die vorhergehenden $i-1$ Householder-Vektoren benötigt. Da nach Gleichung (4.8) auf Seite 49 die Householder-Vektoren stets gleich bleiben, ist es sinnvoll, jeden von ihnen nur *einmal zu berechnen und dann zur weiteren Verfügbarkeit zu speichern.* Genau dies ist in `CLattice` realisiert: Die Householder-Vektoren werden an den nicht benötigten Positionen der R Matrix gespeichert. Das Speichern der Householder-Vektoren $\boldsymbol{v}^{(i)}$ in R ist möglich, weil R eine (untere) Dreiecksmatrix ist. Die Positionen in R, die eigentlich Nullen als Einträge enthalten, nutzen wir aus, um dort die Koordinaten von $(\vec{\boldsymbol{v}}^{(i)})^\mathsf{T}$ zu speichern, die nicht Null sind (genaueres entnimmt man **Algorithmus 6.2** auf Seite 85).

Desweiteren macht es Sinn, den Wert $h_j = -2/\|\vec{\boldsymbol{v}}^{(j)}\|^2$ zwischen zu speichern, weil er ebenfalls des öfteren gebraucht wird. Dazu verwenden wir einen Hilfsvektor $\vec{\boldsymbol{h}} = (h_1, h_2, \ldots, h_n) \in \mathbb{R}^n$. Näheres entnimmt man **Algorithmus 6.2** auf Seite 85 bzw. **Algorithmus 6.1** auf der nächsten Seite. Man bemerke, dass die beiden Algorithmen nur zusammen die gewünschte Orthogonalisierung der Basisvektoren $\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n$ ergeben.

Algorithmus 6.1 Householder-Reflexion

Eingabe: $i \in \{2,3,\ldots,n\}$; $j \in \{1,2,\ldots,i-1\}$; $R^{(j)} = \left[r_{k,\ell}^{(j)}\right]_{1\le k\le n, 1\le \ell\le d} \in \mathbb{R}^{n\times d}$ und es sei sichergestellt, dass $R^{(1)} \leftarrow B = (\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n)^\mathsf{T}$; mit **Algorithmus 6.2** auf der nächsten Seite seien der Householder-Vektor $\vec{v}^{(j)} \in \mathbb{R}^d$ und der Hilfsvektor $\vec{h} = (h_1, h_2, \ldots, h_n) \in \mathbb{R}^n$ berechnet

1: /∗ Analog zur Berechnung der j-ten Householder-Reflexion $\boldsymbol{b}_i^{(j+1)} = H^{(j)}\big(\boldsymbol{b}_j^{(j)}\big) \cdot \boldsymbol{b}_i^{(j)}$ in Abschnitt 4.2.1 (Konstruktion 4.2.8 auf Seite 47 und Gleichung (4.7) auf Seite 49) berechnet dieser Algorithmus die j-te Householder-Reflexion $\vec{\boldsymbol{b}}_i^{(j+1)} = \left(H^{(j)}\big((\vec{\boldsymbol{b}}_j^{(j)})^\mathsf{T}\big) \cdot (\vec{\boldsymbol{b}}_i^{(j)})^\mathsf{T}\right)^\mathsf{T}$. Weil $\vec{\boldsymbol{r}}_i^{(1)} := \vec{\boldsymbol{b}}_i$ gesetzt wird, geschieht dies durch

$$\vec{\boldsymbol{r}}_i^{(j+1)} \leftarrow \left(H^{(j)}\left(\left(\vec{\boldsymbol{r}}_j^{(j)}\right)^\mathsf{T}\right) \cdot \left(\vec{\boldsymbol{r}}_i^{(j)}\right)^\mathsf{T}\right)^\mathsf{T}.$$

Die Iteration dieses Algorithmus' für ein festes i und $j = 1,2,\ldots,i-1$ berechnet nach Gleichung (4.6) bzw. Gleichung (4.7) auf Seite 49 den Vektor $\vec{\boldsymbol{r}}_i^{(i)} = \vec{\boldsymbol{b}}_i^{(i)}$ (siehe auch Gleichung (6.1) in **Algorithmus 6.2** auf der nächsten Seite). Ein anschließender Aufruf von **Algorithmus 6.2** auf der nächsten Seite mit Parameter i vervollständigt die Orthogonalisierung von $\vec{\boldsymbol{b}}_i$, denn der dortige Schritt 7 bewirkt nach Lemma 4.2.6 auf Seite 46 die noch fehlende Multiplikation und Zuweisung

$$\vec{\boldsymbol{r}}_i = \vec{\boldsymbol{r}}_i^{(i+1)} \leftarrow \left(H^{(i)}\left(\left(\vec{\boldsymbol{r}}_i^{(i)}\right)^\mathsf{T}\right) \cdot \left(\vec{\boldsymbol{r}}_i^{(i)}\right)^\mathsf{T}\right)^\mathsf{T}.$$

∗/

2: $R^{(j+1)} \leftarrow R^{(j)}$ /∗ Bedeutet formal, dass sich $R^{(j+1)}$ und $R^{(j)}$ zunächst nicht unterscheiden. ∗/

3: /∗ Der j-te Householder-Vektor $\vec{\boldsymbol{v}}^{(j)} = \left(\boldsymbol{v}^{(j)}\left(\left(\vec{\boldsymbol{r}}_j^{(j)}\right)^\mathsf{T}\right)\right)^\mathsf{T} = \left(\boldsymbol{v}^{(j)}\left(\left(\vec{\boldsymbol{b}}_j^{(j)}\right)^\mathsf{T}\right)\right)^\mathsf{T}$ ist nach **Algorithmus 6.2** auf der nächsten Seite gegeben durch:

$$\vec{\boldsymbol{v}}^{(j)} = (\overbrace{\underbrace{0,0,\ldots,0}_{j-1 \text{ Nullen}},1}^{\text{ist nicht gespeichert}}, r_{j,j+1}^{(j)}, \ldots, r_{j,d}^{(j)}),$$

das heißt also die relevanten Koordinaten von $\vec{\boldsymbol{v}}^{(j)}$ sind in der j-ten Zeile von $R^{(j)}$ gespeichert, genauer gesagt in den Positionen $r_{j,j+1}^{(j)}, r_{j,j+2}^{(j)}, \ldots, r_{j,d}^{(j)}$. ∗/

4: $w \leftarrow 1 \cdot r_{i,j}^{(j)} + \sum_{k=j+1}^{d} r_{j,k}^{(j)} \cdot r_{i,k}^{(j)}$ /∗ $w = \langle \vec{\boldsymbol{v}}^{(j)}, \vec{\boldsymbol{r}}_i^{(j)} \rangle$ ∗/

5: /∗ Es gilt $h_j = -\dfrac{2}{\|\vec{\boldsymbol{v}}^{(j)}\|^2}$ (siehe Schritt 12 von **Algorithmus 6.2** auf der nächsten Seite). ∗/

6: $w \leftarrow w \cdot h_j$ /∗ $w = -2 \cdot \dfrac{\langle \vec{\boldsymbol{v}}^{(j)}, \vec{\boldsymbol{r}}_i^{(j)} \rangle}{\|\vec{\boldsymbol{v}}^{(j)}\|^2}$ ∗/

7: /∗ Die Schritte 8–11 berechnen $\vec{\boldsymbol{r}}_i^{(j+1)} \leftarrow \left(\left(\vec{\boldsymbol{r}}_i^{(j)}\right)^\mathsf{T} + \left(\vec{\boldsymbol{v}}^{(j)}\right)^\mathsf{T} \cdot w\right)^\mathsf{T}$, wobei berücksichtigt wird, dass die ersten $j-1$ Koordinaten von $\vec{\boldsymbol{v}}^{(j)}$ Null sind. ∗/

8: $r_{i,j}^{(j+1)} \leftarrow r_{i,j}^{(j)} + w \cdot 1$

9: **for** $k = j+1, j+2, \ldots, d$ **do**

10: $\quad r_{i,k}^{(j+1)} \leftarrow r_{i,k}^{(j)} + r_{j,k}^{(j)} \cdot w$

11: **end for**

Ausgabe: Matrix $R^{(j+1)}$ die sich von $R^{(j)}$ nur in der i-ten Zeile unterscheidet:

$$\vec{\boldsymbol{r}}_i^{(j+1)} = \left(H^{(j)}\left(\left(\vec{\boldsymbol{r}}_j^{(j)}\right)^\mathsf{T}\right) \cdot \left(\vec{\boldsymbol{r}}_i^{(j)}\right)^\mathsf{T}\right)^\mathsf{T} = \left(\left(\vec{\boldsymbol{r}}_i^{(j)}\right)^\mathsf{T} - 2 \cdot \left(\vec{\boldsymbol{v}}^{(j)}\right)^\mathsf{T} \frac{\langle \vec{\boldsymbol{v}}^{(j)}, \vec{\boldsymbol{r}}_i^{(j)} \rangle}{\|\vec{\boldsymbol{v}}^{(j)}\|^2}\right)^\mathsf{T}$$

Algorithmus 6.2 Householder-Vektor

Eingabe: $i \in \{1,2,\ldots,n\}$; $R^{(i)} = \left[r_{k,\ell}^{(i)}\right]_{1\le k\le n, 1\le \ell\le d} \in \mathbb{R}^{n\times d}$ und es sei sichergestellt, dass $R^{(1)} \leftarrow B = (\vec{\boldsymbol{b}}_1, \vec{\boldsymbol{b}}_2, \ldots, \vec{\boldsymbol{b}}_n)^{\mathsf{T}}$; gegeben sei der Hilfsvektor $\vec{\boldsymbol{h}} = (h_1, h_2, \ldots, h_n) \in \mathbb{R}^n$. Falls $i \ge 2$, so sei der i-te Zeilenvektor

$$\vec{\boldsymbol{r}}_i^{(i)} = \left(H^{(i-1)}\left(\left(\vec{\boldsymbol{r}}_{i-1}^{(i-1)}\right)^{\mathsf{T}}\right) \cdot H^{(i-2)}\left(\left(\vec{\boldsymbol{r}}_{i-2}^{(i-2)}\right)^{\mathsf{T}}\right) \cdots H^{(1)}\left(\left(\vec{\boldsymbol{r}}_1^{(1)}\right)^{\mathsf{T}}\right) \cdot \left(\vec{\boldsymbol{r}}_i^{(1)}\right)^{\mathsf{T}} \right)^{\mathsf{T}} \tag{6.1}$$

von $R^{(i)}$ bereits mit **Algorithmus 6.1** auf der vorherigen Seite berechnet worden.

1: /∗ Dieser Algorithmus berechnet den i-ten Householder-Vektor $\vec{\boldsymbol{v}}^{(i)} = \left(\boldsymbol{v}^{(i)}\left(\left(\vec{\boldsymbol{r}}_i^{(i)}\right)^{\mathsf{T}}\right)\right)^{\mathsf{T}}$ nach Definition 4.2.5 auf Seite 46, und speichert ihn in der i-ten Zeile von $R^{(i+1)}$. Daneben wird die Orthogonalisierung von $\vec{\boldsymbol{b}}_i$ vervollständigt. ∗/

2: $R^{(i+1)} \leftarrow R^{(i)}$ /∗ Bedeutet formal, dass sich $R^{(i+1)}$ und $R^{(i)}$ zunächst nicht unterscheiden ∗/

3: $t \leftarrow \mathsf{sign}\left(r_{i,i}^{(i)}\right)\sqrt{\left(r_{i,i}^{(i)}\right)^2 + \left(r_{i,i+1}^{(i)}\right)^2 + \cdots + \left(r_{i,d}^{(i)}\right)^2}$ /∗ Nach Definition 4.2.5 auf Seite 46. ∗/

4: /∗ Nach [Oevel, 1996, S. 137, Fußnote 4] tritt der Fall $t = 0$ dann und nur dann auf, wenn B nicht invertierbar ist. ∗/

5: $w \leftarrow t + r_{i,i}^{(i)}$

6: $r_{i,i}^{(i+1)} \leftarrow -t$ /∗ Diese Zuweisung vervollständigt nach Lemma 4.2.6 auf Seite 46 die Orthogonalisierung von $\vec{\boldsymbol{b}}_i = \vec{\boldsymbol{r}}_i^{(1)}$. ∗/

7: /∗ Der Householder-Vektor $\vec{\boldsymbol{v}}^{(i)}$ zum Erzeugungs-Vektor $\left(\vec{\boldsymbol{r}}_i^{(i)}\right)^{\mathsf{T}}$ sieht (nach Definition 4.2.5 auf Seite 46) so aus

$$\vec{\boldsymbol{v}}^{(i)} = (v_1^{(i)}, v_2^{(i)}, \ldots, v_{i-1}^{(i)}, v_i^{(i)}, v_{i+1}^{(i)}, \ldots, v_d^{(i)}) = (\underbrace{0,0,\ldots,0}_{i-1 \text{ Nullen}}, t + r_{i,i}^{(i)}, r_{i,i+1}^{(i)}, \ldots, r_{i,d}^{(i)})\,.$$

Indem wir alle Einträge in $\vec{\boldsymbol{v}}^{(i)}$ durch $w = t + r_{i,i}^{(i)}$ teilen, normieren wir die i-te Koordinate $v_i^{(i)} = t + r_{i,i}^{(i)}$ von $\vec{\boldsymbol{v}}^{(i)}$ zu 1. Weil dann immer $v_i^{(i)} = 1$ gilt, ist es nicht nötig den Wert $v_i^{(i)}$ explizit zu speichern.
Die Schritte 8–10 bewirken, dass die Einträge in $\vec{\boldsymbol{v}}^{(i)}$ durch $w = t + r_{i,i}^{(i)}$ geteilt werden. Die ersten $i-1$ Koordinaten von $\vec{\boldsymbol{v}}^{(i)}$ sind Null und werden daher ignoriert und nicht abgespeichert. Der Wert $v_i^{(i)} = 1$ wird ebenfalls nicht abgespeichert. ∗/

8: **for** $k = i+1, i+2, \ldots, d$ **do**

9: $\quad r_{i,k}^{(i+1)} = r_{i,k}^{(i)}/w$

10: **end for**

11: /∗ Der Householder-Vektor $\vec{\boldsymbol{v}}^{(i)}$ zum Erzeugungs-Vektor $\left(\vec{\boldsymbol{r}}_i^{(i)}\right)^{\mathsf{T}}$ wird somit repräsentiert durch

$$\vec{\boldsymbol{v}} = \vec{\boldsymbol{v}}^{(i)} = (\overbrace{\underbrace{0,0,\ldots,0}_{i-1 \text{ Nullen}}, 1}^{\text{wird nicht abgespeichert}}, r_{i,i+1}^{(i+1)}, \ldots, r_{i,d}^{(i+1)}) \leftarrow \Big(\underbrace{0,0,\ldots,0}_{i-1 \text{ Nullen}}, \frac{t + r_{i,i}^{(i)}}{t + r_{i,i}^{(i)}}, \overbrace{\frac{r_{i,i+1}^{(i)}}{t + r_{i,i}^{(i)}}, \ldots, \frac{r_{i,d}^{(i)}}{t + r_{i,i}^{(i)}}}^{\text{relevante Koordinaten}}\Big)\,,$$

das heißt die relevanten Koordinaten von $\vec{\boldsymbol{v}}^{(i)}$ werden in der i-ten Zeile von $R^{(i+1)}$ gespeichert, genauer gesagt in den Positionen $r_{i,i+1}^{(i+1)}, r_{i,i+2}^{(i+1)}, \ldots, r_{i,d}^{(i+1)}$. ∗/

12: $t \leftarrow 1 + \left(r_{i,i+1}^{(i+1)}\right)^2 + \left(r_{i,i+2}^{(i+1)}\right)^2 + \cdots + \left(r_{i,d}^{(i+1)}\right)^2$ /∗ $t = \langle \vec{\boldsymbol{v}}, \vec{\boldsymbol{v}} \rangle = \|\vec{\boldsymbol{v}}\|^2 = \vec{\boldsymbol{v}}\vec{\boldsymbol{v}}^{\mathsf{T}}$ ∗/

13: $h_i \leftarrow -2/t$ /∗ Vergleiche mit Gleichung (4.5) auf Seite 46. ∗/

Ausgabe: Matrix $R^{(i+1)}$ die sich von $R^{(i)}$ nur in der i-ten Zeile unterscheidet: Genaueres entnimmt man obiger Beschreibung.

Die Methode `void HouseholderReflexion(vec_T2 &r, const long j)` berechnet für den Basisvektor `r`$= \vec{r}_i^{(j)}$ die j-te Householder-Reflexion. Zu Beginn setzt man $\vec{r}_i^{(1)} := \vec{b}_i$. Die genaue Vorgehensweise beschreibt **Algorithmus 6.1** auf Seite 84.

Die Methode `void HouseholderVector(const long nu)` berechnet den `nu`-ten Householder-Vector und speichert ihn in der R Matrix ab. Das genaue Vorgehen beschreibt **Algorithmus 6.2** auf der vorherigen Seite.

Die beiden eben erwähnten Methoden sind der Kern aller Orthogonalisierungen, die in `CLattice` durchgeführt werden.

6.3.1.2 Givens-Rotation

Wir erläutern die Implementierung der Givens-Rotation (siehe auch Abschnitt 4.2.2 auf Seite 49) in unserem `latred`-Programm. Nach Gleichung (2.4) auf Seite 19 und Bemerkung 6.2.1 auf Seite 79 hat die Matrix R untere Dreiecksgestalt. Vertauscht man in R zwei Zeilenvektoren entsteht eine Matrix, die fast untere Dreiecksgestalt hat. Wir benutzen die Givens-Rotation zur Korrektur dieser Matrix, so dass sie anschließend wieder untere Dreiecksgestalt hat. Der Definition 4.2.14 auf Seite 52 entsprechend übertragen wir den Begriff einer einfach gestörten Matrix auf eine Matrix in unterer Dreiecksform:

6.3.3 Definition (Einfach gestört in der i-ten Zeile) *Sei $R \in \mathbb{R}^{n\times d}(n \leq d)$ eine Matrix in* unterer *Dreiecksgestalt. Vertauscht man in R die beiden Zeilenvektoren i und $i+1$ für $i \in \{1, 2, \ldots, n-1\}$, so sagen wir für die resultierende Matrix $R' \in \mathbb{R}^{n\times d}$, dass sie* einfach gestört in der i-ten Zeile *ist.*

Einfach gestörte Matrizen haben also fast Dreiecksgestalt. Daher lassen sich sehr effizient mit einer Givens-Rotation „reparieren“. Dazu ein Beispiel (vergleiche auch mit Beispiel 4.2.15 auf Seite 52):

6.3.4 Beispiel (Korrektur einer einfach gestörten Matrix) Gegeben sei eine Matrix $R \in \mathbb{R}^{n\times d}(n \leq d)$ in unterer Dreiecksgestalt. Durch Vertauschung zweier *benachbarter* Zeilenvektoren wird die untere Dreiecksform zerstört. Allerdings ist die Korrektur (das heißt die Matrix wieder auf untere Dreiecksgestalt zu bringen) mit einer Givens-Rotation leicht möglich. Wir veranschaulichen die Situation durch die folgende 6×7 Matrix:

$$R := \begin{bmatrix}\vec{r}_1\\ \vec{r}_2\\ \vec{r}_3\\ \vec{r}_4\\ \vec{r}_5\\ \vec{r}_6\end{bmatrix} = \begin{bmatrix} r_{1,1} & 0 & 0 & 0 & 0 & 0 & 0\\ r_{2,1} & r_{2,2} & 0 & 0 & 0 & 0 & 0\\ r_{3,1} & r_{3,2} & r_{3,3} & 0 & 0 & 0 & 0\\ r_{4,1} & r_{4,2} & r_{4,3} & r_{4,4} & 0 & 0 & 0\\ r_{5,1} & r_{5,2} & r_{5,3} & r_{5,4} & r_{5,5} & 0 & 0\\ r_{6,1} & r_{6,2} & r_{6,3} & r_{6,4} & r_{6,5} & r_{6,6} & 0\end{bmatrix}.$$

$$\begin{bmatrix}\vec{r}_1\\ \vec{r}_2\\ \vec{r}_3\\ \vec{r}_4\\ \vec{r}_5\\ \vec{r}_6\end{bmatrix} \xrightarrow{\vec{r}_3 \leftrightarrow \vec{r}_4} \begin{bmatrix} r_{1,1} & 0 & 0 & 0 & 0 & 0 & 0\\ r_{2,1} & r_{2,2} & 0 & 0 & 0 & 0 & 0\\ r_{4,1} & r_{4,2} & r_{4,3} & r_{4,4} & 0 & 0 & 0\\ r_{3,1} & r_{3,2} & r_{3,3} & 0 & 0 & 0 & 0\\ r_{5,1} & r_{5,2} & r_{5,3} & r_{5,4} & r_{5,5} & 0 & 0\\ r_{6,1} & r_{6,2} & r_{6,3} & r_{6,4} & r_{6,5} & r_{6,6} & 0\end{bmatrix} = \begin{bmatrix}\vec{r}\,'_1\\ \vec{r}\,'_2\\ \vec{r}\,'_3\\ \vec{r}\,'_4\\ \vec{r}\,'_5\\ \vec{r}\,'_6\end{bmatrix} =: R'.$$

Der Zeilenvektor $\vec{r}_3' = (r_{4,1}, r_{4,2}, r_{4,3}, r_{4,4}, 0, 0)$ stört offenbar die Dreiecksgestalt: Die Matrix R' ist einfach gestört in der 3-ten Zeile. Die 4-te Koordinate von $\vec{r}_3'$ wird durch eine Givens-Rotation $G_{3,4}$ in die 3-te Koordinate rotiert. Durch die Multiplikation $R' \cdot \left(G_{3,4}(\vec{r}_3'^{\,\mathsf{T}})\right)^{\mathsf{T}}$ (man bemerke die Anwendung von Lemma 6.2.2 auf Seite 80) erhält man wieder eine untere Dreiecksmatrix $\bar{R}$:

$$\bar{R} := R' \cdot \left(G_{3,4}(\vec{r}_3'^{\,\mathsf{T}})\right)^{\mathsf{T}} = \begin{bmatrix} r_{1,1} & 0 & 0 & 0 & 0 & 0 & 0 \\ r_{2,1} & r_{2,2} & 0 & 0 & 0 & 0 & 0 \\ r_{4,1} & r_{4,2} & \bar{r}_{4,3} & 0 & 0 & 0 & 0 \\ r_{3,1} & r_{3,2} & \bar{r}_{3,3} & \bar{r}_{3,4} & 0 & 0 & 0 \\ r_{5,1} & r_{5,2} & \bar{r}_{5,3} & \bar{r}_{5,4} & r_{5,5} & 0 & 0 \\ r_{6,1} & r_{6,2} & \bar{r}_{6,3} & \bar{r}_{6,4} & r_{6,5} & r_{6,6} & 0 \end{bmatrix}.$$

Man beachte, dass sich R' und $\bar{R}$ nur in den Spalten 3 und 4 unterscheiden.

Algorithmus 6.3 Givens-Rotation (Korrektur einer einfach gestörten Matrix)

Eingabe: Dimension $n \in \mathbb{N}$, Zeile $k \in \{1, 2, \ldots, n\}$, Matrix $R = [r_{i,j}] \in \mathbb{R}^{n \times d}$ einfach gestört in der k-ten Zeile

1: /* Sei $\vec{r}_k := (r_{k,1}, r_{k,2}, \ldots, r_{k,k}, r_{k,k+1}, 0, 0, \ldots, 0) \in \mathbb{R}^d$ der k-te Zeilenvektor von R. Dieser Algorithmus berechnet $R \leftarrow R \cdot \left(G_{k,k+1}(\vec{r}_k^{\mathsf{T}})\right)^{\mathsf{T}}$. Modifiziert werden nur die Einträge $r_{i,k}$ und $r_{i,k+1}$ für $i = k, k+1, \ldots, n$, das heißt nur die Spalten k und $k+1$ in R werden verändert. */
2: $\ell \leftarrow k + 1$
3: $x_k \leftarrow r_{k,k}$
4: $x_\ell \leftarrow r_{k,\ell}$
5: /* Es folgt die Berechnung von c und s nach Bemerkung 4.2.11 auf Seite 50 */
6: **if** $|x_\ell| > |x_k|$ **then**
7: $\quad \tau \leftarrow x_k / x_\ell$
8: $\quad s \leftarrow 1/\sqrt{1+\tau^2}$
9: $\quad c \leftarrow s \cdot \tau$
10: **else**
11: $\quad \tau \leftarrow x_\ell / x_k$
12: $\quad c \leftarrow 1/\sqrt{1+\tau^2}$
13: $\quad s \leftarrow c \cdot \tau$
14: **end if**
15: /* Der Rest ist eine vereinfachte Matrizenmultiplikation (in R werden nur die Spalten k und
16: $\quad \ell = k+1$ verändert) */
17: $r_{k,k} \leftarrow r_{k,k} \cdot c + r_{k,\ell} \cdot s$
18: $r_{k,\ell} \leftarrow 0$
19: **for** $i = k+1, k+2, \ldots, n$ **do**
20: $\quad u \leftarrow r_{i,k}$
21: $\quad r_{i,k} \leftarrow r_{i,k} \cdot c + r_{i,\ell} \cdot s$
22: $\quad r_{i,\ell} \leftarrow r_{i,\ell} \cdot c - u \cdot s$
23: **end for**

Ausgabe: Matrix $R = [r_{i,j}] \in \mathbb{R}^{n \times d}$ in unterer Dreiecksgestalt

Nach Bemerkung 6.2.1 auf Seite 79 repräsentieren in unserem `latred`-Programm die Zeilenvektoren einer Matrix die Basisvektoren eines Gitters. Es ist darauf zu achten, dass die in Lemma 6.2.2 auf Seite 80 angegebene Rechenregel eingehalten wird. Damit können wir jetzt **Algorithmus 6.3** auf der vorherigen Seite in der Form angeben, wie er von uns im `latred`-Programm implementiert worden ist. Die entsprechende Methode in `CLattice` ist `void GivensRotation(mat_T2 &X, const long row, const long k, const long l)`. Mit der Notation von **Algorithmus 6.3** auf der vorherigen Seite werden folgende Parameter übergeben: `row` $=$ `k` $= k$ und `l` $=$ `k+1`, sowie `X` $= R$ (einfach gestört in Zeile k).

6.3.2 Skalierte k-Segment LLL-Reduktion

In diesem Abschnitt wird erläutert, welche Methoden des Klassentemplates `CLattice` bei der skalierten Segment LLL-Reduktion benutzt werden und wie ihre Funktionsweise ist.

Ein wichtiger Bestandteil der skalierten Reduktion ist die Routine HRS (siehe **Algorithmus C.1** auf Seite 121) bzw. leicht modifizierte Varianten davon. Die in Abschnitt 6.3.2.2 auf Seite 90 beschriebenen `HRS`-Methoden verwenden Householder-Reflexionen zur Orthogonalsierung. Wir erläutern in Abschnitt 6.3.2.1 einige Modifikationen, die alle Varianten von `HRS` betreffen: Die in `HRS` durchgeführte Anzahl der Iterationen lässt sich verringern, falls man die zur Verfügung stehenden Präziosionsbits der benutzten Gleitpunktzahlen berücksichtigt. Davon profitiert die Gesamtlaufzeit der skalierten Segmentreduktion.

6.3.2.1 Längenreduktion und Skalieren mit Gleitpunktzahlen

Nach Definition 4.1.8 auf Seite 41 ist bei Gleitpunktzahlen sowohl die Anzahl der Präzionsbits als auch der Exponentenbereich beschränkt. Wir benutzen in der Praxis die in **Tabelle 6.2** auf Seite 76 angegebenen Datentypen für die Gleitpunktzahlen-Rechnungen während der skalierten Segment LLL-Reduktion.

6.3.5 Bemerkung (Längenreduktion und Präzisionsbits bei Ausführung von HRS)
*Die Längenreduktion in dem 3-ten Schritt von HRS (siehe **Algorithmus C.1** auf Seite 121) erfolgt in allen unseren implementierten Varianten nur, falls*

$$|\mu^s_{\nu,j}| > \max\left\{ P^N_{\text{HRS-mu-test}}, \frac{\pi_j(\vec{b}^s_\nu)}{\|\widehat{\vec{b}^s_j}\| \cdot P^N_{\text{HRS-max-prec-loss}}} \right\}. \tag{6.2}$$

Wir berücksichtigen also die Anzahl N der zur Verfügung stehenden Präzisionsbits (siehe **Tabelle 6.3** auf der nächsten Seite, und vergleiche mit **Tabelle 6.2** auf Seite 76). Es wird sichergestellt, dass genug Präzisionsbits die bei der Längenreduktion auftretenden Auslöschungen überstehen. Der Parameter $P^N_{\text{max-prec-loss}}$ gibt an, wieviele Bits höchstens „verloren“ gehen dürfen. Er ist in Abhängigkeit von N gewählt (siehe **Tabelle 6.3** auf der nächsten Seite). Je mehr Präzisionsbits zur Verfügung stehen,

desto weniger oft müssen die Schritte 2,3 und 4 iteriert werden. Man beachte, dass nötigenfalls in Schritt 6 die Skalierung an die numerischen Erfordernisse angepasst wird.

Datentyp	N	$P^N_{\text{HRS-mu-test}}$	$P^N_{\text{HRS-max-prec-loss}}$	$P^N_{\text{HRS-loop-cntrl}}$
`double`	53	0.501	2^{15}	2^{10}
`xdouble`	53	0.501	2^{15}	2^{10}
`quad_float`	105	0.500001	2^{35}	2^{10}

Tabelle 6.3: Kontroll-Parameter, welche die Längenreduktion in HRS beeinflussen

Die Werte in **Tabelle 6.3** sind Standardvorgaben in der Datei `lattice.h`. Beim Aufruf des `latred`-Programms können über die Kommandozeile jedoch andere Werte angegeben werden. Der Parameter $P^N_{\text{HRS-loop-cntrl}}$ kommt in Schritt 4 von **Algorithmus C.1** auf Seite 121 zur Geltung: Er bestimmt wie groß $|\mu^s_{\nu,i}|$ für $i = 1, 2, \ldots, n$ höchstens sein darf; existiert ein zu großer $|\mu^s_{\nu,i}|$-Wert, so geht der Algorithmus zu Schritt 2 zurück. Anhand der Tabelle wird ersichtlich, dass `quad_float` Typen wegen der großen Anzahl von Präzisionsbits sehr geeignet sind, die Anzahl der Iterationen in **Algorithmus C.1** auf Seite 121 klein zu halten. Dies macht sich auch experimentell bemerkbar, wie etwa auf Seite 122 erklärt. Im Vergleich zu `double` ist die Arithmetik mit `quad_float` Typen zwar aufwendiger (vergleiche Abschnitt 6.1.2.1 auf Seite 76), jedoch führt die Verwendung von `quad_float` in der Praxis meistens zu schnelleren Reduktionen.

Die Einträge a_i eines Vektors $\boldsymbol{a} = (a_1, a_2, \ldots, a_d) \in \mathbb{Z}^d$ sind in `CLattice` vom Typ `ZZ`. Die Skalierung von $\boldsymbol{a}$ erfolgt durch die Multiplikation mit 2^e: $\boldsymbol{a} \leftarrow 2^e\boldsymbol{a}$, das heißt jede Koordinate a_i wird mit 2^e multipliziert. Dies entspricht in der Binärdarstellung von a_i einer *Linksverschiebung*[12] jeder Stelle um e Positionen: Für $a = a_i \in \mathbb{Z}$ ergibt sich aus

$$a \;=\; \sum_{k=0}^{s} \alpha_k 2^k \mathrel{\widehat{=}} (\alpha_s, \alpha_{s-1}, \ldots, \alpha_0) \in \{0,1\}^{s+1}$$

die Darstellung

$$2^e a \;=\; \sum_{h=0}^{e-1} 0 \cdot 2^h + \sum_{k=0}^{s} \alpha_k 2^{k+e} \mathrel{\widehat{=}} (\alpha_s, \alpha_{s-1}, \ldots, \alpha_0, 0, \ldots, 0) \in \{0,1\}^{s+1+e}.$$

Die Skalierungen von `ZZ` Datentypen erfolgen daher mit Hilfe der NTL Funktion

```
ZZ LeftShift(const ZZ& a, long e);
```

Die Einträge der Vektoren „runter Skalieren“ kann man aus analogen Gründen mit

[12] Dabei gehen wir davon aus, dass die höchstwertigste Stelle in der Binärdarstellung links steht.

```
ZZ RightShift(const ZZ& a, long e);
```

Details zu den Parametern entnimmt man der Dokumentation von [Shoup, 2001]. Nach Abschnitt 6.1.2.2 auf Seite 77 kann man ZZ Typen durch die Skalierung fast beliebig groß oder klein machen, ohne numerische Auswirkungen. Beeinflusst werden jedoch Speicherplatzbedarf und Rechengeschwindigkeit.

In **Algorithmus C.1** auf Seite 121 muss der Vektor $\boldsymbol{b}^s \in \mathbb{Z}^d$ für die schnelle Orthogonalisierung mit Householder-Reflexionen in einen Gleitpunktzahlen-Datentyp konvertiert werden. Dadurch ergeben sich Beschränkungen für den Exponentenbereich, das heißt die Einträge des Vektors können nicht mehr beliebig groß werden.

6.3.6 Bemerkung (Skalieren und Exponentenbereich) *Die Skalierung des Gleitpunktzahlen-Vektors $\boldsymbol{b}^s_\nu \leftarrow 2^{\widetilde{e}}\boldsymbol{b}^s_\nu$ in Schritt 6 von **Algorithmus C.1** auf Seite 121 kann dazu führen, dass der zulässige Exponentenbereich (siehe Definition 4.1.8 auf Seite 41 und **Tabelle 6.2** auf Seite 76) überschritten wird: Exponentenüberlauf ist unter Umständen möglich.*

In unseren Experimenten haben wir Exponentenüberläufe bei Gitterbasen deren Einträge in der Größenordnung von ca. 500 Bits liegen, beobachtet (bei Dimension 500).

6.3.2.2 Beschreibung der Methoden

```
void ScaledSegmentLLL(const long k, const long m, const double delta_dp,
                      const long shift = 0);
```

Diese Methode entspricht in ihrer Funktionsweise dem **Algorithmus 5.3** auf Seite 62. Die Parameter sind `k` $= k$ (Segmentweite), `m` $= m$ (Anzahl der Segmente) und `delta_dp` $= \delta$ (Reduktionsparameter). `shift` kann dazu benutzt werden um die „linke Grenze der Segmente“ i_l (siehe Gleichung (5.3) auf Seite 56) zu verschieben, das heißt es ist möglich $i_l = k(\ell - 1) +$ `shift` zu setzen.

```
void LocalLLL(const long i_l, const long i_m, const long i_r,
              long  &i_0, const T2 delta_T2, const bool flagSB = false);
```

Diese Methode ist die Umsetzung der Prozedur `loc-sLLL(`ℓ`)` (siehe Seite 125). Für das Doppelsegment $B_\ell, B_{\ell+1}$ werden die zwei `LocalLLL_sb()` Methoden benutzt, um die einzelnen Segmente B_ℓ bzw. $B_{\ell+1}$ gemäß dem LLL-Verfahren zu reduzieren. Die lokalen Transformationen werden global auf die (skalierte) Basismatrix übertragen.

```
bool LocalLLL_sb(const long i_l, const long i_r, const T2 delta_T2,
                 const bool init_s);
```

Diese Methode bearbeitet ein *Einzelsegment*. Hier werden die Basisvektoren skaliert. Lokale LLL-Reduktion des Einzelsegmentes (mit Übertragung der Transformationen auf globale Matrix). Globale Längenreduktion. Die Methode liefert booleschen Wert `true` zurück, falls Transformationen auf dem Segment stattgefunden haben.

```
bool LocalLLL_sb(const long i_l, const long i_r, const T2 delta_T2,
                 const bool flag1, long &e_l);
```

Die Funktionsweise ist analog zu obiger Methode `LocalLLL_sb()` (die 4 Eingabeparameter hat), allerdings wird hier eine zusätzliche Skalierung `M_s2` (siehe Methode `HRS_3()`) berücksichtigt.

```
bool LLL_FP(const long offset, const long k, const T2 delta_T2,
            mat_T2 &loc_R, mat_T2 &loc_T);
```

Die Matrix `loc_R` $= R_{\ell,\ell+1}$ wird lokal LLL-reduziert. Die Reduktion geschieht in purer Gleitpunktzahl-Arithmetik. Transformationen werden in der Matrix `loc_T` gespeichert. Liefert booleschen Wert `true` zurück, falls Transformationen auf dem Doppel-Segment stattgefunden haben.

```
void HRS_1(const long nu);
```

Diese Methode entspricht sinngemäß dem **Algorithmus C.1** auf Seite 121. Sie dient der Skalierung, Längenreduktion und Orthogonalisierung von $\boldsymbol{b}_\nu$. Es wird der Skalierungsexponent $e^{\ell,\ell+1}$ für das Doppelsegment $B_\ell, B_{\ell+1}$ berechnet. Der Vektor $\boldsymbol{b}^s_{i_l+1} := 2^{e_{i_l+1}}\boldsymbol{b}_{i_l+1}$ mit $e_{i_l+1} = e^{\ell,\ell+1}$ wird gespeichert, $i_l = k(\ell-1)$.

```
void HRS_2(const long nu, const long j_max, const long lredOffset);
```

1. Längenreduktion von $\boldsymbol{b}^s_\nu$ bezüglich $\boldsymbol{b}^s_1, \boldsymbol{b}^s_2, \ldots, \boldsymbol{b}^s_{\texttt{j\_max}}$.
2. Längenreduktion von $\boldsymbol{b}_\nu$ bezüglich $\boldsymbol{b}_{\texttt{lredOffset}+1}, \boldsymbol{b}_{\texttt{lredOffset}+2}, \ldots, \boldsymbol{b}_{\texttt{j\_max}}$.
3. Orthogonalisierung von $\boldsymbol{b}_\nu$ nur falls `j_max` $= \nu - 1$.

```
void HRS_3(const long nu, const long j_max, const long lredOffset,
           mat_ZZ &M_s2);
```

1. Längenreduktion von $\boldsymbol{b}^s_\nu$ bezüglich $\boldsymbol{b}^s_1, \boldsymbol{b}^s_2, \ldots, \boldsymbol{b}^s_{\texttt{j\_max}}$.
2. Längenreduktion von $\boldsymbol{b}_\nu$ bezüglich $\boldsymbol{b}_{\texttt{lredOffset}+1}, \boldsymbol{b}_{\texttt{lredOffset}+2}, \ldots, \boldsymbol{b}_{\texttt{j\_max}}$.

3. M_s2 $= \left[\boldsymbol{b}^{s_2}_{\texttt{lredOffset}+1} \;\; \boldsymbol{b}^{s_2}_{\texttt{lredOffset}+2} \;\; \dots \;\; \boldsymbol{b}^{s_2}_{\texttt{j\_max}}\right]$ ist eine weitere Skalierung von $\boldsymbol{b}_{\texttt{lredOffset}+1}, \boldsymbol{b}_{\texttt{lredOffset}+2}, \dots, \boldsymbol{b}_{\texttt{j\_max}}$. Längenreduktion von $\boldsymbol{b}^{s_2}_{\nu}$ bezüglich $\boldsymbol{b}^{s_2}_{\texttt{lredOffset}+1}, \boldsymbol{b}^{s_2}_{\texttt{lredOffset}+2}, \dots, \boldsymbol{b}^{s_2}_{\texttt{j\_max}}$.
4. Orthogonalisierung von $\boldsymbol{b}^{s}_{\nu}$ nur falls j_max $= \nu - 1$.

```
void SizeRed_vs_ScalBasVec(const long nu, const long j_max);
```

Längenreduktion von $\boldsymbol{b}_\nu$ bezüglich der skalierten Basisvektoren $\boldsymbol{b}^s_1, \boldsymbol{b}^s_2, \dots, \boldsymbol{b}^s_{\texttt{j\_max}}$.

```
T2 LogLocalGramDet(const long l, const long k);
```

Diese Methode wird in Schritt 8 von **Algorithmus 5.3** auf Seite 62 benutzt. Sie berechnet $\log_2\big(D(\ell)\big) = \log_2 \prod_{i=k\ell-k+1}^{k\ell} \big\|\widehat{\boldsymbol{b}}_i\big\|^2 = \sum_{i=k\ell-k+1}^{k\ell} 2 \cdot \log_2 \big\|\widehat{\boldsymbol{b}}_i\big\|$ für $\ell \in \{1, 2, \dots, m\}$. Dazu werden die Einträge $r_{i,i}$ der R Matrix benutzt. Da diese Matrix die Orthogonalisierung der skalierten Basisvektoren B^s repräsentiert, müssen die Skalierungsexponenten e_i bei der Berechnung berücksichtigt werden. Es gilt $r_{i,i} = \big\|\widehat{\boldsymbol{b}}^s_i\big\| = \big\|2^{e_i}\widehat{\boldsymbol{b}}_i\big\| = 2^{e_i} \cdot \big\|\widehat{\boldsymbol{b}}_i\big\|$. $D(\ell) = \prod_{i=k\ell-k+1}^{k\ell} \big\|\widehat{\boldsymbol{b}}_i\big\|^2 = \prod_{i=k\ell-k+1}^{k\ell} \frac{|r_{i,i}|^2}{2^{e_i 2}}$. Tatsächlich wird also berechnet:

$$\log_2\big(D(\ell)\big) = \sum_{i=k\ell-k+1}^{k\ell} 2 \cdot \log_2|r_{i,i}| - 2 \cdot e_i.$$

Der Wert $\log_2\big(D(\ell)\big)$ wird als Gleitpunktzahl vom Typ T2 zurückgegeben.

```
bool DecrCondition(const long j, const long t,
                   const T2 delta_T2, const T2 alpha_T2);
```

Für delta_T2 $= \delta$ und alpha_T2 $= \alpha = (\delta - \frac{1}{4})^{-1}$ vergleicht diese Methode

$$\big(\delta^t \|\widehat{\boldsymbol{b}}_j\|^2 \stackrel{?}{>} \alpha \|\widehat{\boldsymbol{b}}_{j+1}\|^2\big) \iff \big(t \cdot \log_2 \delta + 2 \cdot \log_2 \|\widehat{\boldsymbol{b}}_j\| \stackrel{?}{>} \log_2 \alpha + 2 \cdot \log_2 \|\widehat{\boldsymbol{b}}_{j+1}\|\big). \tag{6.3}$$

Falls die Ungleichung erfüllt ist, wird der boolesche Wert true zurückgegeben. Die Überprüfung dieser Ungleichung erfolgt in Schritt 8 von **Algorithmus 5.3** auf Seite 62. Analog zur Methode LogLocalGramDet() müssen auch hier die Skalierungsexponenten $e_j, ej + 1$ berücksichtigt werden. Tatsächlich wird berechnet:

$$t \cdot \log_2 \delta + 2 \cdot \log_2|r_{j,j}| - 2 \cdot e_j \stackrel{?}{>} \log_2 \alpha + 2 \cdot \log_2|r_{j+1,j+1}| - 2 \cdot e_{j+1}.$$

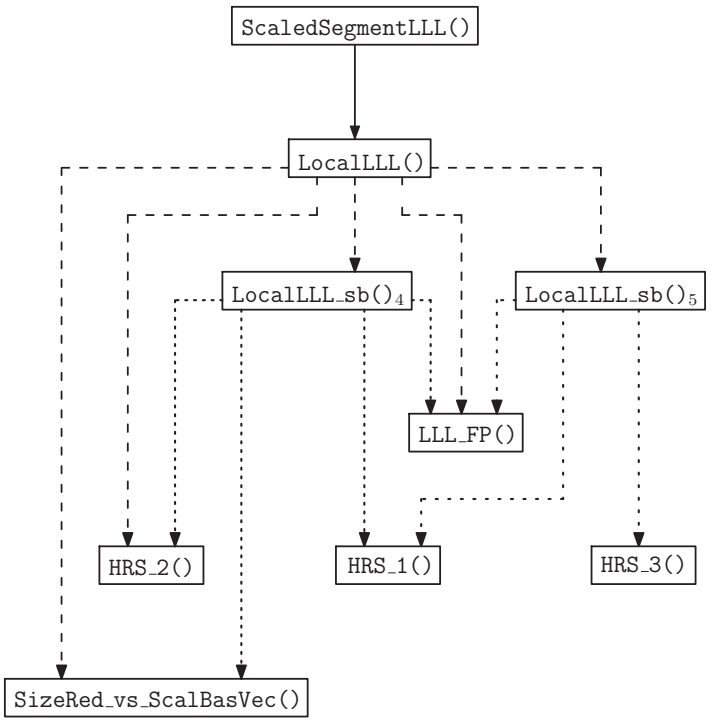


Abbildung 6.2: Abhängigkeiten der Methoden für die skalierte Segment LLL-Reduktion

6.3.7 Bemerkung (Rekursive Unterteilung der Segmente) *In* `LocalLLL()` *ist vorgesehen, dass die einzelnen Segmente rekursiv unterteilt werden können. Dies ist ein erster Schritt in Richtung iterierter Segment LLL-Reduktion nach [Koy und Schnorr, 2001a].*

Abbildung 6.2 veranschaulicht die Abhängigkeiten der Methoden. Die höher platzierten Methoden rufen die tiefer liegenden auf. Die Pfeile zeigen, welche Aufrufe möglich sind. `LocalLLL_sb()`$_4$ sei diejenige der beiden `LocalLLL_sb()` Methoden, die 4 Eingabeparameter hat. `LocalLLL_sb()`$_5$ hat 5 Eingabeparameter.

6.3.3 Primal/Duale Segmentreduktion

Bei der Benutzung der primal/dualen Reduktion beachte man die Bemerkung 5.3.5 auf Seite 70.

In `CLattice` ist die „erweiterte“ primal/duale Segmentreduktion in zwei Phasen implementiert (siehe Abschnitt 5.3.1 auf Seite 67). Die dort angegebenen Algorithmen sind in dieser Form auch programmiert worden.

```
bool PrimalDualRed(const long k, const long l, const double delta,
                   const long shift = 0, const long prune = 0);
```

Diese Methode realisiert **Algorithmus 5.7** auf Seite 71. Die Parameter `k, l, delta` entsprechen der Segmentweite k, der Stufe ℓ, und dem Reduktionsparameter δ. `shift` kann dazu benutzt werden um die „linke Grenze der Segmente“ i_l (siehe Gleichung (5.7) auf Seite 67) zu verschieben, das heißt es ist möglich $i_l = k(\ell - 1) +$ `shift` zu setzen. Falls der Parameter `prune` größer Null gesetzt wird, so wird die geschnittene Enumeration aktiviert; je größer `prune` ist umso stärker wird geschnitten. Empfehlenswert ist $10 \leq$ `prune` ≤ 16 für $k > 30$ zu wählen. Die Methode gibt einen booleschen Wert zurück, welcher F_{trans} in **Algorithmus 5.7** auf Seite 71 entspricht. Es werden `localENUM::BNEU_DUAL()` und `localENUM::BNEU_PRIMAL()` aus `enum.cpp` benutzt. Ihre Funktionsweise ist in [Ritter, 1997, S. 26 f.] beschrieben.

```
void PrimalDualSegLLL(const long k, const long m, const double delta_dp,
                      const long shift = 0);
```

Dies ist die Umsetzung von **Algorithmus 5.6** auf Seite 70. Die Parameter sind `k` $= k$ (Segmentweite), `m` $= m$ (Anzahl der Segmente) und der Reduktionsparameter `delta_dp` $= \delta$. Der Parameter `shift` ist bereits bei `PrimalDualRed()` erläutert worden.

6.4 Erzeugung von Gitterbasen

Damit wir die Eigenschaften (etwa Reduktionsgüte und Geschwindigkeit) der neuen Algorithmen zur Gitterbasenreduktion in Segmenten experimentell evaluieren können, benötigen wir geeignete Eingaben. Bei unseren Experimenten wurden nur Gitterbasen reduziert, die ganzzahlige und vollständige Gitter erzeugen. Dies entspricht der üblichen Praxis.

Wir beschreiben die Erzeugung von GGH-Gitterbasen und von „zufälligen Gitterbasen“. Genauere Erläuterungen folgen in Abschnitt 6.4.1 auf der nächsten Seite bzw. in Abschnitt 6.4.2 auf Seite 97. Beide Methoden haben ein ähnliches Vorgehen: Zuerst wird eine „gute“ (reduzierte) Gitterbasis $B_{\text{red}} \in \mathbb{Z}^{n \times n}$ erzeugt und durch geeignete unimodulare Transformationen konstruieren wir aus dieser eine zusätzliche, „schlechte“ (gemixte) Basismatrix $B_{\text{mix}} \in \mathbb{Z}^{n \times n}$. Nach Satz 2.2.5 auf Seite 15 erzeugen die zwei Gitterbasen dasselbe Gitter. Der wesentliche Unterschied zwischen ihnen ist die (euklidische) Länge der Basisvektoren. B_{red} hat sehr kurze Basisvektoren, wohingegen B_{mix} sehr lange Basisvektoren repräsentiert.

6.4.1 Definition (Korrespondierende Gitterbasen) *Gegeben seien zwei Gitterbasen $B', B \in \mathbb{R}^{d \times n}$. Wir sagen B' und B sind* korrespondierende *Gitterbasen, falls es eine unimodulare Matrix $T \in \mathsf{SL}_n(\mathbb{Z})$ gibt, so dass gilt: $B' = BT$ (B' und B erzeugen das gleiche Gitter).*

Mit B_{red} und B_{mix} haben wir korrespondierende Basismatrizen, die sich sehr gut dazu eignen, die Güte von Gitterreduktions-Programmen zu testen. Erhält ein solches Programm die Basismatrix B_{mix} als Eingabe, so sollte es im Idealfall – nach einer angemessenen Bearbeitungszeit – die reduzierte Matrix B_{red} ausgeben. Im allgemeinen wird dieses Ziel jedoch nicht erreichbar sein, da der Aufwand dafür zu groß ist.

Meistens ist man „nur“ am kürzesten Gittervektor $\lambda_1(L)$ eines Gitters L interessiert und nicht an der kürzesten Basis. Wir möchten daher feststellen, wie gut unsere Reduktionsalgorithmen den kürzesten Gittervektor in der Praxis approximieren. Dazu

sind die Basismatrizen $B_{\text{red}}, B_{\text{mix}}$ nützlich, insbesondere falls folgende Annahme erfüllt ist:

6.4.2 Annahme (Kürzester Gittervektor in B_{red}) *Gegeben seien zwei korrespondierende Gitterbasen $B_{\text{red}} = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{n\times n}$ und $B_{\text{mix}} \in \mathbb{Z}^{n\times n}$. Sind die Basisvektoren der Länge nach aufsteigend sortiert, so nehmen wir* heuristisch *an, dass $\|\boldsymbol{b}_1\| = \lambda_1\big(\mathcal{L}(B_{\text{red}})\big) = \lambda_1\big(\mathcal{L}(B_{\text{mix}})\big)$ gilt.*

Mit den korrespondierenden Gitterbasen $B_{\text{red}}, B_{\text{mix}}$ und der Annahme 6.4.2 testen wir die Reduktionsgüte: Ein Gitterreduktions-Algorithmus wird mit der Eingabe B_{mix} gestartet, und nach der Beendigung überprüft man, wie gut der kürzeste Gittervektor $\boldsymbol{b}_1$ approximiert worden ist.

Ohne Annahme 6.4.2, müssten wir den kürzesten Gittervektor mit Gleichung (2.6) oder Gleichung (2.7) auf Seite 21 und einer Approximation[13] der Hermite-Konstanten γ_n nach Gleichung (2.8) auf Seite 21 abschätzen. In diesem Fall wäre für den kürzesten Gittervektor eines Gitters $L = \mathcal{L}(B)$, $B \in Z^{n\times n}$, etwa

$$\lambda_1(L) \approx \sqrt{\gamma_n} \cdot (\mathsf{det}\ L)^{\frac{1}{n}}$$

anzunehmen. Natürlich ist auch dieses Vorgehen heuristisch. Die Nachteile hierbei, im Vergleich zu Annahme 6.4.2, sind offensichtlich: (a) die Gitterdeterminante ist zu berechnen, (b) im allgemeinen wird γ_n nur approximiert und (c) der kürzeste Gittervektor $\lambda_1(L)$ könnte tatsächlich noch kürzer als $\sqrt{\gamma_n} \cdot (\mathsf{det}\ L)^{\frac{1}{n}}$ sein.

6.4.1 GGH-Gitterbasen

Die GGH-Gitterbasen wurden in [Goldreich, Goldwasser und Halevi, 1997] vorgestellt. Mittels eines effizienten Algorithmus werden zwei verschiedene Basen $B_{\text{red-GGH}} \in \mathbb{Z}^{n\times n}$ und $B_{\text{mix-GGH}} \in \mathbb{Z}^{n\times n}$ desselben ganzzahligen Gitters $L \subseteq \mathbb{Z}^n$ erzeugt. Es gilt also $L = \mathcal{L}(B_{\text{red-GGH}}) = \mathcal{L}(B_{\text{mix-GGH}})$. Allerdings unterscheiden sich die beiden Basismatrizen in der Größe ihrer Einträge. Die Matrix $B_{\text{red-GGH}}$ besteht aus sehr kurzen Vektoren, wohingegen die Matrix $B_{\text{mix-GGH}}$ sehr lange Basisvektoren repräsentiert. In dem asymmetrischen GGH-Kryptosystem (siehe [Goldreich, Goldwasser und Halevi, 1997]) dient die „gute“ Basis $B_{\text{red-GGH}}$ als geheimer Schlüssel und die „schlechte“ Basis $B_{\text{mix-GGH}}$ ist der öffentliche Schlüssel. Wir stellen das Kryptoschema nicht vor, da es für unsere Zwecke nicht relevant ist. Interessierte Leser seien an [Goldreich, Goldwasser und Halevi, 1997; Nguyen, 1999; Koy, 1999; Micciancio, 2001] verwiesen.

Wir gehen davon aus, dass $B_{\text{red-GGH}}$ nicht weiter reduziert werden kann und das Annahme 6.4.2 gültig ist. Von $B_{\text{red-GGH}}$ ausgehend wird durch festgelegte Operationen die *Mix-Basismatrix* $B_{\text{mix-GGH}}$ konstruiert. Dazu werden in [Goldreich, Goldwasser und Halevi, 1997] mehrere Möglichkeiten vorgeschlagen und diskutiert. Das wesentliche Ziel von Goldreich, Goldwasser und Halevi ist gewesen, dass die resultierende Basismatrix $B_{\text{mix-GGH}}$ möglichst schwer[14] zu reduzieren ist.

[13] Für $1 \le n \le 8$ kann man natürlich die genauen Werte γ_n aus **Tabelle 2.1** auf Seite 21 nehmen.

[14] Die Konstruktion von $B_{\text{red-GGH}}$ und $B_{\text{mix-GGH}}$ erfolgt nach heuristischen Kriterien. Es scheint jedoch gerechtfertigt zu sein, anzunehmen dass eine Reduktion von $B_{\text{mix-GGH}}$ sehr aufwendig ist. Dies belegen unter anderen die Experimente in [Nguyen, 1999] und [Koy, 1999].

Algorithmus 6.4 Erzeugung einer Gitterbasis $B_{\text{red-GGH}}$

Eingabe: Dimension $n \in \mathbb{N}$; Schranken $s_1, s_2 \in \mathbb{Z}\ ,\ s_1 < s_2$; Multiplikationsfaktor $s_3 \in \mathbb{N}$

1: $P \leftarrow [p_{i,j}]_{1\le i,j\le n} \in \mathbb{Z}^{n\times n}\ ,\ p_{i,j} \in_R \{s_1, s_1+1, \ldots, s_2-1, s_2\}$
2: $B_{\text{red-GGH}} \leftarrow s_3\lceil\sqrt{n}+1\rceil E_n + P$ /* E_n ist die $n\times n$-Einheitsmatrix */

Ausgabe: Basismatrix $B_{\text{red-GGH}} \in \mathbb{Z}^{n\times n}$

Algorithmus 6.5 Erzeugung einer Gitterbasis $B^{(n,r)}_{\text{mix-GGH}}$

Eingabe: Anzahl der Mix-Runden $r \in \mathbb{N}$; Dimension $n \in \mathbb{N}$; mit **Algorithmus 6.4** erzeugte Matrix $B^{(n,r)}_{\text{red-GGH}} \in \mathbb{Z}^{n\times n}$

1: $B \leftarrow B^{(n,r)}_{\text{red-GGH}}$ /* $B = [b_{i,j}]_{1\le i,j\le n} = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{n\times n}$ */
2: **for** $i = 1, 2, \ldots, r$ **do**
3: /* i zählt die Mix-Runden */
4: **for** $j = 1, 2, \ldots, n$ **do**
5: /* bearbeite den j-ten Basisvektor */
6: Erzeuge Mix-Matrix $M_j = [m_{i,j}]_{1\le i,j\le n} \in \mathsf{SL}_n(\mathbb{Z})$ /* siehe **Algorithmus 6.7** */
7: $B \leftarrow BM_j$ /* Mixen: BM_j beschreibt die Addition $\boldsymbol{b}_j \leftarrow \boldsymbol{b}_j + \sum_{k\ne j} m_{k,j}\boldsymbol{b}_k$ */
8: **end for**
9: **end for**
10: $B^{(n,r)}_{\text{mix-GGH}} \leftarrow B$

Ausgabe: Mit r Mix-Runden erzeugte Basismatrix $B^{(n,r)}_{\text{mix-GGH}} \in \mathbb{Z}^{n\times n}$;
es gilt $\mathcal{L}\big(B^{(n,r)}_{\text{mix-GGH}}\big) = \mathcal{L}\big(B^{(n,r)}_{\text{red-GGH}}\big)$

Algorithmus 6.6 Erzeugung einer Gitterbasis $B^{[n,\ell]}_{\text{mix-GGH}}$

Eingabe: Mittlere Bitlänge $\ell \in \mathbb{N}$ der Einträge in der zu erzeugenden Gitterbasis $B^{[n,\ell]}_{\text{mix-GGH}}$; Dimension $n \in \mathbb{N}$; mit **Algorithmus 6.4** erzeugte Matrix $B^{[n,\ell]}_{\text{red-GGH}} \in \mathbb{Z}^{n\times n}$

1: $B \leftarrow B^{[n,\ell]}_{\text{red-GGH}}$ /* $B = [b_{i,j}]_{1\le i,j\le n} = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{n\times n}$ */
2: **repeat**
3: **for** $j = 1, 2, \ldots, n$ **do**
4: /* bearbeite den j-ten Basisvektor */
5: Erzeuge Mix-Matrix $M_j = [m_{i,j}]_{1\le i,j\le n} \in \mathsf{SL}_n(\mathbb{Z})$ /* siehe **Algorithmus 6.7** */
6: $B \leftarrow BM_j$ /* Mixen: BM_j beschreibt die Addition $\boldsymbol{b}_j \leftarrow \boldsymbol{b}_j + \sum_{k\ne j} m_{k,j}\boldsymbol{b}_k$ */
7: **end for**
8: $m \leftarrow \big(\sum_{i=1}^{n}\sum_{j=1}^{n} \mathsf{bitlength}(b_{i,j})\big)/n^2$
9: /* m ist die mittlere Bitlänge der aktuellen Einträge in B */
10: **until** $m \ge \ell$
11: $B^{[n,\ell]}_{\text{mix-GGH}} \leftarrow B$

Ausgabe: Matrix $B^{[n,\ell]}_{\text{mix-GGH}} = [b_{i,j}]_{1\le i,j\le n} \in \mathbb{Z}^{n\times n}$ mit $\ell \approx \big(\sum_{i=1}^{n}\sum_{j=1}^{n} \mathsf{bitlength}(b_{i,j})\big)/n^2$;
es gilt $\mathcal{L}\big(B^{[n,\ell]}_{\text{mix-GGH}}\big) = \mathcal{L}\big(B^{[n,\ell]}_{\text{red-GGH}}\big)$

Algorithmus 6.4 auf der vorherigen Seite erzeugt eine reduzierte Matrix $B_{\text{red-GGH}}$. Zur Konstruktion von $B_{\text{red-GGH}}$ wird in [Goldreich, Goldwasser und Halevi, 1997] unter anderen vorgeschlagen, die Parameter $s_1 = -4, s_2 = s_3 = 4$ zu wählen. Dieser Empfehlung folgen wir, falls nichts anderes angegeben ist. Man beachte die *diagonaldominante* Struktur von $B_{\text{red-GGH}}$. **Algorithmus 6.5** bzw. **Algorithmus 6.6** auf der vorherigen Seite erzeugt eine zu $B_{\text{red-GGH}}$ korrespondierende Matrix $B_{\text{mix}} \in \left\{B^{(n,r)}_{\text{mix-GGH}}, B^{[n,\ell]}_{\text{mix-GGH}}\right\}$. Der Unterschied zwischen **Algorithmus 6.5** und **Algorithmus 6.6** liegt in der Anzahl der Mix-Runden: In **Algorithmus 6.5** ist sie als Eingabe r vorgegeben, wohingegen in **Algorithmus 6.6** solange gemixt wird, bis eine spezifizierte mittlere Bitlänge ℓ der Matrixeinträge in $B^{[n,\ell]}_{\text{mix-GGH}}$ erreicht worden ist.

Algorithmus 6.7 Erzeugung einer Mix-Matrix $M_j \in \mathsf{SL}_n(\mathbb{Z})$

Eingabe: Dimension $n \in \mathbb{N}$; Spalte $j \in \{1, 2, \ldots, n\}$

1: Wähle die $n-1$ Zahlen $m_{1,j}, \ldots, m_{j-1,j}, m_{j+1,j}, \ldots, m_{n,j}$ zufällig aus $\{\pm 1, 0\}$. Hierbei ist die Wahrscheinlichkeitsverteilung $\Pr[X = 1] = \Pr[X = -1] = 1/7$.

2: $M_j \leftarrow \begin{bmatrix} 1 & & & m_{1,j} & & & \\ & \ddots & & \vdots & & 0 & \\ & & 1 & m_{j-1,j} & & & \\ & & & 1 & & & \\ & & & m_{j+1,j} & 1 & & \\ & 0 & & \vdots & & \ddots & \\ & & & m_{n,j} & & & 1 \end{bmatrix}$

3: /* $M_j \in \mathbb{Z}^{n \times n}$ hat bis auf die Hauptdiagonale und Spalte j nur Nullen als Einträge */

Ausgabe: $M_j \in \mathsf{SL}_n(\mathbb{Z})$

6.4.2 Zufällige Gitterbasen

Um Gitter-Basismatrizen erzeugen zu können, die nicht die diagonaldominante Struktur von $B_{\text{red-GGH}}$ aufweisen, ist es möglich, auf Schritt 2 von **Algorithmus 6.4** auf der vorherigen Seite zu verzichten. Wir bezeichnen solche Gitterbasen als „zufällig“ und schreiben $B_{\text{red-rand}}$. Erzeugt werden sie mit **Algorithmus 6.8** auf der nächsten Seite. Für große n sind die so konstruierten Basisvektoren mit sehr hoher Wahrscheinlichkeit linear unabhängig und erzeugen somit tatsächlich ein volldimensionales Gitter mit Rang n.

Algorithmus 6.9 bzw. **Algorithmus 6.10** auf der nächsten Seite erzeugt eine zu $B_{\text{red-rand}}$ korrespondierende Matrix $B_{\text{mix}} \in \left\{B^{(n,r)}_{\text{mix-rand}}, B^{[n,\ell]}_{\text{mix-rand}}\right\}$. Der Unterschied zwischen **Algorithmus 6.9** und **Algorithmus 6.10** liegt in der Anzahl der Mix-Runden: In **Algorithmus 6.9** ist sie als Eingabe r vorgegeben, wohingegen in **Algorithmus 6.10** solange gemixt wird, bis eine spezifizierte mittlere Bitlänge ℓ der Matrixeinträge in $B^{[n,\ell]}_{\text{mix-rand}}$ erreicht worden ist.

Algorithmus 6.8 Erzeugung einer Gitterbasis $B_{\text{red-rand}}$

Eingabe: Dimension $n \in \mathbb{N}$; Schranken $s_1, s_2 \in \mathbb{Z}\ ,\ s_1 < s_2$

1: $B_{\text{red-rand}} \leftarrow [p_{i,j}]_{1 \le i,j \le n} \in \mathbb{Z}^{n \times n}\ ,\ p_{i,j} \in_R \{s_1, s_1+1, \ldots, s_2-1, s_2\}$

Ausgabe: Basismatrix $B_{\text{red-rand}} \in \mathbb{Z}^{n \times n}$

Algorithmus 6.9 Erzeugung einer Gitterbasis $B^{(n,r)}_{\text{mix-rand}}$

Eingabe: Anzahl der Mix-Runden $r \in \mathbb{N}$; Dimension $n \in \mathbb{N}$; mit **Algorithmus 6.8** erzeugte Matrix $B^{(n,r)}_{\text{red-rand}} \in \mathbb{Z}^{n \times n}$

1: $B \leftarrow B^{(n,r)}_{\text{red-rand}}$ /* $B = [b_{i,j}]_{1 \le i,j \le n} = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{n \times n}$ */
2: **for** $i = 1, 2, \ldots, r$ **do**
3: /* i zählt die Mix-Runden */
4: **for** $j = 1, 2, \ldots, n$ **do**
5: /* bearbeite den j-ten Basisvektor */
6: Erzeuge Mix-Matrix $M_j = [m_{i,j}]_{1 \le i,j \le n} \in \mathsf{SL}_n(\mathbb{Z})$ /* siehe **Algorithmus 6.7** */
7: $B \leftarrow BM_j$ /* Mixen: BM_j beschreibt die Addition $\boldsymbol{b}_j \leftarrow \boldsymbol{b}_j + \sum_{k \ne j} m_{k,j}\boldsymbol{b}_k$ */
8: **end for**
9: **end for**
10: $B^{(n,r)}_{\text{mix-rand}} \leftarrow B$

Ausgabe: Basismatrix $B^{(n,r)}_{\text{mix-rand}} \in \mathbb{Z}^{n \times n}$; es gilt $\mathcal{L}\big(B^{(n,r)}_{\text{mix-rand}}\big) = \mathcal{L}\big(B^{(n,r)}_{\text{red-rand}}\big)$

Algorithmus 6.10 Erzeugung einer Gitterbasis $B^{[n,\ell]}_{\text{mix-rand}}$

Eingabe: Mittlere Bitlänge $\ell \in \mathbb{N}$ der Einträge in der zu erzeugenden Gitterbasis $B^{[n,\ell]}_{\text{mix-rand}}$; Dimension $n \in \mathbb{N}$; mit **Algorithmus 6.8** erzeugte Matrix $B^{[n,\ell]}_{\text{red-rand}} \in \mathbb{Z}^{n \times n}$

1: $B \leftarrow B^{[n,\ell]}_{\text{red-rand}}$ /* $B = [b_{i,j}]_{1 \le i,j \le n} = [\boldsymbol{b}_1\ \boldsymbol{b}_2\ \ldots\ \boldsymbol{b}_n] \in \mathbb{Z}^{n \times n}$ */
2: **repeat**
3: **for** $j = 1, 2, \ldots, n$ **do**
4: /* bearbeite den j-ten Basisvektor */
5: Erzeuge Mix-Matrix $M_j = [m_{i,j}]_{1 \le i,j \le n} \in \mathsf{SL}_n(\mathbb{Z})$ /* siehe **Algorithmus 6.7** */
6: $B \leftarrow BM_j$ /* Mixen: BM_j beschreibt die Addition $\boldsymbol{b}_j \leftarrow \boldsymbol{b}_j + \sum_{k \ne j} m_{k,j}\boldsymbol{b}_k$ */
7: **end for**
8: $m \leftarrow \big(\sum_{i=1}^{n}\sum_{j=1}^{n} \mathsf{bitlength}(b_{i,j})\big)/n^2$
9: /* m ist die mittlere Bitlänge der aktuellen Einträge in B */
10: **until** $m \ge \ell$
11: $B^{[n,\ell]}_{\text{mix-rand}} \leftarrow B$

Ausgabe: Matrix $B^{[n,\ell]}_{\text{mix-rand}} = [b_{i,j}]_{1 \le i,j \le n} \in \mathbb{Z}^{n \times n}$ mit $\ell \approx \big(\sum_{i=1}^{n}\sum_{j=1}^{n} \mathsf{bitlength}(b_{i,j})\big)/n^2$; es gilt $\mathcal{L}\big(B^{[n,\ell]}_{\text{mix-rand}}\big) = \mathcal{L}\big(B^{[n,\ell]}_{\text{red-rand}}\big)$

7 Experimentelle Ergebnisse

Wir geben mit `latred` erstellte experimentelle Resultate an.

Sei $[\vec{\boldsymbol{b}}_1\ \vec{\boldsymbol{b}}_2\ \ldots\ \vec{\boldsymbol{b}}_n] = [b_{i,j}]_{1\le i,j\le n} \in \mathbb{Z}^{n\times n}$ eine Gitterbasis. Wir setzen

$$\mathsf{SV}_{\|\cdot\|} := \min_{1\le i\le n} \|\vec{\boldsymbol{b}}_i\|\ , \quad \mathsf{AV}_{\|\cdot\|} := \frac{1}{n}\left(\sum_{i=1}^{n}\|\vec{\boldsymbol{b}}_i\|\right)\ , \quad \ell^{\mathrm{av}} := \frac{1}{n^2}\left(\sum_{i}^{n}\sum_{j}^{n}\mathsf{bitlength}(b_{i,j})\right).$$

7.1 Skalierte k-Segment LLL-Reduktion

Reduktionsexperimente 1. Das Verhalten der skalierten k-Segment LLL-Reduktion in Abhängigkeit der Segmentlänge k zeigt **Abbildung 7.1** auf Seite 101. Die Eingabematrix $B^{[500,100]}_{\text{mix-GGH}}$ wurde mit **Algorithmus 6.6** auf Seite 96 erzeugt (für **Algorithmus 6.4** auf Seite 96 benutzten wir $s_1 = -4, s_2 = s_3 = 4$). **Abbildung 7.1(b)** auf Seite 101 zeigt unter anderen die mittleren Bitlängen $\widetilde{\ell}^{\mathrm{av}}$ der mit Segmentlängen $k = 5, 10, \ldots, 95, 100$ reduzierten Basen. Die Experimente wurden auf einem Rechner mit 400 MHz Intel Pentium II Prozessor und 128 MByte Hauptspeicher durchgeführt. Verwendet wurden `quad_float` Typen und $\delta = 0.99$.

Reduktionsexperimente 2. Mit **Algorithmus 6.6** auf Seite 96 (wobei $s_1 = -4, s_2 = s_3 = 4$ in **Algorithmus 6.4** auf Seite 96 benutzt wurde) berechneten wir Basismatrizen $B^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}} \in \mathbb{Z}^{n\times n}$ für $n = 100, 200, 300, \ldots, 900, 1000$ und $\ell^{\mathrm{av}} = \frac{n}{2}, 200$ ($\ell^{\mathrm{av}} = 450, 500$ haben wir jedoch ausgelassen). Zusätzlich erzeugten wir auch Matrizen $B^{[600,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$ für $\ell^{\mathrm{av}} = 100, 150, 250, 350, 400$ und $B^{[1000,400]}_{\text{mix-GGH}}$.

Die Basen $B^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$ wurden mit dem skalierten k-Segment LLL-Verfahren reduziert. Die Durchführung der Experimente erfolgte auf einem Rechner mit 800 MHz AMD Athlon Prozessor und 448 MByte Hauptspeicher. Die Segmentlänge setzten wir fest auf $k = 50$. Wir wählten $\delta = 0.99$ bzw. $\delta = 0.999$. Mit $\delta = 0.99$ verwendeten wir `double` und `quad_float` Typen für die Gleitpunktzahlen-Arithmetik. Im Fall $\delta = 0.999$ wurde nur `quad_float` benutzt.

Die reduzierten Basen bezeichnen wir mit $\widetilde{B}^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$. Ihre mittleren Bitlängen $\tilde{\ell}^{\mathrm{av}}$ in Abhängigkeit der benutzten Parameter zeigt **Tabelle 7.1** auf Seite 102. Die Zeit ist im Format *hh:mm:ss* angegeben. Die grafische Darstellung zeigt **Abbildung 7.2** auf Seite 103. Die Reduktionen der Matrizen mit Dimensionen 700,

800 und 900 wurden bei der Verwendung von `double` abgebrochen, weil in einer der `HRS`-Routinen, eine vor der Reduktion spezifizierte maximale Anzahl von Iterationen beim längenreduzieren und skalieren überschritten wurde.

7.2 Primal/Duale Segmentreduktion

Wir vergleichen experimentell die Reduktionseigenschaften der primal/dualen Reduktion mit denen der BKZ-Reduktion.

Mit **Algorithmus 6.4** auf Seite 96 wurden Basismatrizen $B^{(n,2)}_{\text{red-GGH}} \in \mathbb{Z}^{n\times n}$ mit $n = 100, 128, 200, 256, 300$ erzeugt, wobei $s_1 = -4, s_2 = s_3 = 4$ gewählt worden ist. **Algorithmus 6.5** auf Seite 96 lieferte mit r=2 Mix-Runden die korrespondierenden Basen $B^{(n,2)}_{\text{mix-GGH}}$ für $n = 100, 128, 200, 256, 300$.

Reduktionsexperimente 3a. Reduziert wurden die Matrizen $B^{(n,2)}_{\text{mix-GGH}}$ für $n = 100$, 200, 300. Die Experimente wurden auf einem Rechner mit 400 MHz Intel Pentium II Prozessor und 128 MByte Hauptspeicher durchgeführt. Bei allen Verfahren wurden `double` Typen für die Gleitpunktzahlen-Arithmetik benutzt und $\delta = 0.99$ gesetzt. Es wurde nur die vollständige Aufzählung benutzt (keine geschnittene Enumeration). Sei $w_a \in \{15, 20, 25\}$.

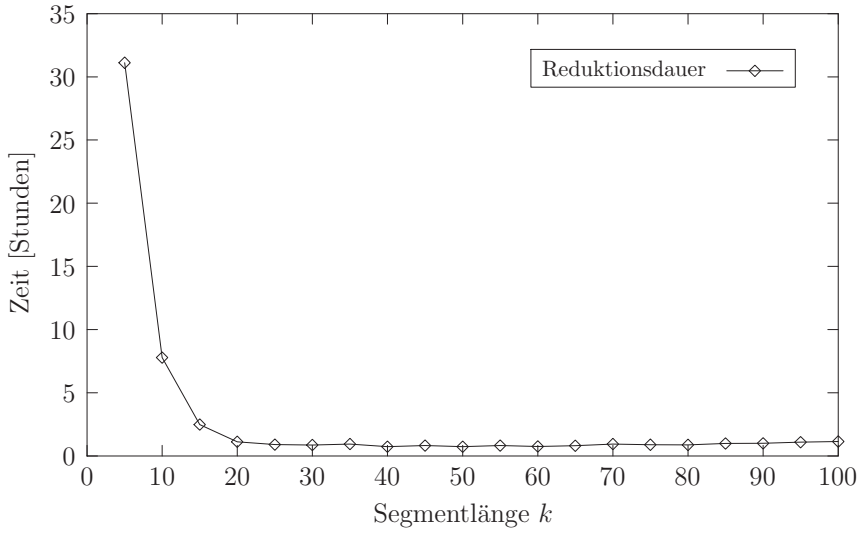
Primal/Duale Segmentreduktion: Zuerst erfolgte eine skalierte Segment LLL-Reduktion mit Segmentlänge 25. Die so vorreduzierte Basismatrix wurde dann mit der primal/dualen Segmentreduktion mit Segmentlänge w_a weiterreduziert.

BKZ-Reduktion: Zuerst erfolgte eine skalierte Segment LLL-Reduktion mit Segmentlänge 25. Die so vorreduzierte Basismatrix wurde mit dem LLL-Verfahren (Orthogonalisierung nach Givens) transformiert. Abschließend erfolgte die BKZ-Reduktion (Orthogonalisierung nach Givens) mit Blocklänge w_a.

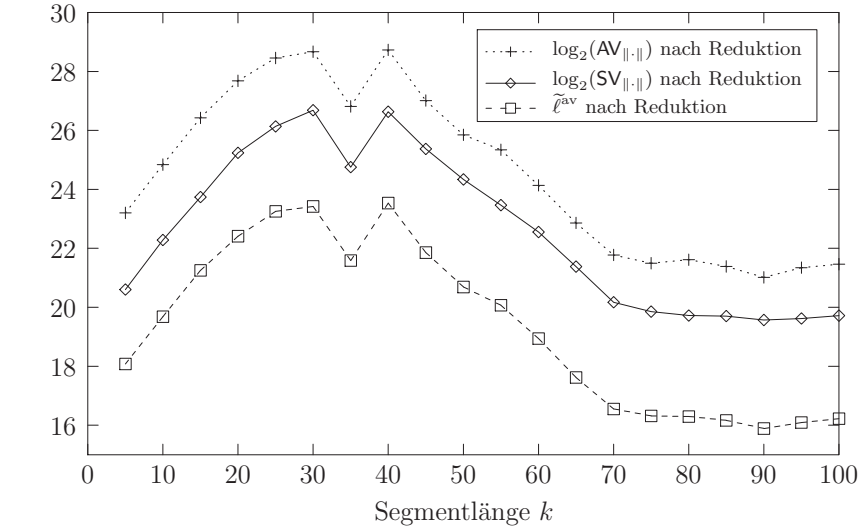
Reduktionsexperimente 3b. Reduziert wurden die Matrizen $B^{(n,2)}_{\text{mix-GGH}}$ für $n = 128$, 256. Die Experimente wurden auf einem Rechner mit 400 MHz Intel Pentium II Prozessor und 128 MByte Hauptspeicher durchgeführt. Bei allen Verfahren wurden `double` Typen für die Gleitpunktzahlen-Arithmetik benutzt und $\delta = 0.99$ gesetzt. Es wurde nur die vollständige Aufzählung benutzt (keine geschnittene Enumeration). Sei $w_b \in \{16, 24, 32\}$.

Primal/Duale Segmentreduktion: Zuerst erfolgte eine skalierte Segment LLL-Reduktion mit Segmentlänge w_b. Die so vorreduzierte Basismatrix wurde dann mit der primal/dualen Segmentreduktion mit Segmentlänge w_b weiterreduziert.

BKZ-Reduktion: Zuerst erfolgte eine skalierte Segment LLL-Reduktion mit Segmentlänge w_b. Die so vorreduzierte Basismatrix wurde mit dem LLL-Verfahren (Orthogonalisierung nach Givens) transformiert. Abschließend erfolgte die BKZ-Reduktion (Orthogonalisierung nach Givens) mit Blocklänge w_b.



(a) Reduktionsdauer

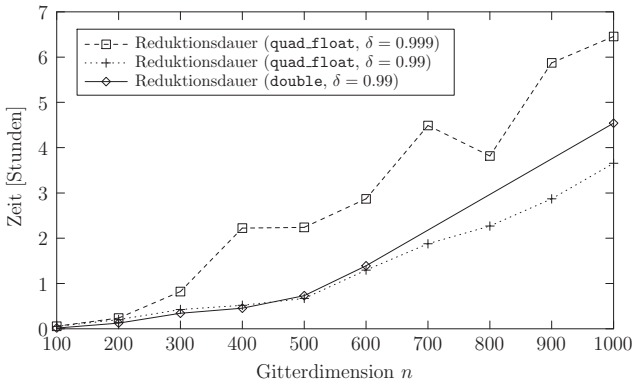


(b) Reduktionsgüte

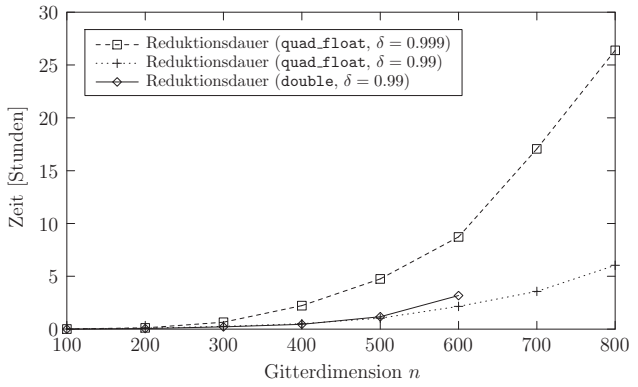
Abbildung 7.1: Reduktion von $B^{[500,100]}_{\text{mix-GGH}}$ mit variierender Segmentlänge k (Reduktionsexperimente 1)

$\widetilde{B}^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$		`double`		`quad_float`			
		$\delta = 0.99$		$\delta = 0.99$		$\delta = 0.999$	
n	ℓ^{av}	Zeit	$\widetilde{\ell}^{\mathrm{av}}$	Zeit	$\widetilde{\ell}^{\mathrm{av}}$	Zeit	$\widetilde{\ell}^{\mathrm{av}}$
100	50	00:00:14	4.764	00:00:45	4.738	00:00:46	4.798
100	200	00:01:01	4.890	00:03:10	4.785	00:03:22	4.641
200	100	00:03:12	12.483	00:05:11	12.947	00:07:11	9.409
200	200	00:07:26	11.710	00:12:02	11.703	00:14:15	9.209
300	150	00:12:26	23.785	00:15:35	23.825	00:39:05	14.329
300	200	00:20:31	19.910	00:25:33	23.968	00:49:19	14.085
400	200	00:27:16	30.139	00:31:06	30.345	02:13:24	18.179
500	200	00:43:52	32.830	00:40:16	32.894	02:14:23	23.097
500	250	01:10:16	33.829	01:01:40	33.738	04:44:58	24.414
600	100	00:28:45	20.718	00:34:12	20.747	00:50:21	18.934
600	150	00:50:07	31.569	00:48:48	31.571	01:50:16	24.397
600	200	01:23:23	31.871	01:17:49	31.625	02:52:10	26.570
600	250	01:43:07	41.526	01:21:14	41.523	05:06:12	28.107
600	300	03:10:16	39.235	02:09:02	39.427	08:43:36	28.875
600	350	02:57:41	45.620	01:55:03	45.919	12:19:32	27.298
600	400	04:19:00	44.789	02:59:06	44.994	16:03:46	27.245
700	200	*Abbruch*	?	01:52:37	34.692	04:29:20	28.569
700	350	*Abbruch*	?	03:34:07	43.685	17:03:44	31.329
800	200	*Abbruch*	?	02:16:02	35.079	03:49:02	30.511
800	400	*Abbruch*	?	06:02:30	46.057	26:23:12	36.588
900	200	*Abbruch*	?	02:52:10	34.359	05:52:19	26.462
1000	200	04:32:23	34.153	03:39:12	33.995	06:27:26	26.619
1000	400	—	—	10:06:38	52.953	—	—

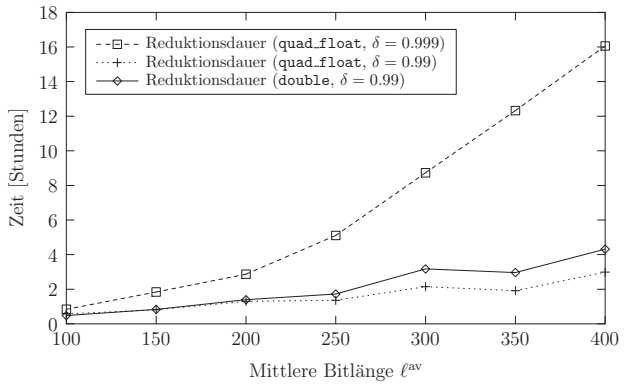
Tabelle 7.1: Für die Reduktion von $B^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$ benötigte Zeit (Segmentlänge $k = 50$), und die mittleren Bitlängen $\widetilde{\ell}^{\mathrm{av}}$ der *reduzierten* Basen $\widetilde{B}^{[n,\ell^{\mathrm{av}}]}_{\text{mix-GGH}}$ (Reduktionsexperimente 2)



(a) Reduktion von $B^{[n,200]}_{\text{mix-GGH}}$



(b) Reduktion von $B^{[n,n/2]}_{\text{mix-GGH}}$



(c) Reduktion von $B^{[600,\ell^{\text{av}}]}_{\text{mix-GGH}}$

Abbildung 7.2: Skalierte k-Segment LLL-Reduktion mit Segmentlänge $k = 50$ (Reduktionsexperimente 2)

Dimension n	$B^{(n,2)}_{\text{red-GGH}}$		$B^{(n,2)}_{\text{mix-GGH}}$	
	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$
100	46.1	51.1	$\approx 0.23824 \cdot 10^{10}$	$0.65975 \cdot 10^{11}$
128	55.1	59.6	$\approx 0.14306 \cdot 10^{12}$	$0.53568 \cdot 10^{14}$
200	69.1	73.6	$\approx 0.25660 \cdot 10^{19}$	$0.33217 \cdot 10^{21}$
256	75.3	79.4	$\approx 0.41625 \cdot 10^{26}$	$0.59762 \cdot 10^{28}$
300	83.7	88.2	$\approx 0.28480 \cdot 10^{27}$	$0.72954 \cdot 10^{30}$

Tabelle 7.2: Eigenschaften der Basen (Reduktionsexperimente 3)

In **Tabelle 7.2** sind die Eigenschaften der Eingabe-Basen $B^{(n,2)}_{\text{mix-GGH}}$ dargestellt. Die Spalte $B^{(n,2)}_{\text{red-GGH}}/\mathsf{SV}_{\|\cdot\|}$ zeigt vermutlich die Längen der kürzesten Gittervektoren.

Die Eigenschaften der reduzierten Basen sind den folgenden Tabellen zu entnehmen. Die mit Zeit_Σ beschrifteten Spalten in den Tabellen geben die jeweiligen *Gesamtzeiten* (Zeiten für Vorreduktion(en) + Zeiten für finale Reduktionen) für die Reduktionen im Format *hh:mm:ss* an.

Segment-/	Primal/Duale-Reduktion			BKZ-Reduktion		
Blocklänge	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$	Zeit_Σ	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$	Zeit_Σ
15	46.1	51.1	00:01:08	46.1	51.1	00:00:50
20	46.1	51.1	00:00:57	46.1	51.1	00:00:52
25	46.1	51.1	00:01:29	46.1	51.1	00:00:55

Tabelle 7.3: Reduktion von $B^{(100,2)}_{\text{mix-GGH}}$ (Reduktionsexperimente 3a)

Segment-/	Primal/Duale-Reduktion			BKZ-Reduktion		
Blocklänge	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$	Zeit_Σ	$\mathsf{SV}_{\|\cdot\|}$	$\mathsf{AV}_{\|\cdot\|}$	Zeit_Σ
16	337	507	00:08:56	55.1	59.6	00:05:16
24	332	457	00:05:26	55.1	59.6	00:04:19
32	55.1	59.6	00:10:55	55.1	59.6	00:04:49

Tabelle 7.4: Reduktion von $B^{(128,2)}_{\text{mix-GGH}}$ (Reduktionsexperimente 3b)

Segment-/	Primal/Duale-Reduktion			BKZ-Reduktion		
Blocklänge	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$
15	1396	2376	01:29:55	968	1228	00:36:39
20	1328	2126	00:54:45	772	1080	01:39:48
25	1148	2069	00:43:51	662	1007	34:51:01

Tabelle 7.5: Reduktion von $B^{(200,2)}_{\text{mix-GGH}}$ (Reduktionsexperimente 3a)

Segment-/	Primal/Duale-Reduktion			BKZ-Reduktion		
Blocklänge	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$
16	3083	7563	05:16:56	2118	2774	02:23:25
24	2695	5106	03:06:26	1762	2327	41:39:57
32	2545	3955	02:40:43	?	?	?[1]

Tabelle 7.6: Reduktion von $B^{(256,2)}_{\text{mix-GGH}}$ (Reduktionsexperimente 3b)

Segment-/	Primal/Duale-Reduktion			BKZ-Reduktion		
Blocklänge	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$	$SV_{\|\cdot\|}$	$AV_{\|\cdot\|}$	$Zeit_{\Sigma}$
15	7813	18951	11:18:22	4251	5649	04:57:50
20	7121	15865	07:27:06	3699	4805	12:06:16
25	6227	14353	05:05:11	3575	4933	30:10:53

Tabelle 7.7: Reduktion von $B^{(300,2)}_{\text{mix-GGH}}$ (Reduktionsexperimente 3a)

[1] Die Reduktion wurde nach ca. 3 Tagen abgebrochen.

Anhang A Benutzung von `latred`

`latred` ist ein Kommandozeilenprogramm. Beim Programmaufruf muss bzw. kann durch *Optionen* und *Argumente* angegeben werden welche Aktionen mit welchen Parametern ausgeführt werden sollen. Immer anzugeben ist `latred --mode` *`mode-string`*. Durch *`mode-string`* wird der „Betriebsmodus“ festgelegt (vgl. **Tabelle A.1** auf der nächsten Seite). Je nach gewählten Modus sind manche Optionen pflicht, verboten oder optional. Die folgenden Angaben beziehen sich auf Version 0.55 von `latred`. Eingabematrizen können entweder aus einer Datei (mit `--input`) oder über die Standardeingabe `stdin` (zum Beispiel mit dem Umleitungsoperator „`<`“) eingelesen werden. Die Matrizen müssen im „NTL-Textformat für Matrizen“ vorliegen [siehe Shoup, 2001]. Die *Zeilenvektoren* der Matrix bilden eine Gitterbasis.

Gültige[1] Programmaufrufe sind zum Beispiel:

```
latred --mode gen-mat-GGH --gen-mat-quadratic 100 --gen-mat-mix-rounds 2
       --gen-mat-init-mult 4 --gen-mat-init-lower -4 --gen-mat-init-upper 4
       [> ausgabe.txt]

latred --mode gen-mat-GGH --gen-mat-quadratic 100 --gen-mat-av-bitlength 50
       --gen-mat-init-lower -4 --gen-mat-init-upper 4 --gen-mat-init-mult 4
       [> ausgabe.txt]
```

Die zu reduzierende Basismatrix sei in der Textdatei `dim100-bit50.ggh.mix` gespeichert.

```
latred --mode scal-seg-LLL --scal-seg-LLL-segsize 25 [--delta 0.985] [--time-diff 30]
       [--fpa quad_float] --input dim100-bit50.ggh.mix
```

Die reduzierte Basismatrix wird in der Datei `s_segL3_r100.red` gespeichert.

```
latred --mode info [--first-vec 5 --last-vec 20] --input s_segL3_r100.red [--det]
       [> ausgabe.txt]

latred --mode ntl-G-LLL [--time-diff 30] [--fpa double] [--delta 0.95]
       --basmat-file LLL_reduzierte_basis.txt --input s_segL3_r100.red
       [> ausgabe.txt] [2> fehlerausg.txt]

latred --mode scal-seg-LLL+primdual --primdual-segsize 25 [--prune 12] [--delta 0.85]
       [--fpa quad_float] [--time-diff 60] [--basmat-file primaldual_reduz_basis.txt]
       < dim100-bit50.ggh.mix [> ausgabe.txt] [2> fehlerausg.txt]

latred --mode ntl-BKZ --blocksize 35 [--prune 12] [--delta 0.90] [--time-diff 120]
       [--fpa double+quad_float] --basmat-file BKZ_reduzierte_basis.txt
       --input dim100-bit50.ggh.mix [> ausgabe.txt] [2> fehlerausg.txt]
```

[1] Reihenfolge der Optionen und Argumente ist irrelevant. In eckigen Klammern [...] eingeschlossene Optionen/Argumente können weggelassen werden.

mode-string	Bedeutung	Benutztes Orthog.-Verfahren
info	Informationen über Matrix ausgeben	—
gen-mat-GGH	Korrespondierende GGH-Gitterbasen (wie in Abschnitt 6.4.1 auf Seite 95) erzeugen	—
gen-mat-random	Korrespondierende Gitterbasen (wie in Abschnitt 6.4.2 auf Seite 97) erzeugen	—
scal-LLL	Skalierte LLL-Reduktion (nach [Koy und Schnorr, 2001b])	Householder & Givens
scal-seg-LLL	Skalierte k-Segment LLL-Reduktion (**Algorithmus 5.3** auf Seite 62)	Householder & Givens
scal-seg-LLL+primdual	Erweiterte Primal/Dual Segmentreduktion (**Algorithmus 5.6** auf Seite 70)	Householder & Givens
ntl-BKZ	BKZ Reduktion (NTL-Routine)	Gram-Schmidt
ntl-G-BKZ	BKZ Reduktion (NTL-Routine)	Givens
ntl-LLL	LLL Reduktion (NTL-Routine	Gram-Schmidt
ntl-G-LLL	LLL Reduktion (NTL-Routine)	Givens

Tabelle A.1: Mit `--mode` *mode-string* auswählbare Betriebsmodi von `latred`

Die Optionen und Argumente beschreiben wir in den folgenden Tabellen. In der Spalte Pflicht ist angegeben, ob die Option Pflicht ist (Eintrag: Ja) oder nicht (Eintrag: Nein). Die Angabe von Optionen, deren Pflicht-Spalte ein „$*$“ enthält, ist von anderen Optionen abhängig. Ein „$+$“ bedeutet, dass die Verwendung möglich, aber nicht zwingend ist. Das „$-$“ in **Tabelle A.4** auf Seite 110 zeigt an, welche Optionen nicht mit den NTL-Routinen verwendbar sind.

Optionen/Argumente	Pflicht	Bedeutung
`--first-vec` i	Nein	Siehe nächste Zeile
`--last-vec` j	Nein	Sind $i, j \in \mathbb{N}$ $(i \leq j)$ angegeben, so werden auch Informationen über die Vektoren $\vec{b}_i, \vec{b}_{i+1}, \ldots, \vec{b}_j$ ausgegeben
`--det`	Nein	Determinante der Matrix berechnen

Tabelle A.2: Mit `--mode info` benutzbare Optionen/Argumente

Optionen/Argumente	Pflicht	Bedeutung
`--gen-mat-quadratic` n	Ja	Dimension $n \in \mathbb{N}$ der zu erzeugenden Gitterbasen
`--gen-mat-mix-rounds` r	$*$	Anzahl der Mix-Runden $r \in \mathbb{N}$, siehe **Algorithmus 6.5** auf Seite 96 bzw. **Algorithmus 6.9** auf Seite 98
`--gen-mat-av-bitlength` ℓ	$*$	Mittlere Bitlänge $\ell \in \mathbb{N}$ der Einträge, siehe **Algorithmus 6.6** auf Seite 96 bzw. **Algorithmus 6.10** auf Seite 98
`--gen-mat-init-lower` s_1	Ja	siehe **Algorithmus 6.4** auf Seite 96 bzw. **Algorithmus 6.8** auf Seite 98
`--gen-mat-init-upper` s_2	Ja	siehe **Algorithmus 6.4** auf Seite 96 bzw. **Algorithmus 6.8** auf Seite 98
`--gen-mat-init-mult` s_3	$*$	siehe **Algorithmus 6.4** auf Seite 96
`--gen-mat-init-seed` *string*	$+$	Hashwert von *string* als Saat für Pseudozufallszahlengenerator benutzen
`--gen-mat-file` *string*	$+$	Erzeugte Matrizen in den Dateien *string*.[ggh|matrix].[red|mix] speichern
`--gen-mat-dir` *dir*	$+$	Dateien im Verzeichnis *dir* speichern

Tabelle A.3: Mit `--mode gen-mat-GGH` bzw. `--mode gen-mat-random` benutzbare Optionen/Argumente

Optionen/Argumente	Pflicht	NTL	Bedeutung
`--fpa` *string*	+	+	Gleitpunktzahlen vom Type *string* $\in$ {double, quad_float, xdouble, double+quad_float} verwenden
`--delta` δ	+	+	Reduktionsparameter $\delta \in \mathbb{R}_{\texttt{GZ-DT}}$
`--sizered-max-iter` m	+	–	Iteriere die Längenreduktion in HRS höchstens $m \in \mathbb{N}$ mal
`--reduce-to` j	+	–	Reduziere nur die ersten $j \in \mathbb{N}$ Basisvektoren
`--time-diff` t	+	+	Reduktionsfortschritt alle t Sekunden speichern bzw. (bei NTL-Routinen) an `stdout` ausgeben
`--data-operat-off`	Nein	–	Anzahl der Operationen nicht protokollieren
`--data-time-off`	Nein	–	In den Routinen verbrachte Zeit nicht protokollieren
`--progress-off`	Nein	–	Zeitpunkte wann die Reduktion die einzelnen Basisvektoren zum ersten mal „anfasst“ nicht protokollieren
`--stat-start-off`	Nein	–	Status/Informationen über Basismatrix zu Beginn der Reduktion nicht protokollieren
`--stat-final-off`	Nein	–	Status/Informationen über Basismatrix am Ende der Reduktion nicht protokollieren
`--stat-progr-off`	Nein	–	Reduktionsfortschritt nicht protokollieren
`--basmat-save-off`	Nein	–	Reduzierte Basismatrix nicht speichern
`--progress-file` *dateiname*	+	–	Datei in der Zeitpunkte protokolliert werden, wann die einzelnen Basisvektoren zum ersten mal „angefasst“ werden
`--basmat-dir` *dir*	+	–	Speichere reduzierte Basismatrix ins Verzeichnis *dir*
`--basmat-file` *dateiname*	+	+	Speichere reduzierte Matrix in der Datei *dateiname*
`--dir` *dir*	+	–	Alle erzeugten Dateien im Verzeichnis *dir* speichern
`--stat-file-dir` *dir*	+	–	Protokolldateien des Reduktionsfortschrittes im Verzeichnis *dir* speichern
`--prefix` *string*	+	–	Benutze *string* als Präfix für Protokoll-Dateinamen

Tabelle A.4: Allgemeine Optionen/Argumente für die Reduktionsverfahren

Optionen/Argumente	Pflicht	Bedeutung
`--abs-mu` p_μ	+	Ersetzt in Schritt 3 von **Algorithmus C.1** auf Seite 121 die Schranke 0.51 durch $p_\mu \in \mathbb{R}_{\texttt{GZ-DT}}$
`--loop` p_l	+	Ersetzt in Schritt 4 von **Algorithmus C.1** auf Seite 121 die Schranke 2^{10} durch $p_l \in \mathbb{R}_{\texttt{GZ-DT}}$
`--loop-bits` b_l	+	Ersetzt in Schritt 4 von **Algorithmus C.1** auf Seite 121 die Schranke 2^{10} durch $2^{b_l} \in \mathbb{R}_{\texttt{GZ-DT}}$, $b_l \in \mathbb{N}$
`--exp-bits` b_e	+	Ersetzt in Schritt 5 von **Algorithmus C.1** auf Seite 121 den Wert 30 durch $b_e \in \mathbb{N}$

Tabelle A.5: Mit `--mode scal-LLL` benutzbare Optionen/Argumente

Optionen/Argumente	Pflicht	Bedeutung
`--scal-seg-LLL-segsize` k	Ja	Segmentlänge $k \in \mathbb{N}$
`--recur` k_r	Nein	Unterteile Segmente der Länge $k \geq k_r$ in 2 gleich grosse Teilsegmente und wende `LocalLLL()` rekursiv an (vgl. Bemerkung 6.3.7 auf Seite 93)
`--decr-cond-variant` v	+	Verändert die Bedingung $\delta^{k^2} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha\|\widehat{b}_{k(\ell-1)+1}\|^2$ in Schritt 7 von **Algorithmus 5.3** auf Seite 62; siehe **Tabelle A.7** auf der nächsten Seite. Ohne Angabe von „`--decr-cond-variant` v“ wird die „Standardbedingung“ ($v = 1$) verwendet.
`--seg-abs-mu` p_μ	+	Ersetzt in Schritt 3 von **Algorithmus C.1** auf Seite 121 die Schranke 0.51 durch $p_\mu \in \mathbb{R}_{\texttt{GZ-DT}}$
`--seg-loop` p_l	+	Ersetzt in Schritt 4 von **Algorithmus C.1** auf Seite 121 die Schranke 2^{10} durch $p_l \in \mathbb{R}_{\texttt{GZ-DT}}$
`--seg-loop-bits` b_l	+	Ersetzt in Schritt 4 von **Algorithmus C.1** auf Seite 121 die Schranke 2^{10} durch $2^{b_l} \in \mathbb{R}_{\texttt{GZ-DT}}$, $b_l \in \mathbb{N}$
`--maxnorm` m	+	Siehe `CLattice::LLL_FP()` in `lattice.cpp`
`--maxnorm-bits` m	+	Siehe `CLattice::LLL_FP()` in `lattice.cpp`
`--max-prec-loss-bits` P	+	Setzt $P^N_{\text{HRS-max-prec-loss}} = 2^P$ (vgl. Gleichung (6.2) auf Seite 88)

Tabelle A.6: Mit `--mode scal-seg-LLL` benutzbare Optionen/Argumente

v	Verwendete Bedingung
1	$\delta^{k^2} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha \|\widehat{b}_{k(\ell-1)+1}\|^2$
2	$\delta^{k} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha \|\widehat{b}_{k(\ell-1)+1}\|^2$
4	$\delta^{n} \|\widehat{b}_{k(\ell-1)}\|^2 > \alpha \|\widehat{b}_{k(\ell-1)+1}\|^2$
8	Nur die Determinantenbedingung wird überprüft

Tabelle A.7: Heuristische Reduktionsbedingungen für die skalierte Segment LLL-Reduktion (siehe auch **Tabelle A.6** auf der vorherigen Seite)

Optionen/Argumente	Pflicht	Bedeutung
`--primdual-segsize` k	Ja	Segmentlänge $k \in \mathbb{N}$
`--prune` p	Nein	Geschnittene Enumeration verwenden, falls Pruningparameter $p > 0, p \in \mathbb{N}$
`--shift-segments` σ_1:σ_2:...:σ_u	Nein	Siehe `::SetShiftingParameters()` in `param.cpp`

Tabelle A.8: Mit `--mode scal-seg-LLL+primdual` benutzbare Optionen/Argumente

Optionen/Argumente	Pflicht	Bedeutung
`--blocksize` β	Ja	Blocklänge $\beta \in \mathbb{N}$
`--prune` p	Nein	Geschnittene Enumeration verwenden, falls Pruningparameter $p > 0, p \in \mathbb{N}$

Tabelle A.9: Mit `--mode ntl-BKZ` bzw. `--mode ntl-G-BKZ` benutzbare Optionen/Argumente

Anhang B Quellcode von `latred`

Der Quellcode von `latred` (Version 0.55), GNU MP (Version 4.0.1) und NTL (Version 5.2) ist auf dem beiliegenden Datenträger gespeichert. Dort sind auch Informationen über die `latred` Quellcode-Dateien und zur Compilation enthalten. Der Datenträger ist nur den Exemplaren der Diplomarbeit beigelegt, die dem Prüfungsvorsitzenden des Institus für Informatik der Goethe-Universität Frankfurt am Main ausghändigt worden sind. Bei Interesse an dem Quellcode von `latred` kann eine Anfrage per e-mail an bartol@informatik.uni-frankfurt.de geschickt werden. Sollte diese Adresse nicht mehr gültig sein, so kann eventuell über die Arbeitsgruppe von Prof. Dr. Schnorr (http://www.mi.informatik.uni-frankfurt.de) der Quellcode bezogen werden.

Anhang C Die skalierte Segment LLL-Reduktion mit Gleitpunktzahlen

In diesem Anhang geben wir die Arbeit [Koy und Schnorr, 2001b] inhaltlich vollständig wieder. Für die freundliche Genehmigung ihre Arbeit abzudrucken, sei an dieser Stelle Henrik Koy und Claus Peter Schnorr herzlichst gedankt. Aus Satz-technischen Gründen entspricht das Layout hier nicht dem in der original Veröffentlichung, sondern ist an die „Optik“ dieser Diplomarbeit angelehnt. Auch ist die Nummerierung der Abschnitte, Definitionen, Sätze, Gleichungen, Abbildungen und Tabellen verändert.

Leider enthält das Paper einige kleine Fehler. Darauf gehen wir im folgenden kurz ein:

- Die Beschreibung der Subroutine **loc-LLL**(l) auf Seite 124 passt nicht zu dem **Algorithmus C.3** auf Seite 124. Falls der Aufruf von **loc-LLL**(l) – wie im Paper angegeben – die beiden Segmente B_{l-1} und B_l bearbeitet, so würde aufgrund der (**while** $l \leq m-1$)-Schleife in **Algorithmus C.3** das Segment B_m niemals bearbeitet werden. Damit der Algorithmus eine gemäß Definition C.6.1 auf Seite 124 k-Segment LLL-reduzierte Basis ausgibt, muss **loc-LLL**(l) die Segmente B_l und B_{l+1} bearbeiten und der Rumpf des Algorithmus muss auch für $l = m$ durchlaufen werden. Das heißt die Abbruchbedingung lautet **while** $l \leq m$ (der Aufruf von **loc-LLL** für l=m entfällt dann natürlich). Eine korrekte Version dieses Algorithmus haben wir auf auf Seite 60 aufgeschrieben.
- Auch die Beschreibung der Subroutine **loc-sLLL**(l) auf auf Seite 125 passt nicht zu dem **Algorithmus C.4** auf Seite 125. Siehe obigen Punkt.

  Allerdings ist in **Algorithmus C.4** noch ein Fehler: Die Bedingung

  $$D(l-1) > (\alpha/\delta)^{k^2} D(l) \textbf{ or } \delta^{k^2} \|\widehat{b}_{kl}\|^2 > \alpha \|\widehat{b}_{kl+1}\|^2$$

  „betrachtet“ für festes l gleich drei Segmente. Die Ungleichung vor dem **or** bezieht sich auf die Segmente B_{l-1} und B_l. Die Ungleichung nach dem **or** vergleicht den „Übergang“ von Segment B_l zu B_{l+1}. Korrekterweise müssen sich jedoch sowohl die linke als auch die rechte Ungleichung auf die selben zwei Segmente beziehen, falls beide Ungleichungen – wie in **Algorithmus C.4** eigentlich beabsichtigt – auf derselben Stufe l überprüft werden sollen (vergleiche mit Definition C.6.1 auf Seite 124).
- Die in dem Paper erwähnten Gleitpunktzahlen mit 106 Präzisionsbits sind vom NTL Typ `quad_float`. Daher haben sie tatsächlich nur eine 105 Bits breite Mantisse (siehe Abschnitt 6.1.2.1 auf Seite 76).

Die in Abschnitt C.8 auf Seite 126 angegebenen experimentellen Ergebnisse wurden mit einer früheren Version von `latred` erzeugt.

Segment LLL-Reduction with Floating Point Orthogonalization

Henrik Koy[1] and Claus Peter Schnorr[2]

Abstract. We associate with an integer lattice basis a *scaled basis* that has orthogonal vectors of nearly equal length. The orthogonal vectors or the QR-factorization of a scaled basis can be accurately computed up to dimension 2^{16} by Householder reflexions in floating point arithmetic (fpa) with 53 precision bits.

We develop a highly practical fpa-variant the new *segment* LLL-*reduction* of Koy and Schnorr [KS01]. The LLL-steps are guided in this algorithm by the Gram-Schmidt coefficients of an associated scaled basis. The new reduction algorithm is much faster than previous codes for LLL-reduction and performs well beyond dimension 1000.

Keywords. LLL-reduction, Householder reflexion, floating point arithmetic, stability, scaled basis, segment LLL-reduction, local LLL-reduction.

A.1 Introduction.

Practical algorithms for LLL-reduction compute the orthogonal vectors of a basis in floating point arithmetic (fpa). The corresponding fpa-errors must be small otherwise the size-reduction included in the orthogonalization gets unstable. Up to dimension 250 Gram-Schmidt orthogonalization of an LLL-reduced basis can be done in fpa with some correction steps [SE91]. Householder orthogonalization has better stability, as was shown in [RS96]. In practice it yields up to dimension 350 a sufficient QR-factorization of the basis matrix — but this process gets unstable in dimension 400. [S88] presents a method for correcting the Gram-Schmidt coefficients using Schulz's method of matrix inversion. This method is provably stable but requires $O(n + \log_2 M)$ precision bits, where M bounds the Euclidean length of the basis vectors. It is useless for fpa with 53 precision bits.

In this paper we associate with an integer lattice basis a *scaled basis* that has orthogonal vectors of nearly equal length. The orthogonal vectors or the QR-factorization of a scaled basis can be accurately computed up to dimension 2^{16} by Householder reflexions in floating point arithmetic (fpa) with 53 precision bits. We present an algorithm **HRS** that efficiently generates an associated scaled basis. We present a fpa-variant of LLL-reduction where LLL-steps are guided by the orthogonal vectors of an associated scaled basis. In particular, size-reduction is done against the scaled basis. The weaker size-reduction does in practice not degrade the quality of the reduced basis.

We present a fpa-variant of segment LLL-reduction, a novel concept proposed in [KS01]. The algorithm **segment sLLL** performs scaled segment LLL-reduction, so that all LLL-steps are guided by the orthogonaliation of an associated scaled basis. The algorithm **segment sLLL** is a very efficient reduction algorithm. Its efficiency comes from local LLL-reduction of two consecutive segments B_{l-1}, B_l that is done by reducing the local matrix R_l in fpa. To make this local LLL-reduction possible in the limits of fpa it is necessary to bound the

[1] Deutsche Bank AG, Frankfurt am Main, henrik.koy@db.com

[2] Fachbereiche Mathematik und Informatik, Universität Frankfurt, PSF 111932, D-60054 Frankfurt am Main, Germany. schnorr@cs.uni-frankfurt.de

integer transformation matrix of the local LLL-reduction. For this we carefully prepare the local matrix R_l of B_{l-1}, B_l prior to local LLL-reduction.

In practice, **segment sLLL** is much faster than previous codes for LLL-reduction. It performs well beyond dimension 1000 and provides lattice bases that are in practice of similar quality as LLL-reduced bases. For dimension 1000 and basis vectors with 400 bit integer coordinates, the new LLL-reduction takes only about 10 hours on a 800 MHz PC. It is the first code for LLL-reduction that performs well beyond dimension 350.

In this paper we focus on practical aspects related to *fpa*. A rigorous analysis of segment LLL-reduction in the model of integer arithmetic is in the companion paper [KS01]. For general background on *fpa* and Householder transformation see [LH95].

A.2 Stability Properties of Householder Orthogonalization.

For an introduction of LLL-reduced lattice bases and of our notation on lattices we refer to Section 2 of the companion paper [KS01]. For easy reference we recall Definition 1 and Theorem 1 from [KS01]. We let $\delta \in]\frac{1}{4}, 1]$ and $\alpha = 1/(\delta - \frac{1}{4})$.

C.2.1 Definition *An ordered basis $b_1, \ldots, b_n \in \mathbf{Z}^d$ of the lattice L is* LLL-reduced *with $\delta \in]\frac{1}{4}, 1]$ if it has properties* **1.** **2.***:*

1. $|\mu_{j,i}| \leq 1/2 \qquad$ *for* $1 \leq i < j \leq n$,

2. $\delta \, \|\widehat{b}_i\|^2 \;\leq\; \mu_{i+1,i}^2 \, \|\widehat{b}_i\|^2 + \|\widehat{b}_{i+1}\|^2 \qquad$ *for* $i = 1, \ldots, n-1$.

C.2.2 Theorem *A basis $b_1, \ldots, b_n$ of lattice L that is* LLL*-reduced with δ satisfies*:

1. $\|b_i\|^2 \leq \alpha^{n-1} \, \lambda_i^2$ *and* $\|b_1\|^2 \leq \alpha^{i-1} \, \|\widehat{b}_i\|^2 \qquad$ *for* $i = 1, \ldots, n$,

2. $\|b_1\|^2 \leq \alpha^{\frac{n-1}{2}} (\det L)^{\frac{2}{n}}$ *and* $\|\widehat{b}_n\|^2 \geq \alpha^{-\frac{n-1}{2}} (\det L)^{\frac{2}{n}}$.

Accuracy of Householder reflexions. Consider the QR-factorization $B = QR$ of the basis matrix $B = [b_1, \ldots, b_n] \in \mathbf{Z}^{d \times n}$, where $Q \in \mathbf{R}^{d \times d}$ is an orthogonal matrix and $R = [r_{i,j}] = [\mathbf{r}_1, ..., \mathbf{r}_n] \in \mathbf{R}^{d \times n}$ is an upper triangular matrix, $r_{i,j} = 0$ for $i > j$. We have $\mu_{j,i} = r_{i,j}/r_{i,i}$ and $|r_{i,i}| = \|\widehat{b}_i\|$. The vector $\mathbf{r}_l$ is the orthogonal transform of b_l.

Consider the process of orthogonalization of a basis matrix $B = [b_1, ..., b_n]$. In ideal arithmetic we get the QR-factorization $B = QR$ by a sequence of Householder transformations

$$C_1 := B, \; C_{j+1} := Q_j C_j \quad \text{for } j = 1, \ldots, n,$$

where $Q_j \in \mathbf{R}^{d \times d}$ is an orthogonal matrix — an Householder reflexion — that produces zeros in positions $j+1$ through d of column j of $Q_j C_j$. Thus $C_{j+1} \in \mathbf{R}^{d \times n}$ is upper triangular in the first j columns. Finally, $R = C_{n+1}, \; Q = Q_n \cdots Q_1$.

In actual computation, however, we use floating point operations

$$\bar{C}_1 = fl(B), \; \bar{C}_{j+1} := fl(\bar{Q}_j \bar{C}_j) \quad \text{for } j = 1, \ldots, n.$$

We assume the standard *fpa*-model of Wilkinson, see [LH95, p. 85] for details. Let $0 < \eta \ll 1$ be the relative precision — each floating point operation induces a normalized relative error bounded in magnitude by η. [3] In this model, it has been shown [LH 95, p. 87, formula (15.38)] that

$$\|\bar{C}_{j+1} - Q_j \cdots Q_1 B\|_F \leq (6d - 3j + 40) j \eta \, \|B\|_F + O(\eta^2), \tag{C.1}$$

[3] standard double length, wired *fpa* has 53 precision bits, $\eta = 2^{-53}$.

where $\|A\|_F = (\sum_{i,j} a_{i,j}^2)^{\frac{1}{2}}$ denotes the FROBENIUS NORM of the matrix $A = [a_{i,j}]$. In the following we neglect the low order term $O(\eta^2)$. Thus, in actual computation we get $\bar{R} = [\bar{r}_{i,j}] = \bar{C}_{n+1}$ satisfying for $n \geq 14$.

$$\|\bar{R} - R\|_F \leq 6dn\eta \, \|B\|_F. \tag{C.2}$$

It follows that the approximate Gram-Schmidt coefficients $\bar{\mu}_{j,i} = \bar{r}_{i,j}/\bar{r}_{i,i}$ satisfy for $i < j$

$$|\bar{\mu}_{j,i} - \mu_{j,i}| \leq 6dj\eta \, \|b_1, \ldots, b_j\|_F / (|r_{i,i}|(1-\varepsilon)) \tag{C.3}$$

provided that $|\bar{r}_{i,i} - r_{i,i}| \leq \varepsilon |r_{i,i}|$. Therefore, we get from inequality (C.2) the

C.2.3 Lemma *Inequality* (C.3) *holds provided that*

$$6dj\eta \, \|b_1, \ldots, b_j\|_F \leq \varepsilon |r_{i,i}|. \tag{C.4}$$

Instability of Householder QR-factorization for dimension 400. Inequalities (C.1), (C.2), (C.3) are rather sharp. To combine Householder transformation and size-reduction we need accurate coefficients $\mu_{j,i}$. Condition (C.4) characterizes the stability of Householder transformation — stability requires that $6dn\eta \, \|B\|_F < \min_i |r_{i,i}|$. On the other hand, Theorem C.2.4 shows that random LLL-reduced bases on the average satisfy $\|b_1\| \approx 1.1^{n-1} \|\widehat{b}_n\|$. For dimension $n = 400$ and $\eta = 2^{-53}$ this yields $6dn\eta \, \|b_1\|_F \approx 6dn\eta 1.1^{n-1} \|\widehat{b}_n\| \approx 4 \cdot 10^6 \|\widehat{b}_n\|$. Inequality (C.4) is grossly violated. Therefore, Householder transformation of LLL-reduced bases is necessarily unstable in dimension 400 for *fpa* with 53 precision bits.

C.2.4 Theorem *Consider a random LLL-reduced basis with random coefficients* $\mu_{i+1,i} \in_R [-\frac{1}{2}, \frac{1}{2}]$ *for* $i = 1, ..., n-1$, *and let the inequalities* **2.** *of Definition C.2.1 be tight. Then* $\|b_1\| \approx 1.1^{n-1} \|\widehat{b}_n\|$ *holds on the average.*

Proof of C.2.4:

While $(\delta - \mu_{i+1,i}^2) \|\widehat{b}_i\|^2 \leq \|\widehat{b}_{i+1}\|^2$ holds for LLL-reduced bases, the converse $(\delta - \varepsilon^2) \|\widehat{b}_i\|^2 \geq \|\widehat{b}_{i+1}\|^2$ holds provided that $\delta \|\widehat{b}_i\|^2 = \|\widehat{b}_{i+1}\|^2 + \mu_{i+1,i}^2 \|\widehat{b}_i\|^2$ and $|\mu_{i+1,i}| \leq \varepsilon$. The inequality $|\mu_{i+1,i}| \leq \varepsilon$ holds with probability 2ε for random $\mu_{i+1,i} \in_R [-\frac{1}{2}, \frac{1}{2}]$. As $\int_{-\frac{1}{2}}^{\frac{1}{2}} 2\varepsilon^2 d\varepsilon = \frac{1}{6}$ and $\sqrt{\delta - \frac{1}{6}} \approx 1.1^{-1}$ holds for $\delta \approx 1$ we see that $\|b_1\| \approx 1.1^{n-1} \|\widehat{b}_n\|$ holds on the average.

A.3 The Scaled Basis Matrix.

Scaling is a well known method for improving the stability of *fpa*. We associate with an integer lattice basis $b_1, ..., b_n$ a scaled basis $b_1^s, ..., b_n^s$ that has orthogonal vectors of nearly equal length.

C.3.1 Definition *Let* $b_1, \ldots, b_n$ *be a basis of an integer lattice* L. *We call* $b_1^s, ..., b_n^s \in L$ *an associated* scaled basis *with* scaling factors $2^{e_1}, ..., 2^{e_n}$ — $e_1, ..., e_n \in \mathbf{N}$ — *if* $b_1^s, ..., b_n^s$ *form a size-reduced basis of a sublattice of* L *satisfying*

$$\|b_1^s\| \leq 2^{e_i} \|\widehat{b}_i\| = \|\widehat{b}_i^s\| < 2\|b_1^s\| \quad \text{for } i = 1, ..., n. \tag{C.5}$$

We show in Theorem C.3.4 that a given scaled basis yields accurate Gram-Schmidt coefficients by Householder reflexions in fpa. In Section C.4 we show how to produce an associated scaled basis efficiently in fpa. In Sections C.5 and C.7 we use an associated scaled basis to guide LLL-reduction and segment LLL-reduction.

We can easily transform a basis $b_1, ..., b_n$ into an associated scaled basis $b_1^s, ..., b_n^s$ using exact arithmetic — first scale then size-reduce:

1. *scaling.* $e_1 := \lfloor \log_2(\max_j \|\widehat{b}_j\|/\|b_1\|)\rfloor$, $b_1^s := 2^{e_1}\, b_1$,

$$\textbf{for } i = 2, ..., n \textbf{ do } \; e_i := \max(0, \lceil \log_2 \|b_1^s\|/\|\widehat{b}_i\|\rceil), \;\; b_i^s := 2^{e_i} b_i$$

2. *size-reduction.* **for** $j = 2, ..., n$ **for** $i = j-1, ..., 1$ **do** $b_j^s := b_j^s - \lceil \frac{\langle b_j^s, \widehat{b}_i^s\rangle}{\|\widehat{b}_i^s\|^2} \rfloor \; b_i^s$. [4]

New Notation. For the remaining of the paper we let the column vector $\mathbf{r}_\nu$ of the R-matrix be related to the scaled vector b_ν^s rather than to the original vector b_ν.

The size of scaled Gram-Schmidt coefficients. The coefficients $\mu_{\nu,j}^s$ of an arbitrary, unscaled vector $b_\nu = \widehat{b}_\nu + \sum_{j=1}^{\nu-1} \mu_{\nu,j}^s \widehat{b}_j^s$ are uniformly bounded:

C.3.2 Corollary $|\mu_{\nu,j}^s| \;=\; \frac{|\langle b_\nu, \widehat{b}_j^s\rangle|}{\|\widehat{b}_j^s\|^2} \;\le\; \frac{\|b_\nu\|}{\|\widehat{b}_j^s\|} \overset{(C.5)}{\le} 2 \cdot \frac{\|b_\nu\|}{\|b_1^s\|}$.

In contrast, LLL-reduced bases $b_1, \ldots, b_{\nu-1}$ satisfy $b_\nu = \sum_{j=1}^{\nu} \mu_{\nu,j} \widehat{b}_j$ with $|\mu_{\nu,j}|^2 \le \frac{\|b_\nu\|^2}{\|b_1\|^2}\, \alpha^{j-1}$. The coefficient $\mu_{\nu,\nu-1}$ that enters first into size-reduction of b_ν tends to be very large due to the factor $\alpha^{\nu-1}$. A small relative error of $\mu_{\nu,\nu-1}$ confuses the size-reduction of b_ν.

C.3.3 Corollary *The basis $b_1^s, \ldots, b_n^s$ satisfies $\|b_j^s\| \le \sqrt{j+3}\, \|b_1^s\|$ for $j = 1, \ldots, n$.*

Proof of C.3.3:

$$\begin{aligned} \|b_j^s\|^2 &= \|\widehat{b}_j^s\|^2 + \textstyle\sum_{i=1}^{j-1} \mu_{j,i}^2 \|\widehat{b}_i^s\|^2 \\ &\le \|\widehat{b}_j^s\|^2 + (j-1)/4 \max_{i<j}(\|\widehat{b}_i^s\|)^2 \\ &\overset{(C.5)}{\le} 4\|b_1^s\|^2 + (j-1)\,\|b_1^s\|^2 = (j+3)\|\widehat{b}_1^s\|^2. \end{aligned}$$

❏

Next we study for a given scaled basis the accuracy of the approximate coefficients $\bar{\mu}_{j,i}^s$, computed in fpa by Householder reflexions.

C.3.4 Theorem *The approximate $\bar{\mu}_{j,i}^s$ of $b_1^s, \ldots, b_n^s$ satisfy $|\bar{\mu}_{j,i}^s - \mu_{j,i}^s| \le \varepsilon/(1-\varepsilon)$ for $\varepsilon = 6dj^2\eta$.*

Proof of C.3.4:

By scaling and size-reduction we have for $j \ne 2$: [5]

$$\|b_j^s\|^2 \le \frac{1}{4}\sum_{i=1}^{j-1} \|\widehat{b}_i^s\|^2 + \|\widehat{b}_j^s\|^2 \le \frac{j+3}{4} \max_{i\le j}\|\widehat{b}_i^s\|^2 \overset{(C.5)}{\le} j \min_{i\le j}\|\widehat{b}_i^s\|^2.$$

Hence $\|b_1^s, \ldots, b_j^s\|_F = (\sum_{i=1}^{j} \|b_i^s\|^2)^{1/2} \le \sqrt{j} \max_{i\le j}\|b_i^s\| \le j\, \|\widehat{b}_i^s\|$, and thus

$$\|b_1^s, \ldots, b_j^s\|_F / \|\widehat{b}_i^s\| \le j \qquad \text{for } i = 1, ..., j. \tag{C.6}$$

[4] Let $\lceil r \rfloor = \lceil r - \frac{1}{2} \rceil$ be the nearest integer to the real number r.

[5] In the following we neglect the exception $j = 2$.

Hence, Inequality (C.4) holds for $\varepsilon = 6dj^2\eta$. By Lemma C.2.3 Inequality (C.3) holds for that ε and thus $|\bar{\mu}^s_{j,i} - \mu^s_{j,i}| \overset{(3)}{\leq} 6dj\eta \, \|b^s_1, \ldots, b^s_j\|_F/(|r_{i,i}|(1-\varepsilon)) \overset{(C.6)}{\leq} 6dj^2\eta/(1-\varepsilon) = \varepsilon/(1-\varepsilon)$.

❏

Stability up to dimension 2^{16}. Consider Theorem C.3.4 in the case $j \leq n = d = 2^{16}$ and $\eta = 2^{-53}$. Then we have $\varepsilon \leq 6n^3\eta \leq 0.19$ and $\varepsilon/(1-\varepsilon) \leq 0.24$. Therefore, Householder reflexions yield up to dimension 2^{16} sufficiently accurate Gram-Schmidt coefficients for the basis $b^s_1, \ldots, b^s_n$.

A.4 Orthogonalization via Scaling and Size-Reduction.

Suppose we are given $b^s_1, \ldots, b^s_{\nu-1}, b_\nu$ and we want to produce a scaled vector b^s_ν. At that point the $\mu^s_{j,i}$ of $b^s_1, \ldots, b^s_{\nu-1}$ are given with high accuracy. We iteratively transform b_ν into b^s_ν using better and better approximations of the $\mu^s_{\nu,i}$. The procedure **HRS** (Householder, Reduction, Scaling) iterates the following steps

1. the first $\nu - 1$ Householder transformations $b_\nu \mapsto \bar{Q}_{\nu-1} \cdots \bar{Q}_1 b_\nu$,

2. size-reduction of b_ν against $b^s_1, \ldots, b^s_{\nu-1}$,

3. scaling of b_ν to b^s_ν.

Steps **1. 2.** must be iterated as the size of b_ν is by Inequality (C.2) crucial for the accuracy of Householder transformation. As the scaling increases the size of b_ν we scale in stages, repeating **1. 2.** after each stage of scaling. Let $Q_j = I_d - 2v_j v_j^\top \|v_j\|^{-2}$, where $I_d \in \mathbf{Z}^{d\times d}$ is the identity matrix and $v_j \in \mathbf{R}^d$ is the Householder vector associated with Q_j. Note that $x \mapsto Q_j x$ reflects x at the hyperplane that is orthogonal to v_j: $Q_j v_j = -v_j$, $Q_j u = u$ for $u \perp v_j$. **HRS** is given for input the Householder vectors $v_1, ..., v_{\nu-1} \in \mathbf{R}^d$, the first $\nu - 1$ columns $\bar{\mathbf{r}}_1, ..., \bar{\mathbf{r}}_{\nu-1}$ of the matrix $\bar{C}_\nu = \bar{Q}_{\nu-1} \cdots \bar{Q}_1 \bar{C}_1$ and the computed scaling exponents $\bar{e}_1, ..., \bar{e}_{\nu-1}$. The operations on $b^s_1, ..., b^s_{\nu-1}$ are in exact integer arithmetic, the other operations are in fpa. Taking fpa-errors into account we relaxe size-reduction of b^s_ν to the relaxed condition $|\mu^s_{\nu,j}| \leq 0.52$.

Supressing backward rescaling. Upon entry of **HRS**(ν), the scaled vectors b^s_1, ..., $b^s_{\nu-1}$ are given while $b^s_\nu, ..., b^s_n$ are unknown. At this stage the scaling factors $2^{e_1}, ..., 2^{e_{\nu-1}}$ correspond to the subbasis $b_1, ..., b_{\nu-1}$. If $\|\widehat{b}_\nu\| > \|\widehat{b^s_1}\|$ we would need to rescale b^s_i by increasing $e_i := e_i + \bar{e}$ and $b^s_i := 2^{\bar{e}} b^s_i$ for $\bar{e} := \lfloor \log_2(\|\widehat{b}_\nu\|/\|\widehat{b^s_1}\|) \rfloor$ and $i = 1, ..., \nu - 1$. We suppress this backward rescaling. It is sufficient to store $\bar{e}$ and to do all subsequent size-reductions against $2^{\bar{e}} b^s_i$ rather than against b^s_i, i.e., we replace subsequent reduction steps $b := b - \lceil \frac{\langle b, \widehat{b^s_i} \rangle}{\langle \widehat{b^s_i}, \widehat{b^s_i} \rangle} \rfloor b^s_i$ by the steps $b := b - 2^{\bar{e}} \lceil \frac{2^{-\bar{e}} \langle b, \widehat{b^s_i} \rangle}{\langle \widehat{b^s_i}, \widehat{b^s_i} \rangle} \rfloor b^s_i$. For simplicity, the program **HRS**(ν) does not include the steps required in case that $\|\widehat{b}_\nu\| > \|\widehat{b^s_1}\|$.

How HRS works. According to (C.5) the input vectors $b^s_1, ..., b^s_{\nu-1}$ satisfy $|r_{1,1}| \leq |r_{i,i}| < 2|r_{1,1}|$ for $i = 1, ..., \nu - 1$. Let $6d\nu^2\eta < \frac{1}{2}$ so that Theorem C.3.4 holds for $\varepsilon = \frac{1}{2}$. Then, by (C.2) and (C.6) we have that $|\bar{r}_{i,i} - r_{i,i}| \leq 6dj^2\eta |r_{i,i}|$ for $i = 1, ..., j$.

Reducing long $\|b_\nu\|$. By (C.2) the relative error of $\bar{r}_{i,\nu}/\|b_\nu\|$ in step **2.** is at most $6d\nu\eta$. The subsequent size-reduction reduces the large $\bar{r}_{i,\nu}$ — in the equation $\|b_\nu\|^2 = \sum_{i=1}^d r^2_{i,\nu}$ — to less than $|\bar{r}_{i,i}|$ in absolute value. Steps **2. 3. 4.** decrease the $(\sum_{i=1}^{\nu-1} \bar{r}^2_{i,\nu})^{\frac{1}{2}}$–part until the inequality $(\sum_{i=1}^{\nu-1} \bar{r}^2_{i,\nu})^{\frac{1}{2}} \leq \|b^s_1, ..., b^s_{\nu-1}\|_F$ holds.

Algorithm C.1 HRS(ν) (*Householder transformation, Reduction and Scaling of* b_ν)

Input: $b_1^s,\ldots,b_{\nu-1}^s \in \mathbf{Z}^d,\ \bar{\mathbf{r}}_1,...,\bar{\mathbf{r}}_{\nu-1},\ v_1,\ldots,v_{\nu-1} \in \mathbf{Q}^d,\ \bar{e}_1,...,\bar{e}_{\nu-1} \in \mathbf{Z}$

Output: b_ν^s (size-reduced against $b_1^s,...,b_{\nu-1}^s$), $v_\nu, \bar{\mathbf{r}}_\nu, \bar{e}_\nu$

1. $\bar{e}_\nu := 0,\ b_\nu^s := b_\nu$

2. $\bar{\mathbf{r}}_\nu := \bar{Q}_{\nu-1}\cdots\bar{Q}_1 \cdot b_\nu^s$

3. *size-reduce* b_ν^s *against* $b_1^s,...,b_{\nu-1}^s$
for $i = \nu-1,\ldots,1$ **do**
$\bar{\mu}_{\nu,i}^s := \bar{r}_{i,\nu} \ / \ \bar{r}_{i,i}$
if $|\bar{\mu}_{\nu,i}^s| \geq 0.51$ **then** $b_\nu^s := b_\nu^s - \lceil \bar{\mu}_{\nu,i}^s \rfloor b_i^s,\ \bar{\mathbf{r}}_\nu := \bar{\mathbf{r}}_\nu - \lceil \bar{\mu}_{\nu,i}^s \rfloor \bar{\mathbf{r}}_i$

4. **if** $\exists i : |\bar{\mu}_{\nu,i}^s| \geq 2^{10}$ **then go to 2.**

5. $\tau := (\ \sum_{i=\nu}^d \bar{r}_{i,\nu}^2\ )^{\frac{1}{2}},\ \ \widetilde{e} := \min(\lceil \log_2(\|b_1^s\|/\tau)\rceil, 30)$

6. **if** $\tilde{e} > 0$ **then** $\bar{e}_\nu := \bar{e}_\nu + \widetilde{e},\ \ b_\nu^s := 2^{\widetilde{e}} b_\nu^s$ **go to 2.**

7. $\bar{\mathbf{r}}_\nu := \bar{Q}_{\nu-1}\cdots\bar{Q}_1 b_\nu^s,\ \ \sigma := \mathrm{sig}(\bar{r}_{\nu,\nu}),\quad \tau := (\ \sum_{i=\nu}^d \bar{r}_{i,\nu}^2\ )^{\frac{1}{2}}$
$v_\nu := (0,\ldots,0,\bar{r}_{\nu,\nu} + \sigma\tau,\ \bar{r}_{\nu+1,\nu},\ldots,\bar{r}_{d,\nu})^\top$
$\bar{\mathbf{r}}_\nu := (\bar{r}_{1,\nu},\ldots,\bar{r}_{\nu-1,\nu},-\sigma\tau,0,\ldots,0)^\top$ *end*

Upon entry of step **3.** the coefficients $\mu_{\nu,i}^s$ are uniformly bounded as $|\mu_{\nu,j}^s| \leq 2\cdot\frac{\|b_\nu\|}{\|b_1^s\|}$, see Corollary C.3.2. Size-reduction of Step **3.** can temporarily increase the coefficients $|\mu_{\nu,j}^s|$ up to a factor $(3/2)^{\nu-j-1}$ in worst case. This could possibly make the size-reduction of step **3.** unstable in worst case. No such instability has been reported in practice, see Figure C.1.

Scaling up small $\|\widehat{b}_\nu\|$. Let $\|b_\nu\| \leq \|b_1^s,...,b_{\nu-1}^s\|_F$ and $|r_{\nu,\nu}| \leq \eta\|b_\nu\|$. By (C.2) and (C.6) we have after step **2.** that $|\bar{r}_{\nu,\nu} - r_{\nu,\nu}| < 6d\nu\eta\,\|b_1^s,...,b_{\nu-1}^s\|_F \leq 6d\nu^2\eta\|b_1^s\|$. This yields a scaling factor $\tilde{e}$ for step **6.** so that $2^{\tilde{e}} \geq (6d\nu^2\eta)^{-1} > 1$. Therefore, steps **2.** to **6.** increase $|\bar{r}_{\nu,\nu}|$, $\|b_\nu^s\|$ until $|\bar{r}_{1,1}| \leq |\bar{r}_{\nu,\nu}| \approx \|b_\nu^s\| < 2|\bar{r}_{1,1}|$.

C.4.1 Corollary **HRS**(ν) *produces a scaled vector* b_ν^s *satisfying* $|\mu_{\nu,i}^s| \leq 0.52$ *for* $i = 1,...,\nu-1$ *provided that size-reduction of Step* **3.** *remains stable.*

Speeding up **HRS**. To speed up **HRS** replace the rounded values $\lceil \bar{\mu}_{\nu,i}^s \rfloor$ in step **3.** by single precision integers consisting of the leading bits of $\lceil \bar{\mu}_{\nu,i}^s \rfloor$. This way **HRS** becomes very fast.

The number of arithmetic steps. A matrix-vector multiplication $b_\nu \mapsto \bar{Q}_1 b_\nu = b_\nu - \frac{2v_1 v_1^\top b_\nu}{\|v_1\|^2}$ requires $2d$ multiplications and one division—as usual we neglect the additions /subtractions. Thus, we get $\bar{\mathbf{r}}_\nu := \bar{Q}_{\nu-1}\cdots\bar{Q}_1 b_\nu$ using $2\nu d$ multiplications/divisions. Size-reduction of step **3.** requires νd multiplications in exact arithmetic. In total, one *round* of steps **2. 3. 4.** requires $2\nu d$ multiplications/divisions in fpa and νd exact multiplications of long integers.

The number of rounds. In practice each round of steps **2. 3. 4.** either decreases $(\sum_{i=1}^{\nu-1} r_{i,\nu}^2)^{\frac{1}{2}}$ by the factor $6d\nu^2\eta$ or increases $\|\widehat{b}_\nu^s\|$ by the scaling factor $2^{\bar{e}_\nu} \geq (6d\nu^2\eta)^{-1} > 1$ or does a combination of both, see the explanations for **HRS**. In practice **HRS**(ν) requires $\log_2(\|b_\nu\|/\|\widehat{b}_\nu\|)/|\log_2(6d\nu^2\eta)|$ rounds.

Practical Performance of **HRS.** Consider a random public-key basis of the GGH-cryptosystem [GGH97] of dimension 400. Figure C.1 shows how **HRS**(400) decreases the coefficients $|\mu_{400,j}|$ of the last basis vector b_{400}. Using 53 bit fpa each of five rounds decreases $|\mu_{400,j}|$ by a factor

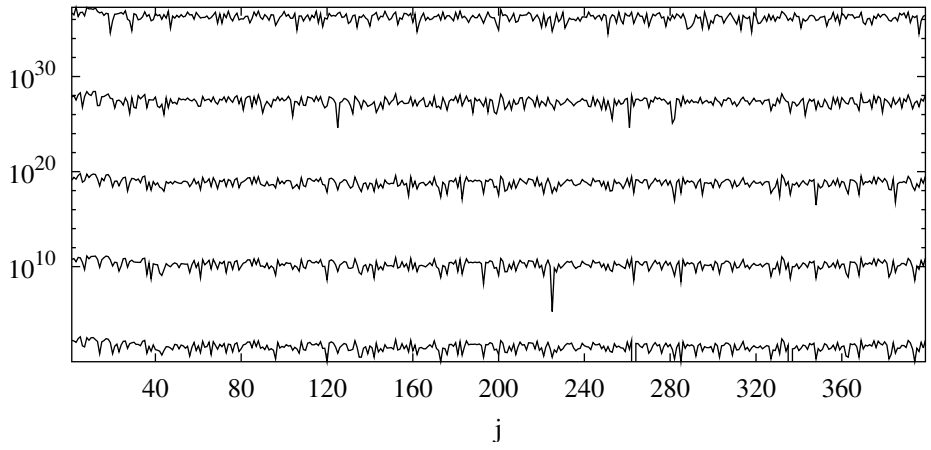


Figure C.1: Displayed are the values $|\mu_{400,j}|$ upon termination of each of five rounds. The $|\mu_{400,j}|$ are measured after step 2 of the next round — after new orthogonalization.

about $10^9 \approx 2^{30}$. After each round all coefficients $|\mu_{400,j}|$ are of nearly equal size because the preceding orthogonal vectors have been scaled to nearly equal length $\|\widehat{b}_j^s\|$. After each of the first 4 rounds however, $|\mu_{\nu,j}^s|$ increases by about a factor 10 as j decreases from 400 to 1. This is due to the temporary increase of $|\mu_{\nu,j}^s|$ by a factor $(3/2)^{\nu-j-1}$ in worst case. Figure C.1 shows that this temporary increase has little effect in practice.

Using 106 bit fpa — instead of 53 bit fpa — the five rounds of **HRS**(400) reduce to two rounds, and the running time of **HRS** reduces accordingly. Software implemented fpa with 106 precision bits makes the reduction clearly faster than the standard fpa with 53 precision bits.

A.5 Scaled LLL-Reduction.

We introduce *scaled* LLL-*reduced* lattice bases, and we present an algorithm **scaled LLL** for scaled LLL-reduction. This algorithm is a useful prelude to the more complicated scaled segment LLL-reduction of Section C.7.

C.5.1 Definition *We call a lattice basis* $b_1, \ldots, b_n \in \mathbf{Z}^d$ scaled LLL-reduced *if it has properties* **1.** **2.***:*

1. *There is an associated scaled basis* $b_1^s, ..., b_n^s$ *so that* $b_1, ..., b_n$ *is size-reduced against* $b_1^s, ..., b_n^s$*, i.e.,* $\left|\frac{\langle b_\nu, \widehat{b}_j^s\rangle}{\|\widehat{b}_j^s\|^2}\right| \le 0.52$ *for* $1 \le i < j \le n$,

2. $\delta\ \|\widehat{b}_i\|^2 \ \le\ \mu_{i+1,i}^2\ \|\widehat{b}_i\|^2 + \|\widehat{b}_{i+1}\|^2 \qquad$ *for* $i = 1, \ldots, n-1$.

We call a basis with property **1.** *scaled-reduced*. Importantly, the inequalities of Theorem C.2.2 still hold for scaled LLL-reduced lattice bases. The weaker size-reduction does not affect these inequalities. Scaled LLL-reduced bases are in practice as good as LLL-reduced bases.

Next we present an algorithm **scaled LLL** that transforms a lattice basis into a scaled LLL-reduced basis of the same lattice. **Scaled LLL** guides the LLL-steps on the original basis

by the orthogonalization of an associated scaled basis. So we keep and update two versions of the basis. LLL-transformations operate on the original basis. Size-reduction is done against the scaled basis. This form of LLL-reduction is quite stable as **HRS** yields an accurate QR-factorization of the scaled basis.

The procedure **HRS'**. Local LLL-reduction of two basis vectors $b_{\nu-1}, b_\nu$ is guided by the orthogonal vector of a modified scaled vector b^s_ν — with a scaling factor $2^{\bar{e}_\nu}$ that coincides with the scaling factor $2^{\bar{e}_{\nu-1}}$ of $b_{\nu-1}$. We get the modified b^s_ν by the following variant **HRS'** of **HRS**: Set in Step **1.** $\bar{e}_\nu := \bar{e}_{\nu-1}$ and skip Step **6.**. Moreover, we let **HRS'** perform in Step **3.** the same size-reduction steps $b^s_\nu := b^s_\nu - \lceil \bar{\mu}^s_{\nu,\nu-1} \rfloor b^s_{\nu-1}$, $b_\nu := b_\nu - \lceil \bar{\mu}^s_{\nu,\nu-1} \rfloor b_{\nu-1}$ on b^s_ν against $b^s_{\nu-1}$ and on b_ν against $b_{\nu-1}$. As a consequence we have

C.5.2 Lemma *The reduction coefficients of b_ν against $b_{\nu-1}$ and of b^s_ν against $b^s_{\nu-1}$ coincide in* **HRS'**(ν)*, and thus $\mu_{\nu,\nu-1} = \mu^s_{\nu,\nu-1}$. The output vector b_ν of* **HRS'**(ν) *is size-reduced against $b_{\nu-1}$. The coefficients $r_{i,j}$ for $\nu - 1 \le i, j \le \nu$ associated with $b_{\nu-1}, b_\nu$ are proportional to the coefficients associated with $b^s_{\nu-1}, b^s_\nu$ with proportionality factor $2^{\bar{e}_\nu}$.*

Algorithm C.2 Scaled LLL (*Algorithm for scaled* LLL*-reduction.*)

Input: $b_1, \ldots, b_n \in \mathbf{Z}^d$, δ

Output: $b_1, \ldots, b_n$ scaled LLL-reduced basis

1. $\nu := 1$

2. **while** $\nu \le n$ **do**

3. **if** $\nu = 1$ **then** **HRS**(1), $\nu := 2$

4. **HRS'**(ν)

5. **if** $\delta\, \bar{r}^2_{\nu-1,\nu-1} > \bar{r}^2_{\nu-1,\nu} + \bar{r}^2_{\nu,\nu}$
then swap $b_{\nu-1}, b_\nu$, $\nu := \nu - 1$
else **HRS**(ν), $\nu := \nu + 1$ *end*

Scaling does not affect *LLL*-exchanges. By Lemma C.5.2, the output vector b_ν of **HRS'**(ν) is size-reduced against $b_{\nu-1}$, i.e., $|\mu_{\nu,\nu-1}| \le 0.52$. Moreover, the decision about swapping $b_{\nu-1}, b_\nu$ in Step **5.** is the same as for the original LLL-algorithm — except for *fpa*-errors.

fpa-errors do not affect the LLL*-exchanges.* We see from $\|b^s_\nu\|^2 = \sum_{i=1}^d r^2_{i,\nu}$ and (C.2) that the relative error of $\bar{r}_{\nu-1,\nu}/\|b^s_\nu\| \approx \bar{r}_{\nu-1,\nu}/|\bar{r}_{\nu,\nu}|$ and $|r_{\nu,\nu}|/\|b^s_\nu\|$ is at most $6d\nu\eta$. On the other hand, $|\bar{r}_{\nu-1,\nu-1}| \ge \|b^s_1\|$ holds by scaling. Therefore, the decision about swapping $b_{\nu-1}, b_\nu$ in Step **5.** is correct for $6d\nu\eta \ll 1$.

A.6 Segment LLL-Reduction.

We summarize the concept of segment LLL-reduced bases of the companion paper, for full coverage see [KS01].

Segments and local coordinates. Let the basis $b_1, \ldots, b_n \in \mathbf{Z}^d$ have dimension $n = k \cdot m$ and the QR-factorization $[b_1, \ldots, b_n] = QR$. We partition the basis-matrix B into m *segments* $B_l = [b_{k(l-1)+1}, \ldots, b_{kl}]$ for $l = 1, ..., m$. Local reduction of two consecutive segments uses the coefficients of the submatrix $R_l := [r_{kl+i,kl+j}]_{-k<i,j\le k} \in \mathbf{R}^{2k\times 2k}$ of $R \in \mathbf{R}^{d\times n}$, corresponding

to two consecutive segments B_{l-1}, B_l. We want to do most of the LLL-exchanges and the corresponding size-reduction in local coordinates of some R_l. Extra global transformations are required after local LLL-reduction. In order to minimize these global costs we introduce *k-segment reduced bases*. We let $D(l) = \|\widehat{b}_{k(l-1)+1}\|^2 \cdots \|\widehat{b}_{kl}\|^2$ denote the *local Gramian determinant* of segment B_l. We have that $D_{kl} = D(1) \cdots D(l)$.

C.6.1 Definition *We call a basis $b_1, \ldots, b_n \in \mathbf{Z}^d, n = km$, k-segment* LLL-reduced *with $\delta \in]\frac{1}{4}, 1]$ if it is size-reduced and satisfies for $\alpha = 1/(\delta - \frac{1}{4})$:*

1. $\delta \|\widehat{b}_i\|^2 \leq \mu_{i+1,i}^2 \|\widehat{b}_i\|^2 + \|\widehat{b}_{i+1}\|^2$ *for* $i \neq 0 \mod k$,

2. $D(l) \leq (\alpha/\delta)^{k^2} D(l+1)$ *for* $l = 1, \ldots, m-1$,

3. $\delta^{k^2} \|\widehat{b}_{kl}\|^2 \leq \alpha \|\widehat{b}_{kl+1}\|^2$ *for* $l = 1, \ldots, m-1$.

C.6.2 Theorem [KS01] *Let $b_1, \ldots, b_n$ be a basis that is k-segment* LLL*-reduced with δ. Then we have for $i = 1, \ldots, n$:*

$$\delta^{2k^2+n-1} \|b_i\|^2 \leq \alpha^{n-1} \lambda_i^2 \quad \text{and} \quad \delta^{k^2+i-1} \|b_1\|^2 \leq \alpha^{i-1} \|\widehat{b}_i\|^2,$$

where $\lambda_1 \leq \cdots \leq \lambda_n$ are the successive minima of the lattice.

Algorithm for segment *LLL*-reduction. The algorithm **segment LLL** transforms a given basis into a k-segment reduced basis using exact integer arithmetic. It iterates local LLL-reduction of two segments $[B_{l-1}, B_l] = [b_{kl-k+1}, ..., b_{kl+k}]$ via

The procedure **loc-LLL**(l). Given the orthogonalization of a k-segment reduced basis $b_1, ..., b_{kl-k}$ the procedure **loc-LLL**(l) computes the orthogonalization and size-reduction of the segments B_{l-1}, B_l. In particular it provides the submatrix $R_l \in \mathbf{R}^{2k \times 2k}$ of $R \in \mathbf{R}^{d \times n}$ corresponding to the segments B_{l-1}, B_l. Thereafter it performs a local LLL–reduction of R_l and stores the LLL-transformation in the matrix $H \in \mathbf{Z}^{2k \times 2k}$. Finally, it transforms $[B_{l-1}, B_l]$ into the locally reduced segments $[B_{l-1}, B_l]H$ and size-reduces $[B_{l-1}, B_l]$ globally.

Algorithm C.3 Segment LLL

Input: $b_1, \ldots, b_n \in \mathbf{Z}^d, k, m, n = km, \ \delta$

Output: $b_1, \ldots, b_n$ k-segment LLL-reduced basis

1. $l := 1$

2. **while** $l \leq m-1$ **do**

loc-LLL(l)
if $l \neq 1$ **and**
($D(l-1) > (\alpha/\delta)^{k^2} D(l)$ **or** $\delta^{k^2} \|\widehat{b}_{k(l-1)}\|^2 > \alpha \|\widehat{b}_{k(l-1)+1}\|^2$)
then $l := l-1$ **else** $l := l+1$. *end*

C.6.3 Theorem [KS01] *For $k = \Theta(m) = \Theta(\sqrt{n})$* **segment LLL** *performs $O(nd \log_{1/\delta} M)$ arithmetic steps using integers of bit length $O(\log_2 M)$.*

A.7 Scaled Segment LLL-Reduction.

We combine the methods of Sections C.4 and C.6 to a stable algorithm for segment LLL-reduction. Size-reduction is done against an associated scaled basis. Segment LLL-reduction is guided by the orthogonal vectors of the scaled basis.

C.7.1 Definition *We call a basis* $b_1, \ldots, b_n \in \mathbf{Z}^d$ k-segment sLLL-reduced, *if it is* k*-segment* LLL*-reduced except that* $b_1, ..., b_n$ *is size-reduced in a weaker sense — it is size-reduced against an associated basis* $b_1^s, ..., b_n^s$ *with the properties* **1.** **2.***:*

1. *The first orthogonal vectors* $\widehat{b}_{kl-k+1}^s$ *of segments are nearly equally long:*

$$\|b_1^s\| \le 2^{e_{kl}} \|\widehat{b}_{kl-k+1}\| = \|\widehat{b}_{kl-k+1}^s\| < 2\|b_1^s\| \quad \text{for } l = 1, ..., m. \tag{C.7}$$

2. *There is a* uniform *scaling factor* $2^{e_{kl}}$ *for segment* B_l *so that the coefficients* $r_{kl-k+i,kl-k+j}$*, for* $1 \le i, j \le k$ *of the* R*-matrix corresponding to* $b_1, ..., b_n$ *and those corresponding to* $b_1^s, ..., b_n^s$ *are proportional with proportionality factor* $2^{e_{kl}}$*.*

The inequalities for k-segment LLL-reduced bases in Theorem C.6.2 also hold for k-segment sLLL-reduced bases. The weaker size-reduction is not important. In practice k-segment sLLL-reduced bases are nearly as good as LLL-reduced bases.

We sketch a procedure **loc-sLLL**(l) — for local scaled LLL-reduction — which replaces **loc-LLL**(l) within **segment LLL**. The algorithm **segment sLLL** iterates local scaled LLL-reduction of two consecutive segments $[B_{l-1}, B_l] = [b_{kl-k+1}, ..., b_{kl+k}]$.

Algorithm C.4 Segment sLLL

Input: $b_1, \ldots, b_n \in \mathbf{Z}^d, k, m, n = km,\ \delta$

Output: $b_1, \ldots, b_n$ k-segment and scaled reduced basis

1. $l := 1$

2. **while** $l \le m - 1$ **do**

loc-sLLL(l)

if $l \ne 1$ **and**

$(\ D(l-1) > (\alpha/\delta)^{k^2} D(l)$ **or** $\delta^{k^2} \|\widehat{b}_{kl}\|^2 > \alpha \|\widehat{b}_{kl+1}\|^2\ )$

then $l := l - 1$ **else** $l := l + 1$. *end*

The procedure **loc-sLLL**(l). Its inputs are a lattice basis, uniform scaling factors $2^{\bar{e}_{k\ell}}$ satisfying $b_{k\ell-j}^s = 2^{\bar{e}_{k\ell}} b_{k\ell-j}$ for $j = 0, ..., k-1$, $\ell = 1, ..., l-1$. The Householder vectors $v_1, ..., v_{k(l-1)}$ of $b_1, ..., b_{k(l-1)}$ and the matrix $R \in \mathbf{R}^{d \times k(l-1)}$ for $[b_1^s, ..., b_{k(l-1)}^s]$ are also given. Local LLL-reduction of B_{l-1}, B_l is done in fpa via the local matrix $R_l \in \mathbf{R}^{2k \times 2k}$.

The procedure **loc-sLLL**(l)

- computes a uniform scaling factor $2^{\bar{e}_{kl}}$ for the two segments $[B_{l-1}, B_l]$
- computes the local matrix R_l of $2^{\bar{e}_{kl}}[B_{l-1}, B_l]$ via **HRS'**
- performs a local LLL-reduction on $[B_{l-1}, B_l]$ using R_l
- stores the transformation in the matrix $H \in \mathbf{Z}^{2k \times 2k}$.
- Upon termination it transforms $[B_{l-1}, B_l]$ into $[B_{l-1}, B_l]H$ in exact arithmetic.

Restarting **loc-sLLL**(l). Whenever $\|H\|_\infty$ surpasses the threshold 2^{15} the procedure **loc-sLLL**(l) is restarted with the transformed segments $[B_{l-1}, B_l]H$. This is necessary as the norm $\|H\|_\infty$ directly translates into additional fpa-errors. As the number of restarts is crucial for the running time we carefully prepare R_l as to prevent an early restart. We show below how to predict the size of the matrix $H \in \mathbf{Z}^{2k \times 2k}$ occuring in the subsequent local LLL-reduction of B_{l-1}, B_l. We slash R_l so that $\|H\|_\infty \le 2^{15}$ holds on the average. The threshold 2^{15} is for wired 53-bit fpa, for 106-bit fpa we increase the threshold accordingly.

Computing the matrix R_l and $2^{\bar{e}_{kl}}$. First compute via **HRS** the uniform scaling factor $2^{\bar{e}_{kl}}$ and the first vector $\mathbf{r}_{kl-k+1}$ of B_{l-1} so that $\|b_1^s\| \le 2^{e_{kl}}\|\widehat{b}_{kl-k+1}\| = \|\widehat{b}^s_{kl-k+1}\| < 2\|b_1^s\|$. With the same scaling factor $2^{\bar{e}_{kl}}$ compute $\mathbf{r}_h$ via **HRS'** for $h = kl-k+2, ..., kl+k$. For local LLL-reduction we have to size-reduce b_h against $b_{kl-k+1}, \dots b_{h-1}$. We generalize Step **3.** of **HRS'** accordingly: perform the same size-reduction steps $b_h^s := b_h^s - \lceil \bar{\mu}^s_{h,j} \rfloor b_j^s$, $b_h := b_h - \lceil \bar{\mu}^s_{h,j} \rfloor b_j$ on b_h^s against b_j^s and on b_h against b_j for $j = kl-k+1, \dots h-1$. This implies that the local matrix R_l of segments B_{l-1}, B_l corresponding to $b_1, ..., b_n$ and the R_l corresponding to $b_1^s, ..., b_n^s$ are proportional with the factor $2^{\bar{e}_{kl}}$.

The heuristic for slashing R_l. We let $r^l_{i,j} = r_{kl-k+i,kl-k+j}$ denote the coefficients of the local matrix R_l. Large values $|r^l_{1,1}/r^l_{j,j}|$ have a double negative effect on the stability of **loc-sLLL**(l). By Lemma C.2.3 the orthogonalization of b_j gets inaccurate. Moreover, $\|H\|_\infty$ will be large due to Lemma 2 [KS01]. A detailed argument shows that $\|H\|_\infty$ is expected to be less than 2^{15} if $\max_j |r^l_{1,1}/r^l_{j,j}| \le 2^{15}$. This suggests to *slash* R_l so that $\max_j |r^l_{1,1}/r^l_{j,j}| \le 2^{15}$.

Slashing the matrix R_l. Slash all values $|r^l_{j,j}|$ — satisfying $|r^l_{1,1}/r^l_{j,j}| < 2^{-15}$ — to $r^l_{j,j} := |r^l_{1,1}|2^{-15}$. Moreover, set $r^l_{h,i} := 0$ for $h = j, ..., 2k$ and $h < i \le 2k$.

$$R_l := \begin{bmatrix} r^l_{1,1} & \cdots & r^l_{1,j-1} & r^l_{1,j} & \cdots & r^l_{1,k} \\ & \ddots & \vdots & \vdots & & \vdots \\ 0 & & r^l_{j-1,j-1} & r^l_{j-1,j} & \cdots & r^l_{j-1,k} \\ 0 & \cdots & 0 & |r^l_{1,1}|2^{-15} & & 0 \\ \vdots & & \vdots & & \ddots & \\ 0 & \cdots & 0 & 0 & & |r^l_{1,1}|2^{-15} \end{bmatrix}$$

We locally LLL-reduce the slashed matrix R_l, and afterwards we transform $[B_{l-1}, B_l]$ into $[B_{l-1}, B_l]H$. If either $\|H\|_\infty > 2^{15}$ or if the slashing did effectively change R_l we restart the local reduction with the transformed segments $[B_{l-1}, B_l]H$. Note that a restart of **loc-sLLL**(l) also adjusts the uniform scaling factor $2^{\bar{e}_{kl}}$. This method of correction works very well for segment size $k \le 100$.

A.8 Performance of Segment sLLL.

Consider a sample of [GGH]-bases of dimension $n = \nu \cdot 100$ for $\nu = 1, ..., 8$. The vectors of the input basis consist of integers with $n/2$ bits. The algorithm uses 106 bit fpa and $\delta = 0.99$ respectively $\delta = 0.999$. All tests have been performed on 350 MHz PC's, the code of **segment sLLL** uses the [NTL00] computer algebra library. We present the segment size and the average bit length of the reduced basis vectors in Table C.1.

The running time increases considerably as δ approaches 1. The size of the reduced basis decreases accordingly. Figure C.2 shows the time of **segment sLLL** for [GGH]-bases.

Acknowledgement. We thank Bartol Filipović for his help in producing and measuring the code of the new reduction algorithm.

Lattice dimension	$\delta = 0.99$		$\delta = 0.999$	
	segment size	l^{av}	*segment size*	l^{av}
100	25	7.9	25	5.2
200	40	13.8	40	9.4
300	45	23.0	45	14.7
400	50	30.5	50	17.7
500	50	34.3	56	22.6
600	60	38.8	60	27.9
700	60	40.9	65	34.3
800	70	46.6	70	37.0

Table C.1: segment sLLL reduced bases. l^{av} = average bit length of the output integers.

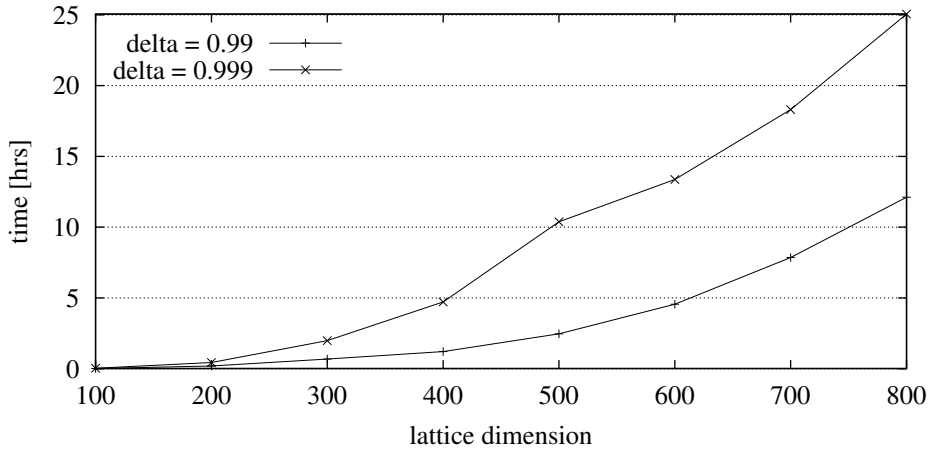


Figure C.2: Running time of **Segment sLLL** using $\delta = 0.99$ resp. $\delta = 0.999$.

References

[GGH] *O. Goldreich, S. Goldwasser, and S. Halevi*, Public-key cryptosystems from lattice reduction problems. Proc. Crypto'97, LNCS 1294, Springer-Verlag, pp. 112–131, 1997.

[KS01] *H. Koy and C.P. Schnorr*, Segment LLL-Reduction of Lattice Bases. Proceedings CaLC 2001.

[LLL82] *A. K. Lenstra, H. W. Lenstra, and L. Lovász*, Factoring polynomials with rational coefficients, *Math. Ann.* **261**, pp. 515–534, 1982.

[LH95] *C.L. Lawson and R.J. Hanson*, Solving Least Square Problems, SIAM, Philadelphia, 1995.

[NTL00] NTL homepage: http://www.shoup.net/ntl, 2000.

[RS96] *C. Rössner and C.P. Schnorr*, An optimal stable continued fraction algorithm for arbitrary dimension. 5.-th IPCO, LNCS **1084**, pp. 31–43, Springer-Verlag, 1996.

[S87] *C.P. Schnorr*, A hierarchy of polynomial time lattice basis reduction algorithms, *Theoretical Computer Science* **53**, pp. 201-224, 1987.

[S88] *C.P. Schnorr*, A more efficient algorithm for lattice basis reduction, *J. Algorithms* **9**, pp. 47–62, 1988.

[SE91] *C.P. Schnorr and M. Euchner*, Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems, *Proc. Fundamentals of Computation Theory '91*, L. Budach, ed., LNCS **529**, Springer-Verlag, pp. 68-85, 1991. (Complete paper in Mathematical Programming Studies **66A**, No 2 , pp. 181–199, 1994.)

[Sc84] *A. Schönhage*, Factorization of univariate integer polynomials by diophantine approximation and improved lattice basis reduction algorithm, *Proc. 11-th Coll. Automata, Languages and Programming, Antwerpen 1984*, LNCS **172**, Springer-Verlag, pp. 436–447, 1984.

Symbolverzeichnis

$\mathtt{A}_2 \leftarrow \mathtt{A}_1$	Zuweisung: Werte $\mathtt{A}_1$ aus und Weise das Ergebnis $\mathtt{A}_2$ zu
$[a, b]$	Intervall der Zahlen von inklusive a bis inklusive b
$[a, b)$	Intervall der Zahlen von inklusive a bis exklusive b
$(a, b]$	Intervall der Zahlen von exklusive a bis inklusive b
(a, b)	Intervall der Zahlen von exklusive a bis exklusive b
$\langle \cdot , \cdot \rangle$	Skalarprodukt (inneres Produkt) auf einem reellen Vektorraum 12
$\perp$	Orthogonalität von Vektoren 12
$U^\perp$	Orthogonales Komplement von U 13
$\lvert \cdot \rvert$	Absolut-Betrag einer reellen Zahl
$\lVert \cdot \rVert$	(euklidische-) Norm auf einem reellen Vektorraum 12
A^{-1}	Inverse Matrix zu der Matrix A
A^T	Transponierte Matrix zu der Matrix A
$\boldsymbol{a}$	Vektor (in der Regel ein Spaltenvektor) 10
$\boldsymbol{a}^\mathsf{T}$	Transponierter Vektor (in der Regel ein Zeilenvektor) 10
$\vec{\boldsymbol{a}}$	Vektor (in der Regel ein Zeilenvektor) 10
$\vec{\boldsymbol{a}}^\mathsf{T}$	Transponierter Vektor (in der Regel ein Spaltenvektor) 10
$a_{i,j}$	Eintrag in der i-ten Zeile und j-ten Spalte einer Matrix $A = [a_{i,j}]$
B	Basismatrix (Spaltenvektoren bilden Basis) eines Gitter 14
B_ℓ	Segment einer Basismatrix B 55
B_ℓ^+	 68
$B_{\ell+1}^-$	 68
$B^{(n,r)}$	..... 96, 98
$B^{[n,\ell]}$	..... 96, 98
$\boldsymbol{b}_i$	i-ter Basisvektor (als Spaltenvektor)
$\vec{\boldsymbol{b}}_i$	i-ter Basisvektor (als Zeilenvektor)
$\widehat{\boldsymbol{b}}_i$	Orthogonalvektor 14, 18
$\boldsymbol{b}^*$	 23

$[\boldsymbol{b}_1 \ \cdots \ \boldsymbol{b}_n]$	Matrix bestehend aus den Spaltenvektoren $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$ 14
$\mathsf{bitlength}(x)$	Anzahl der Bits um die (ganze) Zahl x im Binärsystem zu repräsentieren: $\mathsf{bitlength}(x) = \lfloor \log_2(x) \rfloor + 1$
$\mathsf{d}(\cdot\,,\cdot)$	Abstand von zwei Vektoren 12
$\mathsf{det}\, A$	Determinante der quadratischen Matrix A
$\mathsf{det}\, L$	(Gitter-)Determinante des Gitters L 15
$D(\cdot)$	Lokale Gram'sche Determinante 57
E_n	$n \times n$-Einheitsmatrix
Φ	Isometrie 20
L	Gitter 14
$\mathcal{L}(B)$	von den Spaltenvektoren der Basismatrix B erzeugtes Gitter 14
$\lambda_i(L)$	i-tes sukzessives Minimum des Gitters L 20
$\log_\beta$	Logarithmus zur Basis β
$\ln$	(Natürlicher) Logarithmus zur Basis $e := \lim_{n\to\infty}(1+\frac{1}{n})^n \approx 2.71828\ldots$
$\mathcal{M}$	Menge der vom Rechner darstellbaren Maschinenzahlen 41
$\max(\cdot)$	Maximum-Funktion
$\min(\cdot)$	Minimum-Funktion
$\mu_{i,j}$	Gram-Schmidt-Koeffizienten 18
$\mathbb{N}$	Menge der natürlichen Zahlen (inklusive 0)
$O\,(\cdot)$	$O\,(g(n)) := \{f(n) \mid \exists\, c \in \mathbb{R}_+, \exists\, n_0 \in \mathbb{N} : 0 \le f(n) \le cg(n)\ \forall\, n \ge n_0\}$
$o\,(\cdot)$	$o\,(g(n)) := \{f(n) \mid \forall\, c \in \mathbb{R}_+, \exists\, n_0 \in \mathbb{N} : 0 \le f(n) < cg(n)\ \forall\, n \ge n_0\}$
$\Theta(\cdot)$	$\Theta(g(n)) := \{f(n) \mid \exists\, c_1, c_2 \in \mathbb{R}_+, \exists\, n_0 \in \mathbb{N} : 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\, n \ge n_0\}$
$P(\cdot)$	Parallelepiped 13
$\mathcal{P}(\cdot)$	Grundmasche eines Gitters 16
π_i	Orthogonale Projektion 18
$\mathbb{Q}$	Menge der rationalen Zahlen
R	R ist eine zur Basismatrix B isometrische Matrix 20
R_ℓ	Ein Segment der R Matrix bei der primal/dualen Segmentreduktion .. 62
R^{loc}	 67
R_ℓ^+	 68
$R_{\ell+1}^-$	 68
$R_{\ell,\ell+1}$	Ein Segment der R Matrix bei der k-Segment LLL-Reduktion 56
$\mathbb{R}$	Menge der reellen Zahlen

$\mathbb{R}_{\texttt{GZ-DT}}$	Menge der reellen Zahlen die durch einen Gleitpunktzahlen-Datentyp im Rechner darstellbar ist 76
$\mathbb{R}^d$	d-dimensionaler reeller Vektorraum über $\mathbb{R}$
$\mathbb{R}_+$	Menge der positiven reellen Zahlen (inklusive Null)
$\lfloor r \rfloor$	nächste ganze Zahl $\leq r$ ($r \in \mathbb{R}$)
$\lceil r \rceil$	nächste ganze Zahl $\geq r$ ($r \in \mathbb{R}$)
$\lceil r \rfloor$	$\lceil r - \frac{1}{2} \rceil$ (zu $r \in \mathbb{R}$ nächste ganze Zahl)
rd	Rundungsfunktion 39
$\mathsf{SL}_n(\mathbb{Z})$	Gruppe der $n \times n$ Matrizen A über $\mathbb{Z}$ mit $\lvert\mathsf{det}\, A\rvert = 1$ 10, 15
S_ℓ	Duales Segment 65
S_ℓ^+	 68
$\mathcal{S}^{m\times n}$	Menge aller $m \times n$ Matrizen mit Einträgen aus der Menge $\mathcal{S}$ 10
$\mathsf{span}\ A$	Spann (Erzeugnis) von A 11
$\mathsf{sign}(x)$	Signum-Funktion. Vorzeichen von $x \in \mathbb{R}$, $\mathsf{sign}(x) \in \{\pm 1, 0\}$ 38
$\sigma_{s,k}\left(R^{s,k}\right)$	Darstellung in lokalen Koordinaten 35
vol_n	n-dimensionales Volumen 14
$\omega_{\cdot,\cdot}$	Winkel zwischen zwei Vektoren 12
$\mathbb{Z}$	Menge der ganzen Zahlen

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[Adleman 1983] Adleman, Leonard M.: On Breaking Generalized Knapsack Public Key Cryptosystems. In: *STOC '83, Proceedings of* 15^{th} *Annual ACM Symposium on Theory of Computing*, ACM Press, 1983, S. 402–412

[Ajtai 1996] Ajtai, Miklós: Generating Hard Instances of Lattice Problems. In: *STOC '96, Proceedings of* 28^{th} *Annual ACM Symposium on Theory of Computing*, ACM Press, 1996, S. 99–108. – Erhältlich über http://www.eccc.uni-trier.de/eccc (The Electronic Colloquium on Computational Complexity) als TR96-007

[Ajtai 1998] Ajtai, Miklós: The Shortest Vector Problem in L_2 is NP-hard for Randomized Reductions. In: *STOC '98, Proceedings of* 30^{th} *Annual ACM Symposium on Theory of Computing*, ACM Press, 1998, S. 10–19. – Erhältlich über http://www.eccc.uni-trier.de/eccc (The Electronic Colloquium on Computational Complexity) als TR97-047

[Ajtai und Dwork 1997] Ajtai, Miklós ; Dwork, Cynthia: A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence. In: *STOC '97, Proceedings of* 29^{th} *Annual ACM Symposium on Theory of Computing*, ACM Press, 1997, S. 284–293. – Erhältlich über http://www.eccc.uni-trier.de/eccc (The Electronic Colloquium on Computational Complexity) als TR96-065

[Babai 1986] Babai, László: On Lovász Lattice Reduction and the Nearest Lattice Point Problem. In: *Combinatorica* 6 (1986), S. 1–13

[Backes 1998] Backes, Werner: *Berechnung kürzester Gittervektoren*, Universität des Saarlandes, Fachbereich Informatik, Diplomarbeit, 1998

[Backes und Wetzel 2000] Backes, Werner ; Wetzel, Susanne: New Results on Lattice Basis Reduction in Practice. In: Bosma, Wieb (Hrsg.): *Algorithmic Number Theory – Proceedings of ANTS-IV (LNCS)* Bd. 1838. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Jul 2000, S. 135–152

[Bleichenbacher und Nguyen 2000] Bleichenbacher, Daniel ; Nguyen, Phong Q.: Noisy Polynomial Interpolation and Noisy Chinese Remaindering. In: Preneel, B. (Hrsg.): *Advances in Cryptology – Proceedings of Eurocrypt 2000 (LNCS)* Bd. 1807. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 2000, S. 53–69

[Blömer und Seifert 1999] Blömer, Johannes ; Seifert, Jean-Pierre: On the Complexity of Computing Short Linearly Independent Vectors and Short Bases in a Lattice. In: *STOC '99, Proceedings of* 31^{st} *ACM Symposium on Theory of Computing*, ACM Press, 1999, S. 711–720

[Boneh 2000] Boneh, Dan: Finding Smooth Integers in Short Intervals Using CRT Decoding. In: *STOC '00, Proceedings of* 32^{th} *Annual ACM Symposium on Theory of Computing*, ACM Press, 2000, S. 265–272

[Boneh und Durfee 1999] Boneh, Dan ; Durfee, Glenn: Cryptanalysis of RSA with Private Key d Less than $n^{0.292}$. In: Stern, J. (Hrsg.): *Advances in Cryptology – Proceedings of Eurocrypt 1999 (LNCS)* Bd. 1592. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1999, S. 1–11

[Boneh u. a. 1999] Boneh, Dan ; Durfee, Glenn ; Howgrave-Graham, Nicholas A.: Factoring $N = p^r q$ for Large r. In: Wiener, M. (Hrsg.): *Advances in Cryptology – Proceedings of Crypto 1999 (LNCS)* Bd. 1666. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1999, S. 326–337

[Cai 1999] Cai, Jin-Yi: Some Recent Progress on the Complexity of Lattice Problems. In: *Proceedings of FCRC*, 1999, S. ?–?. – Erhältlich über http://www.eccc.uni-trier.de/eccc (The Electronic Colloquium on Computational Complexity) als TR99-006

[Cai 2000] Cai, Jin-Yi: The Complexity of some Lattice Problems. In: Bosma, Wieb (Hrsg.): *Algorithmic Number Theory – Proceedings of ANTS-IV (LNCS)* Bd. 1838. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 2000, S. 1–32

[Cohen 1996] Cohen, Henri ; Ewing, J. H. (Hrsg.) ; Gering, F. W. (Hrsg.) ; Halmos, P. R. (Hrsg.): *Graduate Texts in Mathematics*. Bd. 138: *A Course in Computational Algebraic Number Theory*. 3. Auflage. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1996

[Conway und Sloane 1998] Conway, John Horton ; Sloane, Neil J. A.: *Sphere Packings, Lattices and Groups*. 3. Auflage. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1998

[Coppersmith 1997] Coppersmith, Don: Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. In: *Journal of Cryptology* 10 (1997), Nr. 4, S. 233–260

[Coppersmith 2001] Coppersmith, Don: Finding Small Solutions to Small Degree Polynomials. 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), S. ?–?

[Deuflhard und Hohmann 1991] Deuflhard, Peter ; Hohmann, Andreas: *Numerische Mathematik: eine algorithmisch orientierte Einführung*. 1. Auflage. Berlin, New York : Walter de Gruyter, 1991

[Domich u. a. 1987] Domich, P. D. ; Kannan, R. ; Trotter, L. E.: Hermite Normal Form Computation Using Modulo Determinant Arithmetic. In: *Mathematics of Operation Research* 12 (1987), Feb, Nr. 1, S. 50–59

[Fincke und Pohst 1985] Fincke, U. ; Pohst, M.: Improved Methods for Calculating Vactors of Short Length in a Lattice, Including a Complexity Analysis. In: *Mathematics of Computation* 44 (1985), Nr. 170, S. 463–471

[Fischlin 1999] Fischlin, Roger: *Block-L^3-Reduktion*. Oct 1999. – Unterlagen zu den Vorträgen am 22. und 29. Oktober 1999 im Rahmen des Gitterpraktikums von Prof. Dr. Schnorr, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, WS 1999/2000

[Fischlin 2001] Fischlin, Roger: *Gitter und Gitterreduktion – Eine Zusammenfassung*. Sep 2001. – Persönliche Zusammenfassung wissenswerter Resultate aus der Gittertheorie

[Fischlin und Seifert 1999] Fischlin, Roger ; Seifert, Jean-Pierre: Tensor-Based Trapdoors for CVP and Their Application to Public Key Cryptography. In: Walker, M. (Hrsg.): *Cryptography and Coding (LNCS)* Bd. 1746. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1999, S. 244–257. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Free Software Foundation 2001] Free Software Foundation: *The GNU C Library Reference Manual, for Version 2.2.x of the GNU C Library.* 0.10. Free Software Foundation, Inc., 59 Temple Place – Suite 330, Boston, MA 02111-1307, USA: , Jul 2001. – http://www.gnu.org/manual/manual.html

[Gauß 1801] Gauss, Carl Friedrich: *Disquisitiones Arithmeticæ.* Leipzig : Gerhard Fleischer, 1801. – Deutsche Übersetzung: „Untersuchung über höhere Arithmetik“, Berlin, Heidelberg: Springer, 1889

[Goldberg 1991] Goldberg, David: What Every Computer Scientist Should Know About Floating-Point Arithmetic. In: *ACM Computing Surveys* Bd. 23. New York, NY : ACM Press, 1991, S. 5–48

[Goldreich u. a. 1997] Goldreich, Oded ; Goldwasser, Shafi ; Halevi, Shai: Public-Key Cryptosystems from Lattice Reduction Problems. In: Kaliski, B. S. (Hrsg.): *Advances of Cryptology – Proceedings of Crypto 1997 (LNCS)* Bd. 1294. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1997, S. 112–131. – Erhältlich über http://www.eccc.uni-trier.de/eccc (The Electronic Colloquium on Computational Complexity) als TR96-056

[Goloub und Loan 1989] Goloub, G.H. ; Loan, C.F.: *Matrix Computations.* London : The Johns Hopkins University Press, 1989

[Gruber und Lekkerkerker 1987] Gruber, M. ; Lekkerkerker, C. G.: *Geometry of Numbers.* 2. Auflage. Amsterdam : North-Holland, 1987

[Hermite 1850] Hermite, Charles: Extraits de lettres de M. Ch. Hermite á M. Jacobi sur différents objets de la théorie des nombres, deuxiéme lettre. In: *Journal für die Reine und Agewandte Mathematik* 40 (1850), S. 279–290

[Hörner 1994] Hörner, Horst H.: *Verbesserte Gitterbasenreduktion; getestet am Chor-Rivest Kryptosystem und an allgemeinen Rucksack-Problemen*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, Diplomarbeit, Aug 1994. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/mastertheses.html online erhältlich

[Howgrave-Graham 1997] Howgrave-Graham, Nicholas A.: Finding Small Solutions of Univariate Modular Equations Revisited. In: *Cryptography and Coding (LNCS)* Bd. 1355. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1997, S. 131–142

[Howgrave-Graham 1998] Howgrave-Graham, Nicholas A.: *Computational Mathematics Inspired by RSA*, University of Bath, Dissertation, 1998

[Howgrave-Graham 2001] Howgrave-Graham, Nicholas A.: Approximate Integer Common Divisors. In: Silverman, Joseph H. (Hrsg.): *Cryptography and Lattices Conference 2001 (LNCS).* Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Mar 2001, S. ?–?

[Jänich 1993] Jänich, Klaus: *Lineare Algebra.* 5. Auflage. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1993

[Joux und Stern 1998] Joux, Antoine ; Stern, Jacques: Lattice Reduction: A Toolbox for the Cryptanalyst. In: *Journal of Cryptology* 11 (1998), S. 161–185

[Kaib 1994] Kaib, Michael: *Gitterbasenreduktion für beliebige Normen*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, Dissertation, 1994. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/phdtheses.html online erhältlich

[Koecher 1992] Koecher, Max: *Lineare Algebra und analytische Geometrie*. 3. Auflage. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1992

[Korkine und Zolotareff 1872] Korkine, A. ; Zolotareff, G.: Sur les formes quadratiques positives ternaires. In: *Mathematische Annalen* 5 (1872), S. 581–583

[Korkine und Zolotareff 1873] Korkine, A. ; Zolotareff, G.: Sur les formes quadratiques. In: *Mathematische Annalen* 6 (1873), S. 366–389

[Korkine und Zolotareff 1877] Korkine, A. ; Zolotareff, G.: Sur les formes quadratiques positives. In: *Mathematische Annalen* 11 (1877), S. 242–292

[Koy 1999] 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/mastertheses.html online erhältlich

[Koy 2002] Koy, Henrik: Gitterbasenreduktion in Primalen/Dualen Segmenten / J.W. Goethe-Universität Frankfurt am Main, Institut für Informatik. Jan 2002. – Manuskript

[Koy und Schnorr 2001a] Koy, Henrik ; Schnorr, Claus Peter: Segment LLL-Reduction of Lattice Bases. 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), S. ?–?. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Koy und Schnorr 2001b] 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), S. ?–?. – Die Arbeit ist in Anhang C auf Seite 115 abgedruckt

[Lagarias u. a. 1990] Lagarias, J.C. ; Lenstra, Hendrik W. ; Schnorr, Claus Peter: Korkin-Zolotarev Bases and successive Minima of a Lattice and its reciprocal lattice. In: *Combinatorica* 10 (1990), Nr. 4, S. 333–348

[Lagrange 1773] Lagrange, Joseph Louis: Recherches d'arithmétique. In: *Nouveaux Mémoires de l'Académie Royale des Sciences et Belles-Lettres* (1773), S. 265–312

[Lenstra u. a. 1982] Lenstra, Arjen K. ; Lenstra, Hendrik W. ; Lovász, László: Factoring Polynomials with Rational Coefficients. In: *Mathematische Annalen* 261 (1982), S. 513–534

[Lenstra 1983] Lenstra, Hendrik W.: Integer Programming with a Fixed Number of Variables. In: *Mathematics of Operation Research* Bd. 8, Nov 1983, S. 538–548

[Louis 1998] Louis, Dirk: *C/C++-Kompendium*. München : Markt und Technik, Buch- und Software-Verlag, 1998

[Martinet 1996] Martinet, J.: *Les Résaux Parfaits des Espaces Euclidiens.* Paris : Masson, 1996

[May 1999] May, Alexander: *Auf Polynomgleichungen basierende Public-Key-Kryptosysteme*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Informatik, Diplomarbeit, Jun 1999. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/mastertheses.html online erhältlich

[Micciancio 2001] Micciancio, Daniele: Improving Lattice Based Cryptosystems Using the Hermite Normal Form. 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), S. ?–?

[Microsoft Corporation 1998] Microsoft Corporation: *Microsoft Developer Network (MSDN) Library, Visual Studio 6.0 release.* Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA: , 1998. – Microsoft Visual Studio 6.0 Development System. http://www.microsoft.com

[Minkowski 1891] Minkowski, Hermann: Über die positiven quadratischen Formen und über kettenbruchähnliche Algorithmen. In: *Journal für die Reine und Angewandte Mathematik* 107 (1891), S. 278–297

[Minkowski 1896] Minkowski, Hermann: *Geometrie der Zahlen.* 1. Auflage. Leipzig : Teubner-Verlag, 1896

[Nguyen 1999] Nguyen, Phong Q.: Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97. In: Wiener, M. (Hrsg.): *Advances in Cryptology – Proceedings of Crypto 1999 (LNCS)* Bd. 1666. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Aug 1999, S. 288–304

[Nguyen 2001] Nguyen, Phong Q.: The Two Faces of Lattices in Cryptology. In: Joseph H. Silverman (Hrsg.): *Cryptography and Lattices Conference 2001 (LNCS).* Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Mar 2001 (Lecture Notes in Computer Science), S. ?–?

[Nguyen und Stern 1997] Nguyen, Phong Q. ; Stern, Jacques: Merkle-Hellman Revisited: A Cryptanalysis of the Qu-Vanstone Cryptosystem Based on Group Factorizations. In: Kaliski, B. S. (Hrsg.): *Advances in Cryptology – Proceedings of Crypto 1997 (LNCS)* Bd. 1294. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1997, S. 198–212

[Nguyen und Stern 1998] Nguyen, Phong Q. ; Stern, Jacques: Cryptanalysis of the Ajtai-Dwork Cryptosystem. In: Krawczyk, H. (Hrsg.): *Advances in Cryptology – Proceedings of Crypto 1998 (LNCS)* Bd. 1462, 1998, S. 223–242

[Nguyen und Stern 1999] Nguyen, Phong Q. ; Stern, Jacques: The Hardness of the Hidden Subset Sum Problem and its Cryptographic Implications. In: Wiener, M. (Hrsg.): *Advances of Cryptology – Proceedings of Crypto 1999 (LNCS)* Bd. 1666. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Aug 1999, S. 31–46

[Nguyen und Stern 2000] Nguyen, Phong Q. ; Stern, Jacques: Lattice Reduction in Cryptology: An Update. In: Bosma, Wieb (Hrsg.): *Algorithmic Number Theory – Proceedings of ANTS-IV (LNCS)* Bd. 1838. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, Jul 2000, S. 85–112

[Oevel 1996] Oevel, Walter: *Einführung in die Numerische Mathematik*. 1. Auflage. Heidelberg, Berlin, Oxford : Spektrum der Wissenschaft, 1996

[Priest 1992] Priest, Douglas M.: *On Properties of Floating Point Arithmetics : Numerical Stability and Cost of Accurate Computations*. Nov 1992. – Draft of PhD Thesis, Berkeley, 9 Nov 1992

[Ritter 1997] Ritter, Harald: *Aufzählung von kurzen Gittervektoren in allgemeiner Norm*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, Dissertation, 1997. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/phdtheses.html online erhältlich

[Rössner 1996] Rössner, Carsten: *Kettenbruchentwicklung in beliebiger Dimension, Stabilität und Approximation*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, Dissertation, 1996. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/phdtheses.html online erhältlich

[Schnorr 1987] Schnorr, Claus Peter: A Hierarchy of Polynomial Lattice Basis Reduction Algorithms. In: *Theoretical Computer Science* 53 (1987), S. 201–224

[Schnorr 1988] Schnorr, Claus Peter: A more Efficient Algorithm for Lattice Basis Reduction. In: *Journal of Algorithms* 9 (1988), S. 47–62

[Schnorr 1993] 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. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Schnorr 1994] Schnorr, Claus Peter: Block Reduced Lattice Bases and Successive Minima. In: *Combinatorics, Probability and Computing* 3 (1994), S. 507–522. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Schnorr 1996] Schnorr, Claus Peter: Ganzzahlige Optimierung und Gitterreduktion / J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik. Sep 1996. – Vorlesungsskript

[Schnorr 1997] Schnorr, Claus Peter: Gittertheorie und algorithmische Geometrie, Reduktion von Gitterbasen und Polynomidealen / J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik. 1997. – Vorlesungsskript

[Schnorr 1998] Schnorr, Claus Peter: Gittertheorie und algorithmische Geometrie, Reduktion von Gitterbasen und Polynomidealen / J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik. Nov 1998. – Vorlesungsmitschrift. Roger Fischlin's Mitschrift der Vorlesungen „Gittertheorie und algorithmische Geometrie“ und „Reduktion von Gitterbasen und Polynomidealen“ von Prof. Dr. Claus Peter Schnorr (SS1994 bzw. WS1994/95, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik)

[Schnorr 1999] Schnorr, Claus Peter: Lineare Algebra I / J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik. 1999. – Vorlesungsskript

[Schnorr 2001] Schnorr, Claus Peter: New Practical Algorithms For The Approximate Shortest Lattice Vector / J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik. 2001. – Manuskript. Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Schnorr und Euchner 1994] Schnorr, Claus Peter ; Euchner, Martin: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In: *Mathematical Programming Studies* 66A (1994), Nr. 2, S. 181–199. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Schnorr und Hörner 1995] Schnorr, Claus Peter ; Hörner, Horst Helmut: Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In: *Advances in Cryptology – Proceedings of Eurocrypt 1995 (LNCS)* Bd. 921. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1995, S. 1–12. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/papers.html online erhältlich

[Schönhage 1984] Schönhage, A.: A Factorization of Univariate Integer Polynomials by Diophantine Approximation and an Improved Basis Reduction Algorithm. In: *Proceedings* 11^{th} *ICALP '84 (LNCS)* Bd. 172. Berlin, Heidelberg, New York, London, Paris, Tokyo : Springer, 1984, S. 436–447

[Schrijver 1986] Schrijver, A.: *Theory of Linear and Integer Programming.* NewYork, Chichester, Brisbane, Toronto : John Wiley and Sons, 1986 (Wiley-Interscience Series in discrete Mathematics and Optimization)

[Seifert 2000] Seifert, Jean-Pierre: *Komplexität von Gitterproblemen: Nicht-Approximierbarkeit und Grenzen der Nicht-Approximierbarkeit*, J.W. Goethe-Universität Frankfurt am Main, Fachbereich Mathematik, Dissertation, 2000. – Die Arbeit ist über http://www.mi.informatik.uni-frankfurt.de/research/phdtheses.html online erhältlich

[Shoup 2001] Shoup, Victor: *NTL: A Library for doing Number Theory, Version 5.2.* http://www.shoup.net/ntl. 2001. – Eine auf C/C++ basierende Programmbibliothek für zahlentheoretische Algorithmen

[Vallée 1991] Vallée, B.: Gauss' Algorithm Revisited. In: *Journal of Algorithms* 12 (1991), Nr. 4, S. 556–572

Index der Literaturverweise